# Introduce

Let me introduce my background and my experience in Android.

As you can see in my background, I have more than 7 yrs of experience in Android development based object-oriented programming languages and worked for several companies for building diverse Android applications.

I handled various responsibilities in my previous Android Developer position and quickly established talents in prioritizing tasks, meeting deadlines and finding solutions to eliminate obstacles.

For the past years, I have built several useful and popular Android apps that includes video streaming app, shopping app and so on and also uploaded them on Google Play store.

Those apps are popular and have more than millions of users.

During the building several kinds of, several types of Android applications, I have implemented tons of new features, optimized their performance, improved user interface and experience, and also shared my knowledge for collaborating other team members.

# About Disney

While working on Upwork, I met one client who's working for Disney as Senior developer. He had some problems in his task and was looking for the solution by help of Upwork talents and it looked like my solution inspired him. After that he suggested me to join in his team and I applied to the Disney having simple process by his reference.

--------------------------------

1. I collaborated with cross-functional teams to define, design, implement, test, and deploy new features in a fast paced environment.

2. I worked with internal and external data sources and API's.

3. I wrote unit tests in order to increase reliability and quality of applications.

4. I addressed system defects and implemented enhancements to existing functionality.

5. I troubleshot issues with minimal guidance, identified bottlenecks in existing workflows and provided solutions for a scalable, defect-free application.

6. I kept up on industry trends and current technological standards, languages, coding techniques, utilities and operational considerations.

7. I made suggestions for process, coding, implementation, and performance improvements.

-------------------------------

At Walt Disney, I started with the role of Junior android developer. About two years later, feeling my skills grown up I was wanted to be promoted. But they have many of senior developers so that I couldn't get promoted. That's why I decided to quit Walt Disney and looked for new job position.

# About NASurfaces

The manager of NASurfaces was one of my client who was impressed by my talent as well. While thinking of new job at Walt Disney, I got the invitation from him to take care of their software development.

-------------------------------

1. I designed and built the company website, ios/android application.

2. I kept high performance, quality, and responsiveness of iOS and Android application.

3. I reviewed business requirements, wire frames, and designs for technical feasibility, estimated build timelines, and followed established software development best practices.

4. I evaluated and reviewed design frameworks and methodologies in order to achieve functional and non-functional requirements and conformance to architecture plans.

5. I did Unit-test code for robustness, usability, and general reliability.

6. I identified solutions and implemented bug fixes to improve application performance and usability.

--------------------------------

When I started NASurface, It was the Full-Time role and for several months I finished all of their software platforms and deployed on live server. So there had been no more work for me but maintenance. So under the discussion, my contract level had been dropped down to freelancer level that when they meet bugs or upgrade feature, they assign to me on hourly pay.

# Motive

1. I’ve been honing my android development skills for a few years now and, first and foremost, I’m looking for a position where I can continue to exercise those skills.

2. Another thing that’s important to me is that the position allows me to not only write the code, but also present my findings and suggestions directly to the team members. That would be really refreshing! I’m always very motivated by being able to see the impact of my work on other people.

3. And, I’m definitely looking for a position where I can grow—professional development is something that’s really important to me since I hope to take on managerial responsibilities in the future.

4. To sum it up, I’d love a position where I can use my skills to make an impact that I can see with my own eyes. Of course, the position is only part of the equation. Being at a company where I can grow and work toward something I care about matters, too.

# App Link

I have developed about 8 apps but I'm not sure they are still alive on google play store. But I have an amazing app that I worked on while working for Walt Disney which is very famous on the appstore having more than 50 Millions of Users and that is covering most of my skills.

https://play.google.com/store/apps/details?id=me.tidesnear.free&hl=en

https://play.google.com/store/apps/details?id=com.espn.score\_center&hl=en&gl=US

# Low Salary Reason

# When you find it difficult to work with someone, how did you handle the situation?

I had to try to find a way to communicate which is less time consuming.

Before communication with someone, I spent some time thoroughly reading and understanding the work details and write down what I think the problem is.

And then I spent some time looking at the code and thinking up a proper solution. And I wrote down in a new paragraph what I believe the ideal solution would be. And then I communicated on both paragraphs of me outlining the problem and the solution. And then I asked if he agrees with my solution, and took his feedback if any and modified my solution if need be. And then implemented the solution and only after all of that, submitted my work done.

So:

1. Thoroughly read and understand the work details (on my own)

2. Write down the problem in my own words

3. Spend time reading the code and thinking of a solution

4. Write down the solution

5. Communicate with someone on my understanding of the problem / solution

6. Incorporate any feedback and implement the solution

7. Finally submit my work

# troubleshooting

If an Android application is crashing frequently, you can follow the below-given techniques:

1. Compatibility Check:

It is not possible to test an application for all kinds of devices and operating systems. There might be a possibility that an application is not compatible with your OS.

2. Memory Management:

2-1. Some apps run perfectly on one mobile device but might crash on other devices. This is where processing power, memory management, and CPU speed are considered.

2-2. As there is a limited amount of memory space on mobile devices, you can free up memory space for the application to function properly.

2-3. If an application is frequently crashing, you can delete the application’s data, which will clear its cache memory and allow some free space on your device and might boost the app’s performance.

# how to prevent memory leaks?

1. Use Application Context rather than Activity Context because Activities are more likely to be leaked.

2. Avoid long-lived references to Activities i.e.AsyncTask.

3. Avoid non-static inner classes in Activities. Use static inner classes with weak references so they can't be Garbge Collected when they are not used.

4. Use LeakCanary - A memory leak detection library for Android.

# !! double-bang

The not-null assertion operator !! converts any value to a non-null type and throws a KotlinNullPointerException exception if the value is null.

Consider:

fun main(args: Array<String>) {

var email: String?

email = null

println(email!!)

}

This operator should be used in cases where the developer is guaranteeing – it allows you to be 100% sure that its value is not null.

# !! vs ?. (not null assertion operator vs safe call operator)

!! is used to force unwrap the nullable type to get the value. If the value returned is a null, it would lead to a runtime crash. Hence a !! operator should be only used when you’re absolutely sure that the value won’t be null at all. Otherwise, you’ll get the dreaded null pointer exception. On the other hand, a ?. is an Elvis Operator that does a safe call. We can use the lambda expression let on the nullable value to unwrap safely as shown below.

# !!(null check) vs ?.(safe calls)

The safe call operator i.e. ?. is used to check if the variable's value is null or not. If it is null, then null will be returned otherwise it will return the desired value.

If you want to throw NullPointerException when the variable's value is null, you can use the null check or !! Operator.

# ""object"" kotlin

To use the singleton pattern for our class we must use the keyword `object`

An `object` cannot have a constructor set. We can use the init block inside it though.

# ?.(safe calls) vs !!(null check)

The safe call operator i.e. ?. is used to check if the variable's value is null or not. If it is null, then null will be returned otherwise it will return the desired value.

If you want to throw NullPointerException when the variable's value is null, you can use the null check or !! Operator.

# @Bindable

One way to improve performance when using data binding is to use the @Bindable annotation. This annotation tells the data binding system to only update the UI when the data has actually changed. Without this annotation, the data binding system will update the UI every time any data is changed, even if the UI doesn’t need to be updated.

# @JvmField

@JvmField: To access the fields of a Kotlin class from Java code without using any getters and setters, we need to use the @JvmField in the Kotlin code.

# @JvmOverloads

@JvmOverloads: To use the default values passed as an argument in Kotlin code from the Java code, we need to use the @JvmOverloads annotation.

# @JvmStatic

@JvmStatic: This annotation is used to tell the compiler that the method is a static method and can be used in Java code.

# @JvmStatic, @JvmOverloads, and @JvmFiled

1. @JvmStatic: This annotation is used to tell the compiler that the method is a static method and can be used in Java code.

2. @JvmOverloads: To use the default values passed as an argument in Kotlin code from the Java code, we need to use the @JvmOverloads annotation.

3. @JvmField: To access the fields of a Kotlin class from Java code without using any getters and setters, we need to use the @JvmField in the Kotlin code.

# == vs ===

== is used to compare the values are equal or not. === is used to check if the references are equal or not.

# AbsoluteLayout, LinearLayout, RelativeLayout

LinearLayout arranges its children in a single row or single column one after the other.

RelativeLayout arranges it's children in positions relative to each other or relative to parent depending upon the LayoutParams defined for each view.

AbsoluteLayout needs the exact positions of the x and y coordinates of the view to position it. Though this is deprecated now.

# abstract class

A class that is declared as abstract is known as an abstract class. It needs to be extended and its method implemented. It cannot be instantiated. It can have abstract methods, non-abstract methods, constructors, and static methods. It can also have the final methods which will force the subclass not to change the body of the method.

Can there be an abstract method without an abstract class?

No, if there is an abstract method in a class, that class must be abstract.

Can you use abstract and final both with a method?

No, because we need to override the abstract method to provide its implementation, whereas we can't override the final method.

Can you use abstract and final both with a method?

No, because we need to override the abstract method to provide its implementation, whereas we can't override the final method.

Is it possible to instantiate the abstract class?

No, the abstract class can never be instantiated even if it contains a constructor and all of its methods are implemented.

# abstract class static variable

Yes, we can declare static variables and methods in an abstract method. As we know that there is no requirement to make the object to access the static context, therefore, we can access the static context declared inside the abstract class by using the name of the abstract class.

# abstract class vs interface

1. An abstract class can have a method body (non-abstract methods).

1-1. The interface has only abstract methods.

2. An abstract class can have instance variables.

2-1. An interface cannot have instance variables.

3. An abstract class can have the constructor.

3-1. The interface cannot have the constructor.

4. An abstract class can have static methods.

4-1. The interface cannot have static methods.

5. You can extend one abstract class.

5-1. You can implement multiple interfaces.

6. The abstract class can provide the implementation of the interface.

6-1. The Interface can't provide the implementation of the abstract class.

7. The abstract keyword is used to declare an abstract class.

7-1. The interface keyword is used to declare an interface.

8. An abstract class can extend another Java class and implement multiple Java interfaces.

8-1. An interface can extend another Java interface only.

9. An abstract class can be extended using keyword extends

9-1. An interface class can be implemented using keyword implements

10. A Java abstract class can have class members like private, protected, etc.

10-1. Members of a Java interface are public by default.

# abstract method static

In Java, if we make the abstract methods static, It will become the part of the class, and we can directly call it which is unnecessary. Calling an undefined method is completely useless therefore it is not allowed.

# abstract method vs final method

The main difference between the final method and abstract method is that the abstract method cannot be final as we need to override them in the subclass to give its definition.

# abstraction

Abstraction is a process of hiding the implementation details and showing only functionality to the user. It displays just the essential things to the user and hides the internal information, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery. Abstraction enables you to focus on what the object does instead of how it does it. Abstraction lets you focus on what the object does instead of how it does it.

In Java, there are two ways to achieve the abstraction.

1. Abstract Class

2. Interface

# abstraction vs encapsulation

Abstraction hides the implementation details whereas encapsulation wraps code and data into a single unit.

# access specifiers

1. Public The classes, methods, or variables which are defined as public, can be accessed by any class or method.

2. Protected Protected can be accessed by the class of the same package, or by the sub-class of this class, or within the same class.

3. Default Default are accessible within the package only. By default, all the classes, methods, and variables are of default scope.

4. Private The private class, methods, or variables defined as private can be accessed within the class only.

# action android

In Android development, an action is what the intent sender wants to do or expected to get as a response. Most application functionality is based on the intended action.

# ActionBarActivity and Activity, AppCompatActivity, FragmentActivity

Activity is the base class.

FragmentActivity extends Activity.

AppCompatActivity extends FragmentActivity.

ActionBarActivity extends AppCompatActivity.

FragmentActivity is used for fragments.

Since the build version 22.1.0 of the support library, ActionBarActivity is deprecated.

It was the base class of appcompat-v7.

At present, AppCompatActivity is the base class of the support library.

It has come up with many new features like ToolBar, tinted widgets, material design color pallets etc.

# Activity

Activity in java is a single screen that represents GUI(Graphical User Interface) with which users can interact in order to do something like dial the phone, view email, etc.

# Activity AsyncTask lifecycle

An AsyncTask is not tied to the life cycle of the Activity that contains it. So, for example, if you start an AsyncTask inside an Activity and the user rotates the device, the Activity will be destroyed (and a new Activity instance will be created) but the AsyncTask will not die but instead goes on living until it completes.

Then, when the AsyncTask does complete, rather than updating the UI of the new Activity, it updates the former instance of the Activity (i.e., the one in which it was created but that is not displayed anymore!). This can lead to an Exception (of the type java.lang.IllegalArgumentException: View not attached to window manager if you use, for instance, findViewById to retrieve a view inside the Activity).

There’s also the potential for this to result in a memory leak since the AsyncTask maintains a reference to the Activty, which prevents the Activity from being garbage collected as long as the AsyncTask remains alive.

For these reasons, using AsyncTasks for long-running background tasks is generally a bad idea . Rather, for long-running background tasks, a different mechanism (such as a service) should be employed.

# activity- clear back stack by intent

1. The first approach is to use a `FLAG\_ACTIVITY\_CLEAR\_TOP` flag.

Intent intent= new Intent(ActivityA.this, ActivityB.class);

intent.setFlags(Intent.FLAG\_ACTIVITY\_CLEAR\_TOP);

startActivity(intent);

finish();

2.The second way is by using `FLAG\_ACTIVITY\_CLEAR\_TASK` and `FLAG\_ACTIVITY\_NEW\_TASK` in conjunction.

Intent intent= new Intent(ActivityA.this, ActivityB.class);

intent.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK | Intent.FLAG\_ACTIVITY\_CLEAR\_TASK);

startActivity(intent);

# activity critical loops

Loop 1, Entire Lifetime: The activity happens between onCreate and onDestroy.

Loop 2, Visible Lifetime: The activity happens between onStart and onStop

Loop 3, Foreground Lifetime: The activity happens between onResume and onPause

# Activity kill

We'll declare and assign a class instance of the FirstActivity to itself as shown below.

public class FirstActivity extends AppCompatActivity {

public static FirstActivity firstActivity;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

firstActivity=this;

}

}

We'll call finish() on the above instance of the FirstActivity to kill the activity from any other activity.

FirstActivity.firstActivity.finish()

# Activity Lifecycle

1. OnCreate(): It is called when activity is created. Using this, the views are created and data is collected from bundles.

2. OnStart(): It is called if the activity is becoming visible to the user. It may be succeeded by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden.

3. OnResume(): It is called when the activity will start an interaction with the user.

4. OnPause(): This is called when the activity is moving to the background but hasn’t been killed yet.

5. OnStop(): This is called when an activity is no longer visible to the user.

6. OnDestroy(): This is called when the activity is finished or destroyed.

7. OnRestart(): This is called after the activity has been stopped, prior to it being started again.

https://s3.ap-south-1.amazonaws.com/myinterviewtrainer-domestic/public\_assets/assets/000/000/441/original/Life\_Cycle\_of\_Android.png

Differentiate:

onCreate() is the first method that’s invoked when an activity is launched for the first time. onStart() is invoked after onCreate() has completed it’s task. onResume() is called after onStart() has completed. When an activity leaves its foreground (probably for a smaller duration such as standby/sleep) onPause() is invoked followed by onStop()(when the activity is not visible. eg. some other application is launched). onDestroy() is called when the activity or application is killed.

Essentially the lifecycle methods are divided into three layers of duration :

1. onCreate() and onDestroy() are present during the entire duration of the activity

2. onStart() and onStop() are present while the activity is visible

3. onResume() and onPause() are present while the activity is in foreground

# activity lifetime

1. Entire lifetime – activity happens between onCreate and onDestroy

2. Visible lifetime – activity happens between onStart and onStop

3. Foreground lifetime – activity happens between onResume and onPause

# activity process states

1. foreground activity

2. visible activity

3. background activity

4. empty process

# activity restart

public class MainActivity extends AppCompatActivity {

public void restartActivity() {

MainActivity.this.recreate();

}

}

# activity- screen rotate

When the screen is rotated, the current instance of the activity is destroyed a new instance of the Activity is created in the new orientation. The onRestart() method is invoked first when a screen is rotated. The other lifecycle methods get invoked in the similar flow as they were when the activity was first created.

How to prevent the data from reloading and resetting when the screen is rotated?

The most basic approach is to add an element attribute tag `android:configChanges` inside the activity tag in the AndroidManifest.xml as shown below.

<activity android:name="".MainActivity""

android:configChanges=""orientation|screenSize"">

<intent-filter>

<action android:name=""android.intent.action.MAIN"" />

<category android:name=""android.intent.category.LAUNCHER"" />

</intent-filter>

</activity>

In general, the configChanges for any activity are defined as

android:configChanges=""orientation|screenSize|keyboardHidden""

The `keyboardHidden` configuration is to prevent the keyboard from resetting if it's pulled out.

`android:configChanges` is not the recommended way by Google. Though it's the simplest way to use, it comes with its own share of drawbacks. First, the common perception that android:configChanges = ""orientation"" will magically retain the data is a complete misinterpretation. The orientation changes can occur from a number of other events such as changing the default language can trigger a configuration change and destroy and recreate the activity. Second, the activity can restart itself if it's in the background and Android decides to free up its heap memory by killing it. When the application returns to the foreground it'll restart it's data to the original state and the user may not like that. A better alternative of `android:configChanges` is; Saving the current state of the activity when it's being destroyed and restoring the valuable data when it's restarted can be done by overriding the methods `onSaveInstanceState()` and `onRestoreInstanceState()` of the activity class.

# activity states

1. Active state: The activity is in the foreground

2. Paused state: The activity is in the background and visible

3. Stopped state: The activity is in the background but not visible, even hidden or obscuring other activities

4. Destroyed state: The activity is completely terminated or killed/removed

# Activity vs AppCompatActivity

AppCompatActivity provides native ActionBar support that is consistent across the application.

Also, it provides backward compatibility for other material design components till SDK version 7(ActionBar was natively available since SDK 11).

Extending an Activity doesn’t provide any of these.

Note: Since SDK 21 every activity by default, extends AppCompatActivity.

# activity vs fragment

1. Activity is an application component that gives a user interface where the user can interact.

1-1.The fragment is only part of an activity, it basically contributes its UI to that activity.

2. Activity is not dependent on fragment.

2-1. Fragment is dependent on activity. It can’t exist independently.

3. we need to mention all activity it in the manifest.xml file.

3-1. Fragment is not required to mention in the manifest file

4. We can’t create multi-screen UI without using fragment in an activity.

4-1. After using multiple fragments in a single activity, we can create a multi-screen UI.

5. Activity can exist without a Fragment.

5-1. Fragment cannot be used without an Activity.

6. Creating a project using only Activity then it’s difficult to manage.

6-1. While Using fragments in the project, the project structure will be good and we can handle it easily.

7. Lifecycle methods are hosted by the OS. The activity has its own life cycle.

7-1. Lifecycle methods in fragments are hosted by hosting the activity.

8. Activity is not light weight.

8-1. The fragment is the lite weight.

# Activity vs Service

1. Activity is designed to run in the foreground. / Service is mainly designed to run in the background. Foreground services are also available.

2. Activity is used when the user interface is necessary. / Service is used when the user interface is not necessary.

3. Activity is dependent. / Service act independently.

# Activity, AppCompatActivity, FragmentActivity and ActionBarActivity

Activity is the base class.

FragmentActivity extends Activity.

AppCompatActivity extends FragmentActivity.

ActionBarActivity extends AppCompatActivity.

FragmentActivity is used for fragments.

Since the build version 22.1.0 of the support library, ActionBarActivity is deprecated.

It was the base class of appcompat-v7.

At present, AppCompatActivity is the base class of the support library.

It has come up with many new features like ToolBar, tinted widgets, material design color pallets etc.

# activityCreator

An activityCreator is the first step towards the creation of a new Android project. It is made up of a shell script that will be used to create new file system structure necessary for writing codes within the Android IDE.

# Adapter

An adapter in Android acts as a bridge between an AdapterView and the underlying data for that view. The adapter holds the data and sends the data to the adapter view, the view can take the data from the adapter view and shows the data on different views like a spinner, list view, grid view, etc.

# Adb

Adb is short for Android Debug Bridge. It allows developers the power to execute remote shell commands. Its basic function is to allow and control communication towards and from the emulator port.

# addFlags() vs setFlags()

When we're using setFlags, we're replacing the old flags with a new set of Flags. When we use addFlags, we're appending more flags.

# aggregation

Aggregation can be defined as the relationship between two classes where the aggregate class contains a reference to the class it owns. Aggregation is best described as a has-a relationship. For example, The aggregate class Employee having various fields such as age, name, and salary also contains an object of Address class having various fields such as Address-Line 1, City, State, and pin-code. In other words, we can say that Employee (class) has an object of Address class.

# aggregation vs composition

Aggregation represents the weak relationship whereas composition represents the strong relationship. For example, the bike has an indicator (aggregation), but the bike has an engine (composition).

# AIDL

On Android what happens with the processors is one process can’t normally access the memory of another process. So in order to interact they need to decompose their objects into primitives that the OS can understand and gather the objects across that boundary. The code to do that gathering is very complex to write, so Android handles it with AIDL. So generally AIDL is similar to the other IDLs and it allows to define the programming interface that both the client and service agree upon in order to interact with each other using interprocess communication (IPC).

# AIDL create service

1. create the .aidl file, which defines the programming interface

2. implement the interface, which involves extending the inner abstract Stub class as well as implanting its methods.

3. expose the interface, which involves implementing the service to the clients.

# AIDL data types

-string

-charSequence

-List

-Map

-all native Java data types like int,long, char and Boolean

# also:-

Context object: it

Return value: context object

It's used when we need to do additional operations after the object members have been initialised.

# Android

Android is an open-sourced operating system that is used on mobile devices, such as mobiles and tablets. The Android application executes within its own process and its own instance of Dalvik Virtual Machine(DVM) or Android RunTime(ART).

# Android application Architecture

1. Services: Used to perform background functionalities.

2. Intent: Used to perform the interconnection between activities and the data passing mechanism.

3. Resource Externalization: strings and graphics.

4. Notification: light, sound, icon, notification, dialog box and toast.

5. Content Providers: It will share the data between applications.

# android application Widgets

Application widgets are miniature application views that can be embedded in other applications and receive periodic updates.

# android- database

SQLite is the open-source relational database used in Android. The SQLite engine is serverless, transactional, and also self-contained. Instead of the client-server relationship of most database management systems, the SQLite engine is integrally linked with the application. The library can be called dynamically and it can make use of simple function calls that reduce latency in database access.

# Android SDK

The Google Android SDK is a toolset which is used by developers to write apps on Android-enabled devices. It contains a graphical interface that emulates an Android-driven handheld environment and allows them to test and debug their codes.

# Android- storage

1. Shared Preferences

2.Internal Storage

3. External Storage

4. SQLite Databases

5. Network Connection

# Android Support Library

The android platform supports a wide variety of the versions and devices to choose from. With the release of every new version, new Android APIs are added and evolved. To make these new Android APIs available to users on older devices the Android Support Library was designed. Android Support Library provides developers with newer APIs that are compatible on older framework releases.

1. Compatibility Libraries: These focus on back porting features so that older frameworks can take advantage of newer releases. The major libraries include v4 and v7-appcompat. v4 includes classes like DrawerLayout and ViewPager while appcompat-v7 provides classes for support ActionBar and ToolBar.

2. Component Libraries: These include libraries of certain modules that don’t depend on other support library dependencies. They can be easily added or removed. Examples include v7-recyclerview and v7-cardview.

3. Miscellaneous libraries: The Android support libraries consists of few other libraries such as v8 which provides support for RenderScript, annotations for supporting annotations like @NonNull.

# android:launchMode

The `android:launchMode` of an Activity can be of the following types:

- \*\*standard\*\* : It's the default launch mode for an activity wherein every new instance of the activity called will be put on top of the stack as a separate entity. Hence calling startActivity() for a particular class 10 times will create 10 activities in the task list.

- \*\*singleTop\*\*: It differs from the standard launch mode in the fact that when the Activity instance that's invoked is already present on the top of the stack, instead of creating a new Activity, that instance will be called. In cases where the same Activity instance is not on the top of the stack or if it doesn't exist in the stack at all then a new instance of the activity will be added to the stack. Hence we need to handle the upcoming intent in both the `onCreate()` and `onNewIntent()` methods to cover all cases.

- \*\*singleTask\*\*: This is different from singleTop in the case that if the Activity instance is present in the stack, the onNewIntent() would be invoked and that instance would be moved to the top of the stack. All the activities placed above the singleTask instance would be destroyed in this case. When the activity instance does not exist in the stack, the new instance would be placed on the top of the stack similar to the standard mode.

- \*\*singleInstance\*\* : An activity with this launchMode defined would place only a singleton activity instance in the Task. The other activities of the application will be placed in a separate Task.

# AndroidManifest.xml

1. The AndroidManifest.xml file contains information regarding the application that the Android system must know before the codes can be executed.

2. This file is essential in every Android application.

3. It is declared in the root directory.

4. This file performs several tasks such as:

4-1. Providing a unique name to the java package.

4-2. Describing various components of the application such as activity, services, and many more.

4-3. Defining the classes which will implement these components.

# anonymous function vs lambda expression

A lambda expression is a function that can be passed as an argument to another function. An anonymous function is a function that does not have a name and cannot be passed as an argument to another function. Thus, they’re actually opposites.

# anonymous inner class

Anonymous inner classes are the classes that are automatically declared and instantiated within an expression. We cannot apply different access modifiers to them. Anonymous class cannot be static, and cannot define any static fields, method, or class. In other words, we can say that it a class without the name and can have only one object that is created by its definition.

# ANR prevent

ANR stands for Application Not Responding. An ANR will occur if you’re running a process on the UI thread which takes an extended time, usually around 5 seconds. During this point, the GUI (Graphical User Interface) will lock up which can end in anything the user presses won’t be actioned. After the 5 seconds approx. has occurred, if the thread still hasn’t recovered then an ANR dialogue box is shown informing the user that the appliance isn’t responding and can give the user the choice to either wait, in the hope that the app will eventually recover, or to force close the app.

Stop doing heavy tasks on the main thread. Instead, use worker threads such as IntentService, AsyncTask Handler, or another Thread simply. Detecting where ANRs happen is straightforward if it’s a permanent block (deadlock acquiring some locks for instance), but harder if it’s just a short-lived delay. First, re-evaluate your code and appearance for vulnerable spots and long-running operations.

# AppCompatActivity vs Activity

AppCompatActivity provides native ActionBar support that is consistent across the application.

Also, it provides backward compatibility for other material design components till SDK version 7(ActionBar was natively available since SDK 11).

Extending an Activity doesn’t provide any of these.

Note: Since SDK 21 every activity by default, extends AppCompatActivity.

# application Widgets in android

Application widgets are miniature application views that can be embedded in other applications and receive periodic updates.

# apply() vs commit() in Shared Preferences

Data is stored in SharedPreferences in the form of a key-value pair(HashMap). commit() was introduced in API 1 whereas apply() came up with API 9. commit() writes the data synchronously and returns a boolean value of success or failure depending on the result immediately. apply() is asynchronous and it won't return any boolean response. Also, if there is an apply() outstanding and we perform another commit(), then the commit() will be blocked until the apply() is not completed. commit() is instantaneous and performs disk writes. If we're on the main UI thread apply() should be used since it's asynchronous.

# apply:-

Context object: this

Return value: context object

“Apply these to the object,” as the name suggests. It can be used to operate on receiver object members, primarily to initialise them.

# Architecture Android application

1. Services: Used to perform background functionalities.

2. Intent: Used to perform the interconnection between activities and the data passing mechanism.

3. Resource Externalization: strings and graphics.

4. Notification: light, sound, icon, notification, dialog box and toast.

5. Content Providers: It will share the data between applications.

# Architecture Components

1. Room

2. WorkManager

3. Lifecycle

4. ViewModel

5. LiveData

6. Navigation

7. Paging

8. Data Binding

# argument type in constructor kotlin

By default, the constructor arguments are `val` unless explicitly set to `var`.

# Array vs ArrayList

1 The Array is of fixed size, means we cannot resize the array as per need. ArrayList is not of the fixed size we can change the size dynamically.

2 Arrays are of the static type. ArrayList is of dynamic size.

3 Arrays can store primitive data types as well as objects. ArrayList cannot store the primitive data types it can only store the objects.

# array vs collection

Array and Collection are somewhat similar regarding storing the references of objects and manipulating the data, but they differ in many ways. The main differences between the array and Collection are defined below:

1. Arrays are always of fixed size, i.e., a user can not increase or decrease the length of the array according to their requirement or at runtime, but In Collection, size can be changed dynamically as per need.

2. Arrays can only store homogeneous or similar type objects, but in Collection, heterogeneous objects can be stored.

3. Arrays cannot provide the ?ready-made? methods for user requirements as sorting, searching, etc. but Collection includes readymade methods to use.

# Array vs List kotlin

The major difference from usage side is that Arrays have a fixed size while (Mutable)List can adjust their size dynamically. Moreover Array is mutable whereas List is not.

Furthermore kotlin.collections.List is an interface implemented among others by java.util.ArrayList. It's also extended by kotlin.collections.MutableListto be used when a collections that allows for item modification is needed.

On the jvm level Array is represented by arrays. List on the other hand is represented by java.util.List since there are no immutable collections equivalents available in Java.

# Array<Int> vs IntArray

Array<Int> is an Integer[] under the hood, while IntArray is an int[].

This means that when you put an Int in an Array<Int>, it will always be boxed (specifically, with an Integer.valueOf() call). In the case of IntArray, no boxing will occur, because it translates to a Java primitive array.

So no, we can't use them interchangeably.

# ArrayList remove duplicates

There are two ways to remove duplicates from the ArrayList.

1. Using HashSet: By using HashSet we can remove the duplicate element from the ArrayList, but it will not then preserve the insertion order.

2. Using LinkedHashSet: We can also maintain the insertion order by using LinkedHashSet instead of HashSet.

The Process to remove duplicate elements from ArrayList using the LinkedHashSet:

1. Copy all the elements of ArrayList to LinkedHashSet.

2. Empty the ArrayList using clear() method, which will remove all the elements from the list.

3. Now copy all the elements of LinkedHashset to ArrayList.

# ArrayList synchronize

1. Using Collections.synchronizedList() method

2. Using CopyOnWriteArrayList<T>

# ArrayList vs LinkedList

1) ArrayList uses a dynamic array. / LinkedList uses a doubly linked list.

2) ArrayList is not efficient for manipulation because too much is required. / LinkedList is efficient for manipulation.

3) ArrayList is better to store and fetch data. / LinkedList is better to manipulate data.

4) ArrayList provides random access. / LinkedList does not provide random access.

5) ArrayList takes less memory overhead as it stores only object. / LinkedList takes more memory overhead, as it stores the object as well as the address of that object.

# ArrayList vs Vector

1) ArrayList is not synchronized. / Vector is synchronized.

2) ArrayList is not a legacy class. / Vector is a legacy class.

3) ArrayList increases its size by 50% of the array size. / Vector increases its size by doubling the array size.

4) ArrayList is not thread-safe as it is not synchronized. / Vector list is thread-safe as it's every method is synchronized.

# async/await vs launch/join

1. launch / join:-

The launch command is used to start and stop a coroutine. It's as though a new thread has been started. If the code inside the launch throws an exception, it's considered as an uncaught exception in a thread, which is typically written to stderr in backend JVM programs and crashes Android applications. Join is used to wait for the launched coroutine to complete before propagating its exception. A crashed child coroutine, on the other hand, cancels its parent with the matching exception.

2. async / await:-

The async keyword is used to initiate a coroutine that computes a result. You must use await on the result, which is represented by an instance of Deferred. Uncaught exceptions in async code are held in the resultant Deferred and are not transmitted anywhere else. They are not executed until processed.

# asynchronous code vs concurrent code

Asynchronous code is code that can run in the background without blocking the main thread. Concurrent code is code that can run in parallel with other code.

