COMPARISON

	ANDROID NATIVE	FLUTTER	REACT NATIVE	KMM
LANGUAGE				
COMMUNITY SUPPORT	HUGE	NOT THE BEST BUT STILL HUGE ENOUGH AS ITS CLOSE TO KOTLIN	HUGE	BAD AS IT IT NEW
DEVICE CAPABILTY	FULL CAPABILITY	3RD PARTY PLUGINS BRING FULL CAPABILITY	3RD PARTY PLUGINS BRING FULL CAPABILITY	CAN'T WRITE UIS FOR IOS
COST	EXTRA EFFORT ON IOS	10-20% SIZE INCREASE	10-20% SIZE INCREASE	NATIVE - SAME IOS - UNKNOWN
PERFORMANCE	BEST	NEAR TO BEST	IT RELIES ON A BRIDGE TO NATIVE CODE	NOT A BAD OPTION

Kotlin Crash Course for Android

- Setup Java JDK for your OS(compiles to JVM bytecode)
- Basics of Kotlin
  - Data Types
    - Primitive(Built in Data Type)
    - Non-primitive(User Generated)
  - Variables(Use of Val and Var instead of specific data type like int,String etc., null safety)
    - Local variable
    - Instance variable
    - No Static Variable(Here we have Companion Object)
  - Null Safety in Kotlin
    - Safe Call Operator(?)
    - Elvis Operator(?:)
    - Non Null Assertion Operator(!!)
  - if-else statement
  - When statement (Similar to 'Switch Case' in JAVA)
  - Loops(Break and Continue Statement with Labeled loop)
    - For loop
    - While loop
    - Do-while loop
  - Operator
    - Unary
    - Binary
  - methods
    - Kotlin Function As Expression
    - Named Parameter, default parameter and vararg in function
    - Lambada and Higher Order Function
    - Extension Function
  - keyword
- Common Object Oriented Programming Concepts(OOPS)
  - Object
  - Class
  - Constructor
    - Primary constuctor
    - Secondary constructor
  - Inheritance
  - Polymorphism
    - Overloading
    - Overriding
  - Abstraction
  - Access Modifiers
    - private
    - protected
    - public
    - internal
  - Data Class
- Kotlin collections
  - List and MutableList
  - Set and MutableSet
  - Map and MutableMap
  - Predicates : all,any,count and find
  - filter and map
- Other Commonly Used Concepts
  - Kotlin Coroutines
  - Java inter operability
  - Sealed Class
  - Enums
  - Exception Handling
  - Companion Object
  - Interfaces
  - Scope Functions: apply, also, let, with and run
  - Type Checking and Smart Casting
  - getters and setters in kotlin(Concept of Backing field)
  - lateinit keyword for variable
  - Safe Call and Elvis operator

Chat GPT: https://chatgpt.com/share/d6856862-0523-4838-8dea-79d54397ce09

What is Kotlin?

- It is object-oriented language, and a “better language” than Java, but still be fully interoperable with Java code.

- It is sponsored by Google, announced as one of the official languages for Android Development in 2017.

- Combines object-oriented and functional programming features.

Why Kotlin?

- **Concise:** it is more concise than Java and you would need to write approximately 40% fewer lines of code when compared to Java.

- **Interoperability:** this programming language is highly interoperable with Java. You will never face any difficulty using Kotlin in a Java project.

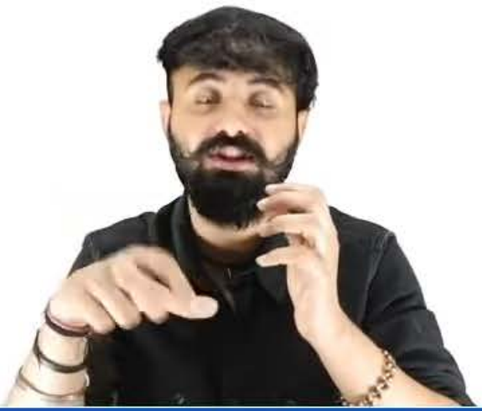
Why Kotlin?

- **Feature-rich:** it provides several advanced features such as Operator overloading, Lambda expressions, String templates, etc.

- **Easy:** it is easy to learn programming language. If you have come from a Java background, you would find it easy to learn Kotlin.

Why Kotlin?

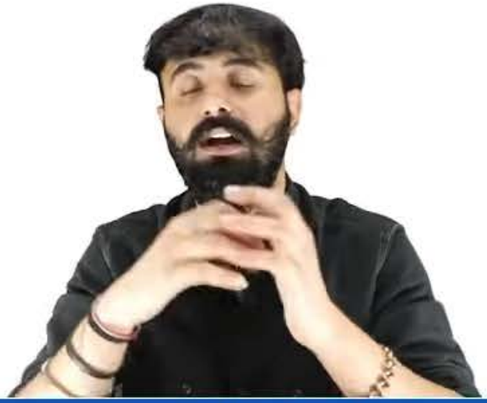
- **Less error-prone:** As mentioned before, Kotlin is a statically-typed programming language, which makes you able to catch errors at compile-time, as Statically typed programming languages do type checking at compile-time.



Why Kotlin?

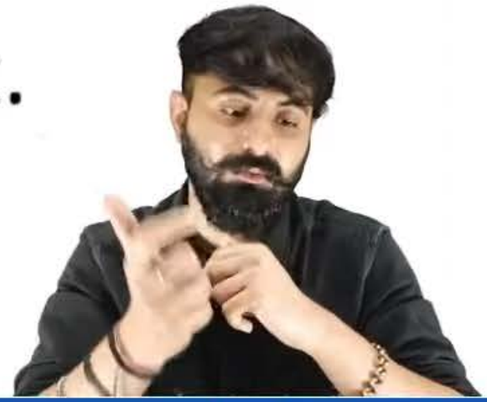
- **Safe:** It provides the safety from most annoying and irritating NullPointerExceptions by supporting nullability as part of its system. Every variable in Kotlin is non-null by default.

- **Smart Cast:** It explicitly typecasts the immutable values and inserts the value in its safe cast automatically.



Where Kotlin is used?

- You can use Kotlin to build Android Application.
- Kotlin can also compile to JavaScript, and making it available for the frontend.
- It is also designed to work well for web development and server-side development.

