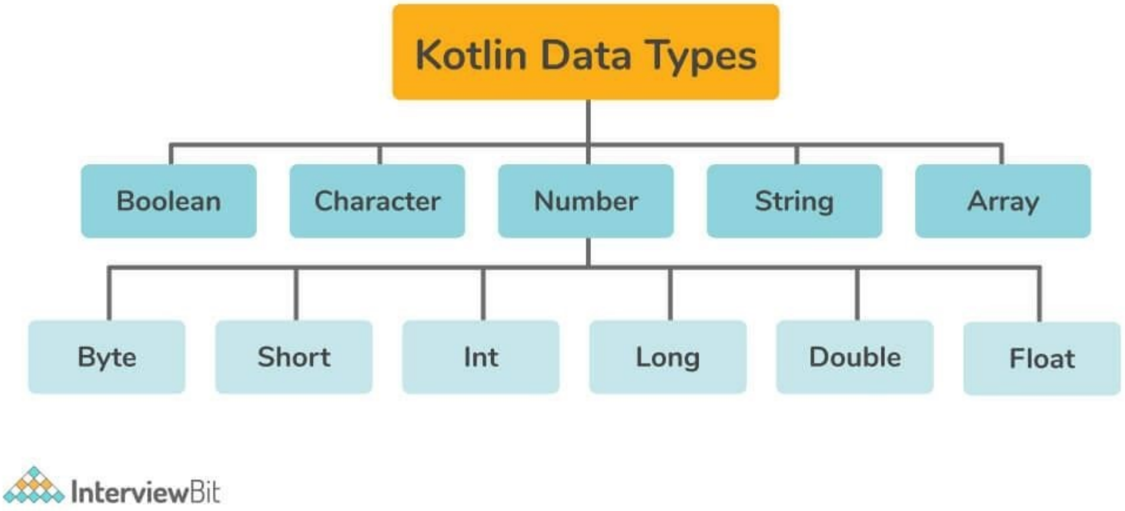
# Kotlin Interview Questions for Freshers

### 1. What are the various data types available in Kotlin? Explain them.

Primitive data types are the most basic data types in Kotlin, and all others are reference types like array and string. Kotlin contains all data types as objects.
Following are the different data types available in Kotlin:-



**Integer Data Type -**

| Data Type | Space Required |
|---|---|
| byte | 8 bits |
| short | 16 bits |
| int | 32 bits |
| long | 64 bits |

**Floating Point Data Type -**

| Data Type | Space Required |
|---|---|
| float | 32 bits |
| double | 64 bits |

**Boolean Data Type -**

True or false is the only bit of information represented by the Boolean data type. In Kotlin, the Boolean type is the same as in Java.

| Data Type | Space Required |
|---|---|

| | |
|---|---|
| boolean | 1 bit |

### Character Data Type -

Small letters (a-z), capital letters (A-Z), numerals (0-9), and other symbols are represented by the character data type.

| Data Type | Space Required |
|---|---|
| char | 8 bits |

### String Data Type -

Strings are represented in Kotlin by the type String. A string value is often a sequence of characters enclosed in double quotations ("). The space required in this case depends on the number of characters in the string.

### Array Data Type -

The **Array class** in Kotlin is used to represent arrays. It has the get and set functions that, due to operator overloading conventions, can be used as '[]' as well. The space required by the array also depends on the number of elements it posses.

## 2. How are variables declared in Kotlin? What are the different types of variables in Kotlin? Explain with examples.

Every variable in Kotlin must be declared before it can be used. An attempt to use a

variable without declaring it results in a syntax error. The type of data you are authorised to put in the memory address is determined by the variable type declaration. The type of variable can be determined from the initialised value in the case of local variables. For example,

```
var site = "interviewbit"
```

The above code declares a variable “site” of type String because the value with which the variable is initialised is a String.

There are broadly two types of variables in Kotlin. They are as follows:-

- **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.

