**Topic wise QA - Android**

**Android Studio Version: Flamingo | 2022.2.1 Patch**

**Android SDK version: API 33**

**Android Target SDK Version: API 33**

**Android Compile SDK version: API 33**

**Kotlin Version: 1.8.20-release**

**Manifest Related QA**

<application  
android:name=".application.BaseClass"  
android:allowBackup="true"  
android:dataExtractionRules="@xml/data\_extraction\_rules"  
android:fullBackupContent="@xml/backup\_rules"  
android:supportsRtl="true"  
android:theme="@style/Theme.KotlinRNDProject"  
android:usesCleartextTraffic="true"  
 >  
  
  
<service  
android:name=".service.MyService"  
android:enabled="true"  
android:exported="true" />  
  
  
  
<activity  
android:name="com.example.project.MainActivity"  
android:label="@string/app\_name"  
android:exported="true">  
<intent-filter>  
 <action android:name="android.intent.action.MAIN" />  
  
 <category android:name="android.intent.category.LAUNCHER" />  
</intent-filter>  
</activity>

1. **What is Intent-Filter**

Intent Filter are the components which decide the behavior of an intent. Intent filters specify the type of intents that an Activity, service or Broadcast receiver can respond to. It declares the functionality of its parent component (i.e. activity, services or broadcast receiver). It declares the capability of any activity or services or a broadcast receiver.

Specifies the types of intents that an activity, service, or broadcast receiver can respond to. An intent filter declares the capabilities of its parent component — what an activity or service can do and what types of broadcasts a receiver can handle.Most of the contents of the filter are described by its [**<action>**](https://developer.android.com/guide/topics/manifest/action-element), [**<category>**](https://developer.android.com/guide/topics/manifest/category-element), and [**<data>**](https://developer.android.com/guide/topics/manifest/data-element) subelements.

<intent-filter android:[icon](https://developer.android.com/guide/topics/manifest/intent-filter-element#icon)="*drawable resource*"

               android:[label](https://developer.android.com/guide/topics/manifest/intent-filter-element#label)="*string resource*"

               android:[priority](https://developer.android.com/guide/topics/manifest/intent-filter-element#priority)="*integer*">

    . . .

</intent-filter>

1. **What Export Tag in Manifest file**

The **android:exported** attribute sets whether a component (activity, service, broadcast receiver, etc.) can be launched by components of other applications: If true , any app can access the activity and launch it by its exact class name

1. **What is Action**

These are the current standard actions that Intent defines for launching activities (usually through Context.startActivity. The most important, and by far most frequently used, are ACTION\_MAIN and ACTION\_EDIT.

* ACTION\_MAIN =>
* ACTION\_VIEW => Use this action in intent with startActivity() when you have some information that activity can show to the user like showing an image in a gallery app or an address to view in a map app
* ACTION\_ATTACH\_DATA
* ACTION\_EDIT
* ACTION\_PICK
* ACTION\_CHOOSER
* ACTION\_GET\_CONTENT
* ACTION\_DIAL
* ACTION\_CALL
* ACTION\_SEND => You should use this in intent with startActivity() when you have some data that the user can share through another app, such as an email app or social sharing app
* ACTION\_SENDTO
* ACTION\_ANSWER
* ACTION\_INSERT
* ACTION\_DELETE
* ACTION\_RUN
* ACTION\_SYNC
* ACTION\_PICK\_ACTIVITY
* ACTION\_SEARCH
* ACTION\_WEB\_SEARCH
* ACTION\_FACTORY\_TEST

1. **What is Lunch-Mode**

Activity Lunch Mode standard

* **standard**
* **singleTop**
* **singleTask**
* **singleInstance**

<activity android:launchMode = [“standard” | “singleTop” | “singleTask” | “singleInstance”] ../>

1. **standard**: This is the default launch mode of activity. If you don’t set any launch mode to your activity, it will use the standard mode by default. It creates a new instance of activity every time even if activity instance is already present.

Suppose we have A, B, C, and D activities and your activity B has standard launch mode. Now again launching activity B

State of Activity Stack before launch B

A →B→C→D

State of Activity Stack after launch B

A →B →C→D→B

We can see that new instance of B is created again

2. **singleTop**: If an instance of activity already exists at the top of the current task, a new instance will not be created and the Android system will route the intent information through onNewIntent().

If an instance is not present on top of the task then a new instance will be created.

Suppose we have A, B, C, and D activities. A →B →C →D and set the C as SingleTop

If we launch C then a new instance of C will be created as it is not on top.

So it will look like A →B →C →D →C

Now suppose we have A →B →C →D →C like this

then we if again launch C activity then in this case new instance will not be created. Instead, we will receive the callback on onNewIntent() method.

3. **singleTask**: If An activity declared with launch mode as singleTask can have only one instance in the system (singleton). At a time only one instance of activity will exist.

If activity instance is not present then the new instance will be created and if the instance is already present in the system then the onNewIntent() method will receive the callback.

Suppose we have A, B, C activities(A →B →C ) and we are launching D that has a singleTask launch mode. In that case, the new instance of D will be created so the current state will look like this. (A →B →C →D)

Now let suppose if we launch B that also have has a singleTask launch mode then current state will look like

A →B

Here old instance gets called and intent data route through onNewIntent() callback. Also, notice that C and D activities get destroyed here.

4. **singleInstance** : It is similar to singleTask except that no other activities will be created in the same task. If another Activity is called from this kind of Activity, a new Task would be automatically created to place that new Activity.

Case 1:

Suppose you have A, B, and C activities(A →B →C) and your activity D has a singleInstance launch mode. In this case, if we launch D then D will be launch in the diffrent task. New task for D will be created.

Task1: A →B →C

Task2 : D (here D will be in the different task)

Now if you continue this and start E and D then Stack will look like

Task1: A →B →C →E

Task2: D

1. **How to add Different launchModes using Java**

**Java:**

Intent openIntent = new Intent(getApplicationContext(), GeeksforGeeksActivity.class);  
openIntent.addFlags(Intent.FLAG\_ACTIVITY\_CLEAR\_TASK | Intent.FLAG\_ACTIVITY\_NEW\_TASK);  
getApplicationContext().startActivity(openIntent);

**Kotlin:**

val openIntent = Intent(*applicationContext*, GeeksforGeeksActivity::class.java)  
openIntent.addFlags(Intent.*FLAG\_ACTIVITY\_CLEAR\_TASK* or Intent.*FLAG\_ACTIVITY\_NEW\_TASK*)  
*applicationContext*.startActivity(openIntent)

1. **AllowBackup**

**android:allowBackup**. Whether to let the application participate in the backup and restore infrastructure. If this attribute is set to "false" , no backup or restore of the application is ever performed, even by a full-system backup that otherwise causes all application data to save using adb

1. **taskAffinity**

Task affinity means **a single application can have more than one stack; You can give any name to stack** (android:taskaffinity=” com.some.task.

An affinity indicates which task an activity "prefers" to belong to. By default, all the activities from the same app have an affinity for each other: they "prefer" to be in the same task.

However, you can modify the default affinity for an activity. Activities defined in different apps can share an affinity, and activities defined in the same app can be assigned different task affinities

1. **allowTaskReparenting**
2. **clearTaskOnLaunch**
3. **alwaysRetainTaskState**
4. **finishOnTaskLaunch**

**Gradle Related QA**

1. **What is Gradle file**

Gradle, an advanced build toolkit, to automate and manage the build process, while allowing you to define flexible custom build configurations.

Gradle is a build automation tool known for its flexibility to build software. A build automation tool is used to automate the creation of applications. **The building process includes compiling, linking, and packaging the code**. The process becomes more consistent with the help of build automation tools.

The Module-level build.gradle supports various build configurations like:

1. **android:** This block is used for configuring the specific android build options.
   * *compileSdkVersion* – This is used to define the API level of the app and the app can use the features of this and lower level.
2. **defaultConfig:**
   * *applicationId*– This is used for identifying unique id for publishing of the app.
   * *minSdkVersion*– This defines the minimum API level required to run the application.
   * *targetSdkVersion*– This defines the API level used to test the app.
   * *versionCode*– This defines the version code of the app. Every time an update needs to be of the app, the version code needs to be increased by 1 or more.
   * *versionName*– This defines the version name for the app. this could be increased by much while creating an update.
3. **buildTypes(release):**
   * *minifyEnabled*– this will enable code shrinking for release build.
   * *proguardFiles*– this will specify the [progaurd](https://www.geeksforgeeks.org/how-to-use-proguard-to-reduce-apk-size-in-android/)settings file.
4. **dependencies:** This specifies the dependencies that are needed to build the project.

Both the top-level and module-level build.gradle files are the main script files for automating the tasks in an android project and are used by Gradle for generating the APK from the source files

1. **What is APK(Android Application Package) and aab(Android App Bundle)**

App bundles are publishing format, whereas APK (Android application Package) is the packaging format which eventually will be installed on device. Google uses app bundle to generate and serve optimized APKs for each user's device configuration, so they download only the code and resources they need to run your app.

The sections above point to some differences between APKs and AABs. The table below summarizes the points covered.

|  |  |  |
| --- | --- | --- |
| **Factor** | **AABs** | **APKs** |
| **Full form** | Android application bundle | Android application package or Android package kit |
| **Submission** | AABs are split into smaller APK files. Each split APK corresponds to a specific device configuration | APK file is submitted and later downloaded in the same form as it is created |
| **Installation** | AABs cannot themselves trigger installation. They defer APK generation to the Google Play | APKs can be used to directly install an app on an Android device |
| **Size** | AAB files ensure smaller app size because each split APK contains device specific resources | APK files ensure larger app size because all resources are downloaded to the device irrespective of their need |
| **Dynamic features** | Dynamic feature resources are stored on Google Play and called upon when needed | All feature-based resources are stored within the APK file |
| **Asset packs** | All assets not needed to install the app are stored on Google Play | All assets are stored in the main APK file. |
| **App churn** | Smaller app size causes lower app churn | Larger app size causes greater app churn |
| **Storage efficiency** | Are more efficient in terms of space | Unnecessary resources take up a lot of device storage and reduce efficiency |

1. **What is signing capability**
2. **What is Build Variant**

Build Type:Build types: In Android apps, build types usually refer to the environment in which you're testing. By default, when you create an app in Android Studio from any of the templates, you get two build types: debug and release.

The buildTypes block is where you can configure multiple build types. By default, the build system defines two build types: debug and release. The debug build type is not explicitly shown in the default build configuration, but it includes debugging tools and is signed with the debug key. The release build type applies Proguard settings and is not signed by default.

**File => Project structure => Build Variants => Select Module(Default App) => Build Type => (Add +, Delete -, Edit)**

debug {

applicationIdSsuffiix “.bebug”

versionNameSuffix “-Debug”

minifyEnabled false

debuggable true

}

qa {

applicationIdSsuffiix “.qa”

versionNameSuffix “-QA”

minifyEnabled true

debuggable false

}

staging {

applicationIdSsuffiix “.staging”

versionNameSuffix “-Staging”

minifyEnabled true

debuggable false

}

release {

minifyEnabled true

debuggable false

proguardFiles getDefaultProguardFile(‘proguard-android- optimize.txt’), ‘proguard-rules.pro’

}

1. **What is Flavor**

**Product Flavours:** Product flavours lets you create multiple variants(Free/paid flavours/Stable/staining/experimental) of an android app while using a single codebase. To create product flavours you need to define rules in the build.

The **productFlavors** block is where you can configure multiple product flavors.This allows you to create different versions of your app that can override the defaultConfig block with their own settings. Product flavors are optional, and the build system does not create them by default.

This example creates a free and paid product flavor. Each product flavor then specifies its own application ID, so that they can exist on the Google Play Store, or an Android device, simultaneously

**File => Project structure => Build Variants => Select Module(Default App) => Flavours => (Add +, Delete -, Edit)**

flavorDimensions 'default'

productFlavors {

redFlavor {

dimension 'default'

applicationId "com.sample.productFlavors"

minSdkVersion 15

targetSdkVersion 28

buildConfigField "String", "HOST\_URL", "\"host url 1\""

}

greenFlavor {

dimension 'default'

applicationId "com.sample.productFlavors"

minSdkVersion 18

targetSdkVersion 28

buildConfigField "String", "HOST\_URL", "\"host url 2\""

}

}

sourceSets {

main {

manifest.srcFile 'AndroidManifest.xml'

java.srcDirs = ['src/commonFiles/java']

resources.srcDirs = ['src/commonFiles/java']

aidl.srcDirs = ['src/commonFiles/java']

renderscript.srcDirs = ['src/commonFiles/java']

res.srcDirs = ['res']

assets.srcDirs = ['assets']

}

redFlavor {

manifest.srcFile 'AndroidManifest-flavor2.xml'

res.srcDirs = ['res-flavor2', 'res']

}

greenFlavor{

manifest.srcFile 'AndroidManifestXXX.xml'

java.srcDirs = ['src\_qa']

}

1. **What is Target SDK version**

targetSdkVersion: Specifies the API level used to test the app.

1. **What is Compile SDK version**

**compileSdkVersion:** compileSdkVersion specifies the Android API level Gradle should use to compile your app. This means your app can use the API features included in this API level and lower.

1. **Minimum SDK Version**

minSdkVersion: Defines the minimum API level required to run the app.

1. **How you can generate multiple apk**
2. **Heap size**
3. **Memory Management**
4. **What is Progard and code srinking**

ProGuard is a code optimization and obfuscation tool used in the Android development ecosystem. It is included in the Android SDK (Software Development Kit) and is typically used during the release build process of an Android application.

By using ProGuard, developers can achieve smaller APK sizes, protect their code from reverse engineering, and potentially optimize the performance of their Android applications.

When an Android application is built in release mode, ProGuard can be enabled to perform several tasks:

**Code shrinking**: ProGuard analyzes the code and removes unused classes, methods, and resources from the application. This helps reduce the size of the APK (Android Application Package) and can result in smaller app downloads and installations. It’s detects and safely removes unused classes, fields, methods, and attributes from your app and its library dependencies

**Resource shrinking:** Removes unused resources from your packaged app, including unused resources in your app’s library dependencies. It works in conjunction with code shrinking such that once unused code has been removed, any resources no longer referenced can be safely removed as well

**Code obfuscation**: ProGuard renames classes, methods, and variables to meaningless, shorter names. This makes the code more difficult to understand and reverse engineer, as the original names are no longer present.

**Optimization**: ProGuard performs certain optimizations on the code, such as removing redundant instructions and replacing them with more efficient alternatives. This can improve the performance of the application. It’s shortens the name of classes and members, which results in reduced DEX file sizes

android {  
 buildTypes {  
 getByName("release") {  
 // Enables code shrinking, obfuscation, and optimization for only  
 // your project's release build type. Make sure to use a build  
 // variant with `isDebuggable=false`.  
 isMinifyEnabled = true

// Enables resource shrinking, which is performed by the  
 // Android Gradle plugin.  
 isShrinkResources = true

// Includes the default ProGuard rules files that are packaged with  
 // the Android Gradle plugin. To learn more, go to the section about  
 // R8 configuration files.  
 proguardFiles(  
 getDefaultProguardFile("proguard-android-optimize.txt"),  
 "proguard-rules.pro"  
 )  
 }  
 }  
}

1. **What is APK Profiler**

An app is considered to have poor performance if it responds slowly, shows choppy animations, freezes, or consumes too much power.

Fixing performance problems involves identifying areas in which your app makes inefficient use of resources such as the CPU, memory, graphics, network, or the device battery.

To find and fix these problems, use the profiling and benchmarking tools and techniques described in this topic.

**CPU profiler:** Helpstrack down runtime performance issues. ;

**Memory profiler:**  Helps track memory allocations. ;

**Energy profiler:**  Tracks energy usage , which can contribute to battery drain

1. **What is App Quality Inspection**
2. **What is App Inspection**

**What is Design Pattern**

**Clean Architecture**

Clean Architecture is **the blueprint for a modular system**, which strictly follows the design principle called separation of concerns.