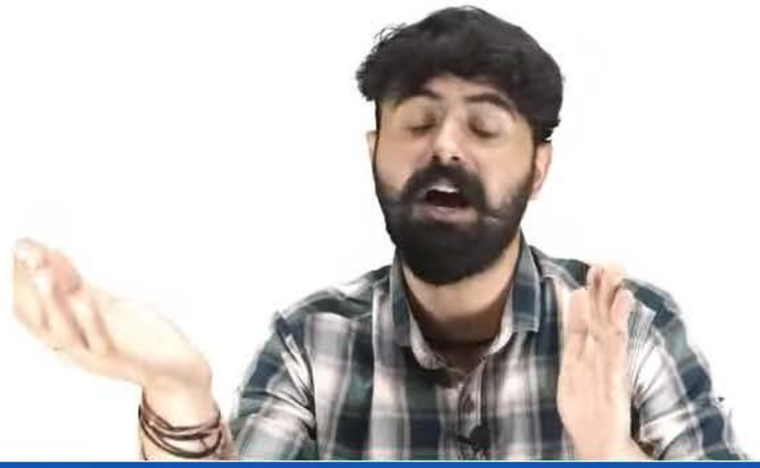


Classes and Object

WsCube Tech™
System For Satisfaction

- a class is a blueprint, and an object is an instance of a class.

- Usually, we define a class and then create multiple instances of that class by creating Objects.

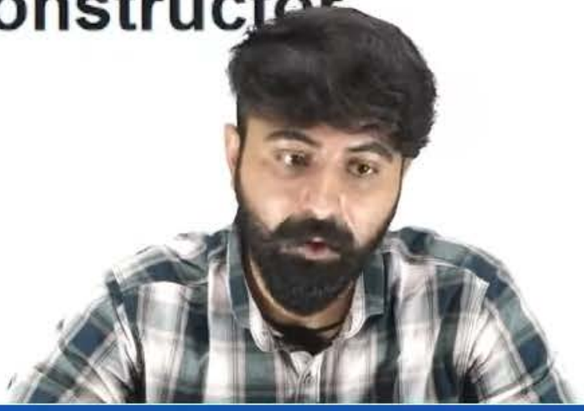


Constructors

WsCube Tech™
System For Satisfaction

- A constructor is a special member function that is invoked when an object of the class is created primarily to initialize variables or properties.

- A class needs to have a constructor and if we do not declare a constructor, then the compiler generates a default constructor.



Constructors

WsCube Tech™
System For Satisfaction

- Kotlin has two types of constructors –

  Primary Constructor

  Secondary/Custom Constructor

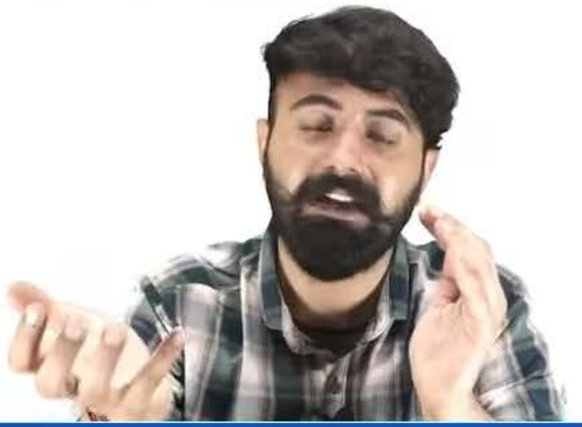
Constructors

WsCube Tech™
System For Satisfaction

- **Primary Constructor**

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

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

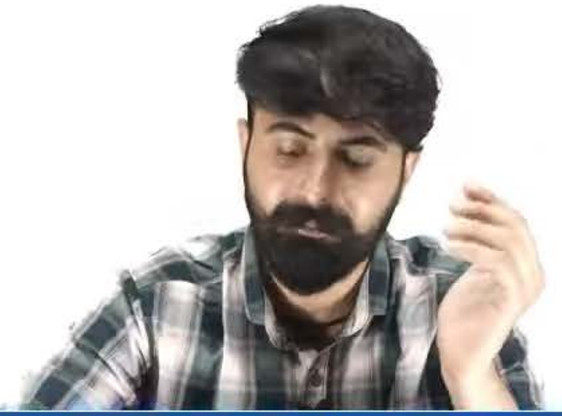


Init Block

WsCube Tech™
System For Satisfaction

- The primary constructor cannot contain any code, the initialization code can be placed in a separate initializer block prefixed with the init keyword.

```
class Sum {
    //code

    init {
        println("Init Called")
    }
```

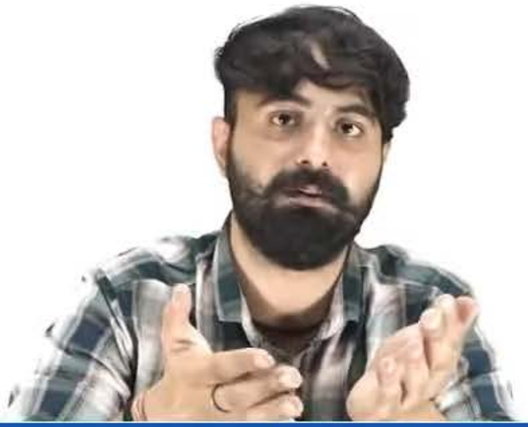


Custom Constructors

WsCube Tech™
System For Satisfaction

- A class in Kotlin can have at most one primary constructor, and one or more custom/secondary constructors.

- The primary constructor initializes the class, while the secondary constructor is used to initialize the class and introduce some extra logic.

