# 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.

# 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

# 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

# !! 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.

# !! 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.

?. 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.

# ?. 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.

# ?: 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.

# == vs ===

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

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

# 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. Consider the following example.

If there is an abstract method in a class, that class must be abstract.

We can't use abstract and final both with a method.

The abstract class can never be instantiated even if it contains a constructor and all of its methods are implemented.

# abstract class vs interface

1. An abstract class can have a method body (non-abstract methods). / The interface has only abstract methods.

2. An abstract class can have instance variables. / An interface cannot have instance variables.

3. An abstract class can have the constructor. / The interface cannot have the constructor.

4. An abstract class can have static methods. / The interface cannot have static methods.

5. You can extend one abstract class. / You can implement multiple interfaces.

6. The abstract class can provide the implementation of the interface. / The Interface can't provide the implementation of the abstract class.

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

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

9. An abstract class can be extended using keyword extends. / An interface class can be implemented using keyword implements

10. A Java abstract class can have class members like private, protected, etc. / Members of a Java interface are public by default.

# 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.

(Abstract Class, Interface)

# access specifiers

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

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.

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

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

# action

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.

# 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.

For example, the Facebook start page where you enter your email/phone number and password to log in acts as an activity.

# activity lifecycle

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

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.

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

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

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

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

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?1621314355

# activity loops

Entire lifetime – activity happens between onCreate and onDestroy

Visible lifetime – activity happens between onStart and onStop

Foreground lifetime – activity happens between onResume and onPause

# activity vs fragment

An activity is a single, focused thing that the user can do. A fragment is a piece of an activity, which contributes its own UI. You would choose a fragment when you want to modularize your code, and reuse the fragment UI in multiple activities.

# activity vs service

Activities can be terminated or closed anytime the user wishes. On the other hand, services are designed to run in the background, and they can act independently.

Most of the services run continuously, irrespective of whether there are certain or no activities being executed.

Activities are designed to run in the foreground. / Services are mainly designed to run in the background. Foreground services are also available.

Activities are used when the user interface is necessary. / Services are used when the user interface is not necessary.

Activities are dependent. / Services act independently.

# 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.

# 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.

# 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).

# androidmanifest.xml

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

This file is essential in every Android application.

It is declared in the root directory.

This file performs providing a unique name to the java package, describing various components of the application such as activity, services, and many more, defining the classes which will implement these components.

# 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.

# 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.

# ArrayList vs Array

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.

# 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.

# 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.

# 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.

# 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.

# @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.

# bitmap vs 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.

# 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.

# 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.

# 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 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.

# Callable vs Runnable

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.

# 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.

# 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.

# 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.

# Collections vs Collection

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.

# 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.

All the required static member functions and member variables can be kept inside the companion object created.

# 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

compileSdkVersion:

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).

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.

targetSdkVersion:

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.

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).

# 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.

# 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

# const vs val

val properties are set at runtime by default. 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.

# 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 type of arguments

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

# constructors

\*\*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.

# 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.

Activity context is attached to the lifecycle of an activity.

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.

Application context is attached to the lifecycle of an application.

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

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.

# covariant return typet

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.

# 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 automatically generates code that mimics the code you would otherwise have hand-written. Because the code is generated at compile time, it's traceable and more performant than other reflection-based solutions such as Guice.

# dagger injecting vs manual

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.

# data binding

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.

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.

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

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.

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

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

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

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

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

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

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

>Only interfaces may be implemented by data classes.

# 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.

# 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.

# dependency injection pros

1. Reusability of classes and decoupling of dependencies: It's easier to swap out implementations of a dependency. Code reuse is improved because of inversion of control, and classes no longer control how their dependencies are created, but instead work with any configuration.

2. Ease of refactoring: The dependencies become a verifiable part of the API surface, so they can be checked at object-creation time or at compile time rather than being hidden as implementation details.

3. Ease of testing: A class doesn't manage its dependencies, so when you're testing it, you can pass in different implementations to test all of your different cases.

# 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 declarations

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.

# Dictionary

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

# encapsulation

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

Advantages of Encapsulation:

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.

# 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.

# event handling react-programming

Event handling 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 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.

# 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 type

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 built-in java

ArithmeticException: It is thrown when an exceptional condition has occurred in an arithmetic operation.

ArrayIndexOutOfBoundsException: It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

ClassNotFoundException: This Exception is raised when we try to access a class whose definition is not found

FileNotFoundException: This Exception is raised when a file is not accessible or does not open.

