

06. More Language Constructs presentation

Exception

- 메소드 본문 채우기

```
fun readNumber(reader: BufferedReader) {  
    // ...  
    println(number)  
}
```

1. reader.readLine() 으로 읽어온 String 을 Integer 로 변환하여 콘솔에 출력하자
2. 읽어온 문자열 값에 따라 NumberFormatException 이 발생할 수도 있다. 예외를 처리하기 위해 try-catch 를 사용하자
3. (1) 예외가 발생한 경우 println 명령이 처리되지 않게 하거나 (2) null 을 출력하게끔 하자

▼ 답

```
fun readNumber(reader: BufferedReader) {  
    val number = try {  
        Integer.parseInt(reader.readLine())  
    } catch (e: NumberFormatException) {  
        return // null 로 수정할 경우 Exception 발생 시 null을 반환하게 된다.  
    }  
    println(number)  
}
```

Scope functions 비교

| Aa 이름 | ▼ 컨텍스트 객체 | ▼ 리턴 값 | ☰ 확장함수인가? |
|-------------|-----------|----------|-----------|
| <u>let</u> | 람다 인자(it) | 람다 결과 | Yes |
| <u>also</u> | 람다 인자(it) | 객체 자기 자신 | Yes |

| Aa 이름 | ▼ 컨텍스트 객체 | ▼ 리턴 값 | ☰ 확장함수인가? |
|--------------|--------------|----------|-----------|
| <u>run</u> | 람다 수신자(this) | 람다 결과 | Yes |
| <u>run</u> | | 람다 결과 | No |
| <u>with</u> | 람다 수신자(this) | 람다 결과 | No |
| <u>apply</u> | 람다 수신자(this) | 객체 자기 자신 | Yes |

▼ 널이 아닌 객체에 람다를 실행할 때:

let

▼ 지역(local) scope에서 표현식(expression)을 변수로 선언할 때:

let

▼ 객체 선언?(configuration)

apply

▼ 객체 선언?(configuration)과 결과를 계산할 때

run

▼ 표현식(expression)이 필요한 곳에 문(statements)을 실행할 때: 비 확장(non-extension)

run

▼ 부가적인 실행

also

▼ 객체에 대한 그룹 함수 호출

with

```
val str: String? = "Hello"
// processNonNullString(str)           // 컴파일 에러: str can be null

val length = str?.[____] {
    println("let() called on $it")
    processNonNullString(it)           // OK: '?.[____] { }' 안에서는 'it'이 null이 아님
    it.length
}
```

▼ 답

let

let은 종종 널이 아닌 값만을 가지는 코드 블록을 실행시키는 데에 사용됩니다. 널이 아닌 객체에 동작을 수행하기 위해서는, 해당 객체에 안전한 호출 연산자(safe call operator)인 ?.을 사용하고 람다의 action으로 let을 호출하면 됩니다.

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

▼ 답

with

with는 람다 결과 없이 컨텍스트 객체의 호출 함수에서 사용하는 것을 권장합니다. 아래 코드에서 with는 “이 객체로, 다음을 수행하라(with this object, do the following).”로 읽힐 수 있습니다

```
val numbers = mutableListOf("oen", "two", "three")
val firstAndLast = [____](numbers) {
    "The first element is ${first()}, " +
    " the last element is ${last()}"
}
println(firstAndLast)
```

▼ 답

with

with의 또 다른 사용법은 값을 계산하는 데에 사용되는 헬퍼 객체(helper object)의 프로퍼티나 함수를 선언하는 데에 사용하는 것입니다.

```
val adam = Person("Adam").[____] {
    age = 32
    city = "London"
}
```

▼ 답

apply

apply는 값을 반환하지 않고, 주로 수신 객체의 멤버를 연산하는 곳에 사용하면 됩니다. apply를 사용하는 가장 일반적인 경우는 객체 생성(configuration)입니다. 이러한 호출은 “다음의 지시를 객체에 적용하라(apply the following assignments to the object).” 로 읽힐 수 있습니다.

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

val result = service.[__] {
    port = 8080
    query(prepareRequest() + " to port $port")
}
```

▼ 답

run은 with와 같은 역할(이 객체로 다음을 수행하라)을 하지만, let처럼-컨텍스트 객체의 확장 함수처럼- 호출(involve)됩니다.

```
val hexNumberRegex = [__] {
    val digits = "0-9"
    val hexDigits = "A-Fa-f"
    val sign = "+-"

    Regex("[$sign]?[$digits$hexDigits]+")
}

for (match in hexNumberRegex.findAll("+1234 -FFFF not-a-number")) {
    println(match.value)
}
```

▼ 답

비확장(non-extension) run은 표현식이 필요한 곳에서 다수의 구문 블록을 실행할 수 있도록 합니다.

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

▼ 답

also


`also` 는 컨텍스트 객체를 인자로 가지는 것과 같은 작업을 수행하는 데에 좋습니다. `also` 는 로깅이나 디버그 정보를 출력하는 것과 같이 객체를 변화시키지 않는 부가적인 작업에 적합합니다. 대부분 프로그램의 로직을 파괴하지 않고도 호출 체인에서 `also` 의 호출을 제거하는 것이 가능합니다.

`also` 를 코드에서 보면, “**그리고 또한 다음을 수행하라(and also do the following)**”의 의미를 읽을 수 있습니다.

참고자료:

[Kotlin] 코틀린의 Scope 함수

코틀린 표준 라이브러리는 객체의 컨텍스트 내에서 코드 블록을 실행하기 위한 목적만을 가진 여러가지 함수를 제공합니다. 이런 함수들을 람다식으로 호출할 때, 이는 임시로 범위(scope)를 형성합니다. 이 범위

 <https://shinjekim.github.io/kotlin/2019/12/05/Kotlin-%EC%BD%94%ED%8B%80%EB%A6%B0%EC%9D%98-Scope-%ED%95%A8%EC%88%98/>

