Kotlin Refresher: <https://play.kotlinlang.org/byExample>

Interview QA

1. <https://www.journaldev.com/20567/kotlin-interview-questions>
2. <https://www.interviewbit.com/kotlin-interview-questions/>
3. <https://www.tutorialkart.com/kotlin/kotlin-interview-questions/>

**What is kotlin:** Kotlin is an open-source, statically-typed programming language that supports both object-oriented and functional programming and it is 100% interoperate with Java. We can Write better Android apps & faster with Kotlin as compare to java because there is lot of new class & feature are introduce like: Object, Data & Sealed class, Lamda & Higher order function, Coroutine for currency class, we can write the code in very less line.

Statically typed languages: each variable and expression is already known at compile time.

**immutable & mutable:**

An **immutable variable is one whose value cannot be changed**, also known as unchangeable or read-only variable.

On the other hand the value of the **mutable variable** can be changed.

**Synchronous & Asynchronous:**

**Synchronous =** happens at the same time. Means multiple task execution at same time.

**Asynchronous** = doesn't happen at the same time. Means Only one task execute at time.

**Referential Equality(===):** Referential equality, we use the === symbol which allows us to evaluate the reference of an object (if it’s pointing to the same object). This is an equivalent of “==” operator in Java.

val a = Integer(10)

val b = Integer(10)

val c = a

a === b (False)

c === a (True)

and we check them both by doing a === b, which will return false because they’re two separate objects, each pointing to a different location in memory.

**Structural Equality(==)**

Now for structural equality, we use the == symbol that evaluates if both object/variable values are the same (or equal). This is usually achieved by implementing equals() method in Java.

So, using the same Integers example, we just need to do a == b, and in this case, it will return true, since both variables have the same value.

**Note**: In Kotlin, == operator only compares the data or variables, whereas in Java or other languages == operator generally used to compare the references.

**.equals() method (x.equal(y))**

the String equals() method compares the two given strings based on the data/content of the string. If all the contents of both the strings are the same, it returns true. If all characters are not matched, then it returns false.

**Variable in Kotlin(var/val/lateinit/const)**

**var: var is mutable type data type.** var variable value can change frequently or any number of time. it’s just like Mutable and once initialise the object value can change any point of time.

var myName: String = “Abhishek”

myName = “Chandan”

**val:** val can’t be change and it’s just like Immutable. when we declare val variable, it’s mandatory to initialise it’s value and that value can’t be change any point of time.

Note: val keyword must be used to declare for run time values

val myName: String = “Abhishek”

**lateinit:** lateinit means late initialization, Means if want you initialize a variable value in the constructor instead or latter but make sure that variable will initialize before using it

Note: It will not allocate memory until it’s initialisation

lateinit var myName: String

onCcreate(){

myName = “Abhishek”

}

init{

myName = “Abhishek”

}

**const:** const variable value can’t be change once declare and it’s just like Immutable, final and read-only,

Note: const keyword must be used to declare compile time values. Must be at the top level or a member of an object or a companion object.

class MyUtill{

companion object GetUtility{

const val myName: string = “Abhishek”

}

}

**Lazy:** Your variable will not be initialized unless you use that variable in your code. It will initialized only once after that we always use the same value. the concept of ‘lazy initialization’ was designed to prevent unnecessary initialization of objects.

val test: String by lazy {

"some value"

}

**Null Safety:** Kotlin's type system is aimed at eliminating/remove the danger of null references or result in a null reference exception or NullPointerException. In Kotlin, the type system distinguishes between references that can hold null (nullable references) and those that cannot (non-null references).

N**ull safety :** Kotlin null safety is a procedure to eliminate the risk of null reference from the code. Kotlin compiler throws NullPointerException immediately if it found any null argument is passed without executing any other statements. Kotlin's type system is aimed to eliminate NullPointerException form the code.

**Nullable**:That means You have the ability to declare whether a variable can hold a null value or not. By supporting nullability in the type system, the compiler can detect possible NullPointerException errors at compile time and reduce the possibility of having them thrown at runtime.