The **syntax** is as follows :

```
val variableName = value
```

For example,

```
val sample = "interview" sample = "interviewbit"    // results
in compile time error
```

The second line in the above code snippet would result in a compile-time error as expected.

Because it can be initialized with the value of a variable, an immutable variable is not a constant. It means that the value of an immutable variable does not need to be known at compile-time and that if it is defined inside a construct that is called several times, it can take on a different value with each function call. For example,

```
var sample = "interview" val newSample = sample    //
no compile time error
```

The above code snippet runs fine and does not produce any errors.

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

The **syntax** is as follows :

```
var variableName = value
```

For example,

```
var sample = "interview" sample = "fun"    //
no compile time error
```

The above code snippet runs fine and does not produce any errors.

### 3. What are data classes in Kotlin? Explain with a proper example.

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

**Syntax:**

```
data class className ( list_of_parameters)
```

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.

**Example:**

```
data class Sample(var input1 : Int, var input2 : Int)
```

The above code snippet creates a data class Sample with two parameters.

```
fun main(agrs: Array<String>) {      val temp = Sample(1, 2)
  println(temp)
}
```

Here, we create an instance of the data class Sample and pass the parameters to it.

**Output:-**

```
Sample(input1=1, input2=2)
```

## 4. Explain the concept of null safety in Kotlin.

Kotlin's type system aims to eradicate null references from the code. If a program throws NullPointerExceptions at runtime it might result in application failure or system crashes. If the Kotlin compiler finds a null reference it throws a NullPointerException.

The Kotlin type system distinguishes between references that can hold null (nullable references) and those that cannot (non-null references). Null cannot be stored in a String variable. We get a compiler error if we try to assign null to the variable.

```
var a: String = "interview" a = null //
results in compilation error
```

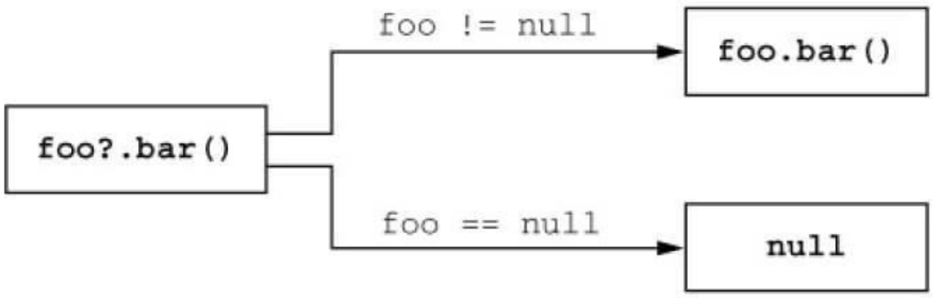
If we want the above string to be able to hold null value as well, we can declare it of type nullable using the ‘?’ operator after the String keyword as follows :

```
var a: String? = "interview" a =
null // no compilation error
```

Kotlin provides Safe Call (?.), Elvis (?:) and Not Null Assertion (!!) operators which define what needs to be done in case of a null encounter. This makes the code more reliable and less prone to errors. Thus, Kotlin enforces null safety by having nullable, non-nullable type variables and the different operators to tackle null encounters.

## 5. Explain Safe call, Elvis and Not Null Assertion operator in the context of Kotlin.

**Safe Call operator ( ?. )** - Null comparisons are trivial, but the number of nested ifelse 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.



For example,

The following expression in Kotlin

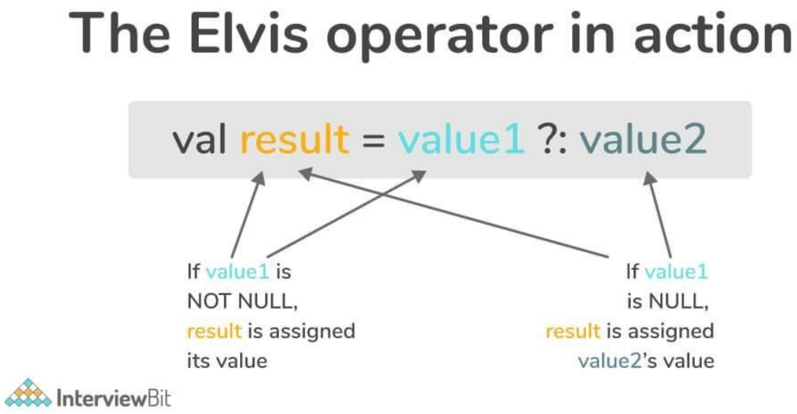
```
name?.toLowerCase()
```

is equivalent to the following

```
if(name != null)
   name.toLowerCase()
else
   null
```