# Asynchronous vs Synchronous

Synchronous programming: In Synchronous programming model, a thread is assigned to complete a task and hence thread started working on it, and it is only available for other tasks once it will end the assigned task.

Asynchronous Programming: In Asynchronous programming, one job can be completed by multiple threads and hence it provides maximum usability of the various threads.

# AsyncTask in orientation change

The lifecycle of an AsyncTask is not tied onto the Activity since it's occurring on a background thread. Hence an orientation change won't stop the AsyncTask. But if the AsyncTask tries to update the UI thread after the orientation is changed, it would give rise to `java.lang.IllegalArgumentException: View not attached to window manager` since it will try to update the former instances of the activity that got reset.

# AsyncTask vs Threade

A Thread is generally used for long tasks to be run in the background. We need a Handler class to use a Thread.

An AsyncTask is an intelligent Thread subclass. It's recommended to use AsyncTask when the caller class is the UI Thread as there is no need to manipulate the handlers.

AsyncTask is generally used for small tasks that can communicate back with the main UI thread using the two methods onPreExecute() and onPostExecute() it has. A Handler class is preferred when we need to perform a background task repeatedly after every x seconds/minutes.

# AsyncTask, Activity lifecycle

An AsyncTask is not tied to the life cycle of the Activity that contains it. So, for example, if you start an AsyncTask inside an Activity and the user rotates the device, the Activity will be destroyed (and a new Activity instance will be created) but the AsyncTask will not die but instead goes on living until it completes.

Then, when the AsyncTask does complete, rather than updating the UI of the new Activity, it updates the former instance of the Activity (i.e., the one in which it was created but that is not displayed anymore!). This can lead to an Exception (of the type java.lang.IllegalArgumentException: View not attached to window manager if you use, for instance, findViewById to retrieve a view inside the Activity).

There’s also the potential for this to result in a memory leak since the AsyncTask maintains a reference to the Activty, which prevents the Activity from being garbage collected as long as the AsyncTask remains alive.

For these reasons, using AsyncTasks for long-running background tasks is generally a bad idea . Rather, for long-running background tasks, a different mechanism (such as a service) should be employed.

# Atomic action

1. The Atomic action is the operation which can be performed in a single unit of a task without any interference of the other operations.

2. The Atomic action cannot be stopped in between the task. Once started it fill stop after the completion of the task only.

3. An increment operation such as a++ does not allow an atomic action.

4. All reads and writes operation for the primitive variable (except long and double) are the atomic operation.

5. All reads and writes operation for the volatile variable (including long and double) are the atomic operation.

6. The Atomic methods are available in java.util.Concurrent package.

# autoboxing, unboxing

The autoboxing is the process of converting primitive data type to the corresponding wrapper class object, eg., int to Integer. The unboxing is the process of converting wrapper class object to primitive data type. For eg., integer to int. Unboxing and autoboxing occur automatically in Java. However, we can externally convert one into another by using the methods like valueOf() or xxxValue().

It can occur whenever a wrapper class object is expected, and primitive data type is provided or vice versa.

1. Adding primitive types into Collection like ArrayList in Java.

2. Creating an instance of parameterized classes ,e.g., ThreadLocal which expect Type.

3. Java automatically converts primitive to object whenever one is required and another is provided in the method calling.

4. When a primitive type is assigned to an object type.

# unboxing, autoboxing

The autoboxing is the process of converting primitive data type to the corresponding wrapper class object, eg., int to Integer. The unboxing is the process of converting wrapper class object to primitive data type. For eg., integer to int. Unboxing and autoboxing occur automatically in Java. However, we can externally convert one into another by using the methods like valueOf() or xxxValue().

It can occur whenever a wrapper class object is expected, and primitive data type is provided or vice versa.

1. Adding primitive types into Collection like ArrayList in Java.

2. Creating an instance of parameterized classes ,e.g., ThreadLocal which expect Type.

3. Java automatically converts primitive to object whenever one is required and another is provided in the method calling.

4. When a primitive type is assigned to an object type.

# backing field

A backing field is an auto-generated field for any property that may only be used inside accessors (getter or setter) and will only be present if it utilizes the default implementation of at least one of the accessors, or if a custom accessor refers to it through the field identifier. This backing field is used to avoid an accessor's recursive call, which would result in a StackOverflowError.

Fields are not allowed in Kotlin classes. When employing custom accessors, however, it is occasionally required to have a backing field. Kotlin includes an automatic backing field for these purposes, which may be accessed by the field identifier.

# Backpressure

Backpressure is the inability of a subscriber to handle incoming events in time. Backpressure can occur when the producer of events is much faster than consumers; if not will error stream.

The first important thing is to choose the correct RxJava construct for your stream. If you thing Backpressure might occur, then Flowable with a correct BackpressureStrategy is the safest choice. You can also try to manually “slow-down” the Producer by adding buffer-type (buffer(), window(), etc) operators before your event handling. Finally you can try to speed up your Consumer – ideally it should be doing small and fast operations. If you require more computation-intensive ones, perhaps some of that logic can be moved to- and parallelised by the Rx stream ifself.

# Bindable annotation (@Bindable)

One way to improve performance when using data binding is to use the @Bindable annotation. This annotation tells the data binding system to only update the UI when the data has actually changed. Without this annotation, the data binding system will update the UI every time any data is changed, even if the UI doesn’t need to be updated.

# blocking vs suspending

1. A blocking call to a function means that a call to any other function, from the same thread, will halt the parent’s execution. Following up, this means that if you make a blocking call on the main thread’s execution, you effectively freeze the UI. Until that blocking calls finishes, the user will see a static screen, which is not a good thing.

2. Suspending doesn’t necessarily block your parent function’s execution. If you call a suspending function in some thread, you can easily push that function to a different thread. In case it is a heavy operation, it won’t block the main thread. If the suspending function has to suspend, it will simply pause its execution. This way you free up its thread for other work. Once it’s done suspending, it will get the next free thread from the pool, to finish its work.

# BlockingQueue

The java.util.concurrent.BlockingQueue is the subinterface of Queue that supports the operations such as waiting for the space availability before inserting a new value or waiting for the queue to become non-empty before retrieving an element from it.

BlockingQueue is an interface which extends the Queue interface. It provides concurrency in the operations like retrieval, insertion, deletion. While retrieval of any element, it waits for the queue to be non-empty. While storing the elements, it waits for the available space. BlockingQueue cannot contain null elements, and implementation of BlockingQueue is thread-safe.

# Bound vs Unbound Service

Services are largely divided into two categories : \*\*Bound Services\*\* and \*\*Unbound/Started Services\*\*

1. \*\*Bound Services\*\*: An Android component may bind itself to a Service using `bindservice()`. A bound service would run as long as the other application components are bound to it. As soon as the components call `unbindService()`, the service destroys itself.

2. \*\*Unbound Services\*\*: A service is started when a component (like activity) calls startService() method and it runs in the background indefinitely even if the original component is destroyed.

# broadcast receiver

A broadcast receiver is a mechanism used for listening to system-level events like listening for incoming calls, SMS, etc. by the host application. It is implemented as a subclass of BroadcastReceiver class and each message is broadcasted as an intent object.

The Broadcast Receiver is defined inside the receiver tags with the necessary actions defined inside the intent filter as shown below.

```

<receiver android:name="".ConnectionReceiver"" >

<intent-filter>

<action android:name=""android.net.conn.CONNECTIVITY\_CHANGE"" />

</intent-filter>

</receiver>

```

# Broadcast Receivers vs Services

A service is used for long running tasks in the background such as playing a music or tracking and updating the user's background location. A Broadcast Receiver is a component that once registered within an application executes the onReceive() method when some system event gets triggered. The events the receiver listens to are defined in the AndroidManifest.xml in the intent filters. Types of system events that a Broadcast Receiver listens to are: changes in the network, boot completed, battery low, push notifications received etc. We can even send our own custom broadcasts using `sendBroadcast(intent)`.

# BufferedInputStream / BufferedOutputStream

Java BufferedOutputStream class is used for buffering an output stream. It internally uses a buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast. Whereas, Java BufferedInputStream class is used to read information from the stream. It internally uses the buffer mechanism to make the performance fast.

# BufferedInputStream / BufferedOutputStream

Java BufferedOutputStream class is used for buffering an output stream. It internally uses a buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast. Whereas, Java BufferedInputStream class is used to read information from the stream. It internally uses the buffer mechanism to make the performance fast.

# Bundle

Bundles are used to pass the required data between various Android activities. These are like HashMap that can take trivial data types.

# Callable interface vs Runnable interface

The Callable interface and Runnable interface both are used by the classes which wanted to execute with multiple threads. However, there are two main differences between the both :

1. A Callable <V> interface can return a result, whereas the Runnable interface cannot return any result.

2. A Callable <V> interface can throw a checked exception, whereas the Runnable interface cannot throw checked exception.

3. A Callable <V> interface cannot be used before the Java 5 whereas the Runnable interface can be used.

# Callable vs Future

Java Callable interface: In Java5 callable interface was provided by the package java.util.concurrent. It is similar to the Runnable interface but it can return a result, and it can throw an Exception. It also provides a run() method for execution of a thread. Java Callable can return any object as it uses Generic.

public interface Callable<V>

Java Future interface: Java Future interface gives the result of a concurrent process. The Callable interface returns the object of java.util.concurrent.Future.

Java Future provides following methods for implementation:

1. cancel(boolean mayInterruptIfRunning): It is used to cancel the execution of the assigned task.

2. get(): It waits for the time if execution not completed and then retrieved the result.

3. isCancelled(): It returns the Boolean value as it returns true if the task was canceled before the completion.

4. isDone(): It returns true if the job is completed successfully else returns false.

# cancel coroutine

Call the cancel function on the CoroutineScope object to cancel a coroutine in Kotlin. This will cancel the coroutine and free up any resources.

# CharArray()

String stays in the string pool until the garbage is collected. If we store the password into a string, it stays in the memory for a longer period, and anyone having the memory-dump can extract the password as clear text. On the other hand, Using CharArray allows us to set it to blank whenever we are done with the password. It avoids the security threat with the string by enabling us to control the memory.

# Checked Exception vs Unchecked Exception

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions, e.g., IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions, e.g., ArithmeticException, NullPointerException, etc. Unchecked exceptions are not checked at compile-time.

# class method vs instance method

1. A method that is declared as static is known as the static method.

1-1. A method that is not declared as static is known as the instance method.

2. We don't need to create the objects to call the static methods.

2-1. The object is required to call the instance methods.

3. Non-static (instance) members cannot be accessed in the static context (static method, static block, and static nested class) directly.

3-1. Static and non-static variables both can be accessed in instance methods.

4. For example: public static int cube(int n){ return n\*n\*n;}

4-1. For example: public void msg(){...}.

# class serializable

A class can become serializable by implementing the Serializable interface.

# class vs object

A class is a template for creating objects. An object is an instance of a class. Classes can contain properties and methods, while objects contain only data.

# clear back stack activity by intent

1. The first approach is to use a `FLAG\_ACTIVITY\_CLEAR\_TOP` flag.

Intent intent= new Intent(ActivityA.this, ActivityB.class);

intent.setFlags(Intent.FLAG\_ACTIVITY\_CLEAR\_TOP);

startActivity(intent);

finish();

2.The second way is by using `FLAG\_ACTIVITY\_CLEAR\_TASK` and `FLAG\_ACTIVITY\_NEW\_TASK` in conjunction.

Intent intent= new Intent(ActivityA.this, ActivityB.class);

intent.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK | Intent.FLAG\_ACTIVITY\_CLEAR\_TASK);

startActivity(intent);

# COLD to HOT transform

1. COLD to HOT

One way to make a Cold observable Hot is by using publish().connect(). publish() converts the Cold observable to a ConnectableObservable, which pretty much behaves like a Hot one. Once triggered with the .connect() operator, it’ll publish events regardless if there are any subscribers.

Another way to transform a Cold observable to a Hot one is by wrapping it with a Subject. The Subject subscribes to the Cold observable immediately and exposes itself as an Observable to future subscribers. Again, the work is performed regardless whether there are any subscribers … and on the other hand multiple subscribers to the Subject won’t trigger the initial work multiple times.

2. HOT to COLD

The first way of transforming (or rather “masking”) a Hot observable to Cold is by using the defer() operator. It defers the creation of the Hot observable altogether, so each new subscriber will trigger the work again (feature of a Cold observable).

Depending on the use-case the pattern mentioned above might be quite wasteful, so another strategy is using the replay().autoConnect(0) paradigm. The replay() operator will cache the values emitted by the Hot observable and re-emit them to future subscribers. autoConnect(0) returns an observable that can be triggered even when there are no subscribers to the underlaying Hot observable. The combination of both just replays cached values from the Hot observable as a Cold one.

# COLD vs HOT

Cold observables are created multiple times and each instance can be triggered on it’s own. Hot observables are like a “stream” of ongoing events – observers can come and go, but the stream is created ones and just goes on.

# Collection

Collection (java.util.Collection) is the primary interface, and every collection must implement this interface.

# Collection framework

Collection Framework is a combination of classes and interface, which is used to store and manipulate the data in the form of objects. It provides various classes such as ArrayList, Vector, Stack, and HashSet, etc. and interfaces such as List, Queue, Set, etc. for this purpose.

# collection vs array

Array and Collection are somewhat similar regarding storing the references of objects and manipulating the data, but they differ in many ways. The main differences between the array and Collection are defined below:

1. Arrays are always of fixed size, i.e., a user can not increase or decrease the length of the array according to their requirement or at runtime, but In Collection, size can be changed dynamically as per need.

2. Arrays can only store homogeneous or similar type objects, but in Collection, heterogeneous objects can be stored.

3. Arrays cannot provide the ?ready-made? methods for user requirements as sorting, searching, etc. but Collection includes readymade methods to use.

# Collection vs Collections

1. The Collection is an interface whereas Collections is a class.

2. The Collection interface provides the standard functionality of data structure to List, Set, and Queue. However, Collections class is to sort and synchronize the collection elements.

3. The Collection interface provides the methods that can be used for data structure whereas Collections class provides the static methods which can be used for various operation on a collection.

# commit() vs apply() in Shared Preferences

Data is stored in SharedPreferences in the form of a key-value pair(HashMap). commit() was introduced in API 1 whereas apply() came up with API 9. commit() writes the data synchronously and returns a boolean value of success or failure depending on the result immediately. apply() is asynchronous and it won't return any boolean response. Also, if there is an apply() outstanding and we perform another commit(), then the commit() will be blocked until the apply() is not completed. commit() is instantaneous and performs disk writes. If we're on the main UI thread apply() should be used since it's asynchronous.

# Companion Object kotlin

In some languages, such as Java, the static keyword is used to declare class members and utilise them without creating an object, i.e. by simply calling them by their class name. In Kotlin, there is nothing called the “static” keyword. So, if we want to achieve the functionality of static member functions, we use the companion objects. This is also referred to as Object Extension.

We must use the companion keyword in front of the object definition to construct a companion object.

# Companion Object purpose

Unlike Java or C#, Kotlin doesn’t have static members or member functions. If you need to write a function that can be called without having a class instance but needs access to the internals of a class, you can write it as a member of a companion object declaration inside that class.

class EventManager {

companion object FirebaseManager {

}

}

val firebaseManager = EventManager.FirebaseManager

The companion object is a singleton. The companion object is a proper object on its own, and can have its own supertypes - and you can assign it to a variable and pass it around. If you're integrating with Java code and need a true static member, you can annotate a member inside a companion object with @JvmStatic.

# Comparable vs Comparator

1) Comparable provides only one sort of sequence. / The Comparator provides multiple sorts of sequences.

2) It provides one method named compareTo(). / It provides one method named compare().

3) It is found in java.lang package. / It is located in java.util package.

4) If we implement the Comparable interface, The actual class is modified. / The actual class is not changed.

# Comparator vs Comparable

1) Comparable provides only one sort of sequence. / The Comparator provides multiple sorts of sequences.

2) It provides one method named compareTo(). / It provides one method named compare().

3) It is found in java.lang package. / It is located in java.util package.

4) If we implement the Comparable interface, The actual class is modified. / The actual class is not changed.

# compileSdkVersion vs targetSdkVersion

1. compileSdkVersion:

1-1. The compileSdkVersion is the version of API the application is compiled against. You can use Android API features involved in that version of the API (as well as all previous versions).

1-2. For example, if you try and use API 15 features but set compileSdkVersion to 14, you will get a compilation error. If you set compileSdkVersion to 15 you can still run the app on an API 14 device as long as your app’s execution paths do not attempt to invoke any APIs specific to API 15.

2. targetSdkVersion:

2-1. The targetSdkVersion indicates that you have tested your app on (presumably up to and including) the version you specify. This is like a certification or sign-off you are giving the Android OS as a hint to how it should handle your application in terms of OS features.

2-2. For example, setting the targetSdkVersion value to “11” or higher permits the system to apply a new default theme (Holo) to the application when running on Android 3.0 or higher. It also disables screen compatibility mode when running on larger screens (because support for API level 11 implicitly supports larger screens).

# compile-time vs runtime

1. In compile-time polymorphism, call to a method is resolved at compile-time.

1-1. In runtime polymorphism, call to an overridden method is resolved at runtime.

2. compile-time, It is also known as static binding, early binding, or overloading.

2-1. runtime, It is also known as dynamic binding, late binding, overriding, or dynamic method dispatch.

3. compile-time, Overloading is a way to achieve compile-time polymorphism in which, we can define multiple methods or constructors with different signatures.

3-1. runtime, Overriding is a way to achieve runtime polymorphism in which, we can redefine some particular method or variable in the derived class. By using overriding, we can give some specific implementation to the base class properties in the derived class.

4. compile-time, It provides fast execution because the type of an object is determined at compile-time.

4-1. runtime, It provides slower execution as compare to compile-time because the type of an object is determined at run-time.

5. Compile-time polymorphism provides less flexibility because all the things are resolved at compile-time.

5-1. Run-time polymorphism provides more flexibility because all the things are resolved at runtime.

# Completable RxJava

maps an operation that either completes without returning a value (onComplete()) or errors out (onError(throwable)).

# composition

Holding the reference of a class within some other class is known as composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition. In other words, we can say that composition is the particular case of aggregation which represents a stronger relationship between two objects. Example: A class contains students. A student cannot exist without a class. There exists composition between class and students.

# composition vs aggregation

Aggregation represents the weak relationship whereas composition represents the strong relationship. For example, the bike has an indicator (aggregation), but the bike has an engine (composition).

# concatMap(), flatMap(), switchMap()

1. flatMap() is used to split chain to multiple intermediary streams, and results of which are flattened to a single stream. Emissions of these intermediary streams are directly propagated to the main stream in any order.

2. switchMap() is similar to flatMap(), but whenever a new item is being emitted from a source, all the intermediary streams generated are terminated, and hence only the latest intermediary stream remains active.

3. concatMap() also works similar to flatMap(), with an exception such that intermediary streams are activated based on the order appearance..

# concurrency API

Concurrency API can be developed using the class and interfaces of java.util.Concurrent package. There are the following classes and interfaces in java.util.Concurrent package.

Executor

FarkJoinPool

ExecutorService

ScheduledExecutorService

Future

TimeUnit(Enum)

CountDownLatch

CyclicBarrier

Semaphore

ThreadFactory

BlockingQueue

DelayQueue

Locks

Phaser

# concurrency kotlin

Kotlin has built-in support for concurrency via coroutines. Coroutines are light-weight threads that can be used to improve the performance of concurrent code.

# concurrent code vs asynchronous code

Asynchronous code is code that can run in the background without blocking the main thread. Concurrent code is code that can run in parallel with other code.

# concurrently instead of sequentially

There are a few reasons you might want to run two tasks concurrently instead of sequentially. One reason is if the tasks are independent of each other and can be run in parallel. Another reason is if one task is dependent on the other task and you want to avoid blocking the main thread. Finally, if you have a limited number of resources available, you might want to run tasks concurrently in order to make better use of those resources.

# Conditional Operators RxJava

1. DefaultIfEmpty emits item from the source Observable, or a default item if the source Observable is empty.

2. TakeWhile operator discards items emitted by an Observable after a specified condition becomes false.

3. Of course, there more others operators that could cover our needs like Contain, SkipWhile, SkipUntil, TakeUntil, etc.

# Connectable Observables RxJava

A ConnectableObservable resembles an ordinary Observable, except that it doesn't begin emitting items when it is subscribed to, but only when the connect operator is applied to it.

In this way, we can wait for all intended observers to subscribe to the Observable before the Observable begins emitting items.

# const vs val

By default val properties are set at runtime. Adding a const modifier on a val would make a compile-time constant. A const cannot be used with a var or on its own. A const is not applicable on a local variable.

# constant kotlin

In Kotlin, if you want to create the local constants which are supposed to be used with in the class then you can create it like below:

val MY\_CONSTANT\_1 = ""Constants1""

// or

const val MY\_CONSTANT\_2 = ""Constants2""

Like val, variables defined with the const keyword are immutable. The difference here is that const is used for variables that are known at compile-time.

Also avoid using companion objects. Behind the hood, getter and setter instance methods are created for the fields to be accessible. Calling instance methods is technically more expensive than calling static methods. Instead define the constants in object:

object DbConstants {

const val TABLE\_USER\_ATTRIBUTE\_EMPID = ""\_id""

const val TABLE\_USER\_ATTRIBUTE\_DATA = ""data""

}

# construction arguments Fragment

Construction arguments for a Fragment are passed via Bundle using the Fragment#setArgument(Bundle) method. The passed-in Bundle can then be retrieved through the Fragment#getArguments() method in the appropriate Fragment lifecycle method.

It is a common mistake to pass in data through a custom constructor. Non-default constructors on a Fragment are not advisable because the Fragment may be destroyed and recreated due to a configuration change (e.g. orientation change). Using #setArguments()/getArguments() ensures that when the Fragment needs to be recreated, the Bundle will be appropriately serialized/deserialized so that construction data is restored.

# constructor

The constructor can be defined as the special type of method that is used to initialize the state of an object. It is invoked when the class is instantiated, and the memory is allocated for the object. Every time, an object is created using the new keyword, the default constructor of the class is called. The name of the constructor must be similar to the class name. The constructor must not have an explicit return type.

# constructor argument type kotlin

By default, the constructor arguments are `val` unless explicitly set to `var`.

# constructor as final

The constructor can never be declared as final because it is never inherited. Constructors are not ordinary methods; therefore, there is no sense to declare constructors as final. However, if you try to do so, The compiler will throw an error.

# constructor chaining by super

class Person

{

String name,address;

int age;

public Person(int age, String name, String address)

{

this.age = age;

this.name = name;

this.address = address;

}

}

class Employee extends Person

{

float salary;

public Employee(int age, String name, String address, float salary)

{

super(age,name,address);

this.salary = salary;

}

}

public class Test

{

public static void main (String args[])

{

Employee e = new Employee(22, ""Mukesh"", ""Delhi"", 90000);

System.out.println(""Name: ""+e.name+"" Salary: ""+e.salary+"" Age: ""+e.age+"" Address: ""+e.address);

}

}

# constructor chaining by this

Constructor chaining enables us to call one constructor from another constructor of the class with respect to the current class object. We can use this keyword to perform constructor chaining within the same class. Consider the following example which illustrates how can we use this keyword to achieve constructor chaining.

# constructor- default constructor

The purpose of the default constructor is to assign the default value to the objects. The java compiler creates a default constructor implicitly if there is no constructor in the class.

# constructor final

No, the constructor can't be final.

# constructor inherited

# constructor copy object

There is no copy constructor in java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

1. By constructor

2. By assigning the values of one object into another

3. By clone() method of Object class

# constructor overload

Yes, the constructors can be overloaded by changing the number of arguments accepted by the constructor or by changing the data type of the parameters. Consider the following example.

# constructor static

As we know that the static context (method, block, or variable) belongs to the class, not the object. Since Constructors are invoked only when the object is created, there is no sense to make the constructors static. However, if you try to do so, the compiler will show the compiler error.

# constructor this(),super()

No, because this() and super() must be the first statement in the class constructor.

# constructor kotlin

Constructors in Kotlin are of two types:

\*\*Primary\*\* - These are defined in the class headers. They cannot hold any logic. There's only one primary constructor per class.

\*\*Secondary\*\* - They're defined in the class body. They must delegate to the primary constructor if it exists. They can hold logic. There can be more than one secondary constructors.

# constructor types

1. Default Constructor: default constructor is the one which does not accept any value. The default constructor is mainly used to initialize the instance variable with the default values. It can also be used for performing some useful task on object creation. A default constructor is invoked implicitly by the compiler if there is no constructor defined in the class.

2. Parameterized Constructor: The parameterized constructor is the one which can initialize the instance variables with the given values. In other words, we can say that the constructors which can accept the arguments are called parameterized constructors.

# constructor types

1. Primary Constructor - This type of constructor is initialised in the class header and is provided after the class name. It is declared using the “constructor” keyword. Parameters are optional in this type of constructor.

If no annotations or access modifiers are provided, the constructor keyword can be omitted. The initialization code can be placed in a separate initializer block prefixed with the init keyword because the primary constructor cannot contain any code.

2. Secondary Constructor - Secondary constructors allow for the initialization of variables as well as the addition of logic to the class. They have the constructor keyword prefixed to them.

The compiler determines which secondary constructor will be called based on the inputs provided. We don't specify which constructor to use in the above program, so the compiler chooses for us.

In Kotlin, a class can contain one or more secondary constructors and at most one primary constructor. The primary constructor initializes the class, while the secondary constructor initialises the class and adds some additional logic.

# constructor vs initializer in Kotlin

A constructor is a special method invoked when an object is created. An initializer is a special method you can use to initialize an object before its first use. Both constructors and initializers are typically declared with the unit keyword.

# constructor vs method

1. A constructor is used to initialize the state of an object.

1-1. A method is used to expose the behavior of an object.

2. A constructor must not have a return type.

2-1. A method must have a return type.

3. The constructor is invoked implicitly.

3-1. The method is invoked explicitly.

4. The Java compiler provides a default constructor if you don't have any constructor in a class.

4-1. The method is not provided by the compiler in any case.

5. The constructor name must be same as the class name.

5-1. The method name may or may not be same as class name.

# constructor vs method

1. A constructor is used to initialize the state of an object.

1-1. A method is used to expose the behavior of an object.

2. A constructor must not have a return type.

2-1. A method must have a return type.

3. The constructor is invoked implicitly.

3-1. The method is invoked explicitly.

4. The Java compiler provides a default constructor if you don't have any constructor in a class.

4-1. The method is not provided by the compiler in any case.

5. The constructor name must be same as the class name.

5-1. The method name may or may not be same as class name.

# constructs RxJava

1. Completable – maps an operation that either completes without returning a value (onComplete()) or errors out (onError(throwable)).

Single – either returns a value (onSuccess(value)) or errors out (onError(throwable)).

2. Maybe – has 3 options – returns a value successfully (onSuccess(value)), completes successfully without any value (onComplete()) or errors out (onError(throwable)).

3. Observable – represents a stream of events that emits zero to many events (onNext(value)), then either completes (onComplete()) or errors out (onError(throwable)). It does NOT support backpressure (see #32).

4. Flowable – like an Observable, however it DOES support backpressure.

# Container

Containers carry objects and widgets together, based on which specific items are required and in what particular arrangement is needed. Containers may hold labels, buttons, fields, or even child containers, etc. For example, if you want a form with fields on the left and labels on the right, you will need a container. If you want the OK and Cancel buttons to be below the rest of the form, next to one another, and flush to the right side of the screen, you will need a container. If you have several widgets, you will need a container to have a root element to place the widgets inside.

Android provides a collection of view classes that serve as containers for views. These container classes are called layouts, which are defined in the form of XML files that cannot be changed by our code during execution. The layout managers provided by Android SDK are LinearLayout, RelativeLayout, FrameLayout, AbsoluteLayout, GridLayout, and TableLayout.

# Content provider

Content provider is one of the primary building blocks of Android applications, which manages access to a central repository of data. It acts as a standard interface that connects data in one process with code running in another process. So it can be used to share the data between different applications.

They are responsible for encapsulating the data and providing mechanisms for defining data security. It is implemented as a subclass of ContentProviderclass and must implement a set of APIs that will enable other applications to perform transactions.

# context

The context in Android is the context of the current state of the application or object. The context comes with services like giving access to databases and preferences, resolving resources, and more.

There are two types of context. They are:

1. Activity context

1-1. This activity context is attached to the lifecycle of an activity.

1-2. The activity context can be used when you are passing the context in the scope of an activity or you need the context whose lifecycle is attached to the context of the activity.

2. Application context:

2-1. This application context is attached to the lifecycle of an application.

2-2. The application context should be used where you need a context whose lifecycle is separate from the current context or when you are passing a context beyond the scope of activity.

# context switching

In Context switching the state of the process (or thread) is stored so that it can be restored and execution can be resumed from the same point later. Context switching enables the multiple processes to share the same CPU.

# coroutine

1. A coroutine is an instance of suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one.

2. The Kotlin team defines coroutines as “lightweight threads”. They are sort of tasks that the actual threads can execute. Kotlin coroutines introduce a new style of concurrency that can be used on Android to simplify async code. The official documentation says that coroutines are lightweight threads. By lightweight, it means that creating coroutines doesn’t allocate new threads. Instead, they use predefined thread pools and smart scheduling for the purpose of which task to execute next and which tasks later.

# Coroutinee

Coroutines are similar to thin threads. Coroutines are lightweight since they don't allocate new threads when they're created. Instead, they employ pre-defined thread pools as well as intelligent scheduling. The process of deciding which piece of work you will do next is known as scheduling. Coroutines can also be paused and resumed in the middle of their execution. This means you can have a long-term project that you can work on incrementally. You can pause it as many times as you want and continue it whenever you're ready.

# coroutine cancel

Call the cancel function on the CoroutineScope object to cancel a coroutine in Kotlin. This will cancel the coroutine and free up any resources.

# coroutine launch

Call the launch function on the CoroutineScope object to launch a coroutine, passing in the function you wish to execute. This will create a new coroutine and launch it immediately.

# coroutine pass data

One of the best ways to pass data between coroutines in Kotlin is with channel objects, which allow for safe and synchronized communication between separate threads or processes. To create a channel object, simply use the Channel constructor and define any channels you want to send data through.

# coroutine vs thread

1. A thread is an execution unit that can run independently from other threads.

1-1. A coroutine is a unit of execution that can be suspended and resumed, allowing it to share resources with other concurrent or parallel executions.

2. Threads are typically heavier than coroutines, so they can be more expensive in terms of performance. However, this is not always the case, and it really depends on the specific implementation.

2-1. In general, coroutines tend to be more efficient when it comes to CPU usage, but threads may be better when it comes to I/O bound tasks.

# covariant return type

Now, since java5, it is possible to override any method by changing the return type if the return type of the subclass overriding method is subclass type. It is known as covariant return type. The covariant return type specifies that the return type may vary in the same direction as the subclass.

# create complier plugin kotlin

To create a compiler plugin in Kotlin, you first need to create an abstract class that extends the CompilerPlugin class. This class defines several functions that should be implemented by your plugin, including load and init. After this, you can use the CompilerInstance class to access the Kotlin compiler and use its API.

# create lambda expression

First, you must define the parameters the expression accepts using the parentheses operator. You can then provide an executable block of code within curly braces and use the arrow operator to indicate that this code is the body of the lambda expression.

val items = listOf(1, 2, 3, 4, 5)

items.fold(0, {

acc: Int, i: Int ->

print(""acc = $acc, i = $i, "")

val result = acc + i

println(""result = $result"")

result

})

To pass a lambda expression as an argument to a function, include it within parentheses after the function name and any necessary arguments. This will cause the code within that lambda expression to be executed whenever the function is called.

# critical loops activity

Loop 1, Entire Lifetime: The activity happens between onCreate and onDestroy.