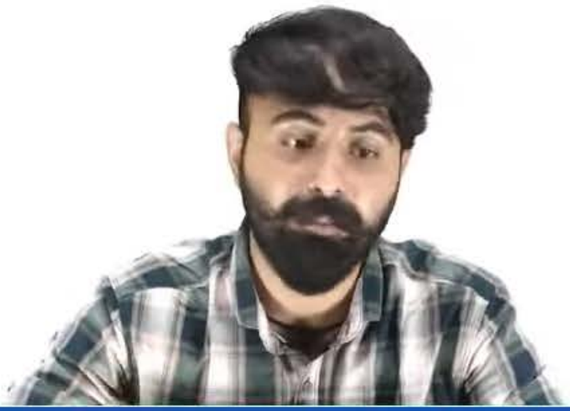


Constructors

WsCube Tech™
System For Satisfaction

- **Secondary Constructor**

```
constructor(a : Int){
    println("Result $a")
}

constructor(a : Int, b: Int){
    println("Result ${a+b}")
}
```



Companion Object

WsCube Tech™
System For Satisfaction

- In Kotlin or any other programming language like Java and C#,

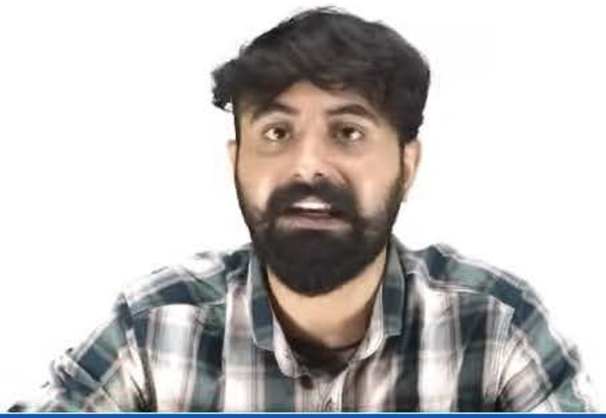
- whenever we want to call the method or whenever we want to access the members of a class then we make the object of the class and with the help of that object, we access the members of the class.

Companion Object

WsCube Tech™
System For Satisfaction

- In some languages like Java and C#, we use static keyword to declare the members of the class and use them without making any object i.e. just call them with the help of class name.

- There is nothing called static in Kotlin. So, in Kotlin, we use a companion object.

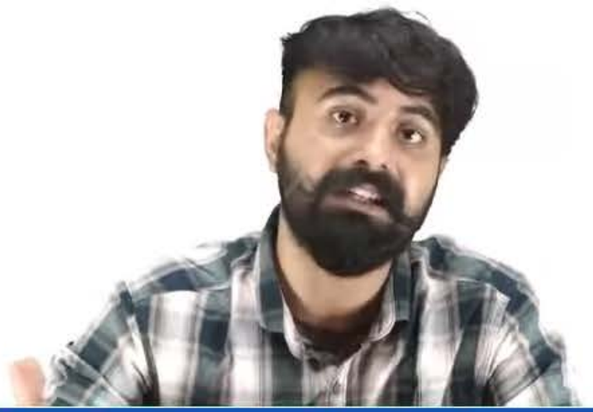


Companion Object

WsCube Tech™
System For Satisfaction

- **Example:**

```
class Sum () {
    // code
    var anotherInt: Int = 20
    companion object{
        var someInt: Int = 10
        fun callMe() = println("This method is called!")
    }

    Sum.someInt
    Sum.callMe()

    Sum().anotherInt
```



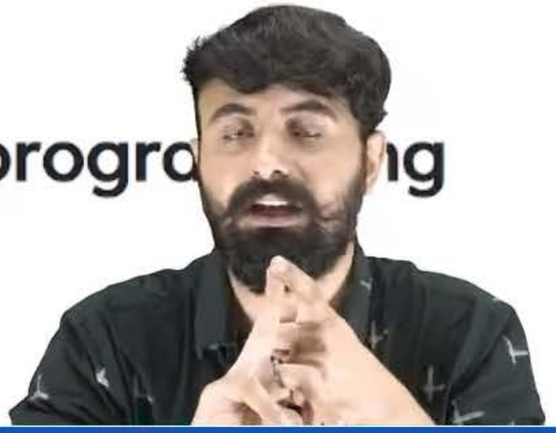
Inheritance

WsCube Tech™
System For Satisfaction

- It is the mechanism by which one class is allow to inherit the features(fields and methods) of another class.

- One object acquires all the properties and behaviors of a parent object.

- It is an important part of OOPs (Object Oriented progra ng system).

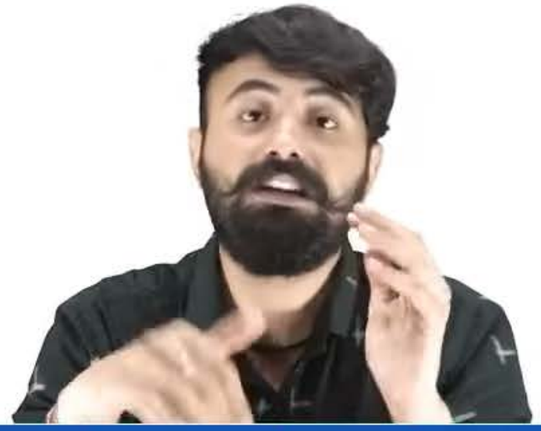


Inheritance

WsCube Tech™
System For Satisfaction

- Subclass can reuse methods and fields of the Parent class.

- By default, Kotlin classes are final – they can't be inherited. To make a class inheritable, mark it with the open keyword



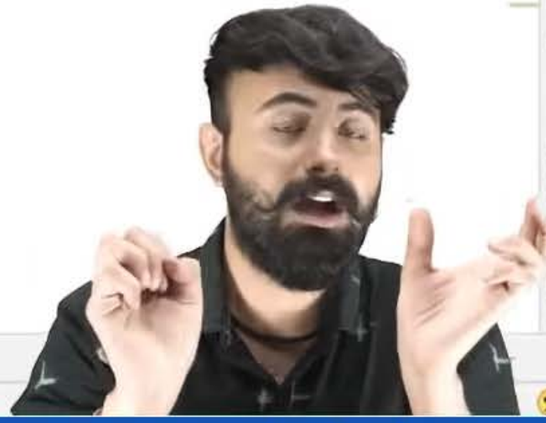
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help KotlinBasics - KotlinBasics.kt [KotlinBasics.app]

src main java wscube devdroid kotlinbasics KotlinBasics.kt main() KotlinBasicsKt Pixel 4 API 28

KotlinBasics.kt

```
1  package wscube.devdroid.kotlinbasics
2
3  fun main(){
4
5      val classB = ClassB()
6
7      println("The Name is: "+classB.name)
8      println("The sum is: "+classB.Add( a: 5, b: 6))
9  }
10
11 open class ClassA {  //Parent Class
12
13     open var name = "John"
14
15     open fun Add(a: Int, b: Int) : Int{
16         return a+b
17     }
```