**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.

right-hand side evaluated.


The Elvis operator in action
val result = value1 ?: value2
If value1 is NOT NULL, result is assigned its value
If value1 is NULL, result is assigned value2's value
InterviewBit

For example,

The following expression in Kotlin

```
val sample1 = sample2 ?: "Undefined"
```

is equivalent to the following

```
val sample1 = if(sample2 != null)
      sample2
    else
      "Undefined"
```

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

```
sample1 = sample2 ?: throw IllegalArgumentException("Invalid")
```

**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.

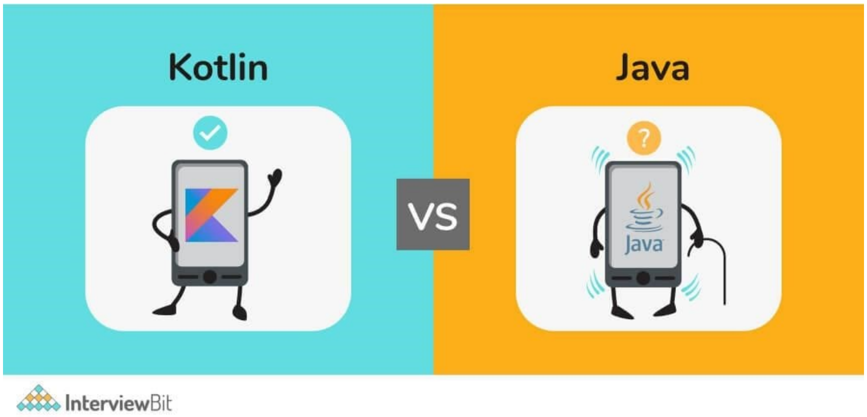
Anyone who wants a NullPointerException can ask for it explicitly with this operator.
For example,

```
// KOTLIN
fun main(args: Array<String>) {
   var sample : String?  = null
   str!!.length
}
```

The above code snippet gives the following output:-

```
Exception in thread "main" kotlin.KotlinNullPointerException
```

## 6. Differentiate between Kotlin and Java.



Following are the differences between Kotlin and Java:-

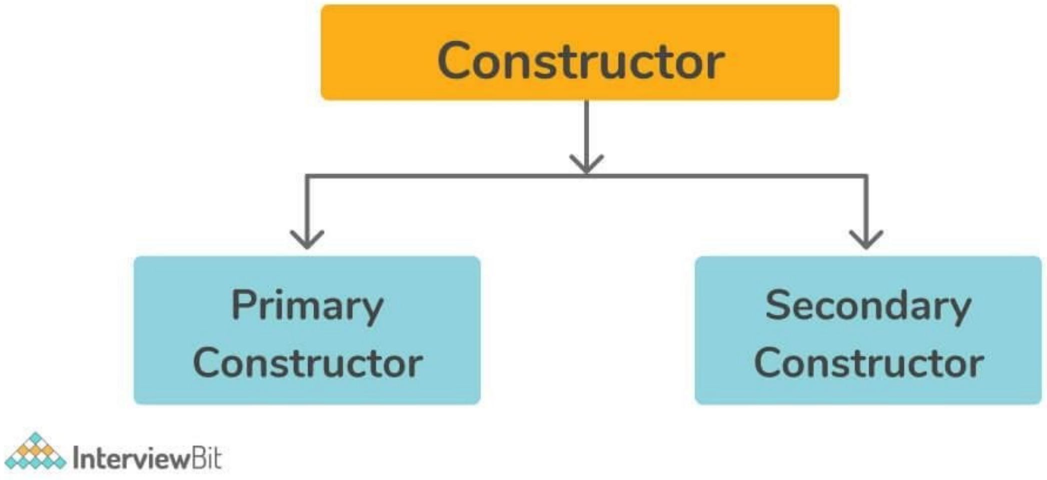
| Basis | Kotlin | Java |
|---|---|---|
| **Null Safety** | By default, all sorts of variables in Kotlin are nonnullable (that is, we can't assign null values to any variables or objects). Kotlin code will fail to build if we try to assign or return null values. If we absolutely want a null value for a variable, we can declare it as follows: value num: Int? = null | NullPointerExceptions are a big source of annoyance for Java developers. Users can assign null to any variable, however, when accessing an object reference with a null value, a null pointer exception is thrown, which the user must manage. |
| **Coroutine Support** | We can perform lor expensive tasks in s threads in Kotlin, bu have coroutines sup halt execution at a g moment without bl threads while doing running demanding | The corresponding thread in Java will be blocked anytime we launch a longrunning network I/0 or CPU-intensive task. Android is a single-threaded operating system by default. Java allows you to create and execute numerous threads in the background, but managing them is a difficult operation. |

