**Kotlin**

**Question:** Kotlin

**Answer:** Kotlin is an open-source, general-purpose, statically typed programming language. It is JVM-based and may be used in all projects where Java is currently used. It may be used to create Android apps, server-side apps, and a variety of other applications. It is an improved version of Java and is completely interoperable with it. We can even mix Kotlin and Java in the same project. It blends Object Oriented Programming (OOPs) and functional programming in a platform that is unlimited, self-contained, and unique. It also enables functional twinning using microcodes.

The JetBrains team created Kotlin. The language was initially launched in February 2016 after a project began in 2010 to develop it. The Apache 2.0 license was used to create Kotlin.

Programs written in Kotlin do not require semicolons. This makes the code more readable and easy to understand. This language enables the interchange and usage of Java in a variety of ways. Furthermore, code written in Java and Kotlin can coexist in the same project. The type system of Kotlin is designed to eliminate NullPointerException from code. Writing new code in Kotlin will take you less time. Kotlin code is even easier to deploy and manage at scale.

Following are the features of Kotlin:-

Compact code: Kotlin is an OOPs-based programming language with code lines that may be reduced by up to 40% when compared to Java, making it an excellent choice for software development.

Open Source: Kotlin for Android is open-source and uses the JVM to combine the benefits of OOPs and functional programming.

Simple Language: When working with Kotlin, compiling the code is simple, resulting in improved performance for Android development. It also explains which types of data functions can be used throughout the code.

High number of extensions: Without modifying the code, Kotlin may support a variety of extension functions. It aids developers in making existing code more appealing and wonderful.

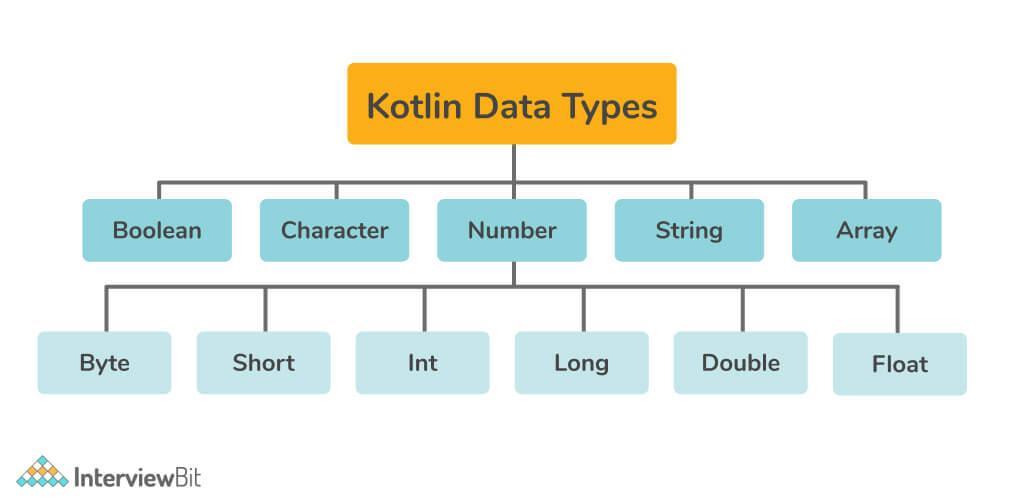
Full Java Interoperability: Java code can utilize Kotlin code, and Kotlin code can use Java code. So, if you're familiar with OOPS programming, switching to Kotlin development is simple. Also, if there are any Java-based applications, they can be used with Kotlin's environment.

Smart Cast: Smart casting is a technique that reduces the cost of an application while also improving its speed and performance. It uses typecasting or immutable data to manage the efficiency of programming. This technique is supported by Kotlin.

Low Learning Curve: Kotlin is preferred by businesses due to its low adoption cost. Most significantly, it is simple for developers to learn, especially if they have programming experience.

**Question:**  What are the various data types available in Kotlin? Explain them.

**Answer:** 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

**Question:**  How are variables declared in Kotlin? What are the different types of variables in Kotlin? Explain with examples.

**Answer:** 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)

**Question:**  Explain the concept of null safety in Kotlin.

**Answer:** 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 :

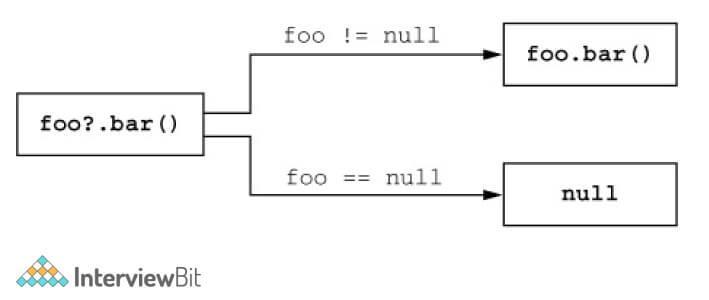
var a: String? = "interview"

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.

**Question:**  Explain Safe call, Elvis and Not Null Assertion operator in the context of Kotlin.

**Answer:** **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.



For example,

The following expression in Kotlin

name?.toLowerCase()

is equivalent to the following

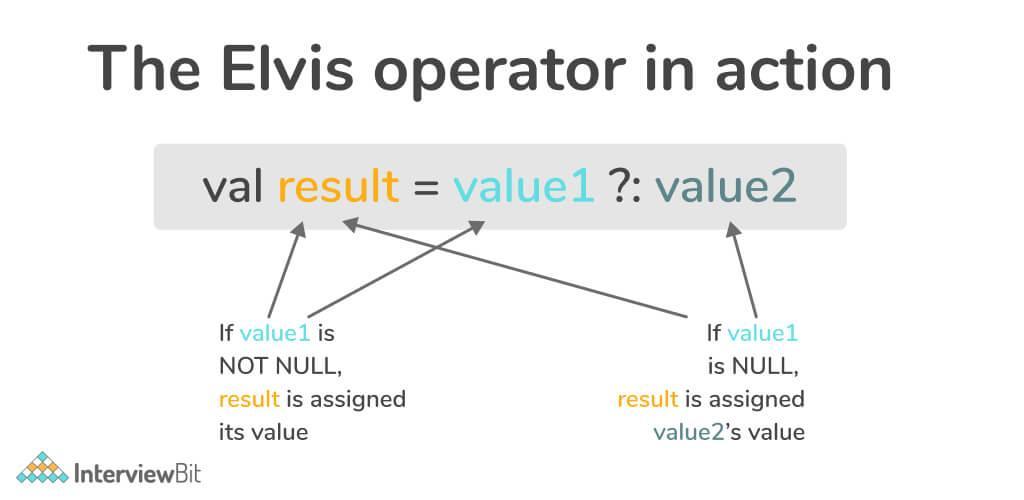
if(name != null)

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.



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

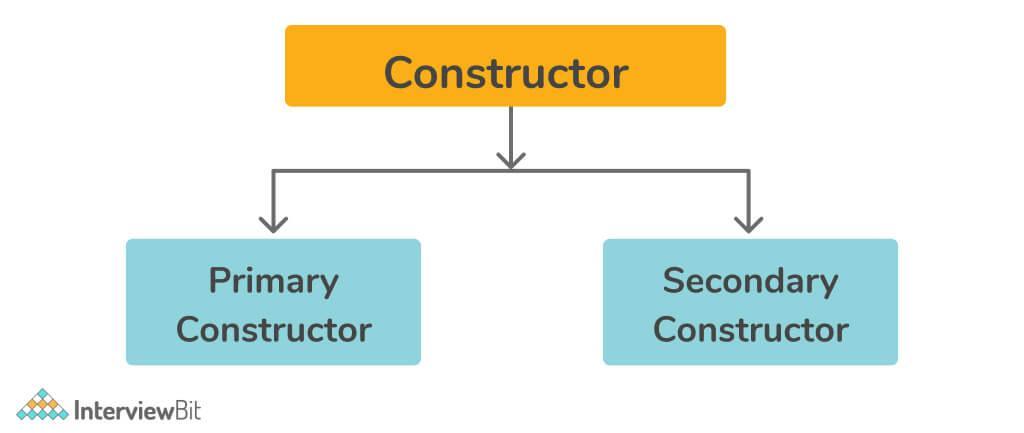
**Question:**  Differentiate between Kotlin and Java.

**Answer:**   
Following are the differences between Kotlin and Java:-

| **Basis** | **Kotlin** | **Java** |
| --- | --- | --- |
| **Null Safety** | By default, all sorts of variables in Kotlin are non-nullable (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. |
| **Coroutines Support** | We can perform long-running expensive tasks in several threads in Kotlin, but we also have coroutines support, which halt execution at a given moment without blocking threads while doing long-running demanding operations. | The corresponding thread in Java will be blocked anytime we launch a long-running 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 Classes** | If we need to have data-holding classes in Kotlin, we may define a class with the keyword "data" in the class declaration, and the compiler will take care of everything, including constructing constructors, getter, and setter methods for various fields. | 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() functions are all required to be written explicitly by the developer. |
| **Functional Programming** | Kotlin is procedural and functional programming (a programming paradigm where we aim to bind everything in functional units) language that has numerous useful features such as lambda expressions, operator overloading, higher-order functions, and lazy evaluation, among others. | 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 developers the ability to add new functionality to an existing class. By prefixing the name of a class to the name of the new function, we can build extended 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 declare the type of each variable based on the assignment it will handle in Kotlin. We can specify explicitly if we want to. | When declaring variables in Java, we must declare the type of each variable explicitly. |
| **Smart Casting** | Smart casts in Kotlin will take care of these casting checks with the keyword "is-checks," which checks for immutable values and conducts implicit casting. | We must examine the type of variables in Java and cast them appropriately for our operation. |
| **Checked Exceptions** | We don't have checked exceptions in Kotlin. As a result, developers do not need to declare or catch exceptions, which has both benefits and 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. |

**Question:**  What are the different types of constructors available in Kotlin? Explain them with proper examples.

**Answer:** 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

**Question:**  Explain the various methods to iterate over any data structure in Kotlin with examples.

**Answer:** 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")

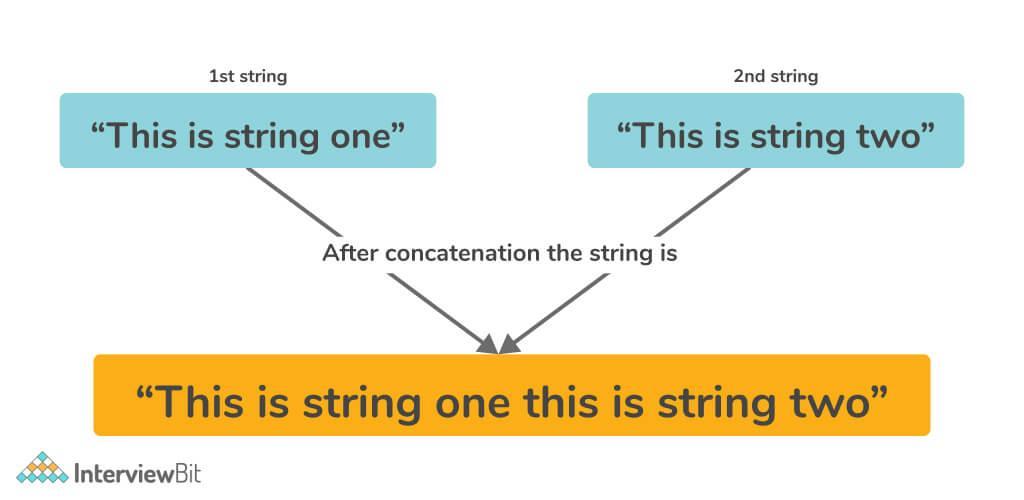
}

**Output -**

Sum of first four natural numbers is 10

**Question:**  How can you concatenate two strings in Kotlin?

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



**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"

**Question:**  What do you understand about function extension in the context of Kotlin? Explain.

**Answer:** 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

**Question:**  What do you understand about Companion Object in the context of Kotlin?

**Answer:** 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.

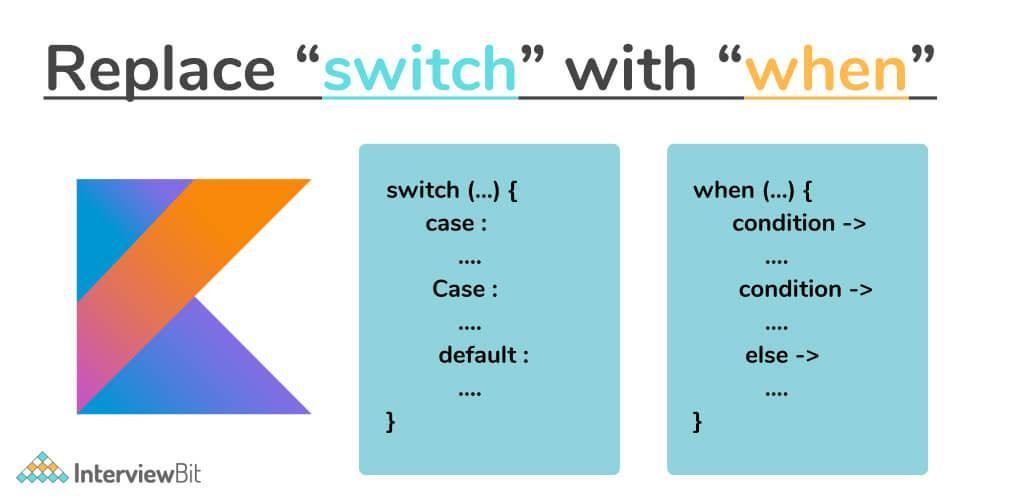
**Question:**  Differentiate between open and public keywords in Kotlin.

**Answer:** 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

**Question:**  Explain about the “when” keyword in the context of Kotlin.

**Answer:** 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

**Question:**  What are the advantages of Kotlin over Java?

**Answer:** 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

**Question:**  Which one is better to use - val mutableList or var immutableList in the context of Kotlin?

**Answer:** 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

**Question:**  What do you understand about lateinit in Kotlin? When would you consider using it?

**Answer:** 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)

**Question:**  Explain lazy initialization in the context of Kotlin.

**Answer:** 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

**Question:**  Differentiate between lateinit and lazy initialisation. Explain the cases when you should use lateinit and when you should use lazy initialisation.

**Answer:** Following are the differences between lateinit and lazy initialisation:-

| **lateinit** | **lazy initialisation** |
| --- | --- |
| The main purpose is to delay the initialisation to a later point in 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 the object from anywhere in the program. | Only the initializer lambda can be used to initialise it. |
| Multiple initializations are possible in this case. | Only a single initialisation is possible in this case. |
| It's not thread-safe. In a multi-threaded system, it is up to the user to correctly initialise. | 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 method is added to verify if the value has previously been initialised. | It is impossible to uninitialize a property. |
| Properties of primitive types are 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

**Question:**  What do you understand about coroutines in the context of Kotlin?

**Answer:** 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.

**Question:**  Explain scope functions in the context of Kotlin. What are the different types of Scope functions available in Kotlin?

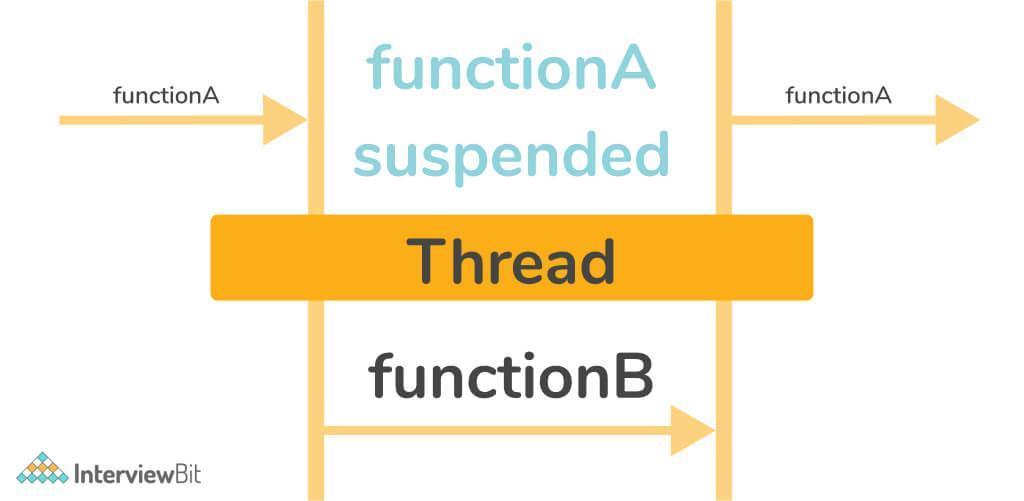
**Answer:** The Kotlin standard library includes numerous functions that aid in the execution of a block of code within the context of an object. When you use a lambda expression to call these functions on an object, temporary scope is created. These functions are referred to as Scope functions. The 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

**Question:**  Explain suspend function in the context of Kotlin.

**Answer:** 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

**Question:**  What do you understand about sealed classes in Kotlin?

**Answer:** 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

**Question:**  What do you understand about the backing field in Kotlin?

**Answer:** 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”

**Question:**  Differentiate between launch / join and async / await in Kotlin.

**Answer:** 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.

**Question:**  What are some of the disadvantages of Kotlin?

**Answer:** 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.

**Question:**  What’s the Target Platform of Kotlin? How is Kotlin-Java interoperability possible?

**Answer:** Java Virtual Machine(JVM) is the Target Platform of Kotlin. Kotlin is 100% interoperable with Java since both, on compilation produce bytecode. Hence Kotlin code can be called from Java and vice-versa.

**Question:**  How do you declare variables in Kotlin? How does the declaration differ from the Java counterpart?

**Answer:** There are two major differences between Java and Kotlin variable declaration:

The type of declaration In Java the declaration look like this:

String s = "Java String";

int x = 10;

In Kotlin the declaration looks like:

val s: String = "Hi"

var x = 5

In Kotlin, the declaration begins with a val and a var followed by the optional type. Kotlin can automatically detect the type using type inference.

Default value The following is possible in Java:

String s:

The following variable declaration in Kotlin is not valid.

val s: String

**Question:**  What’s the difference between val and var declaration? How to convert a String to an Int?

**Answer:** 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.

fun main(args: Array<String>) {

val s: String = "Hi"

var x = 5

x = "6".toInt()

}

We use the toInt() method to convert the String to an Int.

**Question:**  What’s Null Safety and Nullable Types in Kotlin? What is the Elvis Operator?

**Answer:** Kotlin puts a lot of weight behind null safety which is an approach to prevent the dreaded Null Pointer Exceptions by using nullable types which are like String?, Int?, Float? etc. These act as a wrapper type and can hold null values. A nullable value cannot be added to another nullable or basic type of value. To retrieve the basic types we need to use safe calls that unwrap the Nullable Types. If on unwrapping, the value is null we can choose to ignore or use a default value instead. The Elvis Operator is used to safely unwrap the value from the Nullable. It’s represented as ?: over the nullable type. The value on the right hand side would be used if the nullable type holds a null.

var str: String? = "JournalDev.com"

var newStr = str?: "Default Value"

str = null

newStr = str?: "Default Value"

**Question:**  What’s a const? How does it differ from a val?

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

**Question:**  Does Kotlin allow us to use primitive types such as int, float, double?

**Answer:** At the language level, we cannot use the above-mentioned types. But the JVM bytecode that’s compiled does certainly have them.

**Question:**  What’s the entry point of every Kotlin Program?

**Answer:** The main function is the entry point of every Kotlin program. In Kotlin we can choose not to write the main function inside the class. On compiling the JVM implicitly encapsulates it in a class. The strings passed in the form of Array<String> are used to retrieve the command line arguments.

**Question:**  How is !!different from ?. in unwrapping the nullable values? Is there any other way to unwrap nullable values safely?

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

**Question:**  How is a function declared? Why are Kotlin functions known as top-level functions?

**Answer:** fun sumOf(a: Int, b: Int): Int{

return a + b

}

A function’s return type is defined after the : Functions in Kotlin can be declared at the root of the Kotlin file.

**Question:**  What’s the difference between == and === operators in Kotlin?

**Answer:** \== is used to compare the values are equal or not. === is used to check if the references are equal or not.

**Question:**  List down the visibility modifiers available in Kotlin. What’s the default visibility modifier?

**Answer:** - public

- internal

- protected

- private

`public` is the default visibility modifier.

**Question:**  Does the following inheritance structure compile?

**Answer:** ```

class A{

}

class B : A(){

}

```

\*\*NO\*\*. By default classes are final in Kotlin. To make them non-final, you need to add the `open` modifier.

```

open class A{

}

class B : A(){

}

```

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

**Answer:** 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

}

}

```

**Question:**  What’s init block in Kotlin

**Answer:** `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.

**Question:**  How does string interpolation work in Kotlin? Explain with a code snippet?

**Answer:** String interpolation is used to evaluate string templates. We use the symbol $ to add variables inside a string.

```

val name = "Journaldev.com"

val desc = "$name now has Kotlin Interview Questions too. ${name.length}"

```

Using `{}` we can compute an expression too.

**Question:**  What’s the type of arguments inside a constructor?

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

**Question:**  Is new a keyword in Kotlin? How would you instantiate a class object in Kotlin?

**Answer:** \*\*NO\*\*. Unlike Java, in Kotlin, new isn't a keyword. We can instantiate a class in the following way:

```

class A

var a = A()

val new = A()

```

**Question:**  What is the equivalent of switch expression in Kotlin? How does it differ from switch?

**Answer:** when is the equivalent of `switch` in `Kotlin`. The default statement in a when is represented using the else statement.

```

var num = 10

when (num) {

0..4 -> print("value is 0")

5 -> print("value is 5")

else -> {

print("value is in neither of the above.")

}

}

```

`when` statments have a default break statement in them.

**Question:**  What are data classes in Kotlin? What makes them so useful? How are they defined?

**Answer:** 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.

```

data class Book(var name: String, var authorName: String)

fun main(args: Array<String>) {

val book = Book("Kotlin Tutorials","Anupam")

}

```

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

**Question:**  What are destructuring declarations in Kotlin? Explain it with an example.

**Answer:** Destructuring Declarations is a smart way to assign multiple values to variables from data stored in objects/arrays. [![kotlin interview questions destructuring declarations](https://journaldev.nyc3.digitaloceanspaces.com/2018/04/kotlin-interview-questions-destructuring-declarations.png)](https://journaldev.nyc3.digitaloceanspaces.com/2018/04/kotlin-interview-questions-destructuring-declarations.png) Within paratheses, we've set the variable declarations. Under the hood, destructuring declarations create component functions for each of the class variables.

**Question:**  What’s the difference between inline and infix functions? Give an example of each.

**Answer:** [Inline functions](/community/tutorials/kotlin-inline-function-reified) 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. [![kotlin interview questions inline functions](https://journaldev.nyc3.digitaloceanspaces.com/2018/04/kotlin-interview-questions-inline-functions.png)](https://journaldev.nyc3.digitaloceanspaces.com/2018/04/kotlin-interview-questions-inline-functions.png) [infix functions](/community/tutorials/kotlin-functions) on the other are used to call functions without parentheses or brackets. Doing so, the code looks much more like a natural language. [![kotlin interview questions infix notations](https://journaldev.nyc3.digitaloceanspaces.com/2018/04/kotlin-interview-questions-infix-notations.png)](https://journaldev.nyc3.digitaloceanspaces.com/2018/04/kotlin-interview-questions-infix-notations.png)

**Question:**  What’s the difference between lazy and lateinit?

**Answer:** 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.

```

val x: Int by lazy { 10 }

lateinit var y: String

```

**Question:**  How to create Singleton classes?

**Answer:** 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.

**Question:**  Does Kotlin have the static keyword? How to create static methods in Kotlin?

**Answer:** \*\*NO\*\*. Kotlin doesn't have the static keyword. To create static method in our class we use the `companion object`. Following is the Java code:

```

class A {

public static int returnMe() { return 5; }

}

```

The equivalent Kotlin code would look like this:

```

class A {

companion object {

fun a() : Int = 5

}

}

```

To invoke this we simply do: `A.a()`.