Generally, clean architecture is **the software design philosophy that can organize the code according to how the business logic can be kept.**

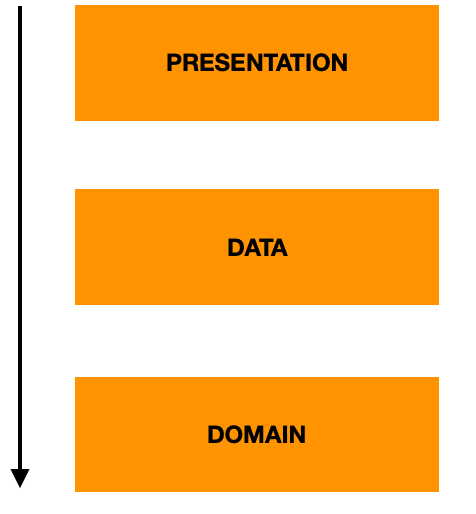
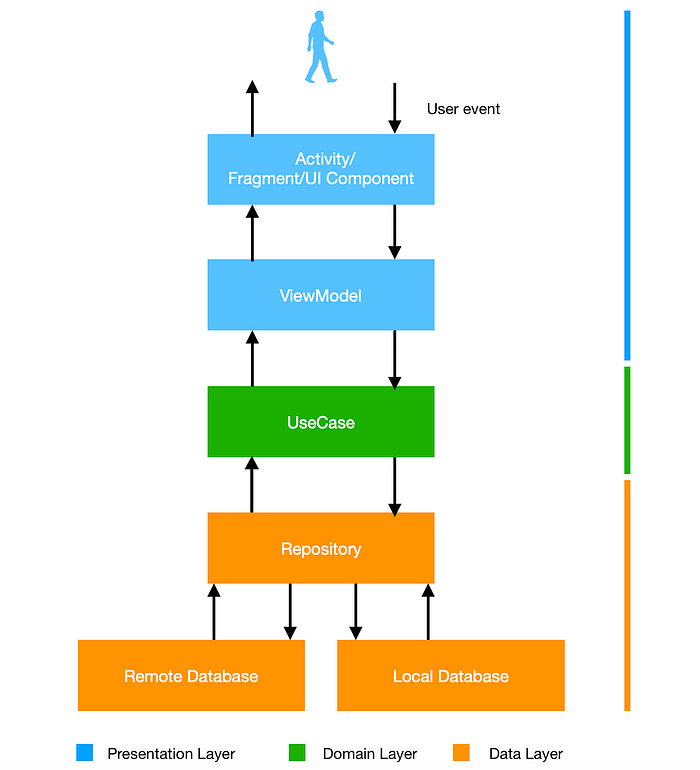
**The way of building reliable, mentainable, scalable and testabale code.**

The Clean Architecture is the most powerful software design which promotes the separation of concerns and the creation of highly maintainable and flexible software systems, and this can be done by organizing the codebase into independent layers that communicate through defined interfaces.

**Domain layer:** Would execute business logic which is independent of any layer and is just a pure kotlin package with no android specific dependency.

**Data layer:** Would dispense the required data for the application to the domain layer by implementing interface exposed by the domain

**Presentation layer:**Would include both domain and data layer and is android specific which executes the UI logic.

**Pros of Clean Architecture**

1. **Separation of Concerns**: Clean Architecture promotes the separation of concerns, which helps to create a more organized and modular codebase. Each layer has a specific responsibility, and the layers are loosely coupled, which makes the code more maintainable and testable.
2. **Testability**: It makes it easier to write automated tests, as the different layers are isolated and can be tested independently. This improves the quality of the software and reduces the risk of introducing bugs.
3. **Flexibility**: Clean Architecture promotes a modular design that allows developers to swap out components or change the implementation details without affecting the rest of the system. This makes it easier to adapt to changing requirements or to introduce new features.
4. **Independence from external frameworks**: It makes it possible to switch out external frameworks or libraries without affecting the rest of the system. This reduces the risk of being locked into a particular technology or vendor.
5. **Maintainability**: It helps to create code that is easier to understand and maintain. By separating concerns and using clear interfaces between components, developers can work on different parts of the system without affecting other parts.

**Cons of Clean Architecture**

1. **Overhead**: It can add some overhead to the development process, as it requires more upfront planning and design. It may also require more code to be written, as each layer needs to have its own set of interfaces and abstractions.
2. **Complexity**: It can make the codebase more complex, especially for smaller projects or simpler applications. It may be overkill for small, straightforward applications.
3. **Learning curve**: It requires developers to understand the principles and concepts behind the architecture. This can be a challenge for developers who are new to the approach.
4. **Time-consuming**: It can take more time to implement than other approaches, especially if the development team is not experienced with the approach.

**Android Architecture**

1. **What is Android Architect Component**
2. **ADB => Android Debug Bridge**

ADB stands for Android Debug Bridge. It is a command-line tool that allows communication between a computer and an Android device. ADB is primarily used by developers and advanced users for various tasks related to debugging, installing, and managing applications on Android devices.

To use ADB, you need to have the Android SDK (Software Development Kit) installed on your computer, which includes the necessary ADB binaries. Additionally, you must enable USB debugging on your Android device to establish a connection between the device and the computer via ADB.

**Here are some common uses of ADB:**

**Debugging:** ADB enables developers to debug their Android applications by connecting their device to a computer and using ADB commands to monitor and analyze the device's behavior, logcat outputs, and error messages.

**Application Installation**: ADB allows users to install applications on their Android devices directly from a computer, bypassing the need to use the Google Play Store. This can be useful when installing applications that are not available through official channels or for testing purposes.

**File Transfer**: ADB provides a way to transfer files between a computer and an Android device using the command-line interface. This is particularly useful when the device's storage is inaccessible or when large files need to be transferred quickly.

**Shell Access**: ADB provides a shell interface that allows users to execute commands directly on an Android device. This can be used to access advanced features, perform system-level tasks, or modify settings that are not accessible through the device's user interface.

1. **DDMS**

In Android development, DDMS stands for "**Dalvik Debug Monitor Server**." It is a tool that provides debugging capabilities for Android applications running on devices or emulators. DDMS is part of the **Android Debug Bridge** (ADB), a command-line tool that allows communication between a development machine and an Android device.

DDMS is a powerful debugging and monitoring tool for Android developers, enabling them to diagnose issues, gather information, and optimize their applications for better performance.

Android studio includes a debugging tool called the Dalvik Debug Monitor Service (DDMS). DDMS provides services like screen capture on the device, threading, heap information on the device, logcat, processes, incoming calls, SMS checking, location, data spoofing, and many other things related to testing your Android application.

DDMS connects the IDE to the applications running on the device. On Android, every application runs in its own process, each of which hosts its own virtual machine (VM). And each process listens for a debugger on a different port.

When it starts, DDMS connects to **ADB**(**Android Debug Bridge** which is a command-line utility included with Google’s **Android**SDK.). An Android Debugger is used for debugging the Android app and starts a device monitoring service between the two. This will notify DDMS when a device is connected or disconnected. When a device is connected, a VM monitoring service is created between ADB and DDMS, which will notify DDMS when a VM on the device is started or terminated.

**DDMS offers several features for developers to monitor and debug their applications:**

**Logcat**: DDMS displays the log messages generated by the Android system, including debugging messages, errors, and application-specific log messages. It helps developers track the flow of their application and identify any issues or errors.

**File Explorer**: Developers can browse and manage the files on an Android device or emulator through DDMS. This feature allows them to push files to the device, pull files from it, or delete files.

**Emulator Control**: When using an Android emulator, DDMS allows developers to simulate various events and actions on the emulator, such as incoming phone calls, SMS messages, GPS coordinates, and network status changes. This feature helps in testing different scenarios and behaviors of an application.

**Heap Analysis**: DDMS provides tools for analyzing the memory usage of an application. Developers can take heap dumps, track object allocations, and analyze memory usage to detect memory leaks and optimize the performance of their applications.

**Network Statistics**: DDMS allows developers to monitor network traffic generated by their applications. It provides insights into the network usage, including data sent and received, network latency, and connection status

1. **What is DVM => Delvik Virtual machine**

Dalvik\* Virtual Machine’s heap size for application processes is limited. Applications start up with 2 MB, and the maximum allocation, marked as “largeHeap,” is limited to 36 MB (depending on the specific device configuration). Examples of large heap applications are Photo/Video Editor, Camera, Gallery, and Home Screen.

In the context of Android, DVM stands for Dalvik Virtual Machine. However, it's important to note that as of Android 5.0 (Lollipop) release, the Dalvik Virtual Machine has been replaced by the Android Runtime (ART) as the primary runtime environment for Android applications.

Dalvik Virtual Machine (DVM) was the original virtual machine used by Android prior to version 5.0. It was specifically designed for running applications on Android devices with limited resources such as memory and processing power. DVM used a register-based architecture and executed bytecode compiled from Java source code.

With the introduction of ART in Android 5.0, the runtime environment was changed to use ahead-of-time (AOT) compilation rather than just-in-time (JIT) compilation. This means that instead of compiling bytecode at runtime, ART compiles the bytecode to machine code during the installation or upgrade of an application. This results in improved performance and efficiency compared to DVM.

So, while DVM was the original virtual machine used in Android, it has been replaced by ART since Android 5.0

**ART => Android Run Time**

In the context of Android development, ART stands for Android Runtime. ART is the managed runtime environment used by the Android operating system to execute and run applications written in the Java programming language or languages that compile to bytecode, such as Kotlin.

ART was introduced as the default runtime in Android 5.0 (Lollipop) to replace the older Dalvik runtime. One of the main differences between Dalvik and ART is that Dalvik used Just-In-Time (JIT) compilation, while ART uses Ahead-of-Time (AOT) compilation.

With ART, the bytecode of an Android app is transformed into a more efficient representation called machine code during the installation or upgrade process. This approach allows the application to run faster and reduces the runtime overhead compared to the on-the-fly bytecode compilation used in Dalvik. It also brings other performance improvements such as reduced battery consumption and improved garbage collection.

In addition to performance enhancements, ART also introduced new features and optimizations, such as improved application debugging, support for 64-bit architectures, enhanced security features, and compatibility with new Android features like multi-window mode and multi-user support.

**Overall**, ART plays a crucial role in the execution of Android applications, providing improved performance and a more efficient runtime environment for developers and users alike.

1. **What is Dex => Dex Compiler**

In the context of Android development, DEX stands for Dalvik Executable. It is a file format used by the Dalvik virtual machine, which is the runtime environment in Android that executes applications.

When you build an Android application, the Java source code is compiled into bytecode, which is a low-level representation of the code. However, the Android operating system doesn't directly execute bytecode. Instead, it uses the Dalvik virtual machine, which requires a special format called DEX.

The DEX file format is optimized for resource-constrained devices like smartphones and tablets. It includes features like bytecode instruction packing, which reduces the overall size of the file and improves runtime performance.

During the build process, the Android build tools convert the bytecode into DEX format, which can then be executed by the Dalvik virtual machine or the newer Android Runtime (ART), introduced in Android 5.0 (Lollipop). The DEX file contains the compiled bytecode, along with additional metadata and resources required by the application.

**So, in summary**, DEX is a file format used by the Dalvik virtual machine or ART in Android to execute compiled Java bytecode

1. **How apk or aab is generating when we generate the build**

When generating a build for an Android application, the APK (Android Package) file is created through a series of steps that involve the Android build tools and the Android Debug Bridge (ADB). Here's a general overview of the process:

Compilation: The source code of the Android application, written in Java or Kotlin, is compiled into Dalvik bytecode (for older versions of Android) or ART bytecode (for newer versions). This step is typically performed by the Java compiler (javac) or the Kotlin compiler (kotlinc).

Resource Packaging: The Android resource files, such as XML layouts, images, and other assets, are compiled and packaged into a binary format known as the Android resource file (.arsc). This step is performed by the Android Asset Packaging Tool (AAPT).

Manifest Merging: The Android manifest file (AndroidManifest.xml), which contains essential information about the application, is merged with other manifests from libraries or dependencies used in the project. This process ensures that the final manifest accurately represents the application's configuration.

Build Variant Configuration: Depending on the build variant (e.g., debug or release) and the project settings, different configurations and optimizations may be applied to the build. These configurations can include enabling/disabling features, setting different signing keys, or applying code minification.

Compilation to DEX: The compiled bytecode from the first step is transformed into Dalvik Executable (DEX) format, which is the format understood by the Android Runtime. This transformation is performed by the dx tool, which is part of the Android build tools.

Packaging: The compiled DEX files, along with the resource files, libraries, and assets, are packaged together into an APK file. This step is carried out by the aapt tool, which creates the APK and signs it with a debug key by default.

Signing: In the case of a release build, the APK needs to be signed with a release key to ensure its authenticity. This step involves generating a private key and using it to sign the APK. The Android build tools provide a keytool utility for key generation and the apksigner tool for signing the APK.

Once the build process is complete, you will have an APK file that can be installed on Android devices. The ADB (Android Debug Bridge) is a command-line tool that allows you to interact with Android devices and perform various actions, including installing and uninstalling APKs. You can use ADB to install the generated APK on a connected device or emulator for testing or distribution purposes

1. **How is apk execute at client side(user device)=> Android Asset Packaging**

APK (Android Application Package) files contain the compiled code and resources of an Android app. When an APK is installed on an Android device and the app is launched, the following steps typically occur for executing the APK code at runtime:

**Package Manager**: The Package Manager on the Android device receives the APK file and performs initial checks, such as verifying the digital signature to ensure the APK hasn't been tampered with and checking for compatibility with the device's hardware and software.

**Installation**: If the APK passes the initial checks, the Package Manager proceeds with the installation process. During installation, the APK file is unpacked and its components, such as code files (Dalvik bytecode or ART bytecode) and resources (images, layouts, etc.), are extracted to the device's storage.

**Class Loading**: Once the installation is complete, the Android runtime system (Dalvik or ART) takes over. The runtime system is responsible for executing the APK code. It starts by loading the classes needed by the app into memory. The class loading process involves locating and reading the bytecode of the classes from the APK file.

**JIT Compilation (Dalvik):** In the case of the Dalvik runtime (used in older Android versions), the bytecode is converted to native machine code at runtime by the Just-In-Time (JIT) compiler. This allows for faster execution of the code, as the native machine code is directly executed by the device's processor.

**Ahead-of-Time Compilation (ART)**: In newer Android versions (starting from Android 5.0), the ART runtime is used. With ART, the bytecode is ahead-of-time (AOT) compiled to native machine code during the app installation process. This compilation step improves app performance during runtime since the machine code is already available and doesn't require JIT compilation.

**App Execution**: Once the necessary classes are loaded and the code is compiled (either JIT or AOT), the app's entry point (usually the main() method) is invoked. The execution then follows the flow of the code, executing various functions, methods, and callbacks defined in the APK.

**Resource Access:** During runtime, the app may need to access various resources, such as images, layouts, or localization files. The Android framework provides APIs to access these resources, and the app can retrieve them from the APK's resources directory as needed.

**Interaction with the Android System**: Apps often interact with the Android system through various APIs, such as accessing device sensors, making network requests, or launching other activities. The app's code calls these APIs, and the runtime system handles the communication between the app and the underlying system components.

**Overall**, the runtime system manages the execution of the APK code, ensuring that the app functions as intended and interacts with the device's hardware and software in a controlled manner.

1. **What is JVM, JRE and JDK => Java Virtual Machine , Java Run Time, Java Development Kit**



JDK stands for Java Development Kit. It is a software development environment used for developing Java applications and applets. The JDK includes a set of tools, libraries, and utilities that enable developers to write, compile, debug, and run Java programs.

In summary, the JDK is a comprehensive package that facilitates Java software development by providing the necessary tools, libraries, and runtime environment. It is a fundamental requirement for developing Java applications across different platforms.

**Here are some key components of the JDK:**

**Java Compiler (javac):** It is used to compile Java source code into bytecode, which can be executed by the Java Virtual Machine (JVM).

**Java Virtual Machine (JVM):** It is an essential part of the JDK and is responsible for executing Java bytecode. It provides a runtime environment for Java applications to run on different platforms.

**Java Runtime Environment (JRE):** The JDK includes the JRE, which is necessary for running Java applications on end-user systems. It contains the JVM and other libraries required to execute Java programs.

**Java Development Tools:** The JDK provides a set of development tools, such as the Java debugger (jdb), Java documentation generator (javadoc), and Java archive tool (jar), among others. These tools assist developers in writing, testing, and packaging Java applications.

**Java API Libraries:** The JDK includes a vast collection of class libraries, known as the Java API (Application Programming Interface). The API provides pre-built classes and methods for various functionalities like file handling, networking, database connectivity, user interface development, and more. Developers can leverage these libraries to accelerate the development process.

1. **JRE => Java Run Time**

JRE stands for Java Runtime Environment. It is a software package developed by Oracle Corporation that provides the necessary resources for executing Java applications. The JRE consists of a Java Virtual Machine (JVM), class libraries, and other supporting files required to run Java programs.

When you write and compile a Java program, it gets converted into bytecode, which is a platform-independent representation of the program. The JRE is responsible for interpreting and executing this bytecode on a specific operating system.

The JRE includes the JVM, which is the component that actually executes the Java bytecode. It provides an abstraction layer between the Java program and the underlying hardware and operating system. The JVM is responsible for various tasks, such as memory management, garbage collection, and dynamic bytecode interpretation.

In addition to the JVM, the JRE also contains a set of class libraries that provide pre-written code for common tasks, such as input/output operations, networking, and graphical user interfaces. These libraries are essential for developing Java applications and simplify the programming process.

Overall, the JRE is required to run Java applications on a computer system, as it provides the necessary runtime environment and resources for executing Java code.

1. **JVM => Java virtual machine**

JVM stands for Java Virtual Machine. It is a crucial component of the Java programming language and runtime environment. The JVM serves as an abstract execution environment for running Java bytecode, which is a compiled format of Java source code.

