Kotlin - Scope Function

JEON YONGTAE
https://github.com/yongtaii/yongapps

1 Scope Function

Scope Function

Scope Function

- 코틀린에서 Scope function을 사용하게 되면, 객체 Context 내의 Block Code를 실행할 수 있게 한다
  In Kotlin, scope functions allow you to execute a function, i.e. a block of code, in the context of an object

- 그러면 해당 객체(Object)는 임시 Scope(범위) 내에서 이름 사용 없이 접근이 가능하다
  The object is then accessible in that temporary scope without using the name

- Scope Function을 사용하면 가독성이 높고, 간결한 코드를 만들 수 있다
  Using them can increase readability and make your code more concise.

- Kotlin 기본 라이브러리에서 제공

Scope Function

Scope Function 사용 안하는 경우

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

- 새로운 변수(alice)를 선언해야 한다
- 해당 변수를 사용하고 싶을 때마다 이름을 반복해서 사용해야 한다

Scope Function

Scope Function 사용하는 경우

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

- Scope Function은 새로운 기술적인 능력(Technical capabilities)을 소개하지는 않지만, 당신의 코드를 간결하고, 가독성이 높게 만들어 준다

2 Distinctions

Distinctions

Distinctions

- Scope Function은 여러 개가 있는데, 매우 유사해서 이들의 차이점을 이해하는 것이 중요하다.
  각 Scope Function은 두 가지 주요 차이점이 있다.
  Because the scope functions are all quite similar in nature, it's important to understand the differences between them. There are two main differences between each scope function:

  - ✓ The way to refer to the context object
  - ✓ The return value.

Scope Function 종료

컨텍스트 객체 참조 방법, Return Value 에 의해 다음과 같이 나누어 진다

	Context Object As Function Argument	Context Object As Function Receiver
Returns: Function Result	`let`	`run`, `with`
Returns: Context Object	`also`	`apply`

컨텍스트 객체 참조 방식 & 함수 별 반환값 별 함수들 특성 요약

범위 함수는 컨텍스트 객체를 함수 인수 또는 함수 수신자로 참조한다

범위 함수 반환 값은 함수 결과 또는 컨텍스트 객체이다

Distinctions

Distinctions

- Scope Function은 여러 개가 있는데, 매우 유사해서 이들의 차이점을 이해하는 것이 중요하다.
  각 Scope Function은 두 가지 주요 차이점이 있다.
  Because the scope functions are all quite similar in nature, it's important to understand the differences between them. There are two main differences between each scope function:

  - ✓ The way to refer to the context object
  - ✓ The return value.

Context Object: this or it

Context Object: this or it

- **Scope Function 의 람다식 안에서 객체를 참조할 때 실제 이름 대신 짧은 reference 사용이 가능하다**
  Inside the lambda of a scope function, the context object is available by a short reference instead of its actual name.

```
val str = "Hello"
// this
str.run {
    println("The receiver string length: $length")
    //println("The receiver string length: ${this.length}") // does the same
}

// it
str.let {
    println("The receiver string's length is ${it.length}")
}
```

각 Scope Function은 전달 된 객체에 접근하기 위해 둘 중 하나의 방법 을 쓴다

- ✓ this (as a lamda receiver)
- ✓ it (as a lamda argument)

this (as a lamda receiver)

this

- run(), with(), apply() 함수는 this 키워드를 사용해 블록내로 전달 된 객체 참조가 가능하다
  run, with, and apply refer to the context object as a lambda receiver – by keyword this
- 따라서 람다 객체는 일반 클래스 함수에서와 같이 사용할 수 있다
  Hence, in their lambdas, the object is available as it would be in ordinary class functions.

```
val adam = Person("Adam").apply {
    // same as this.age = 20 or adam.age = 20
    age = 20
    city = "London"
}
println(adam)
```

it (as a lamda argument)

this

- let() also() 함수는 람다 argument로써 객체를 갖는다
  In turn, let and also have the context object as a lambda argument
- argument 이름을 특별히 지정하지 않았으면, 블록으로 전달 된 객체는 디폴트 이름인 'it'으로 사용이 가능하다
  If the argument name is not specified, the object is accessed by the implicit default name 'it'

```
fun getRandomInt(): Int {
    return Random.nextInt(100).also {
        writeToLog("getRandomInt() generated value $it")
    }
}

val i = getRandomInt()
```

