

Mobile Programming



Kotlin Basics – Part III

Null Safety (1/5)

■ NPE (NullPointerException)

- One of the most common pitfalls in many programming languages (e.g., Java!)
- Accessing a member of a null reference will result in a null reference exception

■ Kotlin's type system is aimed at eliminating the danger of null references

- Kotlin type system distinguishes between references that can hold null (nullable references) and those that cannot (non-null references)

Null Safety (2/5)

■ Example

- A regular variable of type String cannot hold null

```
var a: String = "abc" // Regular initialization means non-null by default  
a = null // compilation error
```

```
val l = a.length // it's guaranteed not to cause an NPE
```

- To allow nulls, you can declare a variable as a nullable string by writing String?

```
var b: String? = "abc" // can be set to null  
b = null // ok  
print(b)
```

```
val l = b.length // error: variable 'b' can be null
```

Null Safety (3/5)

- How to handle nullable references, then?
- Solution 1: **Explicit check**

```
val l = if (b != null) b.length else -1
```

- The compiler tracks the information about the check you performed, and allows the call to length inside the if expression
- More complex example

```
val b: String? = "Kotlin"  
if (b != null && b.length > 0) {  
    print("String of length ${b.length}")  
} else {  
    print("Empty string")  
}
```

Null Safety (4/5)

■ Solution 2: Safe calls *for nullable references*

- Second option for accessing a property on a nullable variable is using the safe call operator “?”.

```
val a = "null"
```

```
val b: String? = "string"
```

```
Log.d("ITM", "${b?.length}") b.length or null
```

```
Log.d("ITM", "${a?.length}") // Unnecessary safe call
```

- This returns b.length if b is not null, and **null** otherwise!

Null Safety (5/5)

■ Solution 3: 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 l: Int = if (b != null) b.length else -1
```

equal {

```
val l = b?.length ?: -1
```

safe call elvis

- If the expression to the left of ?: is not null, the Elvis operator returns it, otherwise it returns the expression to the right
- The expression on the right-hand side is evaluated only if the left-hand side is null

■ Solution 4: !! Operator

- **Not-null assertion operator**, however, throws an exception if the value is null

Scope Functions (1/4)

- 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 which you can access the object without its name!
- **Scope functions**
 - let
 - run
 - with
 - apply
 - also

Scope Functions (2/4)

■ Normal function calls

```
val alice = Person("Alice", 20, "Amsterdam")
println(alice)
alice.moveTo("London")
alice.incrementAge()
println(alice)
```

■ Scope function calls

```
Person("Alice", 20, "Amsterdam").let {
    println(it)
    it.moveTo("London")
    it.incrementAge()
    println(it)
}
```


Scope Functions (3/4)

■ Note

- The scope functions do not introduce any new technical capabilities, but they can make your code **more concise and readable**
- Due to the similar nature of scope functions, choosing the right one for your case can be a bit tricky!
- The choice mainly depends on your intent and the consistency of use in your project

Scope Functions (4/4)

Function	Object reference	Return value	Is extension function
let	it	Lambda result	Yes
run	this	Lambda result	Yes
run	-	Lambda result	No: called without the context object
with	this	Lambda result	No: takes the context object as an argument.
apply	this	Context object	Yes
also	it	Context object	Yes

Scope Functions: **let**

객체 결과값 하나에 하나 이상의 함수 호출 시
Null이 아닌값으로만 코드 실행시키고 싶은 때.

■ *let* can be used to invoke one or more functions on results of call chains

➤ Context object reference: *it*

➤ Return value: lambda result

■ Example)

```
val numbers = mutableListOf("one", "two", "three", "four", "five")
val resultList = numbers.map { it.length }.filter { it > 3 }
Log.d("ITM", "$resultList")
```

```
numbers.map { it.length }.filter { it > 3 } let {
    Log.d("ITM", "$it")
    // and more function calls if needed
}
```

Context
it: List<int> ⇒ extension function

Scope Functions: **with** *with(param)*

- Use *with* for calling functions on the context object without providing the lambda result

- Non-extension function
- Context object reference: *this* (can be omitted, when accessing the member)
- Return value: lambda result

■ Example)

```
val numbers = mutableListOf("one", "two", "three")
with(numbers) {
    Log.d("ITM", "'with' is called with argument $this")
    Log.d("ITM", "It contains $size elements")
    Log.d("ITM", "It contains ${this.size} elements") // this can be skipped
}
```

Scope Functions: **run**

객체 초기화 + 함수 반환값 동시 구현.

- Useful when your **lambda contains both the object initialization and the computation of the return value**

- Both extension and non-extension function
- Context object reference: this (can be omitted, when accessing the member)
- Return value: lambda result

■ Example) *a.run()*

- Extension function

```
val service = MultiportService("https://example.kotlinlang.org", 80)

val result = service.run {
    port = 8080
    query(prepareRequest() + " to port $port")
}

// the same code written with let() function:
val letResult = service.let {
    it.port = 8080
    it.query(it.prepareRequest() + " to port ${it.port}")
}
```

Scope Functions: **run**

■ Example)

run (param)

➤ Non-extension function (without context object)

```
val hexNumberRegex = run {  
    val digits = "0-9"  
    val hexDigits = "A-Fa-f"  
    val sign = "+-"  
  
    Regex("[$sign]?[$digits$hexDigits]+") : return value  
}  
  
for (match in hexNumberRegex.findAll("+1234 -FFFF not-a-number")) {  
    Log.d("ITM", match.value)  
}
```

Scope Functions: **apply**

- Use apply for **code blocks that do not return a value** and mainly operate on the members of the receiver object

- Context object reference: **this** (can be omitted, when accessing the member)

- Return value: context object

■ Example)

- The common case for apply is the **object configuration**

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
    }  
}
```

```
    val adam = Person("Adam").apply{  
        age = 32  
        city = "London"  
    }  
    Log.d("ITM", adam.toString())
```

Adam, 32, London

```
data class Person(var name:String, var age:Int=0, var city:String="")
```

Scope Functions: also

- Good for performing some actions that take the context object as an argument

- Context object reference: *it*
- Return value: context object

■ Example)

```
val numbers = mutableListOf("one", "two", "three")
numbers
  .also { Log.d("ITM", "The list elements before adding new one: $it") }
  .apply {
    this.add("four")
    Log.d("ITM", "$this")
  }
```

one two three

one two three four

Scope Functions: Output Distinction (1/2)

■ Context object

- `apply` and `also`
- Can be included into `call chains`
- You can continue chaining function calls on the same object after them

```
val numberList = mutableListOf<Double>()
numberList.also { Log.d("ITM", "Populating the list, length: ${it.size}") } 0
    .apply {
        add(2.71)
        Log.d("ITM", "Sorting the list, length: $size") /
        add(3.14)
        add(1.0)
    }
    .also { Log.d("ITM", "Sorting the list, length: ${it.size}") } 3
    .sort()
    1.0, 2.71, 3.14
```

Scope Functions: Output Distinction (2/2)

■ Lambda result

- let, run, with
- You can use them when assigning the result to a variable

```
val numbers = mutableListOf("one", "two", "three")
val countEndsWithE = numbers.run {
    add("four")
    add("five")
    count { it.endsWith("e") }
}
Log.d("ITM", "There are $countEndsWithE elements that end with e.")
```

one two three four five
one three five

3