| | | |
|---|---|---|
| **Data Clas** | If we need to have c classes in Kotlin, we a class with the key in the class declarat compiler will take ca everything, | Let's say we need a class in Java that only holds data and nothing else. Constructors, variables to store data, getter and setter methods, hashcode(), function toString(), and equals() |
| **Basis** | **Kotlin** | **Java** |
| | including constructi constructors, getter methods for various | functions are all required to be written explicitly by the developer. |
| **Functiona Programm** | Kotlin is procedural functional programm programming parad we aim to bind ever functional units) lan has numerous usefu such as lambda exp operator overloadir order functions, anc evaluation, among c | Java does not allow functional programming until Java 8, however it does support a subset of Java 8 features when developing Android apps. |
| **Extension Functions** | Kotlin gives develop ability to add new fu to an existing class. prefixing the name c the name of the nev we can build extenc functions. | In Java, we must create a new class and inherit the parent class if we want to enhance the functionality of an existing class. As a result, Java does not have any extension functions. |
| **Data Type Inference** | We don't have to de type of each variabl the assignment it w Kotlin. We can spec if we want to. | When declaring variables in Java, we must declare the type of each variable explicitly. |

| | | |
|---|---|---|
| **Smart Cas** | Smart casts in Kotlir care of these castin with the keyword "i which checks for im values and conduct casting. | We must examine the type of variables in Java and cast them appropriately for our operation. |
| **Checked Exception** | We don't have chec exceptions in Kotlin result, developers d to declare or catch which has both ben drawbacks. | We have checked exceptions support in Java, which enables developers to declare and catch exceptions, resulting in more robust code with better error handling. |

## 7. What are the different types of constructors available in Kotlin? Explain them with proper examples.

There are two types of Kotlin constructors:



- **Primary Constructor** - This type of constructor is initialised in the class header and is provided after the class name. It is declared using the "constructor" keyword. Parameters are optional in this type of constructor. For example,

```
class Sample constructor(val a: Int, val b: Int) {
  // code
}
```

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

```
// KOTLIN
fun main(args: Array<String>) {
   val s1 = Sample(1, 2)
}
class Sample(a : Int , b: Int) {
val p: Int    var q: Int
   // initializer block
init {       p = a       q
= b
       println("The first parameter value is : $p")
println("The second parameter value is : $q")
   }
}
```

**Output:-**

```
The first parameter value is: 1
The second parameter value is: 2
```

**Explanation -** The values 1 and 2 are supplied to the constructor arguments **a** and **b** when the object **s1** is created for the class Sample. In the class **p** and **q**, two attributes are specified. The initializer block is called when an object is created, and it not only sets up the attributes but also prints them to the standard output.

