**1. Why should we use Kotlin?**

Kotlin is concise

Kotlin is null-safe

Kotlin is interoperable

2.**How to ensure null safety in Kotlin?**

One of the major advantages of using Kotlin is null safety. In Java, if you access some null variable then you will get a NullPointerException.

So, to assign null values to a variable, you need to declare the name variable as a nullable string and then during the access of this variable, you need to use a safe call operator i.e. ?.

var name: String? = "MindOrks"

print(name?.length) // ok

name = null // ok

**3.What is the difference between safe calls(?.) and null check(!!)?**

Safe call operator i.e. ?. is used to check if the value of the variable is null or not. If it is null then null will be returned otherwise it will return the desired value.

If you want to throw NullPointerException when the value of the variable is null, then you can use the null check or !! operator.

var name: String? = "MindOrks"

println(name?.length) // 8

name = null

println(name!!.length) // KotlinNullPointerException

**What is the difference between val and const val?**

const and val both represents the immutability and read only values and act as final keyword in java. val keyword must be used to declare for run time values and const keyword must be used to declare compile time values

**Scope Functions:**

The Kotlin standard library contains several functions whose sole purpose is to execute a block of code within the context of an object. When you call such a function on an object with a lambda expression provided, it forms a temporary scope. In this scope, you can access the object without its name. Such functions are called scope functions. There are five of them: let, run, with, apply, and also

Person("Alice", 20, "Amsterdam").let {

println(it)

it.moveTo("London")

it.incrementAge()

println(it)}

If you write the same without let, you'll have to introduce a new variable and repeat its name whenever you use it.

| **Function** | **Object reference** | **Return value** | **Is extension function** |
| --- | --- | --- | --- |
| Let | It | Lambda result | Yes |
| Run | This | Lambda result | Yes |
| Run | - | Lambda result | No: called without the context object |
| With | This | Lambda result | No: takes the context object as an argument. |
| Apply | This | Context object | Yes |
| Also | It | Context object | Yes |

* Executing a lambda on non-null objects: **let**
* Introducing an expression as a variable in local scope: **let**
* Object configuration: **apply**
* Object configuration and computing the result: **run**
* Running statements where an expression is required: non-extension **run**
* Additional effects: **also**
* Grouping function calls on an object: **with**

**let** is often used for executing a code block only with non-null values.

**with**﻿

A non-extension function: the context object is passed as an argument, but inside the lambda, it's available as a receiver (this). The return value is the lambda result.

**run**﻿

The context object is available as a receiver (this). The return value is the lambda result.

run does the same as with but invokes as let - as an extension function of the context object.

run is useful when your lambda contains both the object initialization and the computation of the return value.

**apply﻿**

**The context object** is available as a receiver (this). **The return value** is the object itself.

Use apply for code blocks that don't return a value and mainly operate on the members of the receiver object. The common case for apply is the object configuration. Such calls can be read as “ *apply the following assignments to the object.*”

val adam = Person("Adam").apply {

age = 32

city = "London"

}

println(adam)

### also﻿

**The context object** is available as an argument (it). **The return value** is the object itself.

also is good for performing some actions that take the context object as an argument. Use also for actions that need a reference to the object rather than its properties and functions, or when you don't want to shadow this reference from an outer scope.

val numbers = mutableListOf("one", "two", "three")

numbers

.also { println("The list elements before adding new one: $it") }

.add("four")

**How many constructors are available in Kotlin?**  
Two types of constructors available in Kotlin are:

1. Primary constructor
2. Secondary constructor

**How can you handle null exceptions in Kotlin?**

Elvis Operator is used for handling null expectations in Kotlin.

**What are some of the features which are there in Kotlin but not In Java?**

Here, are few important Kotlin features that Java doesn’t have:

1. Null Safety
2. Operator Overloading
3. Coroutines
4. Range expressions
5. Smart casts
6. Companion Objects

**default behavior of Kotlin classes?**

In Kotlin all classes are final by default.

**Does Kotlin support primitive Datatypes?**

No, Kotlin does not provide support for primitive Data types like in Java.