WsCube Tech System For Satisfaction



www.wscubetech.com

TODO 6: Problems Terminal Database Inspector Profiler 4: Run 5: Debug Build Logcat

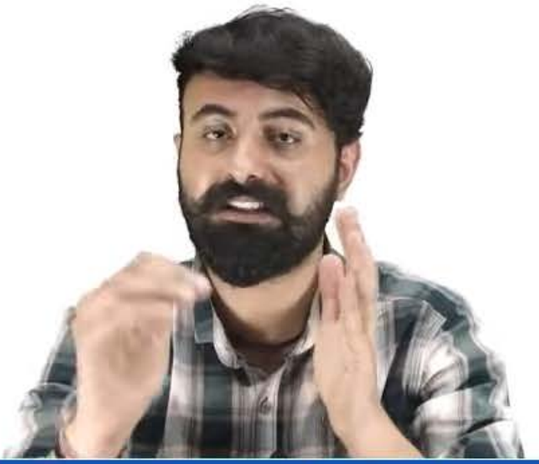
Gradle build finished in 4 s 970 ms (7 minutes ago)

this vs super

WsCube Tech™
System For Satisfaction

- The this keyword points to a reference of the current class, while the super keyword points to a reference of the parent class.

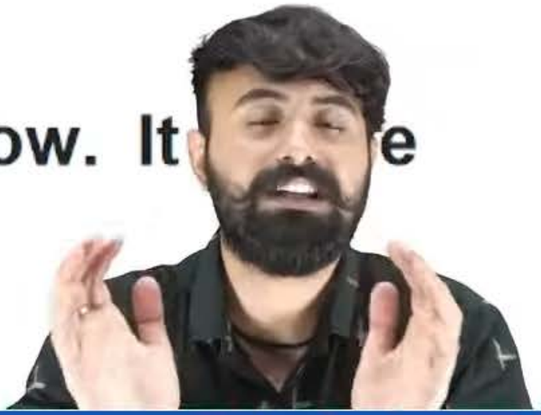
- this can be used to access variables and methods of the current class, and super can be used to access variables and methods of the parent class from the subclass.



Interface

WsCube Tech™
System For Satisfaction

- An interface is a reference type.
- It is similar to class.
- It is a collection of abstract methods.
- A class implements an interface, thereby inheriting the abstract methods of the interface.
- Only method signature, no body
- Interfaces specify what a class must do and not how. It is the blueprint of the class.

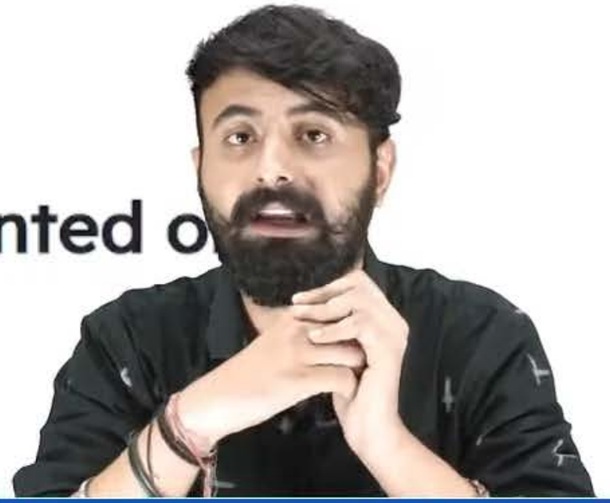


Abstraction

- Abstraction is one of the core concepts of Objected Oriented Programming.

- When there is a scenario that you are aware of what functionalities a class should have,

- but not aware of how the functionality is implemented o



www.wscubetech.com

Abstraction

- if the functionality could be implemented in several ways, it is advised to use abstraction.

- It contains abstract or non-abstract : variables and functions.

- Abstract Class cannot be initiated but could be exten s a Super Class.

www.wscubetech.com

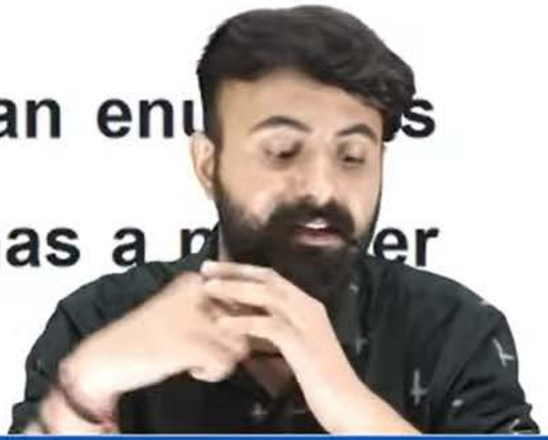
Enum

WsCube Tech™
System For Satisfaction

- In programming, sometimes there arises a need for a type to have only certain values.

- To accomplish this, the concept of enumeration was introduced. Enumeration is a named list of constants.

- In Kotlin, like many other programming languages, an enu s its own specialized type, indicating that something has a n er of possible values.



Properties Aur Methods Add Karna

Enums mein properties aur methods bhi ho sakte hain. Har enum constant ke alag-alag values ho sakte hain.

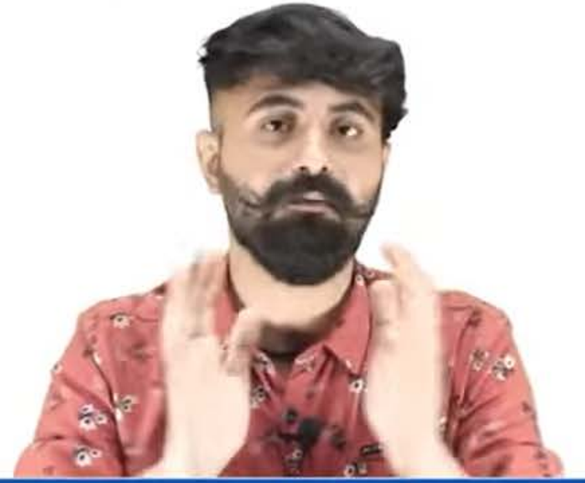
kotlin Copy code

```kotlin
enum class Direction(val degrees: Int) {
    NORTH(0),
    EAST(90),
    SOUTH(180),
    WEST(270);

    fun description(): String {
        return "Direction $name is $degrees degrees from North."
    }
}

fun main() {
    val direction = Direction.EAST
    println(direction.degrees) // Output: 90
    println(direction.description()) // Output: Direction EAST is 90 degrees from North.
}
```

Generics

WsCube Tech™
System For Satisfaction

- Generic is defined as a product without a brand name.
- The definition of generic is something without a brand name.
- An example of generic is the type of soap with a store's label that says "soap," but without a brand name.

