2023 SPRING 캡스톤디자인 중간고사 리포트

20171665 이선호

1. SEQUENCES

Kotlin Sequence는 iterator를 통해 각 element에 순차적으로 operation을 수행할 수 있는 iterable type 중 하나로 terminal operation을 통해서만 evaluation이 시작되는 lazy 속성을 가진다.

```
public interface Sequence<out T> {
	public operator fun iterator(): Iterator<T>
}
```

A. Sequence interface의 extension function으로 구현된 map function과 first function의 Standard Library 소스코드를 예시하여 intermediate operation과 terminal operation 각각의 동작을 설명하시오.

Kotlin에서는 expression의 evaluation 시점에 따라서 크게 eager operation과 lazy operation으로 나눌 수 있다. Eager operation은 filter 등 operation을 적용한 즉시 그 연산이 실행되어서 element에 하나씩 적용하여 바로 새로운 list로 생성하는 방법이고, lazy operation은 runtime에서 필요한 순간에 한해서 연산이 실행되는 방법이다. Kotlin에서는 일반적인 eager operation을 지원하는 collection 뿐만이 아니라 lazy operation도 지원하는 collection을 구현했는데, lazy operation이 실행되는 collection을 sequence라고 부른다. 즉, Sequence는 filter가 바로 적용되는 것이 아니라 runtime에 필요한 순간에 적용되는 lazy operation을 수행할 수 있는 자료 구조이다.

이러한 Sequence 내에서의 lazy operation을 intermediate operation로 부른다. Sequence interface의 intermediate operation은 연산을 적용한 결과를 구해서 바로 반환하는 것이 아니라 runtime에서 필요한 경우 terminal operation의 호출에 의해서 연산이 수행된다. 그래서 `filter`, `map` 처럼 intermediate operation을 적용하면 그 결과가 또 다른 sequence로 반환된다. 대신 `filter`, `map`과 같은 operation 연산은 나중에 필요한 순간에 수행할 수 있도록 그러한 함수를 수행 가능한 새로운 Sequence를 결과로 낸다.

```
public fun <T, R> Sequence<T>.map(transform: (T) -> R): Sequence<R> {
    return TransformingSequence(this, transform)
}
```

위의 코드는 Kotlin Standard Library의 GitHub repository의 `_Sequences.kt` 파일에서 확인할 수 있는 Sequence interface의 `map`을 구현한 것이다. 일반적으로 eager operation으로 `map`을 구현하면 parameter로 넘겨 받은 함수를 적용한 결과를 반환해야 하지만, 여기서는 intermediate operation으로 구현하기 위해 `TransformingSequence`라는 새로운 Sequence를 반환한다. 이 Sequence는 `toList`, `first` 등 terminal operation이 호출되는 순간에 모든 element를 순차적으로 돌면서 앞서 intermediate operation에 적용한 것과 함께 evaluation을 수행한다.

Sequence의 terminal operation은 앞서 설명한 eager operation에 속한다. 앞서 intermediate operation에 의해 각 연산마다 생성된 Sequence는 terminal operation이 호출될 때 모든 element를 하나씩 순회하면서 각 element 마다 앞서 value로 저장했던 함수들을 하나씩 수행한다.

```
public inline fun <T> Sequence<T>.first(predicate: (T) -> Boolean): T {
    for (element in this) if (predicate(element)) return element
    throw NoSuchElementException("Sequence contains no element matching the predicate.")
}
```

위의 코드는 마찬가지로 같은 파일에서 확인할 수 있는 Sequence interface의 `first`을 구현한 것이다. 일반적인 eager operation처럼 sequence의 element를 하나씩 탐색하면서 유효한 첫 번째 element를 찾아서 바로 해당 element를 반환한다.

B. Sequence의 lazy evaluation 구현 방식을 구체적으로 기술하시오.

Sequence의 lazy evaluation은 기본적으로 intermediate operation에서 먼저 해당 연산을 포함하는 새로운 Sequence를 반환하고, 이후 terminal operation을 수행할 때 Sequence의 모든 element를 하나씩 순회하면서 앞서 저장한 연산을 순차적으로 적용하게 된다. 이때, terminal operation 적용 시 Sequence도 iterable 객체처럼 각 element 마다 하나씩 접근하여 처리해야 하므로 `iterator` 함수를 구현하고 있다.