### What is the use of the open keyword in Kotlin?

In Kotlin, the classes and functions are final by default. So, it is not possible to inherit the class or override the functions. To achieve this, we need to use the open keyword before the class and function.

**when** Usage in Kotlin which replace switch in java

1. when(number) {
2. 1 -**>** println("One")
3. 2, 3 -**>** println("Two or Three")
4. 4 -**>** println("Four")
5. else -**>** println("Number is not between 1 and 4")
6. }

**"when" without arguments:**

1. when {
2. number **<** **1** -**>** print("Number is less than 1")
3. number **>** 1 -**>** print("Number is greater than 1")
4. }

What is Lateinit in Kotlin, and when is it used?

Lateinit means late initialization. It is used when you do not want to initialize a variable in the constructor and instead initialize it later.

You should declare that variable with lateinit keyword to guarantee the initialization, not before using it. It will not allocate memory until it is initialized. You cannot use lateinit for primitive type properties like Int, Long, etc.

1. lateinit var test: String
2. fun doSomething() {
3. test = "Some value"
4. println("Length of string is "+test.length)
5. test = "change value"
6. }

What is the difference between == operator and === operator in Kotlin?

In Kotlin, the == operator is generally used to compare the values stored in variables, and the === operator is used to check if the reference of the variables are equal or not.

In the case of primitive types, the === operator is also used to check for the value and not reference.

**Example:**

1. // primitive example
2. val int1 = 10
3. val int2 = 10
4. println(int1 == int2) // true
5. println(int1 === int2) // true
6. // wrapper example
7. val num1 = Integer(10)
8. val num2 = Integer(10)
9. println(num1 == num2) // true
10. println(num1 === num2) //false

### Can we use primitive types such as int, double, float in Kotlin?

Kotlin doesn't support the primitive types so, we can't use primitive types directly in Kotlin. We can use classes like Int, Double, etc., as an object wrapper for primitives.

**How to initialize an array in Kotlin with values?**

In Java an array can be initialized such as:

int numbers[] = new int[] {10, 20, 30, 40, 50}

How does Kotlin's array initialization look like?

Answer

val numbers: IntArray = intArrayOf(10, 20, 30, 40, 50)

### What is the use of Companion Objects in Kotlin?

Companion Objects are required in Kotlin because Kotlin doesn't have static members or member functions. If we need to write a function that can be called without having a class instance but needs access to the internals of a class, we can write it as a member of a companion object declaration inside that class.

1. class EventManager {
2. companion object FirebaseManager {
3. }
4. }
5. val firebaseManager = EventManager.FirebaseManager

The companion object is a singleton, and it is a proper object which you can assign to a variable and pass it around

### How can you handle null exceptions in Kotlin?

In Kotlin, Elvis Operator is used to handling null expectations.

### What is the difference between lateinit and lazy in Kotlin?

Following are the key differences between lateinit and lazy in Kotlin:

* In Kotlin, lazy can only be used for val properties while lateinit can only be applied to var because it can't be compiled to a final field. Thus no immutability can be guaranteed.
* lazy initialization’ was designed to prevent unnecessary initialization of objects
* You have to use lateinit, if you want your property to be initialized from outside in a way probably unknown beforehand.
* lazy → It initializes variable only when it is required for the first time.

**What is difference between companion object and object?**

Companion Object is initialized when class is loaded. But Object is initialized lazily by default — when accessed for the first time

### What are data classes in kotlin?

### Data class is a simple class which is used to hold data/state and contains standard functionality. A data keyword is used to declare a class as a data class.

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

Declaring a data class must contains at least one primary constructor with property argument (val or var).

Only the property name will be used inside the toString(), equals(), hashCode(), and copy() implementations.

Due to presence of above functions internally in data class, the data class eliminates the boilerplate code.

## **Requirements of data class:**

In order to create a data class, we need to fulfill the following requirements:

* Contain primary constructor with at least one parameter.
* Parameters of primary constructor marked as val or var.
* Data class cannot be abstract, inner, open or sealed.
* Before 1.1,data class may only implements interface. After that data classes may extend other classes.