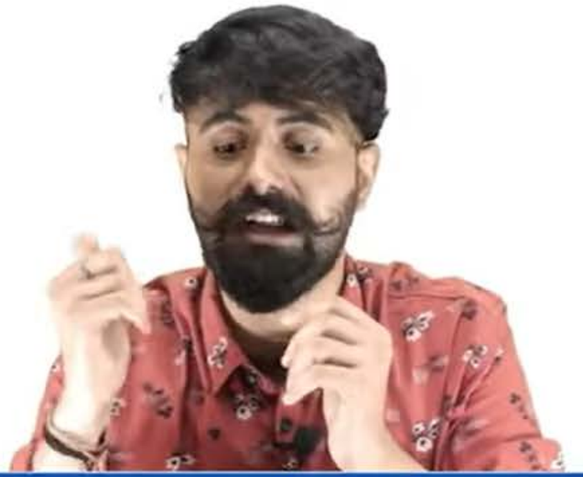


Generics

WsCube Tech™
System For Satisfaction

- Generics are the powerful features that allow us to define classes, methods and properties

- which are accessible using different data types while keeping a check of the compile-time type safety.

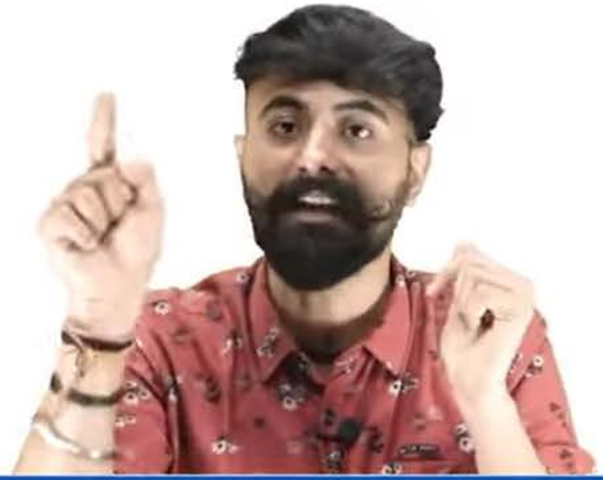


Generics

WsCube Tech™
System For Satisfaction

- A generic type is a class or method that is parameterized over types.

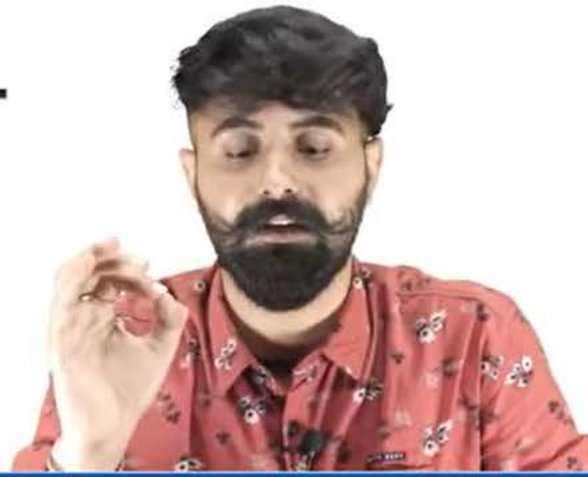
- We always use angle brackets () to specify the type parameter in the program.



Advantages of Generics

WsCube Tech™
System For Satisfaction

- Type casting is evitable- No need to typecast the object.
- Type safety- Generic allows only single type of object at a time.
- Compile time safety- Generics code is checked at compile time for the parameterized type so that it avoids run time error.



Lambdas

Lambda Expression

Parameters

Body

```
{x, y -> x + y}
{x -> x * x}
{x, y, z -> x + y + z}
{str -> println(str)}
```

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help KotlinBasics - KotlinBasics.kt [KotlinBasics.app]

java > wscube > devdroid > kotlinbasics > KotlinBasics.kt > printName: () -> Unit

KotlinBasicsKt | Pixel 4 API 28

1.50

KotlinBasics.kt

```
1  package wscube.devdroid.kotlinbasics
2
3  fun main() {
4
5      println("The Square of 5 is: "+ sqr(5) )
6
7      println("The Sum is: "+ add(5,6))
8
9      printName()
10
11 }
12
13 var num : Int = 11
14
15 val sqr = {x: Int -> x*x}
16
17 val add: (Int, Int) -> Int = {x,y -> x+y}
```

WsCube Tech® System For Satisfaction

Flutter Inspector | Flutter Outline | Flutter Performance | Device File Explorer

www.wscubetech.com

TODO | 6: Problems | Terminal | Database Inspector | Profiler | 4: Run | Build | Logcat

Gradle build finished in 3 s 481 ms (a minute ago)

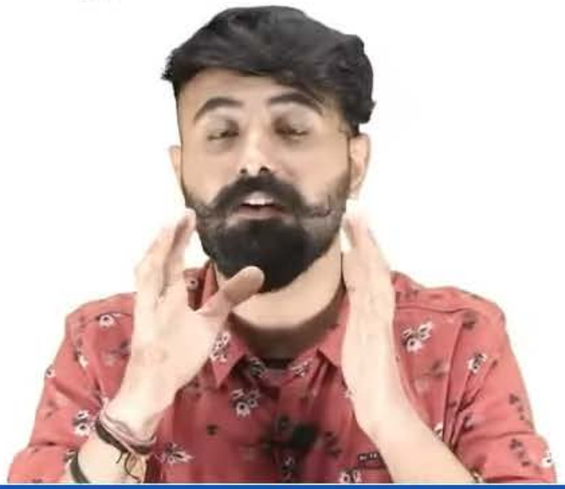
High Order Functions

1.80

WsCube Tech™
System For Satisfaction

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

- Kotlin functions can be stored in variables and data structures, passed as arguments to and returned from other higher-order functions.