Loop 2, Visible Lifetime: The activity happens between onStart and onStop

Loop 3, Foreground Lifetime: The activity happens between onResume and onPause

# custom operators in RxJava

Users can create custom operators in RxJava, and this is highly recommended to reuse existing or any combination. It is tricky to implement a new operator as it would lead to too many of errors, such as thread safety, API breaking, etc.

# Daemon thread

Garbage collector thread

# daemon threads

The daemon threads are the low priority threads that provide the background support and services to the user threads. Daemon thread gets automatically terminated by the JVM if the program remains with the daemon thread only, and all other user threads are ended/died. There are two methods for daemon thread available in the Thread class:

public void setDaemon(boolean status): It used to mark the thread daemon thread or a user thread.

public boolean isDaemon(): It checks the thread is daemon or not.

# Dagger

Dagger is a Dependency Injection framework that will generate a lot of boilerplate code for you to achieve the goal of Dependency injection in Android development. Not that one must have a good understanding of Dependency Injection before answering this question.

# Data Binding

Data Binding library is a support library that provides the feature of binding UI components in an activity/fragment to the data sources of the application. The library carries out this binding task in a declarative format and not in a programmatic way.

Below is an example to understand the working of this library accurately:

To find a TextView widget and bind it to the userName property of the ViewModel variable, the findViewById() method is called:

TextView textView = findViewById(R.id.sample\_text);

textView.setText(viewModel.getUserName());

After using the Data Binding library, the above code changes by using the assignment expression as follows:

<TextView

android:text=”@{viewmodel.userName}” />

Advantages of Data Binding Component:

1. Make code simpler and easy to maintain by removing UI frameworks called in the activity.

2. Allows classes and methods to observe changes in data

3. Allows to make objects and fill which works as collection observables.

# Data Binding pros

Data binding is a process that allows you to automatically synchronize your ViewModel and View. When data binding is enabled, any changes that you make to your ViewModel will be automatically reflected in your View. This makes it easier to keep your View and ViewModel in sync, and can help to reduce the amount of boilerplate code that you need to write.

1. Make code simpler and easy to maintain by removing UI frameworks called in the activity.

2. Allows classes and methods to observe changes in data

3. Allows to make objects and fill which works as collection observables.

# data binding two-way vs one-way

Two-way data binding means that changes to either the model or the view will update the other automatically. One-way data binding means that changes to the model will update the view, but not vice versa.

# data class

In Java, to create a class that stores data, you need to set the variables, the getters and the setters, override the `toString()`, `hash()` and `copy()` functions. In Kotlin you just need to add the `data` keyword on the class and all of the above would automatically be created under the hood.

Thus, data classes saves us with lot of code. It creates component functions such as `component1()`.. `componentN()` for each of the variables.

The Data class is a simple class that holds data and provides typical functions. To declare a class as a data class, use the data keyword.

he following functions are automatically derived by the compiler for the data classes:

1. equals() - The equals() function returns true if two objects have the identical contents. It operates similarly to ""==,"" although for Float and Double values it works differently.

2. hashCode() - The hashCode() function returns the object's hashcode value.

3. copy() - The copy() function is used to duplicate an object, changing only a few of its characteristics while leaving the rest unaltered.

4. toString() - This function returns a string containing all of the data class's parameters.

To ensure consistency, data classes must meet the following requirements:

1. At least one parameter is required for the primary constructor.

2. val or var must be used for all primary constructor parameters.

3. Abstract, open, sealed, or inner data classes are not possible.

4.Only interfaces may be implemented by data classes.

# data class vs regular class

A data class is a class intended to hold data. A regular class is a class that can perform arbitrary operations. Data classes are typically simpler and more efficient than regular classes.

# data types by AIDL

-string

-charSequence

-List

-Map

-all native Java data types like int,long, char and Boolean

# database in android

SQLite is the open-source relational database used in Android. The SQLite engine is serverless, transactional, and also self-contained. Instead of the client-server relationship of most database management systems, the SQLite engine is integrally linked with the application. The library can be called dynamically and it can make use of simple function calls that reduce latency in database access.

# DDMS

1. Port forwarding services.

2. Thread and heap information.

3. Logcat.

4. Screen capture on the device.

5. Network traffic tracking.

6. Incoming call and SMS spoofing.

7. Location data spoofing.

# deadlock

Deadlock is a situation in which every thread is waiting for a resource which is held by some other waiting thread. In this situation, Neither of the thread executes nor it gets the chance to be executed. Instead, there exists a universal waiting state among all the threads. Deadlock is a very complicated situation which can break our code at runtime.

# deadlock condition

We can detect the deadlock condition by running the code on cmd and collecting the Thread Dump, and if any deadlock is present in the code, then a message will appear on cmd.

Ways to avoid the deadlock condition in Java:

1. Avoid Nested lock: Nested lock is the common reason for deadlock as deadlock occurs when we provide locks to various threads so we should give one lock to only one thread at some particular time.

2. Avoid unnecessary locks: we must avoid the locks which are not required.

3. Using thread join: Thread join helps to wait for a thread until another thread doesn't finish its execution so we can avoid deadlock by maximum use of join method.

# default access specifier

1. Public The classes, methods, or variables which are defined as public, can be accessed by any class or method.

2. Protected Protected can be accessed by the class of the same package, or by the sub-class of this class, or within the same class.

3. Default Default are accessible within the package only. By default, all the classes, methods, and variables are of default scope.

4. Private The private class, methods, or variables defined as private can be accessed within the class only.

# default argument in Kotlin

To declare a default argument in Kotlin, use the default keyword when defining a function parameter. This will specify a default value for that argument, which will be used if no actual value is supplied when calling the function.

# default value of variable

The local variables are not initialized to any default value, neither primitives nor object references.

# define an object in Kotlin

To define an object in Kotlin, simply declare a class and instantiate it with the new keyword. This will create a new class instance, which can perform various actions.

val newObject= object {

val one = ""Hello""

val two = ""World""

override fun toString() = ""$one $two""

}

# dependency injection

Dependency injection is a technique used to remove hard-coded dependencies between objects, making it easier to change them later. This is useful in situations where you want to be able to swap out one implementation of a dependency for another, without having to change the code that uses it.

# Dequeue

Dequeue interface: it is a double-ended-queue. It allows the insertion and removal of elements from both ends. It implants the properties of both Stack and queue so it can perform LIFO (Last in first out) stack and FIFO (first in first out) queue, operations.

# deserialization

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

# Destructuring Declaration

Destructuring Declarations is a smart way to assign multiple values to variables from data stored in objects/arrays. Within paratheses, we've set the variable declarations. Under the hood, destructuring declarations create component functions for each of the class variables.

In Kotlin, destructuring is a convenient way to extract multiple values from data stored in objects and Arrays. It can be used in locations that receive data. It is used because sometimes, it is convenient to destructure an object into several variables.

For Example:

val (name, age) = developer

Now, we can use name and age independently as follows:

println(name)

println(age)

# dialog boxes

1. AlertDialog:

The AlertDialog supports 0-3 buttons, along with a list of selectable items such as checkboxes and radio buttons.

It is used when you want to ask the user about taking a decision between yes or no in response to any particular action taken by the user, by remaining in the same activity and without changing the screen.

2. DatePickerDialog:

It is used for selecting the date by the user.

3. TimePickerDialog:

Used for selecting the time by the user.

4. ProgressDialog:

It is an extension of the AlertDialog and is used to display a progress bar. It also supports the addition of buttons.

This class was deprecated in API level 26 because it prevents the user from interacting with the application. Instead of this class, we can use a progress indicator such as ProgressBar, which can be embedded in the user interface of your application.

# Dictionary

The Dictionary class provides the capability to store key-value pairs.

# Dispatchers.Main

The {Dispatchers.Main} expression is used to specify that a particular coroutine should run on the main thread. This is important because some operations can only be performed on the main thread, and so specifying that a coroutine should run on the main thread ensures that it will be able to perform those operations.

# doInBackground

The returned value of the doInBackground goes to the onPostExecute() method. We can update the main UI thread from here. To get the returned value in the onCreate() method we need to use the following code snippet.

```

MyTask myTask= new MyTask();

String result=myTask.execute().get();

```

This approach is not recommended as it blocks the main UI thread until the value is not returned. The ideal scenario to use it is when the other views of the UI thread need the value from the AsyncTask for processing.

# double-bang !!

The not-null assertion operator !! converts any value to a non-null type and throws a KotlinNullPointerException exception if the value is null.

Consider:

fun main(args: Array<String>) {

var email: String?

email = null

println(email!!)

}

This operator should be used in cases where the developer is guaranteeing – it allows you to be 100% sure that its value is not null.

# drawable folder

In Android, a drawable folder is compiled a visual resource that can use as a background, banners, icons, splash screen, etc.

# dynamic binding vs static binding

In case of the static binding, the type of the object is determined at compile-time whereas, in the dynamic binding, the type of the object is determined at runtime.

# Elasticity in RxJava

It means that the throughput of the system scales up or down automatically to meet demand as a resource is proportionally added/ removed. Elasticity, therefore, builds up Scalability and expands by adding the notion of automatic resource management. With this, we shall conclude the topic “RxJava Interview questions.” We have seen the top 15 questions on RxJava, which will be helpful for cracking interviews. Sometimes, the interviewer can even ask you the syntax or how is any particular part coded. Having theoretical knowledge is of much importance to at least access your skills.

# Elvis Operator ( ?: )

Elvis Operator ( ?: ) - When the original variable is null, the Elvis operator is used to return a non-null value or a default value. In other words, the elvis operator returns the left expression if it is not null, otherwise, it yields the right expression. Only if the left-hand side expression is null is the right-hand side evaluated.

Furthermore, on the right side of the Elvis operator, we may use throw and return expressions, which is particularly handy in functions. As a result, instead of returning a default value on the right side of the Elvis operator, we can throw an exception.

# Encapsulation pros

1. By providing only the setter or getter method, you can make the class read-only or write-only. In other words, you can skip the getter or setter methods.

2. It provides you the control over the data. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

3. It is a way to achieve data hiding in Java because other class will not be able to access the data through the private data members.

4. The encapsulate class is easy to test. So, it is better for unit testing.

5. The standard IDE's are providing the facility to generate the getters and setters. So, it is easy and fast to create an encapsulated class in Java.

# encapsulation vs abstraction

Abstraction hides the implementation details whereas encapsulation wraps code and data into a single unit.

# Enumeration vs Iterator

1) The Iterator can traverse legacy and non-legacy elements. / Enumeration can traverse only legacy elements.

2) The Iterator is fail-fast. / Enumeration is not fail-fast.

3) The Iterator is slower than Enumeration. / Enumeration is faster than Iterator.

4) The Iterator can perform remove operation while traversing the collection. / The Enumeration can perform only traverse operation on the collection.

# equals()

The equals method is used to check whether two objects are the same or not. It needs to be overridden if we want to check the objects based on the property.

For example, Employee is a class that has 3 data members: id, name, and salary. However, we want to check the equality of employee object by the salary. Then, we need to override the equals() method.

# Error and Exception base class

The Throwable class is the base class for Error and Exception.

# error handling operators in RxJava

We have two categories of such operators, one for side effects only and the other for handle error and continue. doOnError(…), onErrorReturn(…), onErrorResumeNext(…) are some of the error handling operators in RxJava.

# Escape characters android

Escape characters are preceded by double backslashes. For example, a newline character is created using ‘\\n’

# essential states of activity

1. Active – if the activity is at the foreground

2. Paused – if the activity is at the background and still visible

3. Stopped – if the activity is not visible and therefore is hidden or obscured by another activity

4. Destroyed – when the activity process is killed or completed terminated

# essential states of activity

1. Active – if the activity is at the foreground

2. Paused – if the activity is at the background and still visible

3. Stopped – if the activity is not visible and therefore is hidden or obscured by another activity

4. Destroyed – when the activity process is killed or completed terminated

# event handling vs reactive programming

Reactive programming is a programming paradigm that is concerned with data streams and the propagation of changes. Event handling, on the other hand, is a way of responding to events that occur in a program. In the context of Android development, reactive programming can be used to handle events such as user input, while event handling is used to respond to events such as button clicks.

# Exception

Exception Handling is a mechanism that is used to handle runtime errors. It is used primarily to handle checked exceptions. Exception handling maintains the normal flow of the program. There are mainly two types of exceptions: checked and unchecked. Here, the error is considered as the unchecked exception.

Can subclass overriding method declare an exception if parent class method doesn't throw an exception?

Yes but only unchecked exception not checked.

# Exception and Error base class

The Throwable class is the base class for Error and Exception.

# exception classe hierarchy

Throwable

---Exception

------IOException

------SQLException

------ClassNotFoundException

------RuntimeException

---------ArithmeticException

---------NullPointerException

---------NumberFormatException

---------IndexOutOfBoundsException

------------ArrayIndexOutOfBoundsException

------------StringIndexOutOfBoundsException

---Error

------StackOverflowError

------VirtualMachineError

------OutOfMemoryError

# Exception Handling

Exception Handling is a mechanism that is used to handle runtime errors. It is used primarily to handle checked exceptions. Exception handling maintains the normal flow of the program. There are mainly two types of exceptions: checked and unchecked. Here, the error is considered as the unchecked exception.

Can subclass overriding method declare an exception if parent class method doesn't throw an exception?

Yes but only unchecked exception not checked.

# exception in coroutine

If there’s an exception thrown inside a coroutine, then the coroutine will be cancelled. All the coroutine’s children will also be cancelled, and any pending work in those coroutines will be lost.

# exception propagation

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This procedure is called exception propagation. By default, checked exceptions are not propagated.

# exception types

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception: Checked exceptions are the one which are checked at compile-time. For example, SQLException, ClassNotFoundException, etc.

2. Unchecked Exception: Unchecked exceptions are the one which are handled at runtime because they can not be checked at compile-time. For example, ArithmaticException, NullPointerException, ArrayIndexOutOfBoundsException, etc.

3. Error: Error cause the program to exit since they are not recoverable. For Example, OutOfMemoryError, AssertionError, etc.

# exceptions in Android

1. Inflate Exception

2. Surface.OutOfResourceException

3. SurfaceHolder.BadSurfaceTypeException

4. WindowManager.BadTokenException

# Executor interface

The Executor Interface provided by the package java.util.concurrent is the simple interface used to execute the new task. The execute() method of Executor interface is used to execute some given command.

# ExecutorService

The ExecutorService Interface is the subinterface of Executor interface and adds the features to manage the lifecycle.

# ExecutorService interface vs ScheduledExecutorService

ExecutorServcie and ScheduledExecutorService both are the interfaces of java.util.Concurrent package but scheduledExecutorService provides some additional methods to execute the Runnable and Callable tasks with the delay or every fixed time period.

# Explicit Intent vs Implicit Intent

Explicit Intent:

An Explicit Intent is where you inform the system about which activity should handle this intent. Here target component is defined directly in the intent.

Implicit Intent:

An Implicit Intent permits you to declare the action you want to carry out. Further, the Android system will check which components are registered to handle that specific action based on intent data. Here target component is not defined in the intent.

# extension function vs regular function

An extension function is a function defined for a specific type and can be called on variables of that type. A regular function is not defined for a specific type and can be called on any type of variable.

# extension functions

Extension functions are like extensive properties attached to any class in Kotlin. Extension functions are used to add methods or functionalities to an existing class even without inheriting the class. For example: Suppose, we have views where we need to play with the visibility of the views. So, we can create an extension function for views as follows:

fun View.show() {

this.visibility = View.VISIBLE

}

fun View.hide() {

this.visibility = View.GONE

}

and to use it, we use, like,

toolbar.hide()

# extension methods

Following are some extension methods that Kotlin provides to java.io.File:

1. bufferedReader(): It is used for reading the contents of a file into BufferedReader.

2. readBytes(): It is used for reading the contents of the file to ByteArray.

3. readText(): It is used for reading contents of the file to a single String.

4. forEachLine(): It is used for reading a file line by line in Kotlin.

5. readLines(): It is used for reading lines in the file to List.

# Externalizable

The Externalizable interface is used to write the state of an object into a byte stream in a compressed format. It is not a marker interface.

# Externalizable vs Serializable

1. The Serializable interface does not have any method, i.e., it is a marker interface.

1-1. The Externalizable interface contains is not a marker interface, It contains two methods, i.e., writeExternal() and readExternal().

2. It is used to ""mark"" Java classes so that objects of these classes may get the certain capability.

2-1. The Externalizable interface provides control of the serialization logic to the programmer.

3. Serializable, It is easy to implement but has the higher performance cost.

3-1. Externalizable, It is used to perform the serialization and often result in better performance.

4. Serializable, No class constructor is called in serialization.

4-1. Externalizable,We must call a public default constructor while using this interface.

# fail-fast

The Iterator in java which immediately throws ConcurrentmodificationException, if any structural modification occurs in, is called as a Fail-fast iterator. Fail-fats iterator does not require any extra space in memory.

# features kotlin only

1. Null Safety

2. Operator Overloading

3. Coroutines

4. Range expressions

5. Smart casts

6. Companion Objects

# Filter RxJava

The operator filter emits only those items from an observable that pass a predicate test.

# FilterStreams

FilterStream classes are used to add additional functionalities to the other stream classes. FilterStream classes act like an interface which read the data from a stream, filters it, and pass the filtered data to the caller. The FilterStream classes provide extra functionalities like adding line numbers to the destination file, etc.

# final blank variable

A final variable, not initialized at the time of declaration, is known as the final blank variable. We can't initialize the final blank variable directly. Instead, we have to initialize it by using the class constructor. It is useful in the case when the user has some data which must not be changed by others, for example, PAN Number.

# final blank variable initialize

Yes, if it is not static, we can initialize it in the constructor. If it is static blank final variable, it can be initialized only in the static block.

# final class

If we make any class final, we can't inherit it into any of the subclasses.

# final constructor

The constructor can never be declared as final because it is never inherited. Constructors are not ordinary methods; therefore, there is no sense to declare constructors as final. However, if you try to do so, The compiler will throw an error.

# final interface

No, we cannot declare an interface as final because the interface must be implemented by some class to provide its definition. Therefore, there is no sense to make an interface final. However, if you try to do so, the compiler will show an error.

# final method

If we change any method to a final method, we can't override it. More Details.

# final method vs abstract method

The main difference between the final method and abstract method is that the abstract method cannot be final as we need to override them in the subclass to give its definition.

# final variable

In Java, the final variable is used to restrict the user from updating it. If we initialize the final variable, we can't change its value. In other words, we can say that the final variable once assigned to a value, can never be changed after that. The final variable which is not assigned to any value can only be assigned through the class constructor.

# final, finally, finalize

1) Final is used to apply restrictions on class, method, and variable. The final class can't be inherited, final method can't be overridden, and final variable value can't be changed. / Finally is used to place important code, it will be executed whether an exception is handled or not. / Finalize is used to perform clean up processing just before an object is garbage collected.

2) Final is a keyword. / Finally is a block. / Finalize is a method.

# finalize()

The finalize() method is invoked just before the object is garbage collected. It is used to perform cleanup processing. The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created an object without new, you can use the finalize method to perform cleanup processing (destroying remaining objects). The cleanup processing is the process to free up all the resources, network which was previously used and no longer needed. It is essential to remember that it is not a reserved keyword, finalize method is present in the object class hence it is available in every class as object class is the superclass of every class in java. Here, we must note that neither finalization nor garbage collection is guaranteed.

# finally block

The ""finally"" block is used to execute the important code of the program. It is executed whether an exception is handled or not. In other words, we can say that finally block is the block which is always executed. Finally block follows try or catch block. If you don't handle the exception, before terminating the program, JVM runs finally block, (if any). The finally block is mainly used to place the cleanup code such as closing a file or closing a connection. Here, we must know that for each try block there can be zero or more catch blocks, but only one finally block. The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

# FLAG\_ACTIVITY\_CLEAR\_TASK vs FLAG\_ACTIVITY\_CLEAR\_TOP

`FLAG\_ACTIVITY\_CLEAR\_TASK` is used to clear all the activities from the task including any existing instances of the class invoked.

The Activity launched by intent becomes the new root of the otherwise empty task list. This flag has to be used in conjunction with `FLAG\_ ACTIVITY\_NEW\_TASK`.

`FLAG\_ACTIVITY\_CLEAR\_TOP` on the other hand, if set and if an old instance of this Activity exists in the task list then barring that all the other activities are removed and that old activity becomes the root of the task list.

Else if there's no instance of that activity then a new instance of it is made the root of the task list.

Using `FLAG\_ACTIVITY\_NEW\_TASK` in conjunction is a good practice, though not necessary.

# FLAG\_ACTIVITY\_CLEAR\_TOP vs FLAG\_ACTIVITY\_CLEAR\_TASK

`FLAG\_ACTIVITY\_CLEAR\_TASK` is used to clear all the activities from the task including any existing instances of the class invoked.

The Activity launched by intent becomes the new root of the otherwise empty task list. This flag has to be used in conjunction with `FLAG\_ ACTIVITY\_NEW\_TASK`.

`FLAG\_ACTIVITY\_CLEAR\_TOP` on the other hand, if set and if an old instance of this Activity exists in the task list then barring that all the other activities are removed and that old activity becomes the root of the task list.

Else if there's no instance of that activity then a new instance of it is made the root of the task list.

Using `FLAG\_ACTIVITY\_NEW\_TASK` in conjunction is a good practice, though not necessary.

# FLAG\_ACTIVITY\_NEW\_TASK

When we're trying to launch an activity from outside the activity's context, a FLAG\\_ACTIVITY\\_NEW\\_TASK is compulsory else a runtime exception would be thrown. Example scenarios are: launching from a service, invoking an activity from a notification click. If the activity instance is already on the task list when the flag is set, it will invoke the onNewIntent() method of that Activity. All the implementation stuff goes in that method.

# flatMap() vs map() RxJava

1. The map() operator maps from a concrete ValueA to a concrete ValueB (e.g. from an Int -> String, or User -> String).

2. flatMap() maps from a concrete ValueA to a Stream<ValueB>. If Stream<ValueB> emits multiple items, all of these will be eventually served to the original observer (e.g. they are being “flattened” to a single Observer). Since there’s no restrictions on the Stream<ValueB>, flatMap() is useful to introduce parallelism in executing tasks.

# flatMap(), switchMap(), concatMap()

1. flatMap() is used to split chain to multiple intermediary streams, and results of which are flattened to a single stream. Emissions of these intermediary streams are directly propagated to the main stream in any order.

2. switchMap() is similar to flatMap(), but whenever a new item is being emitted from a source, all the intermediary streams generated are terminated, and hence only the latest intermediary stream remains active.

3. concatMap() also works similar to flatMap(), with an exception such that intermediary streams are activated based on the order appearance.

# Flavors in Android

Customizing an app across different code bases can be tedious and error-prone. Therefore, using product flavors is ideal since you can create app variants with a single code base. You should also mention that Gradle makes it easy to create build-type variants.

# Flowable RxJava

Flowable – like an Observable, however it DOES support backpressure.

# fold vs reduce

1. fold takes an initial value, and the first invocation of the lambda you pass to it will receive that initial value and the first element of the collection as parameters.

listOf(1, 2, 3).fold(0) { sum, element -> sum + element }

The first call to the lambda will be with parameters 0 and 1.

Having the ability to pass in an initial value is useful if you have to provide some sort of default value or parameter for your operation.

2. reduce doesn't take an initial value, but instead starts with the first element of the collection as the accumulator (called sum in the following example)

listOf(1, 2, 3).reduce { sum, element -> sum + element }

The first call to the lambda here will be with parameters 1 and 2.

# foreground activity kill

The foreground activity, being the most important among the other states, is only killed or terminated as a last resort, especially if it is already consuming too much memory. When a memory paging state has been reach by a foreground activity, then it is killed so that the user interface can retain its responsiveness to the user.

# fragment

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

There are mainly 3 types of fragments:

Single Fragment

List Fragment

Fragment Transaction

# Fragment

Fragments are a part of an activity and they contribute there own UI to the activity they are embedded in. A single activity can contain multiple fragments. Fragments are reusable across activities.

The lifecycle methods of a Fragment are :

1. `onAttach(Activity)` : is called only once when it is attached with activity.

2. `onCreate(Bundle)` : it is used to initialise the fragment.

3. `onCreateView(LayoutInflater, ViewGroup, Bundle)` : creates and returns view hierarchy.

4. `onActivityCreated(Bundle)` : it is invoked after the completion of onCreate() method.

5. `onViewStateRestored(Bundle)` : it provides information to the fragment that all the saved state of fragment view hierarchy has been restored.

6. `onStart()` : makes the fragment visible.

7. `onResume()` : makes the fragment interactive.

8. `onPause()` : is called when fragment is no longer interactive.

9. `onStop()` : is called when fragment is no longer visible

10. `onDestroyView()` : it allows the fragment to clean up resources

11. `onDestroy()` : it allows the fragment to do final clean up of fragment state

12. `onDetach()` : it is called when the fragment is no longer associated with the activity

https://journaldev.nyc3.digitaloceanspaces.com/2016/07/fragment-lifecycle.png

# Fragment construction arguments

Construction arguments for a Fragment are passed via Bundle using the Fragment#setArgument(Bundle) method. The passed-in Bundle can then be retrieved through the Fragment#getArguments() method in the appropriate Fragment lifecycle method.

It is a common mistake to pass in data through a custom constructor. Non-default constructors on a Fragment are not advisable because the Fragment may be destroyed and recreated due to a configuration change (e.g. orientation change). Using #setArguments()/getArguments() ensures that when the Fragment needs to be recreated, the Bundle will be appropriately serialized/deserialized so that construction data is restored.

# fragment lifecycle

1. onAttach() The very first method to be called when the fragment has been associated with the activity. This method executes only once during the lifetime of a fragment.

2. onCreate() This method initializes the fragment by adding all the required attributes and components.

3. onCreateView() System calls this method to create the user interface of the fragment. The root of the fragment’s layout is returned as the View component by this method to draw the UI.

4. onActivityCreated() It indicates that the activity has been created in which the fragment exists. View hierarchy of the fragment also instantiated before this function call.

5. onStart() The system invokes this method to make the fragment visible on the user’s device.

6. onResume() This method is called to make the visible fragment interactive.

7. onPause() It indicates that the user is leaving the fragment. System calls this method to commit the changes made to the fragment.

8. onStop() Method to terminate the functioning and visibility of fragments from the user’s screen.

9. onDestroyView() System calls this method to clean up all kinds of resources as well as view hierarchy associated with the fragment.

10. onDestroy() It is called to perform the final clean-up of the fragment’s state and its lifecycle.

11. onDetach() The system executes this method to disassociate the fragment from its host activity.

# fragment vs activity

1. Activity is an application component that gives a user interface where the user can interact.

1-1.The fragment is only part of an activity, it basically contributes its UI to that activity.

2. Activity is not dependent on fragment.

2-1. Fragment is dependent on activity. It can’t exist independently.

3. we need to mention all activity it in the manifest.xml file.

3-1. Fragment is not required to mention in the manifest file

4. We can’t create multi-screen UI without using fragment in an activity.

4-1. After using multiple fragments in a single activity, we can create a multi-screen UI.

5. Activity can exist without a Fragment.

5-1. Fragment cannot be used without an Activity.

6. Creating a project using only Activity then it’s difficult to manage.

6-1. While Using fragments in the project, the project structure will be good and we can handle it easily.

7. Lifecycle methods are hosted by the OS. The activity has its own life cycle.

7-1. Lifecycle methods in fragments are hosted by hosting the activity.

8. Activity is not light weight.

8-1. The fragment is the lite weight.

# FrameLayout vs TableLayout

A FrameLayout stack up child views above each other with the last view added on the top. Though we can control the position of the children inside the FrameLayout using the layout\_gravity attribute. When the width and height of the FrameLayout are set to wrap\_content, the size of the FrameLayout equals the size of the largest child (plus padding). A TableLayout consists of TableRows. The children are arranged in the form of rows and columns.

# function extension

In Kotlin, we can add or delete method functionality using extensions, even without inheriting or altering them. Extensions are statistically resolved. It provides a callable function that may be invoked with a dot operation, rather than altering the existing class.

Function Extension - Kotlin allows users to specify a method outside of the main class via function extension.

# function vs method

A function is a named code block invoked from other locations within the source code. A method is a function associated with an object and can be invoked from other code with the dot notation.

# Future vs Callable

Java Callable interface: In Java5 callable interface was provided by the package java.util.concurrent. It is similar to the Runnable interface but it can return a result, and it can throw an Exception. It also provides a run() method for execution of a thread. Java Callable can return any object as it uses Generic.

public interface Callable<V>

Java Future interface: Java Future interface gives the result of a concurrent process. The Callable interface returns the object of java.util.concurrent.Future.

Java Future provides following methods for implementation:

1. cancel(boolean mayInterruptIfRunning): It is used to cancel the execution of the assigned task.

2. get(): It waits for the time if execution not completed and then retrieved the result.

3. isCancelled(): It returns the Boolean value as it returns true if the task was canceled before the completion.

4. isDone(): It returns true if the job is completed successfully else returns false.

# FutureTask

Java FutureTask class provides a base implementation of the Future interface. The result can only be obtained if the execution of one task is completed, and if the computation is not achieved then get method will be blocked. If the execution is completed, then it cannot be re-started and can't be canceled.

# Garbage collection

Garbage collection is a process of reclaiming the unused runtime objects. It is performed for memory management. In other words, we can say that It is the process of removing unused objects from the memory to free up space and make this space available for Java Virtual Machine. Due to garbage collection java gives 0 as output to a variable whose value is not set, i.e., the variable has been defined but not initialized. For this purpose, we were using free() function in the C language and delete() in C++. In Java, it is performed automatically. So, java provides better memory management.

# Garbage collection control

Garbage collection is managed by JVM. It is performed when there is not enough space in the memory and memory is running low. We can externally call the System.gc() for the garbage collection. However, it depends upon the JVM whether to perform it or not.

# Garbage Collector

Garbage Collector in Android has no compacting. This means the address of objects in the heap never changed after their creation. So garbage collection can be triggered when an allocation fails when an

1. OutOfMemoryError is about to be triggered,

2. When the size of the heap hits some soft limit, and

3. When a GC was explicitly requested.

# Garbage collector thread

Daemon thread

# generic collection pros

1. If we use the generic class, we don't need typecasting.

2. It is type-safe and checked at compile time.

3. Generic confirms the stability of the code by making it bug detectable at compile time.

# Glide

Glide, like Picasso, can load and display images from many sources, while also taking care of caching and keeping a low memory impact when doing image manipulations.

It provides animated GIF support and handles image loading/caching. Animated GIF support is currently not implemented in Picasso. Yes, images play a major role in making the UI of an App more interactive and user-friendly too. So, as an Android Developer, we should take care of using images in App. We should handle the different aspects of an image like the slow unresponsive image, memory issues, loading errors, and many more. If you are not handling these aspects in your application, then your app will make a bad impression on your users.

# Gradle

Gradle is a build system (open source) that is used to automate building, testing, deployment, etc. “Build.gradle” are scripts where one can automate the tasks. For example, the simple task to copy some files from one directory to another can be performed by Gradle build script before the actual build process happens.

Usage: Every Android project needs a Gradle for generating an apk from the .java and .xml files in the project. Simply put, a Gradle takes all the source files (java and XML) and applies appropriate tools, e.g., converts the java files into dex files and compresses all of them into a single file known as apk that is actually used.

# GroupBy RxJava

GroupBy operator allows us to classify the events in the input Observable into output categories.

# Guice

Guice is an open-source software framework for the Java platform released by Google under the Apache License. It provides support for dependency injection using annotations to configure Java objects.

Guice allows implementation classes to be bound programmatically to an interface, then injected into constructors, methods or fields using an @Inject annotation. When more than one implementation of the same interface is needed, the user can create custom annotations that identify an implementation, then use that annotation when injecting it.