The primary purpose of the JVM is to provide platform independence to Java programs. It achieves this by executing the bytecode, which is produced by the Java compiler, on any operating system or architecture that has a compatible JVM implementation. This allows developers to write code once and run it anywhere, as long as a JVM is available for the target platform.

The JVM is responsible for several key tasks, including:

Loading and Verifying: The JVM loads the bytecode, verifies its integrity, and ensures that it adheres to the rules and specifications of the Java language.

Memory Management: The JVM manages the memory allocation and deallocation for Java objects. It includes features like automatic garbage collection, which automatically reclaims memory occupied by objects that are no longer in use.

Just-In-Time (JIT) Compilation: The JVM includes a Just-In-Time compiler that translates frequently executed bytecode into native machine code, optimizing the performance of the Java application.

Security: The JVM provides a secure environment for executing Java code, enforcing various security restrictions and preventing malicious actions.

Runtime Monitoring and Profiling: The JVM offers tools for monitoring and profiling Java applications, allowing developers to analyze their performance, identify bottlenecks, and optimize the code.

It's worth noting that while the JVM is primarily associated with running Java programs, it also supports other programming languages, such as Scala, Kotlin, and Groovy, that can be compiled into Java bytecode and executed on the JVM

1. **Android ADV = Android Debugging Bridge**

Android Debug Bridge (adb) is a versatile command-line tool that lets you communicate with a device. The adb command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device. It is a client-server program that includes three components:

* A client, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing an adb command.
* A daemon (adbd), which runs commands on a device. The daemon runs as a background process on each device.
* A server, which manages communication between the client and the daemon. The server runs as a background process on your development machine.

adb is included in the Android SDK Platform-Tools package. You can download this package with the [SDK Manager](https://developer.android.com/studio/intro/update#sdk-manager), which installs it at *android\_sdk*/platform-tools/.

**How adb works**

When you start an adb client, the client first checks whether there is an adb server process already running. If there isn't, it starts the server process. When the server starts, it binds to local TCP port 5037 and listens for commands sent from adb clients—all adb clients use port 5037 to communicate with the adb server.

The server then sets up connections to all running devices. It locates emulators by scanning odd-numbered ports in the range 5555 to 5585, the range used by the first 16 emulators

**Some ADB Command:**

1. adb devices =>show list of device

**2. adb install** *path\_to\_apk: Install the App from apk path*

*3.* **adb shell pm uninstall com.xxx.xx** => To uninstall the app from device

**Type of Exception in Android**

In Android, exceptions are used to handle errors and abnormal situations that may occur during the execution of an application. Some commonly used exceptions in Android include:

**NullPointerException (NPE)**: This exception is thrown when a program attempts to access or use a reference variable that is null (not pointing to any object).

**ClassCastException:** This exception is thrown when an object is cast to an incompatible class.

**IllegalStateException:** This exception is thrown when a method is called at an inappropriate time or in an incorrect state.

**IllegalArgumentException:** This exception is thrown when a method is passed an illegal argument.

**IndexOutOfBoundsException:** This exception is thrown when an index is either negative or greater than the size of the collection or array.

**SecurityException:** This exception is thrown when a security violation occurs.

**NetworkOnMainThreadException:** This exception is thrown when network operations are performed on the main thread, which is not allowed in Android.

**OutOfMemoryError:** This error is thrown when an application attempts to allocate more memory than is available

**Synchronous (Sync) & Asynchronous(aync)**

**Synchronous (Sync**)

In synchronous programming, tasks are executed one after another in a sequential manner. When a synchronous task is initiated, the program execution waits for that task to complete before moving on to the next task. In other words, synchronous operations block the execution of the program until the task is finished.

print("Task 1 started.")  
task1()  
print("Task 1 completed.")  
  
print("Task 2 started.")  
task2()  
print("Task 2 completed."

**Note**: In this example, Task 1 must complete before Task 2 can start. The program waits for each task to finish before proceeding to the next one.

**Asynchronous (Async):**

Asynchronous programming allows tasks to run independently and concurrently without blocking the execution of the program. Instead of waiting for a task to complete, the program continues its execution and can handle other tasks. When an asynchronous task completes, it notifies the program through a callback, promise, or other mechanisms.

print("Task 1 started.")  
await task1()  
print("Task 1 completed.")  
  
print("Task 2 started.")  
await task2()  
print("Task 2 completed.")

**Note**: In this example, the await keyword is used to pause the execution of the program until an asynchronous task completes. However, during the time the async task is running, the program can proceed to execute other tasks or perform other operations.

**Different between sync and async**

The main difference between sync and async lies in how they handle the flow of execution and manage waiting times. Synchronous operations are simpler to understand and reason about but may lead to blocking and slower program execution. Asynchronous operations allow for better utilization of resources, responsiveness, and improved performance, especially when dealing with tasks that involve I/O operations or long-running processes.

It's important to note that the terms "sync" and "async" are often used in the context of concurrent programming, particularly when dealing with I/O operations, multithreading, or event-driven systems. Different programming languages and frameworks may have their own specific implementations and mechanisms for handling sync and async operations.

**Activity**

1. **Activity Parrent class**

Super class => ContextThemeWrapper

Activity extends ContextThemeWrapper

ComponentActivity extends Activity implements

ComponentActivity extends androidx.core.app.ComponentActivity

FragmentActivity extends ComponentActivity

AppCompatActivity extends FragmentActivity

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| java.lang.Object | | | | | | | |
| ↳ | [android.content.Context](https://developer.android.com/reference/android/content/Context.html) | | | | | | |
|  | ↳ | [android.content.ContextWrapper](https://developer.android.com/reference/android/content/ContextWrapper.html) | | | | | |
|  |  | ↳ | [android.view.ContextThemeWrapper](https://developer.android.com/reference/android/view/ContextThemeWrapper.html) | | | | |
|  |  |  | ↳ | [android.app.Activity](https://developer.android.com/reference/android/app/Activity.html) | | | |
|  |  |  |  | ↳ | [androidx.activity.ComponentActivity](https://developer.android.com/reference/androidx/activity/ComponentActivity) | | |
|  |  |  |  |  | ↳ | [androidx.fragment.app.FragmentActivity](https://developer.android.com/reference/androidx/fragment/app/FragmentActivity) | |
|  |  |  |  |  |  | ↳ | androidx.appcompat.app.AppCompatActivity |

1. **Activity life cycle**

oncreate(), onStart(), onResume(), onPouses(), onStop(), onRestart(), onDestroyed()

Let's see the 7 lifecycle methods of android activity.

|  |  |
| --- | --- |
| **Method** | **Description** |
| **onCreate** | called when activity is first created. |
| **onStart** | called when activity is becoming visible to the user.(e.i. Checking the permission, register BR, check the login session) |
| **onResume** | called when user will start interacting with the Activity. (ragister initiate the class when we need to receive data in form of flow like Timer, scheduler, playing video, whether or chat, or video app) |
| **onPause** | called when activity is not visible to the user.(un-ragister or un-initiate the class when we need to receive data in form of flow like Timer, scheduler, pause video , whether or chat, or video app) |
| **onStop** | called when activity is no longer visible to the user. (expire the page base session, unregister the BR) |
| **onRestart** | called after your activity is stopped, prior to start. |
| **onDestroy** | called before the activity is destroyed. |

**Activity Life Cycle @Important Point**

Suppose we have 4 activity ActvityA, ActvityB, ActvityC, ActvityD

1. **When App lunch or click on app icon (if activity A is lunching activity)**
   1. ActivityA : onCreate()
   2. ActivityA : onStart()
   3. ActivityA : onResume()
2. **When Move to another Activity(From A To B)**
   1. ActivityA : onPause()
   2. ActivityB : onCreate()
   3. ActivityB : onStart()
   4. ActivityB : onResume()
   5. ActivityA : onStop()
   6. ActivityA : onSaveInstanceState()
3. **When click on Device Back Button(back from B To A)** 
   1. ActivityB : onPause()
   2. ActivityA : onRestart()
   3. ActivityA : onStart()
   4. ActivityA : onResume()
   5. ActivityB : onStop()
   6. ActivityB : onDestroy()
4. **When we press device back & have only activity A in backStack**
   1. ActivityA : onPause()
   2. ActivityA : onStop()
   3. ActivityA : onDestroy()
5. **When we relaunch app from app back stack (like we have App & press device back button and relaunch app from app stack ) or Open App via Recent**
   1. ActivityB : onCreate()
   2. ActivityB : onStart()
   3. ActivityB : onResume()
6. **When we press device Home**
   1. ActivityA : onPause()
   2. ActivityA : onStop()/
   3. ActivityA : onSaveInstanceState()
7. **When we press device Home and relaunch app from back stack or Open App via Recent**
   1. ActivityA : onRestart()
   2. ActivityA : onStart()
   3. ActivityA : onResume()
8. **When we press device Luck button**
   1. ActivityA : onPause()
   2. ActivityA : onStop()
9. **When we press device release Luck button**
   1. ActivityA : onRestart()
   2. ActivityA : onStart()
   3. ActivityA : onResume()
10. **Killing The app from recent back-stack**
    1. ActivityA : onPause()
    2. ActivityA : onStop()
    3. ActivityA : onDestroy()
11. **When Device configuration change or rotate device screen** 
    1. ActivityA : onPause()
    2. ActivityA : onStop()
    3. ActivityA : onSaveInstanceState()
    4. ActivityA : onDestroy()
    5. ActivityA : onCreate()
    6. ActivityA : onStart()
    7. ActivityA : onRestoreInstanceState()
    8. ActivityA : onResume()
12. **Suppose, when you on ActivityA on your phone is ringing from top | notification** 
    1. Nothing activity will call
13. **Suppose, when you on ActivityA on your phone is ringing from top | notification and on decline button from notification only** 
    1. Nothing activity will call
14. **Suppose, when you on ActivityA on your phone is ringing from top | notification and you just click only on notification icon /message and your call manager activity open with call details**
    1. ActivityA : onPause()
    2. ActivityA : onStop()
    3. ActivityA : onSaveInstanceState()
15. **Suppose, when you on ActivityA on your phone is ringing from top | notification and you just click only on notification icon /message and your call manager activity open with call details and you didn't Answer that call**
    1. ActivityA : onRestart()
    2. ActivityA : onStart()
    3. ActivityA : onResume()
16. **Suppose, when you on ActivityA on your phone is ringing from top | notification and you just answer/pic/received that call**
    1. ActivityA : onPause()
    2. ActivityA : onStop()
    3. ActivityA : onSaveInstanceState()
17. **Suppose, when you on ActivityA on your phone is ringing from top | notification and you just answer/pic/received that call and after some discussion back to activity**
    1. ActivityA : onRestart()
    2. ActivityA : onStart()
    3. ActivityA : onResume()
18. **Switching A-App to B-App**
    1. ActivityA : onPause()
    2. ActivityA : onStop()
    3. ActivityA : onSaveInstanceState()
19. **Switching Back to from B-App to A-App**
    1. ActivityA : onRestart()
    2. ActivityA : onStart()
    3. ActivityA : onResume()

1. **Lunch Mode**

**Activity Lunch Mode standard**

* **standard**
* **singleTop**
* **singleTask**
* **singleInstance**

<activity android:launchMode = [“standard” | “singleTop” | “singleTask” | “singleInstance”] ../>

1. **standard**: This is the default launch mode of activity. If you don’t set any launch mode to your activity, it will use the standard mode by default. It creates a new instance of activity every time even if activity instance is already present.

Suppose we have A, B, C, and D activities and your activity B has standard launch mode. Now again launching activity B

State of Activity Stack before launch B

A →B→C→D

State of Activity Stack after launch B

A →B →C→D→B

We can see that new instance of B is created again

2. **singleTop**: If an instance of activity already exists at the top of the current task, a new instance will not be created and the Android system will route the intent information through onNewIntent().

If an instance is not present on top of the task then a new instance will be created.

Suppose we have A, B, C, and D activities. A →B →C →D and set the C as SingleTop

If we launch C then a new instance of C will be created as it is not on top.

So it will look like A →B →C →D →C

Now suppose we have A →B →C →D →C like this

then we if again launch C activity then in this case new instance will not be created. Instead, we will receive the callback on onNewIntent() method.

3. **singleTask**: If An activity declared with launch mode as singleTask can have only one instance in the system (singleton). At a time only one instance of activity will exist.

If activity instance is not present then the new instance will be created and if the instance is already present in the system then the onNewIntent() method will receive the callback.

Suppose we have A, B, C activities(A →B →C ) and we are launching D that has a singleTask launch mode. In that case, the new instance of D will be created so the current state will look like this. (A →B →C →D)

Now let suppose if we launch B that also have has a singleTask launch mode then current state will look like

A →B

Here old instance gets called and intent data route through onNewIntent() callback. Also, notice that C and D activities get destroyed here.

4. **singleInstance** : It is similar to singleTask except that no other activities will be created in the same task. If another Activity is called from this kind of Activity, a new Task would be automatically created to place that new Activity.

Case 1:

Suppose you have A, B, and C activities(A →B →C) and your activity D has a singleInstance launch mode. In this case, if we launch D then D will be launch in the diffrent task. New task for D will be created.

Task1: A →B →C

Task2 : D (here D will be in the different task)

Now if you continue this and start E and D then Stack will look like

Task1: A →B →C →E

Task2: D

1. **What is Intent**

Android Intent is **the message that is passed between components such as activities, content providers, broadcast receivers, services etc.**

An intent is **a messaging object used to request any action from another app component**. Intents facilitate communication between different activity, fragment, broadcast , service

1. **Type of Intent**

**There are two types of intent**

1. **Explicit intent:**Explicit intent can do the specific application action which is set by the code like changing activity, In explicit intent user knows about all the things like after clicking a button which activity will start and Explicit intents are used for communication inside the application
2. **Implicit Intent:**Implicit intents do not name a specific component like explicit intent, instead declare general action to perform, which allows a component from another app to handle.
3. **What is Pending Intent**

A PendingIntent is a reference to a token maintained by the system. Application A can pass a PendingIntent to application B in order to allow application B to execute predefined actions on behalf of application A; regardless of whether application A is still alive

A Pending Intent is commonly **used to package Intents that will be fired in response to a future event**, such as a Widget or Notification being clicked

Android PendingIntent is an object that wraps up an [intent](https://www.digitalocean.com/community/tutorials/android-intent-handling-between-activities-example-tutorial) object and it specifies an action to be taken place in future

*/\*\* Inteent Object which you want to lunch in future\*/*Intent intent = new Intent(this, SomeActivity.class);  
  
*/\*\* Creating a pending intent and wrapping our intent \*/*PendingIntent pendingIntent = PendingIntent.getActivity(this, 1, intent, PendingIntent.FLAG\_UPDATE\_CURRENT);  
try {  
 // Perform the operation associated with our pendingIntent  
 pendingIntent.send();  
} catch (PendingIntent.CanceledException e) {  
 e.printStackTrace();  
}

*/\*\*Notification class example \*/*NotificationCompat.Builder builder = new NotificationCompat.Builder(this);  
builder.setSmallIcon(android.R.drawable.ic\_dialog\_alert);  
Intent intent = new Intent(Intent.ACTION\_VIEW, Uri.parse("https://www.journaldev.com/"));  
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, intent, 0);  
builder.setContentIntent(pendingIntent);  
builder.setLargeIcon(BitmapFactory.decodeResource(getResources(), R.mipmap.ic\_launcher));  
builder.setContentTitle("Notifications Title");  
builder.setContentText("Your notification content here.");  
builder.setSubText("Tap to view the website.");  
  
NotificationManager notificationManager = (NotificationManager) getSystemService(NOTIFICATION\_SERVICE);  
  
// Will display the notification in the notification bar  
notificationManager.notify(1, builder.build());

1. **What is STICKY Intent**

**Fragment**

1. **Fragment Life Cycle**

**Fragment Life Cycle:**

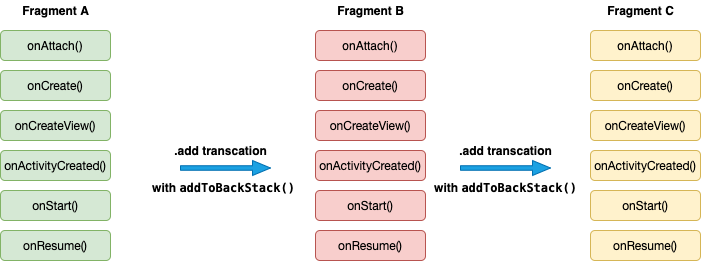
|  |  |
| --- | --- |
| **Method** | **Description** |
| onAttach(Activity) | it is called only once when it is attached with activity. |
| onCreate(Bundle) | It is used to initialize the fragment. |
| onCreateView(LayoutInflater, ViewGroup, Bundle) | creates and returns view hierarchy. |
| onActivityCreated(Bundle) | It is invoked after the completion of onCreate() method. |
| onViewStateRestored(Bundle) | It provides information to the fragment that all the saved state of fragment view hierarchy has been restored. |
| onStart() | makes the fragment visible. |
| onResume() | makes the fragment interactive. |
| onPause() | is called when fragment is no longer interactive. |
| onStop() | is called when fragment is no longer visible. |
| onDestroyView() | allows the fragment to clean up resources. |
| onDestroy() | allows the fragment to do final clean up of fragment state. |
| onDetach() | It is called immediately prior to the fragment no longer being associated with its activity. |

* **Fragment .add Transaction**

For example, there is a **Fragment A**to the **activity**’**s frame layout container**then Fragment A goes through life cycle methods from **onAttach()** to **onResume()**, when you add **Fragment B on top of Fragment A on same container**then Fragment B goes through life cycle methods from **onAttach()**to **onResume()** same happens when you put **Fragment C on top of Fragment B on the same container.**

**Note:**

1. **When we Add fragment B on top A, Then no any method will call from Fragment A**
2. **When Click on Back button Then Current fragment get destroyed and no any method call from backStack fragment**

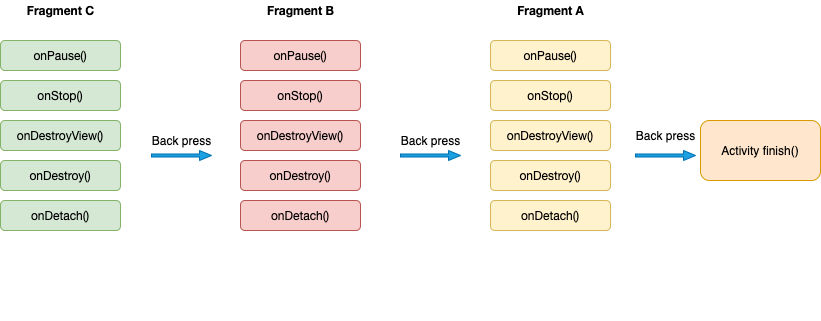


**.add fragment transaction of Fragment A, Fragment B, and Fragment C**

In short when you add a fragment then it calls life cycle methods from onAttach() to onResume().

