Kotlin

- 기본적으로 Java를 다뤄봤다는 전제하에 실용적인 측면으로만 기술.

Language, Scala, JVM, Object Oriented Programming, Functional Programming

기본

```
// data를 붙이면 toString(), equals(), hashCode(), copy() Method 생성
// 기본적으로 모든 변수는 null을 허용하지 않기 때문에, C#처럼 "Nullable" type이 존재.
// val은 Scala처럼 Immutable, var는 Mutable.
data class Person(val name: String, val age: Int? = null)

// 함수 내용이 한 줄인 경우 "="를 이용하여 식(Expression) 표현이 가능하다.
// it은 Scala나 Clojure의 "_"와 같은 의미. 즉, 함수 안에서 자기 자신을 가리키며 "x -> x.age"에서 x를 it으로 축약 표현.
//"?:"은 Elvis Operator라고 하며, 뒤의 값은 null인 경우 0으로 처리하겠다는 의미.
fun getOldest(persons: List<Person>) = persons.maxBy { it.age ?: 0}

// main 함수를 작성하는데 class가 필요하지 않다.
fun main(args: Array<String>) {
    val person1 = Person("john", 11)
    val person2 = Person("doe") //Nullable로 선언했기에 age는 생략 가능
    val person3 = Person(age = 22, name = "Jake") //직접 Property 이름으로 값 삽입
    //list 생성. 모든 Collection은 ...Of로 생성 가능.
    val persons = listOf(person1, person2, person3)
    val oldest = getOoldest(persons)

  // 문자열 Template 지원. 식(Expression)이 아닌 경우 {} 생략 가능.
    println("oldest : ${oldest}")
}
```

```
oldest : Person(name=Jake, age=22)
```

High-Order Functions 시험

https://kotlinlang.org/docs/reference/lambdas.html

아래 예제처럼 Kotlin은 함수를 first-class로 취급하므로 Functional Programming의 기본 특성을 만족한다. (함수를 인자로 받고, 함수를 반환할 수 있는 특성)

```
// Type만을 지정하기 위한 빈 Interface 생성. 상속은 ":" 사용.
```

```
interface Event
data class KeyEvent(val value: String): Event
fun onKeyDown(f: (Event) -> Unit) = f(KeyEvent("keydown event"))

fun main(args: Array<String>) {
    // onKeyDown { event -> println(event) } 와 동치.
    onKeyDown { println(it) }
}
```

```
KeyEvent(value=Hello, World)
```

Control Flow

https://kotlinlang.org/docs/reference/control-flow.html

if, while/do while은 동일하므로 생략.

when

```
interface Event
data class KeyEvent(val value: String): Event
data class MouseEvent(val value: String): Event

fun main(args: Array<String>) {
    val eventControlFlow = {
        event: Event ->
        when (event) {
            is KeyEvent -> println("Key event! ${event.value}")
            is MouseEvent -> println("Mouse event! ${event.value}")
            else -> println("Unknown")
        }
    }

  // 인자가 없는 형태로도 사용 가능. 이 경우 Matching 대상이 계속 Instance를 생성하는 경우 사용.
    /* val eventControlFlow = {
        event: Event ->
        when {
            event is KeyEvent -> println("Key event! ${event.value}")
            event is MouseEvent -> println("Mouse event! ${event.value}")
            else -> println("Unknown")
        }
    } */

    eventControlFlow(KeyEvent("foo"))
```

```
    eventControlFlow(MouseEvent("bar"))
}
```

```
Key event! foo
Mouse event! bar
```

for

foreach만 존재.

```
fun main(args: Array<String>) {

    // in은 infix가 붙은 함수로 중위표현식 가능.
    // in은 Collections 혹은 Range의 원소 검사.
    val test = "Banana" in "Apple".."Cherry"
    println(test)

    for (i in 1..10) {
        print("$i ")
    }

    println()
    for (i in 10 downTo 1) {
        print("$i ")
    }

    println()

    for (i in 10 downTo 1 step 2) {
        print("$i ")
    }

    println()

    val size = 10
    for (i in 0 until size) {
        print("$i ")
    }
}
```

```
true
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
10 8 6 4 2
```

```
0 1 2 3 4 5 6 7 8 9
```

Destructuring

```
data class Person(val name: String, val age: Int? = null)

fun main(args: Array<String>) {
    val l = arrayListOf<Person>()
    l.add(Person("Jake", 11))
    l.add(Person("Luke", 23))
    l.add(Person("Nina", 21))

    // Destructuring
    for((age, name) in l) {
        println("$name $age")
    }

    println()

    for((index, element) in l.withIndex()) {
        // Kotlin은 element에 대해서 Nested Destructuring 을 지원하지 않기 때문에 변수를 이용하여 Destructuring.
        val (name, age) = element
        println("$index $name $age")
    }
}
```

```
11 Jake
23 Luke
21 Nina
0 Jake 11
1 Luke 23
2 Nina 21
```