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

```
// KOTLIN
fun main(args: Array<String>) {    val s1 = Sample(1, 2)
}
class Sample {
   constructor(a: Int, b: Int) {
       println("The first parameter value is : $p")
println("The second parameter value is : $q")
   } }
```

**Output:-**

```
The first parameter value is: 1
The second parameter value is: 2
```

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

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

## 8. Explain the various methods to iterate over any data structure in Kotlin with examples.

Following are the different ways to iterate over any data structure in Kotlin :

- **For Loop -** The for loop is used to scan any data structure that supplies an iterator in this case. It is not used in the same way as the for loop in other programming languages such as Java or C.

In Kotlin, the for loop has the following **Syntax**:

```
for(item in collection) {
    // code
}
```

Here, collection refers to the data structure to be iterated and item refers to each element of the data structure.

For example,

```
// KOTLIN
fun main(args: Array<String>) {
  var numbersArray = arrayOf(1,2,3,4,5,6,7,8,9,10)

  for      (num      in      numbersArray){
if(num % 2 == 0){
         print("$num ")
     }
  }
}
```

**Output -**

```
2 4 6 8 10
```

- **While Loop -** It is made up of a code block and a condition to be checked for each iteration. First, the while condition is assessed, and if it is true, the code within the block is executed. Because the condition is verified every time before entering the block, it repeats until the condition turns false. The while loop can be thought of as a series of if statements that are repeated.

The while loop's syntax is as follows:

```
while(condition) {
      // code  }
```

For example,

```
// KOTLIN
fun main(args: Array<String>) {    var number =
1    while(number <= 5) {       println(number)
      number++;
  } }
```

**Output -**

```
1
2
3
4
5
```

- **Do While Loop -** The condition is assessed after all of the statements inside the block have been executed. If the do-while condition is true, the code block is re-executed. As long as the expression evaluates to true, the code block execution procedure is repeated. The loop ends if the expression becomes false, and control is passed to the sentence following the do-while loop. Because it verifies the condition after the block is executed, it's also known as a post-test loop.

The do-while loop's syntax is as follows:

```
do {     // code
{
while(condition)
```

For example,

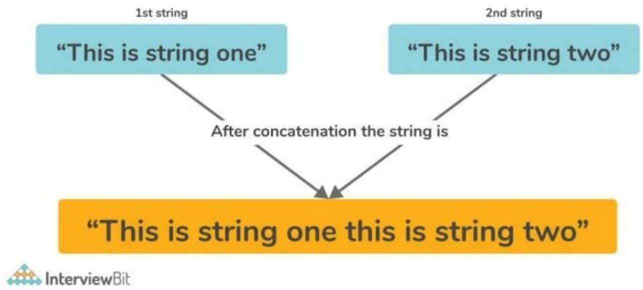
```
// KOTLIN
fun main(args: Array<String>) {    var
number = 4
  var sum = 0


  do {
      sum += number
number--    }while(number >
0)
  println("Sum of first four natural numbers is $sum")
}
```

**Output -**

```
Sum of first four natural numbers is 10
```

## 9. How can you concatenate two strings in Kotlin?

Following are the different ways by which we can concatenate two strings in Kotlin: two strings in Kotlin:


1st string
“This is string one”
2nd string
“This is string two”
After concatenation the string is
“This is string one this is string two”
InterviewBit

**Using String Interpolation:-** We use the technique of string interpolation to concatenate the two strings. Basically, we substitute the strings in place of their placeholders in the initialisation of the third string.

```
val s1 = "Interview" val
s2 = "Bit"
val s3 = "$s1 $s2" // stores "Interview Bit"
```

**Using the + or plus() operator:-** We use the ‘+’ operator to concatenate the two strings and store them in a third variable.

```
val s1 = "Interview" val
s2 = "Bit"
val s3 = s1 + s2 // stores "InterviewBit" val s4
= s1.plus(s2) // stores "InterviewBit"
```