When you click back press button on an android device then fragment C, fragment B, and Fragment A go through some life cycle methods like below.

Image



**Fragment .replace Transaction**

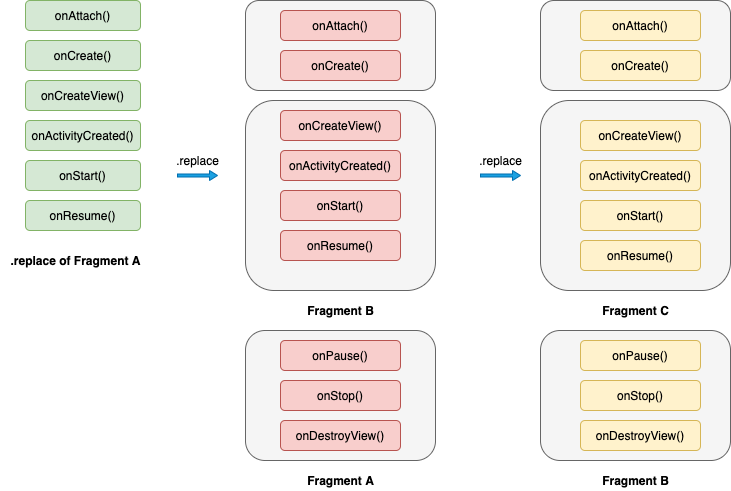
Let’s say you are replacing Fragment A with Fragment B **then first Fragment B**’**s life methods onAttach() and onCreate() gets called** then **Fragment A**’**s life cycle methods get called from onPause() to onDestroyView()**, here **Fragment A won**’**t be detached from the stack it is remembered by the stack.**

Let’s check the below example in which Fragment B is replacing Fragment A and after that Fragment C is replacing Fragment B.

**Note:**

1. When replace Fragment A with B, Then Fragment B life cycle method will from onAttached to OnResume and Fragment A life cycle method will call from onPouse to onDestroyedView.
2. When Press back Button, then current fragment life cycle method will from onPoused to onDeAttached and backStack fragment life cycle method will call onCreateView to onResume

Image

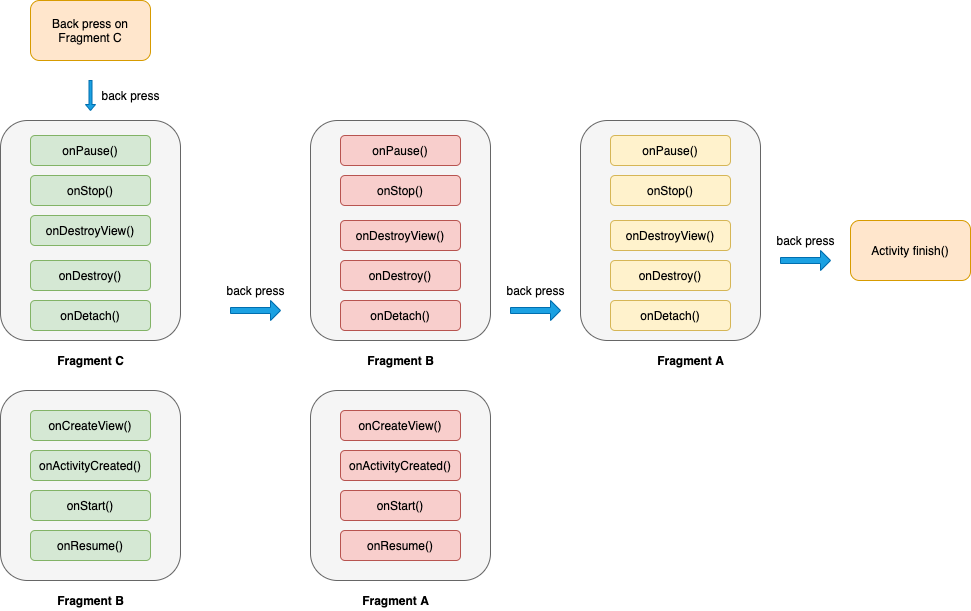


**.replace of Fragment A, then .replace of fragment B and then .replace of fragment C**

**Let**’**s check what happens when we click back press button**

When you click back press button on an android device then fragment C, fragment B, and Fragment A go through some life cycle methods like below.

Image



**back press clicks on fragment C, fragment B, and fragment A respectively**

**add()** method keeps on adding fragments **on top of** the previous fragment in FragmentContainer.

val transaction = supportFragmentManager.beginTransaction()

.add(R.id.fragmentContainer, fragment, "TAG")

.addToBackStack("TAG")

.commit();

**replace()** methods **clears all** the previous Fragment from Containers and then add it in FragmentContainer.

val transaction = supportFragmentManager.beginTransaction()

transaction.replace(R.id.fragment\_layout\_id, homeFragment())

transaction.addToBackStack("TAG")

transaction.commit()

[addToBackStack()](https://developer.android.com/reference/androidx/fragment/app/FragmentTransaction#addToBackStack(java.lang.String)): Calling [addToBackStack()](https://developer.android.com/reference/androidx/fragment/app/FragmentTransaction#addToBackStack(java.lang.String)) commits the transaction to the back stack. The user can later reverse the transaction and bring back the previous fragment by pressing the *Back* button. If you added or removed multiple fragments within a single transaction, all of those operations are undone when the back stack is popped. The optional name provided in the addToBackStack() call gives you the ability to pop back to that specific transaction using [popBackStack()](https://developer.android.com/reference/androidx/fragment/app/FragmentManager#popBackStack(java.lang.String,%20int)).

1. **What is different between Add and Replace Fragment**

**add()** method keeps on adding fragments **on top of** the previous fragment in FragmentContainer.

val transaction = supportFragmentManager.beginTransaction()  
 .add(R.id.fragmentContainer, fragment, "TAG")  
 .addToBackStack("TAG")  
 .commit();

**replace()** methods **clears all** the previous Fragment from Containers and then add it in FragmentContainer.

val transaction = supportFragmentManager.beginTransaction()

transaction.replace(R.id.fragment\_layout\_id, homeFragment())  
 transaction.addToBackStack("TAG")  
 transaction.commit()

1. **What is Fragment Manager**
2. **What is Fragment Transaction**

**Service**

1. **What is service**

A Service is an application component that can perform long-running operations in the background. It does not provide a user interface. Once started, a service might continue running for some time, even after the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

**Caution**: A service runs in the main thread of its hosting process; the service does not create its own thread and does not run in a separate process unless you specify otherwise. You should run any blocking operations on a separate thread within the service to avoid Application Not Responding (ANR) errors

In the context of Android, a service is a component that runs in the background to perform long-running operations. Services do not have a user interface but run independently of the user interface components like activities.

Android services can be used for various purposes, such as playing music in the background, fetching data from a server, performing file downloads, handling network operations, monitoring sensors, and more.

Services in Android can be **started** or **bound**.

**Started services** are initiated by calling the startService() method and continue running until they complete their task or explicitly stop themselves by calling stopSelf(). Started services can also run in the background even if the component that started them is destroyed or the user navigates away from the application.

**Bound services**, on the other hand, allow components to interact with the service by binding to it using the bindService() method. Bound services provide a client-server interface, allowing components to send requests and receive responses from the service. When all clients unbind from a bound service, the service is automatically destroyed

1. **Type of service**

There are two main types of services: Started Services and Bound Services. Additionally, Android also provides a third type of service called Foreground Services

1. **Started Services:**

* Started services are initiated by calling the startService() method.
* They are used for performing background tasks that may take a long time to complete.
* Once started, they continue running until they complete their task or explicitly stop themselves by calling stopSelf() or are stopped by calling stopService().
* Started services can run even if the component that started them (such as an activity) is destroyed or the user navigates away from the application.
* They are commonly used for tasks like playing music, downloading files, or performing network operations in the background.

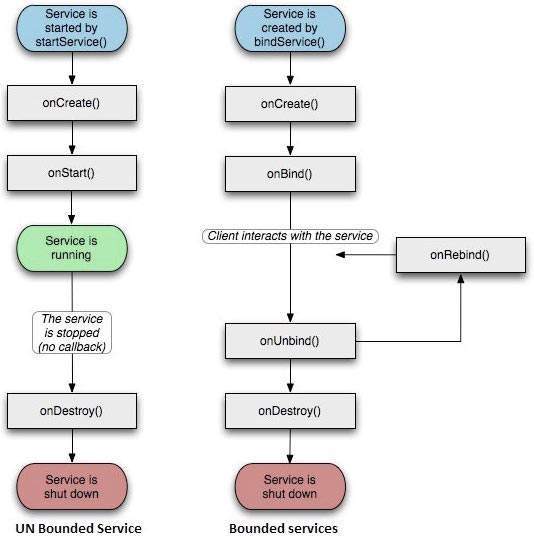
1. **Bound Services:**

* Bound services allow components (such as activities) to bind to them and interact with them using the bindService() method.
* They provide a client-server interface, allowing components to send requests and receive responses from the service.
* Bound services typically offer more complex interactions and can be used for tasks like interprocess communication (IPC).
* When all clients unbind from a bound service, the service is automatically destroyed.
* Bound services are often used for tasks like accessing remote services, implementing a messaging service, or creating a custom content provider.

**3. Foreground Services:**

* Foreground services are a special type of started service that provide a persistent notification to the user while running.
* They have a higher priority than normal background services and are less likely to be killed by the system.
* Foreground services are used for tasks that require ongoing user awareness, such as music playback, navigation, or any service that should be considered important and visible to the user.
* Android requires that foreground services display a notification to keep the user informed about the ongoing operation
* **startForegroundService(). Is uses to start the foreground services**

1. **Life Cycle of Service**



**Note**:

* The [startService()](https://developer.android.com/reference/android/content/Context#startService(android.content.Intent)) method returns immediately, and the Android system calls the service's [onStartCommand()](https://developer.android.com/reference/android/app/Service#onStartCommand(android.content.Intent,%20int,%20int)) method. If the service isn't already running, the system first calls [onCreate()](https://developer.android.com/reference/android/app/Service#onCreate()), and then it calls [onStartCommand()](https://developer.android.com/reference/android/app/Service#onStartCommand(android.content.Intent,%20int,%20int)). Multiple requests to start the service result in multiple corresponding calls to the service's [onStartCommand()](https://developer.android.com/reference/android/app/Service#onStartCommand(android.content.Intent,%20int,%20int)). However, only one request to stop the service (with [stopSelf()](https://developer.android.com/reference/android/app/Service#stopSelf()) or [stopService()](https://developer.android.com/reference/android/content/Context#stopService(android.content.Intent))) is required to stop it.
* If an app that targets API level 28 or higher attempts to create a foreground service without requesting the **FOREGROUND\_SERVICE** permission, the system throws a [**SecurityException**](https://developer.android.com/reference/java/lang/SecurityException).

**Started services** in Android are services that are initiated by calling the startService() method. They are used for performing background tasks that may take a long time to complete. Once started, they continue running until they complete their task or explicitly stop themselves by calling stopSelf() or are stopped by calling stopService().

The **lifecycle of a started service** in Android consists of the following stages:

* **Created**: The service is **created by calling startService(). The onCreate() method** is called, allowing you to perform initialization tasks.
* **Started**: **The onStartCommand() method is called, providing the entry point for the service**. It receives an Intent object that contains the information passed when starting the service. You can extract the relevant data from the Intent and start performing the desired background task.
* **Running**: The service continues to run in the background, executing the desired task. This is where the main work of the service happens. It is important to ensure that long-running operations are performed asynchronously or on a separate thread to avoid blocking the main thread and keeping the application responsive.
* **Stopped**: The service can be stopped in a few different ways:
* The task completes: When the background task is finished, the service can stop itself by calling stopSelf().
* Manual stop: Another component can stop the service by calling stopService(), passing the corresponding Intent.
* System stop: If system resources become scarce, the system may stop the service to free up resources. In this case, the service can be restarted when resources become available.
* **Destroyed**: If the service is stopped and no longer needed, the onDestroy() method is called, allowing you to release any resources or perform cleanup tasks.
* class MyService : Service() {  
    
    
   */\*\* Called when the service is being created. \*/* override fun onCreate() {  
   super.onCreate()  
   // Perform initialization tasks  
   }
* */\*\* The service is starting, due to a call to startService() \*/* override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {  
   // Perform background task here  
   val data = intent?.getStringExtra("data")  
   performBackgroundTask(data)  
    
   // Return START\_STICKY to indicate that the service should be restarted if stopped  
   return *START\_STICKY* }
* */\*\* A client is binding to the service with bindService() \*/* override fun onBind(intent: Intent?): IBinder? {  
   // This method is not relevant for started services  
   return null  
   }
* */\*\* Called when The service is no longer used and is being destroyed \*/* override fun onDestroy() {  
   super.onDestroy()  
   }
* private fun performBackgroundTask(data: String?) {  
   // Long-running operation performed on a separate thread  
   // ...  
   // When the task is complete, stop the service  
   stopSelf()  
   }  
    
    
  }  
    
    
    
    
    
  */\*\* call / start from other component \*/*val *intent* = Intent(context, MyService::class.*java*)  
  intent.putExtra("data", "Some data")  
  context.startService(intent)  
    
    
    
    
  */\*\* Make sure you have provided / register name in me manifest\*/*<service android:name=".MyService" />

**Bound services**

Bound Service in Android are services that allow components (such as activities) to bind to them and interact with them using the bindService() method. They provide a client-server interface, allowing components to send requests and receive responses from the service.

The **lifecycle of a bound service** in Android consists of the following stages:

* **Created**: The service is created **by calling bindService(). The onCreate()** method is called, allowing you to perform initialization tasks.
* **Bound**: The **onBind() method is called,** providing an IBinder object that defines the interface for communication between the service and the client component. The client component can obtain the IBinder object and interact with the service.
* **Running**: The service is now bound to one or more client components, and it can handle requests from those components. The service can perform tasks based on the client's requests and provide any necessary responses.
* **Unbound**: The client component can **unbind from the service by calling unbindService()**. When all clients have unbound from the service, the onUnbind() method is called. This method allows you to perform any cleanup tasks if needed.
* **Destroyed**: If the **service is no longer needed** and there are no clients bound to it, the **onDestroy()** method is called. This is where you can release any resources or perform final cleanup tasks
* class MyService : Service() {  
    
   private val binder = MyBinder()  
    
   inner class MyBinder : Binder() {  
   fun getService(): MyService {  
   return this@MyService  
   }  
   }  
    
    
   */\*\* Called when the service is being created. \*/* override fun onCreate() {  
   super.onCreate()  
   // Perform initialization tasks  
   }  
    
    
   */\*\* A client is binding to the service with bindService() \*/* override fun onBind(intent: Intent): IBinder? {  
   return binder  
   }  
    
    
   fun performTask() {  
   // Perform the requested task  
   }  
    
   fun getData(): String {  
   // Return data to the client  
   return "Some data"  
   }  
    
    
   */\*\* Called when a client is binding to the service with bindService()\*/* override fun onRebind(intent: Intent?) {  
   super.onRebind(intent)  
   }  
    
   */\*\* Called when all clients have unbound with unbindService() \*/* override fun onUnbind(intent: Intent?): Boolean {  
   return super.onUnbind(intent)  
   }  
    
   */\*\* Called when The service is no longer used and is being destroyed \*/* override fun onDestroy() {  
   super.onDestroy()  
   // Perform cleanup tasks or release resources  
   }  
  }  
    
    
    
  */\*\* call / start from other component \*/*val *intent* = Intent(context, MyService::class.*java*)  
  intent.putExtra("data", "Some data")  
  context.boundService(intent)  
    
    
    
  */\*\* Make sure you have provided / register name in me manifest\*/*<service android:name=".MyService" />

1. **What is AIDL**

AIDL stands for Android Interface Definition Language. It is a language used in Android development to define the interface between client and service components in an Android application. AIDL allows inter-process communication (IPC) between different Android components, such as activities, services, and content providers.