**Question:**  What’s the type of the following Array?

**Answer:** ```

val arr = arrayOf(1, 2, 3);

```

The type is Array<Int>.

**Question:**  How does Kotlin work on Android?

**Answer:** Just like Java, the Kotlin code is also compiled into the Java bytecode and is executed at runtime by the Java Virtual Machine i.e. JVM. When a Kotlin file named Main.kt is compiled then it will eventually turn into a class and then the bytecode of the class will be generated. The name of the bytecode file will be MainKt.class and this file will be executed by the JVM.

**Question:**  Why should we use Kotlin?

**Answer:** Kotlin is concise

Kotlin is null-safe

Kotlin is interoperable

**Question:**  What is the difference between the variable declaration with var and val?

**Answer:** If you want to declare some mutable(changeable) variable, then you can use var . For the immutable variable, use val i.e. val variables can't be changed once assigned.

**Question:**  What is the difference between the variable declaration with val and const?

**Answer:** Both the variables that are declared with val and const are immutable in nature. But the value of the const variable must be known at the compile-time whereas the value of the val variable can be assigned at runtime also.

**Question:**  How to ensure null safety in Kotlin?

**Answer:** 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, the following code in Kotlin will produce a compile-time error:

var name: String = "MindOrks"

name = null //error

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

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

**Answer:** 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.

var name: String? = "MindOrks"

println(name?.length) // 8

name = null

println(name?.length) // null

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

**Question:**  Do we have a ternary operator in Kotlin just like java?

**Answer:** No, we don't have a ternary operator in Kotlin but you can use the functionality of ternary operator by using if-else or Elvis operator.

**Question:**  What is Elvis operator in Kotlin?

**Answer:** In Kotlin, you can assign null values to a variable by using the null safety property. To check if a value is having null value then you can use if-else or can use the Elvis operator i.e. ?: For example:

var name:String? = "Mindorks"

val nameLength = name?.length ?: -1

println(nameLength)

The Elvis operator( ?: ) used above will return the length of name if the value is not null otherwise if the value is null, then it will return -1

**Question:**  How to convert a Kotlin source file to a Java source file?

**Answer:** Steps to convert your Kotlin source file to Java source file:

Open your Kotlin project in the IntelliJ IDEA / Android Studio.

Then navigate to Tools > Kotlin > Show Kotlin Bytecode.

Now click on the Decompile button to get your Java code from the bytecode.

**Question:**  What is the use of @JvmStatic, @JvmOverloads, and @JvmFiled in Kotlin?

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

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

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

**Question:**  What is a data class in Kotlin?

**Answer:** Data classes are those classes which are made just to store some data. In Kotlin, it is marked as data. The following is an example of the same:

data class Developer(val name: String, val age: Int)

When we mark a class as a data class, you don’t have to implement or create the following functions like we do in Java: hashCode() , equals() , toString() , copy() . The compiler automatically creates these internally, so it also leads to clean code. Although, there are few other requirements that data classes need to fulfill.

**Question:**  Can we use primitive types such as int, double, float in Kotlin?

**Answer:** In Kotlin, we can't use primitive types directly. We can use classes like Int, Double, etc. as an object wrapper for primitives. But the compiled bytecode has these primitive types.

**Question:**  What is String Interpolation in Kotlin?

**Answer:** If you want to use some variable or perform some operation inside a string then String Interpolation can be used. You can use the $ sign to use some variable in the string or can perform some operation in between {} sign.

var name = "MindOrks"

print("Hello! I am learning from $name")

**Question:**  What do you mean by destructuring in Kotlin?

**Answer:** Destructuring is a convenient way of extracting multiple values from data stored in(possibly nested) objects and Arrays. It can be used in locations that receive data (such as the left-hand side of an assignment). Sometimes it is convenient to d estructure an object into a number of variables, for example:

val (name, age) = developer

Now, we can use name and age independently like below:

println(name)

println(age)

**Question:**  When to use the lateinit keyword in Kotlin?

**Answer:** lateinit is late initialization.

Normally, properties declared as having a non-null type must be initialized in the constructor. However, fairly often this is not convenient.

For example, properties can be initialized through dependency injection, or in the setup method of a unit test. In this case, you cannot supply a non-null initializer in the constructor, but you still want to avoid null checks when referencing the property inside the body of a class. To handle this case, you can mark the property with the lateinit modifier.

**Question:**  How to check if a lateinit variable has been initialized or not?

**Answer:** You can check if the lateinit variable has been initialized or not before using it with the help of isInitialized method. This method will return true if the lateinit property has been initialized otherwise it will return false. For example:

class Person {

lateinit var name: String

fun initializeName() {

println(this::name.isInitialized)

name = "MindOrks" // initializing name

println(this::name.isInitialized)

}

}

fun main(args: Array<String>) {

Person().initializeName()

}

**Question:**  What is the difference between lateinit and lazy in Kotlin?

**Answer:** lazy can only be used for val properties, whereas lateinit can only be applied to var because it can’t be compiled to a final field, thus no immutability can be guaranteed.

If you want your property to be initialized from outside in a way probably unknown beforehand, use lateinit.

**Question:**  Is there any difference between == operator and === operator?

**Answer:** Yes. The == operator is 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. But in the case of primitive types, the === operator also checks for the value and not reference.

// primitive example

val int1 = 10

val int2 = 10

println(int1 == int2) // true

println(int1 === int2) // true

// wrapper example

val num1 = Integer(10)

val num2 = Integer(10)

println(num1 == num2) // true

println(num1 === num2) //false

**Question:**  What is the forEach in Kotlin?

**Answer:** In Kotlin, to use the functionality of a for-each loop just like in Java, we use a forEach function. The following is an example of the same:

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

listOfMindOrks.forEach {

Log.d(TAG,it)

}

**Question:**  What are companion objects in Kotlin?

**Answer:** In Kotlin, if you want to write a function or any member of the class that can be called without having the instance of the class then you can write the same as a member of a companion object inside the class.

To create a companion object, you need to add the companion keyword in front of the object declaration.

The following is an example of a companion object in Kotlin:

class ToBeCalled {

companion object Test {

fun callMe() = println("You are calling me :)")

}

}

fun main(args: Array<String>) {

ToBeCalled.callMe()

}

**Question:**  What is the equivalent of Java static methods in Kotlin?

**Answer:** To achieve the functionality similar to Java static methods in Kotlin, we can use:

companion object

package-level function

object

**Question:**  What is the difference between FlatMap and Map in Kotlin?

**Answer:** FlatMap is used to combine all the items of lists into one list.

Map is used to transform a list based on certain conditions.

**Question:**  What is the difference between List and Array types in Kotlin?

**Answer:** If you have a list of data that is having a fixed size, then you can use an Array. But if the size of the list can vary, then we have to use a mutable list.

**Question:**  Can we use the new keyword to instantiate a class object in Kotlin?

**Answer:** No, in Kotlin we don't have to use the new keyword to instantiate a class object. To instantiate a class object, simply we use:

var varName = ClassName()

**Question:**  What are visibility modifiers in Kotlin?

**Answer:** A visibility modifier or access specifier or access modifier is a concept that is used to define the scope of something in a programming language. In Kotlin, we have four visibility modifiers:

private: visible inside that particular class or file containing the declaration.

protected: visible inside that particular class or file and also in the subclass of that particular class where it is declared.

internal: visible everywhere in that particular module.

public: visible to everyone.

Note: By default, the visibility modifier in Kotlin is public

**Question:**  How to create a Singleton class in Kotlin?

**Answer:** A singleton class is a class that is defined in such a way that only one instance of the class can be created and is used where we need only one instance of the class like in logging, database connections, etc.

To create a Singleton class in Kotlin, you need to use the object keyword.

object AnySingletonClassName

**Question:**  What are init blocks in Kotlin?

**Answer:** init blocks are initializer blocks that are executed just after the execution of the primary constructor. A class file can have one or more init blocks that will be executed in series. If you want to perform some operation in the primary constructor, then it is not possible in Kotlin, for that, you need to use the init block.

**Question:**  What are the types of constructors in Kotlin?

**Answer:** Primary constructor: These constructors are defined in the class header and you can't perform some operation in it, unlike Java's constructor.

Secondary constructor: These constructors are declared inside the class body by using the constructor keyword. You must call the primary constructor from the secondary constructor explicitly. Also, the property of the class can’t be declared inside the secondary constructor. There can be more than one secondary constructors in Kotlin.

**Question:**  Is there any relationship between primary and secondary constructors?

**Answer:** Yes, when using a secondary constructor, you need to call the primary constructor explicitly.

**Question:**  What is the default type of argument used in a constructor?

**Answer:** By default, the type of arguments of a constructor in val. But you can change it to var explicitly

**Question:**  What are Coroutines in Kotlin?

**Answer:** A framework to manage concurrency in a more performant and simple way with its lightweight thread which is written on top of the actual threading framework to get the most out of it by taking the advantage of cooperative nature of functions.

**Question:**  What is suspend function in Kotlin Coroutines?

**Answer:** Suspend function is the building block of the Coroutines in Kotlin. Suspend function is a function that could be started, paused, and resume. To use a suspend function, we need to use the suspend keyword in our normal function definition.

**Question:**  What is the difference between Launch and Async in Kotlin Coroutines?

**Answer:** The difference is that the launch{} does not return anything and the async{} returns an instance of Deferred<T> , which has an await() function that returns the result of the coroutine like we have future in Java in which we do future.get() to the get the result.

In other words:

launch: fire and forget

async: perform a task and return a result

**Question:**  What are scopes in Kotlin Coroutines?

**Answer:** https://amitshekhar.me/blog/kotlin-coroutines

**Question:**  How Exception Handling is done in Kotlin Coroutines?

**Answer:** https://blog.mindorks.com/exception-handling-in-kotlin-coroutines/

**Question:**  How to choose between a switch and when in Kotlin?

**Answer:** Whenever we want to handle many if-else conditions, then we generally use switch-case statements. But Kotlin provides a more concise option i.e. in Kotlin, we can use when in place of the switch. And, when can be used as:

expression

arbitrary condition expression

without argument

with two or more choices

For example:

when(number) {

1 -> println("One")

2, 3 -> println("Two or Three")

4 -> println("Four")

else -> println("Number is not between 1 and 4")

}

**Question:**  What is the open keyword in Kotlin used for?

**Answer:** By default, the classes and functions are final in Kotlin. So, you can't inherit the class or override the functions. To do so, you need to use the open keyword before the class and function. For example: https://blog.mindorks.com/understanding-open-keyword-in-kotlin/

**Question:**  What are lambdas expressions?

**Answer:** Lambdas expressions are anonymous functions that can be treated as values i.e. we can pass the lambdas expressions as arguments to a function return them, or do any other thing we could do with a normal object. For example:

val add : (Int, Int) -> Int = { a, b -> a + b }

val result = add(9, 10)

https://amitshekhar.me/blog/higher-order-functions-and-lambdas-in-kotlin

**Question:**  What are Higher-Order functions in Kotlin?

**Answer:** A higher-order function is a function that takes functions as parameters or returns a function. For example, A function can take functions as parameters.

fun passMeFunction(abc: () -> Unit) {

// I can take function

// do something here

// execute the function

abc()

}

For example, A function can return another function.

fun add(a: Int, b: Int): Int {

return a + b

}

And, we have a function returnMeAddFunction which takes zero parameters and returns a function of the type ((Int, Int) -> Int) .

fun returnMeAddFunction(): ((Int, Int) -> Int) {

// can do something and return function as well

// returning function

return ::add

}

And to call the above function, we can do:

val add = returnMeAddFunction()

val result = add(2, 2)

**Question:**  What are extension functions in Kotlin?

**Answer:** Extension functions are like extensive properties attached to any class in Kotlin. By using extension functions, you can add some methods or functionalities to an existing class even without inheriting the class. For example: Let's say, we have views where we need to play with the visibility of the views. So, we can create an extension function for views like,

fun View.show() {

this.visibility = View.VISIBLE

}

fun View.hide() {

this.visibility = View.GONE

}

and to use it we use, like,

toolbar.hide()

**Question:**  What is an infix function in Kotlin?

**Answer:** An infix function is used to call the function without using any bracket or parenthesis. You need to use the infix keyword to use the infix function.

class Operations {

var x = 10;

infix fun minus(num: Int) {

this.x = this.x - num

}

}

fun main() {

val opr = Operations()

opr minus 8

print(opr.x)}

**Question:**  What is an inline function in Kotlin?

**Answer:** 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.

**Question:**  What is noinline in Kotlin?

**Answer:** While using an inline function and want to pass some lambda function and not all lambda function as inline, then you can explicitly tell the compiler which lambda it shouldn't inline.

inline fun doSomethingElse(abc: () -> Unit, noinline xyz: () -> Unit) {

abc()

xyz()

}

**Question:**  What are Reified types in Kotlin?

**Answer:** When you are using the concept of Generics to pass some class as a parameter to some function and you need to access the type of that class, then you need to use the reified keyword in Kotlin.

For Example

inline fun <reified T> genericsExample(value: T) {

println(value)

println("Type of T: ${T::class.java}")

}

fun main() {

genericsExample<String>("Learning Generics!")

genericsExample<Int>(100)

}

**Question:**  What is the operator overloading in Kotlin?

**Answer:** In Kotlin, we can use the same operator to perform various tasks and this is known as operator overloading. To do so, we need to provide a member function or an extension function with a fixed name and operator keyword before the function name because normally also, when we are using some operator then under the hood some function gets called. For example, if you are writing num1+num2 , then it gets converted to num1.plus(num2)

For Example:-

fun main() {

val bluePen = Pen(inkColor = "Blue")

bluePen.showInkColor()

val blackPen = Pen(inkColor = "Black")

blackPen.showInkColor()

val blueBlackPen = bluePen + blackPen

blueBlackPen.showInkColor()

}

operator fun Pen.plus(otherPen: Pen):Pen{

val ink = "$inkColor, ${otherPen.inkColor}"

return Pen(inkColor = ink)

}

data class Pen(val inkColor:String){

fun showInkColor(){ println(inkColor)}

}

**Question:**  Explain the use-case of let, run, with, also, apply in Kotlin.

**Answer:**

**Scoped Functions in Kotlin**

*By definition, Scoped functions are functions that execute a block of code within the context of an object.*

Well, what does this mean? These functions provide a way to give temporary scope to the object under consideration where specific operations can be applied to the object within the block of code, thereby, resulting in a clean and concise code. Not clear still right? Yeah. In Software Development, things are only better understood by implementing rather than reading. So, let’s go ahead and understand these scoped functions with some examples.

*The context of the object can be referred to as “it” or “this” which we will be understanding through examples in this article.*

#### Differentiating scoped functions with examples

There are **five** scoped functions in Kotlin: ***let*** *,* ***run*** *,* ***with*** *,* ***also*** and ***apply.*** Let’s go through them one by one.

But before going through these examples, let’s consider a Model class “Person”

**class** **Person**() {

var name: String = "Abcd"

var contactNumber: String = "1234567890"

var address: String = "xyz"

fun **displayInfo**() = print("\n Name: $name\n " +

"Contact Number: $contactNumber\n " +

"Address: $address")

}

#### let

Let’s consider the following function:

**private** fun **performLetOperation**() {

val person = Person().let {

**return@let** "The name of the Person is: ${it.name}"

}

print(person)

}

output:

The name of the Person is: Abcd

From the above code snippet, we can see that although “ **let** ” operation is performed on a **Person** object, the output of the code is a string value and not the **Person** object. This implies that the “ **let”** operator provides an option to perform an operation on the current object and return any value based on the use case.

**Note** : It is not necessary to write “return@let”. This is only done to enhance code readability. In Kotlin, if the last statement in a “ **let** ” block is a non-assignment statement, it is by default a return statement. For example:

**private** fun **performLetOperation**() {

val person = Person().let {

"The name of the Person is: ${it.name}"

}

print(person)

}

output:

The name of the Person is: Abcd

The above code also behaves the same way since the last statement is a non-assignment statement. So what happens if we don’t return anything in the “ **let** ” block? It is similar to calling a function that has no return value. So let’s say if we modify the function as:

**private** fun **performLetOperation**() {

val person = Person().let {

it.name = "NewName"

}

print(person)

}

output:

kotlin.Unit

We can see that since there was no return value in the let block and hence printing the same would indicate that the “ **print** ” functionality is called upon a unit function.

There are other advantages of using let:

* It refers to the context of the object by using the “it” keyword and hence, this “it” can be renamed to a readable lambda parameter.

**private** fun **performLetOperation**() {

val person = Person().let { personDetails ->

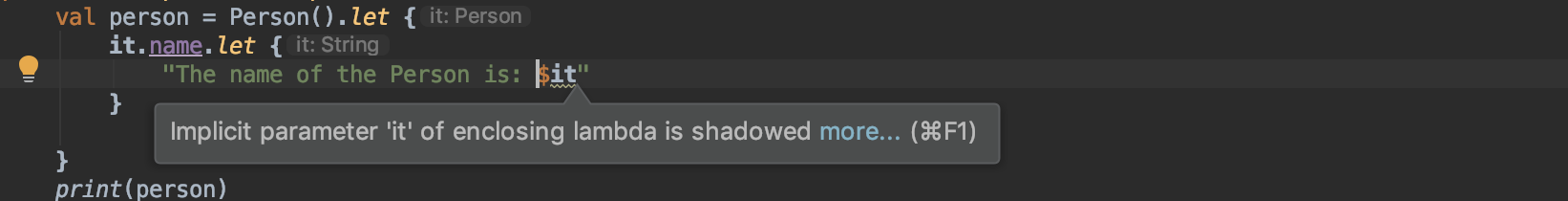
personDetails.name = "NewName"

}

print(person)

}

This can be very helpful if we have nested let blocks, and since all let blocks refer to the context of their respective objects as “it”, it might be difficult for the compiler which objects we are referring to:



* The second advantage is it easily helps in providing null checks. Let’s say we make the “ **name** ” parameter of the “ **Person** ” class nullable and we want to print the name of the person only if it is a not null value, then we can write a clean, simple and concise code as follows:

var name: String? = "Abcd"

**private** fun **performLetOperation**() {

val name = Person().name?.let {

"The name of the Person is: $it"

}

print(name)

}

We Know how lengthy the code would be to perform a null check in Java. Well, that’s the beauty of kotlin!

“ **let** ” can also be used when we want to perform an operation on the result of a call chain. Let’s take the following example:

fun **main**() {

val numbers = mutableListOf("One","Two","Three","Four","Five")

val resultsList = numbers.map { it.length }.filter { it > 3 }

print(resultsList)

}

So our aim is to fetch the values in the ArrayList whose length is greater than 3. Since we had to print the result we stored the result in a separate variable and then printed it. But using “ **let** ” operator, we can modify the code as:

fun **main**() {

val numbers = mutableListOf("One","Two","Three","Four","Five")

numbers.map { it.length }.filter { it > 3 }.let {

print(it)

}

}

This way we can perform an operation on the result of the call chain.

#### run

The “ **run** ” operator is similar to the “ **let** ” operator in terms of accepting a return value that is different from the object on which the scope function is being applied to. Hence, a “ **run** ” operator can be used to initialize an object and return the result of it.

**private** fun **performRunOperation**() {

Person().run {

name = "Asdf"

contactNumber = "0987654321"

**return@run** "The details of the Person is: ${displayInfo()}"

}

}

output:

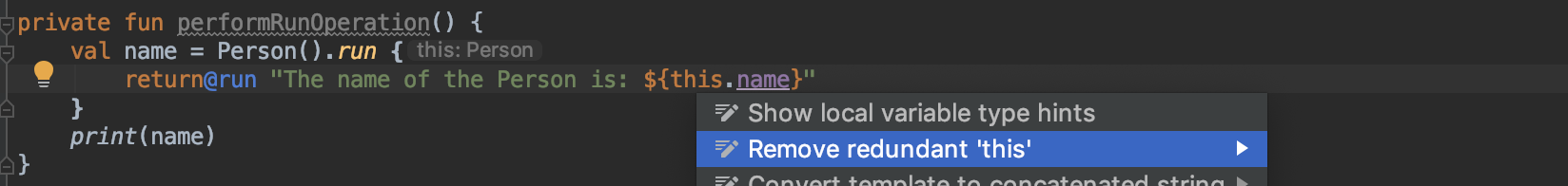
Name: Asdf

Contact Number: 0987654321

Address: xyz

#### run vs let

So if **run** is similar to **let** in terms of accepting any return value, what’s the difference? The difference is run refers to the context of the object as “ **this** ” and not “ **it** ”. That is the reason we did not use “${this.name}” as it would be redundant here since the block of code understands that “ **name** ” is used here concerning the **Person** object.



One point here is that since the context is referred to as “ **this** ”, it cannot be renamed to a readable lambda parameter. So depending on the use case and requirement we have to choose between the let and the run operator. The “ **run** ” operator also helps in easy null checks similar to the “ **let** ” operator

var name: String? = "Abcd"

**private** fun **performRunOperation**() {

val name = Person().name?.run {

"The name of the Person is: $this"

}

print(name)

}

Output:

The name of the Person is: Abcd

#### with

The “ **with** ” operator is completely similar to the run operator that we just discussed. It also refers to the context of the object as “ **this** ”, similar to how the “ **run** ” operator uses it.

**private** fun **performWithOperation**() {

val person = with(Person()) {

**return@with** "The name of the Person is: ${this.name}"

}

print(person)

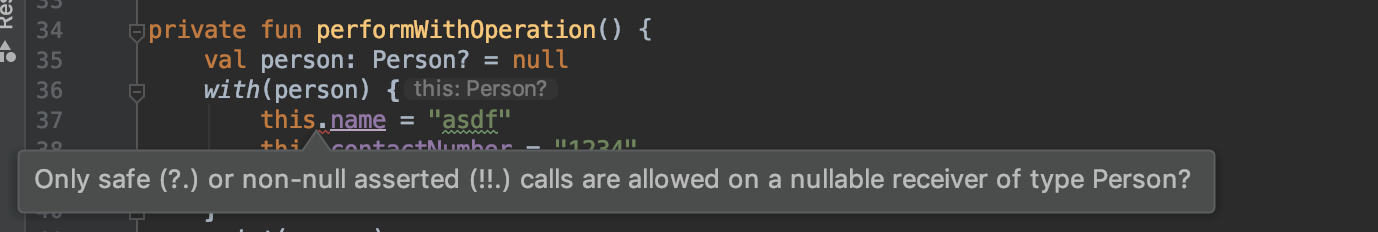
}

Output:

The name of the Person is: Abcd

#### with vs run

So, if “ **with”** is the same as “ **run** ”, what’s the difference? How should we choose between these two? Now, this is an interesting case. Let’s consider a case where a Person object can be nullable.



we can see that the context of the object referred to as “ **this** ” is a nullable type of Person. And hence, to correct this, we need to change the code as:

**private** fun **performWithOperation**() {

val person: Person? = **null**

**with**(person) {

**this**?.name = "asdf"

**this**?.contactNumber = "1234"

**this**?.address = "wasd"

**this**?.displayInfo()

}

}

So performing a null check using a “ **with** ” operator is difficult and this is where we can replace it with “ **run** ” as follows:

**private** fun **performRunOperation**() {

val person: Person? = **null**

person?.run {

name = "asdf"

contactNumber = "1234"

address = "wasd"

displayInfo()

}

}

This looks a lot cleaner.

#### apply

The *apply* function is similar to the *run* functionality only in terms of referring to the context of the object as “ **this** ” and not “ **it** ” and also in providing null safety checks:

**private** fun **performApplyOperation**() {

val person: Person? = **null**

person?.apply {

name = "asdf"

contactNumber = "1234"

address = "wasd"

displayInfo()

}

}

#### "apply" use-case in Android Development

Specifically for Android Development, " **apply** " can be useful in many cases. We have many scenarios where we should return an instance of Intent or an Alert Dialog etc., by adding specific attributes to them. Let's take the example of an intent here:

*// Normal approach*

fun **createIntent**(intentData: String, intentAction: String): Intent {

val intent = Intent()

intent.action = intentAction

intent.data = Uri.parse(intentData)

**return** intent

}

*// Improved approach, by using apply*

fun **createIntent**(intentData: String, intentAction: String) =

Intent().apply {

action = intentAction

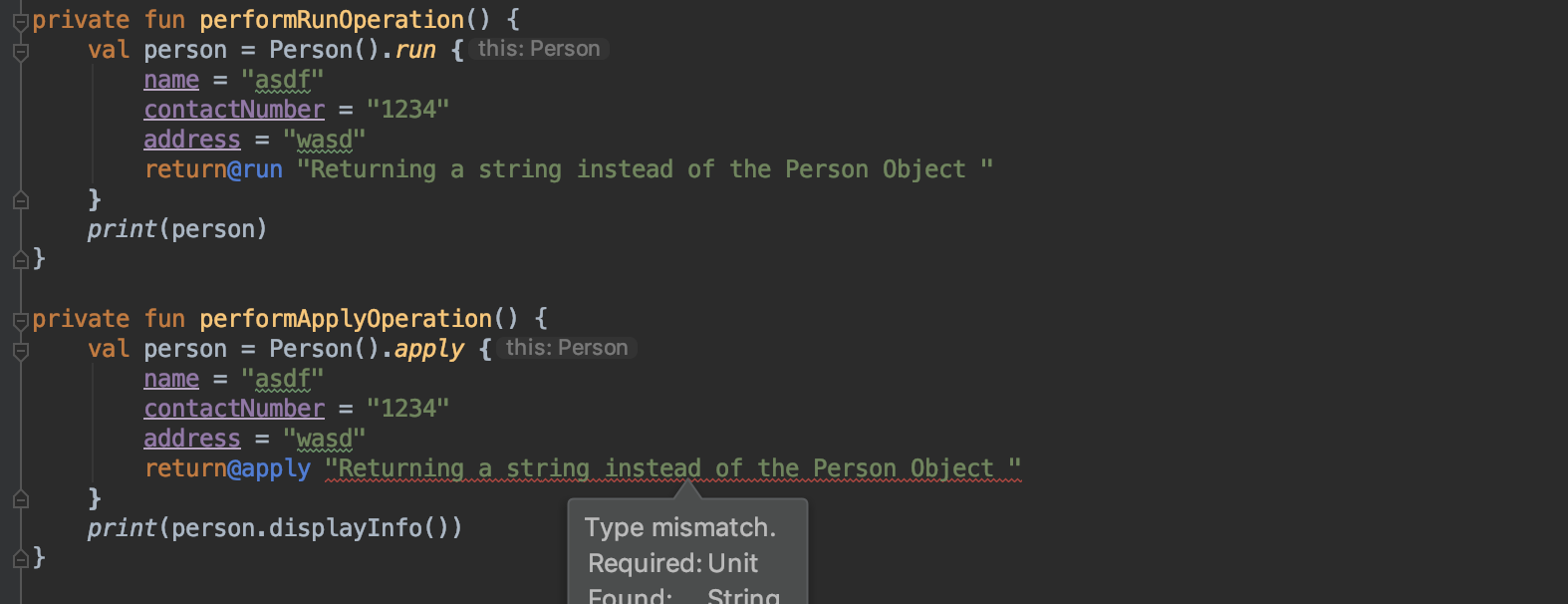
data = Uri.parse(intentData)

}

The improved approach from the above code snippet helps in avoiding variable name redundancy thereby enhancing the code readability and the principle of clean code.

#### apply vs run

So let’s see the difference between **apply** and **run** functions.



We can see that **run** accepts a return statement whereas **“** apply” does not accept a return statement(we can see the error thrown by the IDE in the image) and always returns the same object which it is referring to.

#### also

The “ ***also”*** function is similar to the ***let*** functionality only in terms of referring to the context of the object as “ **it** ” and not “ **this** ” and also in providing null safety checks:

**private** fun **performAlsoOperation**() {

val name = Person().also { currentPerson ->

print("Current name is: ${currentPerson.name}\n")

currentPerson.name = "modifiedName"

}.run {

"Modified name is: $name\n"

}

print(name)

}

output:

Current name is: Abcd

Modified name is: modifiedName

Here we can see the usage of the readable lambda parameters, similar to what we have seen in the “ **let** ” operator case. The advantage of using “ **also** ” operator is that while doing a chain of operations, this operator helps in evaluating the current operating value if required. Let’s suppose we did not use the “ **also”** operator in the above case, we should have written the code as:

var name = Person().name

**print**("Current name is: $name\n")

name = "modifiedName"

name = name.run {

"Modified name is: $name\n"

}

print(name)

Although this gives the same output, the chain in which the operations are being held is broken and that would not make a good readable code and hence “ **also** ” operator is very useful in this case. Hence, by using the also operator, the code can be modified as:

**private** fun **performAlsoOperation**() {

val name = Person().also {

print("Current name is: ${it.name}\n")

it.name = "ModifiedName"

}.run {

"Modified name is: $name\n"

}

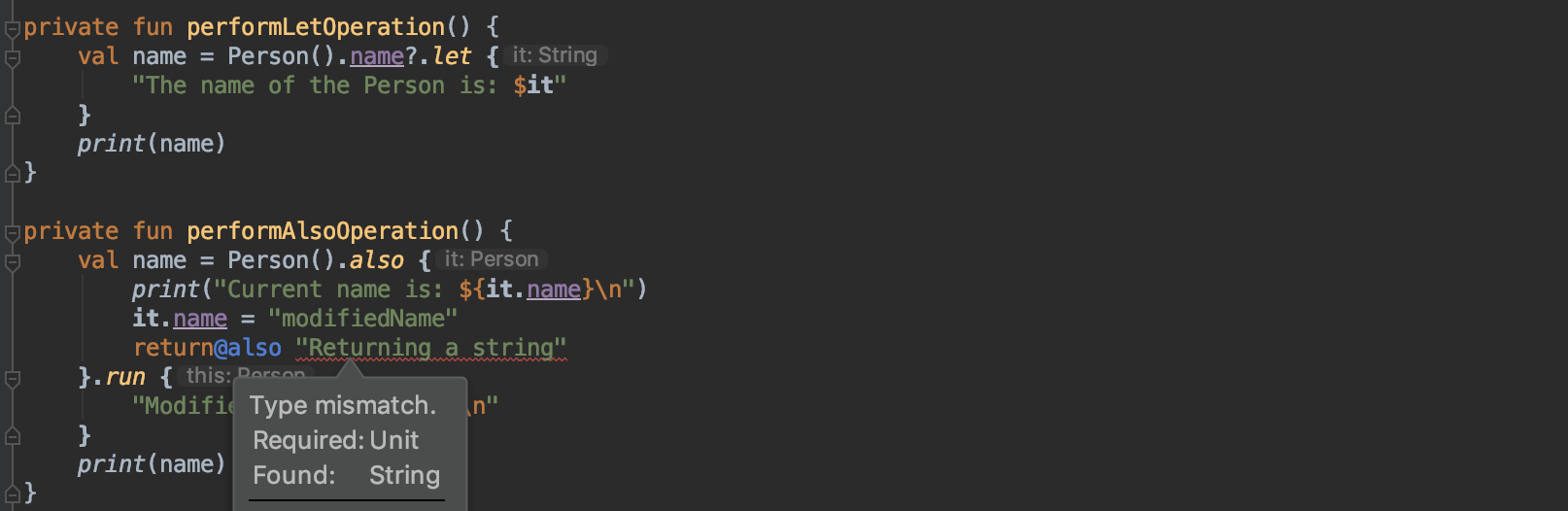
print(name)

}

This way we can execute the intermediate result within the execution chain without breaking the chain, thereby leading to a better readable code.

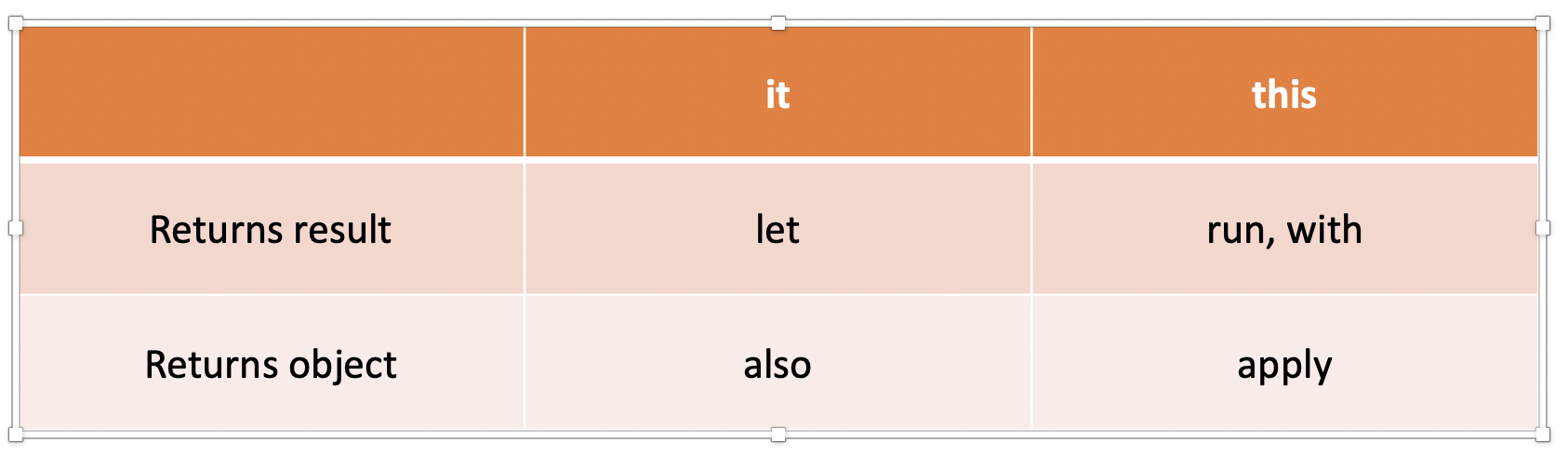
#### “also” vs “let”

So let’s see the difference between **also** and **let** functions.

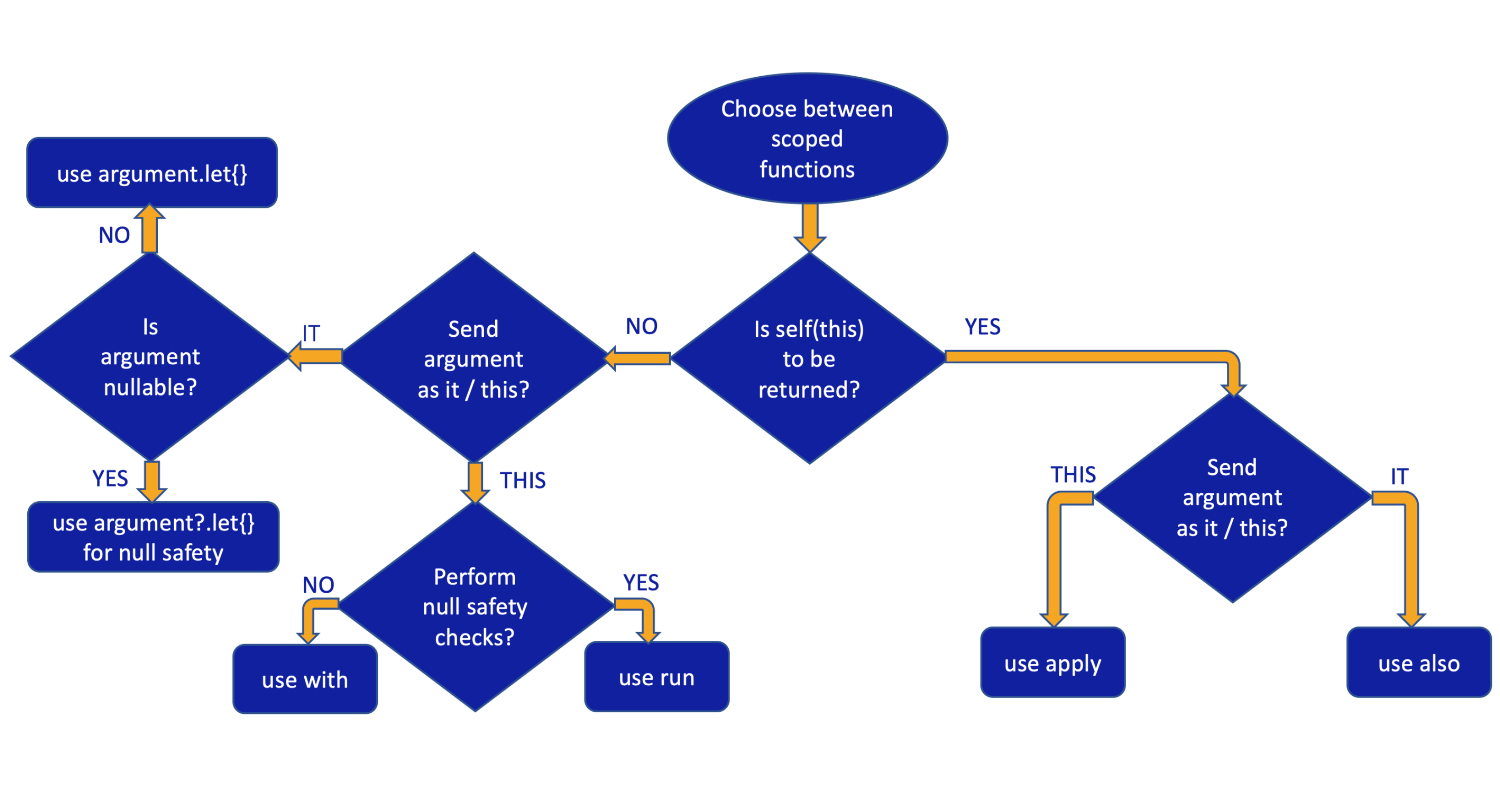


We can see that **let** accepts a return statement whereas **“also** ” does not accept a return statement(we can see the error thrown by the IDE in the image) and always returns the same object which it is referring to.

#### Summary

****

We can also represent the summary of choosing between the scoped functions as a flowchart.



That’s all about the Scoped functions in Kotlin. Although we might be using this in our code, we hope that our article has given you a clearer way of understanding and implementing the right scoped function in the right place.

**Question:**  What are pair and triple in Kotlin?

**Answer:** Pair and Triples are used to return two and three values respectively from a function and the returned values can be of the same data type or different.

val pair = Pair("My Age: ", 25)

print(pair.first + pair.second)

**Question:**  What are labels in Kotlin?

**Answer:** Any expression written in Kotlin is called a label. For example, if we are having a for-loop in our Kotlin code then we can name that for-loop expression as a label and will use the label name for the for-loop .

We can create a label by using an identifier followed by the @ sign. For example, name@ , loop@ , xyz@ , etc. The following is an example of a label:

loop@ for (i in 1..10) {

// some code goes here

}

The name of the above for-loop is loop .

**Question:**  What are the benefits of using a Sealed Class over Enum?

**Answer:** Sealed classes give us the flexibility of having different types of subclasses and also containing the state . The important point to be noted here is the subclasses that are extending the Sealed classes should be either nested classes of the Sealed class or should be declared in the same file as that of the Sealed class.

**Question:**  What are collections in Kotlin?

**Answer:** In our Android application, we deal with various collections such as lists, maps, sets, etc. We perform many operations on these, but do you know there are many Kotlin Collection Functions, that can make our life easier. There is a high probability that if you are want to perform some operation(simple or complex) on any collection, then there must be some collection function that will make your whole Android App development simpler and easier.

In this blog, we will learn about many of these Kotlin Collection Functions. You can bookmark this blog and keep this blog as the cheat sheet for your Kotlin Collection Functions. So, let's start with a simple operation i.e. removing the duplicate strings from an array.

### Remove Duplicate Strings

There are many ways to remove duplicate strings from an array:

*// Maintain the original order of items*

val devs = arrayOf("Amit", "Ali", "Amit", "Sumit", "Sumit", "Himanshu")

print(devs.distinct()) *// [Amit, Ali, Sumit, Himanshu]*

*// Maintain the original order of items*

val devs = arrayOf("Amit", "Ali", "Amit", "Sumit", "Sumit", "Himanshu")

print(devs.toSet()) *// [Amit, Ali, Sumit, Himanshu]*

*// Maintain the original order of items*

val devs = arrayOf("Amit", "Ali", "Amit", "Sumit", "Sumit", "Himanshu")

print(devs.toMutableSet()) *// [Amit, Ali, Sumit, Himanshu]*

*// DO NOT Maintain the original order of items*

val devs = arrayOf("Amit", "Ali", "Amit", "Sumit", "Sumit", "Himanshu")

print(devs.toHashSet()) *// [Amit, Ali, Sumit, Himanshu]*

### Convert an array or list to a string

You can convert an array or list into a string by using joinToString . For example, if you are having a list of cities(Delhi, Mumbai, Bangalore), then you can convert that list into a string such as "India is one the best country for tourism. You can visit Delhi, Mumbai, Bangalore, etc, and enjoy your holidays". Here, Delhi, Mumbai, Bangalore are the list items which you were having.

val someKotlinCollectionFunctions = listOf(

"distinct", "map",

"isEmpty", "contains",

"filter", "first",

"last", "reduce",

"single", "joinToString"

)

val message = someKotlinCollectionFunctions.joinToString(

separator = ", ",

prefix = "Kotlin has many collection functions like: ",

postfix = "and they are awesome.",

limit = 3,

truncated = "etc "

)

print(message) *// Kotlin has many collection functions like: distinct, map, isEmpty, etc and they are awesome.*

### Transform a collection into a single result

If you want to transform a given collection into a single result, then you can use reduce function. For example, you can find the sum of all the elements present in a list:

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

val result = numList.reduce { result, item ->

result + item

}

print(result) *// 15*

*// NOTE: If the list is empty, then it will throw a RuntimeException*

### Find if all elements are satisfying a particular condition

If you have an array or list of data elements and you want to find whether or not all the elements are satisfying a particular condition, then you can use all in Kotlin.

data class **User**(val id: Int, val name: String, val isCricketLover: Boolean, val isFootballLover: Boolean)

val user1 = User(id = 1, name = "Amit", isCricketLover = **true**, isFootballLover = **true**)

val user2 = User(id = 2, name = "Ali", isCricketLover = **true**, isFootballLover = **true**)

val user3 = User(id = 3, name = "Sumit", isCricketLover = **true**, isFootballLover = **false**)

val user4 = User(id = 4, name = "Himanshu", isCricketLover = **true**, isFootballLover = **false**)

val users = arrayOf(user1, user2, user3, user4)

val allLoveCricket = users.all { it.isCricketLover }

print(allLoveCricket) *// true*

val allLoveFootball = users.all { it.isFootballLover }

print(allLoveFootball) *// false*

### Find a particular element based on a certain condition

You can find a particular element from a list of elements that is satisfying a particular condition by using find and single in Kotlin. For example, out of a list of students, you can find the student having roll number 5.

The find returns the first element matching the given condition or null if no such element was found.

While single returns the single element matching the given condition or it will throw an exception if there are more than one matching element or no matching element in the list.

data class **User**(val id: Int, val name: String)

val users = arrayOf(

User(1, "Amit"),

User(2, "Ali"),

User(3, "Sumit"),

User(4, "Himanshu")

)

val userWithId3 = users.single { it.id == 3 }

print(userWithId3) *// User(id=3, name=Sumit)*

val userWithId1 = users.find { it.id == 1 }

print(userWithId1) *// User(id=1, name=Amit)*

### Break your list into multiple sublists of smaller size

There are many cases when you have a bigger list and you want to divide it into smaller parts and then perform some operation on those sublists. So, this can be easily achieved using the chunked function.

val numList = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

val chunkedLists = numList.chunked(3)

print(chunkedLists) *// [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10]]*

### Making copies of the array

You can make copies of your existing array by using various functions such as:

* **copyInto** : This will replace the elements of one array into another array or it will throw an exception if the destination array can't hold the elements of the original array due to size constraints or the indexes are out of bounds.

val arrayOne = arrayOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

val arrayTwo = arrayOf(11, 12, 13, 14, 15, 16, 17, 18, 19, 20)

arrayOne.copyInto(destination = arrayTwo, destinationOffset = 2, startIndex = 0, endIndex = 4)

arrayTwo.forEach {print("$it ")} *// 11 12 1 2 3 4 17 18 19 20*

Similarly, there are other functions that can be used to copy the elements of an array. For example:

* **copyOfRange(fromIndex, toIndex)** : Returns a new array which is a copy of the specified range of the original array.
* **copyOf()** or **copyOf(newSize)** : Returns a new array which is a copy of the original array, resized to the given newSize, or if the newSize is not passed then the whole array will be copied.

### Changing type of collection to other

Depending on the situation, you can change the type of collection. Here, either you can change the type of one collection to another type by making a new collection or by referring to the older one. For example:

**toIntArray** , **toBooleanArray** , **toLongArray** , **toShortArray** , **toByteArray** , **toDoubleArray** , **toList** , **toMap** , **toSet** , **toPair** , etc can be used to change the type of one collection to another type.

var uIntArray = UIntArray(5) { 1U }

var intArray = uIntArray.toIntArray()

intArray[0] = 0

print(uIntArray.toList()) *// [1, 1, 1, 1, 1]*

print(intArray.toList()) *// [0, 1, 1, 1, 1]*

Here, we are making a new collection and changes in the new collection will not be reflected in the older one. But, at the same time, you can change the type of collection by keeping the reference to the older one i.e. changes in one collection will automatically be reflected in the other. For this instead of to , we need to use as . For example:

**asIntArray** , **asLongArray** , **asShortArray** , **asByteArray** , **asList** , etc.

var uIntArray = UIntArray(5) { 1U }

var intArray = uIntArray.asIntArray()

intArray[0] = 0

print(uIntArray.toList()) *// [0, 1, 1, 1, 1]*

print(intArray.toList()) *// [0, 1, 1, 1, 1]*

### Associating the data using some key

If you are having a list of data and you want to associate the data with the help of some key present in your data element, then you can use associateBy .

data class **Contact**(val name: String, val phoneNumber: String)

val contactList = listOf(

Contact("Amit", "+9199XXXX1111"),

Contact("Ali", "+9199XXXX2222"),

Contact("Himanshu", "+9199XXXX3333"),

Contact("Sumit", "+9199XXXX4444")

)

val phoneNumberToContactMap = contactList.associateBy { it.phoneNumber }

print(phoneNumberToContactMap)

*// Map with key: phoneNumber and value: Contact*

*// {*

*// +9199XXXX1111=Contact(name=Amit, phoneNumber=+9199XXXX1111),*

*// +9199XXXX2222=Contact(name=Ali, phoneNumber=+9199XXXX2222),*

*// +9199XXXX3333=Contact(name=Himanshu, phoneNumber=+9199XXXX3333),*

*// +9199XXXX4444=Contact(name=Sumit, phoneNumber=+9199XXXX4444)*

*// }*

In the above example, the key is phoneNumber and the value is Contact. If you don't want to have the whole Contact as the value, then you can simply pass the desired value like this:

val phoneNumberToContactMap = contactList.associateBy({it.phoneNumber}, {it.name})

print(phoneNumberToContactMap)

*// Map with key: phoneNumber and value: name*

*// {*

*// +9199XXXX1111=Amit,*

*// +9199XXXX2222=Ali,*

*// +9199XXXX3333=Himanshu,*

*// +9199XXXX4444=Sumit}*

*// }*

### Finding distinct elements in a collection

We can use the distinct function to get the list of unique elements of a collection.

val list = listOf(1, 2, 2, 3, 3, 3, 4, 4, 4, 4)

println(list.distinct()) *// [1, 2, 3, 4]*

### Union of collections

You can use the union function to get the unique elements of two collections. The order of the elements of both the collections will be preserved but the elements of the second collection will be added after the elements of the first collection.

val listOne = listOf(1, 2, 3, 3, 4, 5, 6)

val listTwo = listOf(2, 2, 4, 5, 6, 7, 8)

println(listOne.union(listTwo)) *// [1, 2, 3, 4, 5, 6, 7, 8]*

### Intersection of collections

To get the elements that are common in two collections, you can use the intersect function which returns a set containing the common element of both collections.

val listOne = listOf(1, 2, 3, 3, 4, 5, 6)

val listTwo = listOf(2, 2, 4, 5, 6, 7, 8)

println(listOne.intersect(listTwo)) *// [2, 4, 5, 6]*

### Keep the specified elements only

If in a collection, you want to keep the specified elements only then you can use retainAll function. Since this function will modify your list, so make sure that your list or array is mutable.

retainAll will return true if any element is removed from the collection otherwise it will return false.

val listOne = mutableListOf(1, 2, 3, 3, 4, 5, 6)

val listTwo = listOf(1, 2, 3, 3, 4, 5, 6)

val listThree = listOf(1, 2, 3, 3, 4, 5, 7)

println(listOne.retainAll(listTwo)) *// false*

println(listOne.retainAll(listThree)) *// true*

println(listOne) *// [1, 2, 3, 3, 4, 5]*

Similarly, you can use removeAll to remove all the elements of one collection that are present in another collection.

### Filter a collection based on some condition

You can filter a collection based on certain conditions by using the filter. This returns a list containing elements that satisfy the given condition.

val list = listOf(1, 2, 3, 4, 5, 6, 7, 8)

val filteredList = list.filter { it % 2 == 0 }

print(filteredList) *// [2, 4, 6, 8]*

Similarly, you can filter the collection based on the index of elements by using filterIndexed .

If you want to store the filtered elements in some collection, then you can use the filterIndexedTo :

val list = listOf(1, 2, 3, 4, 5, 6, 7, 8)

val filteredList = mutableListOf<Int>()

list.filterIndexedTo(filteredList) { index, i -> list[index] % 2 == 0 }

print(filteredList) *// [2, 4, 6, 8]*

You can also find the elements that are instances of a specified type in a collection by using filterIsInstance .

val mixedList = listOf(1, 2, 3, "one", "two", 4, "three", "four", 5, 6, "five", 7)

val strList = mixedList.filterIsInstance<String>()

print(strList) *// [one, two, three, four, five]*

### Zip collections

zip returns a list of pairs. The first element of the pair will be taken from the first collection and the second element of the pair will be taken from the second collection. The size of the returned list will be equal to the size of the shortest collection.

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

val listTwo = listOf("a", "b", "c", "d", "e", "f")

print(listOne zip listTwo) *// [(1, a), (2, b), (3, c), (4, d), (5, e)]*

### Zip with next in a collection

zipWithNext return a list of pairs. The elements of the pair will be the adjacent elements of the collection.

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

print(list.zipWithNext()) *// [(1, 2), (2, 3), (3, 4), (4, 5)]*

### Unzip a collection

unzip returns a pair of lists. The first list is made from the first elements of each pair and the second list is made from the second element of each pair.

val list = listOf("Amit" to 8, "Ali" to 10, "Sumit" to 4, "Himanshu" to 2)

val (players, footballSkills) = list.unzip()

print(players) *// [Amit, Ali, Sumit, Himanshu]*

print(footballSkills) *// [8, 10, 4, 2]*

### Split array into two parts based on some condition

If you want to split your data into two parts based on some conditions like isFootballFan, then you can use partition .

data class **User**(val id: Int, val name: String, val isFootballLover: Boolean)

val users = listOf(

User(1, "Amit", **true**),

User(2, "Ali", **true**),

User(3, "Sumit", **false**),

User(4, "Himanshu", **false**)

)

val (footballLovers, nonFootballLovers) = users.partition { it.isFootballLover }

print(footballLovers) *// [User(id=1, name=Amit, isFootballLover=true), User(id=2, name=Ali, isFootballLover=true)]*

print(nonFootballLovers) *// [User(id=3, name=Sumit, isFootballLover=false), User(id=4, name=Himanshu, isFootballLover=false)]*

### Reverse a list

You can reverse a list in Kotlin by using the reversed and asReversed function.

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

print(list.reversed()) *// [5, 4, 3, 2, 1]*

print(list.asReversed()) *// [5, 4, 3, 2, 1]*

Both are giving the same output but these functions are different.

The reversed() function can be applied on Array, List, and MutableList. It generates a new list that is the reverse of the original list.

But the asReversed() function can be applied on List and MutableList. It doesn't generate a new list because, after reversal, the new elements are still referring to the old one. So any change in one of them will result in a change in the other one.

Similarly, there are other functions that can be used to reverse the elements such as reversedArray() , reverse() .

### Group elements of a collection based on some condition

You can use groupBy() to group the elements of a collection based on certain conditions. For example, the below code will group the elements of the list based on the remainder when divided by 4 i.e. 4 groups will be there(when remainder = 0, 1, 2, and 3)

val list = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

print(list.groupBy { it % 4 })

*// {*

*// 1=[1, 5, 9],*

*// 2=[2, 6, 10],*

*// 3=[3, 7],*

*// 0=[4, 8]*

*// }*

### Sort element of a collection

You can sort the elements of a collection by using the sorted() function. This will return a sorted list.

val list = listOf(10, 4, 1, 3, 7, 2, 6)

print(list.sorted()) *// [1, 2, 3, 4, 6, 7, 10]*

Similarly, there are other functions that can be used to sort the collection based on certain conditions. Some of these functions are **sortedArray** , **sortedArrayWith** , **sortedBy** , **sortedByDescending** , **sortedArraydescending** , **sortedWith** , etc.

So, these are some of the collection functions that can be used while dealing with collections in Kotlin.

**Question:**  What is Kotlin? / Describe Kotlin in brief.

**Answer:** Kotlin is a general-purpose, statically typed, open-source programming language that runs on the JVM. It runs on JVM and can be used anywhere Java is used today. It can be compiled either using Java source code or LLVM compiler. It is generally used to develop Android apps, server-side apps, and much more.

**Question:**  What do you know about the history of Kotlin?

**Answer:** Kotlin was developed by the JetBrains team. This project was started in 2010 to develop a language for Android apps development, and officially its first version was released in February 2016. Kotlin was developed under the Apache 2.0 license.

**Question:**  What are the most important features of Kotlin?

**Answer:** The most popular features of kotlin are:

Kotlin is Concise: Kotlin reduces the writing of the extra codes, making Kotlin more concise.

Compact code: Kotlin is an OOPs-based programming language. Its code lines may be reduced by up to 40% compared to Java, making it an excellent choice for software development.

Kotlin is Simple: Kotlin is a simple language to learn. When working with Kotlin, compiling the code is simple, resulting in improved performance for Android development. It also explains which types of data functions can be used throughout the code.

Open Source: Kotlin is open source for Android, and it uses the JVM to combine the benefits of OOPs and functional programming.

Null safety: Kotlin is null safety language. Kotlin aimed to eliminate the NullPointerException (null reference) from the code.

A high number of extensions: Kotlin supports various extension functions and extension properties without modifying the code. It means that it can help to extend the functionality of classes without touching their code. Kotlin may support a variety of extension functions to help developers make existing code more appealing and wonderful.

Full Java Interoperability: Kotlin provides full interoperability for Java code. Java code can utilize Kotlin code, and Kotlin code can use Java code. So, if you are familiar with OOPs programming and good in Java programming language, you can switch to Kotlin development easily. Also, if there are any Java-based applications, they can be used with Kotlin's environment.

Smart Cast: Kotlin supports the smart cast technique. By using this technique, we can reduce the cost of an application while also improving its speed and performance. It technique uses typecasting or immutable data to manage the efficiency of programming. \

Low Learning Curve: Many businesses prefer Kotlin because of its low adoption cost. Most significantly, it is simple for developers to learn, especially if they have programming experience.

Compilation Time: Kotlin is faster and better than Java in terms of its performance and fast compilation time.

Tools-friendly: Kotlin is Tools-friendly. You can build the Kotlin programs by using the command line as well as any of Java IDE.

**Question:**  Why did you switch to Kotlin from Java? Why do some developers like to switch to Kotlin from Java?

**Answer:** The Kotlin programing language seems to be simpler and cleaner than Java. It removes a lot of redundancies in code as compared to Java. Kotlin also offers some useful features that Java doesn't yet support, making the code more idiomatic. Kotlin has been added to Android Studio's list of supported languages recently. So, there is much to expect from Kotlin in easing out the development efforts and good support in the future.

**Question:**  How does Kotlin work on Android?

**Answer:** Kotlin is very much similar to the Java programming language. Like Java, the Kotlin code is also compiled into the Java bytecode and executed at runtime by the Java Virtual Machine, i.e., JVM. For example, when a Kotlin file named Main.kt is compiled, it will eventually turn into a class, and then the bytecode of the class will be generated. The name of the bytecode file will be MainKt.class, and this file will be executed by the JVM.

**Question:**  What is the difference between the variable declaration with var and variable declaration with val?

**Answer:** The variable declaration with var and the variable declaration with val is used for different purposes. If you want to declare some mutable (changeable) variable, you should use var. If you want to declare the immutable variable, you should use val because val variables can't be changed once you have assigned them.

**Question:**  What is the difference between the variable declaration with val and variable declaration with const?

**Answer:** Both the variables that are declared with val and const are immutable in nature. But the difference between the variable declaration with val and variable declaration with const is that the value of the const variable must be known at the compile-time. In contrast, the value of the val variable can be assigned at runtime also.

**Question:**  How can you create a singleton in Kotlin?

**Answer:** We can create a singleton in Kotlin by using an object.

**Syntax:**

1. object SomeSingleton

The above Kotlin object will be compiled to the following equivalent Java code:

1. public final class SomeSingleton {
2. public static final SomeSingleton INSTANCE;
3. private SomeSingleton() {
4. INSTANCE = (SomeSingleton)this;
5. System.out.println("init complete");
6. }
7. static {
8. new SomeSingleton();
9. }
10. }

The above way is preferred to implement singletons on a JVM because it enables thread-safe lazy initialization without relying on a locking algorithm like the complex double-checked locking.

**Question:**  What is a primary constructor in Kotlin?

**Answer:** In Kotlin, the primary constructor is a part of the class header. Unlike Java, it doesn't need you to declare a constructor in the body of the class.

Kotlin facilitates you to declare the constructor in the class header itself:

**See the following example:**

1. class Person constructor(name: String, age: Int, salary: Int) {
2. }

Just like functions or methods, it takes a series of parameters with their type. These parameters initialize the variables present in the class.

If you do not have any annotations or modifiers (public, private, protected), you can omit the constructor keyword like the following example.

1. class Person (name: String, age: Int, salary: Int) {
2. }

By removing the constructor keyword, you can get code that is simplified and easy to understand.

**Question:**  What do you understand by Null safety in Kotlin?

**Answer:** Null safety is one of the major advantages of using Kotlin. Kotlin's type system ensures eliminating the danger of null references from code, also known as The Billion Dollar Mistake. One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference will result in a null reference exception. In Java, this would be the equivalent of a NullPointerException or NPE for short.

In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that cannot (non-null references). For example, a regular variable of type String can not hold null:

1. var a: String = "abc"
2. a = null // compilation error

To allow nulls, we can declare a variable as nullable string, written **"String?":**

1. var b: String? = "abc"
2. b = null // ok
3. print(b)

**Question:**  How can you ensure null safety in Kotlin?

**Answer:** Null safety is a feature introduced in Kotlin. In Java, if you access some null variable, then you will get a NullPointerException. So, the following code in Kotlin will produce a compile-time error:

1. var name: String = "MindOrks"
2. name = null //error

So, to overcome this issue, you have to assign null values to a variable, and 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.?.

1. var name: String? = "MindOrks"
2. print(name?.length) // ok
3. name = null // ok

**Question:**  What is a data class in Kotlin?

**Answer:** In Kotlin, a data class is a class whose main purpose is to hold data. It is marked as "data".

**Syntax:**

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

The data classes must have to fulfill the following requirements to ensure consistency and meaningful behavior of the generated code:

* The primary constructor must have at least one parameter
* , and all primary constructor parameters need to be marked as val or var.
* Data classes cannot be abstract, open, sealed, or inner.

**Question:**  What is the default behavior of Kotlin classes?

**Answer:** By default, all classes are final in Kotlin. That's because Kotlin allows multiple inheritances for classes, and an open class is more expensive than a final class.

**Question:**  Does Kotlin provide support for primitive Datatypes?

**Answer:** No. Kotlin does not provide support for primitive Data types like in Java.

**Question:**  Does Kotlin provide support for macros?

**Answer:** No. Kotlin does not provide support for macros because the developers of Kotlin find it difficult to include them in the language.

**Question:**  What is the use of the open keyword in Kotlin?

**Answer:** 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.

**Question:**  What do you understand by the Ranges operator in Kotlin?

**Answer:** Ranges operators help to iterate within a range. Its operator form is (..) For Example:

1. for (i in 1..15)
2. print(i)

The above example will give the output to print from 1 to 15.

**Question:**  Where should we use var and where val in Kotlin?

**Answer:** In Kotlin, var is used where value is frequently changing. For example, while getting the location of the android device:

1. var integerVariable : Int? = null

In Kotlin, val is used where there is no change in value in the whole class. For example, when you want to set textview or button's text programmatically:

1. val stringVariables : String = "Button's Constant or final Text"

**Question:**  What is the difference between a safe calls(?.) and a null check(!!) in Kotlin?

**Answer:** Difference between safe calls(?.) and a null check(!!) in Kotlin:

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

1. var name: String? = "JavaTpoint"
2. println(name?.length) // 10
3. name = null
4. println(name?.length) // null

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

**See the example:**

1. var name: String? = "JavaTpoint"
2. println(name?.length) // 10
3. name = null
4. println(name!!.length) // KotlinNullPointerException

**Question:**  What is the basic difference between the fold and reduce in Kotlin? Also, specify when to use which?

**Answer:** **Difference between the fold and reduce in Kotlin:**

**Fold:** The **fold** takes an initial value and the first invocation of the lambda you pass to it. It will receive that initial value and the first element of the collection as parameters.

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

The first call to the lambda will be with parameters 0 and 1. The ability to pass in an initial value is useful if you have to provide a default value or parameter for your operation.

**Reduce:** The **"reduce"** doesn't take an initial value. Instead, it starts with the first element of the collection as the accumulator.

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

In the above example, it is denoted by sum. The first call to the lambda here will be with parameters 1 and 2.

**Question:**  What are the advantages of "when" over "switch" in Kotlin?

**Answer:** The "switch" is used in Java, but in Kotlin, that switch gets converted to "when". When has a better design as compared to "switch", and it is more concise and powerful than a traditional switch. We can use "when" either as an expression or as a statement.

**Following are some examples of when usage in Kotlin:**

**In two or more choices:**

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. }

**Any type passed in "when":**

1. fun describe(obj: Any): String =
2. when (obj) {
3. 1 -**>** "One"
4. "Hello" -**>** "Greeting"
5. is Long -**>** "Long"
6. !is String -**>** "Not a string"
7. else -**>** "Unknown"
8. }

**Smart casting:**

1. when (x) {
2. is Int -**>** print("X is integer")
3. is String -**>** print("X is string")
4. }

**Ranges:**

1. when(number) {
2. 1 -**>** println("One") //statement 1
3. 2 -**>** println("Two") //statement 2
4. 3 -**>** println("Three") //statement 3
5. in 4..8 -**>** println("Number between 4 and 8") //statement 4
6. !in 9..12 -**>** println("Number not in between 9 and 12") //statement 5
7. else -**>** println("Number is not between 1 and 8") //statement 6
8. }

**Question:**  What do you understand by the Null safety in Kotlin?

**Answer:** In Kotlin, the main motive of the type system is to eliminate the danger of null references from code. It is also known as the Billion Dollar Mistake.

One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference will result in a null reference exception. In Java, this would be the equivalent of a NullPointerException.

In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that cannot (non-null references). For example, a regular variable of type String can not hold null:

1. var a: String = "abc"
2. a = null // compilation error

**To allow nulls, we can declare a variable as nullable string, written String?:**

1. var b: String? = "abc"
2. b = null // ok
3. print(b)

**Question:**  Do we have a ternary operator in Kotlin just like Java?

**Answer:** No. In Kotlin, we don't have a ternary operator like Java, but we can use the functionality of the ternary operator by using if-else or Elvis operator.

**Question:**  What is Elvis operator in Kotlin?

**Answer:** In Kotlin, we can assign null values to a variable using the null safety property. To check if a value has null value, we can use if-else or can use the Elvis operator i.e. ?:

**For example:**

1. var name:String? = "Mindorks"
2. val namenameLength = name?.length ?: -1
3. println(nameLength)

In the above example, the Elvis operator(?:) we are using will return the length of the name if the value is not null; otherwise, if the value is null, then it will return -1.

**Question:**  Why is Kotlin interoperable with Java?

**Answer:** Kotlin is interoperable with Java because it uses JVM bytecode. It provides the facility to compile it directly to bytecode that helps to achieve faster compile-time and makes no difference between Java and Kotlin for JVM.

**Question:**  What do you understand by lazy initialization in Kotlin?

**Answer:** Kotlin provides the facility of lazy initialization, which specifies that your variable will not be initialized unless you use that variable in your code. It will be initialized only once. After that, you use the same value.

In lazy initialization, the lazy() function is used that takes a lambda and returns an instance of lazy, which can serve as a delegate for implementing a lazy property: the first call to get() executes the lambda passed to lazy() and remembers the result, subsequent calls to get() simply return the remembered result.

1. val test: String by lazy {
2. val testString = "some value"
3. }

**Question:**  How many types of constructors are used in Kotlin?

**Answer:** There are two types of constructors available in Kotlin:

* Primary constructor
* Secondary constructor

**Question:**  What is Lateinit in Kotlin, and when is it used?

**Answer:** 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. }

**This is mainly used in the following cases:**

* **Android:** variables that get initialized in lifecycle methods.
* **Using Dagger for DI:** injected class variables are initialized outside and independently from the constructor.
* **Setup for unit tests:** test environment variables are initialized in a @Before - annotated method.
* Spring Boot annotations (e.g., @Autowired).

**Question:**  How can we convert a Kotlin source file to a Java source file?

**Answer:** Follow the steps given below to convert your Kotlin source file to a Java source file:

* First, open your Kotlin project in the IntelliJ IDEA / Android Studio.
* Then navigate to Tools > Kotlin > Show Kotlin Bytecode.
* Now, click on the Decompile button to get your Java code from the bytecode.

**Question:**  What kinds of programming types does Kotlin support?

**Answer:** Kotlin supports the following programming types:

* Procedural Programming
* Object-Oriented Programming

**Question:**  What is the use of @JvmStatic, @JvmOverloads, and @JvmFiled in Kotlin?

**Answer:** Following are the main usage of @JvmStatic, @JvmOverloads, and @JvmFiled in Kotlin:

* **@JvmStatic:** The @JvmStatic annotation is used to tell the compiler that the method is a static method, and you can use it in Java code.
* **@JvmOverloads:** The @JvmOverloads annotation is required when we need to use the default values passed as an argument in Kotlin code from the Java code.
* **@JvmField:** The @JvmField annotation is used to access the fields of a Kotlin class from Java code without any getters and setters. We need to use the @JvmField in the Kotlin code.

**Question:**  What are the names of some extension methods that Kotlin provides to java.io.File?

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

* bufferedReader(): It is used for reading the contents of a file into BufferedReader.
* readBytes(): It is used for reading the contents of the file to ByteArray.
* readText(): It is used for reading contents of the file to a single String.
* forEachLine(): It is used for reading a file line by line in Kotlin.
* readLines(): It is used for reading lines in the file to List.

**Question:**  What do you understand by data classes in Kotlin?

**Answer:** Data classes are the type of classes that are made to store some data. In Kotlin, it is marked as data. The following is an example of a data class:

1. data class Developer(val name: String, val age: Int)

When we mark a class as a data class, we don't have to implement or create the following functions like we have to do in Java: hashCode(), equals(), toString(), copy(). The compiler automatically creates these internally, so it also leads to clean code. Although, there are a few other requirements that data classes need to fulfill.

**Question:**  What is the use of Companion Objects in Kotlin?

**Answer:** Companion Objects are required in Kotlin because Kotlin doesn't have static members or member functions, unlike Java or C#. 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.

**For example:**

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. If you integrate with Java code and need a true static member, you can annotate a member inside a companion object with @JvmStatic.

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

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

**Question:**  How can we perform String Interpolation in Kotlin?

**Answer:** In Kotlin, String Interpolation is used when you want to use some variable or perform some operation inside a string. For String Interpolation, we can use the $ sign to use some variable in the string or can perform some operation in between the {} sign.

**For example:**

1. var name = "JavaTpoint"
2. print("The best tutorial website is: $name")

**Question:**  Name some features which are available in Kotlin but not in Java?

**Answer:** Following are some important Kotlin features that are not available in Java:

* Null Safety
* Operator Overloading
* Coroutines
* Range expressions
* Smart casts
* Companion Objects

**Question:**  What is the difference between == operator and === operator in Kotlin?

**Answer:** 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

**Question:**  Can we use primitive types such as int, double, float in Kotlin?

**Answer:** 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. But the compiled bytecode has these primitive types.

**Question:**  What is the difference between lateinit and lazy in Kotlin?

**Answer:** 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.
* You have to use lateinit, if you want your property to be initialized from outside in a way probably unknown beforehand.

**Lateinit vs. lazy in Kotlin:**

| **Lateinit** | **Lazy** |
| --- | --- |
| The lateinit can be initialized from anywhere the object is seen. | The lazy can only be initialized from the initializer lambda. |
| In lateinit, multiple initializations are possible. | The lazy can be initialized a single time only. |
| The lateinit is non-thread safe. It is up to the user to initialize it correctly in a multi-threaded environment. | The lazy support thread-safety by default and ensures that the initializer is invoked once. |
| It is not eligible for nonnull properties. | It is also not eligible for nonnull properties. |
| You can use it only for var. | You can use it only for val. |
| It adds an isInitialized method to check whether the value has been initialized before. | In this, the property is never able to un-initialize. |
| It is not allowed on properties of primitive types. | It is allowed on properties of primitive types. |

**Question:**  What do you understand by destructuring in Kotlin?

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

**For Example:**

1. val (name, age) = developer

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

1. println(name)
2. println(age)

**Question:**  What are coroutines in Kotlin?

**Answer:** Unlike many other programming languages with similar capabilities, Kotlin doesn't have async and await keywords, and these keywords are not even part of its standard library.

In Kotlin, kotlinx.coroutines is a rich library for coroutines developed by JetBrains. This library contains some high-level coroutine-enabled primitives, including launch, async, and others. Kotlin Coroutines provide us with an API to write our asynchronous code sequentially.

According to Kotlin documentation, Coroutines are like lightweight threads. They are lightweight because while creating them, they don't allocate new threads. Instead, they use predefined thread pools and smart scheduling. Scheduling is the process of determining the work in a sequential process, and it decides which piece of work you will execute next. We can suspend and resume the Coroutines while execution. This means we can have a long-running task, which can be executed one by one. We can pause it any number of times and resume it when required.

**Question:**  What is the difference between Launch and Async in Kotlin Coroutines?

**Answer:** In Kotlin, the main difference between Launch and Async is that the launch{} does not return anything and the async{} returns an instance of Deferred, which has an await() function. In other words, we can say that launch is used to fire and forget, and async is used to perform a task and return a result.

**Question:**  What are the extension functions in Kotlin?

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

1. fun View.show() {
2. this.visibility = View.VISIBLE
3. }
4. fun View.hide() {
5. this.visibility = View.GONE
6. }
7. and to use it, we use, like,
8. toolbar.hide()

**Question:**  What do you understand by the Kotlin double-bang (!!) operator?

**Answer:** The Kotlin double-bang (!!) operator converts any value to a non-null type and throws a KotlinNullPointerException exception if the value is null. It is also called the not-null assertion operator.

1. Example:
2. fun main(args: Array**<String>**) {
3. var email: String?
4. email = null
5. println(email!!)
6. }

This operator should be used in cases where the developer is 100% sure that its value is not null.

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

**Answer:** Problem

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)

**Question:**  How to create singleton in Kotlin?

**Answer:** Answer

Just use object.

object SomeSingleton

The above Kotlin object will be compiled to the following equivalent Java code:

public final class SomeSingleton {

public static final SomeSingleton INSTANCE;

private SomeSingleton() {

INSTANCE = (SomeSingleton)this;

System.out.println("init complete");

}

static {

new SomeSingleton();

}

}

This is the preferred way to implement singletons on a JVM because it enables thread-safe lazy initialization without having to rely on a locking algorithm like the complex double-checked locking.

**Question:**  What is a data class in Kotlin?

**Answer:** We frequently create classes whose main purpose is to hold data. In Kotlin, this is called a data class and is marked as data:

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

To ensure consistency and meaningful behavior of the generated code, data classes have to fulfill the following requirements:

The primary constructor needs to have at least one parameter;

All primary constructor parameters need to be marked as val or var;

Data classes cannot be abstract, open, sealed or inner;

**Question:**  What is basic difference between fold and reduce in Kotlin? When to use which?

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

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

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

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

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

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

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

**Question:**  What is the difference between var and val in Kotlin?

* **var** is like general variable and it's known as a *mutable* variable in kotlin and can be assigned multiple times.
* **val** is like Final variable and it's known as *immutable* in Kotlin and can be initialized only single time.

+----------------+-----------------------------+---------------------------+

| | val | var |

+----------------+-----------------------------+---------------------------+

| Reference type | Immutable(once initialized | Mutable(can able to change|

| | can't be reassigned) | value) |

+----------------+-----------------------------+---------------------------+

| Example | val n = 20 | var n = 20 |

+----------------+-----------------------------+---------------------------+

| In Java | final int n = 20; | int n = 20; |

+----------------+-----------------------------+---------------------------+

**Question:**  Where should I use var and where val?

**Answer:** Use var where value is changing frequently. For example while getting location of android device:

var integerVariable : Int? = null

Use val where there is no change in value in whole class. For example you want set textview or button's text programmatically.

val stringVariables : String = "Button's Constant or final Text"

**Question:**  Explain advantages of when vs switch in Kotlin

**Answer:** In Java we use switch but in Kotlin, that switch gets converted to when. When has a better design. It is more concise and powerful than a traditional switch. when can be used either as an expression or as a statement.

Some examples of when usage:

Two or more choices

when(number) {

1 -> println("One")

2, 3 -> println("Two or Three")

4 -> println("Four")

else -> println("Number is not between 1 and 4")

}

"when" without arguments

when {

number < 1 -> print("Number is less than 1")

number > 1 -> print("Number is greater than 1")

}

Any type passed in "when"

fun describe(obj: Any): String =

when (obj) {

1 -> "One"

"Hello" -> "Greeting"

is Long -> "Long"

!is String -> "Not a string"

else -> "Unknown"

}

Smart casting

when (x) {

is Int -> print("X is integer")

is String -> print("X is string")

}

Ranges

when(number) {

1 -> println("One") //statement 1

2 -> println("Two") //statement 2

3 -> println("Three") //statement 3

in 4..8 -> println("Number between 4 and 8") //statement 4

!in 9..12 -> println("Number not in between 9 and 12") //statement 5

else -> println("Number is not between 1 and 8") //statement 6

**Question:**  Explain the null safety in Kotlin

**Answer:** Kotlin's type system is aimed at eliminating the danger of null references from code, also known as the The Billion Dollar Mistake.

One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference will result in a null reference exception. In Java this would be the equivalent of a NullPointerException or NPE for short.

In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that can not (non-null references). For example, a regular variable of type String can not hold null:

var a: String = "abc"

a = null // compilation error

To allow nulls, we can declare a variable as nullable string, written String?:

var b: String? = "abc"

b = null // ok

print(b)

**Question:**  Explain what is wrong with that code?

**Answer:** Problem

Why is this code wrong?

class Student (var name: String) {

init() {

println("Student has got a name as $name")

}

constructor(sectionName: String, var id: Int) this(sectionName) {

}

}

Answer

The property of the class can’t be declared inside the secondary constructor.. This will give an error because here we are declaring a property id of the class in the secondary constructor, which is not allowed.

If you want to use some property inside the secondary constructor, then declare the property inside the class and use it in the secondary constructor:

class Student (var name: String) {

var id: Int = -1

init() {

println("Student has got a name as $name")

}

constructor(secname: String, id: Int) this(secname) {

this.id = id

}

}

**Question:**  How is it recommended to create constants in Kotlin?

**Answer:** Answer

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

val MY\_CONSTANT\_1 = "Constants1"

// or

const val MY\_CONSTANT\_2 = "Constants2"

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

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

object DbConstants {

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