**Using StringBuilder:-** We concatenate two strings using the StringBuilder object. First, we append the first string and then the second string.

```
val s1 = "Interview" val
s2 = "Bit"
val s3 =  StringBuilder()      s3.append(s1).append(s2)
val s4 = s3.toString() // stores "InterviewBit"
```

## 10. What do you understand about function extension in the context of Kotlin? Explain.

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

**Function Extension -** Kotlin allows users to specify a method outside of the main class via function extension. We'll see how the extension is implemented at the functional level in the following example:

```
// KOTLIN class
Sample {
 var str : String = "null"


 fun printStr() {
    print(str)
 }
}
fun main(args: Array<String>) {  var
a = Sample()
 a.str = "Interview"  var
b = Sample()  b.str =
"Bit"
 var  c = Sample()  c.str
= a.add(b)
 c.printStr()
}
// function extension
fun Sample.add(a : Sample):String{
var temp = Sample()  temp.str =
this.str + " " +a.str  return temp.str
}
```

**Output:-**

```
Interview Bit
```

**Explanation:-**

We don't have a method named "addStr" inside the "Sample" class in the preceding example, but we are implementing the same method outside of the class. This is all because of function extension.

## 11. What do you understand about Companion Object in the context of Kotlin?

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

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

```
// Syntax in KOTLIN
class CompanionClass {


   companion object CompanionObjectName {
     // code
   }
}
val obj = CompanionClass.CompanionObjectName
```

We can also remove the CompanionObject name and replace it with the term companion, resulting in the companion object's default name being Companion, as shown below:

```
// KOTLIN class CompanionClass {
companion object {
    // code
  }
}
val obj = CompanionClass.Companion
```

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

```
class Sample {
  companion object Test {
var a: Int = 1
      fun testFunction() = println("Companion Object’s Member function called.")
  }
}
fun main(args: Array<String>) {
println(Sample.a)    Sample.testFunction()
}
```

**Output:-**

```
1
Companion Object’s Member function called.
```

## 12. Differentiate between open and public keywords in Kotlin.

The keyword “open” refers to the term "open for expansion". The open annotation on a class is the polar opposite of the final annotation in Java: it allows others to inherit from it. By default, a class cannot be inherited in Kotlin. In Kotlin, an open method signifies that it can be overridden, whereas it cannot be by default. Instead, any methods in Java can be overridden by default.

In Kotlin, all the classes are final by default. If no visibility modifier is specified, the public is used by default, which means our declarations will be accessible everywhere inside the program.

## 13. Explain about the “when” keyword in the context of Kotlin.

The “when” keyword is used in Kotlin to substitute the switch operator in other languages such as Java. When a certain condition is met, a specific block of code must be run. Inside the when expression, it compares all of the branches one by one until a match is discovered. After finding the first match, it proceeds to the conclusion of the when block and executes the code immediately following the when block. We do not need a break statement at the end of each case, unlike switch cases in Java or any other programming language.

For example,

```
// KOTLIN
fun main(args: Array<String>) {


  var temp = "Interview"    when(temp)
{
     "Interview" -> println("Interview Bit is the solution.")
     "Job" -> println("Interview is the solution.")
     "Success" -> println("Hard Work is the solution.")
  }
}
```

**Output:-**

```
Interview Bit is the solution.
```

**Explanation:-** In the above code, the variable temp has the value “Interview”. The when condition matches for the exact value as that of temp’s and executes the corresponding code statements. Thus, “Interview Bit is the solution” is printed.

## 14. What are the advantages of Kotlin over Java?

Following are the advantages of Kotlin over Java:-

- **Data class:** In Java, you must create getters and setters for each object, as well as properly write hashCode (or allow the IDE to build it for you, which you must do every time you update the class), toString, and equals. Alternatively, you could utilize lombok, but that has its own set of issues. In Kotlin, data classes take care of everything.
- **Patterns of getter and setter:** In Java, for each variable, you use it for, rewrite the getter and setter methods. You don't have to write getter and setter in kotlin, and if you must, custom getter and setter take a lot less typing. There are additional delegates for identical getters and setters.
- **Extension Functions:** In Java, there is no support for extension functions. Kotlin on the other hand provides support for extension functions which makes the code more clear and cleaner.
- **Support for one common codebase:** You may extract one common codebase that will target all of them at the same time using the Kotlin Multi-Platform framework.
- **Support for Null Safety:** Kotlin has built-in null safety support, which is a lifesaver, especially on Android, which is full of old Java-style APIs.