it (as a lamda argument)

this

- **argument로 객체를 전달할 때, 'it' 대신 커스텀 이름을 지정할 수 있다**
  Additionally, when you pass the context object as an argument, you can provide a custom name for the context object inside the scope.

```
fun getRandomInt(): Int {
    return Random.nextInt(100).also { value ->
        writeToLog("getRandomInt() generated value $value")
    }
}

val i = getRandomInt()
```

Return value

Distinctions

- **Scope Function은 또한 Return 값에 의해 함수들이 구분된다**
  The scope functions differ by the result they return:

  - ✓ **apply() and also() return the context object.**
  - ✓ **let(), run(), and with() return the lambda result.**

- **두개의 옵션은 다음코드에서 어떤 동작을 수행할지에 고려해서 선택하면 된다**
  These two options let you choose the proper function depending on what you do next in your code.

Return value - Context Object

Return value – Context Object

- **apply(), also() 함수의 리턴 값은 객체 자신이다**
  The return value of apply and also is the context object itself.
- **함수 호출은 이후 동일한 객체에서 계속 함수 호출을 연결할 수 있다.**
  Hence, they can be included into call chains as side steps: you can continue chaining function calls on the same object after them.

```
val numberList = mutableListOf<Double>()
numberList.also { println("Populating the list") }
    .apply {
        add(2.71)
        add(3.14)
        add(1.0)
    }
    .also { println("Sorting the list") }
    .sort()
```

Return value - Lambda result

Return value – Lambda result

- **let(), run(), with() 는 람다식 결과값을 리턴한다. 따라서 결과값을 변수나 다음동작에 할당할 수 있다.**
  let, run, and with return the lambda result. So, you can use them when assigning the result to a variable, chaining operations on the result, and so on.

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

Return value - Lambda result

Return value – Lambda reult

- **결과 값을 무시하고 사용할 수 도 있다**
  Additionally, you can ignore the return value and use a scope function to create a temporary scope for variables.

```
val numbers = mutableListOf("one", "two", "three")
with(numbers) {
    val firstItem = first()
    val lastItem = last()
    println("First item: $firstItem, last item: $lastItem")
}
```

3 Functions

Fuctions

Functions

- **상황에 맞는 Scope Function을 선택하기 위해 각 함수에 대한 세부적인 내용을 설명할 것이다**
  To help you choose the right scope function for your case, we'll describe them in detail and provide usage recommendations.

  - ✓ apply()
  - ✓ also()
  - ✓ let()
  - ✓ run()
  - ✓ with()

let()

let()

- **let() 함수는 호출한 객체를 이어지는 블록함수의 argument로 전달하고, 람다식 결과값(block 결과 값) 을 reuturn 한다**
  The context object is available as an argument (it). The return value is the lambda result.

let()

let()

- let() 함수를 이용하면 불필요한 선언을 방지할 수 있다

```
//단말기 환경에 맞게 패딩 값을 계산한다.
val padding = TypedValue.applyDimension(
    TypedValue.COMPLEX_UNIT_DIP, 16f,
resources.displayMetrics).toInt()

//패딩 값을 설정한다.
button1.setPadding(padding, 0, padding, 0)
```

```
TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP,
16f, resources.displayMetrics).toInt().let{
    //계산된 값을 인자로 받으므로, 함수에 바로 대입할 수 있다.
    button1.setPadding(it, 0, it, 0)
}
```

let()

let()

- **let()함수는 nun-null value 만 함수실행이 가능하게 할 수 있다**
  let is often used for executing a code block only with non-null values.

```
val x: Int? = null

// null-safe transformation without let
    val y1 = if (x != null) x + 1 else null
    val y2 = if (y1 != null) y1 / 2 else null

// null-safe transformation with let
    val z1 = x?.let { it + 1 }
    val z2 = z1?.let { it / 2 }
```

let()

let()