# hashCode()

The hashCode() method returns a hash code value (an integer number).

The hashCode() method returns the same integer number if two keys (by calling equals() method) are identical.

However, it is possible that two hash code numbers can have different or the same keys.

If two objects do not produce an equal result by using the equals() method, then the hashcode() method will provide the different integer result for both the objects.

# hash-collision

Two different keys with the same hash value are known as hash-collision. Two separate entries will be kept in a single hash bucket to avoid the collision. There are two ways to avoid hash-collision.

Separate Chaining

Open Addressing

# HashMap vs Hashset

1. HashSet contains only values whereas HashMap includes the entry (key, value). HashSet can be iterated, but HashMap needs to convert into Set to be iterated.

2. HashSet implements Set interface whereas HashMap implements the Map interface

3. HashSet cannot have any duplicate value whereas HashMap can contain duplicate values with unique keys.

4. HashSet contains the only single number of null value whereas HashMap can hold a single null key with n number of null values.

# HashMap vs Hashtable

1) HashMap is not synchronized. / Hashtable is synchronized.

2) HashMap can contain one null key and multiple null values. / Hashtable cannot contain any null key or null value.

3) HashMap is not ?thread-safe,? so it is useful for non-threaded applications. / Hashtable is thread-safe, and it can be shared between various threads.

4) HashMap inherits the AbstractMap class. / Hashtable inherits the Dictionary class.

# HashMap vs TreeMap

1. HashMap maintains no order, but TreeMap maintains ascending order.

2. HashMap is implemented by hash table whereas TreeMap is implemented by a Tree structure.

3. HashMap can be sorted by Key or value whereas TreeMap can be sorted by Key.

4. HashMap may contain a null key with multiple null values whereas TreeMap cannot hold a null key but can have multiple null values.

# HashSet vs HashMap

1. HashSet contains only values whereas HashMap includes the entry (key, value). HashSet can be iterated, but HashMap needs to convert into Set to be iterated.

2. HashSet implements Set interface whereas HashMap implements the Map interface

3. HashSet cannot have any duplicate value whereas HashMap can contain duplicate values with unique keys.

4. HashSet contains the only single number of null value whereas HashMap can hold a single null key with n number of null values.

# HashSet vs TreeSet

The HashSet and TreeSet, both classes, implement Set interface. The differences between the both are listed below.

1. HashSet maintains no order whereas TreeSet maintains ascending order.

2. HashSet impended by hash table whereas TreeSet implemented by a Tree structure.

3. HashSet performs faster than TreeSet.

4. HashSet is backed by HashMap whereas TreeSet is backed by TreeMap.

# Hashtabe vs HashMap

1) HashMap is not synchronized. / Hashtable is synchronized.

2) HashMap can contain one null key and multiple null values. / Hashtable cannot contain any null key or null value.

3) HashMap is not ?thread-safe,? so it is useful for non-threaded applications. / Hashtable is thread-safe, and it can be shared between various threads.

4) HashMap inherits the AbstractMap class. / Hashtable inherits the Dictionary class.

# higher-order function in Kotlin

A higher-order function takes one or more functions as arguments or returns a function as its result. Higher-order functions are often used in conjunction with lambda expressions to create concise and powerful code.

# HOT to COLD transform

1. COLD to HOT

One way to make a Cold observable Hot is by using publish().connect(). publish() converts the Cold observable to a ConnectableObservable, which pretty much behaves like a Hot one. Once triggered with the .connect() operator, it’ll publish events regardless if there are any subscribers.

Another way to transform a Cold observable to a Hot one is by wrapping it with a Subject. The Subject subscribes to the Cold observable immediately and exposes itself as an Observable to future subscribers. Again, the work is performed regardless whether there are any subscribers … and on the other hand multiple subscribers to the Subject won’t trigger the initial work multiple times.

2. HOT to COLD

The first way of transforming (or rather “masking”) a Hot observable to Cold is by using the defer() operator. It defers the creation of the Hot observable altogether, so each new subscriber will trigger the work again (feature of a Cold observable).

Depending on the use-case the pattern mentioned above might be quite wasteful, so another strategy is using the replay().autoConnect(0) paradigm. The replay() operator will cache the values emitted by the Hot observable and re-emit them to future subscribers. autoConnect(0) returns an observable that can be triggered even when there are no subscribers to the underlaying Hot observable. The combination of both just replays cached values from the Hot observable as a Cold one.

# HOT vs COLD

Cold observables are created multiple times and each instance can be triggered on it’s own. Hot observables are like a “stream” of ongoing events – observers can come and go, but the stream is created ones and just goes on.

# immutable class

We can create an immutable class by defining a final class having all of its members as final.

# immutable String

The simple meaning of immutable is unmodifiable or unchangeable. In Java, String is immutable, i.e., once string object has been created, its value can't be changed.

# Immutable vs Mutable Variables

Immutable variables are also known as read-only variables. They are declared using the val keyword. Once these variables have been declared, we cannot change their values.

Mutable Variables - In a mutable variable, the value of the variable can be changed. We use the keyword “var” to declare such variables.

# Implicit Intent vs Explicit Intent

Explicit Intent:

An Explicit Intent is where you inform the system about which activity should handle this intent. Here target component is defined directly in the intent.

Implicit Intent:

An Implicit Intent permits you to declare the action you want to carry out. Further, the Android system will check which components are registered to handle that specific action based on intent data. Here target component is not defined in the intent.

# infix vs inline functions

[Inline functions] are used to save us memory overhead by preventing object allocations for the anonymous functions/lambda expressions called. Instead, it provides that functions body to the function that calls it at runtime. This increases the bytecode size slightly but saves us a lot of memory. [infix functions] on the other are used to call functions without parentheses or brackets. Doing so, the code looks much more like a natural language.

# Inheritance

Inheritance is a mechanism by which one object acquires all the properties and behavior of another object of another class. It is used for Code Reusability and Method Overriding. The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also. Inheritance represents the IS-A relationship which is also known as a parent-child relationship.

There are five types of inheritance in Java.

1. Single-level inheritance

2. Multi-level inheritance

3. Multiple Inheritance

4. Hierarchical Inheritance

5. Hybrid Inheritance

Multiple inheritance is not supported in Java through class.

# inheritance multiple

To reduce the complexity and simplify the language, multiple inheritance is not supported in java. Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since the compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have the same method or different, there will be a compile time error.

# Inheritance pros

1. Inheritance provides code reusability. The derived class does not need to redefine the method of base class unless it needs to provide the specific implementation of the method.

2. Runtime polymorphism cannot be achieved without using inheritance.

3. We can simulate the inheritance of classes with the real-time objects which makes OOPs more realistic.

4. Inheritance provides data hiding. The base class can hide some data from the derived class by making it private.

5. Method overriding cannot be achieved without inheritance. By method overriding, we can give a specific implementation of some basic method contained by the base class.

# init block

`init` is the initialiser block in Kotlin. It's executed once the primary constructor is instantiated. If you invoke a secondary constructor, then it works after the primary one as it is composed in the chain.

# initialize final blank variable

Yes, if it is not static, we can initialize it in the constructor. If it is static blank final variable, it can be initialized only in the static block.

# initializer vs constructor in Kotlin

A constructor is a special method invoked when an object is created. An initializer is a special method you can use to initialize an object before its first use. Both constructors and initializers are typically declared with the unit keyword.

# inline functions in Kotlin

Inline functions are expanded inline at the call site, meaning that the function code is copied and pasted into the body of the code where it is called. This can improve performance by eliminating the need for a function call while also increasing the readability of the code.

# inline vs infix functions

[Inline functions] are used to save us memory overhead by preventing object allocations for the anonymous functions/lambda expressions called. Instead, it provides that functions body to the function that calls it at runtime. This increases the bytecode size slightly but saves us a lot of memory. [infix functions] on the other are used to call functions without parentheses or brackets. Doing so, the code looks much more like a natural language.

# inner class anonymous

Anonymous inner classes are the classes that are automatically declared and instantiated within an expression. We cannot apply different access modifiers to them. Anonymous class cannot be static, and cannot define any static fields, method, or class. In other words, we can say that it a class without the name and can have only one object that is created by its definition.

# inner class in interface

An Interface can have a class. they are static implicitly.

# inner class vs nested class

Inner classes are non-static nested classes. In other words, we can say that inner classes are the part of nested classes.

# inner classes

1. Nested classes represent a special type of relationship that is it can access all the members (data members and methods) of the outer class including private.

2. Nested classes are used to develop a more readable and maintainable code because it logically groups classes and interfaces in one place only.

3. Code Optimization: It requires less code to write.

# inner classes cons

1. Inner classes increase the total number of classes used by the developer and therefore increases the workload of JVM since it has to perform some routine operations for those extra classes which result in slower performance.

2. IDEs provide less support to the inner classes as compare to the top level classes and therefore it annoys the developers while working with inner classes.

# inner classes types

1. Member Inner Class: A class created within class and outside method.

2. Anonymous Inner Class: A class created for implementing an interface or extending class. Its name is decided by the java compiler.

3. Local Inner Class: A class created within the method.

# instance method vs static(class) method

1. A method that is declared as static is known as the static method.

1-1. A method that is not declared as static is known as the instance method.

2. We don't need to create the objects to call the static methods.

2-1. The object is required to call the instance methods.

3. Non-static (instance) members cannot be accessed in the static context (static method, static block, and static nested class) directly.

3-1. Static and non-static variables both can be accessed in instance methods.

4. For example: public static int cube(int n){ return n\*n\*n;}

4-1. For example: public void msg(){...}.

# instanceOf

The instanceof in Java is also known as type comparison operator because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has a null value, it returns false.

# IntArray vs Array<Int>

Array<Int> is an Integer[] under the hood, while IntArray is an int[].

This means that when you put an Int in an Array<Int>, it will always be boxed (specifically, with an Integer.valueOf() call). In the case of IntArray, no boxing will occur, because it translates to a Java primitive array.

So no, we can't use them interchangeably.

# intent

An intent is a messaging object that is used to request an action from other components of an application. It can also be used to launch an activity, send SMS, send an email, display a web page, etc.

It shows notification messages to the user from within an Android-enabled device. It alerts the user of a particular state that occurred.

There are two types of intents in Android:

1. Implicit Intent- Used to invoke the system components.

2. Explicit Intent- Used to invoke the activity class.

# intent- clear back stack activity

1. The first approach is to use a `FLAG\_ACTIVITY\_CLEAR\_TOP` flag.

Intent intent= new Intent(ActivityA.this, ActivityB.class);

intent.setFlags(Intent.FLAG\_ACTIVITY\_CLEAR\_TOP);

startActivity(intent);

finish();

2.The second way is by using `FLAG\_ACTIVITY\_CLEAR\_TASK` and `FLAG\_ACTIVITY\_NEW\_TASK` in conjunction.

Intent intent= new Intent(ActivityA.this, ActivityB.class);

intent.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK | Intent.FLAG\_ACTIVITY\_CLEAR\_TASK);

startActivity(intent);

# intent filter

Because every component needs to indicate which intents they can respond to, intent filters are used to filter out intents that these components are willing to receive. One or more intent filters are possible, depending on the services and activities that is going to make use of it.

# Intent filters vs Intent

An Intent is an object passed to Context.startActivity(), Context.startService() or Activity.startActivityForResult() etc. to launch an activity or get an existing activity to do something new. On the other hand, an Intent filter describes the capability of the component(like activities, services, and broadcast receivers).

# Intent - Implicit vs Explicit

Explicit Intent:

An Explicit Intent is where you inform the system about which activity should handle this intent. Here target component is defined directly in the intent.

Implicit Intent:

An Implicit Intent permits you to declare the action you want to carry out. Further, the Android system will check which components are registered to handle that specific action based on intent data. Here target component is not defined in the intent.

# intent- phone call

1. To enable calling from the application we need to add the following permission in the manifest tag of AndroidManifest.xml

<uses-permission android:name=""android.permission.CALL\_PHONE"" />

In the MainActivity the following code invokes an action call to the given number represented as a string. The string is parsed as a URI.

String phone\_number = ""XXXXXXX"" // replace it with the number

Intent intent=new Intent(Intent.ACTION\_CALL,Uri.parse(""tel:""+phone number);

startActivity(intent);

2. To open a URL we need to add the following permission.

<uses-permission android:name=""android.permission.INTERNET"" />

The intent to view a URL is defined below.

Intent intent = new Intent(Intent.ACTION\_VIEW, Uri.parse(""https://www.journaldev.com/""));

startActivity(intent);

# intent setFlags(), addFlags()

When we're using setFlags, we're replacing the old flags with a new set of Flags. When we use addFlags, we're appending more flags.

# intent usages

Android Intents are used to

1. start an activity - startActivity(intent)

2. start a service - startService(intent)

3. deliver a broadcast - sendBroadcast(intent)

# Intent vs Intent filters

An Intent is an object passed to Context.startActivity(), Context.startService() or Activity.startActivityForResult() etc. to launch an activity or get an existing activity to do something new. On the other hand, an Intent filter describes the capability of the component(like activities, services, and broadcast receivers).

# IntentService vs Service

1. Service is the base class for Android services that can be extended to create any service. A class that directly extends Service runs on the main thread so it will block the UI (if there is one) and should therefore either be used only for short tasks or should make use of other threads for longer tasks.

2. IntentService is a subclass of Service that handles asynchronous requests (expressed as “Intents”) on demand. Clients send requests through startService(Intent) calls. The service is started as needed, handles each Intent in turn using a worker thread, and stops itself when it runs out of work. Writing an IntentService can be quite simple; just extend the IntentService class and override the onHandleIntent(Intent intent) method where you can manage all incoming requests.

# interface

The interface is a blueprint for a class that has static constants and abstract methods. It can be used to achieve full abstraction and multiple inheritance. It is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java. In other words, you can say that interfaces can have abstract methods and variables. Java Interface also represents the IS-A relationship. It cannot be instantiated just like the abstract class. However, we need to implement it to define its methods. Since Java 8, we can have the default, static, and private methods in an interface.

Can you declare an interface method static?

No, because methods of an interface are abstract by default, and we can not use static and abstract together.

Can the Interface be final?

No, because an interface needs to be implemented by the other class and if it is final, it can't be implemented by any class.

Can we define private and protected modifiers for the members in interfaces?

No, they are implicitly public.

# interface as final

No, we cannot declare an interface as final because the interface must be implemented by some class to provide its definition. Therefore, there is no sense to make an interface final. However, if you try to do so, the compiler will show an error.

# interface in class

An interface can be defined within the class. It is called a nested interface.

# interface- inner class

An Interface can have a class. they are static implicitly.

# interface Marker

A Marker interface can be defined as the interface which has no data member and member functions. For example, Serializable, Cloneable are marker interfaces.

# interface nested

An Interface that is declared inside the interface or class is known as the nested interface. It is static by default. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The external interface or class must refer to the nested interface. It can't be accessed directly. The nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.

# interface reference casting

An object reference can be cast to an interface reference when the object implements the referenced interface.

# interface vs abstract class

1. An abstract class can have a method body (non-abstract methods).

1-1. The interface has only abstract methods.

2. An abstract class can have instance variables.

2-1. An interface cannot have instance variables.

3. An abstract class can have the constructor.

3-1. The interface cannot have the constructor.

4. An abstract class can have static methods.

4-1. The interface cannot have static methods.

5. You can extend one abstract class.

5-1. You can implement multiple interfaces.

6. The abstract class can provide the implementation of the interface.

6-1. The Interface can't provide the implementation of the abstract class.

7. The abstract keyword is used to declare an abstract class.

7-1. The interface keyword is used to declare an interface.

8. An abstract class can extend another Java class and implement multiple Java interfaces.

8-1. An interface can extend another Java interface only.

9. An abstract class can be extended using keyword extends

9-1. An interface class can be implemented using keyword implements

10. A Java abstract class can have class members like private, protected, etc.

10-1. Members of a Java interface are public by default.

# inter-fragment communication Jetpack

There are two ways of implementing inter-fragment communication:

1. Using an interface: In this approach, you would create an interface in the parent fragment, and then have the child fragment implement that interface. The parent fragment can then call methods on the child fragment through the interface.

2. Using a ViewModel: With this approach, you would create a ViewModel that is shared between the parent and child fragments. The ViewModel can then be used to store data that needs to be shared between the fragments, and the fragments can observe changes to the ViewModel.

# interoperable

Kotlin is interoperable with Java because it uses JVM bytecode. It provides the facility to compile it directly to bytecode that helps to achieve faster compile-time and makes no difference between Java and Kotlin for JVM.

# inter-thread communication

1. The process of communication between synchronized threads is termed as inter-thread communication.

2. Inter-thread communication is used to avoid thread polling in Java.

3. The thread is paused running in its critical section, and another thread is allowed to enter (or lock) in the same critical section to be executed.

4. It can be obtained by wait(), notify(), and notifyAll() methods.

# IO filter

An I/O filter is an object that reads from one stream and writes to another, usually altering the data in some way as it is passed from one stream to another. Many Filter classes that allow a user to make a chain using multiple input streams. It generates a combined effect on several filters.

# IO hierarchy

OutputStream

---FileOutputStream

---ByteArrayOutputStream

---FilterOutputStream

------DataOutputStream

------BufferedOutputStream

------PrintStream

---PipedOutputStream

---ObjectOutputStream

InputStream

---FileInputStream

---ByteArrayInputStream

---FilterInputStream

------DataInputStream

------BufferedInputStream

------PushBackInputStream

---PipedInputStream

---ObjectInputStream

# IO stream

The stream is a sequence of data that flows from source to destination. It is composed of bytes. In Java, three streams are created for us automatically.

System.out: standard output stream

System.in: standard input stream

System.err: standard error stream

# Iterator vs Enumeration

1) The Iterator can traverse legacy and non-legacy elements. / Enumeration can traverse only legacy elements.

2) The Iterator is fail-fast. / Enumeration is not fail-fast.

3) The Iterator is slower than Enumeration. / Enumeration is faster than Iterator.

4) The Iterator can perform remove operation while traversing the collection. / The Enumeration can perform only traverse operation on the collection.

# Iterator vs ListIterator

Iterator traverses the elements in the forward direction only whereas ListIterator traverses the elements into forward and backward direction.

1) The Iterator traverses the elements in the forward direction only. / ListIterator traverses the elements in backward and forward directions both.

2) The Iterator can be used in List, Set, and Queue. / ListIterator can be used in List only.

3) The Iterator can only perform remove operation while traversing the collection. / ListIterator can perform ?add,? ?remove,? and ?set? operation while traversing the collection.

# java.util.regex

MatchResult Interface

Matcher class

Pattern class

PatternSyntaxException class

# JavaBean

JavaBean is a reusable software component written in the Java programming language, designed to be manipulated visually by a software development environment, like JBuilder or VisualAge for Java. t. A JavaBean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides the easy maintenance.

A bean encapsulates many objects into one object so that we can access this object from multiple places. Moreover, it provides the easy maintenance.

# Jetpack

Android Jetpack is a set of software components that help you accelerate your app development. These components provide a range of features, including a new set of tools and libraries, that make it easier to develop high-quality apps.

Android Jetpack is a set of libraries, tools and guidance to help make it easier to write high-quality, robust, and maintainable apps. One way it helps to reduce boilerplate code is by providing libraries that handle common tasks, such as navigation, lifecycle management, and data binding. This means that you don’t have to write as much code to handle these tasks yourself, which can help to reduce the amount of boilerplate in your app.

# Jetpack- Architecture Components

1. Room

2. WorkManager

3. Lifecycle

4. ViewModel

5. LiveData

6. Navigation

7. Paging

8. Data Binding

# Jetpack- Dagger

Dagger is a Dependency Injection framework that will generate a lot of boilerplate code for you to achieve the goal of Dependency injection in Android development. Not that one must have a good understanding of Dependency Injection before answering this question.

There are a number of reasons why you might want to use dagger to inject dependencies instead of injecting them manually. First, dagger can help to reduce the amount of boilerplate code that you need to write in order to inject dependencies. Second, dagger can help to improve the performance of your application by caching injected objects and avoiding repeated object creation. Finally, dagger can help to make your code more modular and easier to test by allowing you to easily swap out different implementations of injected dependencies.

# Jetpack- Data Binding

Data Binding library is a support library that provides the feature of binding UI components in an activity/fragment to the data sources of the application. The library carries out this binding task in a declarative format and not in a programmatic way. Below is an example to understand the working of this library accurately:

To find a TextView widget and bind it to the userName property of the ViewModel variable, the findViewById() method is called:

TextView textView = findViewById(R.id.sample\_text);

textView.setText(viewModel.getUserName());

After using the Data Binding library, the above code changes by using the assignment expression as follows:

<TextView

android:text=”@{viewmodel.userName}” />

Advantages of Data Binding Component:

1. Make code simpler and easy to maintain by removing UI frameworks called in the activity.

2. Allows classes and methods to observe changes in data

3. Allows to make objects and fill which works as collection observables.

# Jetpack- Data Binding library

Data Binding library is used to bind data to UI elements in Android applications. It eliminates the need for manual data handling in the application code, making the code simpler and more efficient. Data Binding library is an important part of Android Jetpack, as it helps to improve the performance of Android applications.

# Jetpack- Data Binding pros

1. Make code simpler and easy to maintain by removing UI frameworks called in the activity.

2. Allows classes and methods to observe changes in data

3. Allows to make objects and fill which works as collection observables.

# Jetpack- inter-fragment communication

There are two ways of implementing inter-fragment communication:

1. Using an interface: In this approach, you would create an interface in the parent fragment, and then have the child fragment implement that interface. The parent fragment can then call methods on the child fragment through the interface.

2. Using a ViewModel: With this approach, you would create a ViewModel that is shared between the parent and child fragments. The ViewModel can then be used to store data that needs to be shared between the fragments, and the fragments can observe changes to the ViewModel.

# Jetpack- Lifecycle

Lifecycle is a class that holds the information about the lifecycle state of a component (like an activity or a fragment) and allows other objects to observe this state.

Lifecycle uses two main enumerations to track the lifecycle status for its associated component:

1. Event

The lifecycle events that are dispatched from the framework and the Lifecycle class. These events map to the callback events in activities and fragments.

2. State

The current state of the component tracked by the Lifecycle object.

# Jetpack- Lifecycles component

The Lifecycles component is designed to help manage the lifecycle of an Android app. It provides a number of features that can help to automate tasks and manage memory usage. For example, you can use the Lifecycles component to automatically start and stop services based on the lifecycle of your app.

# Jetpack- LiveData

LiveData component is an observable data holder class i.e, the contained value can be observed. LiveData is a lifecycle-aware component and thus it performs its functions according to the lifecycle state of other application components. Further, if the observer’s lifecycle state is active i.e., either STARTED or RESUMED, only then LiveData updates the app component. LiveData always checks the observer’s state before making any update to ensure that the observer must be active to receive it. If the observer’s lifecycle state is destroyed, LiveData is capable of removing it, and thus it avoids memory leaks. It makes the task of data synchronization easier.

Advantages of LiveData component:

1. UI is updated as per the appropriate change in the data

2. It removes the stopped or destroyed activities which reduce the chance of app crash

3. No memory leaks as LiveData is a lifecycle-aware component.

# Jetpack- LiveData pros

1. UI is updated as per the appropriate change in the data

2. It removes the stopped or destroyed activities which reduce the chance of app crash

3. No memory leaks as LiveData is a lifecycle-aware component.

# Jetpack Navigation component

The Navigation component is a Jetpack library that helps you manage fragment transactions and back stack management in your app. It also provides a way to create a consistent navigation UI across your app. The benefits of using the Navigation component include a simpler and more robust way to manage fragment transactions, as well as a more consistent navigation experience for users.

# Jetpack- Paging Library

Paging Library makes it easier to load and display large datasets from a database or network asynchronously by providing a way to load data in small chunks (called “pages”). This way, only the data that is needed is loaded at any given time, which makes it more efficient and reduces the amount of time and resources required to load and display large datasets.

# Jetpack pros

1. Forms a recommended way for app architecture through its components

2. Eliminate boilerplate code

3. Simplify complex task

4. Provide backward compatibility as libraries like support are unbundled from Android API and are re-packaged to androidx.\* package

5. Inbuilt productivity feature of the Kotlin Integration

# Jetpack- Room

The requirement of a database in Android is fulfilled by SQLite from the very beginning. However, it comes with some severe drawbacks like not checking the queries at compile-time, it does not save plain-old-Java Objects(commonly referred to as POJOs). Developers also need to write a lot of boilerplate code to make the SQLite database work in the Android OS environment. The Room component comes into the picture as an SQLite Object Mapping Library which overcomes all the mentioned challenges. Room converts queries directly into objects, checks errors in queries at the compile-time, and is also capable of persisting the Java POJOs.

Moreover, it produces LiveData results/observables from the given query result. Because of this versatile nature of the Room component, Google officially supports and recommends developers to use it. The Room consists of the following sub-components:

1. Entity: It is the annotated class for which the Room creates a table within the database. The field of the class represents columns in the table.

2. DAO(Data Access Object): It is responsible for defining the methods to access the database and to perform operations.

3. Database: It is an abstract class that extends RoomDatabase class and it serves as the main access point to the underlying app’s relational data.

# Jetpack- ViewModel

ViewModel is one of the most critical classes of the Android Jetpack Architecture Component that support data for UI components. Its purpose is to hold and manage the UI-related data. Moreover, its main function is to maintain the integrity and allows data to be serviced during configuration changes like screen rotations. Any kind of configuration change in Android devices tends to recreate the whole activity of the application. It means the data will be lost if it has not been saved and restored properly from the activity which was destroyed. To avoid these issues, it is recommended to store all UI data in the ViewModel instead of an activity.

# Jetpack- Workmanager pros

WorkManager is an Android Jetpack library that allows you to schedule and manage background tasks in your app. It is designed to be used with tasks that are not time-critical, and it provides a number of features that make it a good choice for managing background work in Android apps. WorkManager is flexible, allowing you to specify constraints on when your tasks should run, and it is also able to intelligently schedule tasks based on the state of the app and the device. WorkManager is also able to persist tasks across device reboots, and it integrates with other Android Jetpack libraries to provide a complete solution for background work in Android apps.

# Job object kotlin

Job objects are the basic building blocks of coroutines. They define a coroutine’s lifecycle and provide a way to cancel it. CoroutineScope is used to define a scope for a coroutine, which determines its lifetime and other properties.

# JobSchedular

The JobSchedular API is used for scheduling different types of jobs against the framework that will be executed in your app’s own process. This allows your application to perform the given task while being considerate of the device’s battery at the cost of timing control.

The JobScheduler supports batch scheduling of jobs. The Android system can combine jobs for reducing battery consumption. JobManager automatically handles the network unreliability so it makes handling uploads easier.

Here is some example of the situation where you would use this job scheduler:

1. Tasks that should be done when the device is connected to a power supply.

2. Tasks that require a Wi-Fi connection or network access.

3. Tasks that should run on a regular basis as batch where the timing is not critical.

# join()

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task. Join method is overloaded in Thread class in the following ways.

public void join()throws InterruptedException

public void join(long milliseconds)throws InterruptedException

# JUnit

JUnit is a “Unit Testing” framework for Java Applications which is already included by default in android studio. It is an automation framework for Unit as well as UI Testing. It contains annotations such as @Test, @Before, @After, etc. Here we will be using only @Test annotation to keep the article easy to understand.

# kill Activity

We'll declare and assign a class instance of the FirstActivity to itself as shown below.

public class FirstActivity extends AppCompatActivity {

public static FirstActivity firstActivity;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

firstActivity=this;

}

}

We'll call finish() on the above instance of the FirstActivity to kill the activity from any other activity.

FirstActivity.firstActivity.finish()

# kill foreground activity

The foreground activity, being the most important among the other states, is only killed or terminated as a last resort, especially if it is already consuming too much memory. When a memory paging state has been reach by a foreground activity, then it is killed so that the user interface can retain its responsiveness to the user.

# Koin

Koin is a pragmatic and lightweight dependency injection framework for Kotlin developers.

Koin is a DSL, a light container and a pragmatic API

Koin provides a DSL to help your describe your app instead of annotate it or generate code for it. With its Kotlin DSL, Koin offers a smart functional API to achieve to prepare your dependency injection.

startKoin: means run a KoinApplication instance into the GlobalContext.

A Koin module will inject/combine for your application.

module - create a Koin Module

To describe your content in a module, you can use the following functions:

factory - provide a factory bean definition

single - provide a singleton bean definition (also aliased as bean)

get() - resolve a component dependency (also can use name, scope or parameters)

bind() - add type to bind for given bean definition

binds() - add types array for given bean definition

scope - define a logical group for scoped definition

scoped - provide a bean definition that will exists only in a scope

# Kotlin

Kotlin is the latest JVM programming language from the JetBrains. Google has made it the official language for Android Development along with Java.

# kotlin ""object""

To use the singleton pattern for our class we must use the keyword `object`

An `object` cannot have a constructor set. We can use the init block inside it though.

# kotlin argument type in constructor

By default, the constructor arguments are `val` unless explicitly set to `var`.

# kotlin Array vs List

The major difference from usage side is that Arrays have a fixed size while (Mutable)List can adjust their size dynamically. Moreover Array is mutable whereas List is not.

Furthermore kotlin.collections.List is an interface implemented among others by java.util.ArrayList. It's also extended by kotlin.collections.MutableListto be used when a collections that allows for item modification is needed.

On the jvm level Array is represented by arrays. List on the other hand is represented by java.util.List since there are no immutable collections equivalents available in Java.

# kotlin Companion Object

In some languages, such as Java, the static keyword is used to declare class members and utilise them without creating an object, i.e. by simply calling them by their class name. In Kotlin, there is nothing called the “static” keyword. So, if we want to achieve the functionality of static member functions, we use the companion objects. This is also referred to as Object Extension.

We must use the companion keyword in front of the object definition to construct a companion object.

# kotlin concurrency

Kotlin has built-in support for concurrency via coroutines. Coroutines are light-weight threads that can be used to improve the performance of concurrent code.

# kotlin constant

In Kotlin, if you want to create the local constants which are supposed to be used with in the class then you can create it like below:

val MY\_CONSTANT\_1 = ""Constants1""

// or

const val MY\_CONSTANT\_2 = ""Constants2""

Like val, variables defined with the const keyword are immutable. The difference here is that const is used for variables that are known at compile-time.

Also avoid using companion objects. Behind the hood, getter and setter instance methods are created for the fields to be accessible. Calling instance methods is technically more expensive than calling static methods. Instead define the constants in object:

object DbConstants {

const val TABLE\_USER\_ATTRIBUTE\_EMPID = ""\_id""

const val TABLE\_USER\_ATTRIBUTE\_DATA = ""data""

}

# kotlin constructor

Constructors in Kotlin are of two types:

\*\*Primary\*\* - These are defined in the class headers. They cannot hold any logic. There's only one primary constructor per class.

\*\*Secondary\*\* - They're defined in the class body. They must delegate to the primary constructor if it exists. They can hold logic. There can be more than one secondary constructors.

# kotlin limitations

The major limitations of Kotlin are its lack of support for operator overloading and variable arguments. Additionally, Kotlin does not have any built-in string interpolation or formatting features.

# kotlin List vs Array

The major difference from usage side is that Arrays have a fixed size while (Mutable)List can adjust their size dynamically. Moreover Array is mutable whereas List is not.

Furthermore kotlin.collections.List is an interface implemented among others by java.util.ArrayList. It's also extended by kotlin.collections.MutableListto be used when a collections that allows for item modification is needed.

On the jvm level Array is represented by arrays. List on the other hand is represented by java.util.List since there are no immutable collections equivalents available in Java.