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

**Question:**  How would you refactor this code using apply?

**Answer:** Problem

Consider:

class Message(message: String, signature: String) {

val body = MessageBody()

init {

body.text = message + "\n" + signature

}

}

Do you see any refactoring that could be done?

Answer

You can write:

class Message(message: String, signature: String) {

val body = MessageBody().apply {

text = message + "\n" + signature

}

}

**Question:**  May you use IntArray and an Array<Int> is in Kotlin interchangeably?

**Answer:** Answer

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

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

So no, we can't use them interchangeably.

**Question:**  Rewrite this code in Kotlin

**Answer:** Problem

Can you rewrite this Java code in Kotlin?

public class Singleton {

private static Singleton instance = null;

private Singleton(){

}

private synchronized static void createInstance() {

if (instance == null) {

instance = new Singleton();

}

}

public static Singleton getInstance() {

if (instance == null) createInstance();

return instance;

}

Answer

Using Kotlin:

object Singleton

**Question:**  What are coroutines in Kotlin?

**Answer:** Answer

Unlike many other languages with similar capabilities, async and await are not keywords in Kotlin and are not even part of its standard library.

kotlinx.coroutines is a rich library for coroutines developed by JetBrains. It contains a number of high-level coroutine-enabled primitives, including launch, async and others. Kotlin Coroutines give you an API to write your asynchronous code sequentially.

The documentation says Kotlin Coroutines are like lightweight threads. They are lightweight because creating coroutines doesn’t allocate new threads. Instead, they use predefined thread pools, and smart scheduling. Scheduling is the process of determining which piece of work you will execute next.

Additionally, coroutines can be suspended and resumed mid-execution. This means you can have a long-running task, which you can execute little-by-little. You can pause it any number of times, and resume it when you’re ready again.

**Question:**  What are some disadvantages of Kotlin?

**Answer:** Some think that Kotlin is a mess of extra syntax and keywords. Here are a few keywords which have non-obvious meanings: internal, crossinline, expect, reified, sealed, inner, open. Java has none of these. Kotlin is also amusingly inconsistent in its keywords: a function is is declared with ‘fun’, but an interface is declared with ‘interface’ (not ‘inter’?). Kotlin also doesn’t have checked exceptions. Checked exceptions have become unfashionable, yet many (including me) find them a powerful way to ensure that your code is robust. Finally, Kotlin hides a lot of what goes on. In Java, you can trace through almost every step of program logic. This can be vital for hunting down bugs. In Kotlin, if you define a data class, then getters, setters, equality testing, to string, and hash code are added for you invisibly. This can be a bad idea.

Also according docs, 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

**Question:**  What are the advantages of Kotlin over Java?

**Answer:** Basically for me less thinking required to write kotlin equivalent to most java code:

data class

java: you have to write getters and setters for each thing, you have to write hashCode properly (or let IDE auto generate, which you have to do again every time you change the class), toString (same problem as hashcode) and equals (same problem as hashCode). or you could use lombok, but that comes with some quirky problems of its own. record types are hopefully on the way. \*kotlin: data class does it all for you.

getter and setter patterns

java: rewrite the getter and setter for each variable you use it for

kotlin: don't have to write getter and setter, and custom getter and setter take a lot less typing in kotlin if you do want to. also delegates exist for identical getters\setters

abstract vs open classes

java: you have to make an abstract class implementation

kotlin: open class lets you make an inheritable class while also being usable itself. nice mix of interface and regular class imo

extension functions

java: doesnt exist

kotlin: does exist, makes functions more clear in usage and feels more natural.

null

java: Anything but primitives can be null at any time.

kotlin: you get to decide what can and cant be null. allows for nice things like inline class

singleton

java: Memorize singleton pattern

kotlin: object instead of class

generics

java: Theyre alright, nothing fancy

kotlin: Reified generics (you can access the actual type), in and out for covariance

named parameters

java: does not exist, easy to break api back-compatibility if you arent careful.

kotlin: does exist, easy to preserve api back-compatiblity.

primary constructor

java: does not have per-se, you still have to define everything inside the class

kotlin: very nice to be able to quickly write a constructor without any constructor function or extra needless declarations

**Question:**  What is lateinit in Kotlin and when would you use it?

**Answer:** lateinit means late initialization. If you do not want to initialize a variable in the constructor instead you want to initialize it later on and if you can guarantee the initialization before using it, then declare that variable with lateinit keyword. It will not allocate memory until initialized. You cannot use lateinit for primitive type properties like Int, Long etc.

lateinit var test: String

fun doSomething() {

test = "Some value"

println("Length of string is "+test.length)

test = "change value"

}

There are a handful of use cases where this is extremely helpful, for example:

Android: variables that get initialized in lifecycle methods;

Using Dagger for DI: injected class variables are initialized outside and independently from the constructor;

Setup for unit tests: test environment variables are initialized in a @Before - annotated method;

Spring Boot annotations (eg. @Autowired).

**Question:**  What is a purpose of Companion Objects in Kotlin?

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

class EventManager {

companion object FirebaseManager {

}

}

val firebaseManager = EventManager.FirebaseManager

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

**Question:**  What is the Kotlin double-bang !! operator?

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

Consider:

fun main(args: Array<String>) {

var email: String?

email = null

println(email!!)

}

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

**Question:**  What is the difference between suspending vs. blocking?

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

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

**Question:**  What is the difference between List and Array types?

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

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

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

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

**Answer:** consts are compile time constants. Meaning that their value has to be assigned during compile time, unlike vals, where it can be done at runtime.

This means, that consts can never be assigned to a function or any class constructor, but only to a String or primitive.

For example:

const val foo = complexFunctionCall() //Not okay

val fooVal = complexFunctionCall() //Okay

const val bar = "Hello world" //Also okay

**Question:**  What is the difference between open and public in Kotlin?

**Answer:** The open keyword means “open for extension“. The open annotation on a class is the opposite of Java's final: it allows others to inherit from this class.

If you do not specify any visibility modifier, public is used by default, which means that your declarations will be visible everywhere. public is the default if nothing else is specified explicitly.

**Question:**  What is the equivalent of Java static methods in Kotlin?

**Answer:** Place the function in the companion object.

class Foo {

public static int a() { return 1; }

}

will become:

class Foo {

companion object {

fun a() : Int = 1

}

}

// to run

Foo.a();

Another way is to solve most of the needs for static functions with package-level functions. They are simply declared outside a class in a source code file. The package of a file can be specified at the beginning of a file with the package keyword. Under the hood these "top-level" or "package" functions are actually compiled into their own class. In the above example, the compiler would create a class FooPackage with all of the top-level properties and functions, and route all of your references to them appropriately.

Consider:

package foo

fun bar() = {}

usage:

import foo.bar

**Question:**  What is the purpose of Unit-returning in functions? Why is VALUE there? What is this VALUE?

**Answer:** Problem

Explain what is the purpose of Unit-returning in functions? Why is VALUE there? What is this VALUE?

fun printHello(name : String?) : Unit {

if (name != null)

print("Hello, $name!")

else

print("Hi there!")

// We don't need to write 'return Unit.VALUE' or 'return', although we could

}

Answer

The purpose is the same as C's or Java's void. Only Unit is a proper type, so it can be passed as a generic argument etc.

Why we don't call it "Void": because the word "void" means "nothing", and there's another type, Nothing, that means just "no value at all", i.e. the computation did not complete normally (looped forever or threw an exception). We could not afford the clash of meanings.

Why Unit has a value (i.e. is not the same as Nothing): because generic code can work smoothly then. If you pass Unit for a generic parameter T, the code written for any T will expect an object, and there must be an object, the sole value of Unit.

How to access that value of Unit: since it's a singleton object, just say Unit

UNIT actually contains valuable information, it basically just means "DONE". It just returns the information to the caller, that the method has been finished.

**Question:**  What will be result of the following code execution?

**Answer:** Problem

What will be the output?

val aVar by lazy {

println("I am computing this value")

"Hola"

}

fun main(args: Array<String>) {

println(aVar)

println(aVar)

}

Answer

For lazy the first time you access the Lazy property, the initialisation (lazy() function invocation) takes place. The second time, this value is remembered and returned:

I am computing this value

Hola

Hola

**Question:**  When to use lateinit over lazy initialization in Kotlin?

**Answer:** There are some simple rules to determined if you should use one or the other for properties initialisation:

* If properties are mutable (i.e. might change at a later stage) use **lateInit**
* If properties are set externally (e.g. need to pass in some external variable to set it), use **lateinit**. There’s still workaround to use lazy but not as direct.
* If they are only meant to initialized once and shared by all, and it’s more internally set (dependent on variable internal to the class), then use **lazy**. Tactically, you could still use lateinit, but using**lazy** would better encapsulate your initialization code.

Also compare:

| **lateinit var** | **by lazy** |
| --- | --- |
| Can be initialized from anywhere the object seen from. | Can only be initialized from the initializer lambda. |
| Multiple initialization possible. | Only initialize single time. |
| Non-thread safe. It’s up to user to initialize correctly in a multi-threaded environment. | Thread-safety by default and guarntees that the initializer is invoked by once. |
| Can only be used for var. | Can only be used for val. |
| Not eligible for nonnull properties. | Not eligible for nonnull properties. |
| An isInitialized method added to check whether the value has been initialized before. | Property never able to un-initialized. |
| Not allowed on properties of primitive types. | Allowed on properties of primitive types. |

**Question:**  When would you use Elvis operator in Kotlin?

**Answer:** The Elvis operator is part of many programming languages, e.g. Kotlin but also Groovy or C#. The Elvis operator is the ternary operator with its second operand omitted.

x ?: y // yields `x` if `x` is not null, `y` otherwise.

If x isn't null, then it will be returned. If it is null, then the y will be returned.

**Question:**  Why kotlin is better than Java?

**Answer:** Kotlin is a simple general-purpose language, in which there is a code redundancy when compared to java. There are many features in kotlin like null safety, extension functions, primary constructors, inline or lambda expressions, properties, and type interference for the properties and variables.

**Question:**  Explain functions in kotlin?

**Answer:** Functions used in kotlin are simple to store in data structures and variables and can possess arguments that are passed from the high ordered data functions.

An example of the sample function declaration in kotlin:

function double(y:int):

Int

{

return 3 \* y

}

val reslt = double(3)

**Question:**  What are the high order functions?

**Answer:** High order functions consider functions as a parameter and produce a function.

**Question:**  How to convert a String into an int in the kotlin?

**Answer:** To convert a string value to the string value to it in kotlin we use point() method.

Let us see an example:

function main(args: array)

{

val s: str]ng = "Kotlin"

var y = 10

y = "10".toint()

}

**Question:**  How to declare a variable in kotlin?

**Answer:** Declaration of a variable in kotlin is done by Val and var followed by data type.

val s: String = "Hello"

var Z = 5

**Question:**  What is the difference between Val and var in the declaration of variables?

**Answer:** For the variables in Val declaration, we can not reassign the variables, whereas in var declaration, the variables can be reassigned but the value reassigned should be the same type.

Function main(args: array<string>)

{

val s: sting = "Hello"

var z = 5

z = "6".

}

**Question:**  List out the basic data types we use in kotlin?

**Answer:** The data type of a constant or a variable decides what is the type of a variable and how much space needs to store. Types of data that a string allow are:

Strings

Numbers

Arrays

Booleans

Characteristics

**Question:**  What is Null safety?

**Answer:** Null safety feature helps with the exception of null pointers in the expressions. It helps mostly in finding nullable and non-nullable references. We can thread null pointers by using nullable types like int? float? String?

var int: int? = " sourcepoint.com"

var newint = int?: "default value"

int = nill

new int =int?: "default value"

**Question:**  What are the structural expressions in kotlin?

**Answer:** The three important structural expressions in kotlin are:

Break: break expression helps to break the closest enclosing loop

Return: This expression helps to return from the closest functions or default functions.

Continue: This expression helps to proceed for the next loop.

**Question:**  Is there any chance to shift the code from java to kotlin?

**Answer:** We can shift the code which is used in java to koltin by using JetBrains Idea.

**Question:**  What are the available modifiers in kotlin?

**Answer:** Modifiers are the developers to personalize the declarations. There are four modifiers available in the kotlin, they are:

Public: By default, all the declarations are visible everywhere

Internal: This modifier helps to use declarations anywhere in the module.

Private: By these modifiers, the declarations are contained only in the file which is restricted.

Protected: In this, the declarations are secured, and not available for top-level declarations.

**Question:**  What type of programming does a kotlin accept?

**Answer:** Kotlin accept two types of programmings, they are:

Object-oriented programming

Procedural programming

Functional

**Question:**  How many types of strings are available in kotlin?

**Answer:** A collection of characteristics together is known as a string. There are two types of strings available in the kotlin:

Raw string

Escaped string

**Question:**  What is mean by string interpolation?

**Answer:** We can evaluate string templates in kotlin string. This template creation is known as string interpolation.

**Question:**  What we use in replacement to switch statements in kotlin?

**Answer:** We use when in kotlin in place of the switch in java

When(x)

{

case1: code executed if x=1:

break:

case2: code executed fi x=2:

break;

Default code executed if x does not match any case

}

**Question:**  Is New a keyword in kotlin?

**Answer:** “New” is not a keyword in kotlin. We can initiate a new class as follow:

class x

var x = x()

val new = x()

**Question:**  What is mean by init block?

**Answer:** Init is a login block and it is executed in the primary constructor and initialized. If you want to revoke in the secondary constructor then it starts working after the primary constructor in the chain form.

**Question:**  Is inheritance compile in Kotlin?

**Answer:** Formal inheritance structure does not compile in the kotlin. By using an open modifier we can finalize classes.

open class B

{

}

class c = B()

{

}

**Question:**  What is the Ranges Operator?

**Answer:** The Ranges operator helps to iterate a loop with range.

Example:

for(i 1...15)

print (i)

**Question:**  How to compare two strings in kotlin?

**Answer:** We can compare two strings in two ways,

Using”==” operator”.

Using compare() function

**Question:**  What are the class members in kotlin?

**Answer:** A class in kotlin have the following members:

Initializer blocks

Properties

Open declarations

Nested classes

Inner classes

**Question:**  List out some of the extension methods in kotlin?

**Answer:** Some of the extension methods are:

read Text(): Helps to read content in the files to a single string.

buffer Reader(): It is used to read the contents of the file to buffer the reader

read each line(): It reads each line by line in the file

readlines(): It helps to read lines of file for listing

**Question:**  How to handle a null operator?

**Answer:** By using the Elvis operator we can handle all the nullable values.

**Question:**  What is Kotlin?

**Answer:** Kotlin is a statically-typed programming language which runs on the JVM. It can be compiled either using Java source code and LLVM compiler.

**Question:**  Who is the developer of Kotlin?

**Answer:** Kotlin was developed by JetBrains.

**Question:**  Why you should switch to Kotlin from Java?

**Answer:** Kotlin language is quite simple compared to Java. It reduces may redundancies in code as compared to Java. Kotlin can offer some useful features which are not supported by Java.

**Question:**  Tell three most important benefits of using Kotlin?

**Answer:** Kotlin language is easy to learn as its syntax is similar to Java.

Kotlin is a functional language and based on JVM. So, it removes lots of boiler plate

It is an expressive language which makes code readable and understandable.

**Question:**  Explain the use of extension functions

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

**Question:**  What does ‘Null Safety’ mean in Kotlin?

**Answer:** Null Safety feature allows removing the risk of occurrence of NullPointerException in real time. It is also possible to differentiate between nullable references and non-nullable references.

**Question:**  Why is Kotlin interoperable with Java?

**Answer:** Kotlin is interoperable with Java because it uses JVM bytecode. Compiling it directly to bytecode helps to achieve faster compile time and makes no difference between Java and Kotlin for JVM.

**Question:**  Is there any Ternary Conditional Operator in Kotlin like in Java?

**Answer:** No there is no ternary conditional operator in Kotlin language.

**Question:**  How can you declare a variable in Kotlin?

**Answer:** value my\_var: Char

**Question:**  How many constructors are available in Kotlin?

**Answer:** Two types of constructors available in Kotlin are:

Primary constructor

Secondary constructor

**Question:**  Can you tell me what kinds of programming types does Kotlin support?

**Answer:** Procedural Programming

OOPS

**Question:**  Give me name of the extension methods Kotlin provides to java.io.File

**Answer:** bufferedReader(): Use for reading contents of a file into BufferedReader

readBytes() : Use for reading contents of file to ByteArray

readText(): Use of reading contents of file to a single String

forEachLine() : Use for reading a file line by line in Kotlin

readLines(): Use to reading lines in file to List

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

**Answer:** Elvis Operator is used for handling null expectations in Kotlin.

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

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

Null Safety

Operator Overloading

Coroutines

Range expressions

Smart casts

Companion Objects

**Question:**  Explain the use of data class in Kotlin?

**Answer:** Data class holds the basic data types. It does not contain any functionality.

**Question:**  Can we migrate code from Java to Kotlin?

**Answer:** Yes, JetBrains IDEA provides an inbuilt tool to migrate code from java to Kotlin.

**Question:**  Does Kotlin allow macros?

**Answer:** No. Kotlin does not offer support for macros because the developers of Kotlin find it difficult to include it in the language.

**Question:**  Tell me the default behavior of Kotlin classes?

**Answer:** In Kotlin all classes are final by default. That’s because Kotlin allows multiple inheritances for classes, and an open class is more expensive than a final class.

**Question:**  Does Kotlin support primitive Datatypes?

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

**Question:**  What is Ranges operator in Kotlin?

**Answer:** Ranges operator helps to iterate through a range. Its operator form is (..) For Example

for (i in 1..15)

print(i)

It will print from 1 to 15 in output.

**Question:**  Can Kotline offer any additional functionality for standard Java packages or standard Java classes?

**Answer:** Kotlin programs can run on standard JVM like any another compiled Java code. It allows JVM to compile any program to byte-code. It is accessible using Java Virtual Machine. Therefore, Kotlin is almost similar to Java. Moreover, Kotlin applications can be built with parts of Java code.

**Question:**  Give a syntax for declaring a variable as volatile in Kotlin?

**Answer:** Volatile var x: Long? = null

**Question:**  What is the use of abstraction in Kotlin?

**Answer:** Abstraction is the most important concept of Objected Oriented Programming. In Kotlin, abstraction class is used when you know what functionalities a class should have. But you are not aware of how the functionality is implemented or if the functionality can be implemented using different methods.

**Question:**  How to compare two strings in Kotlin?

**Answer:** Compares string in Kotlin are possible in the following ways:

Using “==” operator:

You can use ah operator for comparison of two string. In Kotlin == operator is used.

Using compareTo() extension function

Syntax of compareTo() function is given below :

fun String.compareTo(

other: String,

ignoreCase: Boolean = false

): Int

Another code example

fun main(args: Array & lt; String & gt;) {

val x: String = "Kotlin is simple"

val y: String = "Kotlin language is" + " easy"

if (x == y) {

println(" x and y are similar.")

} else {

println(" x and y are not similar.")

}

}

**Question:**  What does this code do?

**Answer:** bar {

System.out.println("TestKotlin!")

}

The code passes lambda function that prints “TestKotlin!” as an argument into function bar()

**Question:**  What is Kotlin, and what are its key features?

**Answer:** Kotlin is a modern, statically typed programming language that runs on the JVM. Its key features include null safety, support for functional programming, and Java compatibility.

Here’s “Hello, world!” in Kotlin:

fun main(args : Array<String>) {

println("Hello, World!")

}

Kotlin offers several advantages over Java, including greater flexibility, readability, and more concise syntax. Additionally, Kotlin supports modern programming paradigms such as functional programming, which Java doesn’t fully support.

**Question:**  How does Kotlin compare to other languages such as Java?

**Answer:** Here’s “Hello, world!” in Java:

class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, World!");

}

}

Some of the most common libraries and frameworks for Kotlin include Anko, RxJava, and Ktor. Each of these libraries provides important functionality for building complex applications in Kotlin.

**Question:**  What is null safety in Kotlin, and why is it important?

**Answer:** Null safety is a feature of Kotlin that ensures that variables cannot be assigned a null value. This is important because it helps avoid NullPointerException errors, which can be difficult to debug.

**Question:**  What are some of the basic syntax rules for Kotlin?

**Answer:** Some of the basic syntax rules for Kotlin include the use of semicolons to terminate statements, the use of curly braces to delimit blocks of code, and the use of the val keyword to declare immutable variables.

**Question:**  What is the difference between a var and a val in Kotlin?

**Answer:** A var is a mutable variable, meaning its value can be changed. A val is an immutable variable, which means that its value can not be changed. Think of them as variables and constants.

**Question:**  What is the difference between an interface and an abstract class in Kotlin?

**Answer:** An interface is a type that specifies a set of abstract methods that must be implemented by any class that implements the interface. An abstract class is a type that defines both abstract and concrete methods, which subclasses can inherit.

**Question:**  What is the difference between a data class and a regular class in Kotlin?

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

**Question:**  What is a lambda expression in Kotlin?

**Answer:** A lambda expression is an anonymous function that can concisely represent a function with a single parameter. Lambda expressions are often used in conjunction with higher-order functions, such as map and filter.

**Question:**  What is the difference between a function and a method in Kotlin?

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

**Question:**  What is the difference between a class and an object in Kotlin?

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

**Question:**  What is the difference between a constructor and an initializer in Kotlin?

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

**Question:**  How do you declare a default argument in Kotlin?

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

**Question:**  How do you define an object in Kotlin?

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

val newObject= object {

val one = "Hello"

val two = "World"

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

}

The above would print “Hello World.”

**Question:**  How do you declare a function in Kotlin?

**Answer:** To declare a function in Kotlin, use the fun keyword followed by the name of the function and the parameters that it accepts. You can then define the function body within curly braces.

**Question:**  How do you invoke a function in Kotlin?

**Answer:** To invoke a function in Kotlin, simply use its name followed by the parentheses operator and any necessary arguments. This will cause the function to execute with the specified arguments.

**Question:**  What is a higher-order function in Kotlin?

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

**Question:**  What is the difference between a suspend function and a regular function?

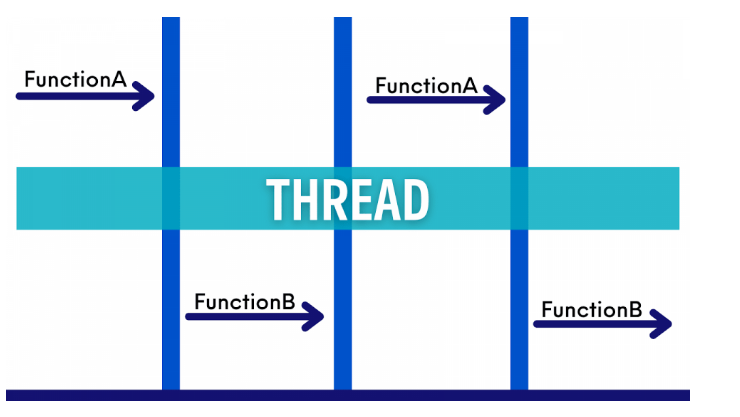
**Answer:** A suspend function can be suspended, meaning that you can pause its execution and resume it at a later time. A regular function cannot be suspended and will always execute to completion.

**Question:**  What are coroutines in Kotlin?

**Answer:** Coroutines are Kotlin features that allow for concurrent or parallel execution of code. Coroutines can improve application performance by taking advantage of multiple cores in a processor.

**Question:**  What is the difference between a thread and a coroutine?

**Answer:** A thread is an execution unit that can run independently from other threads. A coroutine is a unit of execution that can be suspended and resumed, allowing it to share resources with other concurrent or parallel executions.



**Question:**  How do you launch a coroutine in Kotlin?

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

**Question:**  How do you cancel a coroutine in Kotlin?

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

**Question:**  How do you pass data between coroutines in Kotlin?

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

**Question:**  How do you create a lambda expression in Kotlin?

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

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

items.fold(0, {

acc: Int, i: Int ->

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

val result = acc + i

println("result = $result")

result

})

To pass a lambda expression as an argument to a function, include it within parentheses after the function name and any necessary arguments. This will cause the code within that lambda expression to be executed whenever the function is called.25. How do you pass a lambda expression as an argument in Kotlin?

**Question:**  What is the difference between a lambda expression and an anonymous function?

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

**Question:**  What are inline functions in Kotlin?

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

**Question:**  What is the difference between an extension function and a regular function?

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

**Question:**  What is type inference in Kotlin?

**Answer:** Type inference is the process of automatically determining the type of a variable or expression based on its value. In Kotlin, type inference determines the type of variables when they are first declared and the return type of functions.

**Question:**  What are reified types in Kotlin?

**Answer:** Reified types can be accessed at runtime rather than just at compile time. In Kotlin, reified types provide metadata about a type at run time, such as its name or the names of its members. They are also commonly used in reflection and generic programming. It is an advanced feature not commonly used by beginners to the Kotlin language.

**Question:**  How do you create a compiler plugin in Kotlin?

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

**Question:**  What is the kotlin c compiler?

**Answer:** Thekotlinccompiler is a command-line tool that can compile Kotlin source code into JVM bytecode or JavaScript. You can also use it to run Kotlin code directly without compiling it first.

**Question:**  What are the major limitations of Kotlin?

**Answer:** The major limitations of Kotlin are its lack of support for operator overloading and variable arguments. Additionally, Kotlin does not have any built-in string interpolation or formatting features.

**Question:**  When wouldn't you want to use Kotlin?

**Answer:** There are a few scenarios where you might not want to use Kotlin. If you need to write code compatible with Java 6 or earlier, Kotlin will not be an option. Additionally, if you are targeting the Android platform, you will need to use Java 8 or higher to use Kotlin. Finally, if you are working on a project with a large codebase, Kotlin’s slow compile times makes it a poor choice.

**Question:**  How do you sort a list in Kotlin?

**Answer:** There are two main ways to sort a list in Kotlin. The first is via the sort method, which sorts the list in place based on natural ordering. The second is the sortedBy function, which sorts the list in place and returns a new sorted version.

val numbers = listOf("one", "two", "three", "four")

println("Sorted ascending: ${numbers.sorted()}")

println("Sorted descending: ${numbers.sortedDescending()}")

There are a few different ways to find an item or items within a list in Kotlin. You can use the find function or the filter function. You can also index items directly with indexOf or lastIndexOf, but this is less useful. Why? The entire point is discovering the index associated with a given value.

**Question:**  How do you find an item or items within a list in Kotlin?

**Answer:** var colors: List<String> = listOf("Red", "Orange","Yellow","Blue","Green","Indigo","Violet")

val favoriteColor = rainbow.find { color -> "Blue".equals(color) }

You can remove items from a list in Kotlin in a few different ways. The simplest is the filter method, which returns all elements that satisfy a given predicate except for those that match the specified element. Additionally, you can use the partition function, which splits the provided list into two new lists: one that matches the element and one that does not. Finally, just use the remove function like so:

**Question:**  How do you remove an item or items from a list in Kotlin?

**Answer:** list.remove([index])

**Question:**  How do you generate a sequence of data in Kotlin?

**Answer:** You can generate number sequences or data in a few different ways. The simplest is to use the range function, which returns a sliding window of integers based on a starting and ending value. Alternatively, you can use the generate Sequence function, which allows for a more complex generation via a seed value and function that produces the next element in the sequence.

**Question:**  How do you group data by key in Kotlin?

**Answer:** To group data by key in Kotlin, you must first convert the data into a map. From there, you can use the groupBy function to group all of the data in the map by key. Alternatively, you can use the mapAndFlattenToList function to return a flat list of values grouped by key.

**Question:**  Can you use IntArray and Array<Int> interchangeably in Kotlin?

**Answer:** In most cases, you can use IntArray and Array<Int> interchangeably. This could lead you to believe that they are interchangeable, which is not the case — and using them as such is bad practice.

IntArray is an int[] type, whereas Array<Int> is an Integer[] type. They will be treated differently during calls such as Integer.valueOf() and you will receive a type mismatch error.

**Question:**  Can We Execute Kotlin Code Without JVM?

**Answer:** Yes, Kotlin can be compiled to JavaScript and run in a web browser. Additionally, various tools and frameworks allow the execution of Kotlin code without requiring a JVM, such as the Ktor framework.

**Question:**  What Kinds of Programming Types Does Kotlin Support?

**Answer:** Kotlin supports both object-oriented and functional programming paradigms. Additionally, it has support for metaprogramming, which allows for the generation of code at compile time.

**Question:**  What is the Use of Kotlin in Android?

**Answer:** Kotlin is a statically typed programming language designed specifically for the Java Virtual Machine (JVM). As such, you can use it to write Android applications, as it provides several benefits over Java, including enhanced readability and increased code safety. Its popularity for Android applications prompts an entire category for Kotlin interview questions Android developers should know.

The most popular Kotlin frameworks and tools include Ktor and the Kotlin Android Extensions library.

**Question:**  Is Kotlin Frontend or Backend?

**Answer:** Kotlin is a backend programming and frontend programming language, although it works a little differently depending on how it’s being used. Frontend programming leans on Kotlin/JavaScript, whereas backend programming runs on the JVM.

**Question:**  Why did you switch to Kotlin from Java ?

**Answer:** Kotlin seems to be simpler and cleaner than Java. It removes a lot of redundancies in code from Java. Kotlin also adds some needed features that Java doesn’t yet support, and is making code more idiomatic. Also Kotlin has been added to Android Studio’s list of supported languages recently. So, there is much to expect from Kotlin in easing out the development efforts and good support in future.

**Question:**  What are the features you think are there in Kotlin but not in Java ?

**Answer:** Kotlin has quite a number of features that Java doesn’t. To name some of them, they are

Extension Functions

Null Safety

Smart casts

Range expressions

Operator Overloading

Data classes

Companion Objects

Coroutines

etc.

**Question:**  What kinds of programming does Kotlin support ?

**Answer:** Kotlin supports two types of programming. They are

Procedural Programming

Object Oriented Programming

**Question:**  What is the entry point to a Kotlin program ? Provide an example.

**Answer:** Like most of the other procedural languages, main() function is the entry point to a Kotlin program.

An Example for main() function is

Kotlin Program – example.kt

fun main(args: Array<String>) {

val user1 = User(name="Yogi", age=27)

printUser(user1)

}

fun printUser(user: User){

println(user)

}

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

**Question:**  How do you think extension functions are useful ? – Kotlin Interview Question

**Answer:** Extension functions helps to extend a class with new functionality without having to inherit from the class. Also you may use them like an inbuilt function for the class throughout the application.

**Question:**  What are Data classes ? Aren’t they available in Java ? – Kotlin Interview Question

**Answer:** Sometimes we use a class just to hold the data and nothing else. These classes are called Data classes. Of course these kind of classes could be built using Java, but with explicit implementation of getter and setter for each of the properties of class. Also you may need to implement functions like equals, toString and copy separately. What Kotlin does is implementing all these automatically along with special functions called component functions. How cool is that, removing the redundant code bloat.

**Question:**  Does Kotlin provide any additional functionalities for standard Java packages or standard Java classes? – Kotlin Interview Question

**Answer:** Ofcourse, Yes. Kotlin uses the concept of extension functions, that we already talked about, to build some useful and more widely used functions among developers directly into the Kotlin library.

**Question:**  Hmm! Where does this Kotlin run ? Does it have some kind of different runtime environment ?

**Answer:** Once compiled, Kotlin programs can run on standard JVM like some other compiled Java code. This means that Kotlin Compiler compiles Kotlin programs to byte-code, which is understood by JVM. So, Kotlin is like a flavor of Java, that goes alongside Java. Interesting fact is that, Kotlin applications can be built with parts of Java code.

**Question:**  So, how do you migrate the code from Java to Kotlin ? – Kotlin Interview Question

**Answer:** JetBrains IDEA provides inbuilt tools to convert Java code to Kotlin code. Then you may do the magic offered by Kotlin at some of the parts in code, to make it clean.

**Question:**  OK. Is there something called init block in Kotlin ?

**Answer:** Yes

**Question:**  What does init block do and Where does it appear in a class ? – Kotlin Interview Question

**Answer:** Instructions in the init block are executed right after Primary Constructor’s execution. init block goes in a class along with secondary constructors as a method.

**Question:**  How many types of constructors are there ? What are they ?

**Answer:** There are two types of constructors. They are Primary Constructors and Secondary Constructors.

**Question:**  How are Primary Constructors different from Secondary Constructors ?

**Answer:** Primary Constructors are declared intrinsically with class definition. Secondary Constructors are declared exclusively inside the class body.

In the following example, in the first line, the constructor keyword along with the variables declared right after it is the Primary Constructor. Inside the class body, we have another constructor, and this is Secondary Constructor.

Kotlin Program – example.kt

class Person constructor(var name: String, var age: Int){

var profession: String = "Not Mentioned"

constructor (name: String, age: Int, profession: String): this(name,age){

this.profession = profession

}

}

**Question:**  Is there any dependency of Secondary Constructors on Primary Constructors ?

**Answer:** Yes. Secondary Constructor has to make an exclusive call to Primary Constructor or other Secondary Constructor, which of course calls the Primary Constructor. Following is an example, and here the Secondary Constructor makes call to Primary Constructor using this(name, age).

Kotlin Program – example.kt

class Person constructor(var name: String, var age: Int){

var profession: String = "Not Mentioned"

constructor (name: String, age: Int, profession: String): this(name,age){

this.profession = profession

}

fun printPersonDetails(){

println("$name whose profession is $profession, is $age years old.")

}

}

**Question:**  What is the difference between val and var ? – Kotlin Interview Question

**Answer:** Val (Value) is like a constant. Once assigned a value, you cannot change it. On the other hand Var (Variable) is designed to be a storage location that can accept reassignment of values of same data type or what ever feasible by the data type casting.

**Question:**  What is Kotlin’s Null Safety ? – Kotlin Interview Question

**Answer:** Null Safety in Kotlin is to eliminate the risk of occurrence of NullPointerException in real time. Kotlin can differentiate between nullable references and non-nullable references. If a variable has to be allowed to store a null value, that has to be declared with a null (?) operator.

**Question:**  If you have worked with files, name some of the extension methods Kotlin provides to java.io.File

**Answer:** Kotlin provides very useful extension functions to java.io.File. Some of them are :

File.bufferedReader() : to read contents of a file into BufferedReader

File.forEachLine() : to read a file line by line in Kotlin

File.inputStream() : to read contents of file to InputStream

File.readBytes() : to read contents of file to ByteArray

File.readLines() : to read lines in file to List

File.readText() : to read contents of file to a single String

**Question:**  Is there Ternary Conditional Operator in Kotlin like in Java ?

**Answer:** No

**Question:**  How do you realize Ternary Conditional Operator in Kotlin ?

**Answer:** A simple if else should do the job.

if (condition) a else b

**Question:**  How do you declare a variable as volatile in Kotlin ? – Kotlin Interview Question

**Answer:** By providing volatile annotation before the declaration of variable.

@Volatile var a: Long? = null

**Question:**  How do you check if two Strings are equal valued ? – Kotlin Interview Question

**Answer:** Using == (double equal to) operator.

Kotlin Program – example.kt

fun main(args: Array<String>) {

val a: String = "kotlin is easy"

val b: String = "kotlin is" + " easy"

if(a==b){

println(" a and b are equal.")

} else {

println(" a and b are not equal.")

}

}

**Question:**  Features Of Kotlin

**Answer:** Kotlin comes in with several features that have made the lives of developers easier. Highlighting a few of the eye-catching features of Kotlin which will help you understand this language better-

Compact Code- Kotlin is known as an OOPs-based programming language with fewer code lines compared to Java. This feature makes it a perfect choice for software developers.

Open Source- Kotlin combines the attributes of OOPs and functional programming. It is known as an open-source for Android and uses the JVM.

Easy to understand- The coding in Kotlin is simple, which makes it readable and understandable. Not to forget, it also helps improve the performance of Android Development.

Many extensions- This code can support varied extension functions without modifying the code. It helps in improvising the existing code and making it more efficient.

Easy to combine- One can quickly jump from Java Code to Kotlin Code without thinking much as both the codes can be used together. In short, if you have a handful of knowledge of OOPs programming, then switching to Kotlin development shouldn’t be a big deal for you.

Cost-efficient- Kotlin supports the highly efficient and cost-effective technique known as Smart Cast. This technique helps in reducing the cost of an application and makes the programming more efficient.

Easy to learn- Kotlin brings in easy code language, which eases down the whole process and makes the overall experience easier and more efficient. This code is simple to learn for developers in comparison to other codes.

**Question:**  What are the advantages of Kotlin over Java?

**Answer:** Easy Process- In Java, one has to create getters and setters for each object. Not only that, one is required to write hashCode and equals. However, In Kotlin Programming Language, everything is taken care of by classes- making the process easier.

Rewriting the method- If you use Java, then for each variable, it is essential to rewrite/copy the getter and setter method, which is time-consuming. In Kotlin, the rewrite bit takes a lot less time.

Support of Extension- In Java, one can’t avail of extension functions. However, in Kotlin, there is strong support for extension functions making the overall process and code language clear.

No/Fewer Errors- In Kotlin, the chances of errors are less than in Java.

**Question:**  What’s the Target Platform of Kotlin? How is Kotlin-Java interoperability possible?

**Answer:** Java Virtual Machine is known as the Target Platform of Kotlin. Yes, Kotlin-Java interoperability is possible as both produce bytecode on the compilation. Due to this reason, it’s possible to call Kotlin code from Java and vice-versa.

**Question:**  How do you differentiate val and var declaration?

**Answer:** The variables of val can’t be changed. They are considered final modifiers in Java. On the other hand, var can be reassigned with the same data type value.

**Question:**  Is it possible to convert a String to an Int?

**Answer:** Yes, with the toInt() method, it’s possible to convert a String to an Int.

**Question:**  What’s the Elvis Operator?

**Answer:** Elvis Operator is considered as the safeguard of the value. It helps to unwrap the value from the Nullable safely. It can be identified as ?: over the nullable type.

**Question:**  What is Null Safety?

**Answer:** Null Safety refers to initializing the ‘null’ value so that the variable contains the null for the possible execution. With Null Safety, all the runtime null-dereference errors will be reflected in compile time.

**Question:**  What are Nullable Types in Kotlin?

**Answer:** Here are the following Nullable Types in Kotlin- String?, Int?, Float?.

**Question:**  Can we use primitive types like int, float, and double in Kotlin?

**Answer:** One can’t use primitive types at the entry-level. But one can experience it at compiled JVM bytecode.

**Question:**  What’s the entry point of the Kotlin Program?

**Answer:** The primary function is considered the entry point of the Kotlin Program. In this programming language, one can choose to ignore the central part of the class. Later, the strings passed in Array<String> command code are used to reclaim the command line arguments.

**Question:**  How is const. not similar to a val?

**Answer:** It is impossible to use const with a var or as an individual. Const is not permitted on a local variable. On the other hand, val properties are defined on a runtime. Adding const modifier to val can make a compile-time constant.

**Question:**  How do you define that a function has been declared?

**Answer:** A function’s return type is signified after the :

For eg-

fun sumOf(a: Int, b: Int): Int{return a + b}

**Question:**  How do you differentiate between == and === operators in Kotlin?

**Answer:** == is used to identify if the values are equal or not. === is used to check if the reference is equal or not.

**Question:**  What are the visibility modifiers identified in Kotlin?

**Answer:** The Visibility Modifiers in Kotlin are- Public, Internal, Private, and Protected.

**Question:**  What is the default visibility modifier?

**Answer:** The public is known as the default visibility modifier.

**Question:**  Define the init block in Kotlin?

**Answer:** Init is identified as the initializer block in Kotlin. It can be executed only once the primary constructor has been instantiated. However, if someone invokes a secondary constructor, it will start working after the primary one following the chain.

**Question:**  How does String Interpolation work in Kotlin?

**Answer:** String Interpolation is used to evaluate string templates. One uses the symbol $ to insert variables inside a String.

**Question:**  Identify the type of arguments inside a constructor?

**Answer:** The constructor arguments are by default set as val and var.

**Question:**  Do you identify ‘new’ as a keyword in Kotlin?

**Answer:** No, there is no word as ‘new’ in Kotlin. However, one can identify the same in Java.

**Question:**  How can we instantiate a class object in Kotlin?

**Answer:** Here’s a quick example of how we can set it in-

class A

var a = A()

Val new = A()

**Question:**  What do you understand about Companion Objects in Kotlin?

**Answer:** While in other programming languages like Java, a static keyword is used to declare the class members/name. Kotlin doesn’t consist of a static keyword, i.e., to achieve the functionality of static member functions, a companion object is used. Companion Object is also known as Object Extension

**Question:**  What’s the difference between Open and Public keywords in Kotlin?

**Answer:** Open resembles Open For Expansion. While in Java, methods can be overridden by default. In Kotlin, it’s essential to use an available form, which can be overridden but cannot be done by default.

If in Kotlin no visibility modifier is specified, one prefers to use public by default. This means that the declarations will be accessible everywhere inside the program.

**Question:**  What do you understand by the “when” keyword in the context of Kotlin?

**Answer:** The “when” keyword refers to substitutes. It is used to substitute the switch operator in other languages like Java. Once a particular condition is met, a specific block of code must run.

Inside this expression, it compares all the branches one by one. The target is to identify a match. Once the first match is discovered, it heads to the conclusion of the when block and focuses on executing the code immediately.

**Question:**  How do you differentiate lazy from lateinit?

**Answer:** Both of the attributes are known for delaying the property initialization in Kotlin.

Lateinit is considered as a modifier used with var. It is used to set the value to the var at a later point.

The lazy method or lambda expression is developed on a val only. The value would be created at runtime as and when required.

**Question:**  Does Kotlin have the static keyword?

**Answer:** No, in Kotlin, you can’t find the static keyword. To create a static keyword, one has to use the companion object.

**Question:**  What is a primary constructor in Kotlin?

**Answer:** The primary constructor is part of the class header. Unlike Java, you don't need to declare a constructor in the body of the class. Here's an example:

class Person(val firstName: String, var age: Int) {

// class body

}

The main idea is by removing the constructor keyword, our code gets simplified and easy to understand.

**Question:**  What is a data class in Kotlin?

**Answer:** We frequently create classes whose main purpose is to hold data. In Kotlin, this is called a data class and is marked as data:

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

To ensure consistency and meaningful behavior of the generated code, data classes have to fulfill the following requirements:

The primary constructor needs to have at least one parameter;

All primary constructor parameters need to be marked as val or var;

Data classes cannot be abstract, open, sealed or inner;

**Question:**  Explain the Null safety in Kotlin

**Answer:** Kotlin's type system is aimed at eliminating the danger of null references from code, also known as the The Billion Dollar Mistake.

One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference will result in a null reference exception. In Java this would be the equivalent of a NullPointerException or NPE for short.

In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that can not (non-null references). For example, a regular variable of type String can not hold null:

var a: String = "abc"

a = null // compilation error

To allow nulls, we can declare a variable as nullable string, written String?:

var b: String? = "abc"

b = null // ok

print(b)

**Question:**  How are extensions resolved in Kotlin and what doest it mean?

**Answer:** Extensions do not actually modify classes they extend. By defining an extension, you do not insert new members into a class, but merely make new functions callable with the dot-notation on variables of this type.

The extension functions dispatched statically. That means the extension function which will be called is determined by the type of the expression on which the function is invoked, not by the type of the result of evaluating that expression at runtime. In short, they are not virtual by receiver type.

Consider:

open class BaseClass

class DerivedClass : BaseClass()

fun BaseClass.someMethod(){

print("BaseClass.someMethod")

}

fun DerivedClass.someMethod(){

print("DerivedClass.someMethod")

}

fun printMessage(base : BaseClass){

base.someMethod()

}

printMessage(DerivedClass())

This will print

BaseClass.someMethod

because the extension function being called depends only on the declared type of the parameter base in printMessage method, which is the BaseClass class. This is different from runtime polymorphism as here it is resolved statically but not at the runtime.

**Question:**  What is a purpose of Companion Objects in Kotlin?

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

class EventManager {

companion object FirebaseManager {

}

}

val firebaseManager = EventManager.FirebaseManager

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

**Question:**  What is Lateinit in Kotlin and when would you use it?

**Answer:** lateinit means late initialization. If you do not want to initialize a variable in the constructor instead you want to initialize it later on and if you can guarantee the initialization before using it, then declare that variable with lateinit keyword. It will not allocate memory until initialized. You cannot use lateinit for primitive type properties like Int, Long etc.

lateinit var test: String

fun doSomething() {

test = "Some value"

println("Length of string is "+test.length)

test = "change value"

}

There are a handful of use cases where this is extremely helpful, for example:

Android: variables that get initialized in lifecycle methods;

Using Dagger for DI: injected class variables are initialized outside and independently from the constructor;

Setup for unit tests: test environment variables are initialized in a @Before - annotated method;

Spring Boot annotations (eg. @Autowired).

**Question:**  Explain lazy initialization in Kotlin

**Answer:** lazy means lazy initialization. Your variable will not be initialized unless you use that variable in your code. It will be initialized only once after that we always use the same value.

lazy() is a function that takes a lambda and returns an instance of lazy which can serve as a delegate for implementing a lazy property: the first call to get() executes the lambda passed to lazy() and remembers the result, subsequent calls to get() simply return the remembered result.

val test: String by lazy {

val testString = "some value"

}

**Question:**  When to use lateinit over lazy initialization in Kotlin?

**Answer:** There are some simple rules to determined if you should use one or the other for properties initialisation:

* If properties are mutable (i.e. might change at a later stage) use **lateInit**
* If properties are set externally (e.g. need to pass in some external variable to set it), use **lateinit**. There’s still workaround to use lazy but not as direct.
* If they are only meant to initialized once and shared by all, and it’s more internally set (dependent on variable internal to the class), then use **lazy**. Tactically, you could still use lateinit, but using\*\* lazy\*\* would better encapsulate your initialization code.

Also compare:

|  |  |
| --- | --- |
| **lateinit** var | by **lazy** |
| Can be initialized from anywhere the object seen from. | Can only be initialized from the initializer lambda. |
| Multiple initialization possible. | Only initialize single time. |
| Non-thread safe. It’s up to user to initialize correctly in a multi-threaded environment. | Thread-safety by default and guarntees that the initializer is invoked by once. |
| Can only be used for var. | Can only be used for val. |
| Not eligible for nonnull properties. | Not eligible for nonnull properties. |
| An isInitialized method added to check whether the value has been initialized before. | Property never able to un-initialized. |
| Not allowed on properties of primitive types. | Allowed on properties of primitive types. |

**Question:**  May you briefly compare Kotlin vs Java?

**Answer:**

| Java vs Kotlin | Java | Kotlin |
| --- | --- | --- |
| Null Safe | In Java, NullPointerExceptions causes huge frustration for developers. It allows users to assign null to any variables but while accessing an object reference having null value raises a null pointer exception which user needs to handle. | In Kotlin, By default, all types of variables are non-null able (i.e. we can’t assign null values to any type of variables/objects). If we try to assign or return null values, Kotlin code will fail during compile-time. If we really want a variable to have a null value, we can declare as follows: value num: Int? = null |
| Extension Functions | In Java, If we want to extend the functionality of existing class we need to create a new class and inherit the parent class. So Extension functions are not available in Java | Kotlin provides developers the ability to extend an existing class with new functionality. We can create extend functions by prefixing the name of a class to name of the new function. |
| Coroutines Support | In Java, whenever if we initiate a long-running network I/0 or CPU Intensive operations, the corresponding thread will be blocked. As Android is a single-threaded by default. Java provides the ability to create multiple threads in the background and run but managing them is a complex task. | In Kotlin, We can create multiple threads to run these long-running intensive operations but we have coroutines support, which will suspend execution at a certain point without blocking threads while executing long-running intensive operations. |
| No checked exceptions | In Java, We have checked exceptions support which makes developers declare and catch the exception which ultimately leads to robust code with good error handling. | In Kotlin, we don’t have checked exceptions. So developers don’t need to declare or catch the exceptions, which have advantages and disadvantages. |
| Data classes | In Java, suppose we need to have a class which needs to hold data but nothing else. For this we need to define constructors, variables to store data, getter and setter methods, hashcode(), toString(), and equals() functions | In Kotlin, If we need to have classes which need to hold data we can declare a class with keyword “data” in the class definition then the compiler will take care of all of this work such as creating constructors, getter, setter methods for different fields. |
| Smart casts | In Java, We need to check the type of variables and cast according to our operation. | In Kotlin, smart casts will handle these casting checks with keyword “is-checks” which will check for immutable values and performs implicit casting. |
| Type inference | In Java, we need to specify a type of each variable explicitly while declaring. | In Kotlin, we don’t need to specify the type of each variable explicitly based on assignment it will handle. If we want to specify explicitly we can do. |
| Functional Programming | Java doesn’t have functional programming support till Java 8 but while developing Android applications it supports the only subset of Java 8 features. | Kotlin is a mix of procedural and functional programming language which consists of many useful methods such as lambda, operator overloading, higher-order functions, and lazy evaluation, etc. |

**Question:**  What are coroutines in Kotlin?

**Answer:** Unlike many other languages with similar capabilities, async and await are not keywords in Kotlin and are not even part of its standard library.

kotlinx.coroutines is a rich library for coroutines developed by JetBrains. It contains a number of high-level coroutine-enabled primitives, including launch, async and others. Kotlin Coroutines give you an API to write your asynchronous code sequentially.

The documentation says Kotlin Coroutines are like lightweight threads. They are lightweight because creating coroutines doesn’t allocate new threads. Instead, they use predefined thread pools, and smart scheduling. Scheduling is the process of determining which piece of work you will execute next.

Additionally, coroutines can be suspended and resumed mid-execution. This means you can have a long-running task, which you can execute little-by-little. You can pause it any number of times, and resume it when you’re ready again.

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

**Answer:** 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)

**Question:**  What is the difference between var and val in Kotlin?

* **Answer:** **var** is like general variable and it's known as a *mutable* variable in kotlin and can be assigned multiple times.
* **val** is like Final variable and it's known as *immutable* in Kotlin and can be initialized only single time.

+----------------+-----------------------------+---------------------------+

| | val | var |

+----------------+-----------------------------+---------------------------+

| Reference type | Immutable(once initialized | Mutable(can able to change|

| | can't be reassigned) | value) |

+----------------+-----------------------------+---------------------------+

| Example | val n = 20 | var n = 20 |

+----------------+-----------------------------+---------------------------+

| In Java | final int n = 20; | int n = 20; |

+----------------+-----------------------------+---------------------------+

**Question:**  How to correctly concatenate a String in Kotlin?

**Answer:** In Kotlin, you can concatenate

using string interpolation / templates

val a = "Hello"

val b = "World"

val c = "$a $b"

using the + / plus() operator

val a = "Hello"

val b = "World"

val c = a + b // same as calling operator function a.plus(b)

val c = a.plus(b)

print(c)

using the StringBuilder

val a = "Hello"

val b = "World"

val sb = StringBuilder()

sb.append(a).append(b)

val c = sb.toString()

print(c)

**Question:**  What is basic difference between fold and reduce in Kotlin? When to use which?

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

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

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

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

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

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

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

**Question:**  What is the idiomatic way to remove duplicate strings from array?

**Answer:** How to remove duplicates from an Array<String?> in Kotlin?

Answer: Use the distinct extension function:

val a = arrayOf("a", "a", "b", "c", "c")

val b = a.distinct() // ["a", "b", "c"]

You can also use:

toSet, toMutableSet

toHashSet - if you don't need the original ordering to be preserved

These functions produce a Set instead of a List and should be a little bit more efficient than distinct.

**Question:**  How to create singleton in Kotlin?

**Answer:** Just use object.

object SomeSingleton

The above Kotlin object will be compiled to the following equivalent Java code:

public final class SomeSingleton {

public static final SomeSingleton INSTANCE;

private SomeSingleton() {

INSTANCE = (SomeSingleton)this;

System.out.println("init complete");

}

static {

new SomeSingleton();

}

}

This is the preferred way to implement singletons on a JVM because it enables thread-safe lazy initialization without having to rely on a locking algorithm like the complex double-checked locking.

**Question:**  Where should I use var and where val?

**Answer:** Use var where value is changing frequently. For example while getting location of android device:

var integerVariable : Int? = null

Use val where there is no change in value in whole class. For example you want set textview or button's text programmatically.

val stringVariables : String = "Button's Constant or final Text"

**Question:**  Explain what is wrong with that code?

Why is this code wrong?

class Student (var name: String) {

init() {

println("Student has got a name as $name")

}

constructor(sectionName: String, var id: Int) this(sectionName) {

}

}

**Answer**: The property of the class can’t be declared inside the secondary constructor.. This will give an error because here we are declaring a property id of the class in the secondary constructor, which is not allowed.

If you want to use some property inside the secondary constructor, then declare the property inside the class and use it in the secondary constructor:

class Student (var name: String) {

var id: Int = -1

init() {

println("Student has got a name as $name")

}

constructor(secname: String, id: Int) this(secname) {

this.id = id

}

}

**Question:**  What will be result of the following code execution?

What will be the output?

val aVar by lazy {

println("I am computing this value")

"Hola"

}

fun main(args: Array<String>) {

println(aVar)

println(aVar)

}

**Answer**: For lazy the first time you access the Lazy property, the initialisation (lazy() function invocation) takes place. The second time, this value is remembered and returned:

I am computing this value

Hola

Hola

**Question:**  What is the difference between suspending vs. blocking?

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

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

**Question:**  What is the equivalent of Java static methods in Kotlin?

**Answer:** Place the function in the companion object.

class Foo {

public static int a() { return 1; }

}

will become:

class Foo {

companion object {

fun a() : Int = 1

}

}

// to run

Foo.a();

Another way is to solve most of the needs for static functions with package-level functions. They are simply declared outside a class in a source code file. The package of a file can be specified at the beginning of a file with the package keyword. Under the hood these "top-level" or "package" functions are actually compiled into their own class. In the above example, the compiler would create a class FooPackage with all of the top-level properties and functions, and route all of your references to them appropriately.

Consider:

package foo

fun bar() = {}

usage:

import foo.bar

**Question:**  What is suspending function in Kotlin?

**Answer:** A suspending function is just a regular Kotlin function with an additional suspend modifier which indicates that the function can suspend the execution of a coroutine without blocking the current thread. This means that the code you are looking at might stop executing at the moment it calls a suspending function, and will resume at some later time. However, it doesn’t say anything about what the current thread will do in the meantime.

Suspending functions can invoke any other regular functions, but to actually suspend the execution, it has to be another suspending function.A suspending function cannot be invoked from a regular function, therefore several so-called coroutine builders are provided, which allow calling a suspending function from a regular non-suspending scope like launch, async, runBlocking.

**Question:**  Explain advantages of "when" vs "switch" in Kotlin

**Answer:** In Java we use switch but in Kotlin, that switch gets converted to when. When has a better design. It is more concise and powerful than a traditional switch. when can be used either as an expression or as a statement.

Some examples of when usage:

Two or more choices

when(number) {

1 -> println("One")

2, 3 -> println("Two or Three")

4 -> println("Four")

else -> println("Number is not between 1 and 4")

}

"when" without arguments

when {

number < 1 -> print("Number is less than 1")

number > 1 -> print("Number is greater than 1")

}

Any type passed in "when"

fun describe(obj: Any): String =

when (obj) {

1 -> "One"

"Hello" -> "Greeting"

is Long -> "Long"

!is String -> "Not a string"

else -> "Unknown"

}

Smart casting

when (x) {

is Int -> print("X is integer")

is String -> print("X is string")

}

Ranges

when(number) {

1 -> println("One") //statement 1

2 -> println("Two") //statement 2

3 -> println("Three") //statement 3

in 4..8 -> println("Number between 4 and 8") //statement 4

!in 9..12 -> println("Number not in between 9 and 12") //statement 5

else -> println("Number is not between 1 and 8") //statement 6

}

**Question:**  What are the advantages of Kotlin over Java?

**Answer:** Basically for me less thinking required to write kotlin equivalent to most java code:

data class

java: you have to write getters and setters for each thing, you have to write hashCode properly (or let IDE auto generate, which you have to do again every time you change the class), toString (same problem as hashcode) and equals (same problem as hashCode). or you could use lombok, but that comes with some quirky problems of its own. record types are hopefully on the way. \*kotlin: data class does it all for you.

getter and setter patterns

java: rewrite the getter and setter for each variable you use it for

kotlin: don't have to write getter and setter, and custom getter and setter take a lot less typing in kotlin if you do want to. also delegates exist for identical getters\setters

abstract vs open classes

java: you have to make an abstract class implementation

kotlin: open class lets you make an inheritable class while also being usable itself. nice mix of interface and regular class imo

extension functions

java: doesnt exist

kotlin: does exist, makes functions more clear in usage and feels more natural.

null

java: Anything but primitives can be null at any time.

kotlin: you get to decide what can and cant be null. allows for nice things like inline class

singleton

java: Memorize singleton pattern

kotlin: object instead of class

generics

java: Theyre alright, nothing fancy

kotlin: Reified generics (you can access the actual type), in and out for covariance

named parameters

java: does not exist, easy to break api back-compatibility if you arent careful.

kotlin: does exist, easy to preserve api back-compatiblity.

primary constructor

java: does not have per-se, you still have to define everything inside the class

kotlin: very nice to be able to quickly write a constructor without any constructor function or extra needless declarations

**Question:**  What are some disadvantages of Kotlin?

**Answer:** Some think that Kotlin is a mess of extra syntax and keywords. Here are a few keywords which have non-obvious meanings: internal, crossinline, expect, reified, sealed, inner, open. Java has none of these. Kotlin is also amusingly inconsistent in its keywords: a function is is declared with ‘fun’, but an interface is declared with ‘interface’ (not ‘inter’?). Kotlin also doesn’t have checked exceptions. Checked exceptions have become unfashionable, yet many (including me) find them a powerful way to ensure that your code is robust. Finally, Kotlin hides a lot of what goes on. In Java, you can trace through almost every step of program logic. This can be vital for hunting down bugs. In Kotlin, if you define a data class, then getters, setters, equality testing, to string, and hash code are added for you invisibly. This can be a bad idea.

Also according docs, 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

**Question:**  What is the difference between "open" and "public" in Kotlin?

**Answer:** The open keyword means “open for extension“. The open annotation on a class is the opposite of Java's final: it allows others to inherit from this class.

If you do not specify any visibility modifier, public is used by default, which means that your declarations will be visible everywhere. public is the default if nothing else is specified explicitly.

**Question:**  What is the difference between “const” and “val”?

**Answer:** consts are compile time constants. Meaning that their value has to be assigned during compile time, unlike vals, where it can be done at runtime.

This means, that consts can never be assigned to a function or any class constructor, but only to a String or primitive.

For example:

const val foo = complexFunctionCall() //Not okay

val fooVal = complexFunctionCall() //Okay

const val bar = "Hello world" //Also okay

**Question:**  How to convert List to Map in Kotlin?

**Answer:** You have two choices:

The first and most performant is to use associateBy function that takes two lambdas for generating the key and value, and inlines the creation of the map:

val map = friends.associateBy({it.facebookId}, {it.points})

The second, less performant, is to use the standard map function to create a list of Pair which can be used by toMap to generate the final map:

val map = friends.map { it.facebookId to it.points }.toMap()

**Question:**  What is the idiomatic way to deal with nullable values, referencing or converting them?

**Answer:** If I have a nullable type Xyz?, I want to reference it or convert it to a non-nullable type Xyz. What is the idiomatic way of doing so in Kotlin?

Answer: You have several options:

val something: Xyz? = createPossiblyNullXyz()

// access it as non-null asserting that with a sure call

// throws an exception if the value is null

val result1 = something!!.foo()

// access it only if it is not null using safe operator,

// returning null otherwise

val result2 = something?.foo()

// access it only if it is not null using safe operator,