IOException: It is thrown when an input-output operation failed or interrupted

InterruptedException: It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.

NoSuchFieldException: It is thrown when a class does not contain the field (or variable) specified

NoSuchMethodException: It is thrown when accessing a method that is not found.

NullPointerException: This exception is raised when referring to the members of a null object. Null represents nothing

NumberFormatException: This exception is raised when a method could not convert a string into a numeric format.

RuntimeException: This represents an exception that occurs during runtime.

StringIndexOutOfBoundsException: It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string

IllegalArgumentException : This exception will throw the error or error statement when the method receives an argument which is not accurately fit to the given relation or condition. It comes under the unchecked exception.

IllegalStateException : This exception will throw an error or error message when the method is not accessed for the particular operation in the application. It comes under the unchecked exception.

# 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.

# 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.

# 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.

# 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

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

No, 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.

# 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).

Finally block can be used without a catch. According to the definition of finally block, it must be followed by a try or catch block, therefore, we can use try block instead of catch.

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).

# fragment

A fragment is a part or portion of an activity. It is modular in a sense that you can move around or combine with other fragments in a single activity. Fragments are also reusable.

# 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.

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

# 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 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.

# 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.

# Hilt

Hilt is a dependency injection library for Android that reduces the boilerplate of doing manual dependency injection in your project. Doing manual dependency injection requires you to construct every class and its dependencies by hand, and to use containers to reuse and manage dependencies.

Hilt provides a standard way to use Dependency Injection in your application by providing containers for every Android class in your project and managing their lifecycles automatically. Hilt is built on top of the popular Dependency Injection library Dagger to benefit from the compile-time correctness, runtime performance, scalability, and Android Studio support that Dagger provides.

# I/O 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.

# I/O hierarchy

OutputStream

---FileOutputStream

---ByteArrayOutputStream

---FilterOutputStream

------DataOutputStream

------BufferedOutputStream

------PrintStream

---PipedOutputStream

---ObjectOutputStream

InputStream

---FileInputStream

---ByteArrayInputStream

---FilterInputStream

------DataInputStream

------BufferedInputStream

------PushBackInputStream

---PipedInputStream

---ObjectInputStream

# 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 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.

# 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.

Types of inheritance:

1. Single-level inheritance

2. Multi-level inheritance

3. Multiple Inheritance

4. Hierarchical Inheritance

5. Hybrid Inheritance

# inheritance advantage

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.

# inline vs infix

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 in interface

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

# 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.

# 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:

Implicit Intent is used to invoke the system components. (Map geo location, dial a call, launch website)

Explicit Intent is used to invoke the activity class. (Start Broadcast Receivers, launch activiy, start background service)

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.

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 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.

# 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.

We can't declare an interface method static, because methods of an interface are abstract by default, and we can not use static and abstract together.

The Interface can't be final, because an interface needs to be implemented by the other class and if it is final, it can't be implemented by any class.

We can't define private and protected modifiers for the members in interfaces and they are implicitly public.

# nested interface

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

# 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.

# inter-thread communication

The process of communication between synchronized threads is termed as inter-thread communication.

Inter-thread communication is used to avoid thread polling in Java.

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.

It can be obtained by wait(), notify(), and notifyAll() methods.

# 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.

# Jetpack boilerplate code

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 Data Binding

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 Firebase

There is no one-size-fits-all answer to this question, as the best database solution for a given Android Jetpack app will depend on the specific needs of the app. However, in general, I would recommend using Firebase Realtime Database if you need a simple, fast, and lightweight database solution, and Cloud Firestore if you need a more robust and feature-rich solution.

# 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

The LiveData class is a key part of the Android Jetpack architecture. It is responsible for holding and managing the data that is displayed in the UI. The LiveData class is designed to be lifecycle-aware, meaning that it will automatically update the UI when the data changes. This makes it a very important part of the Android architecture and helps to keep the UI up-to-date with the latest data.

# Jetpack Navigatioin

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

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

Android Jetpack is a set of libraries, tools and guidance to help make development faster, easier and more reliable. By using Jetpack, developers can take advantage of the latest technology while still maintaining compatibility with older versions of Android. Jetpack also helps to manage common tasks such as background tasks, navigation, and user interface design, so developers can focus on building great app experiences.