# kotlin new

\*\*NO\*\*. Unlike Java, in Kotlin, new isn't a keyword. We can instantiate a class in the following way:

```

class A

var a = A()

val new = A()

# kotlin open and public

The keyword “open” refers to the term ""open for expansion"". The open annotation on a class is the polar opposite of the final annotation in Java: it allows others to inherit from it. By default, a class cannot be inherited in Kotlin. In Kotlin, an open method signifies that it can be overridden, whereas it cannot be by default. Instead, any methods in Java can be overridden by default.

In Kotlin, all the classes are final by default. If no visibility modifier is specified, the public is used by default, which means our declarations will be accessible everywhere inside the program.

# kotlin primitive types

At the language level, we cannot use the above-mentioned types. But the JVM bytecode that’s compiled does certainly have them.

# kotlin scope function

The Kotlin standard library includes numerous functions that aid in the execution of a block of code within the context of an object. When you use a lambda expression to call these functions on an object, temporary scope is created. These functions are referred to as Scope functions. The object of these functions can be accessed without knowing its name. Scope functions make code more clear, legible, and succinct, which are key qualities of the Kotlin programming language.

1. let:-

Context object: it

Return value: lambda result

The let function is frequently used for null safety calls. For null safety, use the safe call operator(?.) with ‘let'. It only runs the block with a non-null value.

2. apply:-

Context object: this

Return value: context object

“Apply these to the object,” as the name suggests. It can be used to operate on receiver object members, primarily to initialise them.

3. with:-

Context object: this

Return value: lambda result

When calling functions on context objects without supplying the lambda result, ‘with' is recommended.

4. run:-

Context object: this

Return value: lambda result

The ‘run' function is a combination of the ‘let' and ‘with' functions. When the object lambda involves both initialization and computation of the return value, this is the method to use. We can use run to make null safety calls as well as other calculations.

5. also:-

Context object: it

Return value: context object

It's used when we need to do additional operations after the object members have been initialised.

# kotlin singleton

To use the singleton pattern for our class we must use the keyword `object`

An `object` cannot have a constructor set. We can use the init block inside it though.

# kotlin static

\*\*NO\*\*. Kotlin doesn't have the static keyword. To create static method in our class we use the `companion object`. Following is the Java code:

```

class A {

public static int returnMe() { return 5; }

}

```

The equivalent Kotlin code would look like this:

```

class A {

companion object {

fun a() : Int = 5

}

}

```

To invoke this we simply do: `A.a()`.

# kotlin static method

1. Place the function in the companion object.

class Foo {

public static int a() { return 1; }

}

will become:

class Foo {

companion object {

fun a() : Int = 1

}

}

// to run

Foo.a();

2. Another way is to solve most of the needs for static functions with package-level functions. They are simply declared outside a class in a source code file. The package of a file can be specified at the beginning of a file with the package keyword. Under the hood these ""top-level"" or ""package"" functions are actually compiled into their own class. In the above example, the compiler would create a class FooPackage with all of the top-level properties and functions, and route all of your references to them appropriately.

Consider:

package foo

fun bar() = {}

usage:

import foo.bar

# kotlin string interpolation

String interpolation is used to evaluate string templates. We use the symbol $ to add variables inside a string.

```

val name = ""Journaldev.com""

val desc = ""$name now has Kotlin Interview Questions too. ${name.length}""

```

Using `{}` we can compute an expression too.

# kotlin switch - when

when is the equivalent of `switch` in `Kotlin`. The default statement in a when is represented using the else statement.

`when` statments have a default break statement in them.

# kotlin ternary operator

No. In Kotlin, we don't have a ternary operator like Java, but we can use the functionality of the ternary operator by using if-else or Elvis operator.

# kotlin visibility modifier

- public

- internal

- protected

- private

`public` is the default visibility modifier.

# KTX

KTX library is the only one among the foundation components which was introduced for the first time with the release of the Jetpack. Android KTX is a collection of Kotlin extensions that are designed to facilitate developers to remove boilerplate code as well as to write concise code while developing android applications with Kotlin language. Here KTX in the name stands for Kotlin Extensions. Below is an example of a piece of code without using and after using the Android KTX library:

Code snippet of SQLite without using KTX library:

db.beginTransaction()

try {

// insert data

db.setTransactionSuccessful()

}

finally {

db.endTransaction()

}

Above code after using KTX library:

db.transaction {

// insert data

}

# lambda expression

A lambda expression is an anonymous function that can concisely represent a function with a single parameter. Lambda expressions are often used in conjunction with higher-order functions, such as map and filter.

# sqlite

First, you must define the parameters the expression accepts using the parentheses operator. You can then provide an executable block of code within curly braces and use the arrow operator to indicate that this code is the body of the lambda expression.

val items = listOf(1, 2, 3, 4, 5)

items.fold(0, {

acc: Int, i: Int ->

print(""acc = $acc, i = $i, "")

val result = acc + i

println(""result = $result"")

result

})

To pass a lambda expression as an argument to a function, include it within parentheses after the function name and any necessary arguments. This will cause the code within that lambda expression to be executed whenever the function is called.

# lambda expression vs anonymous function

A lambda expression is a function that can be passed as an argument to another function. An anonymous function is a function that does not have a name and cannot be passed as an argument to another function. Thus, they’re actually opposites.

# lateinit

lateinit is an abbreviation for late initiation. If you don't want to initialize a variable in the constructor and instead want to do it later, and you can guarantee the initialization before using it, use the lateinit keyword to declare that variable. It won't start allocating memory until it's been initialized. Lateinit cannot be used for primitive type attributes like Int, Long, and so on. Because the lateinit variable will be initialized later, you cannot use val. When a lateinit property is accessed before it has been initialized, a special exception is thrown that explicitly identifies the property and the fact that it hasn't been initialized.

There are a few scenarios in which this is particularly useful, for example:

1. Variables that are initialized in lifecycle methods in Android;

2. Using Dagger for DI: injected class variables are initialized outside of the constructor and independently;

3. Setup for unit tests: in a @Before - annotated function, test environment variables are initialized;

4. Annotations in Spring Boot (for example, @Autowired)

# lateinit vs lazy

Both are used to delay the property initializations in Kotlin `lateinit` is a modifier used with var and is used to set the value to the var at a later point. `lazy` is a method or rather say lambda expression. It's set on a val only. The val would be created at runtime when it's required.

1. lazy, The main purpose is to delay the initialisation to a later point in time.

1-1. lateinit, The main purpose is to initialise an object only when it is used at a later point in time. Also, a single copy of the object is maintained throughout the program.

2. lazy, It's possible to initialise the object from anywhere in the program.

2-1. lateinit, Only the initializer lambda can be used to initialise it.

3. lazy, Multiple initializations are possible in this case.

3-1. lateinit, Only a single initialisation is possible in this case.

4. lazy, It's not thread-safe. In a multi-threaded system, it is up to the user to correctly initialise.

4-1. lateinit, Thread-safety is enabled by default, ensuring that the initializer is only called once.

5. lazy, It works only with var.

5-1. lateinit, It works only with val.

6. lazy, The isInitialized method is added to verify if the value has previously been initialised.

6-1. lateinit, It is impossible to uninitialize a property.

7. lazy, Properties of primitive types are not allowed

7-1. lateinit, Allowable on primitive type properties.

There are a few easy principles to follow when deciding whether to use lateinit or lazy initialisation for property initialization:

1. Use lateInit if properties are mutable (i.e., they may change later).

2. Use lateinit if properties are set externally (for example, if you need to pass in an external variable to set it). There is still a way to use lazy, but it isn't as obvious.

3. If they're only meant to be initialised once and shared by everybody, and they're more internally set (depending on a class variable), then lazy is the way to go. We could still use lateinit in a tactical sense, but utilising lazy initialisation would better encapsulate our initialization code.

# launch coroutine

Call the launch function on the CoroutineScope object to launch a coroutine, passing in the function you wish to execute. This will create a new coroutine and launch it immediately.

# launch mode

A “launch mode” is the way in which a new instance of an activity is to be associated with the current task.

Launch modes may be defined using one of two mechanisms:

1. Manifest file. When declaring an activity in a manifest file, you can specify how the activity should associate with tasks when it starts. Supported values include:

1-1. standard (default). Multiple instances of the activity class can be instantiated and multiple instances can be added to the same task or different tasks. This is the common mode for most of the activities.

1-2. singleTop. The difference from standard is, if an instance of the activity already exists at the top of the current task and the system routes the intent to this activity, no new instance will be created because it will fire off an onNewIntent() method instead of creating a new object.

1-3. singleTask. A new task will always be created and a new instance will be pushed to the task as the root. However, if any activity instance exists in any tasks, the system routes the intent to that activity instance through the onNewIntent() method call. In this mode, activity instances can be pushed to the same task. This mode is useful for activities that act as the entry points.

1-4. singleInstance. Same as singleTask, except that the no activities instance can be pushed into the same task of the singleInstance’s. Accordingly, the activity with launch mode is always in a single activity instance task. This is a very specialized mode and should only be used in applications that are implemented entirely as one activity.

2. Intent flags. Calls to startActivity() can include a flag in the Intent that declares if and how the new activity should be associated with the current task. Supported values include:

2-1. FLAG\_ACTIVITY\_NEW\_TASK. Same as singleTask value in Manifest file (see above).

2-2. FLAG\_ACTIVITY\_SINGLE\_TOP. Same as singleTop value in Manifest file (see above).

2-3. FLAG\_ACTIVITY\_CLEAR\_TOP. If the activity being started is already running in the current task, then instead of launching a new instance of that activity, all of the other activities on top of it are destroyed and this intent is delivered to the resumed instance of the activity (now on top), through onNewIntent(). There is no corresponding value in the Manifest file that produces this behavior.

# launch/join vs async/await

1. launch / join:-

The launch command is used to start and stop a coroutine. It's as though a new thread has been started. If the code inside the launch throws an exception, it's considered as an uncaught exception in a thread, which is typically written to stderr in backend JVM programs and crashes Android applications. Join is used to wait for the launched coroutine to complete before propagating its exception. A crashed child coroutine, on the other hand, cancels its parent with the matching exception.

2. async / await:-

The async keyword is used to initiate a coroutine that computes a result. You must use await on the result, which is represented by an instance of Deferred. Uncaught exceptions in async code are held in the resultant Deferred and are not transmitted anywhere else. They are not executed until processed.

# lazy

There are some classes whose object initialization is so time-consuming that it causes the entire class creation process to be delayed. Lazy initialisation helps in such problems. When we declare an object using lazy initialisation, the object is initialised only once when the object is used. If the object is not used throughout, the object is not initialised. This makes the code more efficient and faster.

# lazy vs lateinit

Both are used to delay the property initializations in Kotlin `lateinit` is a modifier used with var and is used to set the value to the var at a later point. `lazy` is a method or rather say lambda expression. It's set on a val only. The val would be created at runtime when it's required.

1. lazy, The main purpose is to delay the initialisation to a later point in time.

1-1. lateinit, The main purpose is to initialise an object only when it is used at a later point in time. Also, a single copy of the object is maintained throughout the program.

2. lazy, It's possible to initialise the object from anywhere in the program.

2-1. lateinit, Only the initializer lambda can be used to initialise it.

3. lazy, Multiple initializations are possible in this case.

3-1. lateinit, Only a single initialisation is possible in this case.

4. lazy, It's not thread-safe. In a multi-threaded system, it is up to the user to correctly initialise.

4-1. lateinit, Thread-safety is enabled by default, ensuring that the initializer is only called once.

5. lazy, It works only with var.

5-1. lateinit, It works only with val.

6. lazy, The isInitialized method is added to verify if the value has previously been initialised.

6-1. lateinit, It is impossible to uninitialize a property.

7. lazy, Properties of primitive types are not allowed

7-1. lateinit, Allowable on primitive type properties.

There are a few easy principles to follow when deciding whether to use lateinit or lazy initialisation for property initialization:

1. Use lateInit if properties are mutable (i.e., they may change later).

2. Use lateinit if properties are set externally (for example, if you need to pass in an external variable to set it). There is still a way to use lazy, but it isn't as obvious.

3. If they're only meant to be initialised once and shared by everybody, and they're more internally set (depending on a class variable), then lazy is the way to go. We could still use lateinit in a tactical sense, but utilising lazy initialisation would better encapsulate our initialization code.

# let:-

Context object: it

Return value: lambda result

The let function is frequently used for null safety calls. For null safety, use the safe call operator(?.) with ‘let'. It only runs the block with a non-null value.

# lifecycle AsyncTask, Activity

An AsyncTask is not tied to the life cycle of the Activity that contains it. So, for example, if you start an AsyncTask inside an Activity and the user rotates the device, the Activity will be destroyed (and a new Activity instance will be created) but the AsyncTask will not die but instead goes on living until it completes.

Then, when the AsyncTask does complete, rather than updating the UI of the new Activity, it updates the former instance of the Activity (i.e., the one in which it was created but that is not displayed anymore!). This can lead to an Exception (of the type java.lang.IllegalArgumentException: View not attached to window manager if you use, for instance, findViewById to retrieve a view inside the Activity).

There’s also the potential for this to result in a memory leak since the AsyncTask maintains a reference to the Activty, which prevents the Activity from being garbage collected as long as the AsyncTask remains alive.

For these reasons, using AsyncTasks for long-running background tasks is generally a bad idea . Rather, for long-running background tasks, a different mechanism (such as a service) should be employed.

# Lifecycle Jetpack

Lifecycle is a class that holds the information about the lifecycle state of a component (like an activity or a fragment) and allows other objects to observe this state.

Lifecycle uses two main enumerations to track the lifecycle status for its associated component:

1. Event

The lifecycle events that are dispatched from the framework and the Lifecycle class. These events map to the callback events in activities and fragments.

2. State

The current state of the component tracked by the Lifecycle object.

# Lifecycle of Activty

1. OnCreate(): It is called when activity is created. Using this, the views are created and data is collected from bundles.

2. OnStart(): It is called if the activity is becoming visible to the user. It may be succeeded by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden.

3. OnResume(): It is called when the activity will start an interaction with the user.

4. OnPause(): This is called when the activity is moving to the background but hasn’t been killed yet.

5. OnStop(): This is called when an activity is no longer visible to the user.

6. OnDestroy(): This is called when the activity is finished or destroyed.

7. OnRestart(): This is called after the activity has been stopped, prior to it being started again.

https://s3.ap-south-1.amazonaws.com/myinterviewtrainer-domestic/public\_assets/assets/000/000/441/original/Life\_Cycle\_of\_Android.png

Differentiate:

onCreate() is the first method that’s invoked when an activity is launched for the first time. onStart() is invoked after onCreate() has completed it’s task. onResume() is called after onStart() has completed. When an activity leaves its foreground (probably for a smaller duration such as standby/sleep) onPause() is invoked followed by onStop()(when the activity is not visible. eg. some other application is launched). onDestroy() is called when the activity or application is killed.

Essentially the lifecycle methods are divided into three layers of duration :

1. onCreate() and onDestroy() are present during the entire duration of the activity

2. onStart() and onStop() are present while the activity is visible

3. onResume() and onPause() are present while the activity is in foreground

# lifecycle of service

- `onStartCommand()` : This method is called when startService() is invoked. Once this method executes, the service is started and can run in the background indefinitely. This method is not needed if the service is defined as a bounded service. The service will run indefinitely in the background when this method is defined. We'll have a stop the service ourselves

- `onBind()` This method needs to be overridden when the service is defined as a bounded service. This method gets called when bindService() is invoked. In this method, we must provide an interface that clients use to communicate with the service, by returning an IBinder. We should always implement this method, but if you don’t want to allow binding, then you should return null

- `onCreate()` : This method is called while the service is first created. Here all the service initialization is done

- `onDestroy()` : The system calls this method when the service is no longer used and is being destroyed. All the resources, receivers, listeners clean up are done here

# Lifecycles component Jetpack

The Lifecycles component is designed to help manage the lifecycle of an Android app. It provides a number of features that can help to automate tasks and manage memory usage. For example, you can use the Lifecycles component to automatically start and stop services based on the lifecycle of your app.

# lifetime activity

1. Entire lifetime – activity happens between onCreate and onDestroy

2. Visible lifetime – activity happens between onStart and onStop

3. Foreground lifetime – activity happens between onResume and onPause

# major limitations of Kotlin

The major limitations of Kotlin are its lack of support for operator overloading and variable arguments. Additionally, Kotlin does not have any built-in string interpolation or formatting features.

# LinearLayout, RelativeLayout, AbsoluteLayout

LinearLayout arranges its children in a single row or single column one after the other.

RelativeLayout arranges it's children in positions relative to each other or relative to parent depending upon the LayoutParams defined for each view.

AbsoluteLayout needs the exact positions of the x and y coordinates of the view to position it. Though this is deprecated now.

# LinkedList vs ArrayList

1) ArrayList uses a dynamic array. / LinkedList uses a doubly linked list.

2) ArrayList is not efficient for manipulation because too much is required. / LinkedList is efficient for manipulation.

3) ArrayList is better to store and fetch data. / LinkedList is better to manipulate data.

4) ArrayList provides random access. / LinkedList does not provide random access.

5) ArrayList takes less memory overhead as it stores only object. / LinkedList takes more memory overhead, as it stores the object as well as the address of that object.

# List

List interface extends the Collection interface, and it is an ordered collection of objects. It contains duplicate elements. It also allows random access of elements.

# List vs Array kotlin

The major difference from usage side is that Arrays have a fixed size while (Mutable)List can adjust their size dynamically. Moreover Array is mutable whereas List is not.

Furthermore kotlin.collections.List is an interface implemented among others by java.util.ArrayList. It's also extended by kotlin.collections.MutableListto be used when a collections that allows for item modification is needed.

On the jvm level Array is represented by arrays. List on the other hand is represented by java.util.List since there are no immutable collections equivalents available in Java.

# List vs Set

The List and Set both extend the collection interface. However, there are some differences between the both which are listed below.

1. The List can contain duplicate elements whereas Set includes unique items.

2. The List is an ordered collection which maintains the insertion order whereas Set is an unordered collection which does not preserve the insertion order.

3. The List interface contains a single legacy class which is Vector class whereas Set interface does not have any legacy class.

4. The List interface can allow n number of null values whereas Set interface only allows a single null value.

# ListIterator vs Iterator

Iterator traverses the elements in the forward direction only whereas ListIterator traverses the elements into forward and backward direction.

1) The Iterator traverses the elements in the forward direction only. / ListIterator traverses the elements in backward and forward directions both.

2) The Iterator can be used in List, Set, and Queue. / ListIterator can be used in List only.

3) The Iterator can only perform remove operation while traversing the collection. / ListIterator can perform ?add,? ?remove,? and ?set? operation while traversing the collection.

# ListView vs RecyclerView

- A RecyclerView recycles and reuses cells when scrolling. This is a default behaviour. It's possible to implement the same in a ListView too but we need to implement a ViewHolder there

- A RecyclerView decouples list from its container so we can put list items easily at run time in the different containers (linearLayout, gridLayout) by setting LayoutManager

- Animations of RecyclerView items are decoupled and delegated to `ItemAnimator`

# LiveData

LiveData component is an observable data holder class i.e, the contained value can be observed. LiveData is a lifecycle-aware component and thus it performs its functions according to the lifecycle state of other application components. Further, if the observer’s lifecycle state is active i.e., either STARTED or RESUMED, only then LiveData updates the app component. LiveData always checks the observer’s state before making any update to ensure that the observer must be active to receive it. If the observer’s lifecycle state is destroyed, LiveData is capable of removing it, and thus it avoids memory leaks. It makes the task of data synchronization easier.

Advantages of LiveData component:

1. UI is updated as per the appropriate change in the data

2. It removes the stopped or destroyed activities which reduce the chance of app crash

3. No memory leaks as LiveData is a lifecycle-aware component.

# LiveData pros

1. UI is updated as per the appropriate change in the data

2. It removes the stopped or destroyed activities which reduce the chance of app crash

3. No memory leaks as LiveData is a lifecycle-aware component.

# LiveData vs Observables

1. LiveData is an observable data holder class. It is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.

2. Observables, on the other hand, are not lifecycle-aware. This means they will continue to emit data even if the observer is no longer in an active state, such as when an activity has been destroyed. This can lead to memory leaks if the observer is not unregistered when no longer needed.

# Locale

A Locale object represents a specific geographical, political, or cultural region. This object can be used to get the locale-specific information such as country name, language, variant, etc.

# lock interface

The java.util.concurrent.locks.Lock interface is used as the synchronization mechanism. It works similar to the synchronized block. There are a few differences between the lock and synchronized block that are given below.

1. Lock interface provides the guarantee of sequence in which the waiting thread will be given the access, whereas the synchronized block doesn't guarantee it.

2. Lock interface provides the option of timeout if the lock is not granted whereas the synchronized block doesn't provide that.

3. The methods of Lock interface, i.e., Lock() and Unlock() can be called in different methods whereas single synchronized block must be fully contained in a single method.

# Map

Map interface: A Map (java.util.Map) represents a key, value pair storage of elements. Map interface does not implement the Collection interface. It can only contain a unique key but can have duplicate elements. There are two interfaces which implement Map in java that are Map interface and Sorted Map.

# Map RxJava

The map operator transforms items emitted by an Observable by applying a function to each item.

# Map vs Set

1. Set contains values only whereas Map contains key and values both.

2. Set contains unique values whereas Map can contain unique Keys with duplicate values.

3. Set holds a single number of null value whereas Map can include a single null key with n number of null values.

# map() vs flatMap() RxJava

1. The map() operator maps from a concrete ValueA to a concrete ValueB (e.g. from an Int -> String, or User -> String).

2. flatMap() maps from a concrete ValueA to a Stream<ValueB>. If Stream<ValueB> emits multiple items, all of these will be eventually served to the original observer (e.g. they are being “flattened” to a single Observer). Since there’s no restrictions on the Stream<ValueB>, flatMap() is useful to introduce parallelism in executing tasks.

# Marble Diagram

It is a graphical representation of how the RxJava operators work. In most of the cases, RxJava has a source stream, operator, and resulting stream. Each of the streams is represented by a timeline with all emissions, known as marbles and terminating events.

# Marker interface

A Marker interface can be defined as the interface which has no data member and member functions. For example, Serializable, Cloneable are marker interfaces.

# Maybe RxJava

Maybe – has 3 options – returns a value successfully (onSuccess(value)), completes successfully without any value (onComplete()) or errors out (onError(throwable)).

# memory leak RxJava

As a general good practice in programming, one must clean-up the used resources after they’re no longer needed. In the case of RxJava this means disposing your Disposables correctly. A common pattern is to keep adding all long-running operations from a screen in a CompositeDisposable and ensuring that’s clean-up when the screen is gone.

# metacharacters

Metacharacters have the special meaning to the regular expression engine. The metacharacters are ^, $, ., \*, +, etc. The regular expression engine does not consider them as the regular characters. To enable the regular expression engine treating the metacharacters as ordinary characters, we need to escape the metacharacters with the backslash.

# method vs function

A function is a named code block invoked from other locations within the source code. A method is a function associated with an object and can be invoked from other code with the dot notation.

# Moshi

Moshi is a modern JSON library for Android, Java and Kotlin. It makes it easy to parse JSON into Java and Kotlin classes

# multiple errors RxJava

All unhandled errors (via any of the error handling operators) are propagated downstream. A chain can have only one terminal error event (e.g. one call to onError(throwable)), so the first unhandled error will terminate the stream. In case there’s other undelivered exceptions.

# multiple inheritance

To reduce the complexity and simplify the language, multiple inheritance is not supported in java. Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since the compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have the same method or different, there will be a compile time error.

# multiple observeOn()

Each observeOn() switches the Scheduler (thread) on which all subsequent operators will be executed on. Complex RxJava streams can absolutely benefit from multiple observeOn() operators.

# multiple subscribeOn()

Only the very first subscribeOn() in the chain has the desired effect, all subsequent ones would not have any effect, apart from potentially wasting system resources (depending on the Scheduler specified).

# multithreading

Multithreading is a process of executing multiple threads simultaneously. Multithreading is used to obtain the multitasking. It consumes less memory and gives the fast and efficient performance.

pros:

1. Threads share the same address space.

2. The thread is lightweight.

3. The cost of communication between the processes is low.

# multithreading pros

1. Multithreading allows an application/program to be always reactive for input, even already running with some background tasks

2. Multithreading allows the faster execution of tasks, as threads execute independently.

3. Multithreading provides better utilization of cache memory as threads share the common memory resources.

4. Multithreading reduces the number of the required server as one server can execute multiple threads at a time.

# Mutable vs Immutable Variables

Immutable variables are also known as read-only variables. They are declared using the val keyword. Once these variables have been declared, we cannot change their values.

Mutable Variables - In a mutable variable, the value of the variable can be changed. We use the keyword “var” to declare such variables.

# MVC

The MVC pattern suggests splitting the code into 3 components. While creating the class/file of the application, the developer must categorize it into one of the following three layers:

1. Model: This component stores the application data. It has no knowledge about the interface. The model is responsible for handling the domain logic(real-world business rules) and communication with the database and network layers.

2. View: It is the UI(User Interface) layer that holds components that are visible on the screen. Moreover, it provides the visualization of the data stored in the Model and offers interaction to the user.

3. Controller: This component establishes the relationship between the View and the Model. It contains the core application logic and gets informed of the user’s response and updates the Model as per the need.

# MVC pros/cons

pros:

1. No business logic in UI

2. Scalable

3. Easy to maintain, test, upgrade

4. Friendly development process

5. Great for complex application

6. Suitable for team base work

cons:

1. Doesn't scale. Separates UI, but not model

2. Controller often grows too big

3. Violates Single Responsibility (Controller responsible for almost everything), Interface Segregation SOLID principles

# MVC vs MVP vs MVVM

1. MVC is one of the oldest software architecture.

1-1. MVP was developed as the second iteration of software architecture which is advanced from MVC.

1-2. MVVM is Industry-recognized architecture pattern for applications.

2. MVC, UI(View) and data-access mechanism(Model) are tightly coupled.

2-1. MVP resolves the problem of having a dependent View by using Presenter as a communication channel between Model and View.

2-2. MVVM is more event-driven as it uses data binding and thus makes easy separation of core business logic from the View.

3. MVC, Controller and View exist with the one-to-many relationship. One Controller can select a different view based upon the required operation.

3-1. MVP, A one-to-one relationship exists between Presenter and View as one Presenter class manages one View at a time.

3-2. MVVM, Multiple View can be mapped with a single ViewModel and thus, the one-to-many relationship exists between View and ViewModel.

4. MVC, The View has no knowledge about the Controller.

4-1. MVP, The View has references to the Presenter.

4-2. MVVM, The View has references to the ViewModel

5. MVC, Difficult to make changes and modify the app features as the code layers are tightly coupled.

5-1. MVP, Code layers are loosely coupled and thus it is easy to carry out modifications/changes in the application code.

5-2. MVVM, Easy to make changes in the application. However, if data binding logic is too complex, it will be a little harder to debug the application.

6. MVC, User Inputs are handled by the Controller.

6-1. MVP, The View is the entry point to the Application.

6-2. MVVM, The View takes the input from the user and acts as the entry point of the application.

7. MVC is ideal for small scale projects only.

7-1. MVP is ideal for simple and complex applications.

7-2. MVVM is not ideal for small scale projects.

8. MVC, Limited support to Unit testing.

8-1. MVP, Easy to carry out Unit testing but a tight bond of View and Presenter can make it slightly difficult.

8-2. MVVM, Unit testability is highest in this architecture.

9. MVC has a high dependency on Android APIs.

9-1. MVP has a low dependency on the Android APIs.

9-2. MVVM Has low or no dependency on the Android APIs.

10. MVC does not follow the modular and single responsibility principle.

10-1. MVP follows modular and single responsibility principle.

10-2. MVVM follows modular and single responsibility principle.

# MVP

MVP pattern overcomes the challenges of MVC and provides an easy way to structure the project codes. The reason why MVP is widely accepted is that it provides modularity, testability, and a more clean and maintainable codebase. It is composed of the following three components:

1. Model: Layer for storing data. It is responsible for handling the domain logic(real-world business rules) and communication with the database and network layers.

2. View: UI(User Interface) layer. It provides the visualization of the data and keep a track of the user’s action in order to notify the Presenter.

3. Presenter: Fetch the data from the model and applies the UI logic to decide what to display. It manages the state of the View and takes actions according to the user’s input notification from the View.

# MVP pros/cons

pros:

1. Complex Tasks split into simpler tasks

2. Smaller objects, less bugs, easier to debug

3. Testable

cons:

1. Model can't be resued, tied to specific use case

2. View and Presenter are tied to data objects since they share the same type of object with the model

# MVVM

Model — View — ViewModel (MVVM) is the industry-recognized software architecture pattern that overcomes all drawbacks of MVP and MVC design patterns. MVVM suggests separating the data presentation logic(Views or UI) from the core business logic part of the application.

The separate code layers of MVVM are:

1. Model: This layer is responsible for the abstraction of the data sources. Model and ViewModel work together to get and save the data.

2. View: The purpose of this layer is to inform the ViewModel about the user’s action. This layer observes the ViewModel and does not contain any kind of application logic.

3. ViewModel: It exposes those data streams which are relevant to the View. Moreover, it serves as a link between the Model and the View.

MVVM pattern has some similarities with the MVP(Model — View — Presenter) design pattern as the Presenter role is played by ViewModel. However, the drawbacks of the MVP pattern has been solved by MVVM in the following ways:

1. ViewModel does not hold any kind of reference to the View.

2. Many to-1 relationships exist between View and ViewModel.

3. No triggering methods to update the View.

There are 2 ways to implement MVVM design pattern in Android projects:

1. Using the DataBinding library released by Google

2. Using any tool like RxJava for DataBinding.

# MVVM over MVC

I think that MVVM is a better choice for developing Android apps for a few reasons.

First, it allows for a more separation of concerns between the different parts of the app, which can make development and maintenance simpler.

Second, it can make it easier to bind data to the UI, since the ViewModel can act as a mediator between the data and the View. Finally, it can help to improve performance, since the ViewModel can handle tasks that would otherwise need to be done on the UI thread.

# MVVM pros/cons

pros:

1. MVVM facilitates easier parallel development of a UI and the building blocks that power it.

2. MVVM abstracts the View and thus reduces the quantity of business logic (or glue) required in the code behind it.

3. The ViewModel can be easier to unit test than in the case of event-driven code.

4. The ViewModel (being more Model than View) can be tested without concerns of UI automation and interaction.

cons:

1. For simpler UIs, MVVM can be overkill.

2. While data bindings can be declarative and nice to work with, they can be harder to debug than imperative code where we simply set breakpoints.

3. Data bindings in nontrivial applications can create a lot of bookkeeping. We also don’t want to end up in a situation where bindings are heavier than the objects being bound to.

4. In larger applications, it can be more difficult to design the ViewModel up front to get the necessary amount of generalization.

# native method

A native method is a method that is implemented in a language other than Java. Natives methods are sometimes also referred to as foreign methods.

# Navigation component Jetpack

The Navigation component is a Jetpack library that helps you manage fragment transactions and back stack management in your app. It also provides a way to create a consistent navigation UI across your app. The benefits of using the Navigation component include a simpler and more robust way to manage fragment transactions, as well as a more consistent navigation experience for users.

# NDK

NDK stands for Native Development Kit. By using NDK, you can develop a part of an app using native language such as C/C++ to boost the performance.

The NDK is a tool that allows you to program in C/C++ for Android devices. It provides platform libraries one can use to manage native activities and access physical device components, such as sensors and touch input. NDK can be useful for cases in which you need to do one or more of the following:

1. Squeeze extra performance out of a device to achieve low latency or run computationally intensive applications, such as games or physics simulations.

2. Reuse your own or other developers’ C or C++ libraries.

# nested class

The nested class can be defined as the class which is defined inside another class or interface. We use the nested class to logically group classes and interfaces in one place so that it can be more readable and maintainable. A nested class can access all the data members of the outer class including private data members and methods.

There are two types of nested classes, static nested class, and non-static nested class. The non-static nested class can also be called as inner-class

# nested class vs inner class

Inner classes are non-static nested classes. In other words, we can say that inner classes are the part of nested classes.

# nested interface

An Interface that is declared inside the interface or class is known as the nested interface. It is static by default. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The external interface or class must refer to the nested interface. It can't be accessed directly. The nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.

# new kotlin

\*\*NO\*\*. Unlike Java, in Kotlin, new isn't a keyword. We can instantiate a class in the following way:

```

class A

var a = A()

val new = A()

# nine-patch image

In general, a Nine-patch image allows resizing that can be used as background or other image size requirements for the target device. The Nine-patch refers to the way you can resize the image: 4 corners that are unscaled, 4 edges that are scaled in 1 axis, and the middle one that can be scaled into both axes.