구체적으로 intermediate operation을 적용할 때 해당 연산을 수행할 수 있는 Sequence를 반환하는데, 이때 각 연산마다 새로 생성되는 Sequence에서 `iterator` 함수를 어떻게 overriding 하고 있는지가 다르다. 예를 들어, `map` 함수는 새로 생성되어 반환되는Sequence에서 `iterator` 함수의 `next` 함수를 overriding 하고 있는데, 이 `next` 함수에서 앞서 `map`의 인자로 받은 `transform` 함수를 그대로 적용한 새로운 Sequence를 생성하는 방식으로 구현되어 있다. 즉, intermediate operation만 호출하면 그러한 연산을 `iterator` 함수에서 overriding 하고 있는 새로운 Sequence 객체만 반환할 뿐, 각 element에 관한 어떠한 계산도 수행되지 않는다.

그러나 terminal operation을 호출하면 앞서 overriding 하고 있는 `iterator` 함수가 수행되면서 Sequence의 모든 element를 하나씩 탐색하게 되고, 이때 `iterator`의 `next` 등 특정 메소드에서 그 연산에 맞게 새로 overriding한 방식에 따라서 element 계산이 수행되어 반환된다. 즉, 새로운 Sequence에서 기존 Sequence의 원소들을 `iterator`를 통해 탐색하면서 앞서 지정한 operation을 적용한다. 그 계산은 해당 객체에 적용한 intermediate operation부터 순서에 맞게 적용된다. 예를 들어, 앞서 intermediate operation으로 어떤 list에 `filter`, `map`을 적용하고 terminal operation로 `first`를 호출했을 때 `filter`, `map`, `first` 함수 연산이 순서대로 각 element 마다 수행된다.

이처럼 매 함수마다 직접 연산 결과를 매번 수행하여 데이터를 저장하지 않고 나중에 필요할 때 어떠한 방식으로 연산을 적용하여 반환할지를 지정하는 lazy evaluation 방식은 메모리 관리 면에서 이점이 있다.

2. GENERIC

A. Kotlin Standard Library Collection type 중 generic interface List의 소스코드를 분석하여 Type parameter variance를 확인하고 그 의미를 설명하시오.

```
public interface List<out E> : Collection<E> {
    // Query Operations
    override val size: Int
    override fun isEmpty(): Boolean
    override fun contains(element: @UnsafeVariance E): Boolean
    override fun iterator(): Iterator<E>

    // Bulk Operations
    override fun containsAll(elements: Collection<@UnsafeVariance E>): Boolean

    // Positional Access Operations
    public operator fun get(index: Int): E

    // Search Operations
    public fun indexOf(element: @UnsafeVariance E): Int
    public fun lastIndexOf(element: @UnsafeVariance E): Int

    // List Iterators
    public fun listIterator(): ListIterator<E>
    public fun listIterator(index: Int): ListIterator<E>

    // View
    public fun subList(fromIndex: Int, toIndex: Int): List<E>
}
```

위의 코드는 Kotlin GitHub 레포지토리의 `List` interface 부분이다. List interface에서는 `out` 키워드에 의해 interface 내부에서 구현되는 함수들의 return type으로 `E`가 올 수 있음을 의미하며, subtyping 관계가 그대로 유지되는 covariance를 만족한다. 즉, 어떤 interface를 구현하고 있는 변수의 type보다 subtype으로 선언된 변수를 할당할 수 있다는 의미이며, 이는 `out` 키워드에 의해 해당 generic interface가 parameter type으로서 T를 그 interface 내부의 메소드에서만 사용하고 있다는 제약이 붙어서다. 그래서 위의 코드에서도 `override fun iterator(): Iterator<E>`, `public operator fun get(index: Int): E`, `public fun`

`listIterator(): ListIterator<E>`, `public fun listIterator(index: Int): ListIterator<E>`, `public fun subList(fromIndex: Int, toIndex: Int): List<E>`에서 type parameter에 관한 covariance를 확인할 수 있다. 이 함수들은 모두 `E` type을 반환하도록 선언되었으며, `E`의 subtype을 반환하는 것도 가능하므로 어떠한 변수를 선언할 때 generic으로 `E`에 subtype을 넣는 것이 가능해진다.

