캡스톤디자인I 중간고사 레포트

20191629 이주헌

1. Sequences

Kotlin에서 Sequence는 다른 원소를 **포함**하는 Collection과는 달리, 반복자를 호출할 때마다 값을 **생성**하는 데이터 타입을 이르는 인터페이스이다. 이 동작 방식을 두고 **lazy evaluation**이라고 부르는 경우도 있다.

A. map과 first

Kotlin Sequence의 map 함수는 어떤 시퀀스의 모든 원소에 대해 주어진 변환을 적용한 뒤 만들어진 새로운 시퀀스를 반환한다. 실제 라이브러리 코드는 다음과 같다.

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

internal class TransformingSequence<T, R>
constructor(private val sequence: Sequence<T>, private val transformer:
(T) -> R) : Sequence<R> {
  override fun iterator(): Iterator<R> = object : Iterator<R> {
    val iterator = sequence.iterator()
    override fun next(): R {
      return transformer(iterator.next())
    }

    override fun hasNext(): Boolean {
      return iterator.hasNext()
    }
  }

  internal fun <E> flatten(iterator: (R) -> Iterator<E>): Sequence<E> {
    return FlatteningSequence<T, R, E>(sequence, transformer, iterator)
  }
}
```

시퀀스는 정의상 값이 사용될 때 생성되며, 이 생성 알고리즘이 외부 상태에 전혀 구애받지 않는 순수 함수(pure function)라고 보장할 수 없다. 따라서, TransformingSequence라는

이름의 새로운 시퀀스 오브젝트를 생성하여 map 함수에 넘긴 변환 함수를 저장하는 방식으로 구현되어 있다. 이는 시퀀스 원소의 값이 아닌, 기존에 존재하는 시퀀스를 다른 시퀀스로 감싸 반환하는 함수이므로, 중간 연산(intermediate operation)의 일종이라고 볼 수 있다.

first 함수는 두 가지 종류가 있는데, 하나는 아무 인자를 받지 않는 것이고 다른 하나는 반환할 원소의 조건을 지정하는 한정자를 받는 것이 있다. 두 종류 모두 반환값은 시퀀스가 아닌 시퀀스 내부의 값이다.

```
public fun <T> Sequence<T>.first(): T {
  val iterator = iterator()
  if (!iterator.hasNext())
    throw NoSuchElementException("Sequence is empty.")
  return iterator.next()
}

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.")
}
```

원소 조건을 지정하지 않는 first 함수는 현재 시퀀스의 반복자에 next 메서드를 호출하면서 값을 찾는다. 원소 조건을 지정하는 first 함수는 for…in 반복문을 사용하는데, for…in 반복문 역시 내부적으로는 반복자를 사용해서 구현되어 있으므로 기본적인 구현 방식은 첫 번째 first 함수와 같다. 두 함수 모두 시퀀스의 원소 생성 방식을 바꾸지 않고 원소 그 자체를 생성한 뒤 반환하는 함수이므로, first 함수는 종단 연산(terminal operation)의 일종이다.

B. Sequence의 lazy evaluation

Kotlin의 시퀀스는 실제로 값이 필요할 때까지 (즉, 종단 연산이 수행될 때까지) 값을 계산하지 않는 lazy evaluation 성질을 가지고 있는 데이터 컨테이너이다. 이는 Kotlin에서 제공하는 반복자(iterator) 패턴을 이용해서 구현이 되어 있다.

기본적으로 모든 시퀀스는 반환할 원소를 저장하지 않고, 원소를 생성해내는 알고리즘을 가지고 있다. 또, 보통 다른 데이터 컨테이너는 반복자를 이용해서 다음 원소를 추출할 때 이용하기 위해 현재까지 반환한 데이터의 포인터 등을 가지고 있지만, 시퀀스는 반복자의 next 함수를 호출할 때 비로소 자신이 가진 원소 생성 알고리즘을 이용해서 원소를 만들어내고 반환한다.

2. Generic

Generic(제네릭)은 하나의 클래스나 함수 정의를 여러 타입에 대해 적용할 수 있도록 하는 프로그래밍 언어의 기술을 말한다. 다른 언어의 예시로는 C++의 템플릿이나 Haskell의 타입 클래스 등을 들 수 있다. 이 문제에서는 Kotlin 표준 라이브러리의 List 함수를 분석하며 제네릭의 특성에 대해 알아본다.

