**Null Safety**

In Kotlin, null safety is a core feature designed to help developers avoid null pointer exceptions, a common source of bugs in many programming languages. Kotlin achieves null safety through its type system and a set of conventions.

Here's an overview of the null safety types and concepts in Kotlin:

1. Nullable Types:

In Kotlin, all types are non-nullable by default, meaning they cannot hold a null value. If you want a variable to be able to hold null, you must explicitly declare its type as nullable by appending `?` to the type. For example:

var nullableString: String? = null

1. Non-nullable Types:

These are regular types that cannot hold null values. If you try to assign null to a non-nullable type, it will result in a compilation error. For example:

var nonNullableString: String = "Hello"

// nonNullableString = null // This will cause a compilation error

1. Safe Calls (`?.`):

The safe call operator allows you to safely call a method or access a property on a nullable object. If the object is null, the call will return null instead of throwing a null pointer exception. For example:

val length: Int? = nullableString?.length

1. Elvis Operator (`?:`):

This operator provides a default value in case a nullable expression evaluates to null. It's handy for providing a fallback value when dealing with nullable types. For example:

val length: Int = nullableString?.length ?: -1

1. Not-null Assertion (`!!`):

This operator is used to assert that an expression is not null and throws a `NullPointerException` if it is. It should be used with caution because it bypasses the null safety checks of Kotlin. For example:

val length: Int = nullableString!!.length

**Ternary Operator**

In Java, the ternary operator is a concise way to express conditional statements. It's often used to assign a value to a variable based on a condition. The syntax of the ternary operator in Java is as follows:

variable = (condition) ? expression1 : expression2;

Here's how it works:

If the condition is true, expression1 is evaluated and assigned to variable.

If the condition is false, expression2 is evaluated and assigned to variable.

**== v/s === in Kotlin**

In Kotlin, as in many other programming languages, there's a difference between `==` and `===`:

1. `==` (Equality Operator): This operator is used to compare the **values** of two objects for equality. It typically checks whether the objects have the same content or data. For example:

val x: Int = 5

val y: Int = 5

val z: Int = 6

println(x == y) // Output: true (Values are equal)

println(x == z) // Output: false (Values are not equal)

The `==` operator is overloaded by classes in Kotlin to provide custom value comparison logic.

2. `===` (Identity Operator): This operator is used to compare the **references** of two objects to see if they refer to the same object in memory. It checks whether two references point to the same memory address. For example:

val a: String = "hello"

val b: String = "hello"

val c: String = a

println(a === b) // Output: false (Different objects)

println(a === c) // Output: true (Same object)

```

The `===` operator checks if two references refer to the same instance in memory.

In summary:

- Use `==` for comparing values. It's used to check if two objects have the same content.

- Use `===` for comparing references. It's used to check if two references point to the same object instance in memory.

**Extension Function in Koltin**

Extension functions in Kotlin allow you to add new functions to existing classes without modifying their source code. They are a powerful feature that enables you to extend the behavior of classes, including those from external libraries, without subclassing or inheritance.

**Syntax:**

fun ReceiverType.extensionFunctionName(parameters: ParameterType): ReturnType {

// Function body

}

ReceiverType: The type you're extending, also known as the receiver type. It's the type that the extension function is applied to.

extensionFunctionName: The name you give to your extension function.

parameters: Parameters that your extension function takes, if any.

ReturnType: The type returned by your extension function.