**Kotlin Sealed Class:**

Sealed class is a class which restricts the class hierarchy. A class can be declared as sealed class using "sealed" keyword before the class name. It is used to represent restricted class hierarchy. Sealed class is used when the object have one of the types from limited set, but cannot have any other type.

The constructors of sealed classes are private in default and cannot be allowed as non-private.

**Declaration of sealed class**

sealed class MyClass

The subclasses of sealed classes must be declared in the same file in which sealed class itself. Sealed classes are commonly used with when expression. As the sub classes of sealed classes have their own types act as a case. Due to this, when expression in sealed class covers all the cases and avoid to add else clause.

**Kotlin Generics:**

Generics are the powerful features that allow to define classes, methods, and properties etc. which can be accessed using different types. The type differences of classes, methods, etc. are checked at compile-time.

The generic type class or method is declared as parameterized type. The parameterized types are declared using angle brackets <> Generics are mostly used in collections.

## **Advantage of Generics:**

Following are the key advantages of using generics:

* **Type-safety:** Generic allows to hold only single type of object. Generic does not allow to store other object.
* **Type casting is not required:** There is no need to typecast the object.
* **Compile time checking:** Generics code is checked at compile time so that it can avoid any problems at runtime.

### Explain suspend function in the context of Kotlin.

### 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.

### 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.

* **delay()** function is an example of suspend function.

### What are inline functions ?

### Inline function instruct compiler to insert complete body of the function wherever that function got used in the code. To use an Inline function, all you need to do is just add an inline keyword at the beginning of the function declaration.

### What are Higher Order Functions?

### A higher-order function is a function that takes functions as parameters, or returns a function.

### List down the visibility modifiers available in Kotlin. What’s the default visibility modifier?

* + public
  + internal
  + protected
  + private

public is the default visibility modifier.

### What are the types of constructors in Kotlin? How are they different? How do you define them in your class?

Constructors in Kotlin are of two types:  
**Primary** – These are defined in the class headers. They cannot hold any logic. There’s only one primary constructor per class.  
**Secondary** – They’re defined in the class body. They must delegate to the primary constructor if it exists. They can hold logic. There can be more than one secondary constructors.

class User(name: String, isAdmin: Boolean){

constructor(name: String, isAdmin: Boolean, age: Int) :this(name, isAdmin)

{

this.age = age

}

}

### How to create Singleton classes?

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

object MySingletonClass

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

**What is coroutine in kotlin**

A **coroutine** is a concurrency design pattern that you can use on **Android** to simplify code that executes asynchronously. ... On **Android**, **coroutines** help to manage long-running tasks that might otherwise block the main thread and cause your app to become unresponsive.

It enable you to write clean, simplified asynchronous code that keeps your app responsive while managing long-running tasks such as network calls or disk operations.

[Coroutines](https://kotlinlang.org/docs/reference/coroutines/coroutines-guide.html) were added to Kotlin in version 1.3

* **Lightweight**: You can run many coroutines on a single thread due to support for [*suspension*](https://kotlinlang.org/docs/reference/coroutines/basics.html), which doesn't block the thread where the coroutine is running. Suspending saves memory over blocking while supporting many concurrent operations. One can run many coroutines on a single thread due to support for suspension, which doesn’t block the thread where the coroutine is running.
* **Fewer memory leaks**: Use [*structured concurrency*](https://kotlinlang.org/docs/reference/coroutines/basics.html#structured-concurrency) to run operations within a scope.
* **Built-in cancellation support**: [Cancellation](https://kotlinlang.org/docs/reference/coroutines/cancellation-and-timeouts.html) is propagated automatically through the running coroutine hierarchy.
* **Jetpack integration**: any Jetpack libraries include extensions that provide full coroutines support.