# Jetpack Room

Room is a database library that is part of the Android Jetpack suite of libraries. Room makes it easy to work with SQLite databases on Android.

# Jetpack ViewModel

The ViewModel class is designed to store and manage UI-related data in a lifecycle conscious way. This means that the ViewModel can be used to retain data across configuration changes such as screen rotations. To use the ViewModel class, you first need to create a subclass of ViewModel. In this subclass, you can define all the data that you want to store. Once you have created your ViewModel subclass, you can then access it in your Activity or Fragment by calling the ViewModelProvider.get() method.

# Jetpack WorkManager

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.

# jobscheduler

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:

Tasks that should be done when the device is connected to a power supply.

Tasks that require a Wi-Fi connection or network access.

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

# 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

# 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.

# lateinit vs lazy

lateinit purpose is to delay the initialisation to a later point in time. / lazy 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.

lateinit is possible to initialise the object from anywhere in the program. / lazy, only the initializer lambda can be used to initialise it.

lateinit, multiple initializations are possible. / lazy, only a single initialisation is possible.

lateinit is not thread-safe. In a multi-threaded system, it is up to the user to correctly initialise. / lazy, thread-safety is enabled by default, ensuring that the initializer is only called once.

lateinit works only with var. / lazy works only with val.

lateinit, the isInitialized method is added to verify if the value has previously been initialised. / lazy is impossible to uninitialize a property.

lateinit, properties of primitive types are not allowed. / lazy is allowable on primitive type properties.

To decide using lateinit or lazy:

>Use lateInit if properties are mutable (i.e., they may change later).

>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.

>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 mode

Standard:

This launch mode generates an activity’s new instance in the task from which it originated.

It is possible to create several instances for the same activity.

For Example, suppose our current stack is A -> B -> C. Now, if we launch activity B again with the “standard” launch mode, then the new stack will be A -> B -> C -> B.

SingleTop:

This launch mode is similar to the Standard launch mode except if there exists an activity’s previous instance on the top of the stack, then a new instance will not be created.

But the intent will be sent to the activity’s existing instance.

For example, suppose our current stack is A -> B -> C. Now, if we launch the activity B again with “singleTop” launch mode,then the new stack will be A -> B -> C -> B.

Consider another example, where the current stack is A -> B -> C. Now, if we launch activity C again with the “singleTop” launch mode, then the stack will remain the same i.e., A -> B -> C. The intent will be passed to the onNewIntent() method.

SingleTask:

This launch mode will create a new task and push a new instance to the task as the root.

For example, suppose our current stack is A -> B -> C -> D. Now, if we launch activity B again with the “singleTask” launch mode, then the new stack will be A -> B. Here, a callback has been received on the old instance and C and D activities are destroyed.

SingleInstance:

This launch mode is similar to the SingleTask launch mode. But the system doesn’t support launching any new activities in the same task.

In a situation where the new activity is launched, it is launched in a separate task.

For example, Suppose our current stack is A -> B -> C. Now, if we launch the activity D with the “singleInstance” launch mode, then there will be two stacks:

A -> B -> C

D, If you call activity E, then it will be added to the first stack.

A -> B -> C -> E

D

Again if you Call the activity D, then it will call the same activity from the 2nd stack and pass the intent to onNewIntent()."

# launch/join vs async/await

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.

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 initialization

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.

# 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 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.

# LiveData

LiveData is a data holder class that can be observed within a given lifecycle. This means that you can observe LiveData objects for changes and update the UI accordingly. LiveData is especially useful in Android applications because it helps to avoid memory leaks and can automatically update the UI when data changes.

# LiveData vs Observables

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.

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 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.

# 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. The marker interface can be declared as follows.

# 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.

# 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 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.

# 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 variables

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

# MVC

A model can be defined as the data that will be used by the program. Commonly used examples of models in MVC are the database, a simple object holding data (such as any multimedia file or the character of a game), a file, etc.

A view is a way of displaying objects (user interfaces) within an application. This is the particular vertical through which end users will communicate.

A controller is the third vertical which is responsible for updating both models and views. It accepts input from users as well as performs the equivalent update. In other words, it is the controller which is responsible for responding to user actions.

# MVC action filter

An action filer is used to apply an application logic that runs before and after the controller action executes. These attributes can be applied to the entire controller or the just the controller section and can be used with fields, methods, classes, and properties. The 3 action filters of MVC ASP.NET are Authorize, Output Cache and Handle Error.