1. **a regular variable of type String cannot hold null:**
   1. **var a: String = "abc"** // Regular initialization means non-null by default
   2. **a = null** // compilation error (Null can not be a value of a non-null type String)
   3. Now, if you call a method or access a property on a, it's guaranteed not to cause an NPE, so you can safely say:
2. **To allow nulls, you can declare a variable as a nullable string by writing String?:**
   1. var b: String? = "abc" // can be set to null
   2. b = null // ok
   3. **val** l = b.length // error: variable 'b' can be null (Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?)
   4. **val l = b?.length or val l = if (b != null) b.length else -1 //Will run success**
   5. if you want to access the same property on b, that would not be safe, and the compiler reports an error:

**SafeCall | ?// null sefty =>**

Your second option for accessing a property on a nullable variable is using the safe call operator ?.

val b: String? = null

**val l = b?.length) //Safe Call**

**Elvis operator | ? =>**

When you have a nullable reference, b, you can say "if b is not null, use it, otherwise use some non-null value":

val b: String? = null

**val l: Int = if (b != null) b.length else -1** // Safe Call with ElVis Operator

**non-null assertion operator (!!) =>**

The third option is for NPE-lovers: the not-null assertion operator (!!) converts any value to a non-null type and throws an exception if the value is null. You can write b!!, and this will return a non-null value of b (for example, a String in our example) or throw an NPE if b is null

**val l = b!!.length**

**//** if value then then ok otherwise throw NPE in case of null value **| java.lang.NullPointerException**

**//How to check Given Value is Integer or Not**

val dynamicTypeValue = ""

val p = dynamicTypeValue.toIntOrNull() // convert to int

println("is String $p")

if(p != null){

println("Given Number is Integer")

}else{

println("Given Number is String")

}

**OR**

**fun checkValueType(value: Any){**

if(value is Int){

}else if(value is String){

}else if(value is Bool){

}

else if(value is Long){

}

**}**

**Singleton / Object Class:**  Singleton is design pattern that restrict to create multiple object for that class. Object class is a spacial class in kotlin that has only one object. when we Declare or create a class with object keyword then kotlin compiler create static reference and private constructor for that class & customs constructor are not allowed and if you want /need to initialise something in that class, then we have do it in init block of that class.

**object SingleTonClass{**

var myName: string = “”

init{

myName = “Abhishek”

}

}

**Static/Companion object:** companion object is object whose property and function are tied/associated to a class but not to the instance of that class. Means, if you want a function or any member of the class that can be access / called without having instance of that class. Only one companion object is allowed per class.

Note: it’s just like a java static class and if you want to access it from Java code then we have need to annotate this object with @JvmStatic

**class MyObject{**

**companion object GetUtility{**

val myName: string = “Abhishek”

}

}

val myObj = MyObject. GetUtility

val myObj = MyObject. GetUtility. myName or myObj. myName

**Data Class:** It’s just like Java Model class and whose main purpose is to hold data. Data classes cannot be abstract, open, sealed, or inner.

Note: The compiler automatically derives the following members equals(), hashCode(), or toString()

data class User(val name: String, val withDefault: Int = 0 )

Optional :

init{

}

**Sealed Class:** A sealed class is an abstract class with a restricted class hierarchy. This provides more control over the inheritance. They are restricted but also allow freedom in state representation. All direct subclasses of a sealed class are known at compile time. Sealed classes are used for representing restricted class hierarchies wherein the object or the value can have value only among one of the types, thus fixing your type hierarchies. Sealed classes are commonly used in cases, where you know what a given value to be only among a given set of options.

Sealed class represents set of options wherein value will be among these options only. Sealed class in kotlin is just like enum class. But in Enum class, set of constant can have only one type or only single instance whereas subclass of sealed class can have multiple instance.

in Sealed class Type of Property & value will change. Subclasses of a sealed class are either ordinary classes, data classes or sealed classes themselves and hence it is easy to contain the state of the subclass.