이는 `List` interface 내 일부 method의 반환 값의 type이 `E` 또는 `E`의 subtype이기를 기대하는 것인데, 이는 `E`보다 더 큰 super type이 메소드에서 반환되는 것을 예방하고 `List`의 일부 method에서 반환하는 값이 최소한 `E` type의 모든 속성을 만족하도록 하기 위한 조건을 부여하여 type 안전성 문제를 방지할 수 있다.

B. List의 method 중 선언된 variance에 위배되는 function의 사례 2~3개를 제시하고, compile time error 회피를 위하여 사용된 방법을 설명하시오.

```
override fun contains(element: @UnsafeVariance E): Boolean
```

```
public fun indexOf(element: @UnsafeVariance E): Int
```

```
public fun lastIndexOf(element: @UnsafeVariance E): Int
```

기본적으로 `List`는 `E` 또는 그의 subtype만 generic으로 받을 수 있도록 했으므로 위의 함수들에서 generic `E` type보다 더 큰 super type을 인자로 받는 것은 type parameter 관점에서 invariant 하다. 그래서 위의 함수들의 인자에서 `E`의 super type을 받으면 compile time error가 발생한다.

`List`에서는 이를 회피하고자 `@UnsafeVariance`라는 annotation을 사용하는데, 이는 사용자가 그 메소드의 파라미터로서 `E`보다 super type을 받지 않거나 메소드의 내부에서 super type을 인자로 받는 것으로 인해 문제가 발생하지 않게 하도록 스스로 관리하겠다는 의미이다. 즉, compiler에게 해당 method의 인자에 관하여 모든 type variance를 허용함으로써 compiler는 사용자가 스스로 해당 파라미터 type이 안전한지 그 책임을 사용자가 지겠다고 이해하므로 compiler time error를 발생시키지 않는다.

그러나 실제 코드에서 `E`보다 super type이 오면 안되는데 super type을 인자로 받는 등 등 코드 내용상 안전하지 않은 type이 와서 runtime error가 발생할 수 있으므로 사용자 스스로 type variance를 잘 준수하고 있는지 확인해야 한다.

C. B.에서 제시한 method가 type safety를 유지하기 위한 조건을 해당 method의 operation을 예시하여 기술하시오.

```
override fun contains(element: @UnsafeVariance E): Boolean
```

위의 메소드는 `List` 내에서 인자로 받아온 `element`가 포함되어 있는지를 `Boolean` type의 값으로 반환하는 함수이다. 그래서 이 메소드를 실행할 때 `E`의 super type의 객체를 인자로 받지 않아야 한다. 이는 만약 super type의 element가 `List`에 포함되어 있는지 검사할 경우, `E`로 선언된 `List`의 element와 value를 비교하지 못하는 super type일 수 있으므로 `@UnsafeVariance`에 의해 컴파일러가 generic type safety가 보장되지 않더라도 컴파일은 error 없이 될 수 있지만 runtime에서 error가 발생할 수 있다. 그래서 `E` type으로 선언된 `List`의 `contains` method를 사용할 때는 `E` 또는 `E`의 subtype을 사용하는 것이 안전하다.

```
val integerList: List<Int> = listOf(1, 2, 3)

if (integerList.contains(1.0)) {
    println("Found it!")
} else {
    println("Not found.")
}
```

예를 들면 generic type을 `Int`로 선언한 `List`인 `integerList`에서 `1.0`이 포함되어 있는지를 확인하는 코드이지만, `1.0`은 `Double` 값이므로 `int` 또는 이의 subtype이 아니다. 그래서 runtime error를 발생시킨다.

```
val integerList: List<Int> = listOf(1, 2, 3)
val anyList: List<Any> = integerList // List<Int>를 List<Any>에 할당

if (anyList.contains(1.0)) {
    println("Found it!")
} else {
    println("Not found.")
}
```