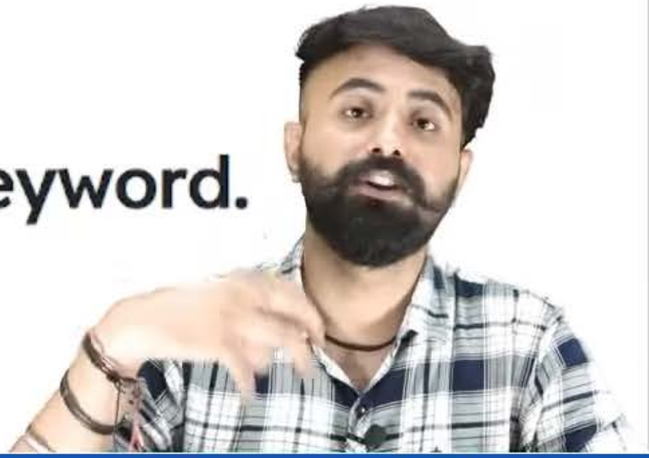
Delegations

WsCube Tech™
System For Satisfaction

- Inheritance implementation in classes and functions can be altered with the help of delegation techniques.

- Object-oriented programing languages support it innately without any boilerplate code.

- Delegation is used in Kotlin with the help of “by” keyword.



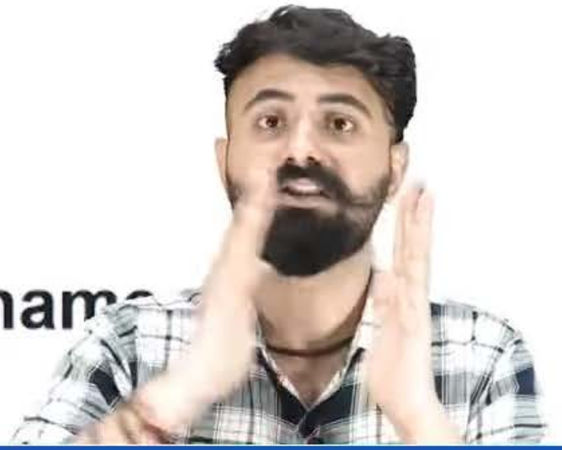
Scope Functions

WsCube Tech™
System For Satisfaction

- 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 nam



Types of Scope Functions

WsCube Tech
System For Satisfaction

with

Return: lambda result

Context object: this

let

Return: lambda result

Context object: it

run

Return: lambda result

Context object: this

apply

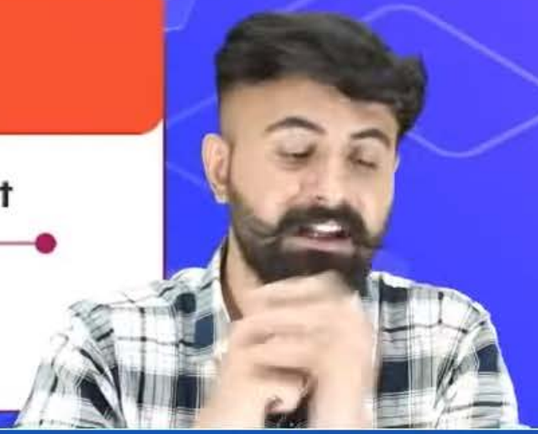
Return: context object

Context object: this

also

Return: context object

Context object: it



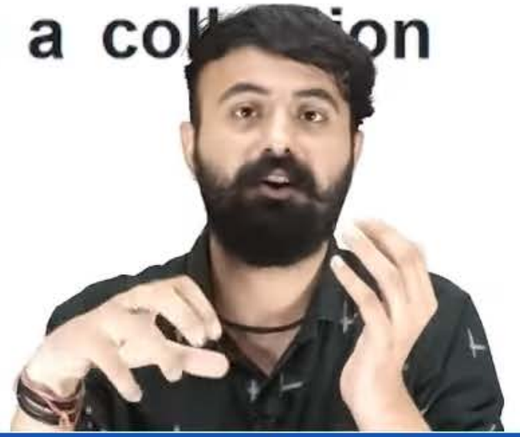
Collections

WsCube Tech™
System For Satisfaction

- A collection usually contains a number of objects (this number may also be zero) of the same type.

- Objects in a collection are called elements or items.

- For example, all the students in a department form a collection that can be used to calculate their average age.



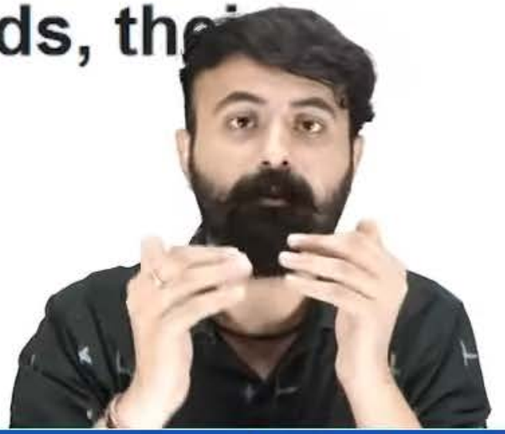
Collections

WsCube Tech™
System For Satisfaction

The following collection types are relevant for Kotlin:

- List

1. It is an ordered collection with access to elements by indices – integer numbers that reflect their position.
2. Elements can occur more than once in a list.
3. An example of a list is a sentence: it's a group of words, their order is important, and they can repeat.



Collections

WsCube Tech™
System For Satisfaction

The following collection types are relevant for Kotlin:

- Set

1. It is a collection of unique elements.
2. It reflects the mathematical abstraction of set: a group of objects without repetitions.
3. Generally, the order of set elements has no significance. For example, an alphabet is a set of letters.