AIDL is primarily used when you want to create a client-server architecture within your Android application. It enables communication between different processes by defining methods that can be called remotely by the client process. AIDL supports both synchronous and asynchronous communication between processes.

Here's a brief overview of how AIDL works:

Define the interface: You define the methods and data types that the client can access or use on the server-side using the AIDL syntax. This includes defining the input and output parameters of each method.

Implement the interface: On the server-side, you implement the AIDL interface. This involves writing the code that handles the method calls and performs the necessary actions.

Generate the stubs: AIDL provides a tool called "aidl" that generates the necessary Java stub code for both the client and the server. This code allows the client to interact with the server and vice versa.

Bind to the service: The client application binds to the remote service using the AIDL-generated stubs. This establishes a connection between the client and the server.

Remote method invocation: The client can then invoke the methods defined in the AIDL interface, which are executed on the server-side. The client can pass data to the server and receive the results of method calls asynchronously or synchronously, depending on the implementation.

AIDL is commonly used for scenarios where you need to interact with background services, perform inter-process communication, or create distributed applications in Android

**In this example**, we'll create a simple calculator application where the client sends two numbers to the server, performs a calculation, and returns the result.

**Server | Publisher | Parent App** =>from where you want get

**Step 1: Create AIDL Folder => Right click on your java folder => Folder => AIDL Folder => <your\_folder\_name>**

**Step 2: Enable aidl build feature from gradle file**

buildFeatures **{** aidl true  
**}**

**Step 3: Define the AIDL interface**

First, we need to define the AIDL interface that specifies the methods available for remote invocation. Create a new AIDL file called "Calculator.aidl" with the following code:

**File => New => AIDL**

interface ICalculator {  
 int add(int num1, int num2);  
 int subtract(int num1, int num2);  
 int multiply(int num1, int num2);  
 float divide(float num1, float num2);  
}

**Note:** Now, we need to generate the AIDL stubs that will be used by the client and the server to communicate. In the Android Studio toolbar, click on "Build" -> "Make Project." This will generate the necessary Java stubs for the AIDL interface.

**Step 4: Build the project**. After build android create/generate some file related to your AIDL interface eclass and also create stud class

**Step 5: Implement the AIDL interface**

Next, we need to implement the AIDL interface in the server-side component. Create a new service called CalculatorService that extends the Service class. Implement the AIDL interface and its methods as shown below:

File => New => Services

// CalculatorService.java  
public class CalculatorService extends Service {  
  
 private final ICalculator.Stub binder = new ICalculator.Stub() **{** @Override  
 public int add(int num1, int num2) **{** return num1 + num2;  
 **}** @Override  
 public int subtract(int num1, int num2) **{** return num1 - num2;  
 **}** @Override  
 public int multiply(int num1, int num2) **{** return num1 \* num2;  
 **}** @Override  
 public float divide(float num1, float num2) **{** return num1 / num2;  
 **}  
 }**;  
  
 @Override  
 public IBinder onBind(Intent intent) **{** return binder;  
 **}**}

**Note:** In this code, we create an instance of the ICalculator.Stub class, which is an auto-generated implementation of the AIDL interface. We override the methods defined in the interface with the actual implementation of the calculator operations.

**Step 6: Declare the service in the manifest**

To make the service accessible, we Step 3: Declare the service in the manifest

To make the service accessible, we need to declare it in the Android manifest file. Open the AndroidManifest.xml file and add the following code within the <application> tag:need to declare it in the Android manifest file. Open the AndroidManifest.xml file and add the following code within the <application> tag:

<service  
 android:name=".service.CalculatorService"  
 android:enabled="true"  
 android:exported="true"></service>

**Note:** The android:name attribute specifies the fully qualified name of the service, and android:exported="true" allows other applications to access this service

**Step : 7 Build and run the server application on a device or emulator.**

**Client | Subscriber | Child App** =>Who can get the data from other app

1. Create a new Android project called "ICalculatorClient".
2. Add the AIDL interface from the server application to the client application. Copy the "ICalculator.aidl" file from the server application to the AIDL folder of the client application's.
3. Make sure in both app(Server/Parent and Client/Child) have same Interface class with same name and package name
4. Generate the AIDL stubs by building the client application.
5. In the client application's MainActivity, bind to the remote service and call the remote method:

Like

**AIDL Folder => <your\_folder\_name>**

**Step 2: Enable aidl build feature from gradle file**

buildFeatures **{** aidl true  
**}**

**Step 3: Define the AIDL interface**

First, we need to define the AIDL interface that specifies the methods available for remote invocation. Create a new AIDL file called "Calculator.aidl" with the following code:

**File => New => AIDL**

interface ICalculator {  
 int add(int num1, int num2);  
 int subtract(int num1, int num2);  
 int multiply(int num1, int num2);  
 float divide(float num1, float num2);  
}

**Note:** Now, we need to generate the AIDL stubs that will be used by the client and the server to communicate. In the Android Studio toolbar, click on "Build" -> "Make Project." This will generate the necessary Java stubs for the AIDL interface.

**Step 4: Build the project**. After build android create/generate some file related to your AIDL interface eclass and also create stud class

Step 5: Create a activity and bind remote / server / parent service with your client / child app through stub and call the respective method

// MainActivity.java  
  
public class MainActivity extends AppCompatActivity {  
  
 private ICalculator calculator;  
  
 private ServiceConnection serviceConnection = new ServiceConnection() {  
 @Override  
 public void onServiceConnected(ComponentName componentName, IBinder iBinder) {  
 calculator = ICalculator.Stub.asInterface(iBinder);  
 }  
  
 @Override  
 public void onServiceDisconnected(ComponentName componentName) {  
 calculator = null;  
 }  
 };  
  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.activity\_main);  
  
 Intent intent = new Intent();  
 intent.setComponent(new ComponentName("com.example.calculator", "com.example.calculator.CalculatorService"));  
 bindService(intent, serviceConnection, Context.BIND\_AUTO\_CREATE);  
 }

// Perform a calculation  
 private void performCalculation() {  
 try {  
 int result = calculator.add(5, 3);  
 Log.d("Calculator", "Result: " + result);  
 } catch (RemoteException e) {  
 e.printStackTrace();  
 }  
 }  
  
 @Override  
 protected void onDestroy() {  
 super.onDestroy();  
 unbindService(serviceConnection);  
 }  
}

1. **How to share data between two application**

There many way / technique to share the data B/W two app

1. AIDL: Android Interface Definition Language
2. Content Provider
3. Share Storage(Using file manager)
4. **Work Manager:**

WorkManager is an Android Jetpack library introduced by Google that provides a flexible and efficient way to schedule and run background tasks in Android applications. It is designed to handle various types of background work, such as syncing data with a server, performing periodic tasks, or executing tasks when certain conditions are met.

WorkManager is built on top of several other background task execution mechanisms, such as JobScheduler & AlarmManager and It automatically selects the appropriate underlying mechanism based on the device's API level, ensuring compatibility across different Android versions.

Android WorkManager is a background processing library which is used to execute background tasks which should run in a guaranteed way but not necessarily immediately. With WorkManager we can enqueue our background processing even when the app is not running and the device is rebooted for some reason. WorkManager also lets us define constraints necessary to run the task e.g. network availability before starting the background task.

**Type of work Manager**

i. **one-time tasks** => Use for one-time execution

ii. **periodic tasks** => Periodic execution

**doWork**() method is responsible to execute your task on the background thread. Whatever task you want to perform has to be written here.

Result returns the status of the work done in **doWork()** method.

If it returns **Result.success()** it means the task was successful

if the status is **Result.failure()** , the task was not-successful and lastly,

if it returns **Result.retry()** it means the task will execute again after some time.

**Key features and benefits of using WorkManager** :

**Compatibility**: Backwards compatibility for devices running Android 4.4 (API level 19) and higher.

**Flexibility**: It allows you to define complex scheduling requirements for your background tasks, including one-time tasks, periodic tasks, and tasks with constraints (such as network availability or device charging status or device status, or memory status).

**Persistence**: WorkManager ensures that scheduled tasks are persisted across device reboots, so your tasks can resume execution even after a restart.

**Observability**: You can monitor the status and progress of your background tasks using LiveData, RxJava, or Kotlin coroutines. WorkManager also supports chaining and parallel execution of tasks.

Example:

**Note-1:**

In Order to uses Work Manager, You 1st need to create sub-class of Worker class and implement the doWork() method in your child class

//Gradle Dependency for Work Manager  
implementation "androidx.work:work-runtime:2.8.1"

import android.content.Context  
import androidx.work.Constraints  
import androidx.work.Data  
import androidx.work.ListenableWorker  
import androidx.work.NetworkType  
import androidx.work.OneTimeWorkRequestBuilder  
import androidx.work.PeriodicWorkRequestBuilder  
import androidx.work.WorkManager  
import androidx.work.Worker  
import androidx.work.WorkerParameters  
import com.example.kotlinrndproject.apiclient.NetworkModule  
import kotlinx.coroutines.Dispatchers  
import kotlinx.coroutines.GlobalScope  
import kotlinx.coroutines.async  
import kotlinx.coroutines.delay  
import java.util.concurrent.TimeUnit  
  
class MyWorkerClass (ctx: Context, param: WorkerParameters) : Worker(ctx, param){

//Do your work header whatever you want do  
 override fun doWork(): Result {  
 // Perform the network request and fetch data from the server  
 // Handle the response and update the UI accordingly  
  
 // Inside the doWork() method of MyWorker class  
 val response = makeNetworkRequest()  
 if(response){  
 val data = response.body()  
 val outputData = Data.Builder()  
 .putString("response", data)  
 .build()  
 //Return Result.success() if the task is successful  
 return Result.success(outputData)  
 } else {  
 // Handle failed response  
 return Result.failure()  
 }  
 }

//Execute your API or long running operation on background thread  
 private fun makeNetworkRequest(): Boolean {  
 val res = GlobalScope.*async*(Dispatchers.IO) **{** delay(1000)  
 val httpRequest = NetworkModule.provideApiService()  
 val data = httpRequest.getUsers()  
 if(data.isEmpty()){  
 false;  
 }else{  
 true;  
 }  
 **}** return true  
 }  
}

**Schedule a one-time work request**: To schedule a one-time work request, use the OneTimeWorkRequestBuilder class and set the required constraints.

val myOneTimeWorkRequest = *OneTimeWorkRequestBuilder*<MyWorkerClass>()  
 .setConstraints(getConstraints()) // Custom method to define the constraints  
 .build()  
  
 WorkManager.getInstance(context).enqueue(myWorkRequest)

**Schedule a periodic work request:** To schedule a periodic work request, use the PeriodicWorkRequestBuilder class and set the required constraints and interval.

val myPeriodicWorkRequest = *PeriodicWorkRequestBuilder*<MyWorkerClass>(1, TimeUnit.*HOURS*)  
 .setConstraints(getConstraints()) // Custom method to define the constraints  
 .build()  
  
 WorkManager.getInstance(context).enqueue(myPeriodicWorkRequest)

**Define constraints:** Create a method to define the constraints for your work request, considering network availability, charging, and device idle status  
 private fun getConstraints(): Constraints {  
 return Constraints.Builder()  
 .setRequiresDeviceIdle(true) //checks whether device should be idle for the WorkRequest to run  
 .setRequiresCharging(true) //checks whether device should be charging for the WorkRequest to run  
 .setRequiredNetworkType(NetworkType.*CONNECTED*) //checks whether device should have Network Connection  
 .setRequiresBatteryNotLow(true) // checks whether device battery should have a specific level to run the work request  
 .setRequiresStorageNotLow(true) // checks whether device storage should have a specific level to run the work request  
 .build()  
 }

**Handle the response:**

To handle the response from the server, you can use the Result.success() or Result.failure() methods in your worker class's doWork() method. You can also pass data between the worker and its caller using the Data class.

val workInfoLiveData = WorkManager.getInstance(context).getWorkInfoByIdLiveData(myPeriodicWorkRequest.id)  
 workInfoLiveData.observe(lifecycleOwner, { workInfo ->  
 if (workInfo != null) {  
 val workStatus = workInfo.state  
 val progress = workInfo.progress  
 // Update UI based on work status and progress  
 // ...  
 }  
 })

**Note**: By following these steps, you can schedule a work request using WorkManager, handle the response from the server, and check the progress/status of the work to update the UI accordingly.

**Chaining in Series**

Means chaining the work request or execute multiple work request or series of work request worker one after

WorkManager.getInstance(context)  
 .beginWith(yourWorkRequestOne)  
 .then(yourWorkRequestTwo)  
 .then(yourWorkRequestThree)  
 .enqueue()

**Parallel Chaining =>**

WorkManager.getInstance(myContext)  
 .beginWith(listOf(work1, work2, work3))  
 .then(work4)  
 .then(work5)  
 .enqueue()

**Note :** If the first task fails the subsequent task following it will also give failure response and if the first task is cancelled all the following task will also be cancelled.

**WorkManager Cancle =>** To cancel a task we use

WorkManager.cancelWorkById(workRequest.id)

**Input/output for your task**  
  
Passing input data to a worker: Passing input data from worker request / requester and get it in in worker class in doWork method

val inputData = Data.Builder()  
 .putString("key", "value")  
 .putInt("count", 10)  
 .build()  
  
 val myWorkRequest = *OneTimeWorkRequestBuilder*<MyWorker>()  
 .setInputData(inputData)  
 .build()

//in sub class of Worker or reciver class  
 override fun doWork(): Result {  
 val inputString = inputData.getString("key")  
 val count = inputData.getInt("count", 0)  
  
 Result Result.success()  
 }

**Returning output data from a worker:** =>

Inside the doWork() method of MyWorker class

Passing data from Worker class to client / requester

To return output data from a worker, you can use the Result.success(outputData) method inside the doWork() method. The output data can be accessed outside the worker using the getResult() method of the work info.

override fun doWork(): Result {  
 val outputData = Data.Builder()  
 .putString("result", "success")  
 .build()  
  
 return Result.success(outputData)  
 }

To retrieve the output data outside the worker, you can observe the work info using WorkManager.getWorkInfoByIdLiveData() and access the output data using the getOutputData() method.

val workInfoLiveData = WorkManager.getInstance(context).getWorkInfoByIdLiveData(workId)  
 workInfoLiveData.observe(lifecycleOwner, { workInfo ->  
 if (workInfo != null && workInfo.state == WorkInfo.State.SUCCEEDED) {  
 val outputData = workInfo.outputData  
 val result = outputData.getString("result")  
 // Handle the output data  
 // ...  
 }  
 })

1. **Job Scheduler**

The Android JobScheduler API helps schedule background tasks that are executed in your application. This API allows you to execute batch jobs that are nonuser facing jobs or that do not require any interaction, such as jobs that download/upload data from the network.

**Note:** Now onwards Job Scheduler already deprecated for next version of Android so now we have WorkManager instead.

1. **IntentServices**

An IntentService is a subclass of Service in Android that is used to handle asynchronous requests (expressed as “Intents”) on demand. It runs in the background and stops itself once it has processed all the intents that were sent to it. IntentService has abstract method **onHandleIntent(Intent intent)** take a request in quee and create sparate work for execution and once execution done service is stop.

**Note**: As of Android API 30 (otherwise known as Android 11) the IntentService class has been deprecated

1. **JonIntentServices**

You can say JobIntentService is a modern way to run the background service from the background application. JobIntentService works in the same way as a Service however it enqueues the work into the JobScheduler on compatible Android targets( SDK 26 or more).

**Note**: Unfortunately JobIntentServices have been deprecated after it was already the replacement for IntentServices

**Alarm Manager**

Alarms (based on the [AlarmManager](https://developer.android.com/reference/android/app/AlarmManager) class) give you a way to perform time-based operations outside the lifetime of your application. For example, you could use an alarm to initiate a long-running operation, such as starting a service once a day to download a weather forecast

**Note:** If your app targets Android 12 or higher, you must [declare one of the "Alarms & reminders"](https://developer.android.com/training/scheduling/alarms#exact-permission-declare) permissions. Otherwise, a [**SecurityException**](https://developer.android.com/reference/java/lang/SecurityException) occurs.