# Non-Blocking

An algorithm is considered to be non-blocking if threads competing for resources do not have execution postponed due to mutual exclusion of protecting the resource.

# Not Null Assertion Operator ( !! )

Not Null Assertion Operator ( !! ) - If the value is null, the not null assertion (!!) operator changes it to a non-null type and throws an exception.

Anyone who wants a NullPointerException can ask for it explicitly with this operator.

# not null assertion operator vs safe call operator (!! Vs ?.)

!! is used to force unwrap the nullable type to get the value. If the value returned is a null, it would lead to a runtime crash. Hence a !! operator should be only used when you’re absolutely sure that the value won’t be null at all. Otherwise, you’ll get the dreaded null pointer exception. On the other hand, a ?. is an Elvis Operator that does a safe call. We can use the lambda expression let on the nullable value to unwrap safely as shown below.

# notify() vs notifyAll()

The notify() is used to unblock one waiting thread whereas notifyAll() method is used to unblock all the threads in waiting state.

# null check(!!) vs safe calls(?.)

The safe call operator i.e. ?. is used to check if the variable's value is null or not. If it is null, then null will be returned otherwise it will return the desired value.

If you want to throw NullPointerException when the variable's value is null, you can use the null check or !! Operator.

# null exceptions in Kotlin

In Kotlin, Elvis Operator is used to handling null expectations.

# null safety ?

Kotlin's type system aims to eradicate null references from the code. If a program throws NullPointerExceptions at runtime it might result in application failure or system crashes. If the Kotlin compiler finds a null reference it throws a NullPointerException.

The Kotlin type system distinguishes between references that can hold null (nullable references) and those that cannot (non-null references). Null cannot be stored in a String variable. We get a compiler error if we try to assign null to the variable.

var a: String = ""interview""

a = null // results in compilation error

If we want the above string to be able to hold null value as well, we can declare it of type nullable using the ‘?’ operator after the String keyword as follows :

var a: String? = ""interview""

a = null // no compilation error

Kotlin provides Safe Call (?.), Elvis (?:) and Not Null Assertion (!!) operators which define what needs to be done in case of a null encounter. This makes the code more reliable and less prone to errors. Thus, Kotlin enforces null safety by having nullable, non-nullable type variables and the different operators to tackle null encounters.

# Null Safety ? Vs Nullable Types

Kotlin puts a lot of weight behind null safety which is an approach to prevent the dreaded Null Pointer Exceptions by using nullable types which are like String?, Int?, Float? etc. These act as a wrapper type and can hold null values. A nullable value cannot be added to another nullable or basic type of value. To retrieve the basic types we need to use safe calls that unwrap the Nullable Types. If on unwrapping, the value is null we can choose to ignore or use a default value instead. The Elvis Operator is used to safely unwrap the value from the Nullable. It’s represented as ?: over the nullable type. The value on the right hand side would be used if the nullable type holds a null.

# Nullable Types vs Null Safety ?

Kotlin puts a lot of weight behind null safety which is an approach to prevent the dreaded Null Pointer Exceptions by using nullable types which are like String?, Int?, Float? etc. These act as a wrapper type and can hold null values. A nullable value cannot be added to another nullable or basic type of value. To retrieve the basic types we need to use safe calls that unwrap the Nullable Types. If on unwrapping, the value is null we can choose to ignore or use a default value instead. The Elvis Operator is used to safely unwrap the value from the Nullable. It’s represented as ?: over the nullable type. The value on the right hand side would be used if the nullable type holds a null.

# object

The Object is the real-time entity having some state and behavior. In Java, Object is an instance of the class having the instance variables as the state of the object and the methods as the behavior of the object. The object of a class can be created by using the new keyword.

# object clone

The object cloning is used to create the exact copy of an object. The clone() method of the Object class is used to clone an object. The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates CloneNotSupportedException.

# object cloning

The object cloning is a way to create an exact copy of an object. The clone() method of the Object class is used to clone an object. The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates CloneNotSupportedException. The clone() method is defined in the Object class. The syntax of the clone() method is as follows:

protected Object clone() throws CloneNotSupportedException

# object cloning pros/cons

pros:

1. You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.

2. It is the easiest and most efficient way of copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.

3. Clone() is the fastest way to copy the array.

cons:

1. To use the Object.clone() method, we have to change many syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone(), etc.

2. We have to implement the Cloneable interface while it does not have any methods in it. We have to use it to tell the JVM that we can perform a clone() on our object.

3. Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.

4. Object.clone() does not invoke any constructor, so we do not have any control over object construction.

5. If you want to write a clone method in a child class, then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.

6. Object.clone() supports only shallow copying, but we will need to override it if we need deep cloning.

# object copy constructor

There is no copy constructor in java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

1. By constructor

2. By assigning the values of one object into another

3. By clone() method of Object class

# object reference

All object references are initialized to null in Java.

# object reference casting interface

An object reference can be cast to an interface reference when the object implements the referenced interface.

# object unreference

1. By nulling the reference

2. By assigning a reference to another

3. By anonymous object etc.

# object vs class

A class is a template for creating objects. An object is an instance of a class. Classes can contain properties and methods, while objects contain only data.

# object-based vs object-oriented

1. Object-oriented languages follow all the concepts of OOPs whereas, the object-based language doesn't follow all the concepts of OOPs like inheritance and polymorphism.

2. Object-oriented languages do not have the inbuilt objects whereas Object-based languages have the inbuilt objects, for example, JavaScript has window object.

3. Examples of object-oriented programming are Java, C#, Smalltalk, etc. whereas the examples of object-based languages are JavaScript, VBScript, etc.

# Object-Oriented

Object-Oriented: Java follows the object-oriented paradigm which allows us to maintain our code as the combination of different type of objects that incorporates both data and behavior.

It is a programming paradigm based on objects having data and methods defined in the class to which it belongs. Object-oriented paradigm aims to incorporate the advantages of modularity and reusability. Objects are the instances of classes which interacts with one another to design applications and programs. There are the following features of the object-oriented paradigm.

1. Follows the bottom-up approach in program design.

2. Focus on data with methods to operate upon the object's data

3. Includes the concept like Encapsulation and abstraction which hides the complexities from the user and show only functionality.

4. Implements the real-time approach like inheritance, abstraction, etc.

5. The examples of the object-oriented paradigm are C++, Simula, Smalltalk, Python, C#, etc.

# object-oriented vs object-based

1. Object-oriented languages follow all the concepts of OOPs whereas, the object-based language doesn't follow all the concepts of OOPs like inheritance and polymorphism.

2. Object-oriented languages do not have the inbuilt objects whereas Object-based languages have the inbuilt objects, for example, JavaScript has window object.

3. Examples of object-oriented programming are Java, C#, Smalltalk, etc. whereas the examples of object-based languages are JavaScript, VBScript, etc.

# Observable chain

List of operations or transformations performed in between the source and end subscriber. One of the examples, it to emit User objects, filtering out the admin users and checking for authentication of users, and finally map full name.

# observable chains test RxJava

TestObserver and TestSubscriber are common classes used to test Observable / Flowable operations. With these you can wait for, and inspect all received events and their exact values.

The TestScheduler class is very useful when testing time-based operators (e.g. timeout(), buffer(), window(), etc) – it allows to manually control the “time” to you can test all possible code paths in your chain.

# Observable RxJava

Observable represents any object that can get data from a data source and whose state may be of interest in a way that other objects may register an interest.

In Observable, there are two types: Cold and Hot Observables. Cold Observables will perform work and subsequently emit items only once is someone has subscribed, whereas Hot Observables will perform work and emit items regardless of observers or not.

Also, there are two types:

1. Non-Blocking – asynchronous execution is supported and is allowed to unsubscribe at any point in the event stream. On this article, we'll focus mostly on this kind of type

2. Blocking – all onNext observer calls will be synchronous, and it is not possible to unsubscribe in the middle of an event stream. We can always convert an Observable into a Blocking Observable, using the method toBlocking:

What’s the difference between a COLD and HOT observables?

They start emitting items differently (see #4).

Cold observables are created multiple times and each instance can be triggered on it’s own. Hot observables are like a “stream” of ongoing events – observers can come and go, but the stream is created ones and just goes on.

Can you transform a COLD observable to a HOT one and vice-versa?

One way to make a Cold observable Hot is by using publish().connect(). publish() converts the Cold observable to a ConnectableObservable, which pretty much behaves like a Hot one. Once triggered with the .connect() operator, it’ll publish events regardless if there are any subscribers.

Another way to transform a Cold observable to a Hot one is by wrapping it with a Subject. The Subject subscribes to the Cold observable immediately and exposes itself as an Observable to future subscribers. Again, the work is performed regardless whether there are any subscribers … and on the other hand multiple subscribers to the Subject won’t trigger the initial work multiple times.

# Observable(keyword) RxJava

Observable – represents a stream of events that emits zero to many events (onNext(value)), then either completes (onComplete()) or errors out (onError(throwable)). It does NOT support backpressure

# Observables vs LiveData

1. LiveData is an observable data holder class. It is lifecycle-aware, meaning it respects the lifecycle of other app components, such as activities, fragments, or services. This awareness ensures LiveData only updates app component observers that are in an active lifecycle state.

2. Observables, on the other hand, are not lifecycle-aware. This means they will continue to emit data even if the observer is no longer in an active state, such as when an activity has been destroyed. This can lead to memory leaks if the observer is not unregistered when no longer needed.

# observables, observers

In reactive programming, observables are objects that emit a stream of data, and observers are objects that consume that data. In the context of Android development, observables can be used to emit data from a ViewModel to a View, and observers can be used to consume that data and update the UI.

# observeOn() vs subscribeOn()

1. subscribeOn() denotes the Scheduler on which the source work will be performed on. Since there’s only one initial source of an Observable chain, it makes sense to only have one subscribeOn() operator.

2. observeOn() denotes the Scheduler on which all downstream operations will be performed. In other words it changes the Scheduler for all operators after it. Since there can be many such operators, having multiple observeOn() operators in a single chain makes sense and works as expected.

# observer RxJava

An observer subscribes to an Observable sequence. The sequence sends items to the observer one at a time.

# observers, observables

In reactive programming, observables are objects that emit a stream of data, and observers are objects that consume that data. In the context of Android development, observables can be used to emit data from a ViewModel to a View, and observers can be used to consume that data and update the UI.

# OkHttp

OkHttp is an efficient HTTP & HTTP/2 client for Android and Java applications. It comes with advanced features, such as connection pooling (if HTTP/2 isn't available), transparent GZIP compression, and response caching, to avoid the network completely for repeated requests.

OkHTTP is an open source project designed to be an efficient HTTP client. It supports the SPDY protocol. SPDY is the basis for HTTP 2.0 and allows multiple HTTP requests to be multiplexed over one socket connection.

# one-way vs tw-way data binding

Two-way data binding means that changes to either the model or the view will update the other automatically. One-way data binding means that changes to the model will update the view, but not vice versa.

# onNext(), onComplete(), onError()

1. OnNext is called on our observer each time a new event is published to the attached Observable. This is the method where we'll perform some action on each event

2. OnCompleted is called when the sequence of events associated with an Observable is complete, indicating that we should not expect any more onNext calls on our observer

3. OnError is called when an unhandled exception is thrown during the RxJava framework code or our event handling code

These are the callbacks an Observable / Flowable will receive. onNext is called for each emission of the Observable / Flowable (e.g. zero to infinity times). onComplete() and onError() are mutually exclusive – only ONE of them will be called at most once. In other words a stream cannot complete and error out at the same time.

onNext() – from zero between infinite number of times

onComplete() – maximum once per stream

onError() – maximum once per stream

# onPause() only scenario

Create and launch a new activity which obscures the current activity partially. This can be done by defining the layout\_width and layout\_height to partially cover the screen. This would keep the first activity visible but not in the foreground. Example: define the layout\_width and layout\_height as 200dp each.

# onSaveInstanceState() vs onRestoreInstanceState()

In general the onSaveInstanceState() is invoked after onPause() and before the onStop(). But the API documentation explicitly states that the onSaveInstanceState( ) method will be called before onStop() but makes no guarantees it will be called before or after onPause(). The onRestoreInstanceState() is called after onStart() is invoked. The onRestoreInstanceState() method is invoked only when the activity was killed before. If the activity is NOT killed the onSaveInstanceState() is NOT called. When the activity is being destroyed, the onSaveInstanceState() gets invoked. The onSaveInstanceState contains a Bundle parameter. The data to be saved is stored in the bundle object in the form of a HashMap. The bundle object is like a custom HashMap object. The data is retrieved in the onRestoreInstanceState() method using the keys.

# open and public kotlin

The keyword “open” refers to the term ""open for expansion"". The open annotation on a class is the polar opposite of the final annotation in Java: it allows others to inherit from it. By default, a class cannot be inherited in Kotlin. In Kotlin, an open method signifies that it can be overridden, whereas it cannot be by default. Instead, any methods in Java can be overridden by default.

In Kotlin, all the classes are final by default. If no visibility modifier is specified, the public is used by default, which means our declarations will be accessible everywhere inside the program.

# open kotlin

In Kotlin, the classes and functions are final by default. So, it is not possible to inherit the class or override the functions. To achieve this, we need to use the open keyword before the class and function.

# Operators RxJava

An operator is a function that takes one Observable (the source) as its first argument and returns another Observable (the destination). Then for every item that the source observable emits, it will apply a function to that item, and then emit the result on the destination Observable.

Operators can be chained together to create complex data flows that filter event based on certain criteria. Multiple operators can be applied to the same observable.

It is not difficult to get into a situation in which an Observable is emitting items faster than an operator or observer can consume them. You can read more about back-pressure here.

# overloading

Method overloading is the polymorphism technique which allows us to create multiple methods with the same name but different signature. We can achieve method overloading in two ways.

1. By Changing the number of arguments

2. By Changing the data type of arguments

Method overloading increases the readability of the program. Method overloading is performed to figure out the program quickly.

Why is method overloading not possible by changing the return type in java?

In Java, method overloading is not possible by changing the return type of the program due to avoid the ambiguity.

Can we overload the methods by making them static?

No, We cannot overload the methods by just applying the static keyword to them(number of parameters and types are the same). Consider the following example.

# overloading type promotion

By Type promotion is method overloading, we mean that one data type can be promoted to another implicitly if no exact matching is found.

The byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int, long, float or double and so on. Consider the following example.

# overloading vs overriding

1. Method overloading increases the readability of the program.

1-1. Method overriding provides the specific implementation of the method that is already provided by its superclass.

2. Method overloading occurs within the class.

2-1. Method overriding occurs in two classes that have IS-A relationship between them.

3. In this case, the parameters must be different.

3-1. In this case, the parameters must be the same.

# overriding

If a subclass provides a specific implementation of a method that is already provided by its parent class, it is known as Method Overriding. It is used for runtime polymorphism and to implement the interface methods.

Rules for Method overriding

1. The method must have the same name as in the parent class.

2. The method must have the same signature as in the parent class.

Two classes must have an IS-A relationship between them.

Can we override the static method?

No, you can't override the static method because they are the part of the class, not the object.

Why can we not override static method?

It is because the static method is the part of the class, and it is bound with class whereas instance method is bound with the object, and static gets memory in class area, and instance gets memory in a heap.

Can we override the overloaded method?

Yes.

Can we override the private methods?

No, we cannot override the private methods because the scope of private methods is limited to the class and we cannot access them outside of the class.

# overriding change scope

Yes, we can change the scope of the overridden method in the subclass. However, we must notice that we cannot decrease the accessibility of the method. The following point must be taken care of while changing the accessibility of the method.

1. The private can be changed to protected, public, or default.

2. The protected can be changed to public or default.

3. The default can be changed to public.

4. The public will always remain public.

# overriding throws clause

Yes, we can modify the throws clause of the superclass method while overriding it in the subclass. However, there are some rules which are to be followed while overriding in case of exception handling.

1. If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception, but it can declare the unchecked exception.

2. If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

# overriding vs overloading

1. Method overloading increases the readability of the program.

1-1. Method overriding provides the specific implementation of the method that is already provided by its superclass.

2. Method overloading occurs within the class.

2-1. Method overriding occurs in two classes that have IS-A relationship between them.

3. In this case, the parameters must be different.

3-1. In this case, the parameters must be the same.

# Paging Library Jetpack

Paging Library makes it easier to load and display large datasets from a database or network asynchronously by providing a way to load data in small chunks (called “pages”). This way, only the data that is needed is loaded at any given time, which makes it more efficient and reduces the amount of time and resources required to load and display large datasets.

# Parallelism RxJava

RxJava will support parallelism, and this is achieved in two ways; using flatMap() operator, each stream inside flatMap() should subscribe background thread. Using ParallelFlowable, it provides easier and explicit API to achieve parallelism.

a) Using the flatMap() operator. Each inner stream inside the flatMap() operation should subscribeOn() a background thread (preferably Schedulers.io()).

b) Although the above pattern works correctly to achieve parallelism, in recent versions of RxJava a better construct was introduced – the ParallelFlowable. It provides an easier and more explicit API to achieve parallelism.

# Parcelable vs Serializable

While developing applications usually it needs to transfer data from one activity to another. This data needs to be added into a corresponding intent object. Some additional actions are required to make the data suitable for transfer. For doing that the object should be either serializable or parcelable.

1. Serializable:

1-1. Serializable is a standard Java interface. In this approach, you simply mark a class Serializable by implementing the interface and java will automatically serialize it.

1-2. Reflection is used during the process and many additional objects are created. This leads to plenty of garbage collection and poor performance.

2. Parcelable:

2-1. Parcelable is an Android-specific interface. In this approach, you implement the serialization yourself.

2-2. Reflection is not used during this process and hence no garbage is created.

2-3. Parcelable is far more efficient than Serializable since it gets around some problems with the default Java serialization scheme. Also, it is faster because it is optimized for usage on the development of Android, and shows better results.

# pass data coroutine

One of the best ways to pass data between coroutines in Kotlin is with channel objects, which allow for safe and synchronized communication between separate threads or processes. To create a channel object, simply use the Channel constructor and define any channels you want to send data through.

# passing this into method

1. this is a final variable. Therefore, this cannot be assigned to any new value whereas the current class object might not be final and can be changed.

2. this can be used in the synchronized block.

# PendingIntent

A PendingIntent is a wrapper for the Intent object. It's passed to a foreign application (NotificationManager, AlarmManager) such that when some given conditions are met, the desired action is performed on the intent object it holds onto. The foreign application performs the intent with the set of permissions defined in our application.

# PendingIntent

PendingIntent is a high-level API that lets you create and manage asynchronous requests to other apps. You can use this class for starting activities, delivering results, and receiving results from other apps.

PendingIntent starts at some point in the future but Intent starts immediately.

Pending Intent is frequently used in:

1. NotificationiManager

2. AlarmManager

3. AppWidgetManager

# Permissions android

Permissions allow certain restrictions to be imposed primarily to protect data and code. Without these, codes could be compromised, resulting to defects in functionality.

# phone call intent

1. To enable calling from the application we need to add the following permission in the manifest tag of AndroidManifest.xml

<uses-permission android:name=""android.permission.CALL\_PHONE"" />

In the MainActivity the following code invokes an action call to the given number represented as a string. The string is parsed as a URI.

String phone\_number = ""XXXXXXX"" // replace it with the number

Intent intent=new Intent(Intent.ACTION\_CALL,Uri.parse(""tel:""+phone number);

startActivity(intent);

2. To open a URL we need to add the following permission.

<uses-permission android:name=""android.permission.INTERNET"" />

The intent to view a URL is defined below.

Intent intent = new Intent(Intent.ACTION\_VIEW, Uri.parse(""https://www.journaldev.com/""));

startActivity(intent);

# preemptive scheduling vs time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

# prevent ANR Application Not Responding

ANR stands for Application Not Responding. An ANR will occur if you’re running a process on the UI thread which takes an extended time, usually around 5 seconds. During this point, the GUI (Graphical User Interface) will lock up which can end in anything the user presses won’t be actioned. After the 5 seconds approx. has occurred, if the thread still hasn’t recovered then an ANR dialogue box is shown informing the user that the appliance isn’t responding and can give the user the choice to either wait, in the hope that the app will eventually recover, or to force close the app.

Stop doing heavy tasks on the main thread. Instead, use worker threads such as IntentService, AsyncTask Handler, or another Thread simply. Detecting where ANRs happen is straightforward if it’s a permanent block (deadlock acquiring some locks for instance), but harder if it’s just a short-lived delay. First, re-evaluate your code and appearance for vulnerable spots and long-running operations.

# primary constructor kotlin

In Kotlin, the primary constructor is a part of the class header. Unlike Java, it doesn't need you to declare a constructor in the body of the class.

Kotlin facilitates you to declare the constructor in the class header itself:

class Person constructor(name: String, age: Int, salary: Int) {

}

Just like functions or methods, it takes a series of parameters with their type. These parameters initialize the variables present in the class.

If you do not have any annotations or modifiers (public, private, protected), you can omit the constructor keyword like the following example.

class Person (name: String, age: Int, salary: Int) {

}

By removing the constructor keyword, you can get code that is simplified and easy to understand.

# private access specifier

1. Public The classes, methods, or variables which are defined as public, can be accessed by any class or method.

2. Protected Protected can be accessed by the class of the same package, or by the sub-class of this class, or within the same class.

3. Default Default are accessible within the package only. By default, all the classes, methods, and variables are of default scope.

4. Private The private class, methods, or variables defined as private can be accessed within the class only.

# process vs thread

1. A Program in the execution is called the process whereas; A thread is a subset of the process

2. Processes are independent whereas threads are the subset of process.

3. Process have different address space in memory, while threads contain a shared address space.

4. Context switching is faster between the threads as compared to processes.

5. Inter-process communication is slower and expensive than inter-thread communication.

6. Any change in Parent process doesn't affect the child process whereas changes in parent thread can affect the child thread.

# protected access specifier

1. Public The classes, methods, or variables which are defined as public, can be accessed by any class or method.

2. Protected Protected can be accessed by the class of the same package, or by the sub-class of this class, or within the same class.

3. Default Default are accessible within the package only. By default, all the classes, methods, and variables are of default scope.

4. Private The private class, methods, or variables defined as private can be accessed within the class only.

# public access specifier

1. Public The classes, methods, or variables which are defined as public, can be accessed by any class or method.

2. Protected Protected can be accessed by the class of the same package, or by the sub-class of this class, or within the same class.

3. Default Default are accessible within the package only. By default, all the classes, methods, and variables are of default scope.

4. Private The private class, methods, or variables defined as private can be accessed within the class only.

# kotlin open and public

The keyword “open” refers to the term ""open for expansion"". The open annotation on a class is the polar opposite of the final annotation in Java: it allows others to inherit from it. By default, a class cannot be inherited in Kotlin. In Kotlin, an open method signifies that it can be overridden, whereas it cannot be by default. Instead, any methods in Java can be overridden by default.

In Kotlin, all the classes are final by default. If no visibility modifier is specified, the public is used by default, which means our declarations will be accessible everywhere inside the program.

# public static void vs static public void

The program compiles and runs correctly because the order of specifiers doesn't matter in Java.

# pure function RxJava

A pure function is one that doesn’t have any side effects and has stable output – e.g. the same input will always produce the same output. Working with pure functions makes code easier to reason, as there’s no hidden side effects and implicit dependencies between functions. Given the composable nature of RxJava operators, a very good combination is keeping each operation a highly isolated pure function – this way alterations of the stream are easier.

# purpose Companion Object

Unlike Java or C#, Kotlin doesn’t have static members or member functions. If you need to write a function that can be called without having a class instance but needs access to the internals of a class, you can write it as a member of a companion object declaration inside that class.

class EventManager {

companion object FirebaseManager {

}

}

val firebaseManager = EventManager.FirebaseManager

The companion object is a singleton. The companion object is a proper object on its own, and can have its own supertypes - and you can assign it to a variable and pass it around. If you're integrating with Java code and need a true static member, you can annotate a member inside a companion object with @JvmStatic.

# Queue

Queue (java.util.Queue) interface defines queue data structure, which stores the elements in the form FIFO (first in first out).

# race-condition

A Race condition is a problem which occurs in the multithreaded programming when various threads execute simultaneously accessing a shared resource at the same time. The proper use of synchronization can avoid the Race condition.

# reactive programming

Reactive programming is a programming paradigm that is concerned with data streams and the propagation of change. This means that when a piece of data changes, all observers of that data are notified and updated automatically. This is in contrast to the more traditional approach of having to manually keep track of what data has changed and then update the observers accordingly.

# reactive programming vs event handling

Reactive programming is a programming paradigm that is concerned with data streams and the propagation of changes. Event handling, on the other hand, is a way of responding to events that occur in a program. In the context of Android development, reactive programming can be used to handle events such as user input, while event handling is used to respond to events such as button clicks.

# read-only class

A class can be made read-only by making all of the fields private. The read-only class will have only getter methods which return the private property of the class to the main method. We cannot modify this property because there is no setter method available in the class.

"

# <starting>RecyclerView

RecyclerView is a ViewGroup added to the Android Studio as a successor of the GridView and ListView. It is an improvement on both of them. It has been created to make possible construction of any lists with XML layouts as an item that can be customized vastly while improving the efficiency of ListViews and GridViews. This improvement is achieved by recycling the views which are out of the visibility of the user. For example, if a user scrolled down to a position where items 4 and 5 are visible; items 1, 2, and 3 would be cleared from the memory to reduce memory consumption.

# RecyclerView improve

1. Set a specific width and height to ImageView in RecyclerView items

2. Avoid using NestedView

3. Use the setHasFixedsize method

4. Use the image loading library for loading images

5. Do less work in the OnBindViewHolder method

6. Use the NotifyItem method for your RecyclerView

# RecyclerView vs ListView

- A RecyclerView recycles and reuses cells when scrolling. This is a default behaviour. It's possible to implement the same in a ListView too but we need to implement a ViewHolder there

- A RecyclerView decouples list from its container so we can put list items easily at run time in the different containers (linearLayout, gridLayout) by setting LayoutManager

- Animations of RecyclerView items are decoupled and delegated to `ItemAnimator`

# Reduce APK size

1. Remove unused sources

2. Use of Vector Drawables

3. Reuse your code

4. Compress PNG and JPEG files

5. Use of Lint

6. Use images in WebP file format

7. Use of proguard

8. Use of ShrinkResources

9. Limit the usage of external libraries

10. Use the Android Size Analyzer tool

11. Generate App Bundles instead of APK

12. Use of Resconfigs

# reduce vs fold

1. fold takes an initial value, and the first invocation of the lambda you pass to it will receive that initial value and the first element of the collection as parameters.

listOf(1, 2, 3).fold(0) { sum, element -> sum + element }

The first call to the lambda will be with parameters 0 and 1.

Having the ability to pass in an initial value is useful if you have to provide some sort of default value or parameter for your operation.

2. reduce doesn't take an initial value, but instead starts with the first element of the collection as the accumulator (called sum in the following example)

listOf(1, 2, 3).reduce { sum, element -> sum + element }

The first call to the lambda here will be with parameters 1 and 2.

# reflection

Reflection is the process of examining or modifying the runtime behavior of a class at runtime. The java.lang.Class class provides various methods that can be used to get metadata, examine and change the runtime behavior of a class. The java.lang and java.lang.reflect packages provide classes for java reflection.

Reflection is used in:

IDE (Integrated Development Environment), e.g., Eclipse, MyEclipse, NetBeans.

Debugger

Test Tools, etc.

# regular class vs data class

A data class is a class intended to hold data. A regular class is a class that can perform arbitrary operations. Data classes are typically simpler and more efficient than regular classes.

# regular function vs extension function

An extension function is a function defined for a specific type and can be called on variables of that type. A regular function is not defined for a specific type and can be called on any type of variable.

# regular function vs suspend function

A suspend function can be suspended, meaning that you can pause its execution and resume it at a later time. A regular function cannot be suspended and will always execute to completion.

# reified types in Kotlin

Reified types can be accessed at runtime rather than just at compile time. In Kotlin, reified types provide metadata about a type at run time, such as its name or the names of its members. They are also commonly used in reflection and generic programming. It is an advanced feature not commonly used by beginners to the Kotlin language.

# RelativeLayout, AbsoluteLayout, LinearLayout

LinearLayout arranges its children in a single row or single column one after the other.

RelativeLayout arranges it's children in positions relative to each other or relative to parent depending upon the LayoutParams defined for each view.

AbsoluteLayout needs the exact positions of the x and y coordinates of the view to position it. Though this is deprecated now.

# restriction static method

Two main restrictions are applied to the static methods.

1. The static method can not use non-static data member or call the non-static method directly.

2. this and super cannot be used in static context as they are non-static.

# Retrofit

Retrofit is a type-safe REST client built by square for Android and Java which intends to make it simpler to expand RESTful web services. Retrofit uses OkHttp as the system’s administration layer and is based on it. Retrofit naturally serializes the JSON reaction utilizing a POJO (PlainOldJavaObject) which must be characterized as cutting edge for the JSON Structure. To serialize JSON we require a converter to change it into Gson first. Retrofit is much simpler than other libraries; we don’t have to parse our JSON. It directly returns objects but there is one disadvantage: it doesn’t provide support to load images from the server, but we can use Picasso for the same.

# Room

The requirement of a database in Android is fulfilled by SQLite from the very beginning. However, it comes with some severe drawbacks like not checking the queries at compile-time, it does not save plain-old-Java Objects(commonly referred to as POJOs). Developers also need to write a lot of boilerplate code to make the SQLite database work in the Android OS environment. The Room component comes into the picture as an SQLite Object Mapping Library which overcomes all the mentioned challenges. Room converts queries directly into objects, checks errors in queries at the compile-time, and is also capable of persisting the Java POJOs.

Moreover, it produces LiveData results/observables from the given query result. Because of this versatile nature of the Room component, Google officially supports and recommends developers to use it. The Room consists of the following sub-components:

1. Entity: It is the annotated class for which the Room creates a table within the database. The field of the class represents columns in the table.

2. DAO(Data Access Object): It is responsible for defining the methods to access the database and to perform operations.

3. Database: It is an abstract class that extends RoomDatabase class and it serves as the main access point to the underlying app’s relational data.

# run:-

Context object: this

Return value: lambda result

The ‘run' function is a combination of the ‘let' and ‘with' functions. When the object lambda involves both initialization and computation of the return value, this is the method to use. We can use run to make null safety calls as well as other calculations.

# Runnable interface vs Callable interface

The Callable interface and Runnable interface both are used by the classes which wanted to execute with multiple threads. However, there are two main differences between the both :

1. A Callable <V> interface can return a result, whereas the Runnable interface cannot return any result.

2. A Callable <V> interface can throw a checked exception, whereas the Runnable interface cannot throw checked exception.

3. A Callable <V> interface cannot be used before the Java 5 whereas the Runnable interface can be used.

# Runnable interface vs Thread class

1. By extending the Thread class, we cannot extend any other class, as Java does not allow multiple inheritances while implementing the Runnable interface; we can also extend other base class(if required).

2. By extending the Thread class, each of thread creates the unique object and associates with it while implementing the Runnable interface; multiple threads share the same object

3. Thread class provides various inbuilt methods such as getPriority(), isAlive and many more while the Runnable interface provides a single method, i.e., run().

# Runtime class

Java Runtime class is used to interact with a java runtime environment. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory, etc. There is only one instance of java.lang.Runtime class is available for one java application. The Runtime.getRuntime() method returns the singleton instance of Runtime class.