// otherwise a default value using the elvis operator

val result3 = something?.foo() ?: differentValue

// null check it with `if` expression and then use the value,

// similar to result3 but for more complex cases harder to do in one expression

val result4 = if (something != null) {

something.foo()

} else {

...

differentValue

}

// null check it with `if` statement doing a different action

if (something != null) {

something.foo()

} else {

someOtherAction()

}

**Question:**  What is the difference between List and Array types?

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

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

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

**Question:**  val mutableList vs var immutableList. When to use which in Kotlin?

**Answer:** : Mutable and immutable list increase the design clarity of the model.

This is to force developer to think and clarify the purpose of collection.

If the collection will change as part of design, use mutable collection

If model is meant only for viewing, use immutable list

Purpose of val and var is different from immutable and mutable list.

val and var keyword talk about the how a value/reference of a variable should be treated.

var - value/reference assigned to a variable can be changed at any point of time.

val - value/reference can be assigned only once to a variable and can't be changed later point in the execution.

There are several reasons why immutable objects are often preferable:

They encourage functional programming, where state is not mutated, but passed on to the next function which creates a new state based on it. This is very well visible in the Kotlin collection methods such as map, filter, reduce, etc.

A program without side effects is often easier to understand and debug (you can be sure that the value of an object will always be the one at its definition).

In multithreaded programs, immutable resources cannot cause race conditions, as no write access is involved.

You have also some disadvantages:

Copying entire collections just to add/remove a single element is computationally expensive.

In some cases, immutability can make the code more complex, when you tediously need to change single fields. In Kotlin, data classes come with a built-in copy() method where you can copy an instance, while providing new values for only some of the fields.

**Question:**  What is a difference between a class and object in Kotlin?

**Answer:** An object is a singleton. You do not need to create an instance to use it.

A class needs to be instantiated to be used.

The primary use case of object in Kotlin is because Kotlin tries to do away with static, and primitives, leaving us with a purely object oriented language. Kotlin still uses static and primitives underneath the hood, but it discourages devs to use those concepts any more. Instead, now Kotlin replaces static with singleton object instances. Where you would previously use static field in Java, in Kotlin you will now create an object, and put that field in the object.

**Question:**  How is it recommended to create constants in Kotlin?

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

val MY\_CONSTANT\_1 = "Constants1"

// or

const val MY\_CONSTANT\_2 = "Constants2"

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

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

object DbConstants {

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

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

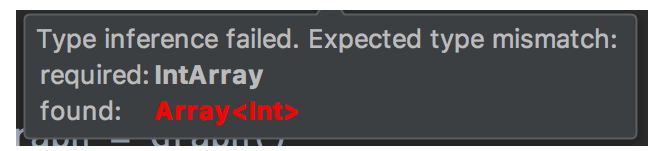
}

**Question:**  May you use IntArray and an Array is in Kotlin interchangeably?

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

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

So no, we can't use them interchangeably.



**Question:**  Rewrite this code in Kotlin

**Answer:** Can you rewrite this Java code in Kotlin?

public class Singleton {

private static Singleton instance = null;

private Singleton(){

}

private synchronized static void createInstance() {

if (instance == null) {

instance = new Singleton();

}

}

public static Singleton getInstance() {

if (instance == null) createInstance();

return instance;

}

Answer: Using Kotlin:

object Singleton

**Question:**  How would you create a singleton with parameter in Kotlin?

**Answer:** Because a Kotlin object can’t have any constructor, you can’t pass any argument to it.

So look at this code from Google's architecture components sample code, which uses the also function:

class UsersDatabase : RoomDatabase() {

companion object {

@Volatile private var INSTANCE: UsersDatabase? = null

fun getInstance(context: Context): UsersDatabase =

INSTANCE ?: synchronized(this) {

INSTANCE ?: buildDatabase(context).also { INSTANCE = it }

}

private fun buildDatabase(context: Context) =

Room.databaseBuilder(context.applicationContext,

UsersDatabase::class.java, "Sample.db")

.build()

}

}

**Question:**  What is the Kotlin double-bang (!!) operator?

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

Consider:

fun main(args: Array<String>) {

var email: String?

email = null

println(email!!)

}

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

**Question:**  What is the purpose of Unit-returning in functions? Why is VALUE there? What is this VALUE?

**Answer:** fun printHello(name : String?) : Unit {

if (name != null)

print("Hello, $name!")

else

print("Hi there!")

// We don't need to write 'return Unit.VALUE' or 'return', although we could

}

Answer: The purpose is the same as C's or Java's void. Only Unit is a proper type, so it can be passed as a generic argument etc.

Why we don't call it "Void": because the word "void" means "nothing", and there's another type, Nothing, that means just "no value at all", i.e. the computation did not complete normally (looped forever or threw an exception). We could not afford the clash of meanings.

Why Unit has a value (i.e. is not the same as Nothing): because generic code can work smoothly then. If you pass Unit for a generic parameter T, the code written for any T will expect an object, and there must be an object, the sole value of Unit.

How to access that value of Unit: since it's a singleton object, just say Unit

UNIT actually contains valuable information, it basically just means "DONE". It just returns the information to the caller, that the method has been finished.

**Question:**  When would you use Elvis operator in Kotlin?

**Answer:** The Elvis operator is part of many programming languages, e.g. Kotlin but also Groovy or C#. The Elvis operator is the ternary operator with its second operand omitted.

x ?: y // yields `x` if `x` is not null, `y` otherwise.

If x isn't null, then it will be returned. If it is null, then the y will be returned.

**Question:**  What are scope functions in Kotlin?

**Answer:** 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,

also.

**Question:**  Why is there no static keyword in Kotlin?

**Answer:** The main advantage of this is that everything is an object. Companion objects can inherit from other classes or implement interfaces and generally behave like any other singleton.

In Java, static members are treated very differently than object members. This means that you can't do things like implementing an interface or putting your class "instance" into a map or pass it as a parameter to a method that takes Object. Companion objects allow for these things. That's the advantage.

**Question:**  What is inline class in Kotlin and when do we need one? Provide an example.

**Answer:** Sometimes it is necessary for business logic to create a wrapper around some type. However, it introduces runtime overhead due to additional heap allocations. Moreover, if the wrapped type is primitive, the performance hit is terrible, because primitive types are usually heavily optimized by the runtime.

Inline classes provide us with a way to wrap a type, thus adding functionality and creating a new type by itself. As opposed to regular (non-inlined) wrappers, they will benefit from improved performance. This happens because the data is inlined into its usages, and object instantiation is skipped in the resulting compiled code.

inline class Name(val s: String) {

val length: Int

get() = s.length

fun greet() {

println("Hello, $s")

}

}

fun main() {

val name = Name("Kotlin")

name.greet() // method `greet` is called as a static method

println(name.length) // property getter is called as a static method

}

Some notes about inline classes:

A single property initialized in the primary constructor is the basic requirement of an inline class

Inline classes allow us to define properties and functions just like regular classes

Init blocks, inner classes, and backing fields are not allowed

Inline classes can inherit only from interfaces

Inline classes are also effectively final

**Question:**  Explain the difference between Inline classes vs type aliases

**Answer:** The crucial difference is that type aliases are assignment-compatible with their underlying type (and with other type aliases with the same underlying type), while inline classes are not.

In other words, inline classes introduce a truly new type, contrary to type aliases which only introduce an alternative name (alias) for an existing type:

typealias NameTypeAlias = String

inline class NameInlineClass(val s: String)

fun acceptString(s: String) {}

fun acceptNameTypeAlias(n: NameTypeAlias) {}

fun acceptNameInlineClass(p: NameInlineClass) {}

fun main() {

val nameAlias: NameTypeAlias = ""

val nameInlineClass: NameInlineClass = NameInlineClass("")

val string: String = ""

acceptString(nameAlias) // OK: pass alias instead of underlying type

acceptString(nameInlineClass) // Not OK: can't pass inline class instead of underlying type

// And vice versa:

acceptNameTypeAlias(string) // OK: pass underlying type instead of alias

acceptNameInlineClass(string) // Not OK: can't pass underlying type instead of inline class

}

**Question:**  What is Coroutine Scope and how is that different from Coroutine Context?

**Answer:** Coroutines always execute in some context represented by a value of the CoroutineContext type, defined in the Kotlin standard library. The coroutine context is a set of various elements. The main elements are the Job of the coroutine.

CoroutineScope has no data on its own, it just holds a CoroutineContext. Its key role is as the implicit receiver of the block you pass to launch, async etc.

runBlocking {

val scope0 = this

// scope0 is the top-level coroutine scope.

scope0.launch {

val scope1 = this

// scope1 inherits its context from scope0. It replaces the Job field

// with its own job, which is a child of the job in scope0.

// It retains the Dispatcher field so the launched coroutine uses

// the dispatcher created by runBlocking.

scope1.launch {

val scope2 = this

// scope2 inherits from scope1

}

}

}

You might say that CoroutineScope formalizes the way the CoroutineContext is inherited. You can see how the CoroutineScope mediates the inheritance of coroutine contexts. If you cancel the job in scope1, this will propagate to scope2 and will cancel the launched job as well.

**Question:**  Imagine you moving your code from Java to Kotlin. How would you rewrite this code in Kotlin?

**Answer:** Details:

public class Foo {

private static final Logger LOG = LoggerFactory.getLogger(Foo.class);

}

Answer: Use Static-like approach:

class MyClass {

companion object {

val LOG = Logger.getLogger(MyClass::class.java.name)

}

fun foo() {

LOG.warning("Hello from MyClass")

}

}

**Question:**  How would you override default getter for Kotlin data class?

**Answer:** Given the following Kotlin class:

data class Test(val value: Int)

How would I override the Int getter so that it returns 0 if the value negative?

Answer:

Have your business logic that creates the data class alter the value to be 0 or greater before calling the constructor with the bad value. This is probably the best approach for most cases.

Don't use a data class. Use a regular class.

class Test(value: Int) {

val value: Int = value

get() = if (field < 0) 0 else field

override fun equals(other: Any?): Boolean {

if (this === other) return true

if (other !is Test) return false

return true

}

override fun hashCode(): Int {

return javaClass.hashCode()

}

}

Create an additional safe property on the object that does what you want instead of having a private value that's effectively overriden.

data class Test(val value: Int) {

val safeValue: Int

get() = if (value < 0) 0 else value

}

**Question:**  What is Kotlin backing field is used for?

**Answer:** Backing field is an autogenerated field for any property which can only be used inside the accessors(getter or setter) and will be present only if it uses the default implementation of at least one of the accessors, or if a custom accessor references it through the field identifier. This backing field is used to avoid the recursive call of an accessor which ultimately prevents the StackOverflowError.

Classes in Kotlin cannot have fields. However, sometimes it is necessary to have a backing field when using custom accessors. For these purposes, Kotlin provides an automatic backing field which can be accessed using the field identifier.

var selectedColor: Int = someDefaultValue

get() = field

set(value) {

field = value

}

**Question:**  What are Object expressions in Kotlin and when to use them?

**Answer:** Sometimes we need to create an object of some class with slight modification, without explicitly declaring a new subclass for it. Java handles this case with anonymous inner classes. Kotlin uses object expression to achieve the same functionality. We can even create an object expression for an interface or abstract class by just implementing their abstract methods.

It is often used as a substitution to a Java anonymous class:

window.addMouseListener(object : MouseAdapter() {

override fun mouseClicked(e: MouseEvent) {

// ...

}

override fun mouseEntered(e: MouseEvent) {

// ...

}

})

**Question:**  How to create empty constructor for data class in Kotlin?

**Answer:** If you give default values to all the fields - empty constructor is generated automatically by Kotlin.

data class User(var id: Long = -1,

var uniqueIdentifier: String? = null)

and you can simply call:

val user = User()

Another option is to declare a secondary constructor that has no parameters:

data class User(var id: Long,

var uniqueIdentifier: String?){

constructor() : this(-1, null)

}

**Question:**  How to create an instance of anonymous class of abstract class in Kotlin?

**Answer:** Assume that KeyAdapter is an abstract class with several methods that can be overridden.

In java I can do:

KeyListener keyListener = new KeyAdapter() {

@Override public void keyPressed(KeyEvent keyEvent) {

// ...

}

};

How to do the same in Kotlin?

Answer: Sometimes we need to create an object of a slight modification of some class, without explicitly declaring a new subclass for it. Kotlin handles this case with object expressions and object declarations.

val keyListener = object : KeyAdapter() {

override fun keyPressed(keyEvent : KeyEvent) {

// ...

}

**Question:**  Why do we use “companion object” as a kind of replacement for Java static fields in Kotlin?

**Answer:** Because statics are not object-oriented. Kotlin does, however, have globals, which function similarly, and objects, which provide static functionality but remain object-oriented.

Java static part of a class can be elegantly expressed in terms of singleton: it's a singleton object that can be called by the class' name. Hence the naming: it's an object that comes with a class.

Apart from naming, it is more powerful than Java static members: it can extend classes and interfaces, and you can reference and pass it around just like other objects.

**Question:**  What is the difference between “\*” and “Any” in Kotlin generics?

**Answer:** List<\*> can contain objects of any type, but only that type, so it can contain Strings (but only Strings)

while List<Any> can contain Strings and Integers and whatnot, all in the same list

**Question:**  How can I create “static” method for enum in Kotiln?

**Answer:** Just like with any other class, you can define a class object in an enum class:

enum class CircleType {

FIRST,

SECOND,

THIRD;

companion object {

fun random(): CircleType = FIRST

}

}

Then you'll be able to call this function as CircleType.random().

**Question:**  What is SAM Conversion in Kotlin?

**Answer:** "SAM" stands for "single abstract method", and "SAM-type" refers to interfaces like Runnable, Callable.

Just like Java 8, Kotlin supports SAM conversions. This means that Kotlin function literals can be automatically converted into implementations of Java interfaces with a single non-default method, as long as the parameter types of the interface method match the parameter types of the Kotlin function.

That means that when you call some Java method from Kotlin, and that method satisfies conditions described above, you can pass lambda or method reference instead.

**Question:**  Explain the difference between lateinit and lazy in details

**Answer:** Here are the significant differences between lateinit var and by lazy { ... } delegated property:

lazy { ... } delegate can only be used for val properties, whereas lateinit can only be applied to vars, because it can't be compiled to a final field, thus no immutability can be guaranteed;

lateinit var has a backing field which stores the value, and by lazy { ... } creates a delegate object in which the value is stored once calculated, stores the reference to the delegate instance in the class object and generates the getter for the property that works with the delegate instance. So if you need the backing field present in the class, use lateinit;

In addition to vals, lateinit cannot be used for nullable properties and Java primitive types (this is because of null used for uninitialized value);

lateinit var can be initialized from anywhere the object is seen from, e.g. from inside a framework code, and multiple initialization scenarios are possible for different objects of a single class. by lazy { ... }, in turn, defines the only initializer for the property, which can be altered only by overriding the property in a subclass. If you want your property to be initialized from outside in a way probably unknown beforehand, use lateinit.

Initialization by lazy { ... } is thread-safe by default and guarantees that the initializer is invoked at most once (but this can be altered by using [another lazy overload][1]). In the case of lateinit var, it's up to the user's code to initialize the property correctly in multi-threaded environments.

A Lazy instance can be saved, passed around and even used for multiple properties. On contrary, lateinit vars do not store any additional runtime state (only null in the field for uninitialized value).

If you hold a reference to an instance of Lazy, [isInitialized()][2] allows you to check whether it has already been initialized (and you can [obtain such instance with reflection][3] from a delegated property). To check whether a lateinit property has been initialized, you can [use property::isInitialized since Kotlin 1.2][4].

A lambda passed to by lazy { ... } may capture references from the context where it is used into its [closure][5].. It will then store the references and release them only once the property has been initialized. This may lead to object hierarchies, such as Android activities, not being released for too long (or ever, if the property remains accessible and is never accessed), so you should be careful about what you use inside the initializer lambda.

**Question:**  Provide a real use case when inline classes may be useful

**Answer:** Imagine an authentication method in an API that looks as follows:

fun auth(userName: String, password: String) { println("authenticating $userName.") }

Since both parameters are of type String, you may mess up their order which gets even more likely with an increasing number of arguments.

Inline class wrappers around these types can help you mitigate that risk and give you simple, type-safe wrappers without introducing additional heap allocations:

inline class Password(val value: String)

inline class UserName(val value: String)

fun auth(userName: UserName, password: Password) { println("authenticating $userName.")}

fun main() {

auth(UserName("user1"), Password("12345"))

//does not compile due to type mismatch

auth(Password("12345"), UserName("user1"))

}

**Question:**  What's wrong with that code?

**Answer:** Let's say I want to override the Int getter so that it returns 0 if the value negative for the data class. What's bad with that approach?

data class Test(private val \_value: Int) {

val value: Int

get() = if (\_value < 0) 0 else \_value

}

Answer: The problem with this approach is that data classes aren't really meant for altering data like this. They are really just for holding data. Overriding the getter for a data class like this would mean that Test(0) and Test(-1) wouldn't equal one another and would have different hashCodes, but when you called .value, they would have the same result. This is inconsistent, and while it may work for you, other people on your team who see this is a data class, may accidentally misuse it.

**Question:**  Why would you use apply in Kotlin?

**Answer:** Look at this code:

person.name = "Tony Stark"

person.age = 52

and equivalent with apply will be:

val person = Person().apply {

name = "Tony Stark" // this. can be omitted

age = 52 // this. can be omitted

// ...

}

This way you don't have to repeat person several times. Apply is used to keep things that belong together in one place (mostly initializations).

**Question:**  Rewrite this code using "run" extension function

**Answer:** Consider:

val generator = PasswordGenerator()

generator.seed = "someString"

generator.hash = {s -> someHash(s)}

generator.hashRepititions = 1000

val password: Password = generator.generate()

How would you refactor this code using run extension function?

Answer: Someone didn’t quite think through the design of this password generator class. Its constructor does nothing, but it needs a lot of initialization. To use this class, I need to introduce a variable generator, set all necessary parameters and use generate to generate the actual password.

val password: Password = PasswordGenerator().run {

seed = "someString"

hash = {s -> someHash(s)}

hashRepetitions = 1000

generate()

}

Lambdas in Kotlin implicitly return the result of the last line. That’s why I can omit the temporary variable and store the password directly. Because an extension function is passed to run I can also access the password generator’s properties like seed or hash directly.

**Question:**  How would you refactor this code using "apply"?

**Answer:** Consider:

class Message(message: String, signature: String) {

val body = MessageBody()

init {

body.text = message + "\n" + signature

}

}

Do you see any refactoring that could be done?

Answer: You can write:

class Message(message: String, signature: String) {

val body = MessageBody().apply {

text = message + "\n" + signature

}

}

**Question:**  What is The Billion Dollar Mistake?

**Answer:** Kotlin's type system is aimed at eliminating the danger of null references from code, also known as the The Billion Dollar Mistake.

One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference will result in a null reference exception. In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that can not (non-null references).

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object-oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn’t resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

**Question:**  What is a motivation to make classes final by default in Kotlin? Do you agree with that decision?

**Answer:** First Kotlin takes many ideas from the functional programming world and uses immutability as often as it can to avoid all the known problems with mutation. Also proper designing a class for inheritance requires an excruciating amount of work (and building at least 3 separate subclasses, to verify that you base class actually is useful). Most classes should be final. Extending them is probably a bad idea.

The second thought which comes to my mind is that inheritance is often missused. There is the principle "Favor composition over inheritance" as a guideline for better designs. So declaring every class as final by default forces the developer to at least stop for a moment and think about alternative ways to solve the problem instead of using inheritance for the wrong reasons.

**Question:**  How does the reified keyword in Kotlin work?

**Answer:** In an ordinary generic function like myGenericFun, you can't access the type T because it is, like in Java, erased at runtime and thus only available at compile time. Therefore, if you want to use the generic type as a normal Class in the function body you need to explicitly pass the class as a parameter like the parameter c in the example.

fun <T> myGenericFun(c: Class<T>)

By marking a type as reified, we’ll have the ability to use that type within the function.

As for a real example, in Java, when we call startActivity, we need to specify the destination class as a parameter. The Kotlin way is:

inline fun <reified T : Activity> Activity.startActivity() {

startActivity(Intent(this, T::class.java))

}

You can only use reified in combination with an inline function. Such a function makes the compiler copy the function's bytecode to every place where the function is being used (the function is being "inlined"). When you call an inline function with reified type, the compiler knows the actual type used as a type argument and modifies the generated bytecode to use the corresponding class directly. Therefore calls like myVar is T become myVar is String (if the type argument were String) in the bytecode and at runtime.

**Question:**  How to implement Builder pattern in Kotlin?

**Answer:** First and foremost, in most cases you don't need to use builders in Kotlin because we have default and named arguments but if you need one use:

class Car( //add private constructor if necessary

val model: String?,

val year: Int

) {

private constructor(builder: Builder) : this(builder.model, builder.year)

class Builder {

var model: String? = null

private set

var year: Int = 0

private set

fun model(model: String) = apply { this.model = model }

fun year(year: Int) = apply { this.year = year }

fun build() = Car(this)

}

}

Usage:

val car = Car.Builder().model("X").build()

**Question:**  When to use and do not use an inline function in Kotlin?

**Answer:** Using higher-order functions imposes certain runtime penalties: each function is an object, and it captures a closure, i.e. those variables that are accessed in the body of the function. Memory allocations (both for function objects and classes) and virtual calls introduce runtime overhead.

But it appears that in many cases this kind of overhead can be eliminated by inlining the lambda expressions.

The inline modifier affects both the function itself and the lambdas passed to it: all of those will be inlined into the call site.

The most important case when we use inline modifier is when we define util-like functions with parameter functions. This is why inline modifier is mostly an important optimization for library developers.

Consider:

inline fun inlined(block: () -> Unit) {

println("before")

block()

println("after")

}

inlined {

println("do something here")

}

// No Function object instance will be created, instead, the code around the

// invocation of block inside the inlined function will be copied

// to the call site

System.out.println("before");

System.out.println("do something here");

System.out.println("after");

When we don’t have function type parameter, reified type parameter, and we don’t need non-local return, then we most likely shouldn’t use inline modifier.

Also there is the code size problem. Inlining a large function could dramatically increase the size of the bytecode because it's copied to every calls site. Inlining may cause the generated code to grow; however, if we do it in a reasonable way (i.e. avoiding inlining large functions), it will pay off in performance.

**Question:**  How Kotlin coroutines are better than RxKotlin/RxJava?

**Answer:** Kotlin coroutines are different from Rx. Both are designed to address a problem of asynchronous programming, however their approach to solution is very different:

Rx comes with a particular functional style of programming that can be implemented in virtually any programming language without support from the language itself. It works well when the problem at hand easily decomposes into a sequence of standard operators and not so well otherwise.

Kotlin coroutines provide a language feature that let library writers implement various asynchronous programming styles, including, but not limited to functional reactive style (Rx). With Kotlin coroutines you can also write your asynchronous code in imperative style, in promise/futures-based style, in actor-style, etc.

How Kotlin coroutines are better than RxKotlin? You just write sequential code, everything is as easy as writing synchronous code except it execute asynchronously. It's easier to grasp.

Coroutines are better to deal with resources

In RxJava you can assign computations to schedulers but subscribeOn() and ObserveOn()are confusing. Every coroutine is given a thread context and return to parent context. For a channel, both side (producer, consumer) execute on his own context. Coroutines are more intuitive on thread or thread pool affectation.

Coroutines give more control on when those computation occur. You can for example pass hand (yield), prioritize (select), parallelize (multiple producer/actor on channel) or lock resource (Mutex) for a given computation. It may not matter on server (where RxJava came first) but on resources limited environment this level of control may be required.

Due to it's reactive nature, backpressure doesn't fit well in RxJava. In the other end send() to channel is a suspensive function that suspend when channel capacity is reached. It's out-of-the-box backpressure given by nature. You could also offer() to channel, in which case the call never suspend but return false in case the channel is full, effectively reproducing onBackpressureDrop() from RxJava. Or you could just write your own custom backpressure logic, which won't be difficult with coroutines, especially compared to do the same with RxJava.

**Question:**  What is the difference between launch/join and async/await in Kotlin coroutines?

**Answer:** launch is used to\*\* fire and forget coroutine\*\*. It is like starting a new thread. If the code inside the launch terminates with exception, then it is treated like uncaught exception in a thread -- usually printed to stderr in backend JVM applications and crashes Android applications. join is used to wait for completion of the launched coroutine and it does not propagate its exception. However, a crashed child coroutine cancels its parent with the corresponding exception, too.

async is used to start a coroutine that computes some result. The result is represented by an instance of Deferred and you must use await on it. An uncaught exception inside the async code is stored inside the resulting Deferred and is not delivered anywhere else, it will get silently dropped unless processed. You MUST NOT forget about the coroutine you’ve started with async.

**Question:**  What is the difference between Java field and Kotlin property?

**Answer:** This is an example of a Java field:

public String name = "Marcin";

Here is an example of a Kotlin property:

var name: String = "Marcin"

They both look very similar, but these are two different concepts. Direct Java equivalent of above Kotlin property is following:

private String name = "Marcin";

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

The default implementation of Kotlin property includes field and accessors (getter for val, and getter and setter for var). Thanks to that, we can always replace accessors default implementation with a custom one.

**Question:**  What is the difference between and interface and an abstract class?

**Answer:** Though both may seem very similar they are in fact very different and serve very different purposes. Abstract classes are meant to serve the purpose of Generalizing behavior while interfaces are meant to serve the purpose of Standardizing behavior.

**Question:**  lateinit vs lazy?

**Answer:** A) lateinit can be used for var properties where Kotlin promises the compiler that the variable will be initialized later failure of which will lead to exception. lazy can only be used for val properties. It will be initialized during the first call where the value will be stored in a cache so that another call to the same variable will serve the value stored in cache.

**Question:**  What are the types of equality in Kotlin?

**Answer:** A) There are two types of equality in Kotlin -

1) Referential Equality (===): It tells whether the two references are pointing to same address or not. In Kotlin it is represented with '===', unlike in Java where it is represented with '=='.

2) Structural Equality (==): Structural equality tells whether the data inside objects is equal or not. Java is Structural equality in Kotlin is represented by '==' where as in Java it is done by .equals() method.

In Kotlin also we can use .equals method but it is recommended to use == because Kotlin internally converts a==b, for example, to the following code:

a?.equals(b) ?: (b === null)

**Question:**  What is the use of vararg keyword in Kotlin?

**Answer:** A) varargs are used to pass unlimited variables to the constructor.

fun sum(vararg values : Int) = values.sum()

assertEquals(5, sum(2,3)) // true

Note: Only one vararg can be passed to a function. Multiple varargs leads to compilation error.

**Question:**  What are Destructuring Declarations in Kotlin?

**Answer:** A) Destructuring declarations allows us to destructure an object to various variables.

Let us take a data class for example. We know that whenever we pass args to data class, Kotlin creates a component for each arg, named - component1(), component2() etc.

Destructured declarations simply points to those components in the same order respectively.

Here's an example:

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

// destructuring declarations

val (username, userAge) = Person("vamsi", "21")

println(username) // vamsi

Here these username and userAge will directly point to the component functions of data class internally as follows:

val username = Person.component1()

val userAge = Person.component2()

**Question:**  What are Data Classes?

**Answer:** A) Data classes are specifically designed to hold the data. In data classes, the Designers of Kotlin has overrided methods - equals(), hashcode() and toString() internally to feciliate the data holding capabilities.

The main uses of data classes -

They can be easily copied structurally using copy() function,

They can be used for destructuring declarations

They also can inherit classes and interfaces

Limitations of Data Classes:

They cannot be open

They cannot inherit another data class

varargs cannot be used as arguments in data class as the data class internally needs to generate toString() and hashcode() method's logic.

**Question:**  If we create two data class objects with same data then are they equal?

**Answer:** A) Those two data are objects are equal structurally but not referentially.

Here is an example:

data class Person(name:String)

//creating objects for data class

val a = Person("Vamsi")

val b = Person("Vamsi")

println(a==b) // true; a normal class (without .equals() overridden) would return false in this case

println(a===b) // false

**Question:**  What are various scoping functions in Kotlin and when to use each of them?

**Answer:** A) There are 5 different scoping functions in Kotlin - let, apply, also, with, run. Basically they are used to execute a block of code within the context of an object.

Scope Functions Table:

Scope Functions Table

1. Let: let is an extension function that takes lambda block as parameter, has 'it' as object reference inside the block and returns the lambda block's result as return type.

Here's the syntax of let:

inline fun <T, R> T.let(block: (T) -> R): R {

return block(this)

}

Here's an example:

val fullName : String = Person()?.let {

it.name = "Vamsi"

it.name + " Tallapudi" // return value

}

println(fullName) //Vamsi Tallapudi

2. with: with takes the context object and the code block (lambda) as the arguments, has 'this' as object reference inside the block and performs the lambda functions operation on the context passed. Returns the lambda result as return value.

Here's the syntax:

inline fun <T, R> with(receiver: T, block: T.() -> R): R {

return receiver.block()

}

Here's an example:

val myList = mutableListOf(1, 2, 3, 4, 5)

val additionOfSquares = with(myList) {

val squaresList= map { it \* it }

squaresList.sum() //returns the sum value

}

println(additionOfSquares) // 55

3. apply: apply is an extension function that takes the code block (lambda) as an argument, has 'this' as object reference inside the block and performs the lambda function on the context. Returns context object as the return value.

Here's the syntax:

public inline fun <T> T.apply(block: T.() -> Unit): T {

...

block()

return this

}

Here's an example:

val vamsi = Person().apply {

name = "Vamsi"

age = 21

}

println(vamsi.age) // 21

4. also: also is an extension function that takes the code block (lambda) as an argument, has 'it' as object reference inside the block and performs the lambda function on the context. Returns context object as the return value.

val vamsi = Person().apply

{

name = "Vamsi"

}.also {

it.age = 21

}

println(vamsi.age) // 21

4. run: run is an extension function that takes the code block (lambda) as an argument, has 'this' as object reference inside the block and performs the lambda function on the context. Returns the lambda result as return value.

val vamsi = Person().apply

{

name = "Vamsi"

}.also {

it.age = 21

}

println(vamsi.age) // 21

**Question:**  What are inline functions? When to use them?

**Answer:** A) Functions that take lambda parameter as arguments generates objects inside calling function's code. If these functions are called at multiple places, multiple objects are created which affects the performance of our Android App. To avoid these memory allocations created by lambda expressions (anonymous objects are created), we make the functions inline by adding a keyword - 'inline' to our function.

inline fun SharedPreferences.edit(commit: Boolean = false, action: SharedPreferences.Editor.() -> Unit) {

val editor = edit()

action(editor)

if(commit)

editor.commit()

else

editor.apply()

}

Inline functions are generally used when we need to pass small functions as parameters. It is generally not advisable to pass large functions to inline functions.

**Question:**  What are noinline keyword? Where we need to use them in realtime scenario?

**Answer:** A) We cannot pass a lambda function, which comes as argument inside inline function, to another function that accepts lambda. We will get an error stating 'Illegal usage of inline-parameter'. In this case we need to pass that lambda function with noinline keyword which makes the compiler instead of writing the code to the called location, creates the function for that specific function.

Here's an example:

inline fun SharedPreferences.edit(

commit: Boolean = false,

noinline anotherFunction: Int.() -> Unit = {},

action: SharedPreferences.Editor.() -> Unit)

{

// passing noinline function to another function

myFun(anotherFunction)

val editor = edit()

action(editor)

if(commit)

editor.commit()

else

editor.apply()

}

fun myFun(importantAction: Int.() -> Unit) {

importantAction(-1)

}

We'll use this only in case if multiple lambdas are passed to function arguments. If there is only one lambda which need to be referenced in another function, we better not use inline function at all.

**Question:**  What is the use of crossinline in Kotlin?

**Answer:** A) When we don't want to return inside lambda function (non-local returns) that is passed as inline argument, we use crossinline keyword on that lambda argument.

Here's an example:

fun createPerson() {

val person = Person()

person.name = "Vamsi"

performFunction {

println("Created a person with name: ${person.name}")

// return // Not allowed here as its crossinline

}

}

inline fun performFunction (crossinline x : () -> Unit) {

x()

}

**Question:**  What is the use of infix in Kotlin?

**Answer:** A) infix functions are used for declaring a short form notation of a function.

Here's an example:

infix fun Int.printSmallest(x: Int) {

print(if(this < x) this else x)

}

1 printSmallest 5 // calling the function directly

**Question:**  What are Sealed classes in Kotlin?

**Answer:** A) Sealed classes are similar to enum classes which also has restrictive set of types allowed, except that Sealed classes can contains additional data to be propagated(which we cannot achieve with enum classes).

sealed class Result<out T: Any> {

data class Success<out T: Any>(val data: T): Result<T>()

data class Error(val exception: Exception): Result<Nothing>()

}

Sealed classes can contain any other clases like data class, pojo class, or even other sealed classes.

**Question:**  What is a Coroutine?

**Answer:** A) Coroutine is a Lightweight thread which helps in performing Asynchronous programming. Basically it is a concurrency design pattern which helps to run the long-running tasks which otherwise block the main thread. Advantages:

Lightweight: We can run many coroutines on a single thread,

Built in Cancellation support: Cancellation is propagated through the running Coroutine Hierarchy.

Fewer Memory leaks due to Structured Concurrency.

**Question:**  Does Coroutine runs in a Single thread?

**Answer:** A) A coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one.

**Question:**  What is Coroutine Scope?

**Answer:** A) Keeps track of any Coroutine created using launch or async functions (these extension functions are available in CoroutineScope).

**Question:**  What are the main features of Kotlin?

**Answer:** Kotlin has some advanced features for web and android development like

Clean, compact Syntax

Efficiency

Extension Function

Interoperability with Java

Reducing crashes at run-time

**Question:**  What are the different data types in Kotlin?

**Answer:** In Kotlin, everything is an object in the sense that we can call member functions and properties on any variable:

Data Type in Kotlin:

Numbers

Integer

Byte

Short

Int

Long

Floating

Float

Double

Boolean

Boolean represent boolean objects that can have only 2 values: TRUE and FALSE.

Built-in operation on boolean includes:

&& – conjunction (logical AND)

|| – disjunction (logical OR)

! – Negation (logical NOT)

Characters

Characters in Kotlin are represented by Char and are used to store single characters. The value of the char must be surrounded by single inverted commas.

String

Strings are the sequence of characters in double quotes. String in Kotlin is represented by a String. Elements are accessed by indexing operations. A string is immutable in nature. String values must be double-quoted.

Arrays

It is used to store multiple-value into a single variable. They are represented by the Array class. An array is invariant, which prevents possible run-time failure.

**Question:**  What are the variables in Kotlin?

**Answer:** Variables are used to store the data, it refers to a memory location. To declare a variable in Kotlin, either the var or val keyword is used.

var name = “Naukri Learning”

val employee = 50

The main advantage of Kotlin over other programming languages is that it does not need to be declared with a specified type. It is smart enough to understand the string or int variable.

**Question:**  What is the difference between Var and Val?

**Answer:** Val: Val is immutable i.e. we can’t change the value of a variable once it is assigned.

var name = “Naukri Learning”

name = “Naukri” // error

Var: Var is mutable in nature i.e. the value of the variables can be changed after assigning too.

var employee = “50”

employee = 15 // execute

**Question:**  What are Data Classes in Kotlin?

**Answer:** The main purpose of the data classes is to hold the data. In kotlin, data classes are marked with data

Requirements to create data classes:

The primary constructor should have at least one parameter

Parameters of the Primary constructor should be marked as var or val

Data classes can’t be abstract, inner, open or sealed.

The compiler automatically derives the following member from the properties declared in the primary constructor:

equal()

hashcode()

tostring()

componentN()

Copy

**Question:**  What are the different operators in Kotlin?

**Answer:** In a programming language, operators are the symbols that are used to perform specific mathematical and logical manipulations.

Similar to other programming languages, Kotlin has different operators:

Arithmetic

Relational

Assignment

Unary

Bitwise

Logical

**Question:**  What are the different functions in Kotlin?

**Answer:** The function is a set of codes that are used to perform a special task. You can pass data, into a function. Kotlin functions are declared using the fun keyword.

Depending upon whether a function is defined or inbuilt, Kotlin has two types of functions:

Standard Library Function

Kotlin has lots of builtin function, here are some of them:

arrayOf()

sum()

printIn()

sqrt()

rem()

compareTo()

**Question:**  What are the different types of Constructors available in Kotlin?

**Answer:** Constructors are special member function that is called when an object is created. It is a concise way to initialize class properties.

There are two types of constructors are available in Kotlin:

Primary:

It is used to initialize the class and declared it in the class header after the class name.

Declared using constructor Keyword

Codes are surrounded by parentheses with optional parameters

Secondary:

In Kotlin, a class can contain more than one secondary constructor

There are not common in Kotlin.

Secondary constructors are created using the “constructor” keyword.

**Question:**  What is the difference between Kotlin and Java?

**Answer:**

| **Features** | **Java** | **Kotlin** |
| --- | --- | --- |
| Object-oriented programming | Not pure OOP | Fully OOP |
| Extension Functions | No | Yes |
| Inline Functions | No | Yes |
| NullPointer Exceptions | Yes | No |
| Lambda Expression | No | Yes |
| Invariant Array | No | Yes |
| Non-private Fields | Yes | No |
| Singletons Objects | Yes | Yes |
| Static Members | Yes | No |

**Question:**  Why should we use Kotlin?

**Answer:** Kotlin is concise, null-safe and interoperable.

**Question:**  Does Kotlin allows to use primitive types such as int, double, float?

**Answer:** No, Kotlin does not allow to use primitive types directly. We can use the class like int, double, float as an object wrapper for primitive but the compiled bytecode has these primitives.

**Question:**  What Is Kotlin? How Is It Interoperable With Java?

**Answer:** Kotlin is a general-purpose JVM-based programming language designed and developed by JetBrains. It’s a statically-typed programming language that supports procedural, functional, and object-oriented programming.

The Kotlin compiler generates the bytecode that is 100% equivalent to the bytecode generated by the Java compiler. Hence, it can seamlessly interoperate with Java.

A few handy features like range expressions, extension functions, null-safety, and smart casts make Kotlin a better alternative over Java.

Additionally, Google promotes Kotlin as the official language for Android development.

**Question:**  What Are the Different Types of Variable Declaration in Kotlin?

**Answer:** In Kotlin, we can either use the val or var keyword followed by an optional data type to declare a variable:

val userName: String = "John"

var age = 25

The val keyword creates a final immutable variable that allows initialization only once. However, var creates a mutable general-purpose variable suitable for multiple assignments.

When we declare a variable without a data type, Kotlin uses the smart cast feature to determine the data type based on the value assignment.

Also, we can use the const keyword along with val to declare compile-time constants.

**Question:**  What’s the Difference Between val and const?

**Answer:** The val keyword creates an immutable variable at runtime, whereas const along with val allows creating a compile-time immutable variable (constant):

val welcomeMsg: String = "Hello $userName"

const val FILE\_EXTENSION: String = ".jpg"

Also, we can declare global or local immutable variables using the val keyword, assignable to a function or any class constructor.

However, we can’t assign it to a function or any class constructor as Kotlin decides the value of the const variable at compile-time.

**Question:**  How to Create Constructors in Kotlin?

**Answer:** There are two types of constructors available in Kotlin – primary and secondary. A Kotlin class can have one primary constructor and one or more secondary constructors.

In Kotlin, we initialize the primary constructor in the class header with a var or val variable declaration:

class Person (var firstName: String, var lastName: String) {

var age: Int = 0

}

var person = Person("John", "Smith");

However, a secondary constructor definition can’t have variable declaration with the var and val keywords:

class Person (var firstName: String, var lastName: String) {

var age: Int = 0

constructor (firstName: String, age: Int) {

this.firstName = firstName

this.age = age

}

}

var personWithAge = Person("John", 27);

Also, a secondary constructor can use the primary constructor internally in the definition:

constructor (firstName: String, lastName: String, age: Int): this(firstName, lastName) {

this.age = age

}

**Question:**  Describe String Interpolation in Kotlin

**Answer:** Kotlin supports two ways of String interpolation – variable substitution using $ and interpolating an expression using ${}. For example:

var welcomeMsg = "Hello $userName";

var secondsInOneYear = "${60 \* 60 \* 24 \* 365} seconds"

**Question:**  What’s Null-Safety in Kotlin?

**Answer:** Null safety in Kotlin guarantees that no object references shall have null or void values.

By default, Kotlin doesn’t allow a variable to have a null value. So, it eliminates the risk of encountering the notorious NullPointerException in the production environment.

However, if required, we can define a variable with the null reference by using ? after the data type:

var name : String?

**Question:**  What Are the Data Classes in Kotlin?

**Answer:** Kotlin encourages code brevity by reducing boilerplate code. For example, the data class provides a way to write concise and readable POJOs.

They eliminate the need to define getters/setters, parameterized constructors, and override methods like hashcode and equals.

All of this is possible in Kotlin by simply adding the data keyword to the class:

data class Person(var firstName: String, var lastName: String)

Furthermore, apart from usual POJO methods like equals, hashCode, and toString, it also creates a few built-in functions like open and componentN.

**Question:**  How to Create a Singleton Class in Kotlin?

**Answer:** Similar to the data class, Kotlin reduces boilerplate code in defining a singleton class by using the object keyword:

object SimpleSingleton

The Kotlin compiler internally implements a singleton class with a thread-safe lazy initialization.

**Question:**  What Is a Double-Bang Operator in Kotlin?

**Answer:** Kotlin’s double bang (!!) operator is a not-null assertion operator. It converts any variable to a non-null type and throws the NullPointerException if it’s null.

So, a developer should use this operator when they are certain for a variable to be not null in any case.

**Question:**  What Are the Range Expressions in Kotlin?

**Answer:** A range is a sequence of values defined by start and end values. Range expressions in Kotlin help to iterate over a range.

Kotlin provides the rangeTo and downTo functions to create a range. Alternatively, we can also use the double dots (..) operator to define a range of values.

For example, let’s define a few ranges in Kotlin:

(i in 1..10) // range from 1 to 10

(i in 1.rangeTo(10)) // range from 1 to 10

(i in 10.downTo(1)) // range from 10 down to 1

(ch in 'a'..'z') // range from a to z

(i in 1..10 step 2) // range from 1 to 10 with a step of 2 (1, 3, 5, 7, 9)

Also, Kotlin allows a few handy functions like reverse, until, first, and last over range expressions for various use cases.

**Question:**  Explain when in Kotlin

**Answer:** Kotlin’s when is an advanced version of Java’s switch-case statements.

When a matching expression is found, Kotlin executes the corresponding code and moves to the next statement:

when (person.firstName) {

"Mike" -> println("hello!")

"John" -> println("howdy!")

else -> println("hi!")

}

Unlike Java, Kotlin doesn’t require a break statement at the end of each case block. Furthermore, Kotlin’s when is useful both as a statement and as an expression.

**Question:**  List All Access Modifiers Available in Kotlin. Which Access Modifier Is Used by Default?

**Answer:** There are four access modifiers available in Kotlin – public, private, protected, and internal.

Similar to Java, the public modifier in Kotlin provides global visibility to the declarations. It’s the default modifier in Kotlin when not declared.

Similarly, the private modifier is the opposite of the public modifier. Hence, it allows accessing the declarations within the file.

However, the protected declarations are secured and not available to the top-level declarations.

Unlike Java, Kotlin has an internal modifier that provides access to the declarations anywhere in the module.

**Question:**  Define open in Kotlin

**Answer:** Unlike Java, which allows extending classes and methods by default, Kotlin uses the open keyword to enable others to inherit them. In other words, it’s opposite to Java’s final keyword.

Kotlin keeps everything final, and through the open keyword, it supports the philosophy of limiting the extension of the classes unless required.

**Question:**  What Are the Advantages of Kotlin Over Java

**Answer:** Kotlin, a JVM programming language, provides tools to write concise, easy, and readable code that’s interoperable with Java.

Furthermore, it’s an official programming language for Andriod development that supports major programming paradigms, like OOP, procedural, and functional.

Also, Kotlin fills a few gaps that Java possesses – null safety, range expressions, string templates, coroutines, inline – infix functions, and smart casts.

**Question:**  Define inline and infix Functions

**Answer:** Kotlin’s inline functions allow the compiler to substitute the body directly into places (inlines) where the function gets called:

inline fun isEven (number: Int): Boolean {

return number % 2 == 0

}

val resultList = (1..10).filter { isEven(it) }

The Kotlin compiler will substitute the body of isEven directly in the filter. Hence, it prevents the object allocations for the functions or lambda expressions. For example:

So, we can write elegant lambdas with the inline keyword without worrying about abstraction and efficiency.

However, infix functions in Kotlin are some functions that can be called without using the period and brackets:

var personDetailsMap = mapOf(

"firtName" to "john",

"lastName" to "smith",

"age" to 25

)

Here, the to method is used as an infix function and results in code that’s more like a natural language – easy to read and understand.

Kotlin provides a variety of infix functions like and, or, and matches. Also, we can define custom infix functions whenever required.

**Question:**  Define lazy and lateinit

**Answer:** lazy means lazy initialization, and lateinit means late initialization.

Kotlin doesn’t initialize a lazy variable before its use. Therefore, it’s supported only by val. A lazy variable supports only a single initialization.

On the other hand, Kotlin delays initialization for a lateinit variable but expects the variable to be initialized before its use. Also, Kotlin throws the UninitializedPropertyAccessException exception without initialization.

It’s supported by var and allows multiple initializations. However, we can’t use lateinit for the primitive types.

A lazy variable can only be initialized from the initializer lambda. However, the lateinit variable allows initialization from anywhere the variable is accessible.

**Question:**  Define fold and reduce Functions

**Answer:** The fold and reduce functions traverse a collection and apply a given operation with and without initial value, respectively.

Let’s take a look at a simple operation using the fold and reduce functions:

var sum = (1..10).fold(10) { res, next -> res + next }

assertEquals(65, sum)

The first iteration of the collection with the fold function uses 10 (initial value) to start with. So, the values of res and next are 10 and 1, respectively.

sum = (1..10).reduce { res, next -> res + next }

assertEquals(55, sum)

However, the iteration starts with 1 and 2 with the reduce function.

Also, the fold function allows changing the result type by defining the data type of the initial value.

**Question:**  What Are the Coroutines in Kotlin?

**Answer:** Coroutines offer lightweight multi-threading programming in Kotlin. Also, we can leverage the suspension and resumption of coroutines mid-execution to write asynchronous programs.

The Kotlin language comes with a rich library called kotlinx-coroutines with a few modules like core, test, debug, and reactive to support multiple features.

**Question:**  What Are the Companion Objects in Kotlin?

**Answer:** Unlike Java, Kotlin doesn’t have static members or member functions. So, we can define a companion object inside the class to execute a method without the class instance.

We can declare a companion object by using the companion keyword:

class Person(var firstName: String, var lastName: String) {

companion object {

val defaultPerson = Person("Mike", "Shaw")

}

// ...

}

assertEquals("Mike", Person.defaultPerson.firstName)

However, companion objects in Kotlin are similar to the standard object with a couple of additional features to ease the development. For example, the name of a companion object is optional, and it offers singleton implementation.

**Question:**  Explain Bitwise Operations in Kotlin

**Answer:** Similar to Java, Kotlin, too, has bitwise operations like or, and, xor, and left and right shifts. However, the operators are named more intuitively than in Java.

For example, the name of Kotlin bit operators are or, and, xor, and inv:

assert(111 or 101 == 111)

assert(111 and 101 == 101)

assert(111 xor 101 == 10)

assert(101.inv() == -102)

Similarly, the binary shift operators are shl – signed left shift, shr – signed right shift, and ushr – unsigned right shift:

assert(3 shl 3 == 24)

assert(12 shr 2 == 3)

assert(-1 ushr 1 == Int.MAX\_VALUE)

**Question:**  Why you should switch to Kotlin from Java?

**Answer:** Kotlin language is quite simple compared to Java. It reduces may redundancies in code as compared to Java. Kotlin can offer some useful features which are not supported by Java.

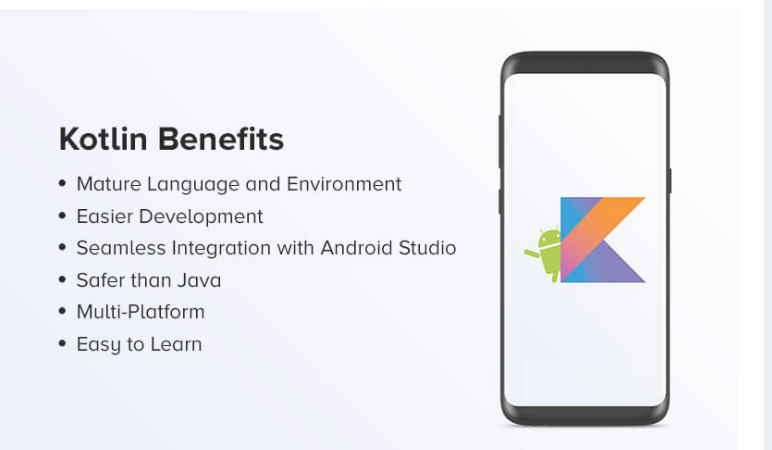
**Question:**  Who is the developer of Kotlin?

**Answer:** Kotlin was developed by JetBrains.

**Question:**  Explain the use of extension functions?

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

**Question:**  Tell three most important benefits of using Kotlin?

**Answer:** 

Benefits of Kotlin

Kotlin language is easy to learn as its syntax is similar to Java.

Kotlin is a functional language and based on JVM. So, it removes lots of boiler plate

It is an expressive language which makes code readable and understandable.

**Question:**  What does ‘Null Safety’ mean in Kotlin?

**Answer:** Null Safety feature allows removing the risk of occurrence of NullPointerException in real time. It is also possible to differentiate between nullable references and non-nullable references.

**Question:**  Is there any Ternary Conditional Operator in Kotlin like in Java?

**Answer:** No there is no ternary conditional operator in Kotlin language.

**Question:**  Why is Kotlin interoperable with Java?

**Answer:** Kotlin is interoperable with Java because it uses JVM bytecode. Compiling it directly to bytecode helps to achieve faster compile time and makes no difference between Java and Kotlin for JVM.

**Question:**  How can you declare a variable in Kotlin?

**Answer:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

value my\_var: Char

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Question:**  How many constructors are available in Kotlin?

**Answer:** Two types of constructors available in Kotlin are:

Primary constructor

Secondary constructor

**Question: Give me name of the extension methods Kotlin provides to java.io.File?**

**Answer:** bufferedReader(): Use for reading contents of a file into BufferedReader

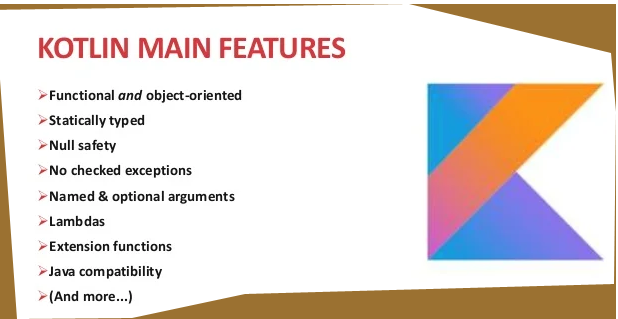
readBytes() : Use for reading contents of file to ByteArray

readText(): Use of reading contents of file to a single String

forEachLine() : Use for reading a file line by line in Kotlin

readLines(): Use to reading lines in file to List

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

**Answer:** 

features

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

Null Safety

Operator Overloading

Coroutines

Range expressions

Smart casts

Companion Objects

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

**Answer:** Elvis Operator is used for handling null expectations in Kotlin.

**Question:**  Does Kotlin allow macros?

**Answer:** No. Kotlin does not offer support for macros because the developers of Kotlin find it difficult to include it in the language.

**Question:**  Does Kotlin support primitive Datatypes?

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

**Question:**  Tell me the default behavior of Kotlin classes?

**Answer:** In Kotlin all classes are final by default. That’s because Kotlin allows multiple inheritances for classes, and an open class is more expensive than a final class.

**Question:**  What is Ranges operator in Kotlin?

**Answer:** Ranges operator helps to iterate through a range. Its operator form is (..) For Example

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

for (i in 1..15)

print(i)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

It will print from 1 to 15 in output.

**Question:**  Give a syntax for declaring a variable as volatile in Kotlin?

**Answer:** Volatile var x: Long? = null

**Question:**  What is the use of abstraction in Kotlin?

**Answer:** Abstraction is the most important concept of Objected Oriented Programming. In Kotlin, abstraction class is used when you know what functionalities a class should have. But you are not aware of how the functionality is implemented or if the functionality can be implemented using different methods.

**Question:**  Differentiate between Kotlin and Java?

**Answer:** Following are the differences between Kotlin and Java:-

| **Basis** | **Kotlin** | **Java** |
| --- | --- | --- |
| **Null Safety** | By default, all sorts of variables in Kotlin are non-nullable (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. |
| **Coroutines Support** | We can perform long-running expensive tasks in several threads in Kotlin, but we also have coroutines support, which halt execution at a given moment without blocking threads while doing long-running demanding operations. | The corresponding thread in Java will be blocked anytime we launch a long-running 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 Classes** | If we need to have data-holding classes in Kotlin, we may define a class with the keyword “data” in the class declaration, and the compiler will take care of everything, including constructing constructors, getter, and setter methods for various fields. | 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() functions are all required to be written explicitly by the developer. |
| **Functional Programming** | Kotlin is procedural and functional programming (a programming paradigm where we aim to bind everything in functional units) language that has numerous useful features such as lambda expressions, operator overloading, higher-order functions, and lazy evaluation, among others. | 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 developers the ability to add new functionality to an existing class. By prefixing the name of a class to the name of the new function, we can build extended 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 declare the type of each variable based on the assignment it will handle in Kotlin. We can specify explicitly if we want to. | When declaring variables in Java, we must declare the type of each variable explicitly. |
| **Smart Casting** | Smart casts in Kotlin will take care of these casting checks with the keyword “is-checks,” which checks for immutable values and conducts implicit casting. | We must examine the type of variables in Java and cast them appropriately for our operation. |
| **Checked Exceptions** | We don’t have checked exceptions in Kotlin. As a result, developers do not need to declare or catch exceptions, which has both benefits and 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. |

**Question:**  What do you understand about function extension in the context of Kotlin? Explain.

**Answer:** 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.

**Question:**  Differentiate between open and public keywords in Kotlin?

**Answer:** 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 public by default. If no visibility modifier is specified, public is used by default, which means our declarations will be accessible everywhere inside the program.

**Question:**  Explain about the “when” keyword in the context of Kotlin?

**Answer:** 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.

**Question:**  What are the basic data types in Kotlin?

**Answer:** Kotlin has both primitive and reference data types. The basic primitive data types are:

Boolean: true or false

Byte: 8-bit signed integer

Short: 16-bit signed integer

Int: 32-bit signed integer

Long: 64-bit signed integer

Float: 32-bit floating point number

Double: 64-bit floating point number

Char: single 16-bit Unicode character

Reference types include String, Array, and user-defined classes. Here are some examples of declaring variables with different data types:

val a: Int = 42

val b: Double = 3.14159

val c: Boolean = true

val d: String = "Hello, world!"

val e: Char = 'A'

val f: Array<Int> = arrayOf(1, 2, 3, 4, 5)

Kotlin also supports type inference, which means that you can omit the type declaration and let the compiler infer the type from the value:

val a = 42 // Inferred type: Int

val b = 3.14159 // Inferred type: Double

val c = true // Inferred type: Boolean

val d = "Hello, world!" // Inferred type: String

val e = 'A' // Inferred type: Char

val f = arrayOf(1, 2, 3, 4, 5) // Inferred type: Array<Int>

**Question:**  How do you declare a variable in Kotlin?

**Answer:** In Kotlin, you can declare a variable using the var or val keyword, followed by the variable name, a colon, and the data type (or you can omit the data type and let the compiler infer it).

The var keyword is used for mutable variables (i.e., those that can be reassigned to a different value later), while val is used for immutable variables (i.e., those that cannot be reassigned once initialized). Here are some examples:

var x: Int = 10 // mutable variable

x = 20 // can be reassigned

val y: String = "Hello" // immutable variable

// y = "World" // error: val cannot be reassigned

You can also omit the data type if you want the compiler to infer it:

var x = 10 // inferred type: Int

val y = "Hello" // inferred type: String

Note that if you omit the data type and initialize the variable to null, you must either provide the data type explicitly or use the ? operator to indicate that the variable can be null:

var z = null // error: type inference failed, cannot determine type of null

var z: String? = null // nullable variable with inferred type: String?

val w: Int? = null // nullable variable with explicit type: Int?

**Question:**  What is the difference between "val" and "var" in Kotlin?

**Answer:** In Kotlin, val and var are used to declare variables. val is short for "value" and is used to declare immutable variables, meaning once they are initialized, they cannot be reassigned to a new value. var is short for "variable" and is used to declare mutable variables, meaning they can be reassigned to a new value after they are initialized. Here are some examples:

val x: Int = 10 // immutable variable

// x = 20 // error: val cannot be reassigned

var y: String = "Hello" // mutable variable

y = "World" // can be reassigned

In general, it's good practice to use val for variables that don't need to be reassigned, since it can help make your code more clear and less prone to bugs.

**Question:**  How do you create a function in Kotlin?

**Answer:** In Kotlin, you can create a function using the fun keyword followed by the function name, parameters (if any), and return type (if any). Here's a simple example of a function that takes two integer parameters and returns their sum:

fun sum(a: Int, b: Int): Int {

return a + b

}

You can then call the function like this:

val result = sum(1, 2) // result = 3

Kotlin also supports default parameter values and named arguments, which can make function calls more concise and expressive:

fun greet(name: String = "World", prefix: String = "Hello"): String {

return "$prefix, $name!"

}

val message1 = greet() // "Hello, World!"

val message2 = greet("Alice") // "Hello, Alice!"

val message3 = greet(prefix = "Hi", name = "Bob") // "Hi, Bob!"

**Question:**  What is null safety in Kotlin?

**Answer:** Null safety is a feature in Kotlin that helps prevent null pointer exceptions by distinguishing between nullable and non-nullable types. When a variable is declared as nullable, it can be assigned a null value, but when a variable is declared as non-nullable, it can never be assigned a null value. Kotlin provides various operators and constructs like the safe call operator, the not-null assertion operator, and the elvis operator to work with nullable types safely. This helps ensure that code is more robust and less prone to runtime exceptions caused by null values.

Example:

var nullableStr: String? = null

var nonNullStr: String = "Hello"

// Safe call operator

val nullableLength = nullableStr?.length

// Not-null assertion operator

val nonNullLength = nonNullStr!!.length

// Elvis operator

val length = nullableStr?.length ?: -1

**Question:**  What are the different types of collections in Kotlin?

**Answer:** In Kotlin, there are several types of collections, including lists, sets, and maps. A List is an ordered collection of elements, a Set is a collection of unique elements with no defined order, and a Map is a collection of key-value pairs. Additionally, there are mutable and immutable versions of each collection type. Here are examples of creating an immutable list, a mutable set, and an immutable map:

// Immutable list

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

// Mutable set

val fruits = mutableSetOf("apple", "orange", "banana")

// Immutable map

val map = mapOf("one" to 1, "two" to 2, "three" to 3)

**Question:**  How do you perform a filter operation on a collection in Kotlin?

**Answer:** To perform a filter operation on a collection in Kotlin, you can use the filter function. This function returns a new collection that contains only the elements that match the given predicate. The predicate is a lambda expression that takes each element of the collection as input and returns a Boolean value. Here is an example code snippet:

val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

val evenNumbers = numbers.filter { it % 2 == 0 }

println(evenNumbers) // prints [2, 4, 6, 8, 10]

In this example, the filter function is used to create a new list that contains only the even numbers from the original list of numbers.

**Question:**  How do you perform a map operation on a collection in Kotlin?

**Answer:** In Kotlin, the map function is used to transform the elements of a collection into a new collection by applying a given function to each element. The resulting collection will have the same size as the original collection.

Here's an example of how to use the map function in Kotlin:

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

val squares = numbers.map { it \* it }

println(squares) // Output: [1, 4, 9, 16, 25]

In the example above, we use the map function to square each element in the numbers list and store the results in the squares list. The lambda function { it \* it } takes each element it in the numbers list and returns its square.

**Question:**  What are higher-order functions in Kotlin?

**Answer:** Higher-order functions in Kotlin are functions that can take other functions as parameters or return functions as results. They are a powerful feature of functional programming that can make code more concise and expressive.

Here's an example of a higher-order function that takes a lambda as a parameter and applies it to each element of a list:

fun <T, R> List<T>.map(transform: (T) -> R): List<R> {

val result = mutableListOf<R>()

for (item in this) {

result.add(transform(item))

}

return result

}

And here's an example of a higher-order function that returns a lambda that adds a given value to its input:

fun add(n: Int): (Int) -> Int {

return { x -> x + n }

}

val addFive = add(5)

val result = addFive(10) // returns 15

**Question:**  How do you define a data class in Kotlin?

**Answer:** In Kotlin, a data class is a special class that is used to hold data. It automatically generates common methods such as equals(), hashCode(), and toString(), based on the properties defined in the class.

To define a data class in Kotlin, simply prefix the class definition with the data keyword, and then list the properties of the class:

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

This data class definition generates the following methods:

equals()

hashCode()

toString()

component1() and component2(), which allow you to destructure instances of the class

You can then create instances of the class as you would with any other class:

val person = Person("Alice", 30)

println(person) // prints "Person(name=Alice, age=30)"