## **Start a coroutine**

You can start coroutines in one of two ways:

* [launch](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/launch.html) starts a new coroutine and doesn't return the result to the caller. Any work that is considered "fire and forget" can be started using launch.
  + The launch will not block the main thread, but on the other hand, the execution of the rest part of the code will not wait for the launch result since launch is not a suspend call.
  + launch{} cannot be used when you need the parallel execution of network calls.
* [async](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines/async.html) starts a new coroutine and allows you to return a result with a suspend function called await.
  + it blocks the main thread at the entry point of the await() function in the program.
  + When making two or more network call in parallel, but you need to wait for the answers before computing the output, ie use async for results from multiple tasks that run in parallel.

**There are basically 3 scopes in Kotlin coroutines:**

* Global Scope.
* LifeCycle Scope.
* ViewModel Scope

1. **Global Scope:**

Global Scope is one of the ways by which coroutines are launched. When Coroutines are launched within the global scope, they live long as the application does. If the coroutines finish it’s a job, it will be destroyed and will not keep alive until the application dies.

Coroutines launched in the global scope will be launched in a separate thread.

1. **LifeCycle Scope:**

The lifecycle scope is the same as the global scope, but the only difference is that when we use the lifecycle scope, all the coroutines launched within the activity also dies when the activity dies. It is beneficial as our coroutines will not keep running even after our activity dies.

  // launching the coroutine in the lifecycle scope

            lifecycleScope.launch {

**while** (**true**) {

                    delay(1000L)

                    Log.d(TAG, "Still Running..")

                }

            }

1. **ViewModel Scope:**

It is also the same as the lifecycle scope, only difference is that the coroutine in this scope will live as long the view model is alive. ViewModel is a class that manages and stores the UI-related data by following the principles of the lifecycle system in android.

## **Executing in a background thread:**

Making a network request on the main thread causes it to wait, or block, until it receives a response. Since the thread is blocked, For a better user experience, let's run this operation on a background thread.

**Coroutines are basically of two types:**

* Stackless
* Stackful

**Kotlin implements stackless coroutines,**it means that the coroutines don’t have their own stack, so they don’t map on the native thread.

**Why we need coroutines?**

As we know android developers today have many async tools in hand. These include RxJava, AsyncTasks, Jobs, Threads. So why there is a need to learn something new?

* While Using Rx, it requires a lot of effort to get it enough, to use it safely. On the Other hand, AsyncTasks and threads can easily introduce leaks and memory overhead. Even using these tools after so many disadvantages, the code can suffer from callbacks, which can introduce tons of extra code..
* Android is a single thread platform, By default, everything runs on the main thread. In Android, almost every application needs to perform some non UI operations like (Network call, I/O operations), so when coroutines concept is not introduced, what is done is that programmer dedicate this task to different threads, each thread executes the task given to it, when the task is completed, they return the result to UI thread to update the changes required. Though In android there is a detailed procedure given, about how to perform this task in an effective way using best practices using threads, this procedure includes lots of callbacks for passing the result among threads, which ultimately introduce tons of code in our application and the waiting time to get the result back to increases.
* On Android, Every app has a main thread (which handles all the UI operations like drawing views and other user interactions. If there is too much work happening on this main thread, like network calls (eg fetching a web page), the apps will appear to hang or slow down leading to poor user experience.

**Kotlin Coroutines vs Threads:**

* Fetching the data from one thread and passing it to another thread takes a lot of time. It also introduces lots of callbacks, leading to less readability of code. On the other hand, coroutines eliminate callbacks.
* Creating and stopping a thread is an expensive job, as it involves creating their own stacks, whereas creating coroutines is very cheap when compared to the performance it offers. coroutines do not have their own stack.
* Threads are blocking, whereas coroutines are suspendable. By blocking it means that when a thread sleeps for some duration, the entire threads get blocked, it cannot do any other operation, whereas since coroutines are suspendable, so when they are delayed for some seconds, they can perform any other work.
* Coroutines offer a very high level of concurrency as compared to threads, as multiple threads involve blocking and context switching. Context switching with threads is slower as compared to coroutines, as with threads context can only be switched when the job of 1 thread gets over, but with coroutines, they can change context any time, as they are suspendable.
* Threads are not lifecycle aware. They do not have any knowledge of Lifecycle components (Activity, Fragment, ViewModel). A thread will be running even if UI component is destroyed which requires us to handle clean up and memory leaks.
* Coroutines are light and super fast. Coroutines are faster than threads, as threads are managed by Operating System, whereas coroutines are managed by users. Having thousands of coroutines working together are much better than having tens of threads working together.

