Kotlin Null 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.

NullPointerException can only possible on following causes:

- An forcefully call to throw NullPointerException();
- An uninitialized of this operator which is available in a constructor passed and used somewhere.
- Use of external java code as Kotlin is Java interoperability.

Kotlin Nullable Types and Non-Nullable Types

Kotlin types system differentiates between references which *can hold null* (*nullable reference*) and which cannot hold null (non null reference).

Normally,types of String are not nullable.

To make string which holds null value, we have to explicitly define them by putting a **?** behind the String as: **String?**

Nullable Types

Nullable types are declared by putting a **?** behind the **String** as:

```
var str1: String? = "hello"
str1 = null // ok
```

Kotlin example of nullable types

```
fun main(args: Array<String>){
    var str: String? = "Hello"
        // variable is declared as nullable
    str = null
    print(str)
}
```

Output:

```
null
```

Non Nullable Types

Non nullable types are normal strings which are declared as String types as:

```
val str: String = null // compile error
str = "hello" // compile error Val cannot be reassign
var str2: String = "hello"
str2 = null // compile error
```

What happens when we assign null value to non nullable string?

```
fun main(args: Array<String>){
var str: String = "Hello"
str = null // compile error
    print(str)
}
```

Output:

```
It will generate a compile time error.
Error:(3, 11) Kotlin: Null can not be a value of a non-
null type String
```

Checking for null in conditions

Kotlin's If expression is used for checking condition and returns value.

```
fun main(args: Array<String>){
    var str: String? = "Hello"
                // variable is declared as nullable
    var len = if(str!=null) str.length else -1
    println("str is : $str")
    println("str length is : $len")

    str = null
    println("str is : $str")
    len = if(str!=null) str.length else -1
    println("b length is : $len")
}
```

Output:

```
str is : Hello
str length is : 5
str is : null
b length is : -1
```

Smart cast

We have seen in previous tutorial Kotlin Nullable Types and Non-Nullable Types how nullable type is declared.

To use this nullable types we have an option to use smart casts.

Smart cast is a feature in which Kotlin compiler tracks conditions inside if expression.

If compiler founds a variable is not null of type nullable then the compiler will allow to access the variable.

For example:

When we try to access a nullable type of String without safe cast it will generate a compile error.

```
var string: String? = "Hello!"
    print(string.length) // Compile error
```

To solve the above expression we use a safe cast as:

```
fun main(args: Array<String>){
var string: String? = "Hello!"
    if(string != null) { // smart cast
print(string.length) // It works now!
    }
}
```

Output:

```
6
```

While using **is** or **!is** for checking the variable, the compiler tracks this information and internally cast the variable to target type.
This is done inside the scope **if** **is** or **!is** returns **true**.

Use of is for smart cast

```
fun main(args: Array<String>){
val obj: Any = "The variable obj is automatically cast to
a String in this scope"
    if(obj is String) {
          // No Explicit Casting needed.
          println("String length is ${obj.length}")
    }
}
```

Output:

```
String length is 64
```

Use of !is for smart cast

```
fun main(args: Array<String>){
    val obj: Any = "The variable obj is automatically
    cast to a String in this scope"
    if(obj !is String) {
        println("obj is not string")
    }else
        // No Explicit Casting needed.
        println("String length is ${obj.length}")
    }
```

Output:

```
String length is 64
```

Smart cast work according to the following conditions:

- A `val` variable always aspect for local properties.
- If a `val` property is private or internal the check is performed in the same module where the property is declared.
- If the local `var` variable is not modified between the check and the usage, is not captured in a lambda that modifies it.

Unsafe and Safe Cast Operator

Unsafe cast operator: as

Sometime it is not possible to cast variable and it throws an exception, this is called as unsafe cast.
The unsafe cast is performed by the infix operator as.

A nullable string (String?) cannot be cast to non nullabe string (String), this throw an exception.

```
fun main(args: Array<String>){
    val obj: Any? = null
    val str: String = obj as String
    println(str)
}
```

The above program throw an exception:

```
Exception in thread "main" kotlin.TypeCastException: null cannot be
cast to non-null type kotlin.String at TestKt.main(Test.kt:3)
```

While try to cast integer value of Any type into string type lead to generate a ClassCastException.

```
val obj: Any = 123
val str: String = obj as String
// Throws java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
```

Source and target variable need to nullable for casting to work:

```
fun main(args: Array<String>){
    val obj: String? = "String unsafe cast"
    val str: String? = obj as String? // Works
    println(str)
}
```

Output:

```
String unsafe cast
```

Kotlin Safe cast operator: as?