**Question:**  What are the different visibility modifiers in Kotlin?

**Answer:** In Kotlin, there are four visibility modifiers that can be used to control the visibility of classes, interfaces, functions, properties, and constructors:

public: visible everywhere

internal: visible within the same module (i.e., the same compiled unit of code)

protected: visible within the same class and its subclasses

private: visible only within the same class or file

Here are some examples of how to use these modifiers:

class MyClass {

public val publicProperty: Int = 1

internal val internalProperty: Int = 2

protected val protectedProperty: Int = 3

private val privateProperty: Int = 4

public fun publicFunction() {}

internal fun internalFunction() {}

protected fun protectedFunction() {}

private fun privateFunction() {}

constructor() {

// this constructor is public by default

}

internal constructor(x: Int) {

// this constructor is internal

}

protected constructor(x: Int, y: Int) {

// this constructor is protected

}

private constructor(x: Int, y: Int, z: Int) {

// this constructor is private

}

}

Note that the public modifier is optional, as it is the default visibility for anything that is not explicitly marked with a visibility modifier.

**Question:**  What is a property in Kotlin?

**Answer:** In Kotlin, a property is a value that is associated with an instance of a class, or with a class itself. Properties can have a backing field that stores the value, or they can be computed on the fly using getter and setter methods.

Here's an example of a class with properties:

class Person {

var name: String = "John"

val age: Int = 30

}

In this example, name is a mutable property with a backing field that can be set and retrieved using the name property syntax, while age is an immutable property with no backing field, whose value is set in the constructor of the class.

You can then create instances of the class and access their properties like this:

val person = Person()

person.name = "Jane"

println(person.name) // prints "Jane"

println(person.age) // prints "30"

Note that Kotlin also provides a shorthand syntax for defining properties with custom getter and setter methods, using the field keyword to refer to the backing field:

var counter = 0

set(value) {

if (value >= 0) {

field = value

}

}

get() = field + 1

In this example, setting the counter property to a negative value is prevented by the custom setter method, while retrieving the value of the counter property returns the value of the backing field plus one.

**Question:**  How do you perform error handling in Kotlin?

**Answer:** In Kotlin, error handling is typically performed using try, catch, and finally blocks. The try block contains the code that might throw an exception, while the catch block handles any exceptions that are thrown. The finally block is optional and is used to perform any cleanup tasks that should be executed regardless of whether an exception was thrown.

Here's an example of how to use try, catch, and finally:

fun divide(a: Int, b: Int): Int {

try {

return a / b

} catch (e: ArithmeticException) {

println("Division by zero!")

throw e

} finally {

println("Done!")

}

}

try {

val result = divide(10, 0)

println(result)

} catch (e: Exception) {

println("An exception occurred: ${e.message}")

}

In this example, the divide function attempts to perform division between two integers, but if the second integer is zero, it throws an ArithmeticException. The catch block then handles this exception by printing an error message and re-throwing the exception, while the finally block prints a message to indicate that the function is done executing.

Finally, the try block in the main function attempts to call the divide function with arguments 10 and 0, which causes an exception to be thrown. The catch block then handles this exception by printing an error message, while the finally block prints a message to indicate that the program is done executing.

**Question:**  What are extension properties in Kotlin?

**Answer:** In Kotlin, extension properties allow you to add new properties to an existing class without having to modify the class itself. To define an extension property, you use the val or var keyword, followed by the name of the property, the class name you are extending, and a lambda that calculates the value of the property.

Here's an example of how to define an extension property:

val String.firstChar: Char

get() = get(0)

In this example, we define an extension property named firstChar that extends the String class. The lambda passed to the property's get method retrieves the first character of the string.

Once the extension property is defined, you can use it on any instance of the class it extends:

val str = "hello"

val firstChar = str.firstChar // returns 'h'

Note that extension properties can only be defined with val or var keywords and cannot have backing fields. Also, extension properties are not supported for primitive types such as Int or Boolean.

**Question:**  How do you define a generic class in Kotlin?

**Answer:** In Kotlin, you can define a generic class by specifying one or more type parameters in angle brackets (<>) after the class name. These type parameters can then be used as types for properties, methods, and other elements of the class.

Here's an example of how to define a generic class:

class Box<T>(val value: T) {

fun getValue(): T {

return value

}

}

In this example, we define a class named Box that takes a type parameter T, which is used as the type of the value property. The class also has a method named getValue that returns the value property.

Once the generic class is defined, you can create instances of the class for different type arguments:

val boxInt = Box(10)

val intValue = boxInt.getValue() // returns 10

val boxString = Box("hello")

val stringValue = boxString.getValue() // returns "hello"

In this example, we create two instances of the Box class, one with type argument Int and one with type argument String. We then use the getValue method to retrieve the values of the value property for each instance.

**Question:**  What is a reified type parameter in Kotlin?

**Answer:** In Kotlin, a reified type parameter is a type parameter that can be used as a regular type at runtime, rather than being erased by the compiler. This allows you to use the type parameter to access the class object of the parameterized type, which can be useful for reflection and other operations that require access to the type at runtime.

To define a reified type parameter, you must use the inline modifier on the function or class that declares the type parameter, and add the reified keyword before the type parameter name.

Here's an example of how to use a reified type parameter:

inline fun <reified T> printType() {

println(T::class.simpleName)

}

printType<String>() // prints "String"

printType<Int>() // prints "Int"

In this example, we define a function named printType that uses a reified type parameter T. The function uses the ::class syntax to access the class object of the parameterized type, and prints its simple name.

When we call the printType function with different type arguments, the function is able to access the class objects of the parameterized types at runtime, and print their simple names.

**Question:**  How do you define an inline function in Kotlin?

**Answer:** In Kotlin, you can define an inline function by using the inline modifier on the function declaration. An inline function is a function that is expanded at the call site, rather than being executed as a separate function call. This can improve performance by reducing the overhead of function calls, especially for functions that are called frequently.

Here's an example of how to define an inline function:

inline fun measureTimeMillis(block: () -> Unit): Long {

val startTime = System.currentTimeMillis()

block()

return System.currentTimeMillis() - startTime

}

In this example, we define an inline function named measureTimeMillis that takes a lambda function as a parameter. The function measures the time it takes to execute the lambda function, and returns the elapsed time in milliseconds.

When we call the measureTimeMillis function with a lambda function, the function is inlined at the call site, which can improve performance:

val elapsedTime = measureTimeMillis {

// code to be measured

}

In this example, we call the measureTimeMillis function with a lambda function that contains the code to be measured. The measureTimeMillis function is inlined at the call site, and the lambda function is executed inline, without the overhead of a separate function call.

**Question:**  What is a suspension function in Kotlin?

**Answer:** In Kotlin, a suspension function is a function that can be paused and resumed later, allowing it to perform long-running or asynchronous operations without blocking the calling thread. Suspension functions are marked with the suspend keyword, and can use the suspendCoroutine and suspendCancellableCoroutine functions to suspend their execution and return control to the caller.

Here's an example of how to define a suspension function:

suspend fun fetchUrl(url: String): String {

return suspendCoroutine { continuation ->

val client = OkHttpClient()

val request = Request.Builder().url(url).build()

client.newCall(request).enqueue(object : Callback {

override fun onFailure(call: Call, e: IOException) {

continuation.resumeWithException(e)

}

override fun onResponse(call: Call, response: Response) {

val body = response.body?.string()

if (body != null) {

continuation.resume(body)

} else {

continuation.resumeWithException(NullPointerException())

}

}

})

}

}

In this example, we define a suspension function named fetchUrl that uses the OkHttp library to fetch the content of a URL. The function uses the suspendCoroutine function to suspend its execution and return control to the caller while waiting for the response from the server.

When we call the fetchUrl function from a coroutine, the function can be suspended and resumed as necessary, allowing other coroutines to run while the function is waiting for the response from the server.

**Question:**  What is a coroutine scope in Kotlin?

**Answer:** In Kotlin, a coroutine scope is an object that provides a context for launching coroutines. It manages the lifecycle of coroutines and provides a way to cancel them when they are no longer needed. Coroutine scope is an instance of the CoroutineScope interface, and it can be created using the CoroutineScope() constructor or the coroutineScope builder.

Here's an example of how to create a coroutine scope and launch a coroutine within it:

val myScope = CoroutineScope(Dispatchers.Default)

myScope.launch {

// Coroutine code goes here

}

In this example, we create a new coroutine scope using the CoroutineScope constructor and the Dispatchers.Default dispatcher. We then use the launch function to launch a new coroutine within the scope. The coroutine will run on the default dispatcher, which is optimized for CPU-bound tasks.

When the coroutine is no longer needed, we can cancel it using the cancel function:

myScope.cancel()

In this example, we cancel the coroutine scope and all the coroutines launched within it. This will stop any running coroutines and free up any resources associated with the scope.

**Question:**  What is a suspend lambda in Kotlin and how does it work?

**Answer:** In Kotlin, a suspend lambda is a lambda function that can be used with suspension functions. A suspend lambda is a lambda that is marked with the suspend keyword, allowing it to suspend its execution and resume later.

Here's an example of how to define a suspend lambda:

val mySuspendLambda: suspend () -> Unit = {

delay(1000)

println("Hello, world!")

}

In this example, we define a suspend lambda named mySuspendLambda that uses the delay function to suspend its execution for one second before printing a message to the console.

We can use the mySuspendLambda lambda with any coroutine scope, for example:

val myScope = CoroutineScope(Dispatchers.Default)

myScope.launch {

mySuspendLambda()

}

In this example, we launch a new coroutine within the myScope coroutine scope, and call the mySuspendLambda lambda within the coroutine. The lambda will suspend its execution for one second, allowing other coroutines to run, before printing the message to the console.

**Question:**  How do you use the "run" function in Kotlin?

**Answer:** In Kotlin, the run function is a scope function that can be used to execute a code block on an object. It returns the result of the code block and can be used to simplify object initialization and configuration.

Here's an example of how to use the run function:

val myString = "Hello"

val result = myString.run {

println("The length of this string is $length")

this.toUpperCase()

}

println(result)

In this example, we define a string variable myString and call the run function on it. The code block passed to the run function prints the length of the string and returns the uppercased version of the string. The result variable is assigned the result of the run function, which is the uppercased string.

The run function can also be used to initialize objects, for example:

val myObject = SomeClass().run {

name = "John"

age = 30

this

}

In this example, we initialize a new SomeClass object using the run function. We set the object's name and age properties using the this keyword, and return the object. The myObject variable is assigned the initialized SomeClass object.

**Question:**  How do you use the "let" function in Kotlin?

**Answer:** In Kotlin, the let function is a scope function that can be used to execute a code block on an object and return a result. It allows for safe access to an object, even if it is null, and can be used to simplify null checks and reduce code duplication.

Here's an example of how to use the let function:

val myString: String? = "Hello"

val result = myString?.let {

println("The length of this string is ${it.length}")

it.toUpperCase()

}

println(result)

In this example, we define a nullable string variable myString and call the let function on it using the safe-call operator ?.. The code block passed to the let function prints the length of the string and returns the uppercased version of the string. If myString is null, the code block is not executed and the result variable is null.

The let function can also be used to simplify null checks, for example:

val myString: String? = "Hello"

myString?.let { println(it) } ?: run { println("String is null") }

In this example, we call the let function on myString using the safe-call operator ?. to print the string if it is not null. If myString is null, we use the elvis operator ?: to execute a default code block that prints a message to the console.

**Question:**  What is the "lateinit" modifier in Kotlin?

**Answer:** In Kotlin, the lateinit modifier is used to delay the initialization of a non-null variable until later in the code. It is typically used with mutable variables that cannot be initialized in the constructor or immediately upon declaration.

Here's an example of how to use the lateinit modifier:

class MyClass {

lateinit var myString: String

fun initString() {

myString = "Hello"

}

fun printString() {

println(myString)

}

}

val myClass = MyClass()

myClass.initString()

myClass.printString()

In this example, we define a class MyClass with a non-null String variable myString that is marked with the lateinit modifier. We initialize myString in the initString() function and print its value in the printString() function. Since myString is marked as lateinit, we can delay its initialization until later in the code.

When using lateinit variables, it is important to ensure that they are initialized before they are accessed to avoid a lateinit property access exception.

**Question:**  What is the difference between "as" and "as?" in Kotlin?

**Answer:** In Kotlin, as and as? are two different type-casting operators.

The as operator is used for explicit type casting, and will throw a ClassCastException if the casting fails:

val obj: Any = "Hello"

val str: String = obj as String // explicit casting with "as"

// Output: "Hello"

println(str)

The as? operator is used for safe type casting, and will return null if the casting fails:

val obj: Any = 123

val str: String? = obj as? String // safe casting with "as?"

// Output: null

println(str)

In the second example, obj is an Int and cannot be cast to a String, so str is set to null without throwing a ClassCastException.

In general, it is recommended to use as? whenever possible, as it provides a safer and more concise way to perform type casting in Kotlin.

**Question:**  How do you use the "require" function in Kotlin?

**Answer:** In Kotlin, the require function is used to validate a condition and throw an IllegalArgumentException if the condition is not met.

Here is an example of how to use the require function:

fun divide(a: Int, b: Int): Double {

require(b != 0) { "Divisor cannot be zero" }

return a.toDouble() / b.toDouble()

}

// Output: 2.0

println(divide(4, 2))

// This will throw an IllegalArgumentException with the message "Divisor cannot be zero"

divide(4, 0)

In this example, the require function is used to check if the b parameter is not equal to zero. If the condition is not met, an IllegalArgumentException is thrown with the specified message. Otherwise, the function returns the result of the division.

**Question:**  What is the "let" block in Kotlin?

**Answer:** The let block in Kotlin is a scoping function that allows you to execute a block of code on a nullable object without having to use the null-safe operator ?.. The let function takes the nullable object as its receiver and executes the specified block of code with a non-null reference to that object. The result of the block is the return value of the let function.

Here is an example of how to use the let block:

val nullableString: String? = "Hello, World!"

nullableString?.let { str ->

// This block will only be executed if nullableString is not null

println(str.toUpperCase()) // Output: "HELLO, WORLD!"

}

In this example, the let block is used to perform an operation on a nullable String object. The block is only executed if the object is not null. The str variable inside the block is a non-null reference to the nullable String object, which can be safely operated on without having to use the null-safe operator ?..

**Question:**  How do you use the "also" function in Kotlin?

**Answer:** The also function in Kotlin is a scoping function that allows you to perform some additional actions on an object within a block of code. The also function takes the object as its receiver and returns the same object after executing the specified block of code.

Here is an example of how to use the also function:

val myNumber = 42

val result = myNumber.also {

// This block will execute before the object is returned

println("The value of myNumber is $it")

}

println("The result is $result") // Output: "The result is 42"

In this example, the also block is used to print the value of myNumber before it is returned. The it variable inside the block refers to the object that is being operated on. The also function returns the original object (myNumber), allowing you to chain additional function calls onto it.

**Question:**  What is the "const" modifier in Kotlin?

**Answer:** The const modifier in Kotlin is used to declare compile-time constants. A const property must be a top-level or a member property with a primitive type or a String type, and its value must be known at compile time.

Here's an example of how to use the const modifier in Kotlin:

const val PI = 3.14159

const val MY\_STRING = "Hello, world!"

class MyClass {

companion object {

const val MY\_CONSTANT = "my\_constant\_value"

}

}

In this example, PI and MY\_STRING are top-level const properties, while MY\_CONSTANT is a const property declared in a companion object. These values are known at compile-time and cannot be changed at runtime. Note that const properties can only be of primitive or String types, and cannot have a custom getter or setter.

**Question:**  What is the "by lazy" delegate in Kotlin?

**Answer:** The by lazy delegate in Kotlin is used to create a lazy-initialized property. It allows you to defer the initialization of a property until it's actually accessed for the first time, and then reuse the same value on subsequent accesses.

Here's an example of how to use the by lazy delegate in Kotlin:

val myLazyValue: String by lazy {

println("Lazy value is being computed")

"Hello, World!"

}

fun main() {

println("Before accessing lazy value")

println(myLazyValue)

println("After accessing lazy value")

println(myLazyValue)

}

In this example, myLazyValue is a property that's initialized using the by lazy delegate. The lambda expression passed to the lazy function is used to compute the value of the property on the first access. When the program is run, the output will be:

Before accessing lazy value

Lazy value is being computed

Hello, World!

After accessing lazy value

Hello, World!

As you can see, the println statement inside the lambda expression is only executed once, when the property is first accessed. On subsequent accesses, the same value is returned without recomputing the initialization logic.

**Question:**  What is the difference between a suspend function and a blocking function, and when should you use each?

**Answer:** In Kotlin, a suspend function is a function that can be paused and resumed later, while a blocking function will block the calling thread until it completes. Suspend functions are typically used for asynchronous operations, such as making network requests, and allow other code to continue executing while waiting for the operation to complete. Blocking functions are typically used for synchronous operations, such as reading from a file, and should only be used on background threads to avoid blocking the UI thread. Here is an example of a suspend function and a blocking function:

// Suspend function

suspend fun fetchUserData(): List<User> {

delay(1000) // Pause for 1 second

return apiService.getUsers()

}

// Blocking function

fun readTextFromFile(file: File): String {

return file.readText()

}

**Question:**  How do you properly use the "when" expression in Kotlin when you need to return a value?

**Answer:** In Kotlin, the when expression is similar to a switch statement in other languages, but it allows for more complex and flexible conditions. When using when to return a value, the else branch is mandatory to handle any remaining cases. For example:

fun checkValue(value: Int): String {

return when (value) {

0 -> "Value is zero"

1 -> "Value is one"

2 -> "Value is two"

else -> "Value is not zero, one or two"

}

}

In this example, the when expression checks the value of the input parameter value, and returns a string depending on its value. The else branch is used to handle any values that don't match the other cases.

**Question:**  What is the difference between "override" and "override fun" when implementing an interface in Kotlin?

**Answer:** In Kotlin, when implementing an interface, the override keyword is used to indicate that a property or method is being overridden. When overriding a method, override fun is used to define the function with its implementation.

For example, consider the following interface and class:

interface MyInterface {

fun myMethod()

}

class MyClass : MyInterface {

override fun myMethod() {

// implementation

}

}

In this example, the override keyword is used to indicate that the myMethod() method is being overridden in the MyClass class, and override fun is used to define the function with its implementation.

**Question:**  How do you ensure thread-safety when working with shared mutable state in Kotlin?

**Answer:** To ensure thread-safety when working with shared mutable state in Kotlin, you can use synchronization constructs such as the synchronized keyword, @Synchronized annotation, or the Mutex interface. You can also use atomic variables and non-blocking algorithms such as AtomicReference, AtomicInteger, AtomicBoolean, and ConcurrentHashMap. Here is an example of using synchronized:

class Counter {

private var count = 0

@Synchronized

fun increment() {

count++

}

@Synchronized

fun decrement() {

count--

}

fun getCount() = count

}

In this example, the increment and decrement methods are marked with the @Synchronized annotation, which ensures that only one thread can execute these methods at a time, preventing race conditions and maintaining thread-safety.

**Question:**  How do you properly implement a custom iterator in Kotlin?

**Answer:** To implement a custom iterator in Kotlin, you need to define a class that implements the Iterator interface and overrides its next and hasNext methods. The next method should return the next element in the iteration, and the hasNext method should return true if there are more elements to be iterated over.

Here's an example of how to implement a custom iterator for a simple list of integers:

class IntListIterator(private val intList: List<Int>) : Iterator<Int> {

private var currentIndex = 0

override fun hasNext(): Boolean {

return currentIndex < intList.size

}

override fun next(): Int {

val nextValue = intList[currentIndex]

currentIndex++

return nextValue

}

}

fun main() {

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

val iterator = IntListIterator(intList)

while (iterator.hasNext()) {

val value = iterator.next()

println(value)

}

}

In this example, we create a custom iterator class IntListIterator that takes a list of integers as a parameter. The hasNext method returns true if the current index is less than the size of the list, and the next method returns the current value and increments the current index. We then create an instance of the iterator and use it to loop over the list, printing each value as we go.

**Question:**  How do you use sealed classes to define state machines in Kotlin?

**Answer:** Sealed classes are often used to define state machines in Kotlin because they provide a way to define a restricted hierarchy of classes that can be used for representing a state machine's states. Each state can be represented by a subclass of the sealed class, and transitions between states can be modeled by returning a new instance of a subclass representing the next state. Here's an example:

sealed class State {

object Idle : State()

object Loading : State()

data class Loaded(val result: String) : State()

}

class StateMachine {

var currentState: State = State.Idle

fun fetchData() {

when (currentState) {

State.Idle -> {

currentState = State.Loading

// Perform network request to load data

currentState = State.Loaded("Data loaded")

}

State.Loading -> {

// Do nothing

}

is State.Loaded -> {

// Data already loaded

}

}

}

}

In this example, the State sealed class represents the possible states of a state machine, and the StateMachine class transitions between states by updating the currentState property.

**Question:**  What is the difference between "listOf" and "mutableListOf" in Kotlin?

**Answer:** In Kotlin, listOf creates an immutable list, while mutableListOf creates a mutable list. Once an immutable list is created, its elements cannot be added, removed or modified. On the other hand, a mutable list can be modified. Here is an example:

val immutableList = listOf("apple", "banana", "cherry")

val mutableList = mutableListOf("apple", "banana", "cherry")

// This will not compile because the list is immutable

immutableList.add("orange")

// This will work because the list is mutable

mutableList.add("orange")

In general, it is recommended to use immutable collections whenever possible to avoid accidental modification of the collection.

**Question:**  How do you implement the "builder" pattern in Kotlin?

**Answer:** The "builder" pattern is a creational design pattern used to create complex objects by separating the construction of an object from its representation. In Kotlin, this pattern can be implemented by using the apply function and a builder class with default values for the required properties. Here's an example:

class PersonBuilder {

var name: String = ""

var age: Int = 0

var address: String = ""

fun build(): Person {

return Person(name, age, address)

}

}

data class Person(val name: String, val age: Int, val address: String)

val person = PersonBuilder().apply {

name = "John Doe"

age = 30

}.build()

In this example, the PersonBuilder class has properties for each of the required properties of the Person class, and a build() method that returns a Person object with those properties. The apply function is used to set the properties of the builder object and return it, allowing for a fluid and readable syntax for constructing objects.

**Question:**  What is the difference between "unit" and "void" in Kotlin?

**Answer:** In Kotlin, Unit is a type that corresponds to the void type in Java and other programming languages. Both Unit and void indicate that a function does not return a value. However, Unit is an actual object that can be used, whereas void is simply a placeholder. For example, a function that returns Unit can be used in a functional expression, while a function with a void return type cannot. Here's an example in Kotlin:

// A function that returns Unit

fun printHello() {

println("Hello!")

}

// A function with void return type in Java

void printHello() {

System.out.println("Hello!");

}

**Question:**  How do you use "by" to implement delegates in Kotlin?

**Answer:** In Kotlin, you can use the by keyword to implement property delegation. This allows you to delegate the implementation of a property to another object. To use by, define a class that implements the ReadOnlyProperty or ReadWriteProperty interface and pass an instance of that class to the property being delegated. Here's an example of using by to implement a lazy property:

class Example {

val lazyProperty: String by lazy {

println("computed!")

"Hello"

}

}

fun main() {

val example = Example()

println(example.lazyProperty)

println(example.lazyProperty)

}

In this example, the lazyProperty property is delegated to a Lazy instance that computes its value lazily when it is first accessed. The println("computed!") statement is only executed once, the first time the property is accessed, demonstrating the lazy evaluation of the property.

**Question:**  What are the different types of functions in Kotlin?

**Answer:** In Kotlin, there are several types of functions:

Top-level functions: These are functions declared outside of any class or object.

Member functions: These are functions declared within a class or object.

Extension functions: These are functions that extend the functionality of an existing class without modifying its source code.

Higher-order functions: These are functions that can take other functions as parameters or return functions as results.

Lambda functions: These are anonymous functions that can be passed as arguments to higher-order functions.

Here are some examples:

// top-level function

fun sum(a: Int, b: Int): Int {

return a + b

}

// member function

class MyClass {

fun greet() {

println("Hello!")

}

}

// extension function

fun String.reverse(): String {

return this.reversed()

}

// higher-order function

fun operation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {

return operation(a, b)

}

// lambda function

val multiply: (Int, Int) -> Int = { a, b -> a \* b }

**Question:**  What is a lambda expression in Kotlin?

**Answer:** A lambda expression is an anonymous function that can be passed as a parameter to other functions or stored in a variable. In Kotlin, lambda expressions are surrounded by curly braces and can have optional parameters and a return type. They are commonly used in functional programming and for defining callbacks in event-driven programming. Here is an example of a lambda expression that multiplies two numbers:

val multiply = { a: Int, b: Int -> a \* b }

val result = multiply(2, 3) // result is 6

In this example, multiply is a lambda expression that takes two Int parameters a and b and returns their product. The result variable is assigned the result of calling multiply with arguments 2 and 3.In Kotlin, lazy initialization is performed using the lazy delegate, which creates a lazily initialized property that is only initialized when it is first accessed. To use lazy, simply declare the property with the by lazy syntax and provide a lambda that initializes the property when it is accessed for the first time. Here's an example:

val myLazyProperty: String by lazy {

// Perform expensive initialization here

"Hello, World!"

}

In this example, the myLazyProperty property will be lazily initialized with the string "Hello, World!" the first time it is accessed. Subsequent accesses will return the same value without re-initializing.

**Question:**  How do you perform lazy initialization in Kotlin?

**Answer:** In Kotlin, lazy initialization is performed using the lazy delegate, which creates a lazily initialized property that is only initialized when it is first accessed. To use lazy, simply declare the property with the by lazy syntax and provide a lambda that initializes the property when it is accessed for the first time. Here's an example:

val myLazyProperty: String by lazy {

// Perform expensive initialization here

"Hello, World!"

}

In this example, the myLazyProperty property will be lazily initialized with the string "Hello, World!" the first time it is accessed. Subsequent accesses will return the same value without re-initializing.

**Question:**  What is a delegated property in Kotlin?

**Answer:** In Kotlin, a delegated property is a property that delegates its getters and setters to another object. This can help reduce boilerplate code and provide additional functionality to properties.

One common use of a delegated property is the lazy delegate, which delays the initialization of the property until it is first accessed. Another example is the observable delegate, which allows you to observe changes to the property value.

Here is an example of using the lazy delegate to perform lazy initialization:

val myLazyValue: String by lazy {

// This code block will be executed only once, when the value is first accessed

"Hello, World!"

}

In this example, the myLazyValue property is declared with the lazy delegate, and its value is initialized by the lambda expression provided. The lambda expression will be executed only once, when the property is first accessed, and the result will be cached and returned for subsequent accesses.

**Question:**  What is a sealed class in Kotlin?

**Answer:** A sealed class in Kotlin is used to represent restricted class hierarchies, where all the subclasses are known in advance. It is defined with the "sealed" modifier and can only be subclassed within the same file where it is declared. Sealed classes are often used in combination with when expressions to create powerful and safe handling of complex data types.

Here is an example of a sealed class in Kotlin:

sealed class Result

class Success(val data: String) : Result()

class Error(val error: Throwable) : Result()

fun handleResult(result: Result) {

when (result) {

is Success -> println(result.data)

is Error -> println(result.error.message)

}

}

In this example, Result is a sealed class with two subclasses, Success and Error. The handleResult function uses a when expression to handle the different types of Result. Since Result is sealed, the compiler knows that all possible subclasses of Result are covered in the when expression and will give a warning if any new subclass is introduced outside of the current file.

**Question:**  How do you define a DSL in Kotlin?

**Answer:** A DSL (Domain-Specific Language) in Kotlin can be defined using function literals with receiver (receiver function literals) and extension functions. It allows for a more natural and expressive syntax for a specific use case. The DSL can be created by defining an object with extension functions, allowing the DSL to be called as a series of function calls on an instance of that object. Here is an example of a simple DSL that generates an HTML table:

class HtmlTable {

private val rows = mutableListOf<List<String>>()

fun tr(init: HtmlTableRow.() -> Unit) {

val row = HtmlTableRow()

row.init()

rows.add(row.cells)

}

override fun toString(): String {

return buildString {

append("<table>")

rows.forEach { row ->

append("<tr>")

row.forEach { cell ->

append("<td>$cell</td>")

}

append("</tr>")

}

append("</table>")

}

}

}

class HtmlTableRow {

val cells = mutableListOf<String>()

fun td(cell: String) {

cells.add(cell)

}

}

fun htmlTable(init: HtmlTable.() -> Unit) = HtmlTable().apply(init)

// Usage

val table = htmlTable {

tr {

td("1")

td("2")

td("3")

}

tr {

td("4")

td("5")

td("6")

}

}

println(table)

In this example, HtmlTable and HtmlTableRow are defined as classes that represent the DSL, and the htmlTable function is used to create an instance of HtmlTable and apply the DSL to it. The tr and td functions represent table rows and cells, respectively, and can be nested to create a table with arbitrary numbers of rows and columns.

**Question:**  What is type projection in Kotlin?

**Answer:** Type projection in Kotlin is a way to specify a type with an unknown generic argument. This is useful when the generic argument is not relevant or known, but a specific instance of the type is required. To project the type, the \* character is used as a wildcard for the generic argument. For example, to create a list of any type, we can use List<\*>. We can also use type projection when dealing with nested types, as in Map<String, List<\*>>, where we don't care about the generic argument of the nested List.

Example:

val list: List<\*> = listOf(1, "two", true)

val map: Map<String, List<\*>> = mapOf(

"numbers" to listOf(1, 2, 3),

"strings" to listOf("a", "b", "c")

)

**Question:**  How do you implement coroutines in Kotlin?

**Answer:** Coroutines are a lightweight concurrency design pattern in Kotlin that can simplify writing asynchronous code. To use coroutines in Kotlin, you need to add the kotlinx.coroutines dependency to your project. You can then create and launch a coroutine using the launch function, passing in a suspend function as a lambda. Within the suspend function, you can use the delay function to pause the coroutine, or use async to create a deferred result. Here's an example of launching a coroutine:

import kotlinx.coroutines.\*

fun main() {

GlobalScope.launch {

delay(1000L)

println("Hello from coroutine!")

}

println("Hello from main thread!")

Thread.sleep(2000L)

}

This code creates a coroutine that waits for one second before printing a message, and also prints a message from the main thread. The Thread.sleep call is needed to keep the main thread running until the coroutine has finished executing.

**Question:**  What is a suspend function in Kotlin?

**Answer:** A suspend function in Kotlin is a function that can be paused and resumed later without blocking the main thread. It is used to perform long-running or asynchronous operations, such as network calls or file I/O, without blocking the UI. Suspend functions are marked with the suspend keyword and can only be called from other suspend functions or from a coroutine. Here's an example of a suspend function that performs a network call using the kotlinx.coroutines library:

suspend fun fetchUserData(userId: String): User {

return withContext(Dispatchers.IO) {

// perform network call on IO thread

apiService.getUser(userId)

}

}

**Question:**  How do you perform type-safe builders in Kotlin?

**Answer:** Type-safe builders in Kotlin allow for creating domain-specific languages (DSLs) with a strongly-typed syntax. To implement a type-safe builder, one can define a builder class with functions that modify its state and return itself. These functions can then be called in a DSL-like fashion, providing a more concise and expressive syntax. Type-safe builders are commonly used for creating XML or HTML documents. Here's an example of how to define and use a type-safe builder in Kotlin:

class PersonBuilder {

var name: String = ""

var age: Int = 0

fun name(value: String) {

name = value

}

fun age(value: Int) {

age = value

}

fun build(): Person {

return Person(name, age)

}

}

fun person(build: PersonBuilder.() -> Unit): Person {

val builder = PersonBuilder()

builder.build()

return builder.build()

}

val john = person {

name = "John"

age = 30

}

In this example, the person function takes a lambda with a receiver of PersonBuilder, allowing the caller to configure the builder with a DSL-like syntax. The PersonBuilder class has functions for modifying its state and returning itself, and a build function that constructs a Person object from the builder's state.

**Question:**  What is a suspend coroutine in Kotlin?

**Answer:** A suspend coroutine is a coroutine that can be paused and resumed at certain points without blocking the calling thread. It allows for asynchronous programming without the complexity of callback-based APIs or blocking threads. Suspend coroutines are defined with the suspend keyword, and can be launched with launch or async functions from the kotlinx.coroutines library. They can also be used with withContext to switch to a different coroutine context for a specific block of code. Here is an example of a simple suspend function:

suspend fun fetchData(url: String): String {

val response = withContext(Dispatchers.IO) {

// Perform a network request on the IO dispatcher

URL(url).readText()

}

return response

}

**Question:**  How do you implement a custom operator in Kotlin?

**Answer:** To implement a custom operator in Kotlin, you can use the operator modifier and define a function with a special name that represents the operator. The function can then be used like any other operator in Kotlin. Here's an example that defines a custom + operator for a custom Complex class:

class Complex(val real: Double, val imaginary: Double) {

operator fun plus(other: Complex): Complex {

return Complex(real + other.real, imaginary + other.imaginary)

}

}

// usage

val c1 = Complex(1.0, 2.0)

val c2 = Complex(3.0, 4.0)

val c3 = c1 + c2 // c3 is now Complex(4.0, 6.0)

In this example, the Complex class defines a custom plus function with the operator modifier, which allows the + operator to be used with instances of Complex. When the + operator is used with two instances of Complex, it calls the plus function to compute the sum of the two complex numbers.

**Question:**  What is a higher-order extension function in Kotlin?

**Answer:** A higher-order extension function is an extension function that takes a function as a parameter or returns a function. It allows you to extend the behavior of an existing class by adding new functions. You can use this pattern to create reusable code and improve code readability. Here's an example of a higher-order extension function that applies a function to a list:

fun <T> List<T>.applyAll(function: (T) -> T): List<T> {

return this.map { function(it) }

}

val list = listOf(1, 2, 3)

val result = list.applyAll { it \* 2 }

// result = [2, 4, 6]

In this example, the applyAll function is a higher-order extension function because it takes a lambda function function as a parameter. The lambda is applied to each element of the list using the map function.

**Question:**  How do you implement a DSL in Kotlin using function literals with receivers?

**Answer:** To create a DSL in Kotlin using function literals with receivers, you define a function that takes a receiver as a parameter, which then enables you to manipulate the receiver's properties and methods within a block of code. This allows for a more natural and intuitive syntax for the DSL. The receiver can be any class, and the function can be named anything. Here's an example of a simple DSL that creates an HTML table:

class HtmlTable {

val rows = mutableListOf<HtmlRow>()

fun row(block: HtmlRow.() -> Unit) {

val row = HtmlRow()

row.block()

rows.add(row)

}

}

class HtmlRow {

val cells = mutableListOf<HtmlCell>()

fun cell(block: HtmlCell.() -> Unit) {

val cell = HtmlCell()

cell.block()

cells.add(cell)

}

override fun toString() = "<tr>${cells.joinToString("")}</tr>"

}

class HtmlCell {

var text = ""

override fun toString() = "<td>$text</td>"

}

fun table(block: HtmlTable.() -> Unit): HtmlTable {

val table = HtmlTable()

table.block()

return table

}

val htmlTable = table {

row {

cell {

text = "1"

}

cell {

text = "2"

}

cell {

text = "3"

}

}

row {

cell {

text = "4"

}

cell {

text = "5"

}

cell {

text = "6"

}

}

}

println(htmlTable)

In this example, we define three classes: HtmlTable, HtmlRow, and HtmlCell. We also define a function table that takes a lambda with receiver of type HtmlTable as a parameter. This function creates an instance of HtmlTable, executes the lambda on it, and returns the resulting object. The DSL itself is defined in the lambda passed to table, which allows for a natural and expressive way of creating an HTML table.

**Question:**  What is the inline class in Kotlin?

**Answer:** The inline class is a feature introduced in Kotlin 1.3 that allows the creation of lightweight object wrappers with little runtime overhead. Inline classes can only have a single property, and the class is compiled as if the property were used directly. This allows for better performance than creating a full class, while still retaining the type safety and organization that classes provide.

Here's an example of an inline class:

inline class Username(val value: String)

This defines an inline class called Username that wraps a single String property value. The class can be used like a regular class, but the compiler will optimize away the extra overhead.

**Question:**  What is a property delegate in Kotlin?

**Answer:** In Kotlin, a property delegate is a way to extract common property operations, such as getting and setting, and define them in a separate class. This can help make code more reusable, modular, and concise. Delegates are defined using the by keyword and a delegate instance. There are several built-in delegate types in Kotlin, such as lazy, observable, and vetoable. Additionally, developers can create their own custom property delegates by implementing the ReadOnlyProperty or ReadWriteProperty interfaces. Here is an example of using the lazy delegate to perform lazy initialization of a property:

val myLazyValue: String by lazy {

// perform complex initialization

"initialized value"

}

**Question:**  What are flow builders in Kotlin?

**Answer:** In Kotlin, Flow builders are functions used to create a new flow with an emitter. They allow the creation of asynchronous streams of values that can be collected by a consumer. The standard library provides a set of flow builders such as flow, flowOf, channelFlow, and callbackFlow. flow creates a flow that emits values synchronously, flowOf creates a flow that emits a fixed set of values, channelFlow creates a flow with a channel, and callbackFlow creates a flow that integrates with callback-based APIs. Here is an example using the flowOf builder:

fun main() = runBlocking {

flowOf(1, 2, 3).collect { println(it) }

}

**Question:**  What is the difference between a sequence and a flow in Kotlin?

**Answer:** In Kotlin, a sequence is a collection of elements that can be iterated one at a time lazily, whereas a flow is a collection of elements that can be computed asynchronously in a suspending manner.

When an element is requested from a sequence, it's computed immediately, whereas a flow emits elements asynchronously, so it can be paused and resumed. This makes flows useful for handling long-running or potentially blocking operations, such as reading from a network socket.

Here is an example of creating a sequence and a flow in Kotlin:

// Create a sequence of numbers

val sequence = sequence {

for (i in 1..5) {

yield(i)

}

}

// Create a flow of numbers

val flow = flow {

for (i in 1..5) {

delay(100) // Simulate a long-running operation

emit(i)

}

}

In the example above, sequence is a sequence of integers from 1 to 5, whereas flow is a flow of integers from 1 to 5, emitted every 100ms.

**Question:**  How do you implement a suspendable sequence in Kotlin?

**Answer:** Kotlin doesn't provide a built-in "suspendable sequence" type. However, you can create a suspendable sequence using a combination of a Sequence and a suspend function. Here is an example of how you can create a suspendableSequence function that returns a sequence of values suspended for a given delay:

import kotlinx.coroutines.delay

suspend fun suspendableSequence() = sequence {

var i = 0

while (true) {

delay(1000)

yield(i++)

}

}

This function creates a Sequence using the sequence builder function and suspends the execution of the coroutine for 1 second using the delay function. The yield function is used to produce the values of the sequence. The suspendableSequence function returns a Sequence of integers, where each element of the sequence is produced after a delay of 1 second.

You can use this function to create a Flow by calling the asFlow() function on the sequence:

import kotlinx.coroutines.flow.flow

fun flowFromSequence() = flow {

for (i in suspendableSequence()) {

emit(i)

}

}

This function returns a Flow of integers that are produced by calling the suspendableSequence function. Each value of the suspendableSequence is emitted as a new value of the Flow.

**Question:**  How do you use the Kotlin compiler plugins to perform static analysis of code?

**Answer:** Kotlin compiler plugins can be used to analyze and transform the Kotlin code during compilation. By implementing the ComponentRegistrar interface, you can provide your own compiler plugin to the Kotlin compiler. You can use AnalysisHandlerExtension to analyze code and find issues such as unreachable code, dead code, and more. Here's an example of how to use the AnalysisHandlerExtension to detect the use of the print function:

class PrintUsageDetector : AnalysisHandlerExtension {

override fun analysisCompleted(project: Project, module: ModuleDescriptor, bindingTrace: BindingTrace, files: Collection<KtFile>) {

val printUsageVisitor = object : KtTreeVisitorVoid() {

override fun visitCallExpression(expression: KtCallExpression) {

if (expression.calleeExpression?.text == "print") {

println("Detected usage of print at ${expression.text}")

}

super.visitCallExpression(expression)

}

}

files.forEach { file -> file.accept(printUsageVisitor) }

}

}

fun main() {

val configuration = CompilerConfiguration()

configuration.addJvmPlugin(PrintUsageDetector())

val environment = KotlinCoreEnvironment.createForProduction(

ClassPath.EMPTY, configuration, EnvironmentConfigFiles.JVM\_CONFIG\_FILES

)

val code = """

fun main() {

print("Hello, world!")

}

""".trimIndent()

val file = KtFile(

KotlinParser().createFileBuilder(KotlinLanguage.INSTANCE, code).build(),

true

)

val ktFiles = listOf(file)

TopDownAnalyzerFacadeForJVM.analyzeFilesWithJavaIntegration(

environment.project, ktFiles, JvmPlatformParameters.defaultInstance(),

environment.configuration, { \_, \_ -> }, { \_, \_ -> }

)

}

In this example, we define a PrintUsageDetector class that implements the AnalysisHandlerExtension interface. In the analysisCompleted method, we use a visitor to traverse the abstract syntax tree of each file and detect calls to the print function. We then print a message indicating where the usage was detected.

We then create a CompilerConfiguration object and add our plugin to it. We create a KotlinCoreEnvironment and pass in the configuration. Finally, we create a KtFile object from some sample code, analyze it using TopDownAnalyzerFacadeForJVM, and our plugin will be executed, detecting the usage of the print function.

**Question:**  What is a type-safe builder in Kotlin and how does it work?

**Answer:** A type-safe builder is a way to create complex object structures with a domain-specific language (DSL) in a type-safe way. It works by creating a builder class that exposes methods to construct the object with a fluent API, using lambdas to define the DSL. The builder class enforces the types of the data being built, and the DSL provides a more intuitive and expressive way to configure the object. Here's an example of a type-safe builder for a Person class:

class PersonBuilder {

var firstName: String? = null

var lastName: String? = null

var age: Int? = null

fun build(): Person {

return Person(firstName!!, lastName!!, age!!)

}

}

fun person(block: PersonBuilder.() -> Unit): Person {

val builder = PersonBuilder()

builder.block()

return builder.build()

}

data class Person(val firstName: String, val lastName: String, val age: Int)

val p = person {

firstName = "John"

lastName = "Doe"

age = 30

}

In this example, the person function is the entry point to the type-safe builder DSL. It creates a new PersonBuilder instance and applies the configuration lambda to it using the block argument. The PersonBuilder class has var properties for the Person fields, and a build() method that returns the final Person object. The DSL lambda sets the PersonBuilder properties, and the types are enforced by the PersonBuilder class. Finally, the person function returns the fully configured Person object.

**Question:**  How do you use the "coerceAtMost" and "coerceAtLeast" functions in Kotlin?

**Answer:** The coerceAtMost and coerceAtLeast are standard library functions in Kotlin used to ensure that a value falls within a given range.

coerceAtMost returns the value that is equal to or less than a given maximum, while coerceAtLeast returns the value that is equal to or greater than a given minimum. If the original value is already within the specified range, then it is returned without modification.

Here are some code snippets that illustrate the usage of these functions:

// Using coerceAtMost

val max = 100

val value = 150

val coercedValue = value.coerceAtMost(max)

println(coercedValue) // prints 100

// Using coerceAtLeast

val min = 10

val value2 = 5

val coercedValue2 = value2.coerceAtLeast(min)

println(coercedValue2) // prints 10

In the first example, coerceAtMost ensures that value is not greater than the maximum max by returning the maximum value. In the second example, coerceAtLeast ensures that value2 is not less than the minimum min by returning the minimum value.

**Question:**  What is the difference between a function and a lambda in Kotlin?

**Answer:** In Kotlin, a function is a named block of code that can be called with a specified set of parameters, while a lambda is an anonymous function that can be passed as a parameter to another function. Lambdas are often used to simplify code and make it more concise. One key difference between functions and lambdas is that lambdas are not defined with a name, whereas functions are. Another difference is that a lambda can be used as an argument in a higher-order function, which is not possible with a regular function.

Here is an example of a function and a lambda in Kotlin:

// Function definition

fun add(x: Int, y: Int): Int {

return x + y

}

// Lambda definition

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

In this example, add() is a function that takes two integer parameters and returns their sum. addLambda is a lambda that has the same behavior as add().

**Question:**  How do you use the "invoke" operator in Kotlin?

**Answer:** In Kotlin, the invoke operator is used to invoke an instance of a class as if it were a function. To use the invoke operator, you need to define the operator fun invoke() function in your class. Here is an example:

class Greeter(val name: String) {

operator fun invoke() {

println("Hello, $name!")

}

}

fun main() {

val greeter = Greeter("John")

greeter() // same as greeter.invoke()

}

In the example, the Greeter class defines the invoke() function, which takes no arguments and prints a greeting message using the name property of the class. The main() function creates an instance of Greeter and invokes it using the () operator, which is the same as calling the invoke() function on the instance.

**Question:**  How do you use the "reified" modifier in Kotlin?

**Answer:** In Kotlin, the reified modifier can be used to indicate that a type parameter should be available at runtime, instead of being erased during compilation. This allows for more flexible and type-safe code. reified can only be used with inline functions.

Here's an example of using reified to check if an object is of a certain type at runtime:

inline fun <reified T> isInstanceOf(obj: Any): Boolean {

return obj is T

}

val str = "Hello, World!"

println(isInstanceOf<String>(str)) // true

println(isInstanceOf<Int>(str)) // false

Note that the is operator can't be used to check if an object is of a type determined at runtime. Using reified and inline solves this problem.

**Question:**  What is the "when" block in Kotlin?

**Answer:** The when expression in Kotlin is a powerful alternative to the switch statement in Java. It allows you to check multiple conditions against a single value and execute different blocks of code based on the matching condition. It can be used as an expression or a statement. The when block supports a wide range of matching options, including checking for specific values, ranges, and types. Here's an example of using when as an expression:

fun getDayOfWeek(day: Int): String {

return when (day) {

1 -> "Monday"

2 -> "Tuesday"

3 -> "Wednesday"

4 -> "Thursday"

5 -> "Friday"

else -> "Weekend"

}

}

In this example, the when block checks the value of the day parameter and returns the corresponding day of the week. If the value doesn't match any of the defined cases, it returns "Weekend".

**Question:**  What is the "requireNotNull" function in Kotlin?

**Answer:** The requireNotNull function is a Kotlin standard library function that throws an IllegalArgumentException if the specified value is null. It is a convenient way to ensure that a non-null value is used in code that follows. The function takes a nullable value and an optional message, and returns the value if it is not null. Otherwise, it throws an exception with the specified message. Here is an example:

fun example(str: String?) {

val s: String = requireNotNull(str) { "Expected a non-null value" }

// Use s in the rest of the function

}

**Question:**  How do you use the "use" function in Kotlin?

**Answer:** The use function is used in Kotlin to manage resources that need to be closed after usage. It guarantees that the resource is properly closed after the execution of a block of code, even in case of exceptions. The use function is defined as an extension function on any class that implements the Closeable or AutoCloseable interfaces. Here's an example of using the use function with a file:

val file = File("example.txt")

file.outputStream().use {

it.write(byteArrayOf(1, 2, 3))

}

In this example, the use function is used to automatically close the output stream after writing the data to the file.

**Question:**  What is the "repeat" function in Kotlin?

**Answer:** The repeat function in Kotlin is used to repeat a block of code a specified number of times. It takes an integer as its parameter and a lambda that will be executed repeatedly for the given number of times. The index of the current iteration can be accessed through the implicit parameter it. Here's an example that uses repeat to print "Hello" 5 times:

repeat(5) {

println("Hello")

}

**Question:**  How do you use the "crossinline" modifier in Kotlin?

**Answer:** In Kotlin, the crossinline modifier is used to restrict the usage of a lambda expression to be used only in the current function. It prevents the lambda from being passed to a higher-order function and used outside the function's scope. This is useful when the lambda contains non-local control flow statements like return, which are not allowed to be used in a higher-order function.

Here's an example of using the crossinline modifier:

inline fun executeAsync(crossinline block: () -> Unit) {

val asyncTask = AsyncTask<Unit, Unit, Unit> {

block()

}

asyncTask.execute()

}

In the above example, the lambda expression block is marked as crossinline to prevent it from using non-local control flow statements that could break the current function's control flow.

**Question:**  What are the common performance issues in Kotlin, and how do you avoid them?

**Answer:** Kotlin, like any other programming language, can suffer from performance issues. Some common ones include creating unnecessary objects, using non-optimized data structures, and performing expensive operations on the main thread. To avoid these issues, you can use techniques like object pooling, optimized data structures, and asynchronous programming. Here are some examples:

// Creating unnecessary objects

fun processStrings(strings: List<String>) {

for (str in strings) {

// This creates a new StringBuilder instance on every iteration

val reversed = StringBuilder(str).reverse().toString()

// Do something with reversed string

}

}

// Using optimized data structures

val map = hashMapOf<String, Int>()

fun processStrings(strings: List<String>) {

for (str in strings) {

// This checks if the key already exists and increments the value

// without creating a new entry in the map

map[str] = (map[str] ?: 0) + 1

}

}

// Performing expensive operations on the main thread

fun loadDataFromNetwork() {

// This will block the main thread until the network call is finished

val result = performNetworkCall()

// Do something with result

}

// Instead, use asynchronous programming

suspend fun loadDataFromNetwork(): Result {

return withContext(Dispatchers.IO) {

performNetworkCall()

}

}

By using these techniques, you can optimize your Kotlin code and avoid common performance issues.

**Question:**  How do you use Kotlin's extension functions to write more concise and expressive code?

**Answer:** Kotlin's extension functions allow you to extend the functionality of existing classes without having to inherit from them or modify their source code. This can make your code more concise and expressive. For example, you can add a function to the String class to count the number of occurrences of a specific character:

fun String.countOccurrences(char: Char): Int {

return this.filter { it == char }.count()

}

val str = "hello world"

val count = str.countOccurrences('l')

println(count) // Output: 3

In this example, the countOccurrences function is added to the String class using an extension function. This makes it possible to call the function on any String instance, making the code more concise and expressive.

**Question:**  What is the difference between "lateinit" and "lazy" in Kotlin, and when should you use each?

**Answer:** In Kotlin, lateinit and lazy are used to initialize properties lazily, but they differ in their initialization strategies. lateinit is used for non-null var properties that are initialized after the class initialization, while lazy is used for val properties that are initialized only once when first accessed. lateinit can be risky because it can throw a NullPointerException if it's accessed before being initialized, whereas lazy is thread-safe and only evaluated once.

Here is an example of lateinit and lazy:

// Example of lateinit

class MyClass {

lateinit var str: String

fun init(str: String) {

this.str = str

}

fun print() {

println(str)

}

}

// Example of lazy

class MyOtherClass {

val str: String by lazy {

"Hello World"

}

fun print() {

println(str)

}

}

**Question:**  How do you implement the "strategy" design pattern in Kotlin?

**Answer:** The "strategy" design pattern can be implemented in Kotlin using interfaces and lambda expressions. First, define an interface that declares the strategy methods. Then, define classes that implement this interface with different strategies. Finally, pass a lambda expression that implements the desired strategy as an argument to a higher-order function that expects the strategy interface. Here's an example:

// Define a strategy interface

interface PaymentStrategy {

fun pay(amount: Double)

}

// Implement payment strategies

class CreditCardStrategy : PaymentStrategy {

override fun pay(amount: Double) {

// implementation for credit card payment

}

}

class PayPalStrategy : PaymentStrategy {

override fun pay(amount: Double) {

// implementation for PayPal payment

}

}

// Use a higher-order function with a lambda to execute a payment strategy

fun executePayment(amount: Double, strategy: PaymentStrategy) {

strategy.pay(amount)

}

// Use the higher-order function with a lambda expression

executePayment(100.0, object : PaymentStrategy {

override fun pay(amount: Double) {

// implementation for payment strategy

}

})

**Question:**  What is the difference between "infix" and "operator" in Kotlin, and when should you use each?

**Answer:** Both infix and operator are used to define custom operations for a type in Kotlin. The infix modifier is used to define functions that can be called using infix notation, with the function name placed between the operands. The operator modifier is used to define functions that overload operators like +, -, \*, and /.

You should use infix when you want to define a custom function that reads naturally in infix notation, like a to b, while you should use operator when you want to overload an existing operator to work with your own types.

Here's an example of an infix function:

infix fun String.hasSuffix(suffix: String) = this.endsWith(suffix)

// Usage

if (str hasSuffix "world") { ... }

Here's an example of an operator function:

data class Complex(val real: Double, val imaginary: Double) {

operator fun plus(other: Complex): Complex {

return Complex(real + other.real, imaginary + other.imaginary)

}

}

// Usage

val c1 = Complex(1.0, 2.0)

val c2 = Complex(3.0, 4.0)

val sum = c1 + c2

**Question:**  How do you use coroutines to implement asynchronous operations in Kotlin, and what are the best practices for doing so?

**Answer:** Coroutines in Kotlin provide a way to write asynchronous code that looks and behaves like synchronous code. To use coroutines, you first need to define a suspend function that performs the asynchronous operation. You can then call this function using a coroutine builder, such as launch, async, or runBlocking. The best practices for using coroutines include minimizing blocking calls, using structured concurrency, and using the withContext function to switch to a different coroutine context when necessary. Here is an example of using coroutines to download data from a remote server:

suspend fun downloadData(url: String): String {

return withContext(Dispatchers.IO) {

URL(url).readText()

}

}

// Usage

GlobalScope.launch {

val data = downloadData("https://example.com/data")

// Use the downloaded data

}

**Question:**  How do you properly handle exceptions in Kotlin, and what are the best practices for doing so?

**Answer:** Properly handling exceptions in Kotlin involves using try-catch blocks and being specific about the type of exception being thrown. Best practices include not catching Exception or Throwable, providing informative error messages, and logging exceptions. You can also use the "use" function to automatically close resources in a try-finally block, and use the "runCatching" function to handle exceptions and avoid crashing the program. Here's an example of using try-catch to handle a specific exception:

try {

// code that may throw an exception

} catch (e: IOException) {

// handle the IOException

Log.e(TAG, "IOException occurred: ${e.message}")

} finally {

// code to be executed regardless of whether an exception was thrown or not

}

**Question:**  What is the difference between "forEach" and "forEachIndexed" in Kotlin, and when should you use each?

**Answer:** Both forEach and forEachIndexed are higher-order functions in Kotlin for iterating over collections. forEach takes a lambda with one parameter that represents the element of the collection, while forEachIndexed takes a lambda with two parameters, the first representing the index and the second representing the element.

forEach is used when you only need to access the elements of the collection, while forEachIndexed is used when you also need the index of each element.

Here's an example usage of forEach and forEachIndexed:

val fruits = listOf("apple", "banana", "orange")

// using forEach

fruits.forEach { fruit ->

println(fruit)

}

// using forEachIndexed

fruits.forEachIndexed { index, fruit ->

println("$index: $fruit")

}

The output of the first example will be:

apple

banana

orange

The output of the second example will be:

0: apple

1: banana

2: orange

**Question:**  How do you implement the "chain of responsibility" design pattern in Kotlin?

**Answer:** The chain of responsibility pattern is used to process a request through a series of handlers, where each handler has the option to handle the request or pass it to the next handler in the chain. In Kotlin, this can be implemented using a series of objects, each of which has a reference to the next object in the chain. Each object in the chain must implement a common interface that defines the request handling method. Here's an example:

interface Handler {

fun handleRequest(request: String): String?

}

class ConcreteHandler1(private val successor: Handler) : Handler {

override fun handleRequest(request: String): String? {

if (request == "request1") {

return "Handled by ConcreteHandler1"

} else {

return successor.handleRequest(request)

}

}

}

class ConcreteHandler2(private val successor: Handler) : Handler {

override fun handleRequest(request: String): String? {

if (request == "request2") {

return "Handled by ConcreteHandler2"

} else {

return successor.handleRequest(request)

}

}

}

class ConcreteHandler3 : Handler {

override fun handleRequest(request: String): String? {

if (request == "request3") {

return "Handled by ConcreteHandler3"

} else {

return null

}

}

}

In this example, ConcreteHandler1 and ConcreteHandler2 each have a reference to the next object in the chain, and ConcreteHandler3 is the last object in the chain. The handleRequest method of each object checks if it can handle the request, and if not, passes the request to the next object in the chain. The client code can create the chain and send requests to the first object in the chain.

**Question:**  How do you use the "withIndex" function to iterate over a collection in Kotlin, and what are the best practices for doing so?

**Answer:** The withIndex() function is used to iterate over a collection in Kotlin while also accessing the index of each element. It returns an iterable of pairs, where the first element of each pair is the index and the second element is the corresponding element in the collection.

To use withIndex(), simply call it on the collection you want to iterate over and use a loop to access the index and element:

val list = listOf("apple", "banana", "orange")

for ((index, element) in list.withIndex()) {

println("Index: $index, Element: $element")

}

When using withIndex(), it is recommended to use destructuring declarations in the loop to make the code more concise and readable. It is also a good practice to avoid modifying the collection while iterating over it to prevent unexpected behavior.

**Question:**  What is Kotlin?

**Answer:** : Programming in Kotlin, a statically typed language that utilizes the JVM, is possible. Both the Java source code and the LLVM compiler can be used to compile it.

**Question:**  : Why you should switch to Kotlin from Java?

**Answer:** In comparison to Java, the Kotlin language is extremely easy. Compared to Java, it reduces a lot of code repetition. Some helpful features that Java does not support can be found in Kotlin.

**Question:**  Tell three most important benefits of using Kotlin?

**Answer:** Three most important benefits of using Kotlin are:- 1. The Java-like syntax of the Kotlin language makes it simple to learn. 2. A functional language built on the JVM, Kotlin. Thus, a lot of boilerplate is eliminated. 3. Because it uses an expressive language, code can be read and understood.

**Question:**  Explain the use of the extension function.

**Answer:** Without having to inherit from the class, extension functions help extend classes

**Question:**  What does ‘Null Safety’ mean in Kotlin?

**Answer:** The Null Safety feature enables immediate removal of the possibility of NullPointerException occurrence. It is also feasible to distinguish between references that can be nulled and those that cannot.

**Question:**  Why is Kotlin interoperable with Java?

**Answer:** Kotlin uses JVM bytecode, which makes it compatible with Java. It can be directly compiled to bytecode, which reduces compile time and does not distinguish between Java and Kotlin for the JVM.

**Question:**  Is there any Ternary Conditional Operator in Kotlin Like in Java?

**Answer:** The ternary conditional operator is not present in Kotlin.

**Question:**  How many constructors are available in Kotlin?

**Answer:** Two constructors are available in kotlin. These are:- 1. Primary constructor 2. Secondary constructor

**Question:**  Give me the name of the extension methods Kotlin provides to java.io.File.

**Answer:** Name of extension methods that kotlin provides to java.io.file are:-

1. Use buffered reader() to read file data into BufferedReader

2. For reading a file’s contents to a ByteArray, use the read bytes() function.

3. For reading a single String, use the read text() function.

4. For reading a file line by line in Kotlin, use the for each line() function.

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

**Answer:** Here are a few key Kotlin characteristics that Java lacks:

1. Null Safety

2. Operator Overloading

3. Coroutines

4. Range expressions

5. Smart casts

6. Companion Objects

**Question:**  How many constructors are available in Kotlin?

**Answer:** Two types of constructors available in Kotlin are:

* Primary constructor
* Secondary constructor
* Who is the developer of Kotlin?
* Answer: Kotlin was developed by JetBrains.

**Question:**  What are the most important benefits of using Kotlin?

**Answer:** Kotlin language is easy to learn as its syntax is similar to Java.

Kotlin is a functional language and based on JVM. So, it removes lots of boiler plate

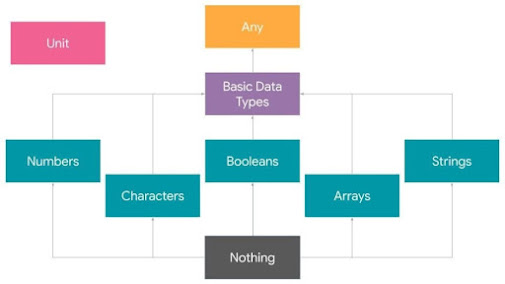
It is an expressive language which makes code readable and understandable.

**Question:**  What are the basic data types used in Kotlin?

**Answer:** Here is the list of basic data types you will find in Kotlin programming language:

* Strings
* Numbers
* Arrays
* Booleans
* Characteristics

Here is a nice diagram which shows the basic data types used in Kotlin programming language:



**Question:**  Which type of programming does Kotlin support?

**Answer:** Kotlin supports only two types of programming which are as follows: Procedural programming Object-oriented programming

**Question:**  What are the structural expressions in Kotlin?

**Answer:** There are three Structural expressions in Kotlin and are as follows: Return: It returns from the nearest enclosing function or anonymous function by default. Break: This expression terminates the closest enclosing loop. Continue: This expression proceeds you to the next closest enclosing loop.

**Question:**  What are the modifiers available in Kotlin?

**Answer:** Access modifier in Kotlin provides the developer to customize the declarations as per the requirements. Kotlin provides four modifiers. They are:

Private: This makes the declaration visible only inside the file containing a declaration.

Public: It is by default, which means that the declarations will be visible everywhere.

Internal: This makes the declaration visible everywhere in the same modules.

Protected: This keeps the declaration protected and is not available for top-level declarations.

**Question:**  What is null safety in Kotlin, and why is it important?

**Answer:** Null safety is a feature of Kotlin that ensures that variables cannot be assigned a null value. This is important because it helps avoid NullPointerException errors, which can be difficult to debug.

**Question:**  What are some of the basic syntax rules for Kotlin?

**Answer:** Some of the basic syntax rules for Kotlin include the use of semicolons to terminate statements, the use of curly braces to delimit blocks of code, and the use of the val keyword to declare immutable variables.

**Question:**  What is a lambda expression in Kotlin?

**Answer:** A lambda expression is an anonymous function that can concisely represent a function with a single parameter. Lambda expressions are often used in conjunction with higher-order functions, such as map and filter.

**Question:**  How do you declare a default argument in Kotlin?

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

**Question:**  Can Kotline offer any additional functionality for standard Java packages or standard Java classes?

**Answer:** Kotlin programs can run on standard JVM like any another compiled Java code. It allows JVM to compile any program to byte-code. It is accessible using Java Virtual Machine. Therefore, Kotlin is almost similar to Java. Moreover, Kotlin applications can be built with parts of Java code.

**Question:**  What is the difference between open and public keywords in Kotlin?

**Answer:** 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. Also, all the classes are public by default. If no visibility modifier is specified, public is used by default, which means our declarations will be accessible everywhere inside the program.

**Question:**  What are the advantages of Kotlin over Java?

**Answer:** Here are key 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 clearer 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.

Here is a nice diagram which illustrate the difference between Kotlin and Java, you can see what you can achieve in just 3 lines of code in Kotlin which normally takes 20 lines in Java:



**Question:**  What is the difference between a data class and a regular class in Kotlin?

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

**Question:**  How do you declare a function in Kotlin?

**Answer:** To declare a function in Kotlin, use the fun keyword followed by the name of the function and the parameters that it accepts. You can then define the function body within curly braces

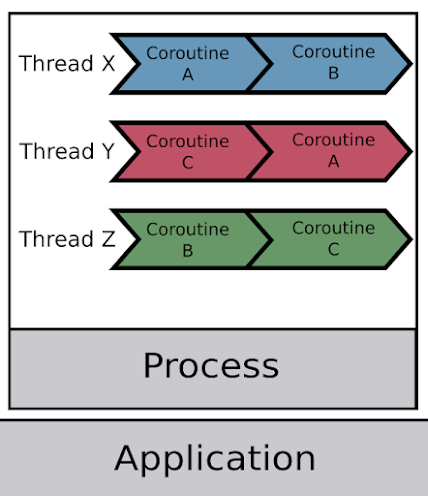
**Question:**  What is the difference between a suspend function and a regular function?

**Answer:** A suspend function can be suspended, meaning that you can pause its execution and resume it at a later time. A regular function cannot be suspended and will always execute to completion.

**Question:**  What are coroutines in Kotlin?

**Answer:** Coroutines is one of the amazing Kotlin features that allow for concurrent or parallel execution of code. Coroutines can improve application performance by taking advantage of multiple cores in a processor.

Here is a nice diagram which shows how you can use multiple coroutine inside each thread in a Kotlin application:



**Question:**  How do you cancel a coroutine in Kotlin?

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

**Question:**  What is type inference in Kotlin?

**Answer:** Type inference is the process of automatically determining the type of a variable or expression based on its value. In Kotlin, type inference determines the type of variables when they are first declared and the return type of functions .

**Question:**  Why should we use Kotlin?

**Answer:** Kotlin is concise

Kotlin is null-safe

Kotlin is interoperable

**Question:**  Differentiate var and val in kotlin?

**Answer:** Value of var variable can be reassigned it can be changed once assigned it is mutable in nature but variable declared using val keyword can not change its value once assigned it is immutable in nature.

**Question:**  Differentiate val and const in kotlin?

**Answer:** Both the variables are immutable in nature we can not reassigned their value. But the value of the const variable can be assigned at the compile-time while the value of the val variable can be assigned at runtime.Both the variables are immutable in nature we can not reassigned their value. But the value of the const variable can be assigned at the compile-time while the value of the val variable can be assigned at runtime.

**Question:**  How to ensure null safety in Kotlin?

**Answer:** One of the most important advantages of using Kotlin is null safety. In Java, if you access some null variable then you’ll get a NullPointerException. So, the below code in Kotlin will produce a compile-time error:

var name: String = “Codinglance”

name = null //error

So, to assign null values to a variable, you would like to declare the name variable as a nullable string and so during the access of this variable, you have to use a secure call operator i.e. ?.

var name: String? = "Codinglance"

print(name?.length) // ok

name = null // ok

**Question:**  what’s the difference between safe calls(?.) and null check(!!)?

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

var name: String? = “Codinglance”

println(name?.length) // 8

name = null

println(name?.length) // null

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

var name: String? = "Codinglance"

println(name?.length) // 8

name = null

println(name!!.length) // KotlinNullPointerException

**Question:**  What is Elvis operator in Kotlin?

**Answer:** In Kotlin, you can easily assign null values to a variable with the help of null safety property. To check if a value is having null value or not you can simply use if-else or can use the Elvis operator i.e. ?: For example:

var name:String? = "Codinglance"

val nameLength = name?.length ?: -1

println(nameLength)

The Elvis operator(?:) in this example will return the length of name variable if the value is not null otherwise it will return -1 if the value is null.

**Question:**  How can we convert a Kotlin source file to a Java source file?

**Answer:** To follow these three steps we can simply do this:

Firstly open your Kotlin project in the IntelliJ IDEA / Android Studio.

Go to the Tools > Kotlin > Show Kotlin Bytecode.

Now you have to click on the Decompile button to get your Java code from the bytecode.

**Question:**  What is a data class in Kotlin?

**Answer:** Data classes are those classes which are used to store some data. In Kotlin, these class is create with data class. The following is an example of data class:

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

In data class we don’t to create functions like we do in java hashCode(), equals(), toString(), copy(). The compiler automatically creates these internally,

**Question:**  what’s String Interpolation in Kotlin?

**Answer:** If you wish to use some variable or perform some operation inside a string then String Interpolation are often used. you’ll use the $ sign to use some variable within the string or can perform some operation in between sign.

var name = “Codinglance”

print("Hello! i'm learning from $name")

**Question:**  When to use the lateinit keyword in Kotlin?

**Answer:** lateinit is late initialization.it means late initialization means if we don’t want to initialize variable in the constructor instead you want to initialize it later. It is used only with the mutable data type

**Question:**  How can we check if a lateinit variable has been initialized or not?

**Answer:** We can easily check if the lateinit variable has been initialized or not with the help of isInitialized method. This method returns true if the lateinit property has been initialized or it returns false. For example:

class User {

lateinit var userName: String

fun initializeUserName() {

println(this::userName.isInitialized)

userName = “Codinglance”

println(this::userName.isInitialized)

}

}

fun main(args: Array<String>) {

User().initializeUserName()

}

**Question:**  Differentiate lateinit and lazy in Kotlin?

**Answer:** lazy keyword used only with val properties, while lateinit keyword used for with var .You can use lazy for val properties only, while lateinit can be used only with var because it is not able to compiled to a final field.

**Question:**  What is the difference between == operator and === operator?

**Answer:** The == operator is used to compare the values which is stored in variables and the === operator is used to check if the reference of the variables are equal or not. But in primitive data types, the === operator also checks for the value and not reference.

// primitive data example

val int1 = 10

val int2 = 10

println(int1 == int2) // true

println(int1 === int2) // true

// wrapper class example

val num1 = Integer(10)

val num2 = Integer(10)

println(num1 == num2) // true

println(num1 === num2) //false

**Question:**  What are companion objects in Kotlin?

**Answer:** In Kotlin, we can call a function or any member function without making the instance of the class with the help of companion object for getting this you have to define function or any variable as a companion object member within the class. So, by doing this, you can easily access the members and member faction of the class by class name only.

**Question:**  what is alternate of Java static methods in Kotlin?

**Answer:** To achieve the functionality just like Java static methods in Kotlin, we are able to use:

companion object

package-level function

object

**Question:**  Differentiate List and Array types in Kotlin?

**Answer:** If you have got a listing of information that’s having a set size, then you’ll use an Array. But if the scale of the list can vary, then we’ve to use a mutable list.

**Question:**  Can we use the new keyword to instantiate a category object in Kotlin?

**Answer:** No, in Kotlin we do not need to use the new keyword to instantiate a category object. To instantiate a category object, simply we use:

var varName = ClassName()

**Question:**  What are visibility modifiers in Kotlin?

**Answer:** A visibility modifier or access specifier or access modifier could be a concept that’s accustomed define the scope of something in an exceedingly artificial language. In Kotlin, we’ve got four visibility modifiers:

private: visible inside that specific class or file containing the declaration.

protected: visible inside that individual class or file and also within the subclass of that individual class where it’s declared.

internal: visible everywhere therein particular module.

public: visible to everyone.

**Question:**  a way to create a Singleton class in Kotlin?

**Answer:** A singleton class may be a class that’s defined in such some way that only 1 instance of the category is created and is employed where we’d like just one instance of the category like in logging, database connections, etc. To create a Singleton class in Kotlin, you wish to use the object keyword.

object AnySingletonClassName

Note: you cannot use constructor in object, but you’ll be able to use init.

**Question:**  What’s the difference between lazy and lateinit?

**Answer:**

| **Lateinit** | **Lazy** |
| --- | --- |
| It is late initialization | It is lazy initialization. |
| It can be initialized from anywhere in the object | It can be initialized only from the lambda function. |
| It can only be used for var type. | It can only be used for Val type. |
| Multiple initializations are possible in lateinit. | Only a single initialization is allowed here. |
| It is not allowed in properties of primitive types. | It is allowed on properties of primitive types. |

**Question:**  Which one is better kotlin or Java?

**Answer:** Although both Java and Kotlin are used for Android app development, it is hard to choose a better one. This is because both have their pros/cons and are dominant programming languages.

#### Advantages of using Java:

* It has a straightforward and readable syntax.
* It is a pure form of OOPS
* It has a faster compilation speed than Kotlin
* Some features of Android studio, like auto-complete and compilation, run faster.

#### Advantages of using Kotlin:

* It is comparatively easy to learn than Java
* It does not have the legacy baggage that Java has been carrying around for a long time.
* It is packed with modern programming features, making development a lot easier.
* It has a safer and smarter compiler as compared to Java.

As you can see, both of these languages have their advantages and disadvantages. So, the actual question you must be asking yourself is, what are my needs and what’s best for my project.

**Question:**  Can you execute Kotlin code without JVM?

**Answer:** Yes, you can execute a Kotlin code without the help of a JVM.

**Question:**  How does string interpolation work in Kotlin?

**Answer:** String interpolation is a variable substitution having its value inside a string. In Kotlin, the $ character is used for interpolating a variable, and the ${} is used to interpolate an expression. Kotlin allows users the liberty of accessing variables and expressions directly from the string literals, thus eliminating the need for concatenation.

**Question:**  What do you understand by Kotlin, explain its working?

**Answer:** Kotlin is an open-source programming language that entirely runs over the virtual machine for its proceeding. This language executes utilizing the Java platform and it was developed by JetBrains. This language is quite simpler than most of its variants specially Java. All of its codes are converted into native quotes that are processed by the byte code conversion where it refers to the object-oriented programming language.

**Question:**  Why Kotlin is better than Java?

**Answer:** As Kotlin is an open-source programming language it significantly refers to a general language serving all the purposes. It also offers code redundancy in which most of the features are included which are not available in Java. It is also a perfect combination of features like null safety, primary constructors, extension functions, properties, inline expressions, lambda expressions and various other types of interference are included in it. Kotlin also serves for all the major properties and its variables.

**Question:**  What do you understand by structural expressions?

**Answer:** Structural expressions refer to the representative form of the loops in Kotlin. There are three important expressions which are:

* Break- this expression helps the user to break the nearest enclosing loop
* Return- this expression helps the user to return the nearest functions or any other default functions.
* Continue- this expression helps the user to proceed further for the next loop.

If you are looking for a decent set of questions then you can categorize it under the best Kotlin interview questions for experienced professionals.

**Question:**  Please elaborate on the modifiers in Kotlin?

**Answer:** Developers which are used to personalize declarations are known as modifiers and there are four types of modifiers available in Kotlin.  
These modifiers are

* Public- all the declarations are visible here
* Internal- it uses the declarations present on specific locations
* Private- declarations are contained in restricted files
* Protected- here declarations are secured and usually not available for top-level declarations.

**Question:**  Can you mention some of the extension methods used in Kotlin?

**Answer:** Some of the significant extension methods used in Kotlin are-

* Read texts () - it is helped to read the content present in files to a single string
* Buffer reader () - it is used to read the files to buffer reader
* Read each line () - it is used to read the lines by the following line after line
* Headlines () - it helps to read the content for listing.

**Question:**  What do you understand by data class in Kotlin?

**Answer:** The classes which are used to hold the data or content are known as data classes. Earlier these data classes were only used to build the codes in Java but following the explicit implementation of its higher and greater set of properties, it is significant in kotlin as well. Along with this language, functions like tostring, equals are also combined with it. Kotlin offers a variety of these functions by implementing them along with the component functions.

**Question:**  Mention the type of strings present in Kotlin?

**Answer:** Basically, strings refer to a collection of various characters altogether. There are two types of strings present in Kotlin which are-

Raw strings

Escaped strings

Under these strings, templates can also be evaluated easily and this evaluation is known by term as string template interpolation.

**Question:**  List down the basic data types in Kotlin?

**Answer:** Data types are the instances of consonants or variables tending to decide the type of variable. It also specifies the space required for the storage of these variables.

##### The data types are-

* Numbers
* Characters
* Strings
* Arrays
* Booleans

**Question:**  What are constructors, explain its types?

**Answer:** There are two types of constructors in Kotlin which are-

Primary constructor- It is a specific section of class header and mostly it is always declared after the class name.

Secondary constructor- This section of the constructor is always declared inside the class.

**Question:**  What is the difference between inline and infix functions?

**Answer:** Inline Functions

Used to prevent the object allocations for the functions or lambda expressions that have been called.

Infix Functions

It is used for calling functions without any parentheses or brackets.

**Question:**  How to create empty constructor for data class in Kotlin?

**Answer:** If you give default values to all the primary constructor values, you shall automatically create an empty constructor for data class.

For example, if you have a data class like this:

data class Activity(

var updated\_on: String,

var tags: List,

var description: String,

var user\_id: List,

)

And you want to initialize the data class with an empty constructor, all you have to do is assign the default values to all the primary constructors, like this:

data class Activity(

var updated\_on: String = "",

var tags: List = emptyList(),

var description: String = "",

var user\_id: List = emptyList(),

)

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

**Answer:** Here’s how to initialize an array in Kotlin.

Example

val numbers: IntArray = intArrayOf(11, 12, 13, 14, 15)

**Question:**  Where should I use var and where val?

**Answer:** In Kotlin, you should use var where the value of the variable is changing very frequently. A good example would be getting the location of an android device: var integerVariable : Int? = null

Whereas, Val should be used in a case where there would be no change of value throughout the whole class. For example, setting the value of text view or maybe a button’s text through code:

val stringVariables : String = "Button's Final Text"

**Question:**  How to create singleton class in kotlin?

**Answer:** Here is the best way to create a singleton class in Kotlin:

Example

enum class Singleton {

INSTANCES;

}

Singleton.INSTANCES.myFunction()

**Question:**  What is a purpose of Companion Objects in Kotlin?

**Answer:** In Kotlin, if you need to write a code for a function that can be called without a class instance but it needs access to the internals of a class, you can use a companion object declaration in that specific class.

Example

class EventsManager {

companion object FirebasesManager {

}

}

val firebasesManager = EventsManager.FirebasesManager

**Question:**  What’s Null Safety and Nullable Types in Kotlin?

**Answer:** Null safety, also known as void safety is a guarantee that no object references shall have null or void values. Nullable types in Kotlin are a type of reference which are declared by putting a “?” behind the String.

**Question:**  What do you understand by Kotlin?

**Answer:** Kotlin is an open-source programming language that includes concepts from object-oriented programming. It is JVM-based and may be compiled with either Java source code or the LLVM compiler. It's commonly used to create Android apps, server-side apps, and a wide range of other applications. The properties like Range Expression, Extension Function, Companion Object, Smart casts, and Data classes are regarded as a Kotlin Language bonus.

**Question:**  What is Extension Function?

**Answer:** The programmer can extend the functionality of existing classes in Kotlin without inheriting them. This is accomplished via a feature called extensions.

Extension Functions are functions that are added to an existing class.

The prefix receiver-type with method name is used to declare the extension function.

fun <Class\_name>.<Method\_name>()

**Question:**  What is Kotlin Null Safety?

**Answer:** Null safety in Kotlin is a procedure for removing the risk of a null reference in the code. If any null argument is supplied without any additional statements being executed, the Kotlin compiler immediately throws NullPointerException.

**Question:**  How many Constructors are there in Kotlin?

**Answer:** In Kotlin, there are two types of constructors:

The Primary Constructor: It is a simple way to initialize a class.

Student(val first name: String, var age: Int){

//class body

}

The Secondary Constructor: Enables us to add further initialization logic

class Mycourse{

constructor(id: Int){

//code

}

constructor(Name: String, id: Int){

//code

}

}

**Question:**  Do primary and secondary constructors have any relationship?

**Answer:** Yes, one must explicitly invoke the primary constructor while using a secondary constructor.

**Question:**  What is the critical difference between ‘var’ and ‘val’ for Variable declaration in Kotlin?

**Answer:** If we want to declare a mutable(changeable) variable, we can use the var keyword. Use val for immutable variables, which can't be modified once they've been allocated.

**Question:**  What is the key difference between ‘val’ and ‘const’ for Variable declaration in Kotlin?

**Answer:** The variables which are declared with the val and ‘const’ keywords are immutable. The value of the const variable must be known at compile-time, whereas we can assign the value of the ‘val’ variable at any time(runtime also).

**Question:**  Tell about some features of Kotlin that are not present in Java?

**Answer:** Java lacks a few major Kotlin features:

Overloading of the Operator

Coroutines

The Null Safety

Extension Function

The Range Expressions

Smart Casts

The Companion Objects

**Question:**  What is the Key Difference between ‘ fold’ and ‘reduce’ in Kotlin?

**Answer:** Fold: The fold is a function that takes an initial value, and the lambda users give in the first invocation. The starting value and the collection's first element will be given as parameters.

listOf(1, 2, 3).fold(0) { acc,next -> acc + next }

The lambda will be called for the first time with parameters 0 and 1.

Reduce: The reduce function does not require a starting value. Instead, it starts with the initial element of the collection.

listOf(1, 2, 3).reduce { acc, next -> acc + next }

Here, The lambda will be called for the first time with parameters 1 and 2.

**Question:**  What is Ranges Operator?

**Answer:** The ranges operator supports the iteration of a range. It has the operator form (..) As an example,

for (num in 1..5){ @ print(num) @ }

It will print 1 to 5 in Output.

**Question:**  What is the method to Compare Two Strings in Kotlin?

**Answer:** In Kotlin, you can compare strings in the following ways:

By Using the “ = =” operator

The == operator in Kotlin is used to compare the structural equality of two objects. If both objects have the same value, it will return true.

By using equals() function

This is a case-sensitive comparison. In Kotlin, set the second parameter to True for case-insensitive string comparison.

By Using Extension function compareTo()

Instead of a Boolean value, CompareTo() produces an Int value. It is also Case-sensitive.

**Question:**  Is there a ternary operator in Kotlin like there is in Java?

**Answer:** No, we don't have a ternary operator in Kotlin, but we can use the if-else or Elvis operators to mimic the functionality of a ternary operator.

**Question:**  What is Elvis operator?

**Answer:** The Elvis operator in Kotlin takes two arguments and returns the first if it's non-null or the second if it's not. The null coalescing operator is a fancy name for it. It's a null-safety-checking variation of the ternary operator.

Example:

var name:String? = " Coding Ninjas"

val nameLength = name?.length ?: -1

println(nameLength)

If the value is not null, the Elvis operator(?:) returns the length of the name;

otherwise, if the value is null, it returns -1.

**Question:**  What do @JvmStatic, @JvmOverloads, and @JvmFiled mean in Kotlin?

**Answer:** The uses of @JvmStatic, @JvmOverloads, and @JvmField in Kotlin are as follows:

@JvmStatic: This annotation informs the compiler that the method is a static method called from Java code.

@JvmOverloads: The @JvmOverloads annotation is used to utilize the default values passed as an argument in Kotlin code from Java code.

@JvmField: We need to use the @JvmField in the Kotlin code to access the fields of a Kotlin class from Java code without requiring any getters or setters.

**Question:**  In Kotlin, can we use primitive types like int, double, and float?

**Answer:** We can't utilize primitive types directly in Kotlin. As an object wrapper for primitives, we can use classes like Int, Double, etc. However, these primitive types exist in compiled bytecode.

**Question:**  What do you mean by String Interpolation?

**Answer:** String Interpolation is useful if we want to use a variable or execute an operation on a string. The $ character is used to interpolate a variable in Kotlin, and the ${} character is used to interpolate an expression.

String formatting in Kotlin is more powerful than simple interpolation.

val name = "Coding Ninjas"

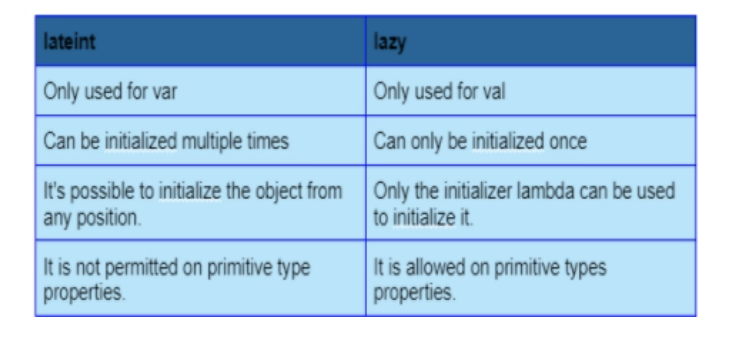
print("I am learning from $name!")

**Question:**  When should we use Kotlin's lateinit keyword?

**Answer:** The lateinit keyword in Kotlin is used for variables that are initialized after the declaration, or we may say that a lateinit variable is a variable that is late initialized. When we know that the variable will be initialized before we use it, we use the lateinit keyword.

**Question:**  In Kotlin, what is the difference between lateinit and lazy?

**Answer:** The given table shows the difference between lateint and lazy



**Question:**  What is the main difference between FlatMap and Map?

**Answer:** FlatMap is a way for combining all the elements in a list into a single one. A map is a tool for transforming a list based on specified conditions.

**Question:**  What is the critical difference between List and Array types in Kotlin?

**Answer:** You can use an Array if you have a list of data that has a fixed size. However, since the list's size can change, we'll need to use a mutable list.

**Question:**  In Kotlin, can we use the new keyword to create a class object?

**Answer:** No, we don't have to use the new keyword to create a class object in Kotlin. To create a class object, we just type

var varName = ClassName()

**Question:**  What is the double-bang!! Operator in Kotlin?

**Answer:** The "not-null assertion operator" is another name for this operator. The double exclamation(!!) operator converts any value to a non-NULL type value, and if the corresponding value is NULL, it throws an exception (KotlinNullPointerException).

Example:

fun main(args: Array<String>) {

var name: String?

name = null

println(name!!)

It allows you to be absolutely sure that the value isn't null.

**Question:**  What's the difference between blocking and suspending?

**Answer:** Blocking call to a function means that any further calls from the same thread will cause the parent's execution to halt. As a result, if you make a blocking call on the main thread's execution, the UI will effectively be frozen. The user will see a static screen until the blocking calls are completed, which is not a good thing.

Suspending does not always prevent your parent function from running. You can easily push a suspended function to a different thread if you call it on one thread. It will not block the main thread if it is a significant operation. The suspending process will pause its execution if it needs to suspend. This frees up the thread for other tasks.

**Question:**  In Kotlin, what is the main difference between open and public?

**Answer:** "Open for extension" is what the open keyword indicates. The open annotation on a class is the total opposite of the final annotation in Java: it allows others to inherit from it.

If no visibility modifier is specified, the public is used by default, which means your declarations will be accessible everywhere.

**Question:**  In Kotlin, what is used as an equivalent of Java static?

**Answer:** Companion Object is similar to the static method in Kotlin. When you're extending a companion object with an interface, it's a better idea to define functions inside the companion object. Static class in java becomes the companion object in Kotlin.

Example:

Static Class in Java:

class CodingNinja {

public static int a() { return 1; }

Companion Object in Kotlin:

class CodingNinja{

companion object {

fun a() : Int = 1

}

}

// to run

CodingNinja.a();

**Question:**  What is Inline function?

**Answer:** Higher-order functions perform better when inline functions are used. The compiler is instructed to copy parameters and functions to the call location using the inline function.

Example:

fun main(args: Array<String>) {

inlineFunction({ println("calling the inline functions")})

}

inline fun inlineFunction(myFun: () -> Unit ) {

myFun()

print("The code inside the inline function")

}

Output:

Calling the inline functions

The code inside the inline function.

**Question:**  What is a noinline Function in Kotlin?

**Answer:** This is used to specify whether or not expressions should be inlined in the call. If there are no noinline function parameters and no reified type parameters in an inline function, the compiler will issue a warning.

inline fun noinline(A: () -> Unit, noinline B: () -> Unit) {

A()

B()

}

**Question:**  What is the meaning of Pair and Triple in Kotlin?

**Answer:** Pairs and Triples are used to return two and three values from a function, respectively, and the returned values might be of the same or different data types.

**Question:**  What do you mean by Label?

**Answer:** A label can be applied to any expression in Kotlin. Labels consist of an identifier followed by the @ symbol, such as abc@ or xyz@. Place a label symbol in front of an expression to label it.

**Example:**

fun Primes() {

// Here we are using labels to specify which loop will skip the current iteration

outer@for (num in 2..100) {

for (check in 2..(num / 2)) {

if (num % check == 0) {

continue@outer

}

}

println(num)

}

}

**Question:**  What are the benefits of Sealed class over Enum?

**Answer:** We can simply add many custom constructors to a sealed class depending on our needs. We can also define many functions, each with its name, parameters, and return type.

**Question:**  Can we execute Kotlin code without JVM?

**Answer:** The Java Virtual Machine (JVM) compiles Kotlin code to native code. However, this can also be done without it.

**Question:**  What kinds of Programming are supported by Kotlin?

**Answer:** Kotlin is a statically typed, open-source programming language that may be used for both object-oriented and functional programming. Similar syntax and features from other languages, such as C#, Java, and Scala, are available in Kotlin.

**Question:**  Is Kotlin only used for Android?

**Answer:** Kotlin is an open-source programming language that is mostly used for Android development. In addition, Kotlin is utilized to create Gradle plugins. One can use it for backend development or wherever java is used after compiling it to the JVM.

**Question:**  Is Kotlin suitable for Beginners?

**Answer:** If you've worked with Java before and can read and comprehend Java code produced by others, it's a bright idea to start studying Kotlin and writing all of your future Android projects in the Kotlin language.

**Question:**  What is Kotlin? / Describe Kotlin in brief.

**Answer:** Kotlin is a general-purpose, statically typed, open-source programming language that runs on the JVM. It runs on JVM and can be used anywhere Java is used today. It can be compiled either using Java source code or LLVM compiler. It is generally used to develop Android apps, server-side apps, and much more.

**Question:**  What do you know about the history of Kotlin?

**Answer:** Kotlin was developed by the JetBrains team. This project was started in 2010 to develop a language for Android apps development, and officially its first version was released in February 2016. Kotlin was developed under the Apache 2.0 license.

**Question:**  What are the most important features of Kotlin?

**Answer:** The most popular features of kotlin are:

Kotlin is Concise: Kotlin reduces the writing of the extra codes, making Kotlin more concise.

Compact code: Kotlin is an OOPs-based programming language. Its code lines may be reduced by up to 40% compared to Java, making it an excellent choice for software development.

Kotlin is Simple: Kotlin is a simple language to learn. When working with Kotlin, compiling the code is simple, resulting in improved performance for Android development. It also explains which types of data functions can be used throughout the code.

Open Source: Kotlin is open source for Android, and it uses the JVM to combine the benefits of OOPs and functional programming.

Null safety: Kotlin is null safety language. Kotlin aimed to eliminate the NullPointerException (null reference) from the code.

A high number of extensions: Kotlin supports various extension functions and extension properties without modifying the code. It means that it can help to extend the functionality of classes without touching their code. Kotlin may support a variety of extension functions to help developers make existing code more appealing and wonderful.

Full Java Interoperability: Kotlin provides full interoperability for Java code. Java code can utilize Kotlin code, and Kotlin code can use Java code. So, if you are familiar with OOPs programming and good in Java programming language, you can switch to Kotlin development easily. Also, if there are any Java-based applications, they can be used with Kotlin's environment.

Smart Cast: Kotlin supports the smart cast technique. By using this technique, we can reduce the cost of an application while also improving its speed and performance. It technique uses typecasting or immutable data to manage the efficiency of programming. \

Low Learning Curve: Many businesses prefer Kotlin because of its low adoption cost. Most significantly, it is simple for developers to learn, especially if they have programming experience.

Compilation Time: Kotlin is faster and better than Java in terms of its performance and fast compilation time.

Tools-friendly: Kotlin is Tools-friendly. You can build the Kotlin programs by using the command line as well as any of Java IDE.

**Question:**  Why did you switch to Kotlin from Java? Why do some developers like to switch to Kotlin from Java?

**Answer:** The Kotlin programing language seems to be simpler and cleaner than Java. It removes a lot of redundancies in code as compared to Java. Kotlin also offers some useful features that Java doesn't yet support, making the code more idiomatic. Kotlin has been added to Android Studio's list of supported languages recently. So, there is much to expect from Kotlin in easing out the development efforts and good support in the future.

**Question:**  How does Kotlin work on Android?

**Answer:** Kotlin is very much similar to the Java programming language. Like Java, the Kotlin code is also compiled into the Java bytecode and executed at runtime by the Java Virtual Machine, i.e., JVM. For example, when a Kotlin file named Main.kt is compiled, it will eventually turn into a class, and then the bytecode of the class will be generated. The name of the bytecode file will be MainKt.class, and this file will be executed by the JVM.

**Question:**  What is the difference between the variable declaration with var and variable declaration with val?

**Answer:** The variable declaration with var and the variable declaration with val is used for different purposes. If you want to declare some mutable (changeable) variable, you should use var. If you want to declare the immutable variable, you should use val because val variables can't be changed once you have assigned them.

**Question:**  What is the difference between the variable declaration with val and variable declaration with const?

**Answer:** Both the variables that are declared with val and const are immutable in nature. But the difference between the variable declaration with val and variable declaration with const is that the value of the const variable must be known at the compile-time. In contrast, the value of the val variable can be assigned at runtime also.

**Question:**  How can you create a singleton in Kotlin?

**Answer:** We can create a singleton in Kotlin by using an object.

Syntax:

object SomeSingleton

The above Kotlin object will be compiled to the following equivalent Java code:

public final class SomeSingleton {

public static final SomeSingleton INSTANCE;

private SomeSingleton() {

INSTANCE = (SomeSingleton)this;

System.out.println("init complete");

}

static {

new SomeSingleton();

}

}

The above way is preferred to implement singletons on a JVM because it enables thread-safe lazy initialization without relying on a locking algorithm like the complex double-checked locking.

**Question:**  What is a primary constructor in Kotlin?

**Answer:** In Kotlin, the primary constructor is a part of the class header. Unlike Java, it doesn't need you to declare a constructor in the body of the class.

Kotlin facilitates you to declare the constructor in the class header itself:

See the following example:

class Person constructor(name: String, age: Int, salary: Int) {

}

Just like functions or methods, it takes a series of parameters with their type. These parameters initialize the variables present in the class.

If you do not have any annotations or modifiers (public, private, protected), you can omit the constructor keyword like the following example.

class Person (name: String, age: Int, salary: Int) {

}

By removing the constructor keyword, you can get code that is simplified and easy to understand.

**Question:**  What do you understand by Null safety in Kotlin?

**Answer:** Null safety is one of the major advantages of using Kotlin. Kotlin's type system ensures eliminating the danger of null references from code, also known as The Billion Dollar Mistake. One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference will result in a null reference exception. In Java, this would be the equivalent of a NullPointerException or NPE for short.

In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that cannot (non-null references). For example, a regular variable of type String can not hold null:

var a: String = "abc"

a = null // compilation error

To allow nulls, we can declare a variable as nullable string, written "String?":

var b: String? = "abc"

b = null // ok

print(b)

**Question:**  How can you ensure null safety in Kotlin?

**Answer:** Null safety is a feature introduced in Kotlin. In Java, if you access some null variable, then you will get a NullPointerException. So, the following code in Kotlin will produce a compile-time error:

var name: String = "MindOrks"

name = null //error

So, to overcome this issue, you have to assign null values to a variable, and 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

**Question:**  What is a data class in Kotlin?

**Answer:** In Kotlin, a data class is a class whose main purpose is to hold data. It is marked as "data".

Syntax:

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

The data classes must have to fulfill the following requirements to ensure consistency and meaningful behavior of the generated code:

The primary constructor must have at least one parameter

, and all primary constructor parameters need to be marked as val or var.

Data classes cannot be abstract, open, sealed, or inner.

**Question:**  What is the default behavior of Kotlin classes?

**Answer:** By default, all classes are final in Kotlin. That's because Kotlin allows multiple inheritances for classes, and an open class is more expensive than a final class.

**Question:**  Does Kotlin provide support for primitive Datatypes?

**Answer:** No. Kotlin does not provide support for primitive Data types like in Java.

**Question:**  Does Kotlin provide support for macros?

**Answer:** No. Kotlin does not provide support for macros because the developers of Kotlin find it difficult to include them in the language.

**Question:**  What is the use of the open keyword in Kotlin?

**Answer:** 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.

**Question:**  - What do you understand by the Ranges operator in Kotlin?

**Answer:** Ranges operators help to iterate within a range. Its operator form is (..) For Example:

for (i in 1..15)

print(i)

The above example will give the output to print from 1 to 15.

**Question:**  Where should we use var and where val in Kotlin?

**Answer:** In Kotlin, var is used where value is frequently changing. For example, while getting the location of the android device:

var integerVariable : Int? = null

In Kotlin, val is used where there is no change in value in the whole class. For example, when you want to set textview or button's text programmatically:

val stringVariables : String = "Button's Constant or final Text"

**Question:**  What is the difference between a safe calls(?.) and a null check(!!) in Kotlin?

**Answer:** Difference between safe calls(?.) and a null check(!!) in Kotlin:

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

var name: String? = "JavaTpoint"

println(name?.length) // 10

name = null

println(name?.length) // null

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

See the example:

var name: String? = "JavaTpoint"

println(name?.length) // 10

name = null

println(name!!.length) // KotlinNullPointerException

**Question:**  What is the basic difference between the fold and reduce in Kotlin? Also, specify when to use which?

**Answer:** Difference between the fold and reduce in Kotlin:

Fold: The fold takes an initial value and the first invocation of the lambda you pass to it. It will receive that initial value and the first element of the collection as parameters.

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

The first call to the lambda will be with parameters 0 and 1. The ability to pass in an initial value is useful if you have to provide a default value or parameter for your operation.

Reduce: The "reduce" doesn't take an initial value. Instead, it starts with the first element of the collection as the accumulator.

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

In the above example, it is denoted by sum. The first call to the lambda here will be with parameters 1 and 2.

**Question:**  What are the advantages of "when" over "switch" in Kotlin?

**Answer:** The "switch" is used in Java, but in Kotlin, that switch gets converted to "when". When has a better design as compared to "switch", and it is more concise and powerful than a traditional switch. We can use "when" either as an expression or as a statement.

Following are some examples of when usage in Kotlin:

In two or more choices:

when(number) {

1 -> println("One")

2, 3 -> println("Two or Three")

4 -> println("Four")

else -> println("Number is not between 1 and 4")

}

"when" without arguments:

when {

number < 1 -> print("Number is less than 1")

number > 1 -> print("Number is greater than 1")

}

Any type passed in "when":

fun describe(obj: Any): String =

when (obj) {

1 -> "One"

"Hello" -> "Greeting"

is Long -> "Long"

!is String -> "Not a string"

else -> "Unknown"

}

Smart casting:

when (x) {

is Int -> print("X is integer")

is String -> print("X is string")

}

Ranges:

when(number) {

1 -> println("One") //statement 1

2 -> println("Two") //statement 2

3 -> println("Three") //statement 3

in 4..8 -> println("Number between 4 and 8") //statement 4

!in 9..12 -> println("Number not in between 9 and 12") //statement 5

else -> println("Number is not between 1 and 8") //statement 6

}

**Question:**  What do you understand by the Null safety in Kotlin?

**Answer:** In Kotlin, the main motive of the type system is to eliminate the danger of null references from code. It is also known as the Billion Dollar Mistake.

One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference will result in a null reference exception. In Java, this would be the equivalent of a NullPointerException.

In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that cannot (non-null references). For example, a regular variable of type String can not hold null:

var a: String = "abc"

a = null // compilation error

To allow nulls, we can declare a variable as nullable string, written String?:

var b: String? = "abc"

b = null // ok

print(b)

**Question:**  Do we have a ternary operator in Kotlin just like Java?

**Answer:** No. In Kotlin, we don't have a ternary operator like Java, but we can use the functionality of the ternary operator by using if-else or Elvis operator.

**Question:**  What is Elvis operator in Kotlin?

**Answer:** In Kotlin, we can assign null values to a variable using the null safety property. To check if a value has null value, we can use if-else or can use the Elvis operator i.e. ?:

For example:

var name:String? = "Mindorks"

val namenameLength = name?.length ?: -1

println(nameLength)

In the above example, the Elvis operator(?:) we are using will return the length of the name if the value is not null; otherwise, if the value is null, then it will return -1.

**Question:**  Why is Kotlin interoperable with Java?

**Answer:** Kotlin is interoperable with Java because it uses JVM bytecode. It provides the facility to compile it directly to bytecode that helps to achieve faster compile-time and makes no difference between Java and Kotlin for JVM.

**Question:**  What do you understand by lazy initialization in Kotlin?

**Answer:** Kotlin provides the facility of lazy initialization, which specifies that your variable will not be initialized unless you use that variable in your code. It will be initialized only once. After that, you use the same value.

In lazy initialization, the lazy() function is used that takes a lambda and returns an instance of lazy, which can serve as a delegate for implementing a lazy property: the first call to get() executes the lambda passed to lazy() and remembers the result, subsequent calls to get() simply return the remembered result.

val test: String by lazy {

val testString = "some value"

}

**Question:**  - How many types of constructors are used in Kotlin?

**Answer:** There are two types of constructors available in Kotlin:

Primary constructor

Secondary constructor

**Question:**  What is Lateinit in Kotlin, and when is it used?

**Answer:** 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.

lateinit var test: String

fun doSomething() {

test = "Some value"

println("Length of string is "+test.length)

test = "change value"

}

This is mainly used in the following cases:

Android: variables that get initialized in lifecycle methods.

Using Dagger for DI: injected class variables are initialized outside and independently from the constructor.

Setup for unit tests: test environment variables are initialized in a @Before - annotated method.

Spring Boot annotations (e.g., @Autowired).

**Question:**  How can we convert a Kotlin source file to a Java source file?

**Answer:** Follow the steps given below to convert your Kotlin source file to a Java source file:

First, open your Kotlin project in the IntelliJ IDEA / Android Studio.

Then navigate to Tools > Kotlin > Show Kotlin Bytecode.

Now, click on the Decompile button to get your Java code from the bytecode.

**Question:**  What kinds of programming types does Kotlin support?

**Answer:** Kotlin supports the following programming types:

Procedural Programming

Object-Oriented Programming

**Question:**  What is the use of @JvmStatic, @JvmOverloads, and @JvmFiled in Kotlin?

**Answer:** Following are the main usage of @JvmStatic, @JvmOverloads, and @JvmFiled in Kotlin:

@JvmStatic: The @JvmStatic annotation is used to tell the compiler that the method is a static method, and you can use it in Java code.

@JvmOverloads: The @JvmOverloads annotation is required when we need to use the default values passed as an argument in Kotlin code from the Java code.

@JvmField: The @JvmField annotation is used to access the fields of a Kotlin class from Java code without any getters and setters. We need to use the @JvmField in the Kotlin code.

**Question:**

**Answer:**

**Question:**  What are the names of some extension methods that Kotlin provides to java.io.File?

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

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

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

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

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

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

**Question:**  What do you understand by data classes in Kotlin?

**Answer:** Data classes are the type of classes that are made to store some data. In Kotlin, it is marked as data. The following is an example of a data class:

data class Developer(val name: String, val age: Int)

When we mark a class as a data class, we don't have to implement or create the following functions like we have to do in Java: hashCode(), equals(), toString(), copy(). The compiler automatically creates these internally, so it also leads to clean code. Although, there are a few other requirements that data classes need to fulfill.

**Question:**  What is the use of Companion Objects in Kotlin?

**Answer:** Companion Objects are required in Kotlin because Kotlin doesn't have static members or member functions, unlike Java or C#. 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.

For example:

class EventManager {

companion object FirebaseManager {

}

}

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. If you integrate with Java code and need a true static member, you can annotate a member inside a companion object with @JvmStatic.

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

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

**Question:**  How can we perform String Interpolation in Kotlin?

**Answer:** In Kotlin, String Interpolation is used when you want to use some variable or perform some operation inside a string. For String Interpolation, we can use the $ sign to use some variable in the string or can perform some operation in between the {} sign.

For example:

var name = "JavaTpoint"

print("The best tutorial website is: $name")

**Question:**  Name some features which are available in Kotlin but not in Java?

**Answer:** Following are some important Kotlin features that are not available in Java:

Null Safety

Operator Overloading

Coroutines

Range expressions

Smart casts

Companion Objects

**Question:**  What is the difference between == operator and === operator in Kotlin?

**Answer:** 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:

// primitive example

val int1 = 10

val int2 = 10

println(int1 == int2) // true

println(int1 === int2) // true

// wrapper example

val num1 = Integer(10)

val num2 = Integer(10)

println(num1 == num2) // true

println(num1 === num2) //false

**Question:**  Can we use primitive types such as int, double, float in Kotlin?

**Answer:** 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. But the compiled bytecode has these primitive types.

**Question:**  What do you understand by destructuring in Kotlin?

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

For Example:

val (name, age) = developer

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

println(name)

println(age)

**Question:**  What are coroutines in Kotlin?

**Answer:** Unlike many other programming languages with similar capabilities, Kotlin doesn't have async and await keywords, and these keywords are not even part of its standard library.

In Kotlin, kotlinx.coroutines is a rich library for coroutines developed by JetBrains. This library contains some high-level coroutine-enabled primitives, including launch, async, and others. Kotlin Coroutines provide us with an API to write our asynchronous code sequentially.

According to Kotlin documentation, Coroutines are like lightweight threads. They are lightweight because while creating them, they don't allocate new threads. Instead, they use predefined thread pools and smart scheduling. Scheduling is the process of determining the work in a sequential process, and it decides which piece of work you will execute next. We can suspend and resume the Coroutines while execution. This means we can have a long-running task, which can be executed one by one. We can pause it any number of times and resume it when required.

**Question:**  What is the difference between Launch and Async in Kotlin Coroutines?

**Answer:** In Kotlin, the main difference between Launch and Async is that the launch{} does not return anything and the async{} returns an instance of Deferred, which has an await() function. In other words, we can say that launch is used to fire and forget, and async is used to perform a task and return a result.

**Question:**  What are the extension functions in Kotlin?

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

fun View.show() {

this.visibility = View.VISIBLE

}

fun View.hide() {

this.visibility = View.GONE

}

and to use it, we use, like,

toolbar.hide()

**Question:**  What do you understand by the Kotlin double-bang (!!) operator?

**Answer:** The Kotlin double-bang (!!) operator converts any value to a non-null type and throws a KotlinNullPointerException exception if the value is null. It is also called the not-null assertion operator.

Example:

fun main(args: Array) {

var email: String?

email = null

println(email!!)

}

This operator should be used in cases where the developer is 100% sure that its value is not null.

**Question:**  What is the difference between the variable declaration with var and val?

**Answer:** If you want to declare some mutable(changeable) variable, then you can use var. For the immutable variable, use val i.e. val variables can't be changed once assigned.

**Question:**  What is the difference between the variable declaration with val and const?

**Answer:** Both the variables that are declared with val and const are immutable in nature. But the value of the const variable must be known at the compile-time whereas the value of the val variable can be assigned at runtime also.

**Question:**  How to ensure null safety in Kotlin?

**Answer:** 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, the following code in Kotlin will produce a compile-time error:

var name: String = "MindOrks"

name = null //error

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

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

**Answer:** 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.

var name: String? = "MindOrks"

println(name?.length) // 8

name = null

println(name?.length) // null

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

**Question:**  How to convert a Kotlin source file to a Java source file?

**Answer:** Steps to convert your Kotlin source file to Java source file:

Open your Kotlin project in the IntelliJ IDEA / Android Studio.

Then navigate to Tools > Kotlin > Show Kotlin Bytecode.

Now click on the Decompile button to get your Java code from the bytecode.

**Question:**  What is String Interpolation in Kotlin?

**Answer:** If you want to use some variable or perform some operation inside a string then String Interpolation can be used. You can use the $ sign to use some variable in the string or can perform some operation in between {} sign.

var name = "MindOrks"

print("Hello! I am learning from $name")

**Question:**  What do you mean by destructuring in Kotlin?

**Answer:** Destructuring is a convenient way of extracting multiple values from data stored in(possibly nested) objects and Arrays. It can be used in locations that receive data (such as the left-hand side of an assignment). Sometimes it is convenient to destructure an object into a number of variables, for example:

val (name, age) = developer

Now, we can use name and age independently like below:

println(name)

println(age)

**Question:**  When to use the lateinit keyword in Kotlin?

**Answer:** Normally, properties declared as having a non-null type must be initialized in the constructor. However, fairly often this is not convenient.

For example, properties can be initialized through dependency injection, or in the setup method of a unit test. In this case, you cannot supply a non-null initializer in the constructor, but you still want to avoid null checks when referencing the property inside the body of a class. To handle this case, you can mark the property with the lateinit modifier.

**Question:**  How to check if a lateinit variable has been initialized or not?

**Answer:** You can check if the lateinit variable has been initialized or not before using it with the help of isInitialized method. This method will return true if the lateinit property has been initialized otherwise it will return false. For example:

class Person {

lateinit var name: String

fun initializeName() {

println(this::name.isInitialized)

name = "MindOrks" // initializing name

println(this::name.isInitialized)

}

}

fun main(args: Array) {

Person().initializeName()

}

**Question:**

**Answer:**

**Question:**  What is the difference between lateinit and lazy in Kotlin?

**Answer:** lazy can only be used for val properties, whereas lateinit can only be applied to var because it can’t be compiled to a final field, thus no immutability can be guaranteed.

If you want your property to be initialized from outside in a way probably unknown beforehand, use lateinit.

**Question:**  Is there any difference between == operator and === operator?

**Answer:** Yes. The == operator is 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. But in the case of primitive types, the === operator also checks for the value and not reference.

// primitive example

val int1 = 10

val int2 = 10

println(int1 == int2) // true

println(int1 === int2) // true

// wrapper example

val num1 = Integer(10)

val num2 = Integer(10)

println(num1 == num2) // true

println(num1 === num2) //false

**Question:**  What is the forEach in Kotlin?

**Answer:** In Kotlin, to use the functionality of a for-each loop just like in Java, we use a forEach function. The following is an example of the same:

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

listOfMindOrks.forEach {

Log.d(TAG,it)

}

**Question:**  What are companion objects in Kotlin?

**Answer:** In Kotlin, if you want to write a function or any member of the class that can be called without having the instance of the class then you can write the same as a member of a companion object inside the class.

To create a companion object, you need to add the companion keyword in front of the object declaration.

The following is an example of a companion object in Kotlin:

class ToBeCalled {

companion object Test {

fun callMe() = println("You are calling me :)")

}

}

fun main(args: Array) {

ToBeCalled.callMe()

}

**Question:**  What is the equivalent of Java static methods in Kotlin?

**Answer:** To achieve the functionality similar to Java static methods in Kotlin, we can use:

companion object

package-level function

object

**Question:**  What is the difference between FlatMap and Map in Kotlin?

**Answer:** FlatMap is used to combine all the items of lists into one list. Map is used to transform a list based on certain conditions.

**Question:**  What is the difference between List and Array types in Kotlin?

**Answer:** If you have a list of data that is having a fixed size, then you can use an Array. But if the size of the list can vary, then we have to use a mutable list.

**Question:**  Can we use the new keyword to instantiate a class object in Kotlin?

**Answer:** No, in Kotlin we don't have to use the new keyword to instantiate a class object. To instantiate a class object, simply we use: var varName = ClassName()

**Question:**  What are visibility modifiers in Kotlin?

**Answer:** A visibility modifier or access specifier or access modifier is a concept that is used to define the scope of something in a programming language. In Kotlin, we have four visibility modifiers:

private: visible inside that particular class or file containing the declaration.

protected: visible inside that particular class or file and also in the subclass of that particular class where it is declared.

internal: visible everywhere in that particular module.

public: visible to everyone.

**Question:**  How to create a Singleton class in Kotlin?

**Answer:** A singleton class is a class that is defined in such a way that only one instance of the class can be created and is used where we need only one instance of the class like in logging, database connections, etc.

To create a Singleton class in Kotlin, you need to use the object keyword.

object AnySingletonClassName

**Question:**  What are init blocks in Kotlin?

**Answer:** init blocks are initializer blocks that are executed just after the execution of the primary constructor. A class file can have one or more init blocks that will be executed in series. If you want to perform some operation in the primary constructor, then it is not possible in Kotlin, for that, you need to use the init block.

**Question:**  What are the types of constructors in Kotlin?

**Answer:** Primary constructor: These constructors are defined in the class header and you can't perform some operation in it, unlike Java's constructor. Secondary constructor: These constructors are declared inside the class body by using the constructor keyword. You must call the primary constructor from the secondary constructor explicitly. Also, the property of the class can’t be declared inside the secondary constructor. There can be more than one secondary constructors in Kotlin.

**Question:**  Is there any relationship between primary and secondary constructors?

**Answer:** Yes, when using a secondary constructor, you need to call the primary constructor explicitly.

**Question:**  What is the default type of argument used in a constructor?

**Answer:** By default, the type of arguments of a constructor in val. But you can change it to var explicitly.

**Question:**  What is suspend function in Kotlin Coroutines?

**Answer:** Suspend function is the building block of the Coroutines in Kotlin. Suspend function is a function that could be started, paused, and resume. To use a suspend function, we need to use the suspend keyword in our normal function definition.

**Question:**  How to choose between a switch and when in Kotlin?

**Answer:** Whenever we want to handle many if-else conditions, then we generally use switch-case statements. But Kotlin provides a more concise option i.e. in Kotlin, we can use when in place of the switch. And, when can be used as:

expression

arbitrary condition expression

without argument

with two or more choices

For example:

when(number) {

1 -> println("One")

2, 3 -> println("Two or Three")

4 -> println("Four")

else -> println("Number is not between 1 and 4")

}

**Question:**  What is the open keyword in Kotlin used for?

**Answer:** By default, the classes and functions are final in Kotlin. So, you can't inherit the class or override the functions. To do so, you need to use the open keyword before the class and function.

**Question:**  What are lambdas expressions?

**Answer:** Lambdas expressions are anonymous functions that can be treated as values i.e. we can pass the lambdas expressions as arguments to a function return them, or do any other thing we could do with a normal object. For example:

val add : (Int, Int) -> Int = { a, b -> a + b }

val result = add(9, 10)

**Question:**  What are Higher-Order functions in Kotlin?

**Answer:** A higher-order function is a function that takes functions as parameters or returns a function. For example, A function can take functions as parameters.

fun passMeFunction(abc: () -> Unit) {

// I can take function

// do something here

// execute the function

abc()

}

For example, A function can return another function.

fun add(a: Int, b: Int): Int {

return a + b

}

And, we have a function returnMeAddFunction which takes zero parameters and returns a function of the type ((Int, Int) -> Int).

fun returnMeAddFunction(): ((Int, Int) -> Int) {

// can do something and return function as well

// returning function

return ::add

}

And to call the above function, we can do:

val add = returnMeAddFunction()

val result = add(2, 2)

**Question:**  What are extension functions in Kotlin?

**Answer:** Extension functions are like extensive properties attached to any class in Kotlin. By using extension functions, you can add some methods or functionalities to an existing class even without inheriting the class. For example: Let's say, we have views where we need to play with the visibility of the views. So, we can create an extension function for views like,

fun View.show() {

this.visibility = View.VISIBLE

}

fun View.hide() {

this.visibility = View.GONE

}

and to use it we use, like,

toolbar.hide()

**Question:**  What is an infix function in Kotlin?

**Answer:** An infix function is used to call the function without using any bracket or parenthesis. You need to use the infix keyword to use the infix function.

class Operations {

var x = 10;

infix fun minus(num: Int) {

this.x = this.x - num

}

}

fun main() {

val opr = Operations()

opr minus 8

print(opr.x)

}

**Question:**  What is an inline function in Kotlin?

**Answer:** 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.

**Question:**  What is noinline in Kotlin?

**Answer:** While using an inline function and want to pass some lambda function and not all lambda function as inline, then you can explicitly tell the compiler which lambda it shouldn't inline. inline fun doSomethingElse(abc: () -> Unit, noinline xyz: () -> Unit) { abc() xyz() }

**Question:**  What are Reified types in Kotlin?

**Answer:** When you are using the concept of Generics to pass some class as a parameter to some function and you need to access the type of that class, then you need to use the reified keyword in Kotlin.

For example:

inline fun genericsExample(value: T) {

println(value)

println("Type of T: ${T::class.java}")

}

fun main() {

genericsExample("Learning Generics!")

genericsExample(100)

}

**Question:**  What is the operator overloading in Kotlin?

**Answer:** In Kotlin, we can use the same operator to perform various tasks and this is known as operator overloading. To do so, we need to provide a member function or an extension function with a fixed name and operator keyword before the function name because normally also, when we are using some operator then under the hood some function gets called. For example, if you are writing num1+num2, then it gets converted to num1.plus(num2).

For example:

fun main() {

val bluePen = Pen(inkColor = "Blue")

bluePen.showInkColor()

val blackPen = Pen(inkColor = "Black")

blackPen.showInkColor()

val blueBlackPen = bluePen + blackPen

blueBlackPen.showInkColor()

}

operator fun Pen.plus(otherPen: Pen):Pen{

val ink = "$inkColor, ${otherPen.inkColor}"

return Pen(inkColor = ink)

}

data class Pen(val inkColor:String){

fun showInkColor(){ println(inkColor)}

}

**Question:**  What are pair and triple in Kotlin?

**Answer:** Pair and Triples are used to return two and three values respectively from a function and the returned values can be of the same data type or different.

val pair = Pair("My Age: ", 25)

print(pair.first + pair.second)

**Question:**  What are labels in Kotlin?

**Answer:** Any expression written in Kotlin is called a label. For example, if we are having a for-loop in our Kotlin code then we can name that for-loop expression as a label and will use the label name for the for-loop.

We can create a label by using an identifier followed by the @ sign. For example, name@, loop@, xyz@, etc. The following is an example of a label:

loop@ for (i in 1..10) {

// some code goes here

}

The name of the above for-loop is loop.

**Question:**  What are the benefits of using a Sealed Class over Enum?

**Answer:** Sealed classes give us the flexibility of having different types of subclasses and also containing the state. The important point to be noted here is the subclasses that are extending the Sealed classes should be either nested classes of the Sealed class or should be declared in the same file as that of the Sealed class.

**Question:**  Why kotlin is better than Java?

**Answer:** Kotlin is a simple general-purpose language, in which there is a code redundancy when compared to java. There are many features in kotlin like null safety, extension functions, primary constructors, inline or lambda expressions, properties, and type interference for the properties and variables.

**Question:**  Explain functions in kotlin?

**Answer:** Functions used in kotlin are simple to store in data structures and variables and can possess arguments that are passed from the high ordered data functions.

An example of the sample function declaration in kotlin:

function double(y:int):

Int

{

return 3 \* y

}

val reslt = double(3)

**Question:**  What are the high order functions?

**Answer:** High order functions consider functions as a parameter and produce a function.

**Question:**  How to convert a String into an int in the kotlin?

**Answer:** To convert a string value to the string value to it in kotlin we use point() method.

Let us see an example:

function main(args: array)

{

val s: str]ng = "Kotlin"

var y = 10

y = "10".toint()

}

**Question:**  List out the basic data types we use in kotlin?

**Answer:** The data type of a constant or a variable decides what is the type of a variable and how much space needs to store. Types of data that a string allow are:

Strings

Numbers

Arrays

Booleans

Characteristics

**Question:**  What is the difference between and interface and an abstract class?

**Answer:** Though both may seem very similar they are in fact very different and serve very different purposes. Abstract classes are meant to serve the purpose of Generalizing behavior while interfaces are meant to serve the purpose of Standardizing behavior.

**Question:**  What is the difference between Java field and Kotlin property?

**Answer:** This is an example of a Java field:

public String name = "Marcin";

Here is an example of a Kotlin property:

var name: String = "Marcin"

They both look very similar, but these are two different concepts. Direct Java equivalent of above Kotlin property is following:

private String name = "Marcin";

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

The default implementation of Kotlin property includes field and accessors (getter for val, and getter and setter for var). Thanks to that, we can always replace accessors default implementation with a custom one.

**Question:**  What is the difference between launch/join and async/await in Kotlin coroutines?

**Answer:** launch is used to\*\* fire and forget coroutine\*\*. It is like starting a new thread. If the code inside the launch terminates with exception, then it is treated like uncaught exception in a thread -- usually printed to stderr in backend JVM applications and crashes Android applications. join is used to wait for completion of the launched coroutine and it does not propagate its exception. However, a crashed child coroutine cancels its parent with the corresponding exception, too.

async is used to start a coroutine that computes some result. The result is represented by an instance of Deferred and you must use await on it. An uncaught exception inside the async code is stored inside the resulting Deferred and is not delivered anywhere else, it will get silently dropped unless processed. You MUST NOT forget about the coroutine you’ve started with async.

**Question:**  How Kotlin coroutines are better than RxKotlin/RxJava?

**Answer:** Kotlin coroutines are different from Rx. Both are designed to address a problem of asynchronous programming, however their approach to solution is very different:

Rx comes with a particular functional style of programming that can be implemented in virtually any programming language without support from the language itself. It works well when the problem at hand easily decomposes into a sequence of standard operators and not so well otherwise.

Kotlin coroutines provide a language feature that let library writers implement various asynchronous programming styles, including, but not limited to functional reactive style (Rx). With Kotlin coroutines you can also write your asynchronous code in imperative style, in promise/futures-based style, in actor-style, etc.

How Kotlin coroutines are better than RxKotlin? You just write sequential code, everything is as easy as writing synchronous code except it execute asynchronously. It's easier to grasp.

Coroutines are better to deal with resources

In RxJava you can assign computations to schedulers but subscribeOn() and ObserveOn()are confusing. Every coroutine is given a thread context and return to parent context. For a channel, both side (producer, consumer) execute on his own context. Coroutines are more intuitive on thread or thread pool affectation.