|  |
| --- |
| //pseudo kotlin code for demonstration  suspend fun fetchAndShowUser()  {    //fetch on IO thread    val user = fetchUser()    // back on UI thread    showUser(user)  } |

**Advantages of coroutines:**

* using coroutines is much better to use in terms of readability and performance as well.
* **It is nonblocking, easy, and nonexpensive to create.**

By lightweight, it means that creating coroutines doesn’t allocate new threads. Instead, they use predefined thread pools and smart scheduling for the purpose of which task to execute next and which tasks later.



In snippet 1, We are calling fun1 and fun2 methods sequentially on the main thread. Execution will have a delay of 1 second during which thread would be blocked and would not be able to perform any other task. Now let’s try to write same code using coroutine.

In snippet 2, It looks like they are running in parallel but that’s not possible since both of them are getting executed by a single thread. Both functions are running concurrently and that’s because delay function does not block the thread, it suspends the thread so now without wasting time same thread can start performing next task and can return to it once the other suspended function (delay) returns to it.

A coroutine can provide a very high level of concurrency with very small overhead. Multiple threads can also provide parallelism but there is blocking and context switching. Coroutine suspends the thread and does not block it so that it can switch to another work.

E.x.

* **Add the following dependencies in build.gradle.**

implementation 'org.jetbrains.kotlin:kotlin-stdlib'

implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.9'

* **Add the following code in MainActivity.kt.**

override fun onCreate(savedInstanceState: Bundle?)

{

GlobalScope.launch {

val time = measureTimeMillis {

val one = sampleOne()

val two = sampleTwo()

println("The answer is ${one + two}")

}

println("Completed in $time ms")

}

println("EOF")

}

private suspend fun sampleOne(): Int {

println( "sampleOne"+System.currentTimeMillis())

delay(1000L) // pretend we are doing something useful here

return 10

}

private suspend fun sampleTwo(): Int {

println( "sampleTwo"+System.currentTimeMillis())

delay(1000L) // pretend we are doing something useful here, too

return 10

}

**withContext:** It is a scope function that allows us to create a new cancelable coroutine. If we pass a CoroutineContext arg, withContext merges the parent context and our arg to create a new CoroutineContext, then executes the coroutine within this merged context.

**val** time = measureTimeMillis { **val** dispatcher = newFixedThreadPoolContext(2, "withc") withContext(dispatcher) { doTheTask(DELAY) } withContext(dispatcher) { doTheTask(DELAY) } }

**Dependency Injection:**

Dependency injection (DI) is a technique widely used in programming and well suited to Android development. By following the principles of DI, you lay the groundwork for good app architecture.

**dependency injection** is a technique in which an object receives other objects that it depends on, called dependencies

Classes often require references to other classes. For example, a Car class might need a reference to an Engine class. These required classes are called dependencies.

Implementing dependency injection provides you with the following advantages:

* Reusability of code
* Ease of refactoring
* Ease of testing
* Reduces the boilerplate code.

There are two major ways to do dependency injection in Android:

* **Constructor Injection**. This is the way described above. You pass the dependencies of a class to its constructor.
* **Field Injection (or Setter Injection)**. Certain Android framework classes such as activities and fragments are instantiated by the system, so constructor injection is not possible. With field injection, dependencies are instantiated after the class is created. The code would look like this:

class Car {  
  
    private Engine engine;  
  
    public void setEngine(Engine engine) {  
        this.engine = engine;  
    }  
  
    public void start() {  
        engine.start();  
    }  
}  
  
class MyApp {  
    public static void main(String[] args) {  
        Car car = new Car();  
        car.setEngine(new Engine());  
        car.start();  
    }  
}

Method Injection, you can use @Inject annotation with method also.

class Car {  
  
    private final Engine engine;  
  
    public Car(Engine engine) {  
        this.engine = engine;  
    }  
  
    public void start() {  
        engine.start();  
    }  
}  
  
  
class MyApp {  
    public static void main(String[] args) {  
        Engine engine = new Engine();  
        Car car = new Car(engine);  
        car.start();  
    }  
}

**Note**: — If a class contains all types of Injection i.e. constructor, field and method injection, dagger will inject all types in the following sequence:

**use of extension functions**

Extension functions are beneficial for extending class without the need to inherit from the class.