**Thread**

A thread is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.

When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named main of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.

All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread. This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started. For example, a thread that computes primes larger than a stated value could be written as follows:

class PrimeThread extends Thread {  
 long minPrime;  
 PrimeThread(long minPrime) {  
 this.minPrime = minPrime;  
 }  
  
 public void run() {  
 // compute primes larger than minPrime  
 . .  
 }  
}

The following code would then create a thread and start it running:

PrimeThread p = new PrimeThread(143);  
p.start();

**Runnable** : The other way to create a thread is to declare a class that implements the Runnable interface. That class then implements the run method. An instance of the class can then be allocated, passed as an argument when creating Thread, and started. The same example in this other style looks like the following:

class PrimeRun implements Runnable {  
 long minPrime;  
 PrimeRun(long minPrime) {  
 this.minPrime = minPrime;  
 }  
  
 public void run() {  
 // compute primes larger than minPrime  
 }  
}

The following code would then create a thread and start it running:

PrimeRun p = new PrimeRun(143);  
new Thread(p).start();

1. **How do you perform background opration in / from service class**

Using handler and looper. Handler and Looper are often used together to implement asynchronous operations, background tasks, and UI updates in Android applications. They provide a way to schedule work on a particular thread and handle the results when the processing is complete.

Note:

*/\*\*   
1st Create Runnable Object wich implement run() method*

*2nd Create Handler Object which Constructor accept Looper.myLooper() instance*

*3rd call handler postDelayed() with runnableObj and delayed time  
 \*/*

private Handler handlerObj;  
// Create a new Handler associated with the main thread's Looper  
handlerObj = new Handler(Looper.myLooper());  
  
// Run the background task after a delay  
handlerObj.postDelayed(runableObj, 2);

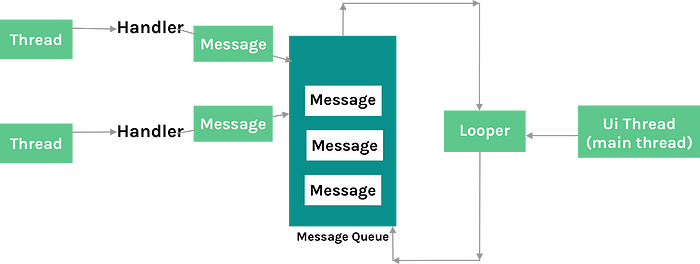
private Runnable runableObj = new Runnable() {  
 @Override  
 public void run() {  
 // Perform the background task here  
 // This method runs on the background thread  
 // Run the background task again after a delay  
 handlerObj.postDelayed(runableObj, 1);  
 }  
};

OR

Handler handlere = new Handler(Looper.*myLooper*()){

@Override  
 public void handleMessage(Message msg){   
 if(msg.what == 0){  
 // updateUI();  
 }else{  
 //showError();  
 }  
 }  
};  
  
  
handlere.sendEmptyMessage(0);  
handlere.sendEmptyMessage(1);

1. **What is Thread, MessageQues, Looper, Handler**



1. **Thread** =>
2. **MessageQueue** => It is a low-level class, and it’s holding the list of messages to be dispatched by a Looper. Messages are not added directly to a MessageQueue, but rather through Handler objects associated with the Looper
3. **Looper** => It loops over a MessageQueue which contains the messages to be dispatched. The actual task of managing the queue is done by the Handler which is responsible for handling (adding, removing, dispatching) messages in the message queue.
4. a Looper is a class that allows you to create a message queue for a thread.
5. It is typically used to handle tasks and events in a sequential manner, without blocking the thread
6. A Looper runs in a continuous loop, retrieving messages from the message queue and processing them one by one.
7. The main purpose of a Looper is to enable message passing and event handling within a thread.
8. It provides a way for different components of an application to communicate and execute tasks asynchronously.
9. The Looper will take care of processing these messages or executing the runnables in the order they were added by Handler.
10. **Handler** => It allows you to send and process Message and Runnable objects associated with a thread's MessageQueue. Each Handler instance is associated with a single thread and that thread's message queue
11. It is used to send and process messages and runnable objects in a thread's message queue.
12. A Handler allows you to schedule tasks, delay execution, and perform other operations on the message queue. The Looper will take care of processing these messages or executing the runnables in the order they were added.

The **Handler** and **Looper** are closely related components in Android that work together to handle message processing and threading. Here's a breakdown of the differences between them:

1. Functionality:
   * **Looper**: A **Looper** is responsible for creating and managing the message queue for a particular thread. It continuously loops through the message queue, retrieves messages, and dispatches them for processing.
   * **Handler**: A **Handler** is an object that is associated with a specific **Looper**. It allows you to send messages and runnables to the message queue of the associated **Looper** and handles these messages or executes the runnables in the order they were added.
2. Creation:
   * **Looper**: You create a **Looper** by calling **Looper.prepare()** and **Looper.loop()** in the thread's **onCreate()** method. This sets up the message queue and starts the loop for that thread.
   * **Handler**: You create a **Handler** by instantiating it with a reference to the **Looper** of the desired thread. It binds the **Handler** to that **Looper** for message processing.
3. Thread Association:
   * **Looper**: Each thread can have its own associated **Looper**, and it manages the message queue for that specific thread.
   * **Handler**: A **Handler** is associated with a specific **Looper**. It can only send messages and runnables to the message queue of the **Looper** it is bound to.
4. Message Processing:
   * **Looper**: The **Looper** runs in a loop, continuously checking the message queue for new messages. It retrieves and dispatches each message to the appropriate **Handler** for processing.
   * **Handler**: The **Handler** receives messages or runnables from the message queue and processes them within its **handleMessage()** method or executes the runnables using **Runnable.run()**.
5. Communication:
   * **Looper**: The **Looper** provides the infrastructure for inter-thread communication by managing the message queue.
   * **Handler**: The **Handler** facilitates communication between threads by allowing them to send messages and runnables to each other's message queues

**Handler Example for Android:**

*/\*\*  
  
 Pre Requirement   
 1. One Thread class -> where looper prepare, handler initialize for looper, looper loop*

*2. Tow Handler -> 1-for for looper(Looper Handler) and 2-for handle message(Caller-Handler)*

*3. One Runnable -> Looper handler call post method with nunnable param*

*4. Message -> create instance using looper handler and send message using  
 \* \*\*/*

*/\*\* Creating 10 thread with diff name\*\*/*for(int i =0; i<10; i++) {  
  
 */\*\*Create the MyThread Class Instance \*/* MyThread myThread = new MyThread();  
  
 */\*\*start the thread using MyThread class Instance \*/* myThread.start();  
  
 */\*\* Wait and check for MyThread class Looper & Handler Initialization \*/* while (!(myThread.isReady)) {  
  
 }  
  
 */\*\*Handler for handle massage / response which send my message Que \*/* Handler mainHander = new Handler(Looper.*getMainLooper*()) {  
 @Override  
 public void handleMessage(@NonNull Message msg) {  
 int pg = msg.arg1;  
 //Log.e("Handler", "handleMessage Progress: " + pg + " Thread Name: " + Thread.currentThread().getName());  
 }  
 };  
  
 */\*\*Setting The name of Thread \*/* myThread.setName("MyThread Rudra:-"+i);  
  
 */\*\* Calling the Long Running task from MyTread for execute task in background \*/* myThread.startDownloading(mainHander);  
  
}

import android.os.Handler;  
import android.os.Looper;  
import android.os.Message;  
import android.util.Log;  
  
public class MyThread extends Thread{  
  
 Handler hander = null;  
 public Boolean isReady = false;  
  
 public static final int *progress* =1 ;  
  
  
 @Override  
 public void run() {  
 super.run();  
  
 */\*\*  
 Initialize the current thread as a looper.  
 This gives you a chance to create handlers that then reference this looper, before actually starting the loop.  
 Be sure to call loop() after calling this method, and end it by calling quit().  
 \*/* Looper.*prepare*();  
  
 hander = new Handler(Looper.*myLooper*());  
 isReady = true;  
  
 */\*\*  
 Run the message queue in this thread. Be sure to call quit() to end the loop.  
 \*/* Looper.*loop*();  
 //Log.e("Handler 1 ", "startDownloading MyThread Run Thread Name "+Thread.currentThread().getName());  
 }

*/\*\*This method execute the long running task with background or separate thread \*/* public void startDownloading(Handler mainHandler){  
 */\*\*  
 Checking is current thread is to this thread  
 This no then we post the the handler task with help of runnable  
 Which create a separate thread for us and run tha task in background thread  
 \* \*/* if(Thread.*currentThread*() != this) {  
 */\*\*  
 Causes the Runnable r to be added to the message queue.  
 The runnable will be run on the thread to which this handler is attached.  
 \* \*/* hander.post(new Runnable() {  
 @Override  
 public void run() {  
 //Log.e("Handler 3 ", "startDownloading run Thread Name: "+Thread.currentThread().getName());  
 startDownloading(mainHandler);  
 }  
 });  
 return;  
 }  
  
  
  
 */\*\* Here we are performing the long running task \*/* for (int i = 0; i <= 100; i++) {  
 try {  
 Thread.*sleep*(1000);  
 //Log.e("Handler 4 ", "startDownloading run loop:"+i +" Thread Name: "+Thread.currentThread().getName());  
  
 }catch (Exception e){  
 e.printStackTrace();  
 }  
  
 */\*\*Creating Message que as task  
 Defines a message containing a description and arbitrary data object that can be sent to a Handler.  
 This object contains two extra int fields and an extra object field that allow you to not do allocations in many cases.  
 Same as obtain(), but sets the value for the target member on the Message returned.  
 \* \*/* Message msg = Message.*obtain*(hander);  
 msg.arg1 = i;  
 msg.what = *progress*;  
  
 */\*\*Handler Adding the task / msg in in message-que for which looper execute it one by one  
 Pushes a message onto the end of the message queue after all pending messages before the current time.  
 It will be received in handleMessage, in the thread attached to this handler.  
 \* \*/* mainHandler.sendMessage(msg);  
 }  
  
 }  
  
  
  
}

**Broadcast Receiver**

1. **What is broadcast receiver**

A broadcast receiver (receiver) is **an Android component which allows you to register for system or application events**.

Broadcast in android is the system-wide events that can occur when the device starts, when a message is received on the device or when incoming calls are received, or when a device goes to airplane mode, etc. Broadcast Receivers are used to respond to these system-wide events. Broadcast Receivers allow us to register for the system and application events, and when that event happens, then the register receivers get notified.

**Note**: Since from API Level 26, most of the broadcast can only be caught by the dynamic receiver

1. **How to or Way of Register the broadcast**
2. Static => from menifest file => These types of Receivers are declared in the manifest file and works even if the app is closed.

<receiver android:name=".MyBroadcastReceiver"  
android:permission="android.permission.BLUETOOTH\_CONNECT">  
<intent-filter>  
 <action android:name="android.intent.action.ACTION\_FOUND"/>  
</intent-filter>  
</receiver>

1. Dynamic => from app component These types of receivers work only if the app is active or minimized.

IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION\_FOUND);

filter.addAction(BluetoothDevice.ACTION\_NAME\_CHANGED);

registerReceiver(mReceiver, filter);

1. **How to De-Register broadcast receiver**

unregisterReceiver(receiver)

1. **How register action and received message for broadcast**

class SampleBootReceiver : BroadcastReceiver() {  
 fun onReceive(context: Context?, intent: Intent) {  
 if (intent.*action* == "android.permission.BLUETOOTH\_CONNECT") {  
 // Set the alarm here.  
 }  
 }  
}

1. **How to send Broadcast manually or through programmatically with own / custom actions**

Intent intent = new Intent();  
intent.setAction("MyCustomeAction");  
intent.putExtra("data", "Nothing to see here, move along.");  
sendBroadcast(intent);  
  
  
  
class MyBroadcastReceiver : BroadcastReceiver() {  
 fun onReceive(context: Context?, intent: Intent) {  
 if ("MyCustomeAction" == intent.*action*) {  
 val value = intent.getStringExtra("data")  
 }  
 }  
}

**OPPS**

1. **What is class**
2. **What is object**
3. **What is Inheritance**
4. **What is Abstract class**
5. **What is Abstraction**
6. **What is Interface**
7. **Different B/W Abstract & Interface**
8. **What is Encapsulation**
9. **What is Polymorphism**
10. **What is Overloading**
11. **What is overriding**
12. **Different B/W Compile & Runtime Polymorphism**

**Kotlin & OPPS**

1. **How to create class in kotlin**
2. **How many type of Constrictor in Kotlin and how to call them**
3. **How to achieve Inheritance in kotlin**
4. **How to Create Abstract class in kotlin and extent it in child class**
5. **How create inheritance in kotlin and how to implement it in child class**
6. **What is Data class**
7. **What is Object class**
8. **What is Companion Object**
9. **How to declare variable in kotlin**
10. **What is lateinit var in kotlin**
11. **What is Lazy in Kotlin**
12. **What is Different B/W in Var & Val**
13. **What is Different B/W lateinit var & Lazy**
14. **What is Const keyword in kotlin**
15. **What is Different B/W Const & Val keyword**
16. **What is lamda function**

Lambda functions in Kotlin are a concise way to define anonymous functions. They are often used with higher-order functions, which are functions that can accept other functions as parameters or return them as results

Lambda is a function which has no name. Lambda is defined with a curly braces **{}** which takes variable as a parameter (if any) and body of function. The body of function is written after variable (if any) followed by **->** operato

//Lamda funtion

val sum = { x: Int, y: Int -> x + y }

//Lamda funtion

val sum: (Int, Int) -> Int = { x, y -> x + y }

1. **What is Extension function**

In Kotlin, extension functions allow you to add new functions to existing classes without modifying their source code. You can think of extension functions as a way to extend the behavior of a class without actually inheriting from it. This feature is particularly useful when you want to add utility functions or enhance existing classes with additional functionality.

//Extention funtion Example  
fun String.PrintName(strSirName: String): String {  
 *println*("full name is $this $strSirName ")  
 return "$this + $strSirName";  
}

//calling Extention funtion   
val *printNameExt* = "Abhishek".*PrintName*("Rai")

1. **What is inline function**

In Kotlin, an inline function is a function modifier that suggests the compiler to inline the function's code at the call site. When a function is marked as "inline," the compiler replaces the function call with the actual code of the function, eliminating the overhead of the function call itself.

The inline functions are expanded during the compilation process, which means that the function's code is copied wherever the function is called instead of creating a separate function call. This can lead to improved performance by reducing the overhead of function calls, especially when the function is called frequently or from within a loop.

When you mark a function as inline in Kotlin, you're suggesting the compiler to replace the function call with the actual code of the function at the call site. This means that instead of generating bytecode for the function call and jumping to the function's code, the compiler copies the function's code directly where the function is called.

inline fun calculateSum(a: Int, b: Int): Int {  
 return a + b  
}  
  
val result = calculateSum(3, 4)

In this example, the calculateSum function is marked as inline. When the code is compiled, the compiler replaces the function call calculateSum(3, 4) with the actual implementation of the function, which is return 3 + 4. As a result, the generated bytecode will contain the direct addition of 3 and 4 without any function call overhead.

1. **What is Infix function**

In Kotlin, the infix keyword is used to define an infix function. An infix function allows you to call a function using a special infix notation, where the function name is placed between the operands. It provides a more readable and expressive way of invoking certain functions, especially when they represent operations between two objects.

* To define an infix function, you need to follow these rules:
* The function must be a member or an extension function of a class or a top-level function.
* The function must have a single parameter.
* The function must be marked with the infix modifier.

val *result* = 5 *add* 3  
  
  
infix fun Int.add(other: Int): Int {  
 return this + other  
}

1. **What is Higher Order Function**

In Kotlin, a higher-order function is a function that can accept other functions as parameters and/or return functions as results. This concept is based on the principles of functional programming, where functions are treated as first-class citizens.

*/\*\* Lamda funtion \*\*/*var sumLamda : (Int, Int -> Int = { x, y -> x + y }

*/\*\* Normal Funtion \*\*/*fun sumNorFun(x: Int, y: Int): Int {  
 return x \* y  
}  
  
  
*/\*\* Higher Order Funtion \*\*/*fun higherOrderFunction(n: Int, dynamicMethodWithTwoIntParm: (Int, Int) -> Int): Int{  
 return dynamicMethodWithTwoIntParm(n + 3, 8)  
}  
  
  
*/\*\* Calling the Higher Order function with normal function  
Note: Mark double :: when you are passing function but not required with lamda function. Like  
 \*/*val sumVal: Int = higherOrderFunction(10, :: sumNorFun)   
  
  
*/\*\* Calling the Higher Order function with lamda function \*\*/*val sumVal: Int = higherOrderFunction(10, sumLamda)

1. **What is Scope Function**
2. **Type & uses of Scope Function**
3. **What is Nullable & NullType**
4. **What is Null Safety**
5. **What is Safe Call (?), Assertion Call (!!)**
6. **Access Modifier or Visibility in Kotlin**

* **Private**: The private modifier restricts the visibility of a member to the containing class only. A private member cannot be accessed from outside the class.
* **Internal**: The internal modifier restricts the **visibility** of a member to the **same module**. A module is a set of Kotlin files compiled together.
* **Protected**: The protected modifier restricts the **visibility** of a member to the **containing class and its subclasses / child class**. A protected modifier with class or interface allows visibility to its class or subclass only. A protected declaration (when overridden) in its subclass is also protected modifier unless it is explicitly changed.
* **Public**: The public modifier makes a member visible to any code. This is the default visibility for members in Kotlin. A **public modifier is accessible from everywhere in the project**. It is a default modifier in Kotlin. If any class, interface etc. are not specified with any access modifier then that class, interface etc. are used in public scope.