# Runtime Polymorphism

Runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Can you achieve Runtime Polymorphism by data members?

No, because method overriding is used to achieve runtime polymorphism and data members cannot be overridden. We can override the member functions but not the data members.

# runtime vs compile-time

1. In compile-time polymorphism, call to a method is resolved at compile-time.

1-1. In runtime polymorphism, call to an overridden method is resolved at runtime.

2. compile-time, It is also known as static binding, early binding, or overloading.

2-1. runtime, It is also known as dynamic binding, late binding, overriding, or dynamic method dispatch.

3. compile-time, Overloading is a way to achieve compile-time polymorphism in which, we can define multiple methods or constructors with different signatures.

3-1. runtime, Overriding is a way to achieve runtime polymorphism in which, we can redefine some particular method or variable in the derived class. By using overriding, we can give some specific implementation to the base class properties in the derived class.

4. compile-time, It provides fast execution because the type of an object is determined at compile-time.

4-1. runtime, It provides slower execution as compare to compile-time because the type of an object is determined at run-time.

5. Compile-time polymorphism provides less flexibility because all the things are resolved at compile-time.

5-1. Run-time polymorphism provides more flexibility because all the things are resolved at runtime.

# RxJava- Conditional Operators

1. DefaultIfEmpty emits item from the source Observable, or a default item if the source Observable is empty.

2. TakeWhile operator discards items emitted by an Observable after a specified condition becomes false.

3. Of course, there more others operators that could cover our needs like Contain, SkipWhile, SkipUntil, TakeUntil, etc.

# RxJava Connectable Observables

A ConnectableObservable resembles an ordinary Observable, except that it doesn't begin emitting items when it is subscribed to, but only when the connect operator is applied to it.

In this way, we can wait for all intended observers to subscribe to the Observable before the Observable begins emitting items.

# RxJava constructs

1. Completable – maps an operation that either completes without returning a value (onComplete()) or errors out (onError(throwable)).

Single – either returns a value (onSuccess(value)) or errors out (onError(throwable)).

2. Maybe – has 3 options – returns a value successfully (onSuccess(value)), completes successfully without any value (onComplete()) or errors out (onError(throwable)).

3. Observable – represents a stream of events that emits zero to many events (onNext(value)), then either completes (onComplete()) or errors out (onError(throwable)). It does NOT support backpressure (see #32).

4. Flowable – like an Observable, however it DOES support backpressure.

# RxJava custom operators

Users can create custom operators in RxJava, and this is highly recommended to reuse existing or any combination. It is tricky to implement a new operator as it would lead to too many of errors, such as thread safety, API breaking, etc.

# RxJava- Elasticity

It means that the throughput of the system scales up or down automatically to meet demand as a resource is proportionally added/ removed. Elasticity, therefore, builds up Scalability and expands by adding the notion of automatic resource management. With this, we shall conclude the topic “RxJava Interview questions.” We have seen the top 15 questions on RxJava, which will be helpful for cracking interviews. Sometimes, the interviewer can even ask you the syntax or how is any particular part coded. Having theoretical knowledge is of much importance to at least access your skills.

# RxJava- error handling operators

We have two categories of such operators, one for side effects only and the other for handle error and continue. doOnError(…), onErrorReturn(…), onErrorResumeNext(…) are some of the error handling operators in RxJava.

# RxJava- Filter

The operator filter emits only those items from an observable that pass a predicate test.

# RxJava- flatMap() vs map()

1. The map() operator maps from a concrete ValueA to a concrete ValueB (e.g. from an Int -> String, or User -> String).

2. flatMap() maps from a concrete ValueA to a Stream<ValueB>. If Stream<ValueB> emits multiple items, all of these will be eventually served to the original observer (e.g. they are being “flattened” to a single Observer). Since there’s no restrictions on the Stream<ValueB>, flatMap() is useful to introduce parallelism in executing tasks.

# RxJava- GroupBy

GroupBy operator allows us to classify the events in the input Observable into output categories.

# RxJava- Map

The map operator transforms items emitted by an Observable by applying a function to each item.

# RxJava memory leak

As a general good practice in programming, one must clean-up the used resources after they’re no longer needed. In the case of RxJava this means disposing your Disposables correctly. A common pattern is to keep adding all long-running operations from a screen in a CompositeDisposable and ensuring that’s clean-up when the screen is gone.

# RxJava multiple errors RxJava

All unhandled errors (via any of the error handling operators) are propagated downstream. A chain can have only one terminal error event (e.g. one call to onError(throwable)), so the first unhandled error will terminate the stream. In case there’s other undelivered exceptions.

# RxJava- Observable

Observable represents any object that can get data from a data source and whose state may be of interest in a way that other objects may register an interest.

In Observable, there are two types: Cold and Hot Observables. Cold Observables will perform work and subsequently emit items only once is someone has subscribed, whereas Hot Observables will perform work and emit items regardless of observers or not.

What’s the difference between a COLD and HOT observables?

They start emitting items differently (see #4).

Cold observables are created multiple times and each instance can be triggered on it’s own. Hot observables are like a “stream” of ongoing events – observers can come and go, but the stream is created ones and just goes on.

Can you transform a COLD observable to a HOT one and vice-versa?

One way to make a Cold observable Hot is by using publish().connect(). publish() converts the Cold observable to a ConnectableObservable, which pretty much behaves like a Hot one. Once triggered with the .connect() operator, it’ll publish events regardless if there are any subscribers.

Another way to transform a Cold observable to a Hot one is by wrapping it with a Subject. The Subject subscribes to the Cold observable immediately and exposes itself as an Observable to future subscribers. Again, the work is performed regardless whether there are any subscribers … and on the other hand multiple subscribers to the Subject won’t trigger the initial work multiple times.

# RxJava- observable chains

TestObserver and TestSubscriber are common classes used to test Observable / Flowable operations. With these you can wait for, and inspect all received events and their exact values.

The TestScheduler class is very useful when testing time-based operators (e.g. timeout(), buffer(), window(), etc) – it allows to manually control the “time” to you can test all possible code paths in your chain.

# RxJava- observer

An observer subscribes to an Observable sequence. The sequence sends items to the observer one at a time.

# RxJava- Operators

An operator is a function that takes one Observable (the source) as its first argument and returns another Observable (the destination). Then for every item that the source observable emits, it will apply a function to that item, and then emit the result on the destination Observable.

Operators can be chained together to create complex data flows that filter event based on certain criteria. Multiple operators can be applied to the same observable.

It is not difficult to get into a situation in which an Observable is emitting items faster than an operator or observer can consume them. You can read more about back-pressure here.

# RxJava- Parallelism

RxJava will support parallelism, and this is achieved in two ways; using flatMap() operator, each stream inside flatMap() should subscribe background thread. Using ParallelFlowable, it provides easier and explicit API to achieve parallelism.

a) Using the flatMap() operator. Each inner stream inside the flatMap() operation should subscribeOn() a background thread (preferably Schedulers.io()).

b) Although the above pattern works correctly to achieve parallelism, in recent versions of RxJava a better construct was introduced – the ParallelFlowable. It provides an easier and more explicit API to achieve parallelism.

# RxJava pure function

A pure function is one that doesn’t have any side effects and has stable output – e.g. the same input will always produce the same output. Working with pure functions makes code easier to reason, as there’s no hidden side effects and implicit dependencies between functions. Given the composable nature of RxJava operators, a very good combination is keeping each operation a highly isolated pure function – this way alterations of the stream are easier.

# RxJava- Scan

The scan operator applies a function to each item emitted by an Observable sequentially and emits each successive value.

# RxJava Scheduler

Schedulers are used to switch execution to a different thread. RxJava is single-threaded by default, i.e., all operations are executed on a single thread. Also used as an abstraction overtime concept for time-sensitive operations such as delay(), buffer(), timeout(), window(), etc.

# RxJava- Single

Single is like an Observable who, instead of emitting a series of values, emits one value or an error notification.

With this source of data, we can only use two methods to subscribe:

OnSuccess returns a Single that also calls a method we specify.

OnError also returns a Single that immediately notifies subscribers of an error.

# RxJava- Subject

Subject means both subscriber and observer at the same time. With subjects in RxJava, users can transform cold observables to hot ones. They are also used to introduce some type of local and temporary caching of the stream. Also, help in transforming non-reactive code to reactive if the user does not find any operator for use case creation.

Types of Subjects in RxJava:

1. PublishSubject: It passes incoming events to all subscribers. New subscribers will receive events only from the point of subscription.

2. BehaviourSubject: Similar to publish subject, but each new subscriber will receive the latest value of the stream, i.e., the default value. Here, the default value of the stream provides a good user experience.

3. AsyncSubject: It emits only the last value of Observable, and that too only after the source observable completes emitting.

4. ReplaySubject: Each subscriber will receive all the events emitted by the source, regardless of at which point it is subscribed. If Observable emits too many items, they need to be in-memory cache.

# RxJava Transformer

A Transformer is a convenient way to encapsulate common operations in a reusable way. This encapsulated logic can be tested in isolation, which is easier, and further simplifies the tests of all chains that use it. For example if you find yourself repeating a sequence of operators multiple times in your code (e.g. .map(user -> user.age).filter(age -> age > 18)), this logic can be factored out in a UserLegalAgeTransformer. It can be reused in the chains like this .compose(new UserLegalAgeTransformer()).

# RxJava1 vs RxJava2

1. Null not supported anymore – previously Observable.just(user) would work even if user == null. In RxJava2 the same will throw a NullPointerException. Not supporting null values makes streams a bit easier to work with (as there’s no null checks everywhere), but one must be more cautious what data flows through them. Wrapping emissions with classes like Java Optional is handy for places where you’re not sure if the data is null-safe.

2. RxJava2 Observable type doesn’t support backpressure anymore, so places where that might be an issue had to be changed to use Flowable instead. It’s a great change as it makes pretty clear where Backpressure care is needed. On the other hand – it requires careful inspection of the flows that were using RxJava1 Observable. Common places where Backpressure might occur is when fetching data from a database or mapping user-actions to events. On a related note – in RxJava2 Subjects no longer support Backpressure. The new Processors classes do.

3. Global error handling – in RxJava2 no error can be swallowed (as it could in RxJava1). All such undelivered errors are passed to a global error handler, which can be set using the RxJavaPlugins.setErrorHandler(). Such exceptions usually occur on streams that have parallel execution. When updating, one must always set such handler and at least log the errors. Ideally this global error handler will never receives anything.

# RxRelay vs Subject

A Relay is a Subject that cannot be terminated (cannot call onError() or onComplete() on it). That’s sometimes useful, as terminating a regular Subject makes it unusable in the future.

# Safe Call operator ( ?. )

Safe Call operator ( ?. ) - Null comparisons are trivial, but the number of nested if-else expressions can be exhausting. So, in Kotlin, there's a Safe call operator,?, that simplifies things by only doing an action when a specified reference holds a non-null value. It allows us to use a single expression to perform both a null check and a method call.

# safe calls(?.) vs null check(!!)

The safe call operator i.e. ?. is used to check if the variable's value is null or not. If it is null, then null will be returned otherwise it will return the desired value.

If you want to throw NullPointerException when the variable's value is null, you can use the null check or !! Operator.

# Scan RxJava

The scan operator applies a function to each item emitted by an Observable sequentially and emits each successive value.

# ScheduledExecutorService vs ExecutorService interface

ExecutorServcie and ScheduledExecutorService both are the interfaces of java.util.Concurrent package but scheduledExecutorService provides some additional methods to execute the Runnable and Callable tasks with the delay or every fixed time period.

# Scheduler RxJava

Schedulers are used to switch execution to a different thread. RxJava is single-threaded by default, i.e., all operations are executed on a single thread. Also used as an abstraction overtime concept for time-sensitive operations such as delay(), buffer(), timeout(), window(), etc.

# scope function kotlin

The Kotlin standard library includes numerous functions that aid in the execution of a block of code within the context of an object. When you use a lambda expression to call these functions on an object, temporary scope is created. These functions are referred to as Scope functions. The object of these functions can be accessed without knowing its name. Scope functions make code more clear, legible, and succinct, which are key qualities of the Kotlin programming language.

1. let:-

Context object: it

Return value: lambda result

The let function is frequently used for null safety calls. For null safety, use the safe call operator(?.) with ‘let'. It only runs the block with a non-null value.

2. apply:-

Context object: this

Return value: context object

“Apply these to the object,” as the name suggests. It can be used to operate on receiver object members, primarily to initialise them.

3. with:-

Context object: this

Return value: lambda result

When calling functions on context objects without supplying the lambda result, ‘with' is recommended.

4. run:-

Context object: this

Return value: lambda result

The ‘run' function is a combination of the ‘let' and ‘with' functions. When the object lambda involves both initialization and computation of the return value, this is the method to use. We can use run to make null safety calls as well as other calculations.

5. also:-

Context object: it

Return value: context object

It's used when we need to do additional operations after the object members have been initialised.

# screen orientation fixed

The screen orientation can be fixed by adding the attribute `android:screenOrientation=""portrait""` or `android:screenOrientation=""landscape""` in the activity tag. To keep the screen always on for a particular screen add the `android:keepScreenOn=""true""` in the root tag of the activity layout.

# sealed classe

Kotlin introduces a crucial new form of class that isn't seen in Java. These are referred to as ""sealed classes."" Sealed classes, as the name implies, adhere to constrained or bounded class hierarchies. A sealed class is one that has a set of subclasses. When it is known ahead of time that a type will conform to one of the subclass types, it is employed. Type safety (that is, the compiler will validate types during compilation and throw an exception if a wrong type has been assigned to a variable) is ensured through sealed classes, which limit the types that can be matched at compile time rather than runtime.

# Sensor

Android-based devices have a collection of built-in sensors in them, which measure certain parameters like motion, orientation, and many more through their high accuracy. The sensors can be both hardware and software based on nature. There are three prominent categories of sensors in Android devices.

They are:

1. Position Sensor: It is used for measuring the physical position of the Android device. This has orientation sensors and magnetometers.

2. Motion Sensors: These sensors consist of gravity, rotational activity, and acceleration sensors which measure the rotation of the device or the acceleration, etc.

3. Environmental Sensor: It includes sensors that measure temperature, humidity, pressure, and other environmental factors.

# Sensor class

1. Sensor class: This class helps you to create an instance of a specific sensor. It provides methods that let you determine a sensor’s capabilities.

2. SensorManager class: This class is used to create an instance of the sensor service. It provides methods to access and list sensors, to register and unregister sensor listeners, etc.

3. SensorEvent class: This Java class is used to create a sensor event object. It provides information about the sensor event including raw sensor data, the accuracy of data, type of sensor, timestamp of event, etc.

4. SensorEventListener interface: This interface is used to create two callback methods that receive sensor event notifications when sensor value changes or when sensor accuracy changes. Those two methods are onAccuracyChanged which is called when sensor accuracy is changed and

onSensorChanged which is called when sensor values are changed.

# serializable class

A class can become serializable by implementing the Serializable interface.

# serializable not in child

It is very tricky to prevent serialization of child class if the base class is intended to implement the Serializable interface. However, we cannot do it directly, but the serialization can be avoided by implementing the writeObject() or readObject() methods in the subclass and throw NotSerializableException from these methods.

# Serializable vs Externalizable

1. The Serializable interface does not have any method, i.e., it is a marker interface.

1-1. The Externalizable interface contains is not a marker interface, It contains two methods, i.e., writeExternal() and readExternal().

2. It is used to ""mark"" Java classes so that objects of these classes may get the certain capability.

2-1. The Externalizable interface provides control of the serialization logic to the programmer.

3. Serializable, It is easy to implement but has the higher performance cost.

3-1. Externalizable, It is used to perform the serialization and often result in better performance.

4. Serializable, No class constructor is called in serialization.

4-1. Externalizable,We must call a public default constructor while using this interface.

# Serializable vs Parcelable

While developing applications usually it needs to transfer data from one activity to another. This data needs to be added into a corresponding intent object. Some additional actions are required to make the data suitable for transfer. For doing that the object should be either serializable or parcelable.

1. Serializable:

1-1. Serializable is a standard Java interface. In this approach, you simply mark a class Serializable by implementing the interface and java will automatically serialize it.

1-2. Reflection is used during the process and many additional objects are created. This leads to plenty of garbage collection and poor performance.

2. Parcelable:

2-1. Parcelable is an Android-specific interface. In this approach, you implement the serialization yourself.

2-2. Reflection is not used during this process and hence no garbage is created.

2-3. Parcelable is far more efficient than Serializable since it gets around some problems with the default Java serialization scheme. Also, it is faster because it is optimized for usage on the development of Android, and shows better results.

# serialization

Serialization in Java is a mechanism of writing the state of an object into a byte stream. It is used primarily in Hibernate, RMI, JPA, EJB and JMS technologies. It is mainly used to travel object's state on the network (which is known as marshaling). Serializable interface is used to perform serialization. It is helpful when you require to save the state of a program to storage such as the file. At a later point of time, the content of this file can be restored using deserialization. It is also required to implement RMI(Remote Method Invocation). With the help of RMI, it is possible to invoke the method of a Java object on one machine to another machine.

How can you make a class serializable in Java?

A class can become serializable by implementing the Serializable interface.

How can you avoid serialization in child class if the base class is implementing the Serializable interface?

It is very tricky to prevent serialization of child class if the base class is intended to implement the Serializable interface. However, we cannot do it directly, but the serialization can be avoided by implementing the writeObject() or readObject() methods in the subclass and throw NotSerializableException from these methods.

# Service

Service is an application component that facilitates an application to run in the background in order to perform long-running operations without user interaction. A service can run continuously in the background even if the application is closed or even after the user switches to another application.

1. Foreground Service: Operations that are immediately recognizable to users Background

2. Service: Operations that are not immediately recognizable to users

3. Bound Service: Services that provide a client-server interface for component interactions

# service

A service is a component in android that's used for performing tasks in the background such as playing Music, location updating etc. Unlike activities, a service does not have a UI. Also, a service can keep running in the background even if the activity is destroyed.

# service- Bound vs Unbound

Services are largely divided into two categories : \*\*Bound Services\*\* and \*\*Unbound/Started Services\*\*

1. \*\*Bound Services\*\*: An Android component may bind itself to a Service using `bindservice()`. A bound service would run as long as the other application components are bound to it. As soon as the components call `unbindService()`, the service destroys itself.

2. \*\*Unbound Services\*\*: A service is started when a component (like activity) calls startService() method and it runs in the background indefinitely even if the original component is destroyed.

# service by AIDL

1. create the .aidl file, which defines the programming interface

2. implement the interface, which involves extending the inner abstract Stub class as well as implanting its methods.

3. expose the interface, which involves implementing the service to the clients.

# service- lifecycle

- `onStartCommand()` : This method is called when startService() is invoked. Once this method executes, the service is started and can run in the background indefinitely. This method is not needed if the service is defined as a bounded service. The service will run indefinitely in the background when this method is defined. We'll have a stop the service ourselves

- `onBind()` This method needs to be overridden when the service is defined as a bounded service. This method gets called when bindService() is invoked. In this method, we must provide an interface that clients use to communicate with the service, by returning an IBinder. We should always implement this method, but if you don’t want to allow binding, then you should return null

- `onCreate()` : This method is called while the service is first created. Here all the service initialization is done

- `onDestroy()` : The system calls this method when the service is no longer used and is being destroyed. All the resources, receivers, listeners clean up are done here

# service- start/stop

A service is started from an activity by executing the following code snippet.

```

startService(new Intent(this, MyService.class));

```

Though just executing the above code won't start a service. We need to register the service first in the AndroidManifest.xml file as shown below.

```

<service android:name=""MyService""/>

```

To stop a service we execute `stopService()`. To stop the service from itself we call `stopSelf()`.

# Service vs Activity

1. Activity is designed to run in the foreground. / Service is mainly designed to run in the background. Foreground services are also available.

2. Activity is used when the user interface is necessary. / Service is used when the user interface is not necessary.

3. Activity is dependent. / Service act independently.

# Service vs IntentService

1. Service is the base class for Android services that can be extended to create any service. A class that directly extends Service runs on the main thread so it will block the UI (if there is one) and should therefore either be used only for short tasks or should make use of other threads for longer tasks.

2. IntentService is a subclass of Service that handles asynchronous requests (expressed as “Intents”) on demand. Clients send requests through startService(Intent) calls. The service is started as needed, handles each Intent in turn using a worker thread, and stops itself when it runs out of work. Writing an IntentService can be quite simple; just extend the IntentService class and override the onHandleIntent(Intent intent) method where you can manage all incoming requests.

# Service vs Thread

1. Service is an application component that facilitates an application to run in the background in order to perform long-running operations without user interaction. / A Thread is a concurrent unit of execution.

2. Service exposes few functionalities to other applications by calling Context.bindService(). / For Thread, Google has brought in handlers and loopers into threads.

3. When an application is killed, service is not killed. / When an application is killed, the thread is killed.

# Services vs Broadcast Receivers

A service is used for long running tasks in the background such as playing a music or tracking and updating the user's background location. A Broadcast Receiver is a component that once registered within an application executes the onReceive() method when some system event gets triggered. The events the receiver listens to are defined in the AndroidManifest.xml in the intent filters. Types of system events that a Broadcast Receiver listens to are: changes in the network, boot completed, battery low, push notifications received etc. We can even send our own custom broadcasts using `sendBroadcast(intent)`.

# Set

Set (java.util.Set) interface is a collection which cannot contain duplicate elements. It can only include inherited methods of Collection interface

# Set vs List

The List and Set both extend the collection interface. However, there are some differences between the both which are listed below.

1. The List can contain duplicate elements whereas Set includes unique items.

2. The List is an ordered collection which maintains the insertion order whereas Set is an unordered collection which does not preserve the insertion order.

3. The List interface contains a single legacy class which is Vector class whereas Set interface does not have any legacy class.

4. The List interface can allow n number of null values whereas Set interface only allows a single null value.

# Set vs Map

1. Set contains values only whereas Map contains key and values both.

2. Set contains unique values whereas Map can contain unique Keys with duplicate values.

3. Set holds a single number of null value whereas Map can include a single null key with n number of null values.

# setContentView()

The reason for doing so is that the activity life cycle onCreate() method is called only once. And this is the big reason we need to call the setContentView() in onCreate(). And it will be inefficient to call this function in onResume(), onStart(), and somewhere else because those methods are called more than once.

# setFlags() vs addFlags()

When we're using setFlags, we're replacing the old flags with a new set of Flags. When we use addFlags, we're appending more flags.

# setFlags(), addFlags() in intent

When we're using setFlags, we're replacing the old flags with a new set of Flags. When we use addFlags, we're appending more flags.

# shallow copy

Object cloning

# Shared Preferences- commit() vs apply()

Data is stored in SharedPreferences in the form of a key-value pair(HashMap). commit() was introduced in API 1 whereas apply() came up with API 9. commit() writes the data synchronously and returns a boolean value of success or failure depending on the result immediately. apply() is asynchronous and it won't return any boolean response. Also, if there is an apply() outstanding and we perform another commit(), then the commit() will be blocked until the apply() is not completed. commit() is instantaneous and performs disk writes. If we're on the main UI thread apply() should be used since it's asynchronous.

# shutdown hook

The shutdown hook is a thread that is invoked implicitly before JVM shuts down. So we can use it to perform clean up the resource or save the state when JVM shuts down normally or abruptly. We can add shutdown hook by using the following method:

Some important points about shutdown hooks are :

1. Shutdown hooks initialized but can only be started when JVM shutdown occurred.

2. Shutdown hooks are more reliable than the finalizer() because there are very fewer chances that shutdown hooks not run.

3. The shutdown hook can be stopped by calling the halt(int) method of Runtime class.

# Single RxJava

Single is like an Observable who, instead of emitting a series of values, emits one value or an error notification.

With this source of data, we can only use two methods to subscribe:

OnSuccess returns a Single that also calls a method we specify.

OnError also returns a Single that immediately notifies subscribers of an error.

# singleton

A singleton class is a class which can create only an object that can be shared by all other classes.

# singleton class

Singleton class is the class which can not be instantiated more than once. To make a class singleton, we either make its constructor private or use the static getInstance method.

# Singleton kotlin

To use the singleton pattern for our class we must use the keyword `object`

An `object` cannot have a constructor set. We can use the init block inside it though.

# sleep mode

In sleep mode, CPU is slept and doesn't accept any commands from android device except Radio interface layer and alarm.

# sleep()

The sleep() method in java is used to block a thread for a particular time, which means it pause the execution of a thread for a specific time. There are two methods of doing so.

When we call the sleep() method, it pauses the execution of the current thread for the given time and gives priority to another thread(if available). Moreover, when the waiting time completed then again previous thread changes its state from waiting to runnable and comes in running state, and the whole process works so on till the execution doesn't complete.

# sleep() vs wait()

1) The wait() method is defined in Object class. / The sleep() method is defined in Thread class.

2) The wait() method releases the lock. / The sleep() method doesn't release the lock.

# SQLite

SQLite is the open-source relational database used in Android. The SQLite engine is serverless, transactional, and also self-contained. Instead of the client-server relationship of most database management systems, the SQLite engine is integrally linked with the application. The library can be called dynamically and it can make use of simple function calls that reduce latency in database access.

# StateListDrawable

A StateListDrawable is a drawable object defined in the XML that allows us to show a different color/background for a view for different states. Essentially it's used for Buttons to show a different look for each state(pressed, focused, selected, none).

# states in activity

1. Active state: The activity is in the foreground

2. Paused state: The activity is in the background and visible

3. Stopped state: The activity is in the background but not visible, even hidden or obscuring other activities

4. Destroyed state: The activity is completely terminated or killed/removed

# static

The methods or variables defined as static are shared among all the objects of the class. The static is the part of the class and not of the object. The static variables are stored in the class area, and we do not need to create the object to access such variables. Therefore, static is used in the case, where we need to define variables or methods which are common to all the objects of the class.

For example, In the class simulating the collection of the students in a college, the name of the college is the common attribute to all the students. Therefore, the college name will be defined as static.

# static abstract method

In Java, if we make the abstract methods static, It will become the part of the class, and we can directly call it which is unnecessary. Calling an undefined method is completely useless therefore it is not allowed.

# static binding vs dynamic binding

In case of the static binding, the type of the object is determined at compile-time whereas, in the dynamic binding, the type of the object is determined at runtime.

# static block

Static block is used to initialize the static data member. It is executed before the main method, at the time of classloading.

# static constructor

As we know that the static context (method, block, or variable) belongs to the class, not the object. Since Constructors are invoked only when the object is created, there is no sense to make the constructors static. However, if you try to do so, the compiler will show the compiler error.

# static kotlin

\*\*NO\*\*. Kotlin doesn't have the static keyword. To create static method in our class we use the `companion object`. Following is the Java code:

```

class A {

public static int returnMe() { return 5; }

}

```

The equivalent Kotlin code would look like this:

```

class A {

companion object {

fun a() : Int = 5

}

}

```

To invoke this we simply do: `A.a()`.

# static method

1. A static method belongs to the class rather than the object.

2. There is no need to create the object to call the static methods.

3. A static method can access and change the value of the static variable.

# static method kotlin

1. Place the function in the companion object.

class Foo {

public static int a() { return 1; }

}

will become:

class Foo {

companion object {

fun a() : Int = 1

}

}

// to run

Foo.a();

2. Another way is to solve most of the needs for static functions with package-level functions. They are simply declared outside a class in a source code file. The package of a file can be specified at the beginning of a file with the package keyword. Under the hood these ""top-level"" or ""package"" functions are actually compiled into their own class. In the above example, the compiler would create a class FooPackage with all of the top-level properties and functions, and route all of your references to them appropriately.

Consider:

package foo

fun bar() = {}

usage:

import foo.bar

# static method restriction

Two main restrictions are applied to the static methods.

1. The static method can not use non-static data member or call the non-static method directly.

2. this and super cannot be used in static context as they are non-static.

# static method vs instance method

1. A method that is declared as static is known as the static method.

1-1. A method that is not declared as static is known as the instance method.

2. We don't need to create the objects to call the static methods.

2-1. The object is required to call the instance methods.

3. Non-static (instance) members cannot be accessed in the static context (static method, static block, and static nested class) directly.

3-1. Static and non-static variables both can be accessed in instance methods.

4. For example: public static int cube(int n){ return n\*n\*n;}

4-1. For example: public void msg(){...}.

# static methods override

No, we can't override static methods.

# static public void vs public static void

The program compiles and runs correctly because the order of specifiers doesn't matter in Java.

# static synchronization

If you make any static method as synchronized, the lock will be on the class not on the object. If we use the synchronized keyword before a method so it will lock the object (one thread can access an object at a time) but if we use static synchronized so it will lock a class (one thread can access a class at a time).

# static variable

The static variable is used to refer to the common property of all objects (that is not unique for each object), e.g., The company name of employees, college name of students, etc. Static variable gets memory only once in the class area at the time of class loading. Using a static variable makes your program more memory efficient (it saves memory). Static variable belongs to the class rather than the object.

# static variable in abstract class

Yes, we can declare static variables and methods in an abstract method. As we know that there is no requirement to make the object to access the static context, therefore, we can access the static context declared inside the abstract class by using the name of the abstract class.

# Sticky Intent

A Sticky Intent is a broadcast from sendStickyBroadcast() method such that the intent floats around even after the broadcast, allowing others to collect data from it.

# storage in Android

1. Shared Preferences

2.Internal Storage

3. External Storage

4. SQLite Databases

5. Network Connection

# super classe for stream

All the stream classes can be divided into two types of classes that are ByteStream classes and CharacterStream Classes. The ByteStream classes are further divided into InputStream classes and OutputStream classes. CharacterStream classes are also divided into Reader classes and Writer classes. The SuperMost classes for all the InputStream classes is java.io.InputStream and for all the output stream classes is java.io.OutPutStream. Similarly, for all the reader classes, the super-most class is java.io.Reader, and for all the writer classes, it is java.io.Writer.

# strictfp

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language has provided the strictfp keyword so that you get the same result on every platform. So, now you have better control over the floating-point arithmetic.

# String immutable

The simple meaning of immutable is unmodifiable or unchangeable. In Java, String is immutable, i.e., once string object has been created, its value can't be changed.

# string interpolation kotlin

String interpolation is used to evaluate string templates. We use the symbol $ to add variables inside a string.

```

val name = ""Journaldev.com""

val desc = ""$name now has Kotlin Interview Questions too. ${name.length}""

```

Using `{}` we can compute an expression too.

# string literal

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

# String objects immutable

Because Java uses the concept of the string literal. Suppose there are five reference variables, all refer to one object ""sachin"". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why string objects are immutable in java.

# String pool

String pool is the space reserved in the heap memory that can be used to store the strings. The main advantage of using the String pool is whenever we create a string literal; the JVM checks the ""string constant pool"" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. Therefore, it saves the memory by avoiding the duplicacy.

# string types in kotlin

A collection of characteristics together is known as a string. There are two types of strings available in the kotlin:

1. Raw string

2. Escaped string

# String vs StringBuffer

1) The String class is immutable. / The StringBuffer class is mutable.

2) The String is slow and consumes more memory when you concat too many strings because every time it creates a new instance. / The StringBuffer is fast and consumes less memory when you cancat strings.

3) The String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. / The StringBuffer class doesn't override the equals() method of Object class.

# StringBuffer vs StringBuilder

1) StringBuffer is synchronized, i.e., thread safe. It means two threads can't call the methods of StringBuffer simultaneously. / StringBuilder is non-synchronized,i.e., not thread safe. It means two threads can call the methods of StringBuilder simultaneously.

2) StringBuffer is less efficient than StringBuilder. / StringBuilder is more efficient than StringBuffer.