이를 위의 코드로 변경하면 `integerList`를 `Any` type의 `List`로 선언된 변수에 할당하여 `1.0`이 포함되어 있는지를 확인한다. 그러면 `Double`은 `Any`의 subtype이 되므로 아무런 error 없이 정상적으로 수행 가능하다. 결과적으로 위의 코드는 `Not found`를 출력한다.

```
public fun indexOf(element: @UnsafeVariance E): Int
```

마찬가지로 `indexOf` 메소드도 `List` 내에서 인자로 받아온 `element`가 포함되어 있는지를 확인하여 존재하지 않으면 `-1`을 반환하고, 존재하면 그 `element`의 가장 처음으로 오는 index를 반환하는 함수이다. 그래서 이 메소드를 실행할 때 type safety를 지키려면 `E`의 super type의 객체를 인자로 받지 않아야 한다.

```
val integerList: List<Int> = listOf(1, 2, 3)
println(integerList.indexOf(2.0))
```

예를 들면 generic type을 `Int`로 선언한 `List`인 `integerList`에서 `2.0`이 몇 번째 index에 있는지를 반환하는 코드이지만, `2.0`은 `Double` 값이므로 `int` 또는 이의 subtype이 아니다. 그래서 runtime error를 발생시킨다.

```
val integerList: List<Int> = listOf(1, 2, 3)
val anyList: List<Any> = integerList

println(anyList.indexOf(2.0))
```

이를 위의 코드로 변경하면 `integerList`를 `Any` type의 `List`로 선언된 변수에 할당하여 `2.0`이 `List`의 몇 번째 원소로 있는지를 확인한다. 그러면 `Double`은 `Any`의 subtype이 되므로 아무런 error 없이 정상적으로 수행 가능하다. 결과적으로 위의 코드는 `-1`를 출력한다.

`lastIndexOf`도 `indexOf` 메소드의 내용과 유사하며, 단지 차이점은 `List`에서 찾고자 하는 element에서 가장 먼저 오는 index가 아니라 마지막에 나타나는 index를 반환하는 메소드이다.

3. SCOPE FUNCTIONS

아래는 scope function also를 이용한 one-line swap 코드이다.

```
var a = 1
var b = 2
a = b.also { b = a }
```

A. 다른 scope function을 사용하여 변수 추가 없이 동일한 기능을 구현하는 one- line swap 코드를 작성하시오.

`with`와 `run` scope function으로 구현 가능하다. 첫 번째는 `with`로 one-line swap을 구현한 코드이고, 두 번째는 `run`으로 구현한 코드이다.

```
var a = 1
var b = 2
a = with(b) { b = a; this }
```

```
var a = 1
var b = 2
a = b.run { b = a; this }
```

B. also와 해당 scope function 간의 차이를 비교하여 기술하시오.

```kotlin
public inline fun <T> T.also(block: (T) -> Unit): T {
    block(this)
    return this
}
```

`also`는 `T` type의 context object의 extension function으로 정의되며, context object를 return value로 갖는다. 또한 인자로 넘기는 함수 `block`에서는 인자로 context object를 받는다. 그래서 `also`의 함수 인자로 lambda를 받을 때 receiver type으로 받지 않아서 context object를 접근할 때 `it`으로 접근할 수 있다.