**sealed class Post{**

class Error(var text: String) : Post()

class Success(var titleMsg: String, var DescripMsg: String) : Post()

}

Post. Error(errorMsg)

Or || Both

Post. Success(titleMsg, DescripMsg)

**Enum Class:** Enumerations/Enum in Kotlin are data types that hold a set of constants. In Enum type of property is fixed but value with change.

Enum classes are used to model types that represent a finite set of distinct values, such as directions, states, modes, and so forth.

Note: **values()** returns the enum constants in the form of an array over which we can iterate to retrieve each enum constant. Enum types are constants and hence, it is difficult to maintain different states of the instances.

**valueOf()** is used to fetch an enum constant using a String as the argument.

**enum class Months{**

January,

February,

March

}

**enum class MonthsWithVal(var shorthand: String) {**

January("JAN"),

February("FEB"),

March("MAR");

}

println(Months.January) //prints January

for (enum in Months.values()) {

println(enum.name) //prints January, February, March

}

var x = Months.January

println(x.shorthand) //prints JAN

**Difference between enum and sealed classes**

1. Enum classes allow only a single type for all constants. Here’s where sealed classes come to our rescue by allowing multiple instances.
2. Enums don't get garbage collected, they stay in memory for the lifespan of your app. This can be an upside or a downside. Objects of the subtypes of the sealed classes get garbage collected like the objects of regular classes.

**Open Method/class:** open method in kotlin means **that the method can be overridden** in subclass, because by default all method in kotlin are final. Instead in Java all the methods can be overridden by default. **The method of an open** class cannot be overridden by default as usual (doesn't matter if the class is not open), they must be declared that they can be overridden :

Kotlin has a special feature i.e. classes and metdhods are not open for extension & overridable by default in kotlin, which means they are by default final class or final function. It means Open classes and methods in Kotlin are equivalent to the opposite of final in Java, an open method is overridable and an open class is extendable in Kotlin.

**open class Car{**

fun steering{…}

open fun steering2{...}

}

**class Supercar:Car{**

override fun steering2 {...} // will worked

override fun steering {...} // But give an error

}

**Extension Function :** Extension functions are the functions where you add a functionality to your/an existing class by just adding a function without extending the class and this will be resolved statically and all the objects of that class can use it.

Extension Function help to add more functionality on our existing class without inheriting them.

To declare an extension function, prefix its name with a *receiver type*, which refers to the type being extended. The following adds a swap function to MutableList<Int>:

**fun MutableList<Int>.swap(index1: Int, index2: Int) {**

val tmp = this[index1] // 'this' corresponds to the list

this[index1] = this[index2]

this[index2] = tmp

}

val list = mutableListOf(1, 2, 3)

**list.swap(0, 2)**

// 'this' inside 'swap()' will hold the value of ‘list'

**Option2**

// Class type Int, means we can apply this on Int type varialbe

//Accepting Int param and Returning String value

**fun Int.Gender(type: Int): String{**

val gType = if(type == 1) "Male" else "Femel"

return gType

}

**1.Gender(1)** // Calling Extention function

2**.Gender(1)**  // Calling Extention function

**Lambda:** Lambda is a function which has no name. Lambda is defined with a curly braces **{}** which takes *variable as a parameter* (if any) and body of function. The *body of function* is written after variable (if any) followed by **->** operator.

Option1: val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }

Option2: val sum = { x: Int, y: Int -> x + y }

**Calling:** println(sum(10, 21))

**Note:**

1. A lambda expression is always surrounded by curly braces.
2. Parameter declarations in the full syntactic form go inside curly braces and have optional type annotations.
3. The body goes after the ->
4. if the inferred return type of the lambda is not Unit, the last (or possibly single) expression inside the lambda body is treated as the return value.

**Higher Order Function:** In Kotlin, **a function which can accepts one/more function as parameter or can returns a function** as output is called Higher-Order function. Instead of Integer, String or Array as a parameter to function, we will pass anonymous function or lambdas. Higher-order functions are just like regular functions with an added ability of receiving and returning other functions are arguments and output.