- **Less prone to errors:** There is less space for error because it is more concise and expressive than Java.

# Kotlin Interview Questions for Experienced

### 1. Which one is better to use - val mutableList or var immutableList in the context of Kotlin?

The program's design clarity is improved by using mutable and immutable lists. This is done to have the developer think about and clarify the collection's purpose.

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.

## 2. What do you understand about lateinit in Kotlin? When would you consider using it?

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.

For example,

```
// KOTLIN
lateinit var test: String
fun testFunction() {    test
= "Interview"
   println("The length of string is "+test.length)    test =
"Bit"
}
```

When the testFunction is called, we get the following output:-

```
9
```

There are a few scenarios in which this is particularly useful, for example: •

Variables that are initialized in lifecycle methods in Android;

- Using Dagger for DI: injected class variables are initialized outside of the constructor and independently;
- Setup for unit tests: in a @Before - annotated function, test environment variables are initialized;
- Annotations in Spring Boot (for example, @Autowired).

## 3. Explain lazy initialization in the context of Kotlin.

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.

For example, let us imagine you have a SlowClass class and you require an object of that SlowClass in a different class called FastClass:

```
// KOTLIN
class FastClass {
  private val slowObject: SlowClass = SlowClass()
}
```

We are generating a large object here, which will cause the development of the FastClass to be slow or delayed. There may be times where the SlowClass object isn't required. As a result, the lazy keyword can assist you in this situation:

```
class FastClass {
  private val slowObject: SlowClass by lazy {
     SlowClass()
  }  }
```

For example,

```
// KOTLIN
class FastClass {
  private val slowObject: SlowClass by lazy {       println("Slow
Object initialised")
     SlowClass()
  }


  fun access() {
     println(slowObject)
  }
}
fun main(args: Array<String>) {    val
fastClass = FastClass()
println("FastClass initialised")
fastClass.access()    fastClass.access()
}
```

**Output:-**

```
FastClass initialised
Slow Object initialised
SlowClass@2b12fkk7
SlowClass@2b12fkk7
```

**Explanation:-** In the above code, we have instantiated an object of the SlowClass inside the class structure of the FastClass using lazy initialisation. The object of the SlowClass is generated only when it is accessed in the above code, that is, when we call the access() method of the FastClass object and the same object is present throughout the main() method.

## 4. Differentiate between lateinit and lazy initialisation. Explain the cases when you should use lateinit and when you should use lazy initialisation.

Following are the differences between lateinit and lazy initialisation:-

| lateinit | lazy initialisation |
|---|---|
| The main purpose is to initialisation to a later p time. | The main purpose is to initialise an object only when it is used at a later point in time. Also, a single copy of the object is maintained throughout the program. |
| It's possible to initialise from anywhere in the p | Only the initializer lambda can be used to initialise it. |
| Multiple initializati possible in this case. | Only a single initialisation is possible in this case. |
| It's not thread-safe. In multithreaded system, the user to correctly in | Thread-safety is enabled by default, ensuring that the initializer is only called once. |
| It works only with var. | It works only with val. |
| The isInitialized metho to verify if the value ha previously been initiali | It is impossible to uninitialize a property. |
| Properties of primitive not allowed | Allowable on primitive type properties. |

There are a few easy principles to follow when deciding whether to use lateinit or lazy initialisation for property initialization: • 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.

## 5. What do you understand about coroutines in the context of Kotlin?

Unlike many other languages with equivalent capabilities, async and await are neither keywords nor part of Kotlin's standard library. JetBrains' kotlinx.coroutines library is a comprehensive library for coroutines. It includes a number of high-level coroutine-enabled primitives, such as launch and async. Kotlin Coroutines provide an API for writing asynchronous code in a sequential manner.