# MVC Controller

The controller is responsible for managing the interaction between the view and the model. When a request is raised by the user after interaction with the view, the controller handles it. It receives the request from the View, and based on it performs the actions using the Model.

# MVC Controller result

View Result

JSON Result

Content Result

Redirect Result

JavaScript Result

# MVC controller vs ViewModel

The ViewModel is used to hold the state of the View and the presentation logic. It encapsulates the state and the logic. It acts as a bridge between the Model and the View. This pattern is optional.

The Controller is a primary component of MVC and is used to performs actions in response to the user requests. It controls the flow of the application.

# MVC dependency injection

The dependency injection in MVC is a design pattern that is used to separate the responsibilities of making dependencies in a class. As a result, there is no dependency between the two implementations. It is a method of inversion of control.

It is very useful as no code has to be altered if an object depends on it, reducing the complexity of the application.

# MVC drawbacks

The model pattern is a little complex.

Inefficiency of data access in view.

With a modern user interface, it is difficult to use MVC.

You need multiple programmers for parallel development.

Multiple technology knowledge is required.

# MVC execution

Receive the first request for the application

Perform routing

Create an MVC request handler

Create Controller

Execute Controller

Invoke action

Execute Result

# MVC partial view

Partial View in MVC is used to display only a part of the view content. It is reusable and can be used in multiple places that help in reducing the code duplication. It works as a child view and helps us see a view within a parent view. The partial view is initialized with a copy of the ViewDataDictionary object.

# MVC pros/cons

pros:

1. Scalable

2. Easy to maintain, test, upgrade

3. Friendly development process

4. Great for complex application

5. Suitable for team base work

cons:

1. Have to design first before implement

2. Need time to learn the Pattern properly

3. Have to understand how to separate logics, Data, View

# MVP pros/cons

pros:

1. No code in the view. Replace View(xml) is very easy.

2. Presenter only handles logics

3. Easy to scale, maintain, test, upgrade

4. Great for complex application

5. Suitable for team base work

cons:

1. Need time to analyze and design

2. Need highly skilled professionals who know it

3. It is very complex and need time to learn

4. Not good for small projects as it is complex

# MVVM

MVVM is a software design pattern that is used for developing user interfaces. It is derived from MVC, but it uses a different approach to separating the concerns of the application. In MVVM, the ViewModel is responsible for handling the data and the business logic, while the View is responsible for displaying the data. This separation of concerns makes it easier to develop and maintain complex user interfaces.

# MVVM ad vs dad

Some advantages of using MVVM in Android development include the ability to easily bind data to UI elements, a clean separation of concerns between the View and ViewModel, and the ability to easily unit test the ViewModel.

Some disadvantages of using MVVM in Android development include the potential for a lot of boilerplate code, and the fact that it can be difficult to debug data bindings.

# MVVM dependency injection

Dependency injection is a technique used to remove the hard-coded dependencies from a class, making it more flexible and easier to test. In the context of MVVM, dependency injection can be used to inject view models into views, allowing the view model and view to be decoupled. This makes it easier to unit test the view model, as it can be tested in isolation.

# MVVM implement

There are a few different ways to implement MVVM on Android. One popular way is to use the Android Data Binding Library, which allows you to bind data directly to views in your layout file. Other popular tools and technologies used for MVVM on Android include RxJava and LiveData.

# MVVM pros/cons

pros:

1. Reduces amount of Code in Views(xml) file.

2. Strong Data binding that saves a lot of code

3. No need to manually refresh View. (live data)

4. Easy to test as only View-Model handles logics.

5. Repository handles API data fetching or services.

cons:

1. Difficult for beginners

2. Code reusability is difficult of views and view-model

3. Manaing Various components is pretty hard

# MVVM vs MVC

The main difference between MVVM and other design patterns is that MVVM separates the view (i.e. the user interface) from the model (i.e. the data). This separation allows for a more modular and testable codebase. Additionally, MVVM typically uses data binding to automatically update the view when the model changes, which further reduces the amount of code that needs to be written.

# MVVM without data binding

There are a few ways to achieve MVVM architecture outside of the official data binding library. One way is to use the LiveData and ViewModel classes from the Android Architecture Components library. Another way is to use the RxJava library to create observables that can be subscribed to by the UI.