**var lambda =** {a: Int , b: Int -> a + b }

// higher order function & which accepting lambda as parameter

**fun higherfunc( lmbd: (Int, Int) -> Int) {**

   var result = **lmbd(2,4)**

// invokes the lambda expression by passing parameters

   println("The sum of two numbers is: $result")

}

**fun main(args: Array<String>) {**

    higherfunc(lambda)           //passing lambda as parameter

}

**Note:**

**lmbd:** is local name for the receiving lambda parameter.

**():** represents that the function does accept two arguments as Int type.

**Init** represents that function that return Int value.

**Calling:** In main function, we have invoked the higher-order function by passing the lambda expression as parameter

**Scope Function** => let, run, with, apply, also

Scope functions make code more **clear**, **readable**, and **concise** which are Kotlin language’s main features.

By definition, Scoped functions are functions that execute a block of code within the context of an object and you can access the object without its name..

**Note**: If the property is nullable, you can use **let** or **runImage**

1. **let** function is often used to provide null safety calls. Use **safe call operator(?.)** with ‘let’ for null safety. It executes the block only with the non-null value.

// nullable variable a

    // with value as null

    var a: Int? = **null**

    // using let function

    a?.let {

        // statement(s) will

        // not execute as a is null

        print(it)

    }

    // re-initializing value of a to 2

    a = 2

    a?.let {

        // statement(s) will execute

        // as a is not null

        print(it)

    }

1. **Apply** As the name implies – “Apply these to the object”. It can be used to operate on members of the receiver object mostly to initialize members

**class** Company() {

    lateinit var name: String

    lateinit var objective: String

    lateinit var founder: String

}

fun main() {

  val comData = Company().apply {

        // same as founder = “Sandeep Jain”

**this**.founder = "Sandeep Jain"

        name = "GeeksforGeeks"

        objective = "A computer science portal for Geeks"

    }

}

**apply : returns its receiver**

**val intent = Intent().*apply* {**

putExtra("Key", "value")

putExtra("Key1", "value")

*action* = "CUSTOM\_APP\_ACTION"

**}**

// here intent's type will be **Intent** ✅

1. **with:** Recommended use of ‘with’ for calling functions on context objects without providing the lambda result. If the property is non-nullable, you can use with

**class** Company() {

    lateinit var name: String

    lateinit var objective: String

    lateinit var founder: String

}

fun main() {

    val gfg = Company().apply {

        this.name = "GeeksforGeeks"

        objective = "A computer science portal for Geeks"

        founder = "Sandeep Jain"

    }

    // with function

    with(gfg) {

        // similar to println( "${this.name}" )

        println(" ${this.name} ")

    }

}

**4. run:**

let : returns the result of thelambda

**Infix functions:** Member functions and extensions with a single parameter can be turned into infix functions.

Infix functions must meet the following requirements: They must be member functions or extension functions. They must have a single parameter. The parameter must not accept variable number of arguments and must have no default value

1. The function is either defined on a class or is an extension method for a class

2. The function takes exactly one parameter

3. The function is defined using the infix keyword

infix fun Int.times(str: String) = str.repeat(this) // 1

println(2 times "Bye ") // output will be Bye Bye

infix fun Double.Percentage(ROI: Double) = ((this \* ROI) / 100) + this

val per = 1000 *Percentage* 9.75

*println*("Ammount 11 % percentahe" +per)

**Varargs** allow you to pass any number of arguments by separating them with commas.

fun printAll(vararg messages: String) { // 1

for (m in messages) {

println(m)

}

}

val val = printAll("Hello", "Hallo", "Salut", "Hola", "Jai")

**Collection in Kotlin:**

**List** is an ordered collection with access to elements by indices – integer numbers that reflect their position. Elements can occur more than once in a list. An example of a list is a telephone number: it's a group of digits, their order is important, and they can repeat.

**Set** is a collection of unique elements. It reflects the mathematical abstraction of set: a group of objects without repetitions. Generally, the order of set elements has no significance. For example, the numbers on lottery tickets form a set: they are unique, and their order is not important.

**Map** (or dictionary) is a set of key-value pairs. Keys are unique, and each of them maps to exactly one value. The values can be duplicates. Maps are useful for storing logical connections between objects, for example, an employee's ID and their position.