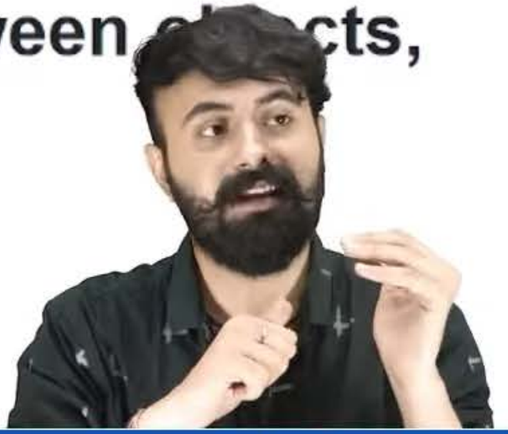
Collections

WsCube Tech™
System For Satisfaction

The following collection types are relevant for Kotlin:

- Map

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



File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help KotlinSample - MyClass.kt [KotlinSample.Practise] - Android Studio

KotlinSample › Practise › src › main › java › com › devdroid › practise › MyClass

MyClass.kt

```
4
5       companion object{
6
7           @JvmStatic
8           fun main(args: Array<String>) {
9               println("Hello World!")
10
11              val aList = listOf("Raman", 1, "Rajeev", 2.0, true)
12
13              println(aList)
14
15          }
16
17      }
18
```

MyClass › companion object › main()

TODO Terminal Database Inspector 0: Messages 4: Run 5: Debug Profiler Build 6: Logcat

Gradle build finished in 775 ms (moments ago)

4 chars 11:63

Type here to search

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help KotlinSample - MyClass.kt [KotlinSample.Practise] - Android Studio

KotlinSample > Practise > src > main > java > com > devdroid > practise > MyClass

MyClass.kt

```
9       println("Hello World!")
10
11      val aList = listOf("Raman", 1, "Rajeev", 2.0, listOf(1,2,3))
12
13      println(aList)
14
15      val mList = mutableListOf<Any>("Raman", "Rajeev")
16
17      mList.add(0,)
18      mList.add("Ramiz")
19      mList.add(1)
20
21      val mAList = mutableListOf("Pawan", false)
22
23      mList.addAll(mAList)
```

MyClass > companion object > main()

TODO Terminal Database Inspector 0: Messages 4: Run 5: Debug Profiler Build 6: Logcat

Typo: In word 'Rajeev'

15:59

Type here to search

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help KotlinSample - MyClass.kt [KotlinSample.Practise] - Android Studio

KotlinSample > Practise > src > main > java > com > devdroid > practise > MyClass

MyClass.kt

```
7       @JvmStatic
8       fun main(args: Array<String>) {
9           println("Hello World!")
10
11          val aSet = setOf("12",1,"Raj", "Rahul")
12
13          println(aSet)
14
15          val mSet = mutableSetOf("Raj", "Rajeev", 1, false)
16
17          mSet.add("true")
18
19          println(mSet)
20
21
```

MyClass > companion object > main()

TODO Terminal Database Inspector 0: Messages 4: Run 5: Debug Profiler Build 6: Logcat

Gradle build finished in 676 ms (2 minutes ago)

20:1

Type here to search

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help KotlinSample - MyClass.kt [KotlinSample.Practise] - Android Studio

KotlinSample > Practise > src > main > java > com > devdroid > practise > MyClass

MyClass.kt

```
1  package com.devdroid.practise
2
3  class MyClass {
4
5      companion object{
6
7          @JvmStatic
8          fun main(args: Array<String>) {
9              println("Hello World!")
10
11             val aMap = mapOf(1 to "Raman", "Raman" to "Rahul", true to "Check")
12
13             println(aMap)
14
15
```

MyClass > companion object > main()

TODO Terminal Database Inspector 0: Messages 4: Run 5: Debug Profiler Build 6: Logcat

Gradle build finished in 995 ms (a minute ago)

11:30

Type here to search

Kotlin Full Course for Beginners [FREE] | Android Kotlin Tutorial | Learn Kotlin in 6 Hours

1.90

```kotlin
fun main(args: Array<String>) {
    println("Hello World!")

    val aMap = mapOf(1 to "Raman", "Raman" to "Rahul", true to "Check")

    println(aMap)

    val mMap = mutableMapOf<Int, String>()

    val mAMap = mutableMapOf(1 to "Raj", 12 to "Rohit", 11 to "Rajeev")

    mMap.putAll(mAMap)

}
```

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help KotlinSample - MyClass.kt [KotlinSample.Practise] - Android Studio

KotlinSample › Practise › src › main › java › com › devdroid › practise › MyClass

MyClass.kt

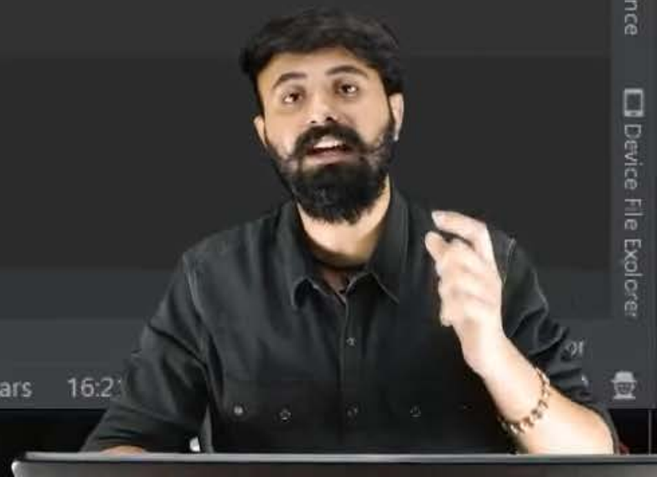
```
5    companion object{
6
7        @JvmStatic
8        fun main(args: Array<String>) {
9            println("Hello World!")
10
11           val arrNames = ArrayList<String>()
12
13           arrNames.add("A")
14           arrNames.add("B")
15
16           arrNames[0] = "Raman"
17
18
19
```

MyClass › companion object › main()

TODO Terminal Database Inspector 0: Messages 4: Run 5: Debug Profiler Build 6: Logcat

Gradle build finished in 853 ms (3 minutes ago)

3 chars 16:2



Type here to search

nullity.md

Kotlin mein nullability ek important concept hai jo null pointer exceptions (NPEs) ko handle karne ke liye introduce kiya gaya hai. Null pointer exceptions programming errors hain jo tab hoti hain jab aap kisi null object reference ko access karne ki koshish karte hain. Kotlin ne nullability ke liye language-level support provide kiya hai, jo aapko safer aur more reliable code likhne mein madad karta hai.

Nullable and Non-Nullable Types

Kotlin mein, har type by default non-nullable hota hai, matlab aap us variable ko `null` assign nahi kar sakte. Agar aapko kisi variable ko `null` assign karna hai, to aapko explicitly usko nullable declare karna padega.