Kotlin provides a safe cast operator **as?** for safely cast to a type.
It returns a null if casting is not possible rather than throwing an ClassCastException exception.

Let's see an example, trying to cast Any type of string value which is initially known by programmer not by compiler into nullable string and nullable int.
It cast the value if possible or return null instead of throwing exception even casting is not possible.

```
fun main(args: Array<String>){
    val location: Any = "Kotlin"
    val safeString: String? = location as? String
    val safeInt: Int? = location as? Int
    println(safeString)
    println(safeInt)
}
```

Output:

```
Kotlin
null
```

Elvis Operator (?:)

Elvis operator **(?:)** is used to return the not null value even the conditional expression is null.

It is also used to check the null safety of values.

In some cases, we can declare a variable which can hold a null reference.

Suppose that a variable **str** which contains null reference, before using **str** in program we will check it nullability.

If variable **str** found as not null then its property will use otherwise use some other non-null value.

```
var str: String? = null
var str2: String? = "May be declare nullable string"
```

In above code, the String **str** contains a null value, before accessing the value of **str** we need to perform safety check, whether string contain value or not.
In conventional method we perform this safety check using ***if ... else*** statement.

```
var len1: Int = if (str != null) str.length else -1
var len2:  Int = if (str2 != null) str.length else -1
```

```
fun main(args: Array<String>){
    var str: String? = null
    var str2: String? = "May be declare nullable string"
    var len1:  Int = if (str != null) str.length else -1
    var len2:  Int = if (str2 != null) str2.length else
    -1
    println("Length of str is ${len1}")
    println("Length of str2 is ${len2}")
}
```

Output:

```
Length of str is -1
Length of str2 is 30
```

Kotlin provides advance operator known as Elvis operator(?:) which return the not null value even the conditional expression is null. The above if . . . else operator can be expressed using Elvis operator as bellow:

```
var len1:  Int = str?.length ?: -1
var len2:  Int = str2?.length ?:  -1
```

Elvis operator returns expression left to ?: i.e -1. (str?.length) if it is not null otherwise it returns expression right to (?:)i.e(-1).
The expression right side of Elvis operator evaluated only if the left side returns null.

Kotlin Elvis Operator example

```
fun main(args: Array<String>){
     var str: String? = null
     var str2: String? = "May be declare nullable string"
     var len1:  Int = str ?.length ?: -1
     var len2:  Int = str2 ?.length ?:  -1
     println("Length of str is ${len1}")
     println("Length of str2 is ${len2}")
}
```

Output:

```
Length of str is -1
Length of str2 is 30
```

As Kotlin throw and return an expression, they can also be used on the right side of the Elvis operator. This can be used for checking functional arguments:

```
fun functionName(node: Node): String? {
     val parent = node.getParent() ?: return null
     val name = node.getName() ?: throw
          IllegalArgumentException("name expected")
      // ...
}
```

Kotlin Elvis Operator using throw and return expression

```
fun main(args: Array<String>){
     val fruitName: String = fruits()
     println(fruitName)
}
fun fruits(): String{
     val str: String? ="abc"
     val strLength: Int = if(str!= null) str.length else
     -1
     val strLength2: Int = str?.length ?: -1
     var string = "str = $str\n"+
                  "strLength = $strLength\n"+
                  "strLength2 = $strLength2\n\n"

fun check(textOne: String?, textTwo: String?): String?{
val textOne = textOne ?: return null
val textTwo = textTwo ?: IllegalArgumentException("text
exception")

         return "\ntextOne = $textOne\n"+
                 "textTwo = $textTwo\n"
    }
    string += "check(null,\"mango\") = $
{check(null,"mango")}\n" +
            "check(\"apple\",\"orange\") = $
{check("apple","orange")}\n"
    return string
}
```

Output:

```
str = abc
strLength = 3
strLength2 = 3

check(null,"mango") = null
check("apple","orange") =
textOne = apple
textTwo = orange
```

https://www.javatpoint.com/kotlin-unsafe-and-safe-cast-operator