# MVVM without library

There are a few ways to implement MVVM without using the support libraries. One way would be to use a library like RxJava to manage your data bindings. Another way would be to use the Data Binding Library from Google.

# 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.

# 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.

# nested interface

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

# 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 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.

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

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, nullable types, elvis

Null safety 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 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 reference vs interface reference

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-oriented paradigm

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.

Features>>>>>>>>>>>>>>

Follows the bottom-up approach in program design.

Focus on data with methods to operate upon the object's data

Includes the concept like Encapsulation and abstraction which hides the complexities from the user and show only functionality.

Implements the real-time approach like inheritance, abstraction, etc.

The examples of the object-oriented paradigm are C++, Simula, Smalltalk, Python, C#, etc.

# observables vs observers

Observables are objects that emit a stream of data, and observers are objects that consume that data.

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.

# OOP

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.

Features>>>>>>>>>>>>>>

Follows the bottom-up approach in program design.

Focus on data with methods to operate upon the object's data

Includes the concept like Encapsulation and abstraction which hides the complexities from the user and show only functionality.

Implements the real-time approach like inheritance, abstraction, etc.

The examples of the object-oriented paradigm are C++, Simula, Smalltalk, Python, C#, etc.

# oop vs obp

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.

Object-oriented languages do not have the inbuilt objects whereas Object-based languages have the inbuilt objects, for example, JavaScript has window object.

Examples of object-oriented programming are Java, C#, Smalltalk, etc. whereas the examples of object-based languages are JavaScript, VBScript, etc.

# open

By default classes are final in Kotlin. To make them non-final, you need to add the `open` modifier.

# open vs 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.

# 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.

In Java, method overloading is not possible by changing the return type of the program due to avoid the ambiguity.

We cannot overload the methods by just applying the static keyword to them(number of parameters and types are the same).

# overloading vs overriding

1) Method overloading increases the readability of the program. / Method overriding provides the specific implementation of the method that is already provided by its superclass.

2) Method overloading occurs within the class. / Method overriding occurs in two classes that have IS-A relationship between them.

3) Method overloading, the parameters must be different. / Method overriding, the parameters must be the same.

# overridden method

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.

# overridden throws clause

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.

# override static

No, we can't override static methods.

# 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.

3. Two classes must have an IS-A relationship between them.

We can't override the static method because they are the part of the class, not the object.

We can override the overloaded method.

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.

# package

A package is a group of similar type of classes, interfaces, and sub-packages. It provides access protection and removes naming collision. The packages in Java can be categorized into two forms, inbuilt package, and user-defined package. There are many built-in packages such as Java, lang, awt, javax, swing, net, io, util, sql, etc.

# permission

Permissions allow certain restrictions to be imposed primarily to protect data and code. Without these, codes could be compromised, resulting to defects in functionality.

# pointer java

The pointer is a variable that refers to the memory address. They are not used in Java because they are unsafe(unsecured) and complex to understand.

# 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.

# 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.

# 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.

# 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.

# 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.

# remove duplicates ArrayList

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.

# Retrofit

Retrofit is type-safe REST client for Android and Java which aims to make it easier to consume RESTful web services.

# Runnable vs Callable

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.

# 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.

# 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.

# scope functions

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.

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.

with:-

Context object: this

Return value: lambda result

When calling functions on context objects without supplying the lambda result, ‘with' is recommended.

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.

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.

# sdk

The Google Android SDK(Android software development kit) is a toolset used by developers to write applications on Android-enabled devices.

Android Emulator - Android Emulator is a software application that simulates Android devices on computer so that you can test the application on a variety of devices and Android API levels without having each physical device.

DDMS(Dalvik Debug Monitoring Services) - It is a debugging tool from the SDK which provides services like message formation, call spoofing, capturing screenshots, etc.

ADB(Android Debug Bridge) - It is a command-line tool used to allow and control communication with the emulator instance.

AAPT(Android Asset Packaging Tool) - It is a build tool that gives the ability to developers to view, create, and update ZIP-compatible archives (zip, jar, and apk)."

# sealed class

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.

Another distinguishing aspect of sealed classes is that their constructors are by default private. Due to the fact that a sealed class is automatically abstract, it cannot be instantiated.

# sensor

Position Sensor: It is used for measuring the physical position of the Android device. This has orientation sensors and magnetometers.

Motion Sensors: These sensors consist of gravity, rotational activity, and acceleration sensors which measure the rotation of the device or the acceleration, etc.