- let()를 이용해 블록의 람다식 결과 값 반환이 가능하다

```
val result = person?.let{
    printPerson(it.lastName + it.firstName)
}

Log.d("yong","result : $result")
```

apply()

apply()

- **이 함수를 호출하는 객체를 이어지는 함수형 인자 block의 리시버로 전달하며, 함수를 호출한 객체를 반환한다**
  The context object is available as a receiver (this). The return value is the object itself.

apply()

apply()

- 일반적으로 객체를 할당할때 쓰인다
  The common case for apply is the object configuration.

```
class PersonBean {
    var firstName: String? = null
    var lastName: String? = null
}
```

```
// Initialization the traditional way
val p1 = PersonBean()
p1.firstName = "Frank"
p1.lastName = "Rosner"

// Initialization using apply
val p2 = PersonBean().apply{
    firstName = "Frank"
    lastName = "Rosner"
}
```

with()

with()

- 인자로 받은 객체를 이어지는 함수형 인자 block의 arguement전달하며, block 함수의 결과를 반환한다.
  A non-extension function: the context object is passed as an argument, but inside the lambda, it's available as a receiver (this). The return value is the lambda result.

with()

with()

- 람다 결과 제공이 필요하지 않는 곳에 쓰이는 것을 추천한다.
  We recommend with for calling functions on the context object without providing the lambda result. In the code, with can be read as “with this object, do the following.”

```
val numbers = mutableListOf("one", "two", "three")
with(numbers) {
    println("'with' is called with argument $this")
    println("It contains $size elements")
}
```

- 객체의 값을 계산할때 사용
  Another use case for with is introducing a helper object whose properties or functions will be used for calculating a value.

```
val result3 = with("text") {
    val tail = substring(1)
    tail.toUpperCase()
}
```

run()

run()

- run()함수는 인자가 없는 익명 함수처럼 사용하는 형태와 객체에서 호출하는 형태를 제공한다
  The context object is available as a receiver (this). The return value is the lambda result.

  - ✓ 익명함수처럼 이용하는 경우 : 함수형 인자 block을 호출하고 그 결과를 반환한다
  - ✓ 객체에서 호출하는 형태 : 이 함수를 호출한 함수형 인자 block의 리시버로 전달하고 그 결과를 반환한다

run() example

객체에서 호출하는 형태

```
// compute result with receiver
val result2 = "text".run {
    val tail = substring(1)
    tail.toUpperCase()
}
```

run() example

익명함수처럼 호출하는 형태

```
// compute result as block result
val result = run {
    val x = 5
    val y = x + 3
    y - 4
}
```

복잡한 계산을 위해 여러 임수 변수가 필요한 경우 유용하게 사용할 수 있다

run() 함수 내부에서 선언되는 변수들은 블록 외부에 노출되지 않으므로 변수 선언 영역을 확실히 분리할 수 있다

also()

also()

- also()함수는 객체를 인자로 받은 객체를 이어지는 함수 블록에 전달하고, 리턴값은 자기자신이다
- also()함수는 객체를 변경하지 않는 상태에서 객체를 이용한 액션을 취할때 유용하다. 예를 들어 로그를 남긴다던지…

also() example

랜덤값 반환 함수

```
fun getRandomInt(): Int {
    return kotlin.random.Random.nextInt(100).also {
        writeToLog("getRandomInt() generated value $it")
    }
}
```

Function Selection

Function Selection

- 올바른 함수 선택을 위한 테이블
  To help you choose the right scope function for your purpose, we provide the table of key differences between them.

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

Function Selection

Function Selection

- **목적에 따른 올바른 함수 선택을 위한 가이드**
  Here is a short guide for choosing scope functions depending on the intended purpose

  - ✓ Executing a lambda on non-null objects: let
  - ✓ Introducing an expression as a variable in local scope: let
  - ✓ Object configuration: apply
  - ✓ Object configuration and computing the result: run
  - ✓ Running statements where an expression is required: non-extension run
  - ✓ Additional effects: also
  - ✓ Grouping function calls on an object: with

Thanks!

Any questions ?

You can find me at

- jeonyt89@gmail.com