# structural expressions in kotlin

1. Break: break expression helps to break the closest enclosing loop

2. Return: This expression helps to return from the closest functions or default functions.

3. Continue: This expression helps to proceed for the next loop.

# Subject RxJava

Subject means both subscriber and observer at the same time. With subjects in RxJava, users can transform cold observables to hot ones. They are also used to introduce some type of local and temporary caching of the stream. Also, help in transforming non-reactive code to reactive if the user does not find any operator for use case creation.

Types of Subjects in RxJava:

1. PublishSubject: It passes incoming events to all subscribers. New subscribers will receive events only from the point of subscription.

2. BehaviourSubject: Similar to publish subject, but each new subscriber will receive the latest value of the stream, i.e., the default value. Here, the default value of the stream provides a good user experience.

3. AsyncSubject: It emits only the last value of Observable, and that too only after the source observable completes emitting.

4. ReplaySubject: Each subscriber will receive all the events emitted by the source, regardless of at which point it is subscribed. If Observable emits too many items, they need to be in-memory cache.

# Subject vs RxRelay

A Relay is a Subject that cannot be terminated (cannot call onError() or onComplete() on it). That’s sometimes useful, as terminating a regular Subject makes it unusable in the future.

# subscribeOn() vs observeOn()

1. subscribeOn() denotes the Scheduler on which the source work will be performed on. Since there’s only one initial source of an Observable chain, it makes sense to only have one subscribeOn() operator.

2. observeOn() denotes the Scheduler on which all downstream operations will be performed. In other words it changes the Scheduler for all operators after it. Since there can be many such operators, having multiple observeOn() operators in a single chain makes sense and works as expected.

# super

The super keyword in Java is a reference variable that is used to refer to the immediate parent class object. Whenever you create the instance of the subclass, an instance of the parent class is created implicitly which is referred by super reference variable. The super() is called in the class constructor implicitly by the compiler if there is no super or this.

# super classe for stream

All the stream classes can be divided into two types of classes that are ByteStream classes and CharacterStream Classes. The ByteStream classes are further divided into InputStream classes and OutputStream classes. CharacterStream classes are also divided into Reader classes and Writer classes. The SuperMost classes for all the InputStream classes is java.io.InputStream and for all the output stream classes is java.io.OutPutStream. Similarly, for all the reader classes, the super-most class is java.io.Reader, and for all the writer classes, it is java.io.Writer.

# super constructor chaning

class Person

{

String name,address;

int age;

public Person(int age, String name, String address)

{

this.age = age;

this.name = name;

this.address = address;

}

}

class Employee extends Person

{

float salary;

public Employee(int age, String name, String address, float salary)

{

super(age,name,address);

this.salary = salary;

}

}

public class Test

{

public static void main (String args[])

{

Employee e = new Employee(22, ""Mukesh"", ""Delhi"", 90000);

System.out.println(""Name: ""+e.name+"" Salary: ""+e.salary+"" Age: ""+e.age+"" Address: ""+e.address);

}

}

# uses of super

1. super can be used to refer to the immediate parent class instance variable.

2. super can be used to invoke the immediate parent class method.

3. super() can be used to invoke immediate parent class constructor.

# super vs this

1. The super keyword always points to the parent class contexts whereas this keyword always points to the current class context.

2. The super keyword is primarily used for initializing the base class variables within the derived class constructor whereas this keyword primarily used to differentiate between local and instance variables when passed in the class constructor.

3. The super and this must be the first statement inside constructor otherwise the compiler will throw an error.

# super(),this in constructor

No, because this() and super() must be the first statement in the class constructor.

# suspend function

A function that may be started, halted, then resumed is known as a suspend function. One of the most important things to remember about the suspend functions is that they can only be invoked from another suspend function or from a coroutine. Suspending functions are merely standard Kotlin functions with the suspend modifier added, indicating that they can suspend coroutine execution without blocking the current thread. This means that the code you're looking at may pause execution when it calls a suspending function and restart execution at a later time. However, it makes no mention of what will happen to the present thread in the meantime.

Suspending functions can call any other ordinary functions, but another suspending function is required to suspend the execution. Because a suspending function cannot be called from a regular function, numerous coroutine builders are supplied, allowing you to call a suspending function from a non-suspending scope like launch, async, or runBlocking.

delay() function is an example of suspend function.

# suspend function vs regular function

A suspend function can be suspended, meaning that you can pause its execution and resume it at a later time. A regular function cannot be suspended and will always execute to completion.

# suspending vs blocking

1. A blocking call to a function means that a call to any other function, from the same thread, will halt the parent’s execution. Following up, this means that if you make a blocking call on the main thread’s execution, you effectively freeze the UI. Until that blocking calls finishes, the user will see a static screen, which is not a good thing.

2. Suspending doesn’t necessarily block your parent function’s execution. If you call a suspending function in some thread, you can easily push that function to a different thread. In case it is a heavy operation, it won’t block the main thread. If the suspending function has to suspend, it will simply pause its execution. This way you free up its thread for other work. Once it’s done suspending, it will get the next free thread from the pool, to finish its work.

# switchMap(), concatMap(), flatMap()

1. flatMap() is used to split chain to multiple intermediary streams, and results of which are flattened to a single stream. Emissions of these intermediary streams are directly propagated to the main stream in any order.

2. switchMap() is similar to flatMap(), but whenever a new item is being emitted from a source, all the intermediary streams generated are terminated, and hence only the latest intermediary stream remains active.

3. concatMap() also works similar to flatMap(), with an exception such that intermediary streams are activated based on the order appearance.

# synchronization

Synchronization is the capability to control the access of multiple threads to any shared resource. It is used:

1. To prevent thread interference.

2. To prevent consistency problem.

When the multiple threads try to do the same task, there is a possibility of an erroneous result, hence to remove this issue, Java uses the process of synchronization which allows only one thread to be executed at a time. Synchronization can be achieved in three ways:

1. by the synchronized method

2. by synchronized block

3. by static synchronization

# synchronization static

If you make any static method as synchronized, the lock will be on the class not on the object. If we use the synchronized keyword before a method so it will lock the object (one thread can access an object at a time) but if we use static synchronized so it will lock a class (one thread can access a class at a time).

# synchronize List, Set, Map

Collections class provides methods to make List, Set or Map elements as synchronized:

public static List synchronizedList(List l){}

public static Set synchronizedSet(Set s){}

public static SortedSet synchronizedSortedSet(SortedSet s){}

public static Map synchronizedMap(Map m){}

public static SortedMap synchronizedSortedMap(SortedMap m){}

# Synchronized block

The Synchronized block can be used to perform synchronization on any specific resource of the method. Only one thread at a time can execute on a particular resource, and all other threads which attempt to enter the synchronized block are blocked.

1. Synchronized block is used to lock an object for any shared resource.

2. The scope of the synchronized block is limited to the block on which, it is applied. Its scope is smaller than a method.

Java object can be locked down for exclusive use by a given thread. You can lock an object by putting it in a ""synchronized"" block. The locked object is inaccessible to any thread other than the one that explicitly claimed it.

# Synchronous vs Asynchronous

Synchronous programming: In Synchronous programming model, a thread is assigned to complete a task and hence thread started working on it, and it is only available for other tasks once it will end the assigned task.

Asynchronous Programming: In Asynchronous programming, one job can be completed by multiple threads and hence it provides maximum usability of the various threads.

# TableLayout vs FrameLayout

A FrameLayout stack up child views above each other with the last view added on the top. Though we can control the position of the children inside the FrameLayout using the layout\_gravity attribute. When the width and height of the FrameLayout are set to wrap\_content, the size of the FrameLayout equals the size of the largest child (plus padding). A TableLayout consists of TableRows. The children are arranged in the form of rows and columns.

# targetSdkVersion vs compileSdkVersion

1. compileSdkVersion:

1-1. The compileSdkVersion is the version of API the application is compiled against. You can use Android API features involved in that version of the API (as well as all previous versions).

1-2. For example, if you try and use API 15 features but set compileSdkVersion to 14, you will get a compilation error. If you set compileSdkVersion to 15 you can still run the app on an API 14 device as long as your app’s execution paths do not attempt to invoke any APIs specific to API 15.

2. targetSdkVersion:

2-1. The targetSdkVersion indicates that you have tested your app on (presumably up to and including) the version you specify. This is like a certification or sign-off you are giving the Android OS as a hint to how it should handle your application in terms of OS features.

2-2. For example, setting the targetSdkVersion value to “11” or higher permits the system to apply a new default theme (Holo) to the application when running on Android 3.0 or higher. It also disables screen compatibility mode when running on larger screens (because support for API level 11 implicitly supports larger screens).

# taskAffinity

A taskAffinity is an attribute tag defined in the activity tag in the AndroidManifest.xml for launchMode singleInstance. Activities with similar taskAffinity values are grouped together in one task.

# ternary operator kotlin

No. In Kotlin, we don't have a ternary operator like Java, but we can use the functionality of the ternary operator by using if-else or Elvis operator.

# test observable chains RxJava

TestObserver and TestSubscriber are common classes used to test Observable / Flowable operations. With these you can wait for, and inspect all received events and their exact values.

The TestScheduler class is very useful when testing time-based operators (e.g. timeout(), buffer(), window(), etc) – it allows to manually control the “time” to you can test all possible code paths in your chain.

# this assign reference

No, this cannot be assigned to any value because it always points to the current class object and this is the final reference in Java. However, if we try to do so, the compiler error will be shown. Consider the following example.

# this constructor chaining

Constructor chaining enables us to call one constructor from another constructor of the class with respect to the current class object. We can use this keyword to perform constructor chaining within the same class. Consider the following example which illustrates how can we use this keyword to achieve constructor chaining.

# this java

The this keyword is a reference variable that refers to the current object. There are the various uses of this keyword in Java. It can be used to refer to current class properties such as instance methods, variable, constructors, etc. It can also be passed as an argument into the methods or constructors. It can also be returned from the method as the current class instance.

# this passing into a method

1. this is a final variable. Therefore, this cannot be assigned to any new value whereas the current class object might not be final and can be changed.

2. this can be used in the synchronized block.

# this static members

Yes, It is possible to use this keyword to refer static members because this is just a reference variable which refers to the current class object. However, as we know that, it is unnecessary to access static variables through objects, therefore, it is not the best practice to use this to refer static members.

# this uses

1. this can be used to refer to the current class instance variable.

2. this can be used to invoke current class method (implicitly)

3. this() can be used to invoke the current class constructor.

4. this can be passed as an argument in the method call.

5. this can be passed as an argument in the constructor call.

6. this can be used to return the current class instance from the method.

# this vs super

1. The super keyword always points to the parent class contexts whereas this keyword always points to the current class context.

2. The super keyword is primarily used for initializing the base class variables within the derived class constructor whereas this keyword primarily used to differentiate between local and instance variables when passed in the class constructor.

3. The super and this must be the first statement inside constructor otherwise the compiler will throw an error.

# this(),super() in constructor

No, because this() and super() must be the first statement in the class constructor.

# thread

A thread is a lightweight subprocess. It is a separate path of execution because each thread runs in a different stack frame. A process may contain multiple threads. Threads share the process resources, but still, they execute independently.

What do you understand by inter-thread communication?

1. The process of communication between synchronized threads is termed as inter-thread communication.

2. Inter-thread communication is used to avoid thread polling in Java.

3. The thread is paused running in its critical section, and another thread is allowed to enter (or lock) in the same critical section to be executed.

4. It can be obtained by wait(), notify(), and notifyAll() methods.

Why must wait() method be called from the synchronized block?

We must call the wait method otherwise it will throw java.lang.IllegalMonitorStateException exception. Moreover, we need wait() method for inter-thread communication with notify() and notifyAll(). Therefore It must be present in the synchronized block for the proper and correct communication.

Is it possible to start a thread twice?

No, we cannot restart the thread, as once a thread started and executed, it goes to the Dead state. Therefore, if we try to start a thread twice, it will give a runtimeException ""java.lang.IllegalThreadStateException"".

# Thread class vs Runnable interface

1. By extending the Thread class, we cannot extend any other class, as Java does not allow multiple inheritances while implementing the Runnable interface; we can also extend other base class(if required).

2. By extending the Thread class, each of thread creates the unique object and associates with it while implementing the Runnable interface; multiple threads share the same object

3. Thread class provides various inbuilt methods such as getPriority(), isAlive and many more while the Runnable interface provides a single method, i.e., run().

# thread daemon

The daemon threads are the low priority threads that provide the background support and services to the user threads. Daemon thread gets automatically terminated by the JVM if the program remains with the daemon thread only, and all other user threads are ended/died. There are two methods for daemon thread available in the Thread class:

public void setDaemon(boolean status): It used to mark the thread daemon thread or a user thread.

public boolean isDaemon(): It checks the thread is daemon or not.

# thread interrupt

We should interrupt a thread when we want to break out the sleep or wait state of a thread. We can interrupt a thread by calling the interrupt() throwing the InterruptedException.

# thread lifecycle states

1. New: In this state, a Thread class object is created using a new operator, but the thread is not alive. Thread doesn't start until we call the start() method.

2. Runnable: In this state, the thread is ready to run after calling the start() method. However, the thread is not yet selected by the thread scheduler.

3. Running: In this state, the thread scheduler picks the thread from the ready state, and the thread is running.

4. Waiting/Blocked: In this state, a thread is not running but still alive, or it is waiting for the other thread to finish.

5. Dead/Terminated: A thread is in terminated or dead state when the run() method exits.

# thread pool

1. Java Thread pool represents a group of worker threads, which are waiting for the task to be allocated.

2. Threads in the thread pool are supervised by the service provider which pulls one thread from the pool and assign a job to it.

3. After completion of the given task, thread again came to the thread pool.

4. The size of the thread pool depends on the total number of threads kept at reserve for execution.

The advantages of the thread pool are :

1. Using a thread pool, performance can be enhanced.

2. Using a thread pool, better system stability can occur.

# thread safety

If a method or class object can be used by multiple threads at a time without any race condition, then the class is thread-safe. Thread safety is used to make a program safe to use in multithreaded programming. It can be achieved by the following ways:

1. Synchronization

2. Using Volatile keyword

3. Using a lock based mechanism

4. Use of atomic wrapper classes

# Thread Scheduler

In Java, when we create the threads, they are supervised with the help of a Thread Scheduler, which is the part of JVM. Thread scheduler is only responsible for deciding which thread should be executed. Thread scheduler uses two mechanisms for scheduling the threads: Preemptive and Time Slicing.

Java thread scheduler also works for deciding the following for a thread:

1. It selects the priority of the thread.

2. It determines the waiting time for a thread

3. It checks the Nature of thread

# Thread vs AsyncTask

A Thread is generally used for long tasks to be run in the background. We need a Handler class to use a Thread.

An AsyncTask is an intelligent Thread subclass. It's recommended to use AsyncTask when the caller class is the UI Thread as there is no need to manipulate the handlers.

AsyncTask is generally used for small tasks that can communicate back with the main UI thread using the two methods onPreExecute() and onPostExecute() it has. A Handler class is preferred when we need to perform a background task repeatedly after every x seconds/minutes.

# thread vs coroutine

1. A thread is an execution unit that can run independently from other threads.

1-1. A coroutine is a unit of execution that can be suspended and resumed, allowing it to share resources with other concurrent or parallel executions.

2. Threads are typically heavier than coroutines, so they can be more expensive in terms of performance. However, this is not always the case, and it really depends on the specific implementation.

2-1. In general, coroutines tend to be more efficient when it comes to CPU usage, but threads may be better when it comes to I/O bound tasks.

# thread vs process

1. A Program in the execution is called the process whereas; A thread is a subset of the process

2. Processes are independent whereas threads are the subset of process.

3. Process have different address space in memory, while threads contain a shared address space.

4. Context switching is faster between the threads as compared to processes.

5. Inter-process communication is slower and expensive than inter-thread communication.

6. Any change in Parent process doesn't affect the child process whereas changes in parent thread can affect the child thread.

# Thread vs Service

1. Service is an application component that facilitates an application to run in the background in order to perform long-running operations without user interaction. / A Thread is a concurrent unit of execution.

2. Service exposes few functionalities to other applications by calling Context.bindService(). / For Thread, Google has brought in handlers and loopers into threads.

3. When an application is killed, service is not killed. / When an application is killed, the thread is killed.

# throw vs throws

1. The throw keyword is used to throw an exception explicitly.

1-1. The throws keyword is used to declare an exception.

2. The checked exceptions cannot be propagated with throw only.

2-1. The checked exception can be propagated with throws

3. The throw keyword is followed by an instance.

3-1. The throws keyword is followed by class.

4. The throw keyword is used within the method.

4-1. The throws keyword is used with the method signature.

5. You cannot throw multiple exceptions.

5-1. You can declare multiple exceptions, e.g., public void method()throws IOException, SQLException.

# throws vs throw

1. The throw keyword is used to throw an exception explicitly.

1-1. The throws keyword is used to declare an exception.

2. The checked exceptions cannot be propagated with throw only.

2-1. The checked exception can be propagated with throws

3. The throw keyword is followed by an instance.

3-1. The throws keyword is followed by class.

4. The throw keyword is used within the method.

4-1. The throws keyword is used with the method signature.

5. You cannot throw multiple exceptions.

5-1. You can declare multiple exceptions, e.g., public void method()throws IOException, SQLException.

# time slicing vs preemptive scheduling

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

# Toast

Toast is a message that pops up on the screen. It is used to display the message regarding the status of the operation initiated by the user and covers only the expanse of space required for the message while the user’s recent activity remains visible and interactive.

Toast notification automatically fades in and out and it does not accept interaction events.

# toString()

The toString() method returns the string representation of an object. If you print any object, java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object, etc. depending upon your implementation. By overriding the toString() method of the Object class, we can return the values of the object, so we don't need to write much code.

# transform COLD observable to a HOT

1. COLD to HOT

One way to make a Cold observable Hot is by using publish().connect(). publish() converts the Cold observable to a ConnectableObservable, which pretty much behaves like a Hot one. Once triggered with the .connect() operator, it’ll publish events regardless if there are any subscribers.

Another way to transform a Cold observable to a Hot one is by wrapping it with a Subject. The Subject subscribes to the Cold observable immediately and exposes itself as an Observable to future subscribers. Again, the work is performed regardless whether there are any subscribers … and on the other hand multiple subscribers to the Subject won’t trigger the initial work multiple times.

2. HOT to COLD

The first way of transforming (or rather “masking”) a Hot observable to Cold is by using the defer() operator. It defers the creation of the Hot observable altogether, so each new subscriber will trigger the work again (feature of a Cold observable).

Depending on the use-case the pattern mentioned above might be quite wasteful, so another strategy is using the replay().autoConnect(0) paradigm. The replay() operator will cache the values emitted by the Hot observable and re-emit them to future subscribers. autoConnect(0) returns an observable that can be triggered even when there are no subscribers to the underlaying Hot observable. The combination of both just replays cached values from the Hot observable as a Cold one.

# transform HOT to COLD

1. COLD to HOT

One way to make a Cold observable Hot is by using publish().connect(). publish() converts the Cold observable to a ConnectableObservable, which pretty much behaves like a Hot one. Once triggered with the .connect() operator, it’ll publish events regardless if there are any subscribers.

Another way to transform a Cold observable to a Hot one is by wrapping it with a Subject. The Subject subscribes to the Cold observable immediately and exposes itself as an Observable to future subscribers. Again, the work is performed regardless whether there are any subscribers … and on the other hand multiple subscribers to the Subject won’t trigger the initial work multiple times.

2. HOT to COLD

The first way of transforming (or rather “masking”) a Hot observable to Cold is by using the defer() operator. It defers the creation of the Hot observable altogether, so each new subscriber will trigger the work again (feature of a Cold observable).

Depending on the use-case the pattern mentioned above might be quite wasteful, so another strategy is using the replay().autoConnect(0) paradigm. The replay() operator will cache the values emitted by the Hot observable and re-emit them to future subscribers. autoConnect(0) returns an observable that can be triggered even when there are no subscribers to the underlaying Hot observable. The combination of both just replays cached values from the Hot observable as a Cold one.

# Transformer

A Transformer is a convenient way to encapsulate common operations in a reusable way. This encapsulated logic can be tested in isolation, which is easier, and further simplifies the tests of all chains that use it. For example if you find yourself repeating a sequence of operators multiple times in your code (e.g. .map(user -> user.age).filter(age -> age > 18)), this logic can be factored out in a UserLegalAgeTransformer. It can be reused in the chains like this .compose(new UserLegalAgeTransformer()).

# transient

If you define any data member as transient, it will not be serialized. By determining transient keyword, the value of variable need not persist when it is restored.

# TreeMap vs HashMap

1. HashMap maintains no order, but TreeMap maintains ascending order.

2. HashMap is implemented by hash table whereas TreeMap is implemented by a Tree structure.

3. HashMap can be sorted by Key or value whereas TreeMap can be sorted by Key.

4. HashMap may contain a null key with multiple null values whereas TreeMap cannot hold a null key but can have multiple null values.

# TreeSet vs HashSet

The HashSet and TreeSet, both classes, implement Set interface. The differences between the both are listed below.

1. HashSet maintains no order whereas TreeSet maintains ascending order.

2. HashSet impended by hash table whereas TreeSet implemented by a Tree structure.

3. HashSet performs faster than TreeSet.

4. HashSet is backed by HashMap whereas TreeSet is backed by TreeMap.

# two-way vs one-way data binding

Two-way data binding means that changes to either the model or the view will update the other automatically. One-way data binding means that changes to the model will update the view, but not vice versa.

# type inference in Kotlin

Type inference is the process of automatically determining the type of a variable or expression based on its value. In Kotlin, type inference determines the type of variables when they are first declared and the return type of functions.

# type promotion overloading

By Type promotion is method overloading, we mean that one data type can be promoted to another implicitly if no exact matching is found.

The byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int, long, float or double and so on. Consider the following example.

# unboxing, autoboxing

The autoboxing is the process of converting primitive data type to the corresponding wrapper class object, eg., int to Integer. The unboxing is the process of converting wrapper class object to primitive data type. For eg., integer to int. Unboxing and autoboxing occur automatically in Java. However, we can externally convert one into another by using the methods like valueOf() or xxxValue().

It can occur whenever a wrapper class object is expected, and primitive data type is provided or vice versa.

1. Adding primitive types into Collection like ArrayList in Java.

2. Creating an instance of parameterized classes ,e.g., ThreadLocal which expect Type.

3. Java automatically converts primitive to object whenever one is required and another is provided in the method calling.

4. When a primitive type is assigned to an object type.

# Unchecked Exception vs Checked Exception

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions, e.g., IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions, e.g., ArithmeticException, NullPointerException, etc. Unchecked exceptions are not checked at compile-time.

# Unit-returning

fun printHello(name : String?) : Unit {

if (name != null)

print(""Hello, $name!"")

else

print(""Hi there!"")

// We don't need to write 'return Unit.VALUE' or 'return', although we could

}

The purpose is the same as Java's void. Only Unit is a proper type, so it can be passed as a generic argument etc.

Why we don't call it ""Void"": because the word ""void"" means ""nothing"", and there's another type, Nothing, that means just ""no value at all"", i.e. the computation did not complete normally (looped forever or threw an exception). We could not afford the clash of meanings.

Why Unit has a value (i.e. is not the same as Nothing): because generic code can work smoothly then. If you pass Unit for a generic parameter T, the code written for any T will expect an object, and there must be an object, the sole value of Unit.

How to access that value of Unit: since it's a singleton object, just say Unit

UNIT actually contains valuable information, it basically just means ""DONE"". It just returns the information to the caller, that the method has been finished.

# unreference object

1. By nulling the reference

2. By assigning a reference to another

3. By anonymous object etc.

# usages of intent

Android Intents are used to

1. start an activity - startActivity(intent)

2. start a service - startService(intent)

3. deliver a broadcast - sendBroadcast(intent)

# val mutableList vs var immutableList

1. Immutable lists are frequently preferred for a variety of reasons:

1-1. They promote functional programming, in which state is passed on to the next function, which constructs a new state based on it, rather than being altered. This is evident in Kotlin collection methods like map, filter, reduce, and so forth.

1-2. It's often easier to understand and debug software that doesn't have any side effects (you can be sure that the value of an object will always be the one at its definition).

1-3. Because no write access is required in multi-threaded systems, immutable resources cannot induce race conditions.

2. However, there are some disadvantages of using immutable lists as well. They are as follows :

2-1. Copying large collections simply to add/remove a single piece is very expensive.

2-2. When you need to alter single fields frequently, immutability can make the code more difficult. Data classes in Kotlin provide a built-in copy() method that allows you to clone an instance while changing only part of the fields' values.

# val vs const

By default val properties are set at runtime. Adding a const modifier on a val would make a compile-time constant. A const cannot be used with a var or on its own. A const is not applicable on a local variable.

# val vs var

val variables cannot be changed. They’re like final modifiers in Java. A var can be reassigned. The reassigned value must be of the same data type.

# var immutableList vs val mutableList

1. Immutable lists are frequently preferred for a variety of reasons:

1-1. They promote functional programming, in which state is passed on to the next function, which constructs a new state based on it, rather than being altered. This is evident in Kotlin collection methods like map, filter, reduce, and so forth.

1-2. It's often easier to understand and debug software that doesn't have any side effects (you can be sure that the value of an object will always be the one at its definition).

1-3. Because no write access is required in multi-threaded systems, immutable resources cannot induce race conditions.

2. However, there are some disadvantages of using immutable lists as well. They are as follows :

2-1. Copying large collections simply to add/remove a single piece is very expensive.

2-2. When you need to alter single fields frequently, immutability can make the code more difficult. Data classes in Kotlin provide a built-in copy() method that allows you to clone an instance while changing only part of the fields' values.

# var vs val

val variables cannot be changed. They’re like final modifiers in Java. A var can be reassigned. The reassigned value must be of the same data type.

# variable default value

The local variables are not initialized to any default value, neither primitives nor object references.

# Vector vs ArrayList

1) ArrayList is not synchronized. / Vector is synchronized.

2) ArrayList is not a legacy class. / Vector is a legacy class.

3) ArrayList increases its size by 50% of the array size. / Vector increases its size by doubling the array size.

4) ArrayList is not thread-safe as it is not synchronized. / Vector list is thread-safe as it's every method is synchronized.

# View

The view is a class that represents the basic building block for UI components. A View occupies a rectangular area on the screen and is responsible for drawing and event handling. It is a superclass for all the UI components. The most common UI components are:

TextView

EditText

ImageView

Button

ProgressBar

CheckBox, etc.

# View Group

View Group is a collection of views and other child views. It is an invisible part and the base class for layouts.

# View vs ViewGroup

1. View is a simple rectangle box that responds to the user’s actions.

1-1. ViewGroup is the invisible container. It holds View and ViewGroup

2. View is the SuperClass of All component like TextView, EditText, ListView, etc.

2-1. ViewGroup is a collection of Views(TextView, EditText, ListView, etc..), somewhat like a container.

3. A View object is a component of the user interface (UI) like a button or a text box, and it’s also called a widget.

3-1. A ViewGroup object is a layout, that is, a container of other ViewGroup objects (layouts) and View objects (widgets)

4. Examples are EditText, Button, CheckBox, etc.

4-1. For example, LinearLayout is the ViewGroup that contains Button(View), and other Layouts also.

5. View refers to the android.view.View class.

5-1 ViewGroup refers to the android.view.ViewGroup class

6. android.view.View which is the base class of all UI classes.

6-1. ViewGroup is the base class for Layouts.

# ViewGroup vs View

1. View is a simple rectangle box that responds to the user’s actions.

1-1. ViewGroup is the invisible container. It holds View and ViewGroup

2. View is the SuperClass of All component like TextView, EditText, ListView, etc.

2-1. ViewGroup is a collection of Views(TextView, EditText, ListView, etc..), somewhat like a container.

3. A View object is a component of the user interface (UI) like a button or a text box, and it’s also called a widget.

3-1. A ViewGroup object is a layout, that is, a container of other ViewGroup objects (layouts) and View objects (widgets)

4. Examples are EditText, Button, CheckBox, etc.

4-1. For example, LinearLayout is the ViewGroup that contains Button(View), and other Layouts also.

5. View refers to the android.view.View class.

5-1 ViewGroup refers to the android.view.ViewGroup class

6. android.view.View which is the base class of all UI classes.

6-1. ViewGroup is the base class for Layouts.

# ViewModel

ViewModel is one of the most critical classes of the Android Jetpack Architecture Component that support data for UI components. Its purpose is to hold and manage the UI-related data. Moreover, its main function is to maintain the integrity and allows data to be serviced during configuration changes like screen rotations. Any kind of configuration change in Android devices tends to recreate the whole activity of the application. It means the data will be lost if it has not been saved and restored properly from the activity which was destroyed. To avoid these issues, it is recommended to store all UI data in the ViewModel instead of an activity.

# virtual

Yes, all functions in Java are virtual by default.

# visibility modifier kotlin

- public

- internal

- protected

- private

`public` is the default visibility modifier.

# visible activity

A visible activity is one that sits behind a foreground dialog. It is actually visible to the user, but not necessarily being in the foreground itself.

# volatile

Volatile keyword is used in multithreaded programming to achieve the thread safety, as a change in one volatile variable is visible to all other threads so one variable can be used by one thread at a time.

# Volley

Volley is an HTTP library that makes networking very easy and fast, for Android apps. It was developed by Google and introduced during Google I/O 2013. It was developed because there is an absence in Android SDK, of a networking class capable of working without interfering with the user experience. Although Volley is a part of the Android Open Source Project(AOSP), Google announced in January 2017 that Volley will move to a standalone library. It manages the processing and caching of network requests, and it saves developers valuable time from writing the same network call/cache code again and again.

# wait()

The wait() method is provided by the Object class in Java. This method is used for inter-thread communication in Java. The java.lang.Object.wait() is used to pause the current thread, and wait until another thread does not call the notify() or notifyAll() method.

# wait() vs sleep()

1) The wait() method is defined in Object class. / The sleep() method is defined in Thread class.

2) The wait() method releases the lock. / The sleep() method doesn't release the lock.

# when

when is the equivalent of `switch` in `Kotlin`. The default statement in a when is represented using the else statement.

`when` statments have a default break statement in them.

# when pros over switch

1. two or more choices:

when(number) {

1 -> println(""One"")

2, 3 -> println(""Two or Three"")

4 -> println(""Four"")

else -> println(""Number is not between 1 and 4"")

}

2. ""when"" without arguments:

when {

number < 1 -> print(""Number is less than 1"")

number > 1 -> print(""Number is greater than 1"")

}

3: Any type passed in ""when"":

fun describe(obj: Any): String =

when (obj) {

1 -> ""One""

""Hello"" -> ""Greeting""

is Long -> ""Long""

!is String -> ""Not a string""

else -> ""Unknown""

}

4:Smart casting:

when (x) {

is Int -> print(""X is integer"")

is String -> print(""X is string"")

}

5:Ranges:

when(number) {

1 -> println(""One"") //statement 1

2 -> println(""Two"") //statement 2

3 -> println(""Three"") //statement 3

in 4..8 -> println(""Number between 4 and 8"") //statement 4

!in 9..12 -> println(""Number not in between 9 and 12"") //statement 5

else -> println(""Number is not between 1 and 8"") //statement 6

}

# with:-

Context object: this

Return value: lambda result

When calling functions on context objects without supplying the lambda result, ‘with' is recommended.

# Workmanager pros Jetpack

WorkManager is an Android Jetpack library that allows you to schedule and manage background tasks in your app. It is designed to be used with tasks that are not time-critical, and it provides a number of features that make it a good choice for managing background work in Android apps. WorkManager is flexible, allowing you to specify constraints on when your tasks should run, and it is also able to intelligently schedule tasks based on the state of the app and the device. WorkManager is also able to persist tasks across device reboots, and it integrates with other Android Jetpack libraries to provide a complete solution for background work in Android apps.

# write-only class

A class can be made write-only by making all of the fields private. The write-only class will have only setter methods which set the value passed from the main method to the private fields. We cannot read the properties of the class because there is no getter method in this class.