1. **Immutable Collection**:

2. **Mutable Collection**:

1. **Immutable Collection**: It means that, it supports only read-only functionalities and can not be modified its elements. Immutable Collections and their corresponding methods are:

List – listOf() and listOf<T>()

Set – setOf()

Map – mapOf()

**ListOf()** – It’s basically follow the insertion order and It is an ordered collection in which we can access elements or items by using indices – integer numbers that define a position for each element. Elements can be repeated in a list any number of times. We can not perform add or remove operations in the immutable list. A list is a generic ordered collection of elements that can contain duplicate values. And, List in Kotlin is an interface that extends the Collection interface.

val immutableList = listOf("Mahipal","Nikhil","Rahul")

immutableList.add = “Praveen” // gives compile time error

println(immutableList.size) // get the size of list

println(immutableList.indexOf(‘Mahipal’)) // get index of item

println(immutableList[2]) // get the element at index

for(item in immutableList){}. // print the element/index wise

for((index, element) in immutableList.withIndex()){}//item(element) with index

for (i in 0 until immutableList.size) { } // run the loop utill list size

val iterator = immutableList.listIterator() // Iterate the item

while (iterator.hasNext()) { // run the while loop if hasNext = true

val country = iterator.next() // return the value

}

**SetOf()**: It is a collection of unordered elements also it does not support duplicate elements. It is a collection of unique elements. Generally, the order of set elements does not have a significant effect. We can not perform add or remove operations because it is an immutable Set

// initialize with duplicate values but output with no repetition

var immutableSet = setOf(6,9,9,0,0,"Mahipal","Nikhil")

immutableSet.add(7) //// gives compile time error

for(item in immutableSet){

println(item)

}

**MapOf():** Map keys are unique and hold only one value for each key, it is a set of key-value pairs. Each key maps to exactly one value. The values can be duplicates but keys should be unique. Maps are used to store logical connections between two objects, for example, a student ID and their name. As it is immutable its size is fixed and its methods support read-only access.

var immutableMap = mapOf(9 to "Mahipal",8 to "Nikhil",7 to "Rahul")

immutableMap.put(9,”Praveen") // // gives compile time error

for(key in immutableMap.keys){

println(immutableMap[key])

}

2**. Mutable Collection**: It supports both read and write functionalities. Mutable collections and their corresponding methods are:

**List** – mutableListOf(),arrayListOf() and ArrayList

**Set** – mutableSetOf(), hashSetOf()

**Map** – mutableMapOf(), hashMapOf() and HashMap

**MutableListOf() & ArrayListOf()**: List – Since mutable list supports read and write operation, declared elements in the list can either be removed or added.

var mutableList = mutableListOf("Mahipal","Nikhil","Rahul")

mutableList[0] = “Praveen” // we can modify the element

mutableList.add(“Abhi") // add one more element in the list

mutableList.add(3, “CK”) // add one more element in the list at index 3

mutableList.set(3, “CK”) // add one more element in the list at index 3

mutableList.addAll(listOf("Singapore", “Moscow")) // add list of data

mutableList.remove(“Abhi") // remove one more element from the list

mutableList.removeAt(0) /remove one more element from the list of index 0

for(item in mutableList){

println(item)

}

**MutableSetOf(), HashSetOf():**  The mutable Set supports both read and write functionality. We can access add or remove elements from the collections easily and it will preserve the order of the elements

var mutableSet = mutableSetOf<Int>(6,10)

mutableSet.add(2) // adding elements in set

mutableSet.add(5)

for(item in mutableSet){

println(item)

}

M**utableMapOf(), HashMapOf():** It is mutable so it supports functionalities like put, remove, clear,

var mutableMap = mutableMapOf<Int,String>(1 to "Mahipal",2 to "Nikhil",3 to "Rahul")

mutableMap.put(1,”Praveen") // we can modify the element

mutableMap.put(4,"Abhi") // add one more element in the list

for(item in mutableMap.values){

println(item)

}