Coroutines give more control on when those computation occur. You can for example pass hand (yield), prioritize (select), parallelize (multiple producer/actor on channel) or lock resource (Mutex) for a given computation. It may not matter on server (where RxJava came first) but on resources limited environment this level of control may be required.

Due to it's reactive nature, backpressure doesn't fit well in RxJava. In the other end send() to channel is a suspensive function that suspend when channel capacity is reached. It's out-of-the-box backpressure given by nature. You could also offer() to channel, in which case the call never suspend but return false in case the channel is full, effectively reproducing onBackpressureDrop() from RxJava. Or you could just write your own custom backpressure logic, which won't be difficult with coroutines, especially compared to do the same with RxJava.

**Question:**  How to implement Builder pattern in Kotlin?

**Answer:** First and foremost, in most cases you don't need to use builders in Kotlin because we have default and named arguments but if you need one use:

class Car( //add private constructor if necessary

val model: String?,

val year: Int

) {

private constructor(builder: Builder) : this(builder.model, builder.year)

class Builder {

var model: String? = null

private set

var year: Int = 0

private set

fun model(model: String) = apply { this.model = model }

fun year(year: Int) = apply { this.year = year }

fun build() = Car(this)

}

}

Usage:

val car = Car.Builder().model("X").build()

**Question:**  How does the reified keyword in Kotlin work?

**Answer:** In an ordinary generic function like myGenericFun, you can't access the type T because it is, like in Java, erased at runtime and thus only available at compile time. Therefore, if you want to use the generic type as a normal Class in the function body you need to explicitly pass the class as a parameter like the parameter c in the example.

fun myGenericFun(c: Class)

By marking a type as reified, we’ll have the ability to use that type within the function.

As for a real example, in Java, when we call startActivity, we need to specify the destination class as a parameter. The Kotlin way is:

inline fun Activity.startActivity() {

startActivity(Intent(this, T::class.java))

}

You can only use reified in combination with an inline function. Such a function makes the compiler copy the function's bytecode to every place where the function is being used (the function is being "inlined"). When you call an inline function with reified type, the compiler knows the actual type used as a type argument and modifies the generated bytecode to use the corresponding class directly. Therefore calls like myVar is T become myVar is String (if the type argument were String) in the bytecode and at runtime.

**Question:**  What is a motivation to make classes final by default in Kotlin? Do you agree with that decision?

**Answer:** First Kotlin takes many ideas from the functional programming world and uses immutability as often as it can to avoid all the known problems with mutation. Also proper designing a class for inheritance requires an excruciating amount of work (and building at least 3 separate subclasses, to verify that you base class actually is useful). Most classes should be final. Extending them is probably a bad idea.

The second thought which comes to my mind is that inheritance is often missused. There is the principle "Favor composition over inheritance" as a guideline for better designs. So declaring every class as final by default forces the developer to at least stop for a moment and think about alternative ways to solve the problem instead of using inheritance for the wrong reasons.

**Question:**  What is The Billion Dollar Mistake?

**Answer:** Kotlin's type system is aimed at eliminating the danger of null references from code, also known as the The Billion Dollar Mistake.

One of the most common pitfalls in many programming languages, including Java, is that accessing a member of a null reference will result in a null reference exception. In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that can not (non-null references).

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object-oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn’t resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

**Question:**  Why would you use apply in Kotlin?

**Answer:** Look at this code:

person.name = "Tony Stark"

person.age = 52

and equivalent with apply will be:

val person = Person().apply {

name = "Tony Stark" // this. can be omitted

age = 52 // this. can be omitted

// ...

}

This way you don't have to repeat person several times. Apply is used to keep things that belong together in one place (mostly initializations).

**Question:**  Provide a real use case when inline classes may be useful

**Answer:** Imagine an authentication method in an API that looks as follows:

fun auth(userName: String, password: String) { println("authenticating $userName.") }

Since both parameters are of type String, you may mess up their order which gets even more likely with an increasing number of arguments.

Inline class wrappers around these types can help you mitigate that risk and give you simple, type-safe wrappers without introducing additional heap allocations:

inline class Password(val value: String)

inline class UserName(val value: String)

fun auth(userName: UserName, password: Password) { println("authenticating $userName.")}

fun main() {

auth(UserName("user1"), Password("12345"))

//does not compile due to type mismatch

auth(Password("12345"), UserName("user1"))

}

**Question:**  Explain the difference between lateinit and lazy in details

**Answer:** Here are the significant differences between lateinit var and by lazy { ... } delegated property:

lazy { ... } delegate can only be used for val properties, whereas lateinit can only be applied to vars, because it can't be compiled to a final field, thus no immutability can be guaranteed;

lateinit var has a backing field which stores the value, and by lazy { ... } creates a delegate object in which the value is stored once calculated, stores the reference to the delegate instance in the class object and generates the getter for the property that works with the delegate instance. So if you need the backing field present in the class, use lateinit;

In addition to vals, lateinit cannot be used for nullable properties and Java primitive types (this is because of null used for uninitialized value);

lateinit var can be initialized from anywhere the object is seen from, e.g. from inside a framework code, and multiple initialization scenarios are possible for different objects of a single class. by lazy { ... }, in turn, defines the only initializer for the property, which can be altered only by overriding the property in a subclass. If you want your property to be initialized from outside in a way probably unknown beforehand, use lateinit.

Initialization by lazy { ... } is thread-safe by default and guarantees that the initializer is invoked at most once (but this can be altered by using [another lazy overload][1]). In the case of lateinit var, it's up to the user's code to initialize the property correctly in multi-threaded environments.

A Lazy instance can be saved, passed around and even used for multiple properties. On contrary, lateinit vars do not store any additional runtime state (only null in the field for uninitialized value).

If you hold a reference to an instance of Lazy, [isInitialized()][2] allows you to check whether it has already been initialized (and you can [obtain such instance with reflection][3] from a delegated property). To check whether a lateinit property has been initialized, you can [use property::isInitialized since Kotlin 1.2][4].

A lambda passed to by lazy { ... } may capture references from the context where it is used into its [closure][5].. It will then store the references and release them only once the property has been initialized. This may lead to object hierarchies, such as Android activities, not being released for too long (or ever, if the property remains accessible and is never accessed), so you should be careful about what you use inside the initializer lambda.

**Question:**

**Answer:**

**Question:**  Explain Higher-Order Functions?

**Answer:** Higher-Order Functions: A higher-order function is a function that takes functions as parameters, or returns a function.

**Question:**  Explain Functions In Kotlin?

**Answer:** Kotlin functions are first-class functions that are easily stored in variables and data structures and can be pass as arguments and returned from other higher-order functions.

Sample function declaration and usage in Kotlin

fun double(x: Int): Int {

return 2 \* x

}

val result = double(2)

**Question:**  How to convert a String to an Int in Kotlin?

**Answer:** toInt() method is used to convert a string value to integer or INT in Kotlin. Below is example uses

fun main(args: Array) {

val s: String = “Kotlin”

var x = 10

x = “8”.toInt()

}

**Question:**  What is the difference between declaration variable using val or var in Kotlin?

**Answer:** In Kotlin a variable declared using val keyword is cannot be changed. It is similar to the final modifiers in Java whereas the variables declared using var keywords can be reassigned.

**Question:**  How to Declare a Variable in Kotlin?

**Answer:** In Kotlin, you can declare a variable using var or val which followed by an optional datatype.

Variable declaration in Kotlin looks like:

val s: String = “Hi”

var x = 5

**Question:**  List the Basic data types of Kotlin?

**Answer:** Data types of a constant or variable decide what type of variable it is and how much space is required to store it.

The basic data types in Kotlin are:

Numbers

Characters

Strings

Arrays

Booleans

**Question:**  Why is Kotlin preferred over Java?

**Answer:** Kotlin eases the coding process as it is simpler than Java and has many features required, that is not provided by Java yet like Extension functions, Null Safety, range expressions etc. In Kotlin, we code approximately 40% less number of code lines as compared with Java.

**Question:**  Where does the Kotlin run and what is the entry point of Kotlin?

**Answer:** The Kotlin program once compiled, can run on standard JVM like other programming codes.And, like many other programming languages main() function is the entry point of the Kotlin.

**Question:**  What are the different types of constructors in Kotlin?

**Answer:** There are two types of constructors in Kotlin:

Primary constructor: It is a section of the Class header and is declared after the class name.

Secondary constructor: This constructor is declared inside the body.

Note: There can be more secondary constructors for a class.

**Question:**  Can you execute Kotlin code without JVM?

**Answer:** JVM, which stands for Java Virtual Machine is a feature of Kotlin. This feature compiles a Kotlin code into a native code, which can be done without JVM too.

**Question:**  Mention the structural expressions in Kotlin?

**Answer:** There are three Structural expressions in Kotlin.They are:

Return: It returns from the nearest enclosing function or anonymous function by default.

Break: This expression terminates the closest enclosing loop.

Continue: This expression proceeds you to the next closest enclosing loop.

**Question:**  Explain the data classes in Kotlin?

**Answer:** In programming, we use classes to hold data and these classes are called as data classes. An object can be initialized in the data class and to access the individual parameters of these data classes, we use component functions.

**Question:**  What are the modifiers that are available in Kotlin?

**Answer:** The modifier in Kotlin provides the developer to customize the declarations as per the requirements. Kotlin provides four modifiers. They are:

Private: This makes the declaration visible only inside the file containing declaration.

Public: It is by default, which means that the declarations will be visible everywhere.

Internal: This makes the declaration visible everywhere in the same modules.

Protected: This keeps the declaration protected and is not available for top-level declarations.

**Question:**  Can you migrate the code from Java to Kotlin? If yes how do you do it?

**Answer:** Yes, we can migrate the code from Java to Kotlin.This can be done using JetBrains IDEA, which facilitates the conversion of Java code to Kotlin code.

**Question:**  State the differences between Val and Var?

**Answer:** Val: Val, which is the short form of value, is a constant and it cannot be changed once assigned.

Var: Var, which is the short form of variable, is a storage location that accepts the reassignment of values that have the same data types.

**Question:**  Explain Kotlin’s Null safety?

**Answer:** In Kotlin, the Null safety is used to eliminate the risk of countering the NullPointer exception in real time.

**Question:**  What are the types of strings available in Kotlin? And, what do you mean by Kotlin String Interpolation?

**Answer:** Strings are a collection of characters together.Kotlin features two types of strings, and they are:

Raw string

Escaped string

In Kotlin String, templates can be evaluated.This evaluation of string templates is called as the string template interpolation.

**Question:**  State the advantages and disadvantages of Kotlin?

**Answer:** Advantages:

Kotlin is simple and easy to learn as its syntax is similar to that of Java.

It is the functional language that is based on JVM (Java Virtual Machine), which removes the boilerplate codes. Upon all this, Kotlin is considered as an expressive language that is easily readable and understandable and the performance is substantially good.

It can be used by any desktop, web server or mobile based applications.

Disadvantages:

Kotlin does not provide the static modifier, which causes problems for conventional java developer.

In Kotlin, the function declaration can be done in many places in the application, which creates the trouble for the developer to understand which function is being called.

**Question:**  What is Kotlin?

**Answer:** It is an open source programming language that combines object-oriented programming features. The features like Range Expression, Extension Function, Companion Object, Smart casts, Data classes are considered to be surplus of the Kotlin Language.

**Question:**  . Which type of Programming does Kotlin support?

**Answer:** Kotlin supports only two types of programming, and they are:

Procedural programming

Object-oriented programming

**Question:**  Why did you switch to Kotlin from Java ?

**Answer:** Kotlin seems to be simpler and cleaner than Java. It removes a lot of redundancies in code from Java. Kotlin also adds some needed features that Java doesn’t yet support, and is making code more idiomatic. Also Kotlin has been added to Android Studio’s list of supported languages recently. So, there is much to expect from Kotlin in easing out the development efforts and good support in future.

**Question:**  What are the features you think are there in Kotlin but not in Java ?

**Answer:** Kotlin has quite a number of features that Java doesn’t. To name some of them, they are

Extension Functions

Null Safety

Smart casts

Range expressions

Operator Overloading

Data classes

Companion Objects

Coroutines

etc.

**Question:**  What kinds of programming does Kotlin support ?

**Answer:** Kotlin supports two types of programming. They are

Procedural Programming

Object Oriented Programming

Q4 – What is the entry point to a Kotlin program ? Provide an example.

Like most of the other procedural languages, main() function is the entry point to a Kotlin program.

**Question:**  How do you think extension functions are useful ?

**Answer:** Extension functions helps to extend a class with new functionality without having to inherit from the class. Also you may use them like an inbuilt function for the class throughout the application.

**Question:**  What are Data classes ? Aren’t they available in Java ?

**Answer:** Sometimes we use a class just to hold the data and nothing else. These classes are called Data classes. Of course these kind of classes could be built using Java, but with explicit implementation of getter and setter for each of the properties of class. Also you may need to implement functions like equals, toString and copy separately. What Kotlin does is implementing all these automatically along with special functions called component functions. How cool is that, removing the redundant code bloat.

**Question:**  Does Kotlin provide any additional functionalities for standard Java packages or standard Java classes?

**Answer:** Ofcourse, Yes. Kotlin uses the concept of extension functions, that we already talked about, to build some useful and more widely used functions among developers directly into the Kotlin library.

**Question:**  Where does this Kotlin run ? Does it have some kind of different runtime environment ?

**Answer:** Once compiled, Kotlin programs can run on standard JVM like some other compiled Java code. This means that Kotlin Compiler compiles Kotlin programs to byte-code, which is understood by JVM. So, Kotlin is like a flavor of Java, that goes alongside Java. Interesting fact is that, Kotlin applications can be built with parts of Java code.

**Question:**  So, how do you migrate the code from Java to Kotlin ?

**Answer:** JetBrains IDEA provides inbuilt tools to convert Java code to Kotlin code. Then you may do the magic offered by Kotlin at some of the parts in code, to make it clean.

**Question:**  OK. Is there something called init block in Kotlin ?

**Answer:** Yes

**Question:**  What does init block do and Where does it appear in a class ?

**Answer:** Instructions in the init block are executed right after Primary Constructor’s execution. init block goes in a class along with secondary constructors as a method.

Reference – Kotlin Init

**Question:**  How many types of constructors are there ? What are they ?

**Answer:** There are two types of constructors. They are Primary Constructors and Secondary Constructors.

**Question:**  How are Primary Constructors different from Secondary Constructors ?

**Answer:** Primary Constructors are declared intrinsically with class definition. Secondary Constructors are declared exclusively inside the class body.

In the following example, in the first line, the constructor keyword along with the variables declared right after it is the Primary Constructor. Inside the class body, we have another constructor, and this is Secondary Constructor.

Example of class with Primary and Secondary Constructors

class Person constructor(var name: String, var age: Int){

var profession: String = “Not Mentioned”

constructor (name: String, age: Int, profession: String): this(name,age){

this.profession = profession

}

}

**Question:**  Is there any dependency of Secondary Constructors on Primary Constructors ?

**Answer:** Yes. Secondary Constructor has to make an exclusive call to Primary Constructor or other Secondary Constructor, which of course calls the Primary Constructor. Following is an example, and here the Secondary Constructor makes call to Primary Constructor using this(name, age).

Example of class with Primary and Secondary Constructors

class Person constructor(var name: String, var age: Int){

var profession: String = “Not Mentioned”

constructor (name: String, age: Int, profession: String): this(name,age){

this.profession = profession

}

fun printPersonDetails(){

println(“$name whose profession is $profession, is $age years old.”)

}

}

**Question:**  What is the difference between val and var ?

**Answer:** Val (Value) is like a constant. Once assigned a value, you cannot change it. On the other hand Var (Variable) is designed to be a storage location that can accept reassignment of values of same data type or what ever feasible by the data type casting.

**Question:**  What is Kotlin’s Null Safety ?

**Answer:** Null Safety in Kotlin is to eliminate the risk of occurrence of NullPointerException in real time. Kotlin can differentiate between nullable references and non-nullable references. If a variable has to be allowed to store a null value, that has to be declared with a null (?) operator.

**Question:**  If you have worked with files, name some of the extension methods Kotlin provides to java.io.File

**Answer:** Kotlin provides very useful extension functions to java.io.File. Some of them are :

File.bufferedReader() : to read contents of a file into BufferedReader

File.forEachLine() : to read a file line by line in Kotlin

File.inputStream() : to read contents of file to InputStream

File.readBytes() : to read contents of file to ByteArray

File.readLines() : to read lines in file to List

File.readText() : to read contents of file to a single String

For examples to these methods refer – Kotlin Read File Content

**Question:**  Is there Ternary Conditional Operator in Kotlin like in Java ?

**Answer:** No

**Question:**  How do you realize Ternary Conditional Operator in Kotlin ?

**Answer:** A simple if else should do the job.

if (condition) a else b

**Question:**  How do you declare a variable as volatile in Kotlin ?

**Answer:** By providing volatile annotation before the declaration of variable. @Volatile var a: Long? = null

**Question:**  How do you check if two Strings are equal valued ?

**Answer:** Using == (double equal to) operator.

fun main(args: Array<String>) {

val a: String = “kotlin is easy”

val b: String = “kotlin is” + ” easy”

if(a==b){

println(” a and b are equal.”)

} else {

println(” a and b are not equal.”)

}

}

**Question:**  How do you define the Target Platform of Kotlin and how is Kotlin-Java interoperability possible?

**Answer:** Java Virtual Machine(JVM) is the Target Platform of Kotlin. Such that Kotlin is 100% interoperable with Java since both, on compilation produce bytecode. Therefore Kotlin code can be referred from Java and vice-versa.

**Question:**  Differentiate between const and a val?

**Answer:** Some of the points of difference between const and val are -

1. By default val properties are set at runtime. Therefore by adding a const modifier on a val would make a compile-time constant.

2.A const cannot be used with a var or on its own.

3 A const is not applicable on a local variable.

**Question:**  Does Kotlin permit the use of primitive types like int, float, double?

**Answer:** Kotlin does not permit the use of primitive types like int, float, double at the language level. The JVM bytecode that’s compiled does certainly have them.

**Question:**  What is the entry point of every Kotlin Program?

**Answer:** The primary function is the entry point of every Kotlin program. Such that in Kotlin we can choose not to write the main function inside the class. Thereby on compiling the JVM implicitly encapsulates it in a class. Also the strings passed in the form of Array are used to retrieve the command line arguments.

**Question:**  Differentiate between == and === operators in Kotlin?

**Answer:** == is used to compare the values are equal or not. On the other hand === is used to check if the references are equal or not.

**Question:**  What are the visibility modifiers available in Kotlin and what is the default visibility modifier?

**Answer:** Some of the visibility modifiers available in Kotlin are - public, internal, protected, and private. Also 'public' is the default visibility modifier.

**Question:**  Name the types of constructors in Kotlin?

**Answer:** Mainly constructors in Kotlin are of two types namely -

1. Primary constructors – These are defined in the class headers. Such that they cannot hold any logic. There’s only one primary constructor per class.

2. Secondary constructors – They are 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.

**Question:**  How do you define 'init block' in Kotlin?

**Answer:** init is the initialiser block in Kotlin which is executed once the primary constructor is instantiated. Such that if you invoke a secondary constructor, then it works after the primary one as it is composed in the chain.

**Question:**  Name the different type of arguments inside a constructor?

**Answer:** The constructor arguments are val unless explicitly set to var.

**Question:**  Differentiate between lazy and lateinit.

**Answer:** The points of difference between lazy and lateinit are -

1. Both are used to delay the property initializations in Kotlin

On one hand lateinit is a modifier used with var and is used to set the value to the var at a later point whereas 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

**Question:**  What are the advantages of using Kotlin?

**Answer:** The advantages of using Kotlin are -

1. Kotlin language is easy to learn as its syntax is similar to Java.

2. Kotlin is a functional language and based on JVM. So, it removes lots of boiler plate

3.It is an expressive language which makes code readable and understandable.

**Question:**  How do you define ‘Null Safety’ in Kotlin?

**Answer:** In Kotlin, Null Safety feature permits to remove the risk of occurrence of NullPointerException in real time. Such that it is also possible to differentiate between nullable references and non-nullable references.

**Question:**  Why is Kotlin interoperable with Java?

**Answer:** The reason for Kotlin to interoperable with Java is sine it uses JVM bytecode. Therefore the process of compiling it directly to bytecode helps to achieve faster compile time and makes no difference between Java and Kotlin for JVM.

**Question:**  What are the extension methods Kotlin provides to java.io.File?

**Answer:** The extension methods Kotlin provides to java.io.File are -

1. bufferedReader(): Use for reading contents of a file into BufferedReader

2. readBytes() : Use for reading contents of file to ByteArray

3. readText(): Use of reading contents of file to a single String

4. forEachLine() : Use for reading a file line by line in Kotlin

5. readLines(): Use to reading lines in file to List

**Question:**  What is the default behavior of Kotlin classes?

**Answer:** In Kotlin all classes are final by default since Kotlin permits multiple inheritances for classes, and an open class is more expensive than a final class.

**Question:**  Does Kotlin offer any additional functionality for standard Java packages or classes?

**Answer:** Kotlin programs can easily run on standard JVM like any another compiled Java code. It permits JVM to compile any program to byte-code. Kotlin is accessible using Java Virtual Machine. Therefore, Kotlin is almost similar to Java. Also Kotlin applications can be built with parts of Java code.

**Question:**  Define the use of abstraction in Kotlin.

**Answer:** Abstraction is defined as one of the most important concept of Objected Oriented Programming. In Kotlin, abstraction class is used when you know the functionalities a class should have. One may not be aware of how the functionality is implemented or if the functionality can be implemented using different methods.

**Question:**  Describe the default behavior of Kotlin classes?

**Answer:** All classes are final by default, in Kotlin because Kotlin allows multiple inheritances for classes, and an open class is more expensive than a final class.

**Question:**  Describe the features that are there in Kotlin but not In Java.

**Answer:** Some of the important Kotlin features that Java does not have -

1. Null Safety

2. Operator Overloading

3. Coroutines

4. Range expressions

5. Smart casts

6. Companion Objects

**Question:**  What are Higher-Order Functions?

**Answer:** Higher-Order Functions can be defined as a function that takes functions as parameters, or returns a function.

**Question:**  Name the different types of strings available in Kotlin and what do you understand by Kotlin String Interpolation?

**Answer:** The different types of strings available in Kotlin are -

1. Strings are a collection of characters together. Such that Kotlin features two types of strings, and they are - Raw string and Escaped string

In Kotlin String, templates can be evaluated. This evaluation of string templates is referred as the string template interpolation

**Question:**  What is Kotlin Native?

**Answer:** Kotlin Native is technology for compiling kotlin code to native binaries which can run without virtual machine, like iOS(arm32, arm64), MacOS(x86\_64), Android(arm32, arm64), Linux(x86\_64, arm32).

**Question:**  Define Kotlin Programming Language?

**Answer:** Kotlin is a general-purpose, open-source programming language. It can be used to develop android apps, server-side apps and much more. It is a high level strongly, statically typed language and its code is more concise, readable and easy to understand. Its popularity has grown day by day.

**Question:**  Who created Kotlin programming language and when?

**Answer:** Kotlin was developed by JetBrains team. This project development started in 2010 and officially, first released in February 2016. This language was developed under the Apache 2.0 license.

**Question:**  What are the major features of Kotlin?

**Answer:** These are some major features of Kotlin.

Kotlin is open source, statically typed programming language.

As it is concise language, it is very easy to read and understand the code pattern.

In a very short period of time, it can give better performance.

It is interoperable and null safety language, it eliminates the NullPointerException.

Kotlin is compatible with all the Java based framework.

**Question:**  Differenciate between val and var?

**Answer:** The var is equivalent to declaring a variable in Java, it is a mutable variable, which can be changed to another value by reassigning it. But, val is an immutable variable or read only variable which cannot be changed later in the program once the value is assigned. They can be assigned a value only once. This is similar to the final variable of Java.

**Question:**  Give example to declare variable using both val and var?

**Answer:** fun main(args : Array<String>) {

var x: String = "Chair"

x = "Table" //valid

val y: String = "Bed"

println("$x and $y")

}

**Question:**  Define String datatype with example?

**Answer:** The **string** is a sequence of characters enclosed within double and triple quotes. It is immutable means you cannot change the individual character of a string. It is represented by the type String.

fun main(args : Array<String>) {

val str1 = "Welcome Alia"

var str2 = """We hope you will enjoy more."""

println("$str1 $str2")

}

**Question:**  What is inner class?

**Answer:** A inner class can access members of outer class. It carries a reference to an object of an outer class. Just like nested class, it is created inside an outer class and unlike a nested class, it is marked with the keyword inner. It has advantages over nested class is that, it can access private members of outer class.

**Question:**  How to write string template expression?

**Answer:** In Kotlin, string can contain template expressions, means we can evaluate the code string. This string expression starts with dollar sign.

fun main(args : Array<String>) {

**var** i = 5

println("i = $i")

}

**Question:**  How to write multiline string in kotlin?

**Answer:** The raw string is multiline string placed inside **triple quotes**. There is no need to escape characters in raw string.

fun main(args : Array<String>) {

val str = """Welcome, Somya!

you are learning

kotlin programming language"""

println(str)

}

**Question:**  How many ways to create array in kotlin?

**Answer:** Kotlin provides two ways to create arrays - **arrayOf()** and **array()** constructor.

fun main(args : Array<String>) {

val x = arrayOf(22,12,43)

**for** (a **in** x) {

println(a)

}

val y = Array(5){0}

**for**(b **in** y) {

println(b)

}

}

**Question:**  Can we create uninitialized array in kotlin?

**Answer:** In kotlin, we can create an uninitialized array or declare it to nullable using **arrayOfNulls()** function. This can be initialized later.

fun main(args : Array<String>) {

**var** x = arrayOfNulls (10)

**for**(a **in** x) {

println(a)

}

}

**Question:**  What is the use of any() function?

**Answer:** The any() function is used to check existence of any element in an array.

fun main(args : Array<String>) {

val arr = arrayOf(22,12,43)

**if**(arr.any())

println(true)

**else**

println(false)

}

**Question:**  Which method is used to get array last index?

**Answer:** Kotlin provides lastIndex() method to get last element index of an array.

**fun** main(args : Array<String>) {

**val** arr = arrayOf(22,12,43)

println(arr.lastIndex)

}

**Question:**  What is !in operator?

**Answer:** The **!in** operator is used to check whether an object not belongs to a collection.

a !**in** b

**Question:**  Define when expression?

**Answer:** In kotlin, the when expression is used if there is need to match the value from many different cases and execute the statements within the matched case. When statement executes line by line and read all the cases. This is replacement of switch statement from other language like PHP, C, C++ and more powerful then switch statement.

**Question:**  What is Range?

**Answer:** Kotlin **range** is a sequence between the given start value and end value. The **(..)** operator is used to create a range expression which is complemented by in and !in.

fun main(args : Array<String>) {

**for**(i **in** 1..10) {

println(i)

}

}

**Question:**  Differenciate between **break** and **continue**?

**Answer:** Kotlin provides the break statement to implement the middle-existing control logic. The break statement causes the immediate exit from the body of the loop.

The continue statement is similar to the break statement. When continue statement is executed, it stops the current iteration of the loop and continue with the next iteration of the loop.

**Question:**  What is the use of rangeTo() function?

**Answer:** The rangeTo() function is itegral type which calls the constructor of Range class. The function is used to return the value from the start to end as mentioned in the range in increasing order.

fun main(args : Array<String>) {

**for** (i **in** 1.rangeTo(10))

println("$i")

}

**Question:**  In kotlin, can we create an empty array?

**Answer:** Yes, In kotlin, we can create an empty array with the help of emptyArray() function.

fun main(args : Array<String>) {

val empty = emptyArray()

}

**Question:**  Define **step()** function?

**Answer:** The **step()** function is used to return the range value in interval of given step value. The step value is required to be always positive, therefore this function never changes the direction of iteration.

fun main(args : Array<String>) {

**for** (x **in** 1..10 step 5)

print("$x")

}

**Question:**  What is the role of matches() method in regular expression?

**Answer:** It is a boolean function that checks all the matched input character sequence in a regular expression.

fun **main**(args : Array<String>) {

val regex = """(abcd|efgh)""".toRegex()

val match = regex.matches("abcd")?.**value**

}

**Question:**  What are the requirements of an infix function?

**Answer:** The following requirements must be fulfill to make a function infix.

* The function must be a member functions or extension functions.
* They must have a single argument.
* The parameter must not have default values or variable number of arguments.

**Question:**  Define high order functions?

**Answer:** In high order function, we can pass a function as a parameter to the other function, means in addition to passing variables as parameters to the function, we can pass the function and the function return type can also be a function.

**Question:**  Give an example of high order functions?

**Answer:** fun main(args : Array<String>) {

val x = "Hello John! "

val y = "How are you? "

val msg:(String,String)->String={x,y->"$x $y"}

highOrderFunc(x,y,msg)

}

fun highOrderFunc(x: String,y: String, msg: (String,String) -> String): Unit {

val message = msg(x,y)

println(message)

}

**Question:**  Define Lambda Function?

**Answer:** A lambda function is a function that does not need to declare, but we can pass immediately as an expression. This is an anonymous function which is usually created during a function call to act as a function parameter.

**Question:**  How to declare an initialization block in kotlin class?

**Answer:** Kotlin provides **init** keyword to declare an initialization block in kotlin class.

**class** **checkInit**() {

init {

println("Our Mission - ")

}

init {

println("About school - ")

}

}

fun main(args : Array<String>) {

checkInit()

}

**Question:**  What is the role of open keyword in Kotlin?

**Answer:** In Kotlin, by default, all the classes are final, means they cannot be inherited from. The open keyword is used to allow inheritance of a class.

**Question:**  What is Data Class?

**Answer:** Kotlin provides facility of data classes. The main purpose of data class is to hold data and it is very simple to create. This can frequently be created without writing a lot of boilerplate code.

**Question:**  List down some kotlin data class built in methods?

**Answer:**

| List down some kotlin data class built in methods? In kotlin, the compiler automatically created the following members with data class.   * equals() * hashcode() * toString() * copy() * componentN() |
| --- |

**Question:**  Define enum?

**Answer:** Enum stands for enumeration, it is a special type of class. This is used to set some predefined constants. The enum keyword is used to create an enumeration class.

**Question:**  What is nested class?

**Answer:** In Kotlin, we can create one class within the body of another class. It is called nested class. A nested class cannot access the data member of an outer class. It is static by default. So we can access its properties and methods without instantiating this.

**Question:**  Differenciate between generics in and out?

**Answer:** The modifier in is used to apply contravariant type parameter, means to assign generic type as input to its function. The modifier out is used to apply covariance type parameter, means to assign a generic type as the output of its function.

**Question:**  Quik Questions About Kotlin

**Answer:**

| Kotlin is a | General Purpose, Programming Language |
| --- | --- |
| Kotlin is inspired by | Java, C#, JavaScript, Scala, and Groovy. |
| Kotlin is developed By | JetBrains |
| Kotlin is first appears on | July 22, 2011 (about 12 years ago) |
| Kotlin file name extensions | .kt, .kts, .ktm |
| Kotlin is licensed under | Apache Licence 2.0 |
| Kotlin is especially used for | Android app development |
| Kotlin top features are | Null safety, Interoperability with Java, Type inference, Concise and expressive syntax |

**Question:**  Key Responsibilities of Kotlin Developer

**Answer:** Key Responsibilities of Kotlin Developer

As a Kotlin developer, some of the key responsibilities you may be expected to have include:

* Developing and maintaining Kotlin-based applications.
* Collaborating with other developers.
* Designing and implementing new features.
* Optimizing the application performance.
* Writing unit and integration tests.
* Documenting code and processes to ensure that other developers can understand and maintain the codebase.
* Staying up-to-date with new technologies.
* contribute to open-source Kotlin projects, which can help you gain exposure and develop your skills further.

**Question:**  Why is Kotlin preferred over Java?

**Answer:** Kotlin eases the coding process as it is simpler than Java and has many features required, that is not provided by Java yet like Extension functions, Null Safety, range expressions etc. In Kotlin, we code approximately 40% less number of code lines as compared with Java.

**Question:**  Explain the data classes in Kotlin?

**Answer:** In programming, we use classes to hold data and these classes are called as data classes. An object can be initialized in the data class and to access the individual parameters of these data classes, we use component functions.

**Question:**  Explain Kotlin’s Null safety?

**Answer:** In Kotlin, the Null safety is used to eliminate the risk of countering the NullPointer exception in real time.

**Question:**  How to Declare a Variable in Kotlin?

**Answer:** In Kotlin, you can declare a variable using var or val which followed by an optional datatype.

Variable declaration in Kotlin looks like:

val s: String = "Hi"

var x = 5

**Question:**  Explain Higher-Order Functions in Kotlin?

**Answer:** Higher-Order Functions: A higher-order function is a function that takes functions as parameters, or returns a function.

**Question:**  Which type of Programming does Kotlin support?

**Answer:** Kotlin supports only two types of programming, and they are:

Procedural programming

Object-oriented programming

**Question:**  Where does the Kotlin run and what is the entry point of Kotlin?

**Answer:** The Kotlin program once compiled, can run on standard JVM like other programming codes.And, like many other programming languages main() function is the entry point of the Kotlin.

**Question:**  What are the different types of constructors in Kotlin?

**Answer:** There are two types of constructors in Kotlin:

Primary constructor: It is a section of the Class header and is declared after the class name.

Secondary constructor: This constructor is declared inside the body.

Note: There can be more secondary constructors for a class.

**Question:**  Can you execute Kotlin code without JVM?

**Answer:** JVM, which stands for Java Virtual Machine is a feature of Kotlin. This feature compiles a Kotlin code into a native code, which can be done without JVM too.

**Question:**  Mention the structural expressions in Kotlin?

**Answer:** There are three Structural expressions in Kotlin. They are:

Return: It returns from the nearest enclosing function or anonymous function by default.

Break: This expression terminates the closest enclosing loop.

Continue: This expression proceeds you to the next closest enclosing loop.

**Question:**  What are the modifiers that are available in Kotlin?

**Answer:** Access modifier in Kotlin provides the developer to customize the declarations as per the requirements. Kotlin provides four modifiers. They are:

Private: This makes the declaration visible only inside the file containing a declaration.

Public: It is by default, which means that the declarations will be visible everywhere.

Internal: This makes the declaration visible everywhere in the same modules.

Protected: This keeps the declaration protected and is not available for top-level declarations.

**Question:**  Can you migrate the code from Java to Kotlin? If yes how do you do it?

**Answer:** Yes, we can migrate the code from Java to Kotlin.This can be done using JetBrains IDEA, which facilitates the conversion of Java code to Kotlin code.

**Question:**  State the differences between Val and Var?

**Answer:** Val: Val, which is the short form of value, is a constant and it cannot be changed once assigned.

Var: Var, which is the short form of variable, is a storage location that accepts the reassignment of values that have the same data types.

**Question:**  List the Basic data types of Kotlin?

**Answer:** Data types of a constant or variable decide what type of variable it is and how much space is required to store it.  
The basic data types in Kotlin are:

* Numbers
* Characters
* Strings
* Arrays
* Booleans

**Question:**  What are the types of strings available in Kotlin? And, what do you mean by Kotlin String Interpolation?

**Answer:** Strings are a collection of characters together.Kotlin features two types of strings, and they are:

* Raw string
* Escaped string

In Kotlin String, templates can be evaluated.This evaluation of string templates is called as the string template interpolation.

**Question:**  State the advantages and disadvantages of Kotlin?

**Answer:** Advantages:

Kotlin is simple and easy to learn as its syntax is similar to that of Java.

It is the functional language that is based on JVM (Java Virtual Machine), which removes the boilerplate codes. Upon all this, Kotlin is considered as an expressive language that is easily readable and understandable and the performance is substantially good.  
It can be used by any desktop, web server or mobile based applications.

Disadvantages:

Kotlin does not provide the static modifier, which causes problems for conventional java developer.

In Kotlin, the function declaration can be done in many places in the application, which creates the trouble for the developer to understand which function is being called.

**Question:**  What is the difference between declaration variable using val or var in Kotlin?

**Answer:** In Kotlin a variable declared using val keyword is cannot be changed. It is similar to the final modifiers in Java whereas the variables declared using var keywords can be reassigned.

**Question:**  How to convert a String to an Int in Kotlin?

**Answer:** toInt() method is used to convert a string value to integer or INT in Kotlin. Below is example uses

fun main(args: Array) {

val s: String = "Kotlin"

var x = 10

x = "8".toInt()

}

**Question:**  Explain Functions In Kotlin?

**Answer:** Kotlin functions are first-class functions that are easily stored in variables and data structures and can be pass as arguments and returned from other higher-order functions.

Sample function declaration and usage in Kotlin

fun double(x: Int): Int {

return 2 \* x

}

val result = double(2

**Question:**  Can you explain the significance of a primary constructor in Kotlin?

**Answer:** A primary constructor is the first thing that is executed when an instance of a class is created. It is used to initialize the class and can contain parameters. In Kotlin, the primary constructor is part of the class header.

**Question:**  How are loops implemented in Kotlin?

**Answer:** Kotlin provides four types of loops for developers to use – for, while, do-while, and forEach. Each loop type has its own specific use cases, so it’s important to choose the right loop for the task at hand. For example, the for loop is typically used for iterating over collections or arrays, while the while loop is used for situations where you don’t know how many times the loop will need to run.

**Question:**  How do you create an object in Kotlin?

**Answer:** To create an object in Kotlin, you use the keyword “object.” This keyword is used to create a singleton object, which is an object that can only have one instance.

**Question:**  Can you give me some examples of how to use control flow statements in Kotlin?

**Answer:** Kotlin offers a few different ways to perform control flow operations, including the use of if/else statements, when expressions, and for loops. If you need to perform a certain action only if a certain condition is met, then you would use an if statement. If you need to perform a different action depending on multiple conditions, then you would use a when expression. And if you need to iterate through a collection of items, then you would use a for loop.

**Question:**  Can you explain what generics are and why they’re important?

**Answer:** Generics are a way of creating reusable code components that can work with different types of data. This is important because it allows you to write code that is more flexible and can be reused in different situations.

**Question:**  What would you say are the main differences between Java and Kotlin?

**Answer:** The main differences between Java and Kotlin are that Kotlin is a statically typed language, meaning that types are checked at compile time, while Java is a dynamically typed language, meaning that types are checked at runtime. Kotlin also has better type inference, meaning that it can often infer the type of a variable from the context, while Java requires the programmer to explicitly declare the type. Kotlin also has a more concise syntax, meaning that it is often easier to write code in Kotlin than in Java.

**Question:**  As a developer, how do you feel about using Kotlin compared to Java?

**Answer:** I love Kotlin! It is so much more concise and easy to read than Java. I find that it makes my code more maintainable and easier to debug.

**Question:**  Is there any support for inheritance in Kotlin? If yes, then can you tell me more about it?

**Answer:** Yes, Kotlin supports inheritance. Classes in Kotlin can inherit from other classes, and they can also override methods from their superclasses. This allows for a great deal of flexibility when it comes to creating new classes and reusing code.

**Question:**  What is your understanding of static members in Kotlin?

**Answer:** Static members are those that are associated with a class, rather than with a specific instance of that class. This means that you can access them without having to create an instance of the class first. In Kotlin, static members can be declared in both the companion object of a class and in top-level functions and properties.

**Question:**  What’s the difference between var and val?

**Answer:** The main difference between var and val is that var is a mutable variable while val is immutable. This means that you can change the value of a var variable after it has been initialized, but you cannot change the value of a val variable.

**Question:**  Can you explain what null safety is in Kotlin and why it’s important?

**Answer:** Null safety is a feature of Kotlin that helps to prevent null pointer exceptions. This is important because it can help to avoid crashes in your Android app. Null safety is achieved by using the special nullable and non-nullable types. When a variable is declared as nullable, it means that it can hold a null value. When a variable is declared as non-nullable, it means that it cannot hold a null value.

**Question:**  In which situations should we consider using data classes instead of regular classes?

**Answer:** Data classes are a great choice when you need to store a large amount of data in a class and you want to avoid a lot of boilerplate code. Data classes also work well with Kotlin’s type system and can help you avoid typecasting errors.

**Question:**  Which class does Kotlin extend from by default?

**Answer:** By default, Kotlin classes extend from the Any class.

**Question:**  Can you explain what higher-order functions are and why they’re so useful?

**Answer:** Higher-order functions are functions that take other functions as arguments and/or return a function as a result. They’re useful because they allow you to abstract away common patterns and avoid code duplication. For example, if you have a list of numbers and you want to filter out the even ones, you could write a higher-order function that takes a predicate (a function that returns a Boolean) and applies it to the list. This higher-order function could then be reused on other lists with different predicates.

**Question:**  What happens when you call a method that throws an exception without handling it?

**Answer:** When you call a method that throws an exception without handling it, the exception will propagate up the call stack until it is either caught or reaches the main function. If the exception is not caught, then it will cause the program to crash.

**Question:**  What is the best way to write multi-line strings in Kotlin?

**Answer:** The best way to write multi-line strings in Kotlin is to use triple quotes (“””) to enclose the string. This allows you to write the string over multiple lines without having to escape each line.

**Question:**  What are typealiases? Why are they helpful?

**Answer:** Typealiases are a way of creating a new name for an existing type. This can be helpful in a number of ways, but one common use case is to make code more readable by giving a more descriptive name to a type that might otherwise be confusing. For example, you could use a typealias to give a name to a complicated generic type that is used throughout your code, making it easier to understand what is going on.

**Question:**  When would I need to use $ inside a string?

**Answer:** You would need to use $ inside a string when you are referencing a variable inside the string. For example, if you had a variable named “name” and you wanted to print out “Hello, name!”, you would use a string like this: “Hello, $name!”.

**Question:**  How do you define a function or property with the same name as its enclosing class in Kotlin?

**Answer:** In Kotlin, you can define a function or property with the same name as its enclosing class by using the keyword “this”. For example:

class MyClass {

fun myFunction() {

// …

}

val myProperty = “some value”

}

In this example, “myFunction” and “myProperty” can be accessed from outside the class using “MyClass.myFunction” and “MyClass.myProperty”.

**Question:**  What are the advantages of the kotlin programming language.

**Answer:** Concise

Interoperable

Safe

**Question:**  What are the Dangling pointers?

**Answer:** The dangling pointers are the pointers in the programming language that do not point to the valid object.A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. These are special cases of memory safety violations.For example A pointer to a null object or a non-static variable after function execution.

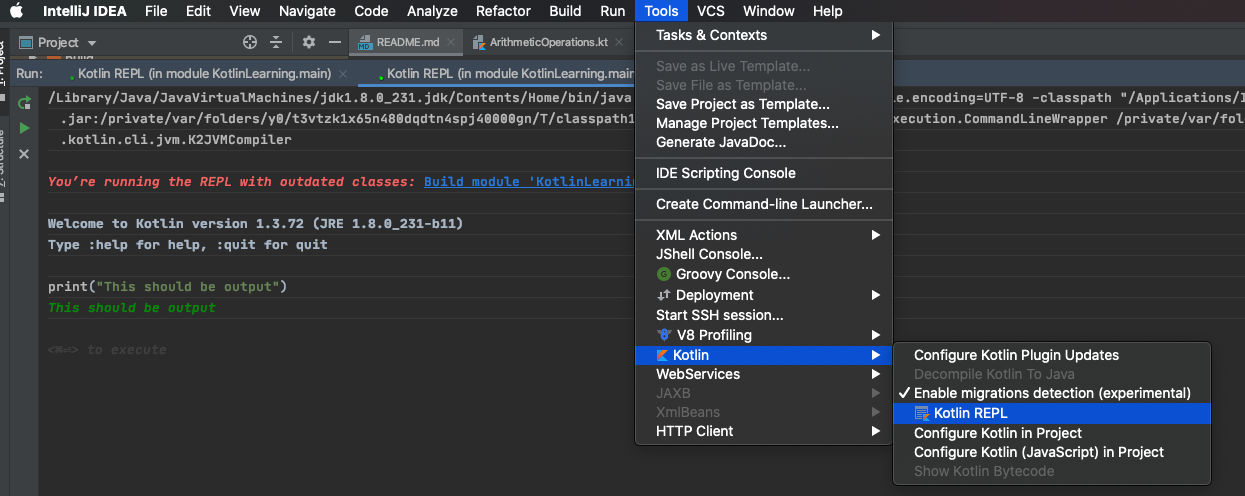
**Question:**  What are the comments types in Kotlin?

**Answer:** Single line: // (command + /) on Mac

1. Multiline: /\*\* \*/ (ctrl + command + /) on Mac

**Question:**  What is kotlin REPL? What’s the use of it?

**Answer:** REPL stands for the Read-eval-Print loop. We can use it to evaluate our expression and code without creating a function. Navigation: IntelliJ IDEA -> Tools -> Kotlin -> Kotlin REPL.



How to work with REPL?

* Command + Enter to execute the code.
* Enter to move the cursor to the next line.

**Question:**  Tell some advantages of Kotlin over Java? or What are the Kotlin functionalities that we can't achieve in java?

**Answer:**

1. Null safety
2. Kotlin coroutines
3. Triple and Pair structures
4. Destructuring Initialization
5. Inline functions
6. Infix functions
7. Scope functions
8. We can write functions at the file level.

**Question:**  Name the two characters that are not directly allowed inside a string?

**Answer:** Backslash: \ This is used to escape characters inside a string.

1. Double quote: " (As it used to construct a string at start and end.)

**Question:**  Key points for variable naming conventions in Kotlin.

**Answer:** Variables can contain letters, number, and underscore( \_ ).

The name must start with a letter or underscore( \_ ).

We can't use special characters in the naming a variable.

The naming convention is camelCase. For example (kotlinVariableName).

**Question:**  How expression evaluation is different from a variable evaluation in string templates?

**Answer:** We can use $ for variable and ${} for expression evaluation.

fun printZooInformation(noOfCats: Int, noOfDogs: Int) {

val catsInfo = "The total $noOfCats cats at zoo"

val dogInfo = "The total $noOfDogs dogs at zoo"

val zooInfo = "The total ${noOfCats+noOfDogs} animal at zoo"

print("$catsInfo\n$dogInfo\n$zooInfo")

}

//Call side

fun main(array: Array<String>) {printZooInformation(3,7)}

//Output

The total 3 cats at zoo

The total dogs at zoo

The total 10 Animal at zoo

**Question:**  What is the difference between ‘Val’ and ‘const Val’ in kotlin?

**Answer:** val is rum time constant and const val are compile-time constant. As you can see in the below example const Val can't evaluate an expression at run time. They will get constant properties at compile time.

val language = "Kotlin"

const val consVal = "I am a $language developer" // Not allowed

val runTimeConstant = "I am a $language developer"// Allowed

**Question:**  How Elvis operator helps us in null safety?

**Answer:** ?: operator is being used by kotlin to handle nullable.

fun function(firstName: String?, lastName: String?) : String = (firstName ?:"").plus(" ").plus(lastName ?:"")

fun main(){

println(function("Maqsood", "Muhammad")) // Maqsood Muhammad

println(function("Maqsood", null)) // Maqsood

}

**Question:**  What are different ways to print Random Integers in Kotlin?

**Answer:** //Print a random Integer with in a Integer range.

println(Random.nextInt())

//Print a random Integer from 0 un till 10(exclusive). println(Random.nextInt(10))

//Print a random Integer from 1(inclusive) un till 10(exclusive). println(Random.nextInt(1,10))

//Print a random Integer from 1(inclusive) un till 10(inclusive). println(Random.nextInt(1..10))

**Question:**  What are the bit sizes for Kotlin basic data types?

**Answer:**

1. Byte 8 bits
2. Short 16 bits
3. Int 32 bits
4. Long 64 bits
5. Float 32 bits
6. Double 64 bits

**Question:**  What is the possibility of error while Long to Int conversion.

**Answer:** That data can be corrupted as we may lose bits while conversion. For example, we have a long length that is bigger then the capacity of Integer then the output will be unexpected.

Val long = 2345454500000

Val toInt = long.toInt()

print(“Converted Integer is $toInt”)

OutPut: Converted Integer is 402356384

**Question:**  How the type of the result of an arithmetic operation is being calculated?

**Answer:** The type of the resulting variable will be the type of the larger operand. Here is the precedence info of the operands:

Byte < Short < Int < Long < Float < Double

fun main(){

val itsLong : Long = 433443

val itsInt : Int = 343

val result = itsLong \* itsInt

println(result.javaClass.kotlin.simpleName) // Long

}

**Question:**  What are the assignment operators in Kotlin?

**Answer:** =, $, ${}, +=, -=, \*=, /=, %/

val sum = a + b

val message = "The sum is $sum"

val message = "The sum is ${a + b}"

val sum += 1

val sum -= 1

val sum \*= 1

val sum /= 1

val sum %= 1

**Question:**  What is the difference between Augmented assignment operator and assignment operator?

**Answer:** =, $, ${} are assignment operators. +=, -=, \*=, /=, %= are augmented assignment operators that are made of assignment and arithmetic operators.

**Question:**  Who is the developer of Kotlin?

**Answer:** Kotlin was developed by JetBrains.

**Question:**  Explain the data classes in Kotlin?

**Answer:** In programming, we use classes to hold data and these classes are called as data classes. An object can be initialized in the data class and to access the individual parameters of these data classes, we use component functions.

**Question:**  What is the basic difference between the fold and reduce in Kotlin? When to use which?

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

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

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

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

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

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

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

**Question:**  Which Type Of Programming Does Kotlin Support?

**Answer:** Kotlin supports only two types of programming, and they are:

Procedural programming

Object-oriented programming

Functional

**Question:**  Why should we use Kotlin?

**Answer:** Kotlin is concise

Kotlin is null-safe

Kotlin is interoperable

**Question:**  What Are The Different Types Of Constructors In Kotlin?

**Answer:** There are two types of constructors in Kotlin:

Primary constructor: It is a section of the Class header and is declared after the class name.

Secondary constructor: This constructor is declared inside the body.

Note: There can be more secondary constructors for a class.

**Question:**  Why Use Kotlin in Mobile App Development?

**Answer:** Since Kotlin simplifies many syntactical elements of Java, it’s easier to write concise, well-documented code. Additionally, since it runs directly on JVM, enterprises hardly need to invest in new tech stacks. So the cost-benefit adjustment is excellent.

Moreover, Kotlin has already started to replace many Java-based Android apps, alongside iOS apps written in Swift. This number will only increase over time and adapting to Kotlin will become a must for modern enterprises. So, to stay ahead of the competition, developers should embrace Kotlin today.

**Question:**  What are Kotlin’s Best Features?

**Answer:** Some of Kotlin’s best features are-

It supports both the object-oriented and functional programming paradigm.

It provides easy-to-use lambda functions that are unavailable in Java.

Maintaining Kotlin is considerably cheap and provides excellent fault tolerance.

Allows developing Node.js applications and JVMs.

Great support for asynchronous communication.

Exceptional compatibility with existing Java codes.

**Question:**  What is Null Safety in Kotlin?

**Answer:** Kotlin comes with in-built protection against unwanted null references which allows it to be more fault-tolerant. It thus allows programs to reduce NullPointerExceptions during runtime and prevents unwanted program crashes. This is a common problem faced by most existing Java software and causes losses costing millions of dollars. This is often coined as Null Safety among Kotlin developers.

**Question:**  Mention The Structural Expressions In Kotlin?

**Answer:** There are three Structural expressions in Kotlin.They are:

Return: It returns from the nearest enclosing function or anonymous function by default.

Break: This expression terminates the closest enclosing loop.

Continue: This expression proceeds you to the next closest enclosing loop.

**Question:**  What Are The Modifiers That Are Available In Kotlin?

**Answer:** The modifier in Kotlin provides the developer to customize the declarations as per the requirements.

Kotlin provides four modifiers.

**They are:**

* **Private:** This makes the declaration visible only inside the file containing the declaration.
* **Public:** It is by default, which means that the declarations will be visible everywhere.
* **Internal:** This makes the declaration visible everywhere in the same modules.
* **Protected:** This keeps the declaration protected and is not available for top-level declarations

**Question:**  What is Elvis operator in Kotlin?

**Answer:** In Kotlin, you can assign null values to a variable by using the null safety property. To check if a value is having null value then you can use if-else or can use the Elvis operator i.e. ?: For example:

var name:String? = "Mindorks"

val nameLength = name?.length ?: -1

println(nameLength)

The Elvis operator(?:) used above will return the length of name if the value is not null otherwise if the value is null, then it will return -1.

**Question:**  How to convert a Kotlin source file to a Java source file?

**Answer:** Steps to convert your Kotlin source file to Java source file:

Open your Kotlin project in the IntelliJ IDEA / Android Studio.

Then navigate to Tools > Kotlin > Show Kotlin Bytecode.

Now click on the Decompile button to get your Java code from the bytecode.

**Question:**  What is the use of @JvmStatic, @JvmOverloads, and @JvmFiled in Kotlin?

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

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

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

**Question:**  Can we use primitive types such as int, double, float in Kotlin?

**Answer:** In Kotlin, we can't use primitive types directly. We can use classes like Int, Double, etc. as an object wrapper for primitives. But the compiled bytecode has these primitive types.

**Question:**  What is the use of abstraction in Kotlin?

**Answer:** Abstraction is the most important concept of Objected Oriented Programming. In Kotlin, the abstraction class is used when you know what functionalities a class should have. But you are not aware of how the functionality is implemented or if the functionality can be implemented using different methods.

**Question:**  List the Basic data types of Kotlin?

**Answer:** Data types of a constant or variable decide what type of variable it is and how much space is required to store it.  
The basic data types in Kotlin are:

* Numbers
* Characters
* Strings
* Arrays
* Booleans

**Question:**  State the advantages and disadvantages of Kotlin?

**Answer:** Advantages:

Kotlin is simple and easy to learn as its syntax is similar to that of Java.

It is the functional language that is based on JVM (Java Virtual Machine), which removes the boilerplate codes. Upon all this, Kotlin is considered as an expressive language that is easily readable and understandable and the performance is substantially good.

It can be used by any desktop, web server or mobile-based applications.

Disadvantages:

Kotlin does not provide the static modifier, which causes problems for conventional java developers.

In Kotlin, the function declaration can be done in many places in the application, which creates trouble for the developer to understand which function is being called.

**Question:**  Explain Functions In Kotlin?

**Answer:** Kotlin functions are first-class functions that are easily stored in variables and data structures and can be pass as arguments and returned from other higher-order functions.

Sample function declaration and usage in Kotlin

fun double(x: Int): Int {

return 2 \* x

}

val result = double(2)

**Question:**  List out some of the extension methods in kotlin?

**Answer:** Some of the extension methods are:

read Text(): Helps to read content in the files to a single string.

buffer Reader(): It is used to read the contents of the file to buffer the reader

read each line(): It reads each line by line in the file

readlines(): It helps to read lines of file for listing

**Question:**  What is an inline function in Kotlin?

**Answer:** 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.

**Question:**  What is noinline in Kotlin?

**Answer:** While using an inline function and want to pass some lambda function and not all lambda function as inline, then you can explicitly tell the compiler which lambda it shouldn't inline.

inline fun doSomethingElse(abc: () -> Unit, noinline xyz: () -> Unit) {

abc()

xyz()

}

**Question:**  Is inheritance compile in Kotlin?

**Answer:** Formal inheritance structure does not compile in the kotlin. By using an open modifier we can finalize classes.

open class B

{

}

class c = B()

{

}

**Question:**  What is mean by init block?

**Answer:** Init is a login block and it is executed in the primary constructor and initialized. If you want to revoke in the secondary constructor then it starts working after the primary constructor in the chain form.

**Question:**  How to Compare Two Strings in Kotlin?

**Answer:** String processing comprises an essential portion of any app development. Interviewees are often asked how to handle this during Kotlin interview questions. You can use the equality operator ‘==’ to do this, as demonstrated by the following example.

val a: String = "This is the first string"

val b: String = "This is the second" + "string"

if (a == b) println("The Strings are Similar")

else println("They don't match!")

**Question:**  Describe For Loops in Kotlin?

**Answer:** Loops are a crucial programming construct that allows us to iterate over things as our program requires. Kotlin features all the commonly used loops such as for, while, and do-while. We’re describing the for loop in a nutshell in the following section.

val sports = listOf("cricket", "football", "basketball")

for (sport in sports) { // for loop

println("Let's play $sport!")

}

The above snippet illustrates the use of the for loop in Kotlin. It’s quite similar to Python and Ruby.

**Question:**  What is the Purpose of Object Keyword?

**Answer:** Kotlin provides an additional keyword called object alongside its standard object-oriented features. Contrary to the traditional object-oriented paradigm where you define a class and create as many of its instances as you require, the object keyword allows you to create a single, lazy instance. The compiler will create this object when you access it in your Kotlin program. The following program provides a simple illustration.

fun calcRent(normalRent: Int, holidayRent: Int): Unit {

val rates = object{

var normal: Int = 30 \* normalRent

var holiday: Int = 30 \* holidayRent

}

val total = rates.normal + rates.holiday

print("Total Rent: $$total")

}

fun main() {

calcRent(10, 2)

}

**Question:**  Explain the Fundamental Data Types of Kotlin?

**Answer:** Kotlin data types define the procedures available on some data. The compiler allocates memory space for variables using their data type. Like many popular programming languages, Kotlin features some often-used data types. Take a look at the below section for a short overview of various Kotlin data types.

integers – contrary to Python, Kotlin has a limited size for integers; available integer types are Long, Int, Short, and Byte

floats – floating-point values contain fractional values; they can be declared using Float or Double

characters – represented by the Char modifier; usually hold a single Unicode character

strings – they are created using the String type and are immutable like in Java

booleans – represents the boolean values true and false

arrays – arrays in Kotlin are represented using the Array class

**Question:**  How do String Interpolations Work in Kotlin?

**Answer:** String interpolations work with multiple placeholders and first evaluate their value to display the final string output. This final output will contain the corresponding values of the placeholders. The below code snippet will illustrate a simple example of Kotlin string interpolation.

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

print("Please enter your name here:")

val name:String? = readLine()

print("Hello, $name!")

}

Here, the Kotlin compiler first receives the user input and interpolates this value in place of the placeholder $name. The last line of the snippet is translated by the compiler as shown below –

new StringBuilder().append("Hello, ").append(name).append("!").toString()

**Question:**  What is the difference between val and var?

**Answer:** var is like a general variable and can be assigned multiple times and is known as the mutable variable in Kotlin. Whereas val is a constant variable which can be Initialized only single time and is known as the immutable variable in Kotlin

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

**Answer:** 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

**Question:**  What is Difference between setValue() and PostValue() in MutableLiveData?

**Answer:** setValue() method must be called from the main thread. But if you need set a value from a background thread, postValue() should be used

**Question:**  How to check if lateinit property is initialized or not?

**Answer:** You can check if the lateinit variable has been initialized or not before using it with the help of isInitialized() method. It returns true if the lateinit property has been initialized otherwise false

**Question:**  When to use lateinit and lazy keywords in kotlin?

**Answer:** Lateinit is used with mutable, while lazy is used with immutablelateinit →lazy → It initializes variable only when it is required for the first time.Lazy :- There are certain classes whose object initialization is very heavy and so much time taking that it results in the delay of the whole class creation process

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

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

**Question:**  Difference between safe calls(?.) and Non-null Assertion(!!)?

**Answer:** Safe Call Operator (?.) is used when you want to make sure that yourapp shouldn’t crash even if variable reference you are holding is null.

var variable: String? = null

variable?.replace(“x”, “z”)

Please note we have not initialized variable above, but it will not throw NullPointerException as Safe call operator is used.But in case of Non-Null Assertion, if you call any method on its reference it will throw KotlinNullPointerException.

variable!!.replace(“x”, “z”)

**Question:**  What are data classes in kotlin?

**Answer:** Only the property name will be used inside the toString(), equals(), hashCode(), and copy() implementations. While two Person objects can have different ages, they will be treated as equal for above example.

**Question:**  Why kotlin classes are final by default ?

**Answer:** Design and document for inheritance or else prohibit it — Core principle from book Effective Java by Joshua Bloch’sIn Simple words -> If classes were open by default and we would forget to mark class as final — (troubles might happen), but when we forget to mark class as open and try to extend it — we will be notified (no trouble)

**Question:**  Difference between == operator and === operator?

**Answer:** The == operator is used to compare the values of variables but === operator is used to check whether references of the variable is equal or not.And in the case of primitive types, the === operator also checks for

the value and not reference.Please note both will result in same in case primitive data types