When a function is added to an existing class it is known as Extension Function.

### Nullable Receiver:

### Extension functions can also be defined with the class type that is **nullable.**

### In this case, when the check for null is added inside the extension function and the appropriate value is returned.

// A sample class to display name name

**class** AB(val name: String){

    override fun toString(): String {

**return** "Name is $name"

    }

}

fun main(){

    // An extension function as a nullable receiver

    fun AB?.output(){

**if**(**this** == **null**){

            println("Null")

        }**else**{

            println(**this**.toString())

        }

    }

    val x = AB("Charchit")

    // Extension function called using an instance

    x.output()

    // Extension function called on null

**null**.output()

}

### Companion Object Extensions

### If a class contains a companion object, then we can also define extension functions and properties for the companion object.

**class** MyClass {

    // companion object declaration

    companion object {

        fun display(){

            println("Function declared in companion object")

        }

    }

}

fun main(args: Array<String>) {

   // invoking member function

   val ob = MyClass.display()

}

# **Kotlin Higher-Order Functions:**

a function which can accepts a function as parameter or can returns a function is called **Higher-Order function.**

Instead of Integer, String or Array as a parameter to function, we will pass anonymous function or lambdas

Passing lambda expression as a parameter to Higher-Order Function –

There are **two** types of lambda expression which can be passed-

* Lambda expression which return Unit
* Lambda expression which return any of the value integer,string etc

### Passing function as a parameter to Higher-Order function –

We can pass a function as a parameter in Higher-Order function.   
There are **two** types of functions which can be passed-

* function which return Unit
* function which return any of the value integer, string etc

### Returning a function from Higher-Order function

We can return a function from higher-order function. While returning the function, we have to specify the parameter types and return type of regular function in the return type of the higher-order function.

**what Java has that Kotlin does not:**

* Checked exceptions
* Primitive types that are not classes
* Static members
* Non-private fields
* Wildcard-types
* Ternary-operator a ? b : c

### Kotlin Syntax/Programs:

### Create variable and assign value:

### var name:String ?="Himesh"

### There are **two** ways to define an array in Kotlin.

**Using the arrayOf() function –**

### We can use the library function arrayOf() to create an array by passing the values of the elements to the function.

val num = arrayOf(1, 2, 3, 4) //implicit type declaration

val num = arrayOf<Int>(1, 2, 3) //explicit type declaration

**Using the Array constructor –**

Since Array is a class in Kotlin, we can also use the Array constructor to create an array.  
The constructor takes **two** parameters:

1. The size of the array, and
2. A function which accepts the index of a given element and returns the initial value of that element.

### val num = Array(3, {i-> i\*1})

### Example: Creating array and print the elements :

### val array = intArrayOf(10,20,30,40,50)

### for (element in array) {

### println(element)

### }

### Alternative to for loop is use println(Arrays.toString(array))

### Print a Multi-dimenstional Array:

### val array = arrayOf(intArrayOf(1, 2),

### intArrayOf(3, 4),

### intArrayOf(5, 6, 7))

### println(Arrays.deepToString(array)) : Output: [[1, 2], [3, 4], [5, 6, 7]]

### To get the numbers from the inner array, we just another function Arrays.deepToString()

var listOfMindOrks = listOf("mindorks.com", "blog.mindorks.com", "afteracademy.com")

listOfMindOrks.forEach {

Log.d(TAG,it)

### }

### Sum of Two numbers

### val num1:Int=10

### val num2: Int=20

### val sum= num1+num2

### println(“Sum is: $sum”)