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.

## 6. Explain scope functions in the context of Kotlin. What are the different types of Scope functions available in Kotlin?

The Kotlin standard library includes numerous functions that aid in the execution of a block of code within the context of an object. When you use a lambda expression to call these functions on an object, temporary scope is created. These functions are referred to as Scope functions. The object of these functions can be accessed without knowing its name. Scope functions make code more clear, legible, and succinct, which are key qualities of the Kotlin programming language.

Following are the different types of Scope functions available in Kotlin:-

- **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.

## 7. 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. Suspending functions are merely standard Kotlin functions with the suspend modifier added, indicating that they can suspend coroutine execution without blocking the current thread. This means that the code you're looking at may pause execution when it calls a suspending function and restart execution at a later time. However, it makes no mention of what will happen to the present thread in the meantime.

Suspending functions can call any other ordinary functions, but another suspending function is required to suspend the execution. Because a suspending function cannot be called from a regular function, numerous coroutine builders are supplied, allowing you to call a suspending function from a non-suspending scope like launch, async, or runBlocking.

delay() function is an example of suspend function.

## 8. What do you understand about sealed classes in Kotlin?

Kotlin introduces a crucial new form of class that isn't seen in Java. These are referred to as "sealed classes." Sealed classes, as the name implies, adhere to constrained or bounded class hierarchies. A sealed class is one that has a set of subclasses. When it is known ahead of time that a type will conform to one of the subclass types, it is employed. Type safety (that is, the compiler will validate types during compilation and throw an exception if a wrong type has been assigned to a variable) is ensured through sealed classes, which limit the types that can be matched at compile time rather than runtime. The syntax is as follows:-

```
sealed class className
```

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.

For example,

```
// KOTLIN sealed class
Sample {     class A :
Sample() {        fun
print()
      {
         println("This is the subclass A of sealed class Sample")
```

```
        }
    }
    class B : Sample() {         fun print()
        {
            println("This is the subclass B of sealed class Sample")
        }
    }
}
fun main()
{
    val obj1 = Sample.B()     obj1.print()

    val obj2 = Sample.A()     obj2.print()
}
```

**Output:-**

```
This is the subclass B of sealed class Sample
This is the subclass A of sealed class Sample
```

**Explanation:-** In the above code, we have created a sealed class named “Sample” and we have created two sub classes within it named “A” and “B”. In the main function, we create an instance of both the sub classes and call their “print” method.

## 9. What do you understand about the backing field in Kotlin?

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.

For example,

```
var marks: Int = someValue
       get() = field        set(value) {
           field = value
       }
```

**Explanation:-** Here the field identifier acts as a reference to the property “marks” value in the get() and set() method. So, whenever we call the get(), we get the field’s value returned. Similarly, whenever we call the set(), we set the “marks” property value to “value”.

## 10. Differentiate between launch / join and async / await in Kotlin.

**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.

## 11. What are some of the disadvantages of Kotlin?

Following are some of the disadvantages of Kotlin:

- In Kotlin, there are a few keywords that have non-obvious meanings: internal, crossinline, expect, reified, sealed, inner, open. Java has none of these.
- Checked exceptions are likewise absent in Kotlin. Although checked exceptions have become less prominent, many programmers believe them to be an effective technique to ensure that their code is stable.
- A lot of what happens in Kotlin is hidden. You can almost always trace the logic of a program in Java. When it comes to bug hunting, this can be really useful. If you define a data class in Kotlin, getters, setters, equality testing, tostring, and hashcode are automatically added for you.
- Learning resources are limited. The number of developers who are moving to Kotlin is growing, yet there is a small developer community accessible to help them understand the language or address problems during development.
- Kotlin has variable compilation speed. In some situations, Kotlin outperforms Java, particularly when executing incremental builds. However, we must remember that when it comes to clean builds, Java is the clear winner.