1. **What is Delegation in Kotlin**

Delegation means passing the responsibility to another class or method. When a property is already declared in some places, then we should reuse the same code to initialize them.

Kotlin supports **“delegation”** design pattern by introducing a new keyword **“by”**. Using this keyword or delegation methodology, Kotlin allows the derived class to access all the implemented public methods of an interface through a specific object. The following example demonstrates how this happens in Kotlin.

interface Base {  
 fun printMe() //abstract method  
}

class BaseImpl(val x: Int) : Base {  
 override fun printMe() { *println*(x) } //implementation of the method  
}

class Derived(b: Base) : Base by b // delegating the public method on the object b  
  
fun main(args: Array<String>) {  
 val b = BaseImpl(10)  
 Derived(b).printMe() // prints 10 :: accessing the printMe() method   
}

**Live Data**

1. **What is Live Data**

**LiveData** is a data holder class provided by the Android Jetpack library. It's designed to hold and emit data in a lifecycle-aware manner, primarily used for building reactive and observable components in Android applications. It allows you to observe changes in data and react accordingly. It ensures that the observers (such as UI components) are updated only when they are active and can receive the data, avoiding potential crashes or memory leaks.

LiveData is a data holder class that can be observed within a given lifecycle. This means that an Observer can be added in a pair with a LifecycleOwner, and this observer will be notified about modifications of the wrapped data only if the paired LifecycleOwner is in active state. LifecycleOwner is considered as active, if its state is androidx.lifecycle.Lifecycle.State.STARTED or androidx.lifecycle.Lifecycle.State.RESUMED. An observer added via observeForever(Observer) is considered as always active and thus will be always notified about modifications. For those observers, you should manually call removeObserver(Observer).

An observer added with a Lifecycle will be automatically removed if the corresponding Lifecycle moves to androidx.lifecycle.Lifecycle.State.DESTROYED state. This is especially useful for activities and fragments where they can safely observe LiveData and not worry about leaks: they will be instantly unsubscribed when they are destroyed.

In addition, LiveData has onActive() and onInactive() methods to get notified when number of active Observers change between 0 and 1. This allows LiveData to release any heavy resources when it does not have any Observers that are actively observing.

// Define a LiveData object in your ViewModel

val myLiveData: MutableLiveData<String> = MutableLiveData()

// Set initial value

myLiveData.value = "Hello, LiveData!"

// Observe the LiveData from your UI component

myLiveData.observe(this) { value ->

// Update UI with the new value

textView.text = value

}

// Update the value of LiveData

myLiveData.value = "LiveData is awesome!"

In the LiveData class in Kotlin, there are two methods available for updating the value of LiveData: setValue() and postValue(). The main difference between them lies in how they handle the threading.

**setValue(value: T)**: This method is used to set the value of LiveData from the main/UI thread. It is a synchronous operation, meaning it updates the value immediately. If there are active observers, they will be notified and receive the new value. However, if you call setValue() from a background thread, it will throw an exception

Ex: myLiveData.value = newValue

**postValue(value: T):** This method is used to set the value of LiveData from a background thread. It is an asynchronous operation, meaning it will schedule the value update to be executed on the main/UI thread. This is useful when you need to update LiveData from a background thread, such as when performing network requests or database operations.

Ex: myLiveData.postValue(newValue)

**Note**: It's worth noting that postValue() guarantees that the value will be delivered, even if multiple calls are made in rapid succession. The observers will receive the most recent value, and intermediate values will be skipped

1. **Type of Live Data**

There is two type of livedata class

1. LiveData:
2. MutableLiveData

**Note**: Remember that LiveData is an abstract class and MutableLiveData is a concrete subclass that provides mutable operations. In most cases, you'll use MutableLiveData when creating instances of LiveData to be able to update the data using setValue() or postValue().

1. **How to update & post the data in LiveData**

Posts a task to a main thread to set the given value. So if you have a following code executed in the main thread:

liveData.postValue("a");

liveData.setValue("b");

The value "b" would be set at first and later the main thread would override it with the value "a".

If you called this method multiple times before a main thread executed a posted task, only the last value would be dispatched.

[**setValue()**](https://developer.android.com/reference/androidx/lifecycle/LiveData#setValue(T)): Sets the value. If there are active observers, the value will be dispatched to them. This method must be called from the main thread.

Ex: myLiveData.value = newValue

[**postValue()**](https://developer.android.com/reference/androidx/lifecycle/LiveData#postValue(T)): Posts a task to a main thread to set the given value. If you called this method multiple times before a main thread executed a posted task, only the last value would be dispatched. The calling **postValue()** method on a background thread does not guarantee that the latest value set will be received.

Ex: myLiveData.postValue(newValue)

1. **What is Different B/W postValue() and setValue()**

[**setValue()**](https://developer.android.com/reference/androidx/lifecycle/LiveData#setValue(T)): Sets the value. If there are active observers, the value will be dispatched to them. This method must be called from the main thread.

[**postValue()**](https://developer.android.com/reference/androidx/lifecycle/LiveData#postValue(T)): Posts a task to a main thread to set the given value. If you called this method multiple times before a main thread executed a posted task, only the last value would be dispatched.

| **setValue()** | **postValue()** |
| --- | --- |
| The setValue is a class that holds observable data. | The postValue() requires certain observable values |
| Live data is lifecycle-aware. | postValue() is not lifecycle aware. |
| The postValue method’s responsibility is to post or add a task to the application’s main thread whenever the value changes | The postValue method’s responsibility is to post or add a task to the application’s main thread whenever the value changes |
| The value will be updated whenever the main thread runs. | The value will be updated whenever the main thread runs later |
| You cannot use setValue if you are working in a background thread | You can use postValue if you are working in a background thread |
| The value is changed immediately | The value is changed after an interval |
| In the case of setValue() the said value is called twice, and the value is updated twice, and the observers are notified about the updated data twice. | In the case of postValue(), the value will be updated twice, and the number of times the observers will receive the notification is determined by the main thread’s execution. |

1. **How to Observe Data in Activity & Fragment**

// Observe the LiveData from your UI component

myLiveData.observe(this) { value ->

// Update UI with the new value

textView.text = value

}

**Flow**

1. **What is Flow**
2. **How to Push/Emit data in Flow**
3. **How to Observe Flow data in Activity & Fragment**
4. **Operator in Flow**
5. **Different B/W LiveData & Flow**

**DataBinding**

**ViewBinding**

**ViewModel**

1. **What is ViewModel**

Is it an Abstract class.

ViewModel is a class that is responsible for preparing and managing the data for an Activity or a Fragment. It also handles the communication of the Activity / Fragment with the rest of the application (e.g. calling the business logic classes).

A ViewModel is always created in association with a scope (a fragment or an activity) and will be retained as long as the scope is alive. E.g. if it is an Activity, until it is finished.

In other words, this means that a ViewModel will not be destroyed if its owner is destroyed for a configuration change (e.g. rotation). The new owner instance just re-connects to the existing model.

The purpose of the ViewModel is to acquire and keep the information that is necessary for an Activity or a Fragment. The Activity or the Fragment should be able to observe changes in the ViewModel. ViewModels usually expose this information via LiveData or Android Data Binding. You can also use any observability construct from your favorite framework.

1. **What is ViewModel Provider**

Creates ViewModelProvider. This will create ViewModels and retain them in a store of the given ViewModelStoreOwner.

**ViewModelProvider .get():** Returns an existing ViewModel or creates a new one in the scope (usually, a fragment or an activity), associated with this ViewModelProvider.

The created ViewModel is associated with the given scope and will be retained as long as the scope is alive (e.g. if it is an activity, until it is finished or process is killed).

Params:

modelClass - The class of the ViewModel to create an instance of it if it is not present.

Returns: A ViewModel that is an instance of the given type T.

Throws:

IllegalArgumentException - if the given modelClass is local or anonymous class.

1. **How to create instance of ViewModel in Activity & Fragment**

Default ViewModelProvider for an Activity or a Fragment can be obtained by passing it to the constructor: ViewModelProvider(myFragment)

//Default  
val *myVM* = ViewModelProvider(this).get(myViewModel::class.java)

//With Factory  
val *myVM* = ViewModelProvider(this,myVMProviderFectory(myRepository(retrofit ))) .get(myViewModel::class.java)

Note:

1. **How to getData or Observe ViewModel Data in Activity & Fragment**
2. **What is Repository Pattern**

**Repository** is a class which purpose is to provide a clean API for accessing data. What that means is that the Repository **can gather data from different data sources**(different REST APIs, cache, local database storage) **and** it **provides** **this** data **to** the rest of the **app**

1. **What is ViewModelProvider.Factory**

**ViewModelProvider.**Factory is an interface which have **create** method. The create method is responsible for creating our **VeiwModel's instance**

Factory a `Factory` which will be used to instantiate new `ViewModels`

1. **When ViewModel Instance get clear**

As we know, ViewModel is LifeCycle-aware component which respect/observer the owner/subscriber life cycle. When activity / fragment is going to destroy then onCleared() method will be call and here we can clear/clean whatever we want.

This method will be called when this ViewModel is no longer used and will be destroyed. It is useful when ViewModel observes some data and you need to clear this subscription to prevent a leak of this ViewModel.

**onCleared():** This method will be called when this ViewModel is no longer used and will be destroyed.

It is useful when ViewModel observes some data and you need to clear this subscription to prevent a leak of this ViewModel.

1. **Is viewmode instance will recreate incase of device screen oriantancetion / configuration get change**

As we know, ViewModel is LifeCycle-aware component which respect/observer the owner/subscriber / others app component life cycle.

1. When activity fragment instance get create then only viewModel instance will create by calling **onCreate()** method
2. When activity fragment get distoyred then only viewmodel instance get distorted by calling **onCleared()**

**Jetpack Component**

**Architecture Components could be classified as follows:**

* Room
* WorkManager.

Work Manager is Already Done above

* Lifecycle.
* ViewModel.
* LiveData.
* Navigation.
* Paging.
* Data Binding

**Room DataBase: ROOM DB**

Room Database is a part of the Android Architecture components which provides an abstraction layer over SQLite which allows for more robust database access while still providing the full power of SQLite.

**Why use Room Database?**

* Compile-time verification of SQL queries. each @Query and @Entity is checked at the compile time.
* Using Annotations to reduce the boilerplate code.
* Easily integrated with other Architecture components like LiveData, and RxJava.

**What is the difference between the room and SQLite database?**

* In the case of SQLite, There is no compile-time verification of raw SQLite queries. But in Room, there is SQL validation at compile time.
* As your schema changes, you need to update the affected SQL queries manually. Room solves this problem.
* You need to use lots of boilerplate code to convert between SQL queries and Java data objects. But, Room maps our database objects to Java Object without boilerplate code.
* Room is built to work with LiveData and RxJava for data observation, while SQLite does not.

**There Are Basically 3 Major Components In Room.**

1. **@Entity:** 
   1. The “@Entity” class represents an entity in a table.
   2. [Data entities](https://developer.android.com/training/data-storage/room/defining-data) that represent tables in your app's database.

@Entity(tableName = "user")  
data class Users(  
 @PrimaryKey(autoGenerate = true)  
 var userId: Int? = null,  
 val userName: String,  
 var location: String,  
 val email: String  
)

1. **@Dao — Data Access Object:** 
   1. [Data access objects (DAOs)](https://developer.android.com/training/data-storage/room/accessing-data) that provide methods that your app can use to query, update, insert, and delete data in the database.
   2. It is An Interface class where we put all our SQL queries.
   3. We don’t require to write whole queries now; we need to make a method and annotate with specific annotations like
   * @**Insert**: Used to insert a record into the Room database.
   * **@Delete**: Used to delete record from Room database.
   * **@Update**: Used to update record in Room Database.
   * **@Query:**  Used to enter the Query like (SELECT FROM\*)

@Dao  
interface UserDao {  
  
 @Insert  
 fun insertUser(users: Users)  
  
  
 @Insert(onConflict = OnConflictStrategy.REPLACE)  
 fun insertUsers(vararg users: User)  
  
  
 @Insert  
 fun insertBothUsers(user1: User, user2: User)  
  
  
 @Insert  
 fun insertUsersAndFriends(user: User, friends: List<User>)  
  
  
 @Query("Select \* from user")  
 fun gelAllUsers(): List<Users>  
  
 @Query("SELECT \* FROM user WHERE userId IN (:userIds)")  
 fun loadAllByIds(userIds: IntArray): List<Users>  
  
  
 @Query("SELECT \* FROM user WHERE userName LIKE :namex AND location LIKE :locx LIMIT 1”)  
 fun findByName(namex: String, locx: String): Users  
  
  
  
 @Query("SELECT \* FROM user WHERE first\_name LIKE :search " +  
 "OR last\_name LIKE :search")  
 fun findUserWithName(search: String): List<User>  
  
  
 @Query(  
 "SELECT \* FROM book " +  
 "INNER JOIN loan ON loan.book\_id = book.id " +  
 "INNER JOIN user ON user.id = loan.user\_id " +  
 "WHERE user.name LIKE :userName"  
 )  
 fun findBooksBorrowedByNameSync(userName: String): List<Book>  
  
  
  
  
 @Query("SELECT first\_name, last\_name FROM user")  
 fun loadFullName(): List<NameTuple>  
  
  
 @Update  
 fun updateUser(users: Users)  
  
 @Delete  
 fun deleteUser(users: Users)  
  
}

1. **@Database:** 
   1. The [database class](https://developer.android.com/reference/kotlin/androidx/room/Database) that holds the database and serves as the main access point for the underlying connection to your app's persisted data.
   2. This is an abstract class that extends **RoomDatabase**, this is where you define the entities (tables)and the version number of your database. It contains the database holder and serves as the main access point for the underlying connection.

@Database(entities = [Users::class], version = 1, exportSchema = false)  
@TypeConverters(Converters::class)  
abstract class AppDatabase : RoomDatabase() {  
  
 abstract fun userDao() : UserDao  
  
 companion object {  
 private var INSTANCE: AppDatabase? = null  
  
 fun getInstance(context: Context): AppDatabase? {  
 if (INSTANCE == null) {  
 synchronized(AppDatabase::class) {  
 INSTANCE = Room.databaseBuilder(context.applicationContext,  
 AppDatabase::class.java, "user.db").allowMainThreadQueries()  
 .build()  
 }  
 }  
 return INSTANCE  
 }  
  
 fun destroyInstance() {  
 INSTANCE = null  
 }  
 }  
}

**Things to notice here:**

* This is an abstract class that has to extend from RoomDatabase.
* It has to be annotated with @Database, it receives a list of entities with all the classes that compose the database (all these classes have to be annotated with @Entity). We also have to provide a database version.
* We have to declare an abstract function for each of the entities included in the @Database annotation, this function has to return the correspondentDAO (A class annotated with @Dao).
* Finally, we declare a companion object to get static access to the method getAppDataBase which gives us a singleton instance of the database.

**Type Converters**

Type Converters are used when we declare a property that Room and SQL don’t know how to serialize. Let’s see an example of how to serialize the List<String> data type.

class Converters {  
  
 @TypeConverter  
 fun fromString(value: String): List<String> {  
 val listType = object : TypeToken<List<String>>() {  
  
 }.type  
 return Gson().fromJson(value, listType)  
 }  
  
 @TypeConverter  
 fun fromArrayList(list: List<String>): String {  
 val gson = Gson()  
 return gson.toJson(list)  
 }  
}

**UserRepository.kt(Optional)**

class UserRepository(context: Context) {  
  
 var db: UserDao = AppDatabase.getInstance(context)?.userDao()!!  
  
