세 가지 Scope Function의 동작과 Source code 예시

> Scope function은 함수를 lambda expression으로 작성할 때 code block 안의 해당 scope에서 context object를 꼭 이름을 사용하지 않고도 `this` 또는 `it`으로 접근할 수 있는 함수들을 의미하며, 크게 두 가지의 기준으로 차이와 그 쓰임을 구분할 수 있다. 첫 번째로 context object를 code block에서 `it`으로 접근하는지 아니면 `this`로 하는지에 따라서 달라지며, 두 번째로는 그 함수가 return하는 value가 context object인지 아니면 lambda result(code block 안에서 맨 마지막에 존재하는 expression)인지에 따라 다르다.

`also`

```
1  public inline fun <T> T.also(block: (T) -> Unit): T {
2      block(this)
3      return this
4  }
```

`also`는 `T` type의 context object의 extension function으로 정의되며, 인자로 넘기는 함수 `block`에서 인자로 context object를 넘기고 context object를 return value로 갖는다. 즉, `also`의 함수 인자로 lambda를 받을 때 receiver type으로 받지 않아서 context object를 접근할 때 `it`으로 접근할 수 있다. 먼저 `block`으로 받은 함수를 실행하는데, `block` 실행 시 위의 코드에서 generic으로 표현된 `T` type의 context object를 `it`으로 접근할 수 있다. 그리고 그 함수의 실행 결과가 반환되는 것이 아니라 `block`이 수행된 후의 반영된 그 context object 자체가 반환된다. `apply`와 유사하지만, `apply`는 함수를 receiver type으로 받아서 context object 접근 시 `this`를 사용한다는 점에서 차이가 있다.

```
1  val numbers = mutableListOf("one", "two", "three")
2  numbers.also { println("The list elements before adding new one: $it") }.add("four")
```

위의 예시 코드에서는 `numbers`를 context object로 받고, `also`에서 `println`라는 함수를 인자로 받는다. `println`을 수행할 때 `numbers`에 접근하기 위해 `it`을 사용하며, `println`의 수행 결과가 아니라 context object인 `numbers` 그 자체가 반환되므로 그 이후에 `MutableList` object의 메소드인 `add`를 실행시킬 수 있다. 실행 결과는 다음과 같다. 만약 `println`의 return value가 반환되었으면 아래와 같은 결과가 나올 수 없으므로 `also` scope function은 context object를 반환한다는 사실을 짐작할 수 있다.

```
The list elements before adding new one: [one, two, three]
res0: kotlin.Boolean = true
```

`with`

```
public inline fun <T, R> with(receiver: T, block: T.() -> R): R {
    return receiver.block()
}
```

`with`는 `receiver`와 `block`라는 두 개의 인자를 받는데, 여기서 `receiver`는 receiver type의 `block` 함수에서 receiver로 접근하기 위한 context object이다. 즉, `run`과는 달리 context object가 lambda expression과 함께 인자로 들어간다. Receiver type으로 `block`을 받으므로 context object에 접근할 때는 `this`를 사용한다. `block` 함수를 다 수행하고 나서 그 receiver를 반환하는 것이 아니라 그 `block` 함수의 return value를 그대로 반환한다.

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

첫 번째 예시 코드에서 context object인 `numbers`가 lambda expression과 함께 `with`의 인자로 들어간다. `block`에 대응되는 두 개의 `println` 함수가 순차적으로 수행되는데, context object인 `numbers`에 접근할 때 `this`를 사용한다. 두 번째 `println`에서는 `size`만 존재하는데, 이는 `this.size`에서 `this`가 생략된 것이다. 실행 결과는 아래와 같다.

```
The first element is one, the last element is three
```

```
val numbers = mutableListOf("one", "two", "three")
val firstAndLast = with(numbers) {
    "The first element is ${first()}," +
    " the last element is ${last()}"
}
println(firstAndLast)
```

두 번째 예시 코드에서는 with의 인자로 context object와 함께 expression이 들어가는데, 이 expression에서 `first`와 `last`는 `MutableList` 객체의 메소드이다. `with`은 expression의 반환값을 return 한다고 했으므로, 맨 마지막 줄에서 `println` 수행 시 인자로 들어간 `firstAndLast`의 반환값인 expression이 출력된다. 수행 결과는 위의 첫 번째 예시 코드와 동일하다.

`run`

```
public inline fun <R> run(block: () -> R): R {
    return block()
}
public inline fun <T, R> T.run(block: T.() -> R): R {
    return block()
}
```

`run`은 context object가 없이 lambda expression만을 인자로 받는 것과 context object의 extension function으로 정의하여 인자로 받는 함수를 context object에 관한 receiver type으로 받는 것의 두 개가 구현되어 있다. 첫 번째는 단순히 인자로 넘어 온 `block` lambda expression을 그대로 수행한 return value를 반환하고, 두 번째는 context object의 메소드인 `block` 함수의 return value를 반환한다는 차이가 있다. 후자는 context object를 T로 받으므로 context object에 접근할 때 `this` keyword로 접근해서 사용한다. 대신에 위의 코드에서는 `this.block()`에서 `this`가 생략되어 작성된 것으로 볼 수 있다. `with`와의 차이점은 바로 context object에 관해 extension function으로 정의되느냐 아니면 context object 그 자체를 scope function의 직접 인자로 넘기냐의 차이다.

```
val hexNumberRegex = run {
    val digits = "0-9"
    val hexDigits = "A-Fa-f"
    val sign = "+-"
    Regex("[$sign]?[$digits$hexDigits]+")
}
for (match in hexNumberRegex.findAll("+123 -FFFF !%*& 88 XYZ")) {
    println(match.value)
}
```

첫 번째 예시 코드에서는 context object가 extension function으로 정의되지 않은 함수이다. Lambda expression 안에서 immutable type의 변수를 선언하고, 이 `run` scope function을 수행 시 정규식에 관한 `Regex` 함수의 return 값이 반환된다. 그래서 아래 `for` loop에서 `hexNumberRegex`의 값이 바로 앞서 `Regex`의 반환값이 되고, 이를 가지고 `findAll`를 수행하여 그 인자로 넘긴 string에 관해 match 되는 부분을 찾아서 출력한다. 실행 결과는 다음과 같다.

```
+123
-FFFF
88
```

```
val service = MultiportService("https://example.kotlinlang.org", 80)
val result = service.run {
    port = 8080
    query(prepareRequest() + " to port $port")
}
```

두 번째 예시 코드에서는 `MultiportService` object의 context인 `service`에 관해 `run` function을 수행하는 것이다. Lambda expression에서 불변 type의 변수를 선언하고 `query` 함수를 실행했을 때의 return 값이 최종적으로 반환되는데, `this.prepareRequest()`와 `this.port`를 통해 context object를 접근하여 가져온 데이터를 query의 인자로 넘기는 과정이 구현되어 있다. 실행 결과는 다음과 같다.

```
Result for query 'Default request to port 8080'
```