Environmental Sensor: It includes sensors that measure temperature, humidity, pressure, and other environmental factors."

# sensor class

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.

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.

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.

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 void onAccuracyChanged(Sensor sensor, int accuracy) which is called when sensor accuracy is changed and

void onSensorChanged(SensorEvent event) 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. / 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. / The Externalizable interface provides control of the serialization logic to the programmer.

3) It is easy to implement but has the higher performance cost. / It is used to perform the serialization and often result in better performance.

4) No class constructor is called in serialization. / 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.

Serializable:

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.

Reflection is used during the process and many additional objects are created. This leads to plenty of garbage collection and poor performance.

Parcelable:

Parcelable is an Android-specific interface. In this approach, you implement the serialization yourself.

Reflection is not used during this process and hence no garbage is created.

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.

# 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.

# service vs thread

>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.

>Service exposes few functionalities to other applications by calling Context.bindService(). For thread Google has brought in handlers and loopers into threads.

>When an application is killed, service is not killed. When an application is killed, the thread is killed."

# 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.

# shallow copy

Object cloning

# 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.

# 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.

# 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.

# 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.

# 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.

# 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 import

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

Less coding is required if you have access any static member of a class oftenly, but If you overuse the static import feature, it makes the program unreadable and unmaintainable.

# static in abstract

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.

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. Consider the following example.

# static kotlin

Kotlin doesn't have the static keyword. To create static method in our class we use the `companion object`

# static method vs instance method

1)A method that is declared as static is known as the static method. / 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. / 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. / 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;} / For example: public void msg(){...}.

# static restriction

The static method can not use non-static data member or call the non-static method directly.

this and super cannot be used in static context as they are non-static.

# 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).

# 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.

# 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 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 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.

# super

super can be used to refer to the immediate parent class instance variable.

super can be used to invoke the immediate parent class method.

super() can be used to invoke immediate parent class constructor.

# 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.

# 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.

# suspending function

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.

# 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

# synchronize ArrayList

1. Using Collections.synchronizedList() method

2. Using CopyOnWriteArrayList<T>

# 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.

# this

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.

Uses:

this can be used to refer to the current class instance variable.

this can be used to invoke current class method (implicitly)

this() can be used to invoke the current class constructor.

this can be passed as an argument in the method call.

this can be passed as an argument in the constructor call.

this can be used to return the current class instance from the method.

# this 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.

# this static

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 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.

# 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.

# 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 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 Runnable

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().

# throw vs throws

1) The throw keyword is used to throw an exception explicitly. / The throws keyword is used to declare an exception.

2) The checked exceptions cannot be propagated with throw only. / The checked exception can be propagated with throws

3) The throw keyword is followed by an instance. / The throws keyword is followed by class.

4) The throw keyword is used within the method. / The throws keyword is used with the method signature.

5) You cannot throw multiple exceptions. / 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.

# 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.

# try, catch, final

It is not necessary that each try block must be followed by a catch block. It should be followed by either a catch block OR a finally block. So whatever exceptions are likely to be thrown should be declared in the throws clause of the method.

# 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.

# unreference object

1. By nulling the reference

2. By assigning a reference to another

3. By anonymous object etc.

# val mutableList / var immutableList

We use a mutable list if the collection will alter as part of the design. On the other hand, we use an immutable list if the model is only meant to be viewed.

val and var serve a distinct purpose than immutable and mutable lists. The val and var keywords specify how a variable's value/reference should be handled. We use var when the value or reference of a variable can be altered at any moment. On the other hand, we use val when a variable's value/reference can only be assigned once and cannot be modified later in the execution.

Immutable lists are frequently preferred for a variety of reasons:

>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.

>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).

>Because no write access is required in multi-threaded systems, immutable resources cannot induce race conditions.

However, there are some disadvantages of using immutable lists as well. They are as follows :

>Copying large collections simply to add/remove a single piece is very expensive.

>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.

# 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.

# ViewModel implement

The recommended way of implementing a ViewModel class in an Android app is to use the Android Architecture Components library. This library provides a ViewModel class that is specifically designed for use with Android apps. The ViewModel class is designed to help manage data that is specific to a particular activity or fragment, and it is also designed to survive configuration changes such as screen rotations.

# virtual

all functions in Java are virtual by default.

# visibility modifiers

public (default)

internal

protected

private

# 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.

# 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.

# wait(), 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.

# 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.