  
 //Fetch All the Users  
 fun getAllUsers(): List<Users> {  
 return db.gelAllUsers()  
 }  
  
 // Insert new user  
 fun insertUser(users: Users) {  
 db..insertUser(users)  
//insertAsyncTask(db).execute(users)  
 }  
  
 // update user  
 fun updateUser(users: Users) {  
 db.updateUser(users)  
 }  
  
 // Delete user  
 fun deleteUser(users: Users) {  
 db.deleteUser(users)  
 }  
}

**Usage**

After you have defined the data entity, the DAO, and the database object, you can use the following code to create an instance of the database:

val db = Room.databaseBuilder(  
 applicationContext,  
 AppDatabase::class.java, "database-name"  
 ).build()

**Some other Point:**

* **autoGenerate**: If you need Room to assign automatic IDs to entity instances, set the [autoGenerate](https://developer.android.com/reference/kotlin/androidx/room/PrimaryKey#autogenerate) property of **@PrimaryKey** to **true**.
* @**PrimaryKey:** Each Room entity must define a [primary key](https://en.wikipedia.org/wiki/Primary_key) that uniquely identifies each row in the corresponding database table.
* **@Ignore:** By default, Room creates a column for each field that's defined in the entity. If an entity has fields that you don't want to persist, you can annotate them using [@Ignore](https://developer.android.com/reference/androidx/room/Ignore)

**Create a view**

To create a view, add the [@DatabaseView](https://developer.android.com/reference/androidx/room/DatabaseView) annotation to a class. Set the annotation's value to the query that the class should represent.

@DatabaseView("SELECT user.id, user.name, user.departmentId," +  
 "department.name AS departmentName FROM user " +  
 "INNER JOIN department ON user.departmentId = department.id")  
data class UserDetail(  
 val id: Long,  
 val name: String?,  
 val departmentId: Long,  
 val departmentName: String?  
)

@Database(entities = [User::class, Department::class], views =[UserDetail::class], version = 1)  
abstract class AppDatabase : RoomDatabase() {  
 abstract fun userDao(): UserDao  
}

**Write asynchrones DAO query**

1. **Write asynchronous one-shot queries using coroutine**

@Dao  
interface UserDao {  
 @Insert(onConflict = OnConflictStrategy.REPLACE)  
 suspend fun insertUsers(vararg users: User)  
  
 @Update  
 suspend fun updateUsers(vararg users: User)  
  
 @Delete  
 suspend fun deleteUsers(vararg users: User)  
  
 @Query("SELECT \* FROM user WHERE id = :id")  
 suspend fun loadUserById(id: Int): User  
  
 @Query("SELECT \* from user WHERE region IN (:regions)")  
 suspend fun loadUsersByRegion(regions: List<String>): List<User>  
}

1. **Write observable query using coroutine Flow**

@Dao  
interfaceUserDao {  
 @Query("SELECT \* FROM user WHERE id = :id")  
 fun loadUserById(id: Int): Flow<User>  
  
 @Query("SELECT \* from user WHERE region IN (:regions)")  
 fun loadUsersByRegion(regions: List<String>): Flow<List<User>>  
}

**Pre-Populate your Room database**

Sometimes, you might want your app to start with a database that is already loaded with a specific set of data. This is called *prepopulating* a database. In Room 2.2.0 and higher, you can use API methods to prepopulate a Room database at initialization with contents from a prepackaged database file in the device's file system.

1. **Prepopulate from an app asset:** To prepopulate a Room database from a prepackaged database file that is located anywhere in your app's assets/ directory, call the [createFromAsset()](https://developer.android.com/reference/kotlin/androidx/room/RoomDatabase.Builder#createfromasset) method from your RoomDatabase.Builder object before calling [build()](https://developer.android.com/reference/kotlin/androidx/room/RoomDatabase.Builder#build):

Room.databaseBuilder(appContext, AppDatabase.class, "Sample.db")

    .createFromAsset("database/myapp.db")

    .build()

**Note**: The createFromAsset() method accepts a string argument that contains a relative path from the assets/ directory to the prepackaged database file.

1. **Pre-populate from the file system:** To prepopulate a Room database from a prepackaged database file that is located anywhere in the device's file system *except* your app's assets/ directory, call the [createFromFile()](https://developer.android.com/reference/kotlin/androidx/room/RoomDatabase.Builder#createfromfile) method from your RoomDatabase.Builder object before calling [build()](https://developer.android.com/reference/kotlin/androidx/room/RoomDatabase.Builder#build):

**Migrating Room databases**

1. **Automated Migrations:** To declare an automated migration between two database versions, add an [@AutoMigration](https://developer.android.com/reference/kotlin/androidx/room/AutoMigration) annotation to the [autoMigrations](https://developer.android.com/reference/kotlin/androidx/room/Database#automigrations) property in [@Database](https://developer.android.com/reference/kotlin/androidx/room/Database):

@Database(version = 2,  
 entities = [User::class, UserDetails::class],  
 autoMigrations = [AutoMigration (from = 1, to = 2, spec = AppDatabase.MyAutoMigration::class)]  
 )  
abstract class AppDatabase : RoomDatabase() {  
   
 @RenameTable(fromTableName = "User", toTableName = "AppUser")  
 class MyAutoMigration : AutoMigrationSpec  
   
}

2. Manual Migration: In cases where a migration involves complex schema changes, Room might not be able to generate an appropriate migration path automatically. For example, if you decide to split the data in a table into two tables, Room is unable to tell how this split should be performed. In cases like these, you must manually define a migration path by implementing a [Migration](https://developer.android.com/reference/kotlin/androidx/room/migration/Migration) class.

val MIGRATION\_1\_2 = object : Migration(1, 2) {  
 override fun migrate(database: SupportSQLiteDatabase) {  
 database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER, `name` TEXT, " + "PRIMARY KEY(`id`))")  
 }  
}  
  
val MIGRATION\_2\_3 = object : Migration(2, 3) {  
 override fun migrate(database: SupportSQLiteDatabase) {  
 database.execSQL("ALTER TABLE Book ADD COLUMN pub\_year INTEGER")  
 }  
}  
  
Room.databaseBuilder(applicationContext, MyDb::class.java, "database-name").addMigrations(MIGRATION\_1\_2, MIGRATION\_2\_3).build()

**Parcelalization**

Parcelable is an Android-specific interface that allows objects to be serialized and deserialized efficiently, especially when they need to be passed between different components within an Android application, such as between activities or fragments.

In Kotlin, It’s very simpale to implement parcelazition in Android by Adding **Extention Plugin** in Gradle file

plugins {  
 id 'kotlin-parcelize'  
}

With Parcelize, you can annotate your Kotlin data classes with the **@Parcelize annotation**, and the Kotlin Android **Extensions plugin** will **automatically generate the Parcelable implementation** for you. This simplifies the process and reduces the amount of boilerplate code that needs to be written.

**Example**:

@Parcelize  
data class Person(val name: String, val age: Int) : Parcelable

By using **@Parcelize**, the **plugin will generate** the necessary **writeToParcel() and createFromParcel()** methods required for Parcelable implementation. You can then pass objects of this class between components using **intents or bundles** without having to write additional serialization or deserialization code.

val *person* = Person("John Doe", 25)  
val *intent* = Intent(this, ReceivingActivity::class.java)  
intent.putExtra("person", person)  
startActivity(intent)

val *receivedPerson* = intent.getParcelableExtra<Person>("person")  
if (receivedPerson != null) {  
 // Do something with the receivedPerson object  
 val name = receivedPerson.name  
 val age = receivedPerson.age  
 // ...  
}

**Corouting & Thread**

1. **What is Corouting**
2. **Type of Corouting**
3. **Type of Corouting builder**
4. **What is Dispatcher & Type of Dispatcher**
5. **What is suspended function**

**MVVM**

1. **What is MVVM**

**MVP**

**MVC**

**SOLID Principal**

1. **What is SOLID Principal**
2. **Explain each in details with Example**

**DI – Dependency Injection**

**Networking / API**

1. **is async task deprecated?**

Google is deprecating Android AsyncTask API in Android 11

1. **What is OKHTTP**

OkHttp is a popular open-source Java library for making HTTP requests and managing network communication in Android and Java applications. OkHttp is built on top of the Java's built-in HttpURLConnection and Apache HttpClient libraries, providing a simpler and more efficient API for working with HTTP.

**Benefits with OKHTTP**

**Easy to Use:** OkHttp provides a high-level API that simplifies the process of making HTTP requests.

**Efficient Network Calls:** It uses connection pooling and transparent response caching, which helps reduce latency and minimize redundant network requests.

**Connection Reuse:** OkHttp supports connection reuse, which means it can reuse established connections for subsequent requests to the same server, reducing the overhead of establishing a new connection for each request.

**Interceptors:** OkHttp provides interceptors, as discussed earlier, which allow you to intercept and modify network requests and responses. This feature offers flexibility in adding custom behavior, authentication, caching, and more to your network calls.

**Security Features**: OkHttp handles secure connections seamlessly. It supports HTTPS by default, verifies SSL certificates, and provides options for customizing SSL/TLS settings and certificate pinning.

**Integration with Retrofit:** OkHttp is the recommended HTTP client for Retrofit, a popular library for building type-safe HTTP clients in Android and Java. The integration between OkHttp and Retrofit provides a powerful and convenient way to handle network requests and responses.

Ex:

// Create an instance of OkHttpClient  
val *client* = OkHttpClient()  
  
// Create a request object with the URL  
val request = Request.Builder()  
 .url("https://jsonplaceholder.typicode.com/posts")  
 .build()  
  
try {  
 // Execute the request and get the response  
 val response: Response = client.newCall(request).execute()  
  
 // Check if the request was successful  
 if (response.isSuccessful) {  
 val responseBody = response.body?.string()  
 println(responseBody)  
 } else {  
 println("Request was not successful: ${response.code}")  
 }  
} catch (e: IOException) {  
 println("Error occurred: ${e.message}")  
}

1. **What is Retrofit**

It is a popular open-source library that simplifies the process of making network requests in your Android applications. Retrofit is built on top of OkHttp and uses annotations to define API endpoints, request parameters, and response types.

**Benefits**:

**Simplified Networking:** It provides a clean and concise way to define API endpoints and their expected responses.

**Easy API Integration:** Retrofit simplifies the integration process By using annotations, you can map API endpoints, HTTP methods, request headers, query parameters, and response types directly to your Kotlin classes and methods.

**Type-Safe Requests and Responses:** Retrofit leverages Kotlin's type system to ensure type safety during network requests and responses. You can define the expected data structures as Kotlin classes, and Retrofit will handle the deserialization of responses into those classes.

**Request Customization:** Retrofit allows you to customize requests by adding headers, query parameters, form data, or multipart data easily. You can also intercept and modify requests or responses by using OkHttp's interceptors.

**Easy Error Handling:** Retrofit simplifies error handling by providing mechanisms to define custom error models or handling specific HTTP error codes. It supports error conversion to meaningful Kotlin objects or exceptions for easier error handling in your app.

Ex:

interface MyApiService {  
 @GET("users/{userId}")  
 suspend fun getUser(@Path("userId") userId: String): Response<User>  
}

object RetrofitInstance{  
  
 val retrofit = Retrofit.Builder()  
 .baseUrl("https://api.example.com/")  
 .addConverterFactory(GsonConverterFactory.create())  
 .build()  
  
 fun getInstance():Retrofit{  
 return retrofit.create(MyApiService::class.java)  
 }  
}

CoroutineScope(Dispatchers.IO).launch {  
 try {  
 val response = RetrofitInstance.getInstance().getUser("123")  
 if (response.isSuccessful) {  
 val user = response.body()  
 // Handle the user data  
 } else {  
 // Handle the error  
 }  
 } catch (e: Exception) {  
 // Handle network or other exceptions  
 }  
}

**Note**: In this example, Retrofit is used to define an API service with a single endpoint to get a user's data. The suspend keyword indicates that the method can be used with coroutines. Retrofit handles the network request and deserializes the response into a User object using GsonConverterFactory.

1. **What is Interceptor**

In Retrofit, "interceptors" are components that can intercept and modify the requests and responses made by the network client. They provide a mechanism to add custom behavior or make modifications to the network traffic before it is sent or received.

Retrofit uses the OkHttp library as its underlying HTTP client, and OkHttp provides a feature called "interceptors." These interceptors can be added to the OkHttpClient instance used by Retrofit to intercept and process the network requests and responses

// Import the necessary libraries  
import okhttp3.Interceptor  
import okhttp3.Response  
  
// Create a RequestInterceptor class that implements Interceptor  
class MyRequestInterceptor : Interceptor {  
 override fun intercept(chain: Interceptor.Chain): Response {  
 // Get the original request from the chain  
 val originalRequest = chain.request()  
  
 // Create a modified request by adding headers or making changes  
 val modifiedRequest = originalRequest.newBuilder()  
 .addHeader("Authorization", "Bearer my\_token")  
 .build()  
  
 // Proceed with the modified request  
 return chain.proceed(modifiedRequest)  
 }  
}

// Create an instance of OkHttpClient with the request interceptor  
val client = OkHttpClient.Builder()  
 .addInterceptor(MyRequestInterceptor())

.addConverterFactory(GsonConverterFactory.create())  
 .build()

// Create a Retrofit instance using the configured OkHttpClient  
val *retrofit* = Retrofit.Builder()  
 .baseUrl("https://api.example.com/")  
 .client(*client*)  
 .build()

// Create your API interface and use it with Retrofit  
interface MyApiService {  
 // Define your API endpoints  
 // ...  
}  
  
val *apiService* = *retrofit*.create(MyApiService::class.*java*)

1. **What is addConverterFactory**

The addConverterFactory(GsonConverterFactory.create()) method in Retrofit is used to specify the converter factory that will be used to convert the response data from the server into the desired format or object representation.

By adding GsonConverterFactory.create() as the converter factory, Retrofit will automatically parse the response body returned by the server, which is typically in JSON format, into Java objects based on their defined data structures.

Ex:

val *retrofit* = Retrofit.Builder()  
 .baseUrl("https://api.example.com/")  
 .addConverterFactory(GsonConverterFactory.create())  
 .build()

1. **How to shaing / SSL Pinning / Certificate**

Hashing is a general cryptographic technique used to transform data into a fixed-length string of characters, called a hash value or hash code. It is commonly used for various purposes, such as data integrity checks, password storage, and verifying the authenticity of data.

However, when it comes to Retrofit and OkHttp, hashing can be utilized in combination with other techniques, such as SSL certificate pinning, to enhance security during network communication. In this context, hashing is used to calculate a hash value from a server's SSL certificate and compare it with a pre-defined value to ensure the authenticity of the certificate.

Here's a general outline of how hashing can be used in combination with Retrofit and OkHttp for SSL certificate pinning:

1. Obtain / Get the SSL certificate of the server you are communicating with.
2. Visit the server's website in a web browser.
3. View the certificate information (usually found in the browser's security settings).
4. Export the certificate as a DER-encoded binary X.509 file (e.g., "certificate.cer").

2. Use a tool like OpenSSL to calculate the hash value (usually SHA-256) of the certificate:

openssl x509 -inform der -in certificate.cer -noout -sha256 -fingerprint  
  
--------------------------OR ---------------------------  
  
openssl x509 -in certificate.pem -noout -sha256 -fingerprint

3. Take the calculated hash value and add it to your application as a trusted value or store it securely. OR Copy the calculated hash value.

4. Implement certificate pinning in OkHttp by creating an instance of CertificatePinner and providing the server's hostname and the trusted hash value:

// Import the necessary libraries  
import okhttp3.CertificatePinner  
import okhttp3.OkHttpClient  
  
// Create a CertificatePinner object with the server's hostname and the hash value  
val *certificatePinner* = CertificatePinner.Builder()  
 .add("example.com", "sha256/<your\_certificate\_hash>")  
 .build()  
  
  
  
// Create an OkHttpClient instance with the CertificatePinner  
val *client* = OkHttpClient.Builder()  
 .certificatePinner(*certificatePinner*)  
 .build()  
  
  
  
// Import the necessary libraries  
import retrofit2.Retrofit  
import retrofit2.converter.gson.GsonConverterFactory  
  
// Create a Retrofit instance with your base URL and the OkHttpClient  
val *retrofit* = Retrofit.Builder()  
 .baseUrl("https://example.com/")  
 .client(*client*)  
 .addConverterFactory(GsonConverterFactory.create())  
 .build()

**Note:** Replace "example.com" with the actual hostname of the server you are communicating with and "<your\_certificate\_hash>" with the trusted hash value you obtained in step 2.

Certificate pinning using hashing allows your app to validate the server's SSL certificate by comparing the calculated hash with the trusted value. If the hash values match, it ensures that the server's certificate is authentic and prevents Man-in-the-Middle (MITM) attacks.

1. **What is Different B/W SOAP and REST**

|  |  |
| --- | --- |
| **SOAP API** | **REST API** |
| Relies on SOAP (Simple Object Access Protocol) | Relies on REST (Representational State Transfer) architecture using HTTP. |
| Transports data in standard XML format. | Generally transports data in JSON. It is based on URI. Because REST follows stateless model, REST does not enforces message format as XML or JSON etc. |
| Because it is XML based and relies on SOAP, it works with WSDL | It works with GET, POST, PUT, DELETE |
| Works over HTTP, HTTPS, SMTP, XMPP | Works over HTTP and HTTPS |
| Highly structured/typed | Less structured -> less bulky data |
| Designed with large enterprise applications in mind | Designed with mobile devices in mind |

**Logical / Descriptive QA**