**Question:**  Access/Visibility Modifiers in Kotlin

**Answer:** Four types of access modifiersprotected: visible inside that particular class or file and also in the subclass of that particular class where it is declared.private: visible inside that particular class or file containing the declaration.internal: visible everywhere in that particular module.public: visible to everyone.Note: By default, the visibility modifier in Kotlin is public

**Question:**  What are extension functions in Kotlin?

**Answer:** Extension Function provides an option to “add” methods to class without inheriting a class. The created extension functions are used as a regular function inside that class. See Example below:Now you can call this same method on any type of view directly on its reference. e.g. textView.hideView()

**Question:**  What are inline functions ?

**Answer:** 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

**Question:**  What are scope functions in Kotlin ?

**Answer:** Scoped functions are functions that execute a block of code within the context of an object. There are five scoped functions in kotlin : let, run, with, also and apply.The scope functions differ by the result they return:apply and also return the context object. So they can be used in chaining function calls on the same object after them. They also can be used in return statements of functionslet, run, and with return the lambda result

**Question:**  What are sealed classes in kotlin?

**Answer:** Sealed classes are similar to enum classes: the set of values for an enum type is also restricted, but each enum constant exists only as a single instance, whereas a subclass of a sealed class can have multiple instances, each with its own state.To declare a sealed class or interface, put the sealed modifier before its name:A sealed class is abstract by itself, it cannot be instantiated directly and can have abstract members

**Question:**  What is significance of annotations : @JvmStatic, @JvmOverloads, and @JvmFiled in Kotlin?

**Answer:** @JvmStatic: This annotation is used to tell the compiler that the method is a static method and can be used in Java code.-> @JvmOverloads: To use the default values passed as an argument in Kotlin code from the Java code.-> @JvmField: To access the fields of a Kotlin class from Java codewithout using any getters and setters

**Question:**  What are infix functions?

**Answer:** An infix function is used to call the function without using any bracket or parenthesis. You need to use the infix keyword to use the infix function

**Question:**  What are advantages to when over switch in kotlin?

**Answer:** It is more concise and powerful than a traditional switch. when can be used either as an expression or as a statement

**Question:**  What are primary and secondary constructors in Kotlin?

**Answer:** The primary constructor cannot contain any code, the initialization code can be placed in a separate initializer block prefixed with the init keyword.Secondary Constructor — Kotlin may have one or more secondary constructors. Secondary constructors allow initialization of variables and allow to provide some logic to the class as well. Theyare prefixed with the constructor keyword

**Question:**  What are Higher Order Functions?

**Answer:** A higher-order function is a function that takes functions as parameters, or returns a function

**Question:**  Is it possible to execute code in Kotlin without the usage of JVM?

**Answer:** Yes, it is possible to execute code in Kotlin without the usage of JVM. The purpose of JVM is to compile a given Kotlin code directly into native code. This process can be easily executed without the usage of JVM.

**Question:**  Why do you think one should choose Kotlin over Java?

**Answer:**

Frequently hiring managers to pose such inquiries to gauge the profundity of the job searchers. Since contrasting two amazing programming languages can never hold up under productive outcomes, you ought to rather talk about their upsides and downsides for exhibiting your aptitude. Java is as yet a superior language that is used to create enterprise-grade customer applications because of its huge list of robust features and ceaseless network support.

Be that as it may, despite being another language, it interoperates pleasantly with Java programming. It permits code developers to make inventive applications without composing many lines of code. Kotlin's solid type interfaces make it a strong decision for creating cutting edge applications. Additionally, since it very well may be likewise utilized for creating iOS applications, engineers don't have to grasp new stacks any longer.

**Question:**  Can you use primitive types for example- int, float, and double in Kotlin?

**Answer:** If I were to say at only the language level, we cannot use the afore-mentioned types. However, on the upside, the JVM’s bytecode which is compiled can surely have them.

**Question:**  What’s the entry point of every Kotlin Program?

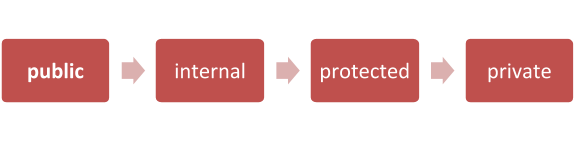
**Answer:** It is the ‘main’ function that is an entry point of every program in Kotlin. In Kotlin we can decide not to compose the main function within the class. At the time of compiling the JVM implicitly encapsulates it within a class. The string functions that are written as ‘Array’ are utilized to recover the order line arguments.

**Question:**  Why is it preferred to utilize Kotlin for a robust Mobile App Development?

**Answer:** As Kotlin improves numerous linguistic components of Java, it's simpler to compose a succinct and all around reported code. Furthermore, since it runs legitimately on JVM, enterprises scarcely need to put resources into new tech stacks. Therefore, the money-saving advantage is phenomenal. Additionally, Kotlin has just begun to supplant numerous Java-based Android applications, and also the iOS applications written in Swift. This scenario will just increment after some time and adjusting to Kotlin will turn into an unquestionable requirement for present-day endeavors. Along these lines, to remain in front of the opposition, mobile app developers should grasp Kotlin today.

**Question:**  What kinds of visibility modifiers can you find in Kotlin?

**Answer:** There are four kinds of visibility modifiers that are available in Kotlin



**Name the visibility modifier that is available to you by default.**

The public modifier is the modifier that is available to us by default.   
**List down some great features of Kotlin**

Here are some great features of Kotlin-

* It is able to support both object's arranged and useful programming worldview.
* It gives simple to utilize lambda works that are inaccessible in Java.
* Maintenance of Kotlin is significantly modest and gives fantastic adaptation to internal failure.
* Allows creating Node.js applications and JVMs.
* It supplants a good support for non-concurring correspondence.
* Exceptional similarity with pre- existing codes of Java.

**Question:**  What do you gather by Null Safety?

**Answer:** Kotlin usually comes along with well in-built insurance against undesirable null references which permits it to be more fault-tolerant. It in this manner permits projects to diminish NullPointerExceptions during runtime and forestalls undesirable program crashes. This is a typical issue looked by most existing Java programming and causes misfortunes costing a huge number of dollars. This is regularly authored as Null Safety among Kotlin engineers

**Question:**  What is the use of abstraction in Kotlin?

**Answer:** Abstraction is the most significant idea of Objected Oriented Programming. In Kotlin, reflection class is utilized when you recognize what functionalities a class ought to have. In any case, you don't know about how the user is executed or if the user can be actualized utilizing various techniques.

**Question:**  Elucidate on the functionality of Null safety along with its types in Kotlin?

**Answer:** Kotlin puts a great deal of weight behind invalid wellbeing which is a way to deal with forestalling the feared Null Pointer Exceptions by utilizing nullable types which resemble String?, Int?, Float? and so forth. These go about as a wrapper type and can hold invalid qualities. A nullable worth can't be added to another nullable or fundamental type of value. To be able to recover the fundamental kinds we have to utilize safe calls that open up the Nullable Types. On the off chance that on opening up, the worth is invalid we can decide to disregard or utilize a default an incentive

**Question:**  Illustrate about an Elvis Operator in Kotlin.

**Answer:** An Elvis Operator is typically used to securely unpack the type value received from the Nullable.

We can represent it as ‘?:’ on the nullable type.

The value that is towards the right-hand side is going to be used in case the nullable type value is null.

var str: String? = “JournalDev.com”

var newStr = str?: “Default Value”

str = null

newStr = str?: “Default Value”

**Question:**  Name the different sorts of constructors in Kotlin. Differentiate between them.

**Answer:** There are two types of constructors in Kotlin, they are Primary constructors and Secondary constructors.

Primary — these are characterized in the class headers. They can't hold any inputs of logic. It is possible to have just a single primary constructor for every class.

Secondary — they're characterized in the class's body. They should delegate to the primary constructor if it exists. These constructors can hold logic very easily. There can be more than one secondary constructor.

**Question:**  How will you execute an init block in Kotlin?

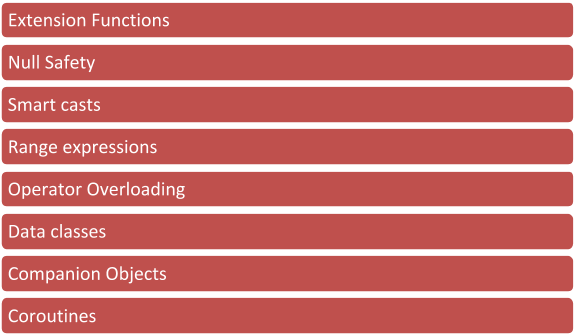
**Answer:** An ‘init’ is the primary class initializer block that a developer can use in Kotlin. It’s implemented only after a primary constructor has been successfully instantiated. In case you raise a secondary constructor, in that case, it functions only after the primary one just in the order of its composition in the development chain.

**Question:**  Can you call Java functions in Kotlin?

**Answer:** This is possible as Kotlin permits the software engineer to call the existing procedures of Java from inside a Kotlin program. Functions, for example, the getter and setter couple are spoken to as properties. Kotlin doles out Unit to each void worth that originates from a Java work. Along these lines, the Unit in Kotlin is void in Java code. You should get away from some keywords of Kotlin Java, however. Since keywords like is, in, and object are legitimate identifiers in Java, you'll have to get away from them in Java libraries utilizing the backtick (') character. Furthermore, Kotlin executes highlights like stage types and explanations to guarantee invalid security when calling outside Java capacities.

**Question:**  List out some features that are available in Kotlin but not in Java?

**Answer:** Here are some features of Kotlin that are unique to it and are not available in Java.



**Question:**  What is the way to declare a variable in Kotlin?

**Answer:** We can declare a variable in Kotlin by using val and var. here is an example of the same-

Here is an example of a standard declaration of a variable in Kotlin

Val x: String = "Welcome"

var a = 9

**Question:**  What are some extensions that are available to java.io.file in the Kotlin environment?

**Answer:** Here is a list of some useful extensions that are available to java.io.file in the Kotlin environment

File.bufferedReader() : this extension facilitates the reading of contents of a given file in the BufferedReader

File.forEachLine() : this extension facilitates the reading of a given file in a line by line manner in the Kotlin environment

File.inputStream() : this extension helps in reading of the given contents of a file to the InputStream

File.readBytes() : this extension facilitates the reading of the contents of a file to the ByteArray

File.readLines() : this extension facilitates to read the lines of a file to the List

File.readText() : this extension helps you in the reading of contents of a given file to a single String

**Question:**  What is the interoperability of Kotlin with JavaScript?

**Answer:** During numerous Kotlin inquiries questions, interviewees are gotten some information about the level of adaptability Kotlin takes into account JavaScript applications. Kotlin empowers Android and iOS devs to flawlessly target JavaScript. This means you can without much of a stretch trans-compile a Kotlin program to local JavaScript code. This permits code designers to effectively make programming for the famous Node.JS stage.

Kotlin empowers developers to control all components of JS programming, for example, controlling the DOM, utilizing illustrations, dealing with the server-side, etc. Furthermore, you can use Kotlin with existing JS libraries like jQuery and React.

**Question:**  What do you mean by the Higher-Order Functions?

**Answer:** In Kotlin a higher-order function is a class function whose job is to take the functions as parameters or to return the value as a function.

**Question:**  Why is there no feature of Macros in Kotlin?

**Answer:** Macros are helpful in various programming situations. Nonetheless, they will, in general, make a wreck of your extent and regularly confound new engineers. This is the reason JetBrains, the engineers behind Kotlin, precluded this component by and large. Besides, engineers frequently think that it's difficult to test or upgrade codebases that comprise a ton of macros. Along these lines, excluding macros is a plan choice. The code developers who work on Kotlin are, in any case, taking a shot at highlights like serialization and compiler modules to address a few deficiencies of this choice.

**Question:**  Highlight some advantages of Kotlin.

**Answer:** Some advantages of Kotlin are-

Kotlin is straightforward and simple to learn as its language structure is like that of Java.

It is the practical language that depends on JVM (Java Virtual Machine), which expels the standard codes. Upon this, Kotlin is considered as an expressive language that is effectively clear and justifiable and the exhibition is generously acceptable.

It tends to be utilized by any work area, web server or portable based applications.

**Question:**  Highlight some disadvantages of Kotlin.

**Answer:** Some of the disadvantages of Kotlin are-

Kotlin doesn't give the static modifier, which messes up ordinary java engineer.

In Kotlin, the capacity presentation should be possible in numerous spots in the application, which makes it inconvenient for the designer to comprehend which capacity is being called.

**Question:**  What is the concept of Multiplatform?

**Answer:** Kotlin developers keep on turning out new and energizing highlights for engineers. The multiplatform highlight is one such exploratory component that empowers developers to share code between a few stages, for example, JavaScript, iOS, and work area applications.

This is getting progressively well known among current designers since it lessens the measure of code by an impressive factor. You can utilize a significant part of the equivalent codebase for composing applications for various stages because of this element. Make a common module for your applications and rundown the conditions. Presently, you can make separate modules for various stages and coordinate the central functionalities utilizing the normal module.

**Question:**  Explain Nothing, Unit and Any?

**Answer:** In Kotlin, any is an information type that speaks to essential sorts like a whole number, buoys, and strings. Any sort can't hold any invalid qualities as a matter of course and executes programmed throwing of lower types. This is like the Java object java.Lang.Object.

The unit type is a sort that is returned by work calls that don't bring anything back. Kotlin doesn't offer void capacities like C or Java do and uses unit for this reason. You can consider the unit only one explicit instance.

The nothing type is returned by Kotlin capacities when they can't arrive at the base of the capacity. It generally occurs because of vast recursions or loops.

**Question:**  What are the structural expressions of Kotlin?

**Answer:** In Kotlin there are three different types of structural expressions and they are as follows-

return – this type of structural expression pauses the program that is under execution and at the same time returns the value from the enclosed function.

break – this type of structural expression is generally used to terminate the nearest loop that is enclosing in Kotlin's environment.

continue – this type of structural expression allows the execution of the Kotlin code to go on to the next iteration of the loop leaving the execution of the current code under operation.

**Question:**  Explain the destructuring declarations.

**Answer:** Destructuring Declaration is a brilliant method to relegate different qualities to factors from information put away in objects/strings/arrays.

Inside paratheses, we've set the variable presentations. In the engine, destructuring assertions create component works for every one of the class factors.

**Question:**  What’s the difference between lazy and lateinit?

**Answer:** Both are utilized to postpone the property instatements in Kotlin

lateinit is a modifier utilized with var and is utilized to set the incentive to the var at a later point.

apathetic is a technique or rather a state lambda articulation. It's determined to a val as it were. The val would be made at runtime when it's needed.

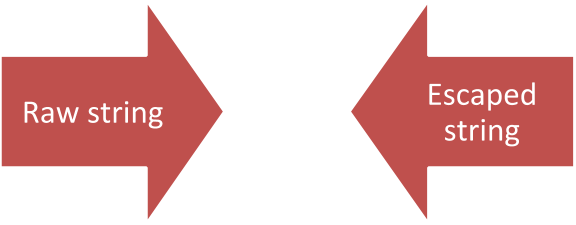
Here is the command

val x: Int by lazy { 10 }

lateinit var y: String

**Question:**  State the number string type you can get in Kotlin.

**Answer:** We have an option of two types of strings in Kotlin and they are-



**Question:**  What do you understand by interpolation of a string?

**Answer:** The process of creating a template in the string is call interpolation.

**Question:**  What is a Sealed Class in Kotlin?

**Answer:** Sealed classes are another incredibly valuable element of this cutting edge programming language. They can be utilized to limit the legacy progressive system of a class. You should simply characterize a class as fixed, and no one will have the option to make subclasses that have a place with this fixed class.

These classes will prove to be useful when you use them inside when articulations. If you can impeccably cover every conceivable case, you won't need to utilize the else statement. In any case, recall that fixed classes are dynamic without anyone else and you cannot start up one straightforwardly.

**Question:**  What is the process of creating a Volatile Variable?

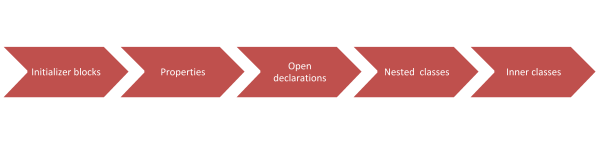
**Answer:** Volatile variable in Kotlin is a convincing element that empowers software engineers to control strings, and CPU time all the more successfully. Pronouncing a variable as unstable is very simple and varies somewhat from Java.

@Volatile var name:String = "something"

Unpredictable isn't a keyword in Kotlin, rather than Java. Or maybe it is a comment and makes every datum compose obvious to every other string right away.

**Question:**  What do you understand by the class members in Kotlin?

**Answer:** The following members are available in a class in Kotlin-



**Question:**  Differentiate between Val and Var?

**Answer:** **Val**: Val, is the acronym used for the term value. This value is a constant and this cannot be changed once it is assigned.

**Var**: Var, is the acronym used for the term variable. A variable is a storage location that receives the relocation of values with the same kind of data types.

**Question:**  Why do we need the Object Keyword?

**Answer:** Kotlin gives an extra keyword called object near its standard item. In opposition to the customary article situated worldview where you characterize a class and make the same number of its cases as you are going to need, the object keyword permits you to make a solitary, slow execution. The compiler will make this object when you get to it in your Kotlin program.

**Question:**  What is the way towards managing a null operator?

**Answer:** The best way of managing a nullable value is by utilizing an Elvis operator.

**Question:**  How do you think extension functions are useful?

**Answer:** Extension functions Augmentation capacities assist with expanding a class with new usefulness without acquiring from the class. Likewise, you may utilize them like an inbuilt capacity for the class all through the application.

**Question:**  Is there any kind of additional functionalities that are available for standard Java packages and the standard Java classes?

**Answer:** Of course, Yes. Kotlin uses the concept of extension functions, that we already talked about, to build some useful and more widely used functions among developers directly into the Kotlin library.

**Question:**  How can you contrast two strings?

**Answer:** One can easily do that in two strings in two ways,

Using”==” operator”

sing compare() function

**Question:**  What are Coroutines?

**Answer:** The Kotlin coroutine is a light-weight string that doesn't require any setting turning on the processor and won't map on local strings. This is the motivation behind why they are quick in preparing the solicitations.

**Question:**  Define the Scope functions in Kotlin.

**Answer:** In Kotlin standard library, multiple functions are available that help carries out the code block within the object's context. Here, the temporary scope is developed when we utilize the lambda expressions to call such functions to an object known as Scope functions. Moreover, Scope functions make the code more engaging, clear, and brief, which represents the core features of Kotlin programming.

Scope functions in Kotlin language are of multiple types, such as:-

Let

Run

Apply

Also

With

Let - In the let function, the context object is "it", and the value that returns is "lambda result". Generally, we use this function for null safety calls.

Run - In the run function, the context object is "this", and the value that returns is "lambda result". It combines the features of both let and with functions in Kotlin and is useful when the lambda object includes initialization and computation of the value that returns. Further, this function helps perform computations along with null safety calls.

Apply - In the apply function, the context object is "this", and it returns the context object as a result. It is helpful to drive the receiver object members mainly to initialize them.

Also - The context object for the function also is "it", and the return value becomes the "context object". This function helps execute some additional activities after initializing the object members.

With - The context object for the function "with" is "it". It returns the "lambda result" as a return value. This function is proper when calling the functions without offering the lambda result.

**Question:**  What are the primary data types used in Kotlin language?

**Answer:** The data type for a variable or a constant plays a crucial role in Kotlin language, determining which variable type is proper and how much space it needs for storage. The following are the different data types that a string allows :-

Numbers

Strings

Arrays

Characteristics

Booleans

**Question:**  What do you mean by the Suspend function in Kotlin?

**Answer:** The suspend function in Kotlin is used to stop, start, and resume at any time. This function can perform long-running activities and remains on standby without blocking until it finishes. Moreover, the suspend function is called on only from other suspend functions or from a coroutine and not from a regular function. A suspending function can call any of the standard functions, but it needs another suspending function to suspend or eliminate the execution. Further, the suspend function is expressed as delay().

**Question:**  What is meant by Open Keyword in Kotlin?

**Answer:** In Kotlin, the Open Keyword refers to "open for extension". All functions and classes in Kotlin are final by default, and we cannot override or inherit them. Therefore, we use an empty keyword to obtain this before the class and functions.

**Question:**  Define Ranges operator in Kotlin?

**Answer:** In Kotlin language, the range operator supports iterating in the range. The range operator is available in the form of (..). The following is an example of it-

for (i in 1..15)

print(i)

The above example defines that it will give the outcome to print data from 1 to 15 numbers.

**Question:**  Define the Public keyword in Kotlin?

**Answer:** In Kotlin, the Public keyword is the visibility or default modifier that enables declarations to be visible and accessible everywhere. It is used when the visibility modifier is not stated.

**Question:**  What is meant by "When" in Kotlin?

**Answer:** When it is used in Kotlin as an expression or statement, it is often used instead of a switch. It is because it has a much better design and is more succinct and powerful to use than the legacy switch.

**Question:**  Define the use of @JvmStatic in Kotlin?

**Answer:** In Kotlin, @JvmStatic is an annotation proper to build static methods for functions and to inform the compiler that it is a static method. We can use it within the Java code.

**Question:**  Explain the use of @JvmFiled in Kotlin.

**Answer:** In Kotlin, @JvmFiled annotation helps to achieve compatibility between the two different languages. Further, it helps to access the Kotlin class fields from the Java code without needing any setters and getters.

**Question:**  What is meant by @JvmOverlaods?

**Answer:** The @JvmOverloads annotation is a proper function when we pass the default values similar to arguments from the Java code to the Kotlin code.

**Question:**  What do you mean by Data classes in Kotlin?

**Answer:** Data classes help store some data in Kotlin and don't offer other features. The advantage of using Data class in Kotlin is that Kotlin provides a large amount of self-produced code, and it holds it without needing any boilerplate code in Java.

**Question:**  Explain the Double Bang operator in Kotlin.operator in Kotlin?

**Answer:** The double bang (!!) operator in Kotlin is applicable to transform any value into a non-null type. It also throws an exception called KotlinNullPointer exception when the value becomes null.

**Question:**  Define the Singleton class in Kotlin?

**Answer:** A Singleton class in Kotlin defines only a single instance that can be developed and used anywhere. Users can easily access the singleton object from their code through this class.

**Question:**  Can we use the word

**Answer:** We cannot use it as a keyword in Kotlin, but we can locate it within the Java language.

**Question:**  Define the use of Lambda Expression in Kotlin?

**Answer:** It is an expression without a name, and these expressions are frequently used with some higher-level functions, like a map.

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

**Answer:** Java:

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

Kotlin

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

**Question:**  What is the basic difference between fold and reduce in Kotlin?

**Answer:** fold: This function accumulates value with the initial value. After that apply the operation from left to right to current accumulator value and each element.

fun main() {

var value= listOf(1, 2, 3).fold(2) { sum, element -> sum \* element }

println(value)

}

Answer: 12

reduce accumulates value from the first element and apply operations from left to right to current accumulator value and each element.

fun main() {

var value= listOf(1, 2, 3,4,5,6).reduce { sum, element -> sum + element }

println(value)

}

Answer: 21

/\* Process

sum= 1, element = 2

sum= 3, element = 3

sum= 6, element = 4

sum= 10, element = 5

sum= 15, element = 6

21

\*/

**Question:**  What is the difference between var and val in Kotlin?

**Answer:** Var

var can be assigned multiple times that's why it is known as the mutable variable in Kotlin.

var v1 = 10

v1 = 30 //works fine

Val

val is a constant or final variable that can not be assigned multiple times. It can be Initialized only once that's why it is known as the immutable variable in Kotlin. It means once the value is assigned to val variable. It can’t be changed later.

val v2 = 10

v2 = 30 //it will show error, "Val cannot be reassigned"

**Question:**  What is a data class in Kotlin?

**Answer:** A data class is a class that only contains state. It does not perform any operation.

The data class provides self-generated code(1. equals(), 2. hashCode(), 3. toString(), 4. copy(), 5. componentN() ).

It avoids getters and setters boilerplate code.

Kotlin Data Class Requirements

To make a class as data, we have to follow the following principles:

1. The primary constructor must have at least one parameter and the parameters are either marked val or var.

2. The class cannot be marked as abstract, open, sealed or inner.

3. The data class can extend (inherit) other class and it can also implement other interfaces.

**Question:**  How to create a singleton class in Kotlin?

**Answer:** We create a Singleton class with the help object keyword only in Kotlin. It can be defined without the use of a class.

An object class contains properties, functions, and the init method.

object Singleton

{

init {

println("Singleton class ")

}

var name = "www.developerlibs.com"

fun printName()

{

println(name)

}

}

**Question:**  Explain lateinit and lazy?

**Answer:** Lateinit

The modifier lateinit allows the compiler to delay the initialization of a variable. The compiler recognizes that the value of the non-null property is not stored in the constructor stage to compile normally. It can be applied to the var properties only because it can’t be compiled to a final field. We have to be careful to assign our lateinit var properties before we use it. Otherwise, it will crash the app on a null value.

public class Devloper {

lateinit var libs: Libs

fun create() {

libs = Libs()

}

fun Libs() {

libs.do()

}

}

Lazy

A lazy property is initialized when it is used for the first time. The value of lazy property computes only one thread, and all threads can use the same value. The lazy property gets initializations when you first-time access the lazy property. For the second time, this value is remembered and returned. It can only be used with the val properties:

val myName by lazy {

println("developerlisb")

"www.developerlibs.com"

}

fun main(args: Array<String>) {

println(myName) //it'll print developerlibs and www.developerlibs.com

println(myName) // only stored value www.developerlibs.com

}

Ans:

developerlisb

www.developerlibs.com

www.developerlibs.com

**Question:**  Explain the Null safety in Kotlin?

**Answer:** The type system of Kotlin distinguishes the references that can hold null (nullable references) and those that can not (non-null references). The compiler gives information about the correctness of a program at compile time. For example, a regular String type variable can not hold null:

var name: String = "developerlibs"

name = null // compilation error

we can declare a variable as a nullable string:

var name: String? = "developerlibs"

name = null // ok

print(name)

Null Safety can eliminate the risk of occurrence of NPE in real-time because Kotlin can detect NPE exception errors at compile time itself and guard against them. Read more about Null safety

**Question:**  Can we declare the properties of the class inside of the secondary constructor?

**Answer:** No, we can not declare the property of the class inside the secondary constructor. As you see below code snippet that'll give an error if you compile it. Because, we are declaring a property id of the class in the secondary constructor, which is not allowed in Kotlin:

class Student (var name: String) {

init() {

println("Student has got a name as $name")

}

constructor(sectionName: String, var id: Int) this(sectionName) {

}

}

You can use property inside the secondary constructor if declared the property inside the class and use it in the secondary constructor:

class Student (var name: String) {

var id: Int = -1

init() {

println("Student has got a name as $name")

}

constructor(secname: String, id: Int) this(secname) {

this.id = id

}

}

**Question:**  Is Kotlin better than Java?

**Answer:** Often employers ask such questions to measure the depth of job seekers. Since comparing two languages can never bear fruitful results, you should instead discuss their individual pros and cons for demonstrating your expertise. Java is still a better language for building enterprise-grade consumer applications due to its massive feature list and unending community support. However, despite being a new language, Kotlin interoperates nicely with Java. It allows developers to create innovative applications without writing hundreds of lines of code. Kotlin's strong type inferences make it a solid choice for developing next-generation apps. Moreover, since it can be also used for developing iOS apps, developers don't need to embrace new stacks anymore.

**Question:**  Why Use Kotlin in Mobile App Development?

**Answer:** Since Kotlin simplifies many syntactical elements of Java, it's easier to write concise, well-documented code. Additionally, since it runs directly on JVM, enterprises hardly need to invest in new tech stacks. So the cost-benefit adjustment is excellent. Moreover, Kotlin has already started to replace many Java-based Android apps, alongside iOS apps written in Swift. This number will only increase over time and adapting to Kotlin will become a must for modern enterprises. So, to stay ahead of the competition, developers should embrace Kotlin today.

**Question:**  What are Kotlin's Best Features?

**Answer:** Some of Kotlin's best features are- It supports both object-oriented and functional programming paradigm. It provides easy to use lambda functions that are unavailable in Java. Maintaining Kotlin is considerably cheap and provides excellent fault-tolerance. Allows developing Node.js applications and JVMs. Great support for asynchronous communication. Exceptional compatibility with existing Java codes.

**Question:**  What is Null Safety in Kotlin?

**Answer:** Kotlin comes with in-built protection against unwanted null references which allows it to be more fault-tolerant. It thus allows programs to reduce NullPointerExceptions during runtime and prevents unwanted program crashes. This is a common problem faced by most existing Java software and causes losses costing millions of dollars. This is often coined as Null Safety among Kotlin developers.

**Question:**  Differentiate between Kotlin's Variable Declaration Methods

**Answer:** Job seekers are often posed with this issue in their Kotlin interview questions. Kotlin allows you to declare variables in two different ways. It exposes the val and var keyword for this purpose. However, as you will see, they're quite different in their working. var number: Int = 10 number = 15 The first line declares an integer variable called number and assigns 10 as its value. The second line then replaces this 10 with a new value of 15. You'll need to declare variables this way if you want to change them later down the program. val number: Int = 10 number = 15 The above code block is, however, invalid and will result in compilation errors. Kotlin doesn't allow users to change the value of variables that were created using the val keyword. You should use it for declaring values that remain the same throughout your code.

**Question:**  What's the Problem with Below Code? val name = "UBUNTU" val upperCase = name.toUpperCase() name.inc()

**Answer:** The above code snippet will fail to compile due to type exception. Kotlin is statically typed and implements strong type inferences. Since the value of the name variable is a String, Kotlin assumes that name is also a type of String. Thus the second line is perfectly valid because the method toUpperCase() is a String method. The last line, however, tries to perform an increment operation. This line causes Kotlin to throw a compilation error since inc() can only work with Int values, not Strings.

**Question:**  What is Nullability in Kotlin?

**Answer:** Many programming languages like C and Java allow variable declarations without setting initial values. These variables usually hold a value of type null. If you invoke methods on such variables later in your program, it will crash in runtime. Kotlin doesn't allow programmers to declare variables this way and reduces null-based crashes significantly.

val name: String = null

Thus, the above line won't compile in Kotlin. You'll need to define variables as nullable if you want it to contain null values.

val name: String? = null

This time, name can contain either String or null.

**Question:**  Does Kotlin Allow Calling Java Functions?

**Answer:** Yes, Kotlin allows the programmer to call existing Java procedures from within a Kotlin program. Functions such as the getter and setter duo are represented as properties. Kotlin assigns Unit to each void value that comes from a Java function. Thus, the Unit in Kotlin is simply void in Java code.

You will need to escape some Kotlin keywords in Java though. Since keywords like is, in, and object are valid identifiers in Java, you'll need to escape them in Java libraries using the backtick (`) character. Additionally, Kotlin implements features like platform types and annotations to ensure null safety when calling external Java functions.

**Question:**  Describe Kotlin's Interoperability with JavaScript

**Answer:** During many Kotlin interview questions, interviewees are asked about the degree of flexibility Kotlin allows for JavaScript applications. Kotlin enables Android and iOS devs to seamlessly target JavaScript. What this means is, you can easily trans-compile a program written in Kotlin to native JavaScript code. This allows developers to easily create software for the popular Node.JS platform.

Kotlin enables developers to control all elements of JS programming- such as manipulating the DOM, leveraging graphics, managing the server-side, and so on. Additionally, you can utilize Kotlin with existing JS libraries like jQuery, and React. Visit this guide for detailed information about Kotlin to JavaScript trans-piling.

**Question:**  Why Doesn't Kotlin Feature Explicit Ternary Conditionals?

**Answer:** Kotlin doesn't offer any specific ternary operator of the form c = (a < b) ? a : b; like Java or C. It omits this option because you can do the same with the if expression in Kotlin. Since the above ternary operator is essentially an expression of the form (condition ? then : else), Kotlin simply allows you to do this using its standard if keyword.

val c = if (a < b) a else b

This line of code does the same thing in Kotlin as the ternary operator does in Java. You can also pack blocks inside if-else branches.

**Question:**  What is the Function of the Elvis Operator?

**Answer:** The Elvis operator in Kotlin allows app developers to handle null-based exceptions. It is a compelling feature of Kotlin which enables programmers to reduce runtime crashes by a considerable margin. Although you can still handle your nulls yourself, the Elvis operator makes it relatively straightforward.

val z = x ?: return y

In this line, z will only contain the value of x if it is not null. Otherwise, the entire expression will halt executing and return y. It works since the return statement is also an expression. So, Elvis operator's look like a ?: b in Kotlin.

**Question:**  Explain the Workings of when in Kotlin

**Answer:** During many Kotlin interview questions, job seekers face questions on conditional statements. Apart from the traditional if-else, Kotlin features another conditional construct called when. You can think of it as a replacement for the switch construct available in C and other popular programming languages. However, in Kotlin, when is an expression; while the switch is a statement in C.

val number = true val final = when(number) { true -> println("It is indeed true!") false -> println("Oops! false") }

We demonstrated a simple example using boolean expressions. You'll find this handy when working with exceptionally large conditional branches.

**Question:**  What is Smart Casting in Kotlin?

**Answer:** Smart cast is a simple but useful mechanism that allows programmers to reduce most null-based errors. The Kotlin compiler does this by inferring the variables. We've witnessed it in a previous question. Below, we're illustrating a simple example of smart casting in Kotlin.

fun test(a: Any) { if (a is String) { print(a.length) // a is cast to String by the compiler automatically } }

**Question:**  What are Co-Routines in Kotlin?

**Answer:** Kotlin aims at increasing app performance via leveraging asynchronous execution. Contrary to traditional execution flows, asynchronous executions don't get blocked on I/O. It makes Kotlin ideal for building large-scale IT infrastructures. Take a look at the below example to understand co-routines more clearly.

import kotlinx.coroutines.\* fun main() { GlobalScope.launch { // creates a new coroutine and continues delay(2000L) // non-blocking delay for 2000 milliseconds or 2 sec. println("Hello") } println("World!") // execution continues even while coroutine waits Thread.sleep(4000L) // block main thread for 4 seconds }

This program will display the string "World!" before displaying "Hello". The program first creates a new coroutine within the GlobalScope and wait for 2 seconds. Meanwhile, the main thread will continue and print "World!". It will wait for 4 seconds then and after two seconds, the coroutine will print "Hello".

**Question:**  List Some Features of Kotlin that are Absent in Java

**Answer:** Sometimes Kotlin interview questions are designed in a way that helps companies understand the potential of future employees. Below, we're listing some functionalities of Kotlin that are simply unavailable in the Java programming language.

Null Safety - a flagship feature of Kotlin

Co-Routines - enables asynchronous programming

Operator Overloading - a key feature missing in Java

Smart Casts - allows casting inferences

Companion Object - another useful functionality

**Question:**  What Extension Methods does Kotlin provide to java.io.File?

**Answer:** Java uses the java.io.File class for providing abstract representations of file or directory paths. Kotlin offers the below extension methods to this file -

bufferedReader() - allows to read the contents of a file and put them into a buffer

readBytes() - can be used for reading the contents of a file into a ByteArray

readText() - allows reading file contents and puts them to a String

forEachLine() - reads a file line by line

readLines() - line by line reads a file and puts them into a List

**Question:**  How to Migrate Java Code to Kotlin?

**Answer:** It's possible for us to migrate existing Java codes to Kotlin easily using the IntelliJ IDEA from JetBrains. The below section demonstrates how to do this in a sequence.

Update the build file to support Kotlin compilation

Open the necessary .java file using IDEA

Copy all the required code snippets

Create a Kotlin file ending with .kt

Paste the code snippets in this Kotlin file

Enter Yes when IDEA asks if it should convert the Java code to Kotlin

**Question:**  Why doesn't Kotlin Feature Macros?

**Answer:** Macros are useful in a number of programming scenarios. However, they tend to create a mess of your project and often confuse new developers. This is why JetBrains, the developers behind Kotlin omitted this feature altogether. Moreover, developers often find it hard to test or optimize codebases that contain a lot of macros. So, omitting macros is a design decision. Ther developers of Kotlin are, however, working on features like serialization and compiler plugins to address some shortcomings of this decision.

**Question:**  Explain the Different Constructors in Kotlin

**Answer:** Kotlin offers two different constructors for initializing class attributes. It varies from Java in this regard since the latter only provides a single constructor. These two constructors are known as primary constructors and secondary constructors in Kotlin. During many Kotlin interview questions, job seekers are asked to point out the differences between these two.

Primary Constructor - resides in the class declaration header

Secondary Constructor - declared inside Kotlin class body and may have multiple instances.

**Question:**  Is It Possible to Execute Kotlin Code without JVM?

**Answer:** As we've mentioned many times already, Kotlin compiles into bytecode and runs on top of the Java Virtual Machine(JVM). However, it's also possible to compile Kotlin into native machine code and thus execute successfully without requiring any JVM at all.

Developers can use the Kotlin/Native tool for doing this effortlessly. It's an effective LLVM backend that allows us to create standalone executables. It exposes some additional functionalities as well. Consult their official documentation for more information.

**Question:**  How do Ranges Work in Kotlin?

**Answer:** Ranges allow our programs to seamlessly iterate over a list or progression. It's one of the many iterators available in Kotlin and enhances the readability of your program. The below code snippets demonstrate some basic functions of Kotlin ranges.

for (i in 1..5) { print(i) // prints 12345 as output } val x = 6 for (i in 1..10){ if( i!=x ) continue print(i) // prints only 6 }

**Question:**  Explain the Structural Expressions of Kotlin

**Answer:** Kotlin has three different structural expressions - namely return, break, and continue. We're discussing each one of them with short notes.

return - this expression halts the program execution and returns from the enclosing function

break - it is used for terminating the nearest enclosing loop in Kotlin

continue - it allows the execution to proceed to the next iteration without performing the current operation

The second snippet of the previous example prints the value 6 since we've used continue. If we had used break instead, it would print nothing.

**Question:**  How to Compare Two Strings in Kotlin?

**Answer:** String processing comprises an essential portion of any app development. Interviewees are often asked how to handle this during Kotlin interview questions. You can use the equality operator '==' to do this, as demonstrated by the following example.

val a: String = "This is the first string" val b: String = "This is the second" + "string" if (a == b) println("The Strings are Similar") else println("They don't match!")

Additionally, you can use the compareTo() function for comparing strings in Kotlin.

**Question:**  Describe For Loops in Kotlin

**Answer:** Loops are a crucial programming construct that allows us to iterate over things as our program requires. Kotlin features all the commonly used loops such as for, while, and do-while. We're describing the for loop in a nutshell in the following section.

val sports = listOf("cricket", "football", "basketball") for (sport in sports) { // for loop println("Let's play $sport!") }

The above snippet illustrates the use of the for loop in Kotlin. It's quite similar to Python and Ruby.

**Question:**  Describe While and Do-While Loops

**Answer:** The while and do-while loops work quite similar but have a specific distinction. The do-while loop executes at least once, even if the first conditional expression is returned as false. Contrary to this, while loops will stop execution if the loop isn't true at a given time.

var i = 1 while (i < 5) { // while loop println(i) i++ }

This code will print the numbers 1 to 4 and then terminate. Now, take a look at the below do-while snippet.

var i = 6 do{ // do-while println(i) i++ }while(i<5)

Although the condition to while is false, it will print the number 6 as its output. This happens since the execution of the do block takes place without checking the condition first.

**Question:**  What are the Kotlin Data Classes?

**Answer:** Kotlin offers a convenient way of storing values by means of the data class. A data class comes with many useful in-built methods such as for copying, string representations, instance collections, and so on. Data classes are defined using the data modifier. Some auto-generated methods available to a newly created data class are - toString, equals, copy, hashCode, and componentN functions. The first method returns a string representation, equals check for equality among properties, and copy allows cloning.

**Question:**  What are Kotlin Sealed Classes?

**Answer:** Sealed classes are another extremely useful feature of this modern-day programming language. They can be used to restrict the inheritance hierarchy of a class. All you need to do is define a class as sealed, and nobody will be able to create subclasses that belong to this sealed class.

These classes will come in handy when you use them inside when expressions. If you can perfectly cover all possible cases, you won't have to use the else clause. However, remember that sealed classes are abstract by themselves and you can not instantiate one directly.

**Question:**  How to Create Volatile Variables?

**Answer:** Volatile variables are a compelling feature that enables programmers to control threads, and CPU time more effectively. Declaring a variable as volatile is quite easy and differs slightly than from Java.

@Volatile var name:String = "something"

Actually, volatile is not a keyword in Kotlin, as opposed to Java. Rather it is an annotation and makes each data write visible to all other threads immediately.

**Question:**  What is the Purpose of Object Keyword?

**Answer:** Kotlin provides an additional keyword called object alongside its standard object-oriented features. Contrary to the traditional object-oriented paradigm where you define a class and create as many of its instances as you require, the object keyword allows you to create a single, lazy instance. The compiler will create this object when you access it in your Kotlin program. The following program provides a simple illustration.

fun calcRent(normalRent: Int, holidayRent: Int): Unit {

val rates = object{

var normal: Int = 30 \* normalRent

var holiday: Int = 30 \* holidayRent }

val total = rates.normal + rates.holiday

print("Total Rent: $$total") }

fun main() { calcRent(10, 2) }

**Question:**  Explain the Class Modifiers in Kotlin

**Answer:** During most Kotlin interview questions, employers like to test job seekers on their grasp of classes and other object-oriented programming constructs. Class modifiers allow developers to customize their class declarations as they need it. We're demonstrating the four modifiers Kotlin exposes to programmers for this purpose.

private - the class declaration is only visible inside the file that contains it

public - these class declarations are visible everywhere, it's the default in Kotlin

protected - makes the class unavailable for top-level Kotlin declarations

internal - makes the declaration available for all the same modules

**Question:**  Explain the Fundamental Data Types of Kotlin

**Answer:** Kotlin data types define the procedures available on some data. The compiler allocates memory space for variables using their data type. Like many popular programming languages, Kotlin features some often-used data types. Take a look at the below section for a short overview of various Kotlin data types.

integers - contrary to Python, Kotlin has limited size for integers; available integer types are Long, Int, Short, and Byte

floats - floating-point values contain fractional values; they can be declared using Float or Double

characters - represented by the Char modifier; usually hold a single Unicode character

strings - they are created using the String type and are immutable like in Java

booleans - represents the boolean values true and false

arrays - arrays in Kotlin are represented using the Array class

**Question:**  How do String Interpolations Work in Kotlin?

**Answer:** String interpolations work with multiple placeholders and first evaluate their value to display the final string output. This final output will contain the corresponding values of the placeholders. The below code snippet will illustrate a simple example of Kotlin string interpolation.

fun main(args: Array<String>) { // String Interpolation print("Please enter your name here:") val name:String? = readLine() print("Hello, $name!") }

Here, the Kotlin compiler first receives the user input and interpolates this value in place of the placeholder $name. The last line of the snippet is translated by the compiler as shown below -

new StringBuilder().append("Hello, ").append(name).append("!").toString()

**Question:**  How to Convert Strings to Integer?

**Answer:** It's common for app developers to convert string to int for various reasons. Kotlin allows you to go about this in multiple ways. Below we're demonstrating a simple Kotlin program for this purpose.

fun main(args: Array<String>) { for (string in args) { try { val parsedValue = string.toInt() println("The parsed integer value is $parsedInt") } catch (nfe: NumberFormatException) { // not a valid int } } }

You can also do this without using explicit try-catch blocks. For this, you'll need to utilize the toIntOrNull() method.

**Question:**  What's the Difference Between const and val?

**Answer:** Earlier we illustrated how to create variables that only contain fixed data using the val keyword. However, Kotlin offers the const keyword for creating constants like the C programming language. The key difference between val and const is their execution phase. Kotlin sets the properties of val at runtime by default. On the other hand, const is set by the compiler during the program's compiler time.

const val str = "Fixed string!" // global variable fun main(args: Array<String>) { const val x = 4 const val str = "New string.." // local variable }

Additionally, you can not use const inside the local scope, thus the above code block will fail to compile. This modifier is also not applicable on var.

**Question:**  What's the Entry Point for Kotlin Programs?

**Answer:** Kotlin, like many popular programming languages, relies on a specific entry point. The main() function is this point, similar to other OOP languages such as C++ and Java. Developers can easily define the command-line arguments taken by Kotlin programs. For this, you'll need to pass args: Array<String> to this main() function.

It takes a somewhat different syntax than traditional Java programs. Below we're illustrating the differences between the main() function in both Java and Kotlin. You can easily compare them for a better understanding.

public static void main(String[] args) // Entry to Java Programs fun main(args: Array<String>) // Entry to Kotlin Programs

**Question:**  Write a Kotlin Program to Display the Fibonacci Series

**Answer:** Most Kotlin interview questions aim to find out the knowledge of the candidates for practical problems. The Fibonacci series is a common question faced by job seekers in many Kotlin interviews. It's a mathematical sequence of numbers where each number is the sum of its previous two numbers.

fun main(args: Array<String>) { val range = 10 var firstNumber = 0 var secondNumber = 1 print("First $range numbers of Fibonacci series: ") for (i in 1..range) { print("$firstNumber + ") val sum = firstNumber + secondNumber firstNumber = secondNumber secondNumber = sum } }

We've used a for loop for computing this series. However, you can solve this problem by using several strategies.

**Question:**  Write a Program for Determining if a Number is Prime or Not

**Answer:** Prime numbers play a major role in modern computing, particularly in number theory. Software developers usually use them to implement safe encryption mechanisms for their applications. We're illustrating a simple Kotlin program that determines whether a particular number is prime or not.

fun main( args: Array<String> ) { print("Enter the number:") var num = readLine()!!.toIntOrNull() var flag = false if( num != null ){ for (i in 2..num / 2) { if (num % i == 0) { flag = true break } } } if (flag) println("$num is not a prime number.") else println("$num is a prime number.") }

**Question:**  Write a Program for Finding the Sum of Natural Numbers

**Answer:** Natural numbers are all positive values starting from 1. The sum of these numbers can be easily calculated using the Kotlin's loop constructs. Below, we're demonstrating a simple program that takes user input and calculates the sum of all natural numbers up to that point.

fun main( args: Array<String> ) { print("Enter the number:") var num = readLine()!!.toIntOrNull() var sum = 0 // inital value of summation if( num != null ){ for (i in 1..num) { sum += i } println("Sum = $sum") } }

**Question:**  Explain the Differences Between ? and !! in Terms of Null Safety

**Answer:** Kotlin provides two different mechanisms for unwrapping the contents of a nullable type. The Elvis operator '?' provides a safe call and doesn't crash your program if the content is of type null. However, on the other hand, !! is used for force unwrapping the contents of a nullable variable. This is performed during runtime and thus may lead to a potential system crash if the value returned is null. So, you should only use the !! modifier when you're certain about the value of your variables.

**Question:**  Find Factorial of Numbers using Recursion

**Answer:** The factorial of a number is defined as the product of all numbers starting at 1 and up to that number. We can easily write a Kotlin program to do this job using either loops or recursion. The latter is a divide and conquer programming strategy which divides a routine into multiple similar but small sub-routines.

fun main(args: Array<String>) { print("Enter the number:") val number = readLine()?.toInt() if(number != null ){ val factorial = multiplyNums(number) println("Factorial of $number = $factorial") } } fun multiplyNums(number: Int): Long { if (number >= 1) return number \* multiplyNums(number - 1) // recursive call to multiplyNums else return 1 }

**Question:**  What is Kotlin Multiplatform?

**Answer:** Kotlin developers continue to roll out new and exciting features for developers. The multiplatform feature is one such experimental feature that enables programmers to share code between several platforms such as JavaScript, iOS, and desktop apps.

This is becoming increasingly popular among modern developers because it reduces the amount of code by a considerable factor. You can use much of the same codebase for writing apps for different platforms thanks to this feature. Simply create a shared module for your apps and list the dependencies. Now, you can create separate modules for different platforms and integrate the core functionalities using the common module.

**Question:**  How do Lambda Functions Work in Kotlin?

**Answer:** A lambda function is a small, self-contained block of code that can be passed around your program for greater flexibility. They're usually written inline and solve basic but frequent programming tasks. We take a closer look at some simple Kotlin lambda functions to understand it in more detail.

fun main(args: Array<String>) { val greet = { println("Hello!")} // first lambda function greet() val product = { x: Int, y: Int -> x \* y } // second lambda function val result = product(3, 5) println("Product of two numbers: $result") }

The first lambda greets the user with the text "Hello" while the second one returns the product of two numbers. Lambda functions are anonymous, meaning they don't have any names.

**Question:**  Explain Why the Following Code Fails to Compile

class A{ } class B : A(){ }

**Answer:** Classes in Kotlin are final by default. So, you can not inherit the attributes of the first class A from the second class B. You will need to declare the first class as open for solving this issue. The below snippet will illustrate this for you.

open class A{ } class B : A(){ }

Now, this code will compile just fine and execute as expected. Kotlin exposes this open modifier to allow flexible yet secure class inheritances.

**Question:**  How do Destructuring Declarations Work in Kotlin?

**Answer:** Kotlin allows developers to assign multiple values to variables from data stored in objects or arrays. It's a very smart feature and employers often ask about this during Kotlin interview questions. We're demonstrating a quick example below to help you understand this concept more clearly.

data class Book(val name: String, val publication: String){ } fun main(args: Array<String>) { val (name, publication) = Book("Kotlin for Dummies", "O'Reilly") println(name) println(publication) }

When compiled, this program will return the name and publication of the book from the specified data class. The destructuring takes place in the first line inside the main function.

**Question:**  Write a Program to Swap Two Numbers Without Using Temporary Variables

**Answer:** Swapping two numbers using a temporary variable is a common practice for many. However, we can easily swap values among variables without using any such variables. Take a look at the below Kotlin program to find out how to do this in action.

fun main(a: Array<String>) { var variable1 = 10 var variable2 = 20 println("Before swapping:") println("First variable has: $variable1") println("Second variable has: $variable2") variable1 = variable1 + variable2 variable2 = variable1 - variable2 variable1 = variable1 - variable2 println("After swapping:") println("First variable contains: $variable1") println("Second variable contains: $variable2") }

**Question:**  What is Any, Unit, and Nothing?

**Answer:** In Kotlin, any is a data type that represents basic types like integer, floats, and strings. Any type can not hold any null values by default and implements automatic casting of lower types. This is similar to the Java object java.Lang.Object.

The unit type is a type that is returned by function calls that don't return anything. Kotlin doesn't offer void functions like C or Java do and utilizes unit for this purpose. You can think of unit as nothing but one specific instance.

The nothing type is returned by Kotlin functions when they can't reach the bottom of the function. It usually happens due to infinite loops or recursions.

**Question:**  Write a Kotlin Program for Calculating the Power of a Number

**Answer:** Many programming logic uses the power of a number for achieving its end goal. Thankfully, calculating the power of a given number is effortless in Kotlin. The below program demonstrates a simple program for this purpose. It is based on recursion.

fun main(args: Array<String>) { print("Enter the base:") val base = readLine()!!.toInt() print("Enter the power:") val power = readLine()!!.toInt() val result = powerRaised(base, power) println("$base^$power = $result") } fun powerRaised(base: Int, power: Int): Int { if (power != 0) return base \* powerRaised(base, power - 1) else return 1 }

**Question:**  How do You Create Static Methods in Kotlin?

**Answer:** Static methods are useful for a number of reasons. They allow programmers to prevent the copying of methods and allow working with them without creating an object first. Kotlin doesn't feature the widely used static keyword found in Java. Rather, you'll need to create a companion object. Below, we're comparing the creation of static methods in both Java and Kotlin. Hopefully, they will help you understand them better.

class A { public static int returnMe() { return 5; } // Java } class A { companion object { fun a() : Int = 5 // Kotlin } }

**Question:**  How to Create Arrays of Different Types in Kotlin

**Answer:** Kotlin allows programmers to easily create different types of arrays using the arrayOf() method. Below, we'll take a look at how to create arrays containing integer, floats, and strings using this simple but robust procedure.

val arr1 = arrayOf(1, 2, 3) val arr2 = arrayOf(1.2, 2.3, 3.4) val arr3 = arrayOf("Hello", "String", "Array)

**Question:**  Key Advantages of Kotlin

**Answer:**

1. It is secure and it has a compact code
2. Able to interoperate with [Java](https://entri.app/blog/career-path-for-java-developer/)
3. Debugging is easy with Kotlin
4. A young Programming language that is fast and safe
5. Relatively easier to maintain
6. Effective optimization of Codes
7. HIghly helpful to boost the productivity of the developer
8. Relatively fewer and clear codes
9. Kotler supports a multi-platform development

**Question:**  Scope of Kotlin Developers

**Answer:** Many companies prefer Kotlin developers for app development. If you know Java language it is an added advantage because Kotlin’s interoperate with Java and the libraries of both will give enormous resources to the developer. Kotlin certainly has a huge growth predicted and it is being widely accepted as a future android application development programming language. Kotlin has another big advantage in that it can be used as a replacement of Javascript as a web language for front-end development.

**Question:**  Define Kotlin Language?

**Answer:** Kotlin is a general-purpose, cross-platform language used for application developments. Especially Android Mobile applications.

**Question:**  List out reasons to use Kotlin

**Answer:** Web Development and Android development is the main use of Kotlin. It interoperates with Java to make programming easier.

**Question:**  Kotlin was developed by?

**Answer:** It is developed by Jetbrains

**Question:**  Is it possible to switch to Kotlin from Java?

**Answer:** Yes. Java codes are automatically converted to Kotlin when you paste them into the editor.

**Question:**  List out the key benefits of Kotlin

**Answer:** It is secure. Easy to debug. Interoperate with Java to code android apps. It is easier to maintain.

**Question:**  What are the main skills needed for a Kotlin Developer?

**Answer:** He must be familiar with Database. Need knowledge in web frameworks.Must know Java.Experience in the field of software development and application development.

**Question:**  Define the use of extension functions in Kotlin?

**Answer:** It helps to add “class” to methods without using any design pattern.

**Question:**  Is Kotlin interoperable with Java?

**Answer:** Yes. If you know java codes it can be used for Kotlin as it is automatically converting.

**Question:**  How to define a variable in Kotlin?

**Answer:** Kotlin uses two different keywords to define variables, ie val, and var.

**Question:**  Is it possible that Kotlin can be used for Web Development?

**Answer:** Yes. Kotlin is used as a tool for web development as it is recommended as a replacement for Javascript.

**Question:**  What is the key advantage of Kotlin when interoperable with Java?

**Answer:** The use of both libraries give vast resources to the developer

**Question:**  Can Kotlin be used as a replacement for Javascript?

**Answer:** Yes. In web development, Kotlin is used as a replacement for Javascript.

**Question:**  Is Kotlin is limited to Android app development?

**Answer:** No. Kotlin is used for software development and web development too.

**Question:**  Is it possible to migrate Java code to Kotlin?

**Answer:** Yes. When you copy-paste Java code in the editor, it will automatically convert to Kotlin code.

**Question:**  List out the various data types available in Kotlin?

**Answer:** Four main data types are Integer, Floating Point, Boolean, and Character

**Question:**  What is ‘null-safety’ in Kotlin?

**Answer:** A variable cannot behold null. If we try to assign null the compiler will give an error.

**Question:**  List out the advantages of Kotlin over Java

**Answer:** There is only a need of writing a few codes in Kotlin. Kotlin is easy to code than Java. Kotlin holds a strong community with a huge support system.

**Question:**  List out some disadvantages of Kotlin

**Answer:** The main disadvantage is the compilation speed is low. Low human resources in the Kotlin field.

**Question:**  Explain the term data class concerning Kotlin?

**Answer:** Data classes in Kotlin means a class that holds data and contains standard functionalities.

**Question:**  How many constructors are there in Kotlin?

**Answer:** There are only two. Primary and Secondary

**Question:**  What are the programming types supported by Kotlin?

**Answer:** Kotlin supports both object-oriented and functional programming.

**Question:**  Explain the use of abstraction in Kotlin?

**Answer:** If we don’t know how the Kotlin functionalities are implemented abstraction helps in that process.

**Question:**  How can we compare two strings in Kotlin?

**Answer:** Equality operator “==” is used to comparing.

**Question:**  What is val and var in Kotlin?

**Answer:** Both used to declare variables in Kotlin. Var is called a mutable variable and val is an immutable one.

**Question:**  Explain the type of modifiers available in Kotlin?

**Answer:** The flour types available are Private, Protected, Internal, and Public.

**Question:**  Why you should shift from Java to kotlin?

**Answer:** Kotlin is very simple compared to java, it minimises redundancies in code and offers many useful features that java doesn’t support.

**Question:**  What are the features that kotlin support and java doesn’t?

**Answer:** The following are the important features that kotlin supports over java:

Operator Overloading

Coroutines

Null Safety

Range expressions

Companion Objects

Smart casts

**Question:**  How can data class be used in Kotlin?

**Answer:** Data class supports basic data types, it doesn’t have any other functionality

**Question:**  How do you export code from Java to Kotlin?

**Answer:** JetBrains software provides an inbuilt tool called jetbrain idea, it helps in exporting the code from java to kotlin.

**Question:**  How do you declare a variable in kotlin?

**Answer:** Declaration of a variable in kotlin differs from java, below is the format for

that:

val s: String = “Hi”

var x = 5

**Question:**  Does Kotlin programming allow macros?

**Answer:** Kotlin developers are facing difficulties in including macros in the language, so offering support for macros in kotlin is not possible.

**Question:**  How do you declare ranges operator in kotlin?

**Answer:** Ranges operator is used to iterate through a range, the operator is of form

(..), for example

for (i in 1..15)

print(i)

The above format will print the output as a range of numbers from 1 to 15.

13. Describe the use of extension function?

Extension function is useful in extending a class without having the need

to inherit from the class.

**Question:**  Name the extension methods Kotlin provides to java.io.File?

**Answer:** Below listed are the extension methods that kotlin provides to java.io.file:

readText(): used for reading the content of a file.

forEachLine(): used for reading a file line by line in Kotlin

readLines(): Used for reading lines in file to List

bufferedReader(): Used for reading the contents of a file into

BufferedReader

readBytes(): Used for reading contents of file to ByteArray

**Question:**  What is the difference between Const and Val?

**Answer:** Const is primarily for compile time initializations and val is used for runtime initializations.

**Question:**  List some of the key modifiers in kotlin?

**Answer:** Kotlin has a list of few key modifiers, they are:

1. public

2. private

3. protected

4. interval

**Question:**  How do you differentiate var and val in Kotin?

**Answer:** A variable defined with var are mutable, it is formatted as

var a: Int=3

a \*= a

A Variable defined with Val are immutable, it is formatted as:

val b: Int = 6

b\*=b

**Question:**  Is kotlin strongly typed?

**Answer:** Kotlin is similar to java with equally strongly typed. Kotlin compiles down to pure Java byte code.

**Question:**  What is the purpose of creating kotlin?

**Answer:** Kotlin was created by a Czech company named after JetBrains and the actual purpose of creating it was to solve their internal development problems

**Question:**  Which companies use kotlin?

**Answer:** Kotlin is used by major companies around the globe, here are some of

them:

Postmates

Atlassian

Evernote

Corda

Spring by pivotal

Coursera

Pinterest

Uber.

**Question:**  What is the latest version of kotlin?

**Answer:** Currently kotlin is updated to version 1.3.30. This version has a bug fix and tooling update for 1.3. This release focuses mainly on KAPT performance, kotlin Native and improvements for IntelliJ IDEA.

**Question:**  How is kotlin integrated with Gradle?

**Answer:** With the release of Gradle 5.0, kotlin offers support for gradle. You can have a look at it here: Gradle Kotlin DSL v1.0

**Question:**  Name some of the features that have Kotlin but not in present in Java?

**Answer:** kotlin has many features that are not present in Java. The following are a

few features:

Null safety

Operator overloading

Extension functions

Data classes etc

**Question:**  Why you should switch to Kotlin from Java?

**Answer:** Kotlin is the easiest language compared to Java as they reduce redundant code in an application. Kotlin offers new features that Java doesn’t support.

**Question:**  what is the use of extension functions in Kotlin?

**Answer:** Extension functions are used to extend the class without inheriting from the class

**Question:**  What is the default visibility modifier and also list the different types of modifiers?

**Answer:** The default visibility modifier is public. The following are a few types of

modifiers:

internal

public

private

protected

**Question:**  What is Kotlin’s target platform? Is Java-kotlin interoperability possible?

**Answer:** The target platform for Kotlin is JVM. Java is 100% interoperable with Kotlin because on compilation both produce the same bytecode. Therefore Java code can be called from Kotlin and the reverse is also possible.

**Question:**  How to declare a Kotlin variable?

**Answer:** In the Kotlin the variable declaration:

Val s: String = “Hello world!!”

var p = 15

**Question:**  What is the difference between Val and var declaration?

**Answer:** Val is similar to final modifiers in Java they can’t be changed. Var can be

reassigned but the data type cannot be changed. An example is shown

below:

Val y: string = “hi”

Var x = 2

31. How to convert a String to an Integer?

toInt() is the method used to convert string to an integer. The following is

an example:

fun main(args: Array<String>) {

Val s: String = “Hi”

var x = 5

x = “8”.toInt()

}

**Question:**  Is it possible to migrate the code from Java to Kotlin?

**Answer:** Yes, it is possible to migrate the code from Java to Kotlin. JetBrains IDE is required to perform this action as it facilitates the conversion.

**Question:**  Does Kotlin support primitive Data types?

**Answer:** No, Kotlin doesn’t support primitive Data types as in Java

**Question:**  Mention the structural expressions in Kotlin?

**Answer:** The following are the three structural expressions in Kotlin. Return: It returns value from the functions by default. Break: terminates the loop condition. Continue: precedes you to the next enclosing loop.

**Question:**  What are the data classes in Kotlin? How are they defined?

**Answer:** In kotlin, we create a class to store huge data, called data class. To create a

data class you need to add a keyword called “data” on the class. The

following is an example:

data class Book(var name: String, var authorName: String)

fun main(args: Array<String>)

{

Val book = Book(“One night at the call center”, “Chetan Bhagat”)

}

**Question:**  what operator is used to handle null exceptions in Kotlin?

**Answer:** Elvis Operator is used for handling null expectations in Kotlin.

**Question:**  What are High-Level Functions in Kotlin?

**Answer:** Functions that accept another function are called high order functions.

Kotlin regards its functions as citizens of first class and passes the

variables to other function.

**Question:**  What are Coroutines?

**Answer:** Coroutine is a light-weight thread that doesn’t require any context

switching on the processor and will not map on native threads. This is the

reason why they are fast in processing the requests.

**Question:**  How to create static methods in Kotlin?

**Answer:** There is no static method in Kotlin but, to create a static method we use a

keyword called object. The following is an example to create a static

method:

class kotlin {

companion object {

fun x() : Int = 45

}

}

**Question:**  What are the advantages and disadvantages of Kotlin?

**Answer:** 