`block` 함수 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 후 context object 자체가 반환된다.

```kotlin
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 `block` 함수에서 receiver로 접근하기 위한 context object이다. 그래서 `run`과는 달리 context object가 lambda expression과 함께 인자로 들어간다. Receiver type으로 `block`을 받으므로 context object에 접근할 때는 `this`를 사용한다.
`block` 함수 실행 후 receiver를 반환하는 것이 아니라 그 `block` 함수의 return value를 반환한다.

`also`에서는 context object 그 자체가 반환되지만, `with`에서는 context object인 `receiver`의 `block` 함수 수행 결과가 반환된다는 게 가장 큰 차이다. 또한 `with`는 `also`와 달리 context object를 직접 인자로 받는다는 점이 다르다.

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

`run`은 `with` 함수와 유사하지만, 전자는 context object에 관해 extension function으로 정의되는 반면에 후자는 context object 그 자체를 scope function에 직접 인자로 넘긴다는 차이가 있다. `also`와도 유사하지만 `run`은 `also`와는 달리 context object를 반환하지 않고 이를 인자로 받는 `block` 함수의 실행 결과를 반환한다.

C. 위 코드의 동작 flow 와 변수 값 swap이 가능한 이유를 구체적으로 설명하시오.

먼저 변수 `a`와 `b`에 각각 1과 2를 할당한다. 이후 `b`에 관하여 `also` scope function을 수행하는데, `also`는 integer type의 context object인 `b`를 갖는다고 해석할 수 있다. 그리고 `b = a` 식을 수행하는 lambda를 `block` 함수 인자로 넘긴다. 이때 `also` 함수 파라미터 괄호 안에 넣지 않고 바깥에 따로 적었는데, 이는 trailing lambda로 작성한 것이다.

`block` 함수에서는 context object `b`를 receiver로 받아서 `block` 함수 내용을 수행하고, 그 결과를 `it`을 통해 context object에 전달하여 업데이트 한다. 그러나 `also` 함수는 lambda 함수를 실행한 후 context object를 가리키는 `this`인 `b`를 반환하므로 `block` 함수에서 `a` 값을 `b`에 새로 할당한다 하더라도 `also` 함수 자체는 이전 `b` 값을 반환한다. 그러므로 위의 코드에서 `also`를 통해 이전의 `b` 값 자체는 그대로 반환되면서 `a`에 할당되지만, `also`에 인자로 넘긴 lambda를 통해 `b`에 `a` 값이 할당된다.

그래서 변수 `a`와 `b`는 2와 1이라는 서로 뒤바뀐 값을 갖게 된다.

4. MULTI-DIMENSIONAL ARRAY

Array를 사용하여 아래 행렬의 곱을 출력하는 Kotlin 프로그램을 구현하시오.

$$X = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \qquad Y = \begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix}$$

- 행렬 `X`, `Y`를 top-level 변수로 선언하고, `ArrayOf()`와 `intArrayOf()`를 사용하여 초기화
- `main()` 함수 안에서 for loop 사용하여 matrix multiplication 구현하고, 행렬 `X`, `Y` 와 결과를 차례로 출력

```
val x = arrayOf(
        intArrayOf(1, 2, 3),
        intArrayOf(4, 5, 6),
        intArrayOf(7, 8, 9),
    )

val y = arrayOf(
    intArrayOf(1, 4, 7, 10),
    intArrayOf(2, 5, 8, 11),
    intArrayOf(3, 6, 9, 12),
)


fun main(){
    val result = Array(x.size) {
        _ -> IntArray(y[0].size) {
            _ -> 0
        }
    }

    for (i in x.indices) {
        for (j in y[0].indices) {
            for (k in y.indices) {
                result[i][j] += x[i][k] * y[k][j]
            }
        }
    }

    println("행렬 X")
    for (i in x){
        var s = ""
        for (j in i) {
            s += j
            s += " "
        }
        println(s)
    }
    println()
```

```
39
40      println("행렬 Y")
41      for (i in y){
42          var s = ""
43          for (j in i) {
44              s += j
45              s += " "
46          }
47          println(s)
48      }
49      println()
50
51      println("행렬 곱 결과")
52      for (i in result){
53          var s = ""
54          for (j in i) {
55              s += j
56              s += " "
57          }
58          println(s)
59      }
60
61      return
62  }
```

`intArrayOf`으로 3개의 row를 지니는 행렬 `x`와 `y`를 `arrayOf`으로 초기화하고, 각 원소마다 대응되는 `x`의 행과 `y`의 열의 원소끼리 곱해서 원소 값을 계산한다. 이는 3개의 `for` loop으로 구현 가능하다. 그리고 행렬의 내용을 출력할 때도 `for` loop으로 각 행렬의 변수의 원소를 하나씩 출력한다. 실제 코드는 첨부한 소스코드 파일 참고.