Non-Nullable Type:

```
var name: String = "Kotlin"
// name = null  // Compilation error
```

Nullable Type:

```
var name: String? = "Kotlin"
name = null  // Allowed
```

Safe Calls (`?.`)

Safe call operator (`?.`) ka use aap nullable variables pe operations karne ke liye kar sakte hain bina null pointer exception ke risk ke. Agar variable null hota hai, to operation execute nahi hota aur null return hota hai.

Example:

```
val name: String? = null
val length = name?.length // length will be null
```

Elvis Operator (`?:`)

Elvis operator (`?:`) ka use aap default value provide karne ke liye kar sakte hain agar expression null return karta hai. Ye operator aapko null handling aur default values assign karne mein madad karta hai.

Example:

```
val name: String? = null
val length = name?.length ?: 0 // length will be 0
```

Safe Casts (`as?`)

Safe cast operator (`as?`) ka use aap casting ke liye kar sakte hain bina exception ke risk ke. Agar cast successful hota hai to casted value return hoti hai, warna null return hota hai.

Example:

```
val obj: Any = "Kotlin"
val str: String? = obj as? String // str will be "Kotlin"
```

The `let` Function

`let` function ka use aap nullable objects pe operations karne ke liye kar sakte hain agar wo null nahi hote. Ye function aapko safer aur more readable code likhne mein madad karta hai.

Example:

```
val name: String? = "Kotlin"
name?.let {
    println(it.length) // Prints length of name if name is not null
}
```

The `!!` Operator

`!!` operator ka use aap explicitly null pointer exception throw karne ke liye kar sakte hain agar variable null hota hai. Is operator ka use carefully aur sparingly karna chahiye, kyunki ye runtime exceptions throw karta hai.

Example:

```
val name: String? = null
val length = name!!.length // Throws NullPointerException
```

Late Initialization (`lateinit`)

Agar aapko kisi non-nullable variable ko baad mein initialize karna hai, to aap `lateinit` modifier ka use kar sakte hain. Ye modifier sirf `var` properties ke liye use hota hai aur primitive types ke liye allowed nahi hota.

Example:

```
lateinit var name: String

fun initializeName() {
    name = "Kotlin"
}

fun printName() {
    if (::name.isInitialized) {
        println(name)
    }
}
```

Nullable Types in Functions

Functions mein nullable types ko parameters aur return types ke roop mein handle kiya ja sakta hai. Aapko functions ke signature mein nullable types ko specify karna hota hai.

Example:

```
fun getNameLength(name: String?): Int {
    return name?.length ?: 0
}

fun main() {
    println(getNameLength("Kotlin")) // Prints 6
    println(getNameLength(null)) // Prints 0
}
```

Summary

Kotlin mein nullability ke concept se aap safer aur more reliable code likh sakte hain. Ye features aapko null pointer exceptions se bachate hain aur code ko concise aur readable banate hain. Yahan key concepts include:

- Non-Nullable aur Nullable Types
- Safe Calls (`?.`)
- Elvis Operator (`?:`)
- Safe Casts (`as?`)
- `let` Function
- `!!` Operator
- `lateinit` Modifier
- Nullable Types in Functions

Inn features ko samajhkar aur effectively use karke aap apne Kotlin code ko robust aur error-free bana sakte hain.

nem.md

In Kotlin, scope functions allow you to execute a block of code within the context of an object. They are particularly useful for improving code readability and conciseness. The five main scope functions in Kotlin are `let`, `run`, `with`, `apply`, and `also`. Each serves different purposes and has its own specific use cases. Let's dive into each of them in detail.

let

- **Purpose:** Executes a block of code on the object and returns the result of the block. It is often used for null checks and chaining operations on non-null objects.
- **Object Reference:** `it` (implicit name of the lambda argument)
- **Return Value:** The result of the lambda expression.
- **Usage Example:**

  ```
  val name: String? = "Kotlin"
  val length = name?.let {
      println("Name length: ${it.length}")
      it.length
  }
  ```

  In this example, `let` is used to perform operations on `name` if it is not null. The length of the name is printed and then returned.

run

- **Purpose:** Executes a block of code within the context of the object and returns the result of the block. It is often used when you want to perform operations on an object and return the final result.
- **Object Reference:** `this` (receiver object)

- **Return Value:** The result of the lambda expression.
- **Usage Example:**

  ```
  val person = Person("John", 30)
  val greeting = person.run {
      "Hello, my name is $name and I am $age years old."
  }
  println(greeting)
  ```

  Here, `run` is used to create a greeting message by accessing the properties of the `person` object directly using `this`.

with

- **Purpose:** Similar to `run`, but it is a non-extension function. It is used to call multiple methods on the same object and improve code readability.
- **Object Reference:** `this` (receiver object)
- **Return Value:** The result of the lambda expression.
- **Usage Example:**

  ```
  val person = Person("Jane", 25)
  val description = with(person) {
      "Name: $name, Age: $age"
  }
  println(description)
  ```

  In this example, `with` is used to create a description string by accessing the properties of the `person` object directly using `this`.

apply

- **Purpose:** Used to configure an object by applying a block of code to it. It is often used for object initialization and configuration.
- **Object Reference:** `this` (receiver object)
- **Return Value:** The object itself.
- **Usage Example:**

  ```
  val person = Person().apply {
      name = "Alice"
      age = 28
  }
  println(person)
  ```

  Here, `apply` is used to set the properties of the `person` object. The initialized object is then returned.

also

- **Purpose:** Similar to `let`, but it returns the original object. It is often used for performing additional operations on an object, such as logging or validation.
- **Object Reference:** `it` (implicit name of the lambda argument)
- **Return Value:** The object itself.
- **Usage Example:**

  ```
  val person = Person("Bob", 35).also {
      println("Created person: $it")
  }
  ```

In this example, `also` is used to log the creation of the `person` object. The original object is then returned.

Comparison and Use Cases

Function	Object Reference	Return Value	Use Case
`let`	`it`	Lambda result	Null checks, chaining operations
`run`	`this`	Lambda result	Object configuration and final result
`with`	`this`	Lambda result	Multiple operations on an object
`apply`	`this`	The object	Object initialization/configuration
`also`	`it`	The object	Additional operations (e.g., logging)

Understanding these scope functions and their appropriate use cases can significantly enhance the readability and maintainability of your Kotlin code. They allow you to write more concise and expressive code by reducing boilerplate and improving the clarity of object manipulations.