```
public interface List<out E> : Collection<E> {

  override val size: Int
  override fun isEmpty(): Boolean
  override fun contains(element: @UnsafeVariance E): Boolean
  override fun iterator(): Iterator<E>
  override fun containsAll(elements: Collection<@UnsafeVariance E>):
Boolean
  public operator fun get(index: Int): E
  public fun indexOf(element: @UnsafeVariance E): Int
  public fun lastIndexOf(element: @UnsafeVariance E): Int
  public fun listIterator(): ListIterator<E>
  public fun listIterator(index: Int): ListIterator<E>
  public fun subList(fromIndex: Int, toIndex: Int): List<E>
}
```

A. Type Parameter Variance

Kotlin에는 제네릭으로 만들어진 타입끼리의 상속 관계를 명확히 하기 위해 타입 인자 변성(type parameter variance)이라는 개념을 도입하였다. 이 개념이 없으면 제네릭 클래스 C, 부모 클래스 Base, 자식 클래스 Derived에 대하여 C<Base>와 C<Derived>의 관계를 명확히 도식하기 어렵다. 여기서 이 관계는 각 객체의 특성에 따르는 것으로, C<Base>와 C<Derived>가 서로 상속 관계일 수도 있고, 역으로 상속받는 관계이거나 혹은 아예 관련이 없는 서로 다른 타입이어야 할 수도 있다.

먼저 공변성(covariance)은 Kotlin의 타입 인자에 out 키워드를 붙여서 명시할 수 있다. 예를 들어, C<out T>로 정의된 제네릭 클래스에 대하여, C<Base>는 C<Derived>의 “자식” 관계임이 컴파일 타임에 보장된다. (실제 JVM 바이트코드로 컴파일되면 Java Wildcard type으로 타입 정보가 지워지므로, 실제 자식 여부는 판별할 수 없다.) 따라서, out으로 정의된 타입 인자는 메서드의 반환값으로밖에 사용될 수 없다.

반대로 반변성(contravariance)은 타입 인자에 in 키워드를 붙여서 명시할 수 있다. 공변성과는 반대로, C<in T>로 정의된 제네릭 클래스에 대하여 C<Derived>가 C<Base>의 "자식" 관계임이 컴파일 타임에 보장된다. 따라서, in으로 정의된 타입 인자는 메서드의 인수 타입으로밖에 사용할 수 없다.

B. List 타입의 타입 변성 충돌

위 List 인터페이스의 정의에서 원소 타입 E는 공변적으로 정의되어 있다. 즉, 일반적인 경우 E 타입의 값은 메서드의 반환값으로밖에 사용될 수 없다. 그러나, 리스트 인터페이스의 contains와 indexOf 메서드 E를 반환값이 아닌, 메서드의 인자로 받고 있다. 이는 타입 시스템이 타입 간의 호환성을 완전히 검증할 수 없으므로 컴파일 타임 에러가 되어야 한다.

그러나 Kotlin 표준 라이브러리에서는 이를 UnsafeVariance 어노테이션을 사용해서 해결한다. 어노테이션은 컴파일 타임에 어떤 JVM 문법 구조에 대하여 메타데이터를 주입하는 Java의 기능으로, Kotlin 컴파일러는 이 메타데이터를 읽어들여 프로그래머가 타입 공변성 위반의 위험성을 알고 쓴 함수인지 판별한다. 이 경우, 컴파일러는 이 오류가 의도적인 것이라고 간주하고 컴파일 에러를 띄우지 않는다.

C. 타입 안정성?

위 함수는 타입 공변성에 어긋나는 것처럼 보이지만 사실 타입 안정성은 보장된다. 먼저 contains 메서드는 인자로 주어진 값이 리스트에 존재하는지 확인하고, 그 여부를 Boolean 타입으로 반환한다. 또, indexOf 메서드는 인자로 주어진 값이 리스트에 몇 번째 원소로 존재하는지 확인하고, 그 원소의 번호를 반환한다.

이 두 메서드의 작동 원리를 생각해 보았을 때, 두 메서드 모두 주어진 리스트의 상태를 변화시키지 않는 메서드임을 알 수 있다. 즉, 해당 리스트의 타입 E와 관련이 있지만 명백히 다른 타입 E'가 인자로 주어지더라도, 두 함수 모두 리스트의 원소와 인자로 주어진 값이 같은지만 확인하면 되기 때문에 리스트 객체 자체의 타입 안정성은 보장된다.

리스트 원소와 인자 값을 비교하는 과정에서 타입 불일치로 인한 오류가 발생할 가능성은 있지만, Kotlin에서 기본으로 제공하는 is와 같은 연산자를 사용하면 같은 타입인지 판별하는 것은 쉽게 구현할 수 있다.

3. Scope functions

아래는 scope function을 이용한 한 줄짜리 swap 코드이다. 이는 Kotlin 사용자들 간에 자주 사용되는 숙어(idiom)처럼 이용되고 있다.

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

A. 다른 방식으로 바꾸는 방법

위 함수는 아래와 같이 작성해도 작동한다.

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

B. also와 apply의 차이

먼저 각 함수의 시그니처는 아래와 같다.

```
inline fun <T> T.apply(block: T.() -> Unit): T
inline fun <T> T.also(block: (T) -> Unit): T
```

여기서 apply 함수는 적용된 대상 객체를 람다 메서드의 this 수신자로 설정한 뒤 해당 블록을 실행하지만, also 함수는 적용된 대상 객체를 람다 함수의 인자로 전달한 뒤 해당 블록을 실행한다. 따라서 전자는 어떤 객체의 멤버를 여러 번 호출해야 하는 경우 주로 사용하고, 후자는 어떤 객체를 가지고 여러 단계의 복잡한 연산을 할 때 보통 사용된다.

그러나 여기서는 원래 값을 다시 반환하는 용도로 사용되기 때문에 블록 호출 방식의 차이는 크게 의미가 없다. 정확한 원리는 아래에 서술한다.

C. 원리?

두 함수 모두 함수 실행 후 함수가 실행된 객체를 반환한다는 특성을 이용한다. 먼저 b.also 함수를 실행하는 시점에서 반환값은 (Int)2로 확정된다. Kotlin에서 Int 값은 불변하기에 a에는 b.also가 실행된 후 b에 들어 있던 2로 할당된다. 다만 Kotlin에서 = 연산자는 right-associative한 연산이므로 b.also { b = a }가 먼저 실행된다.

also 함수를 실행하는 과정에서 람다 함수를 만들게 되는데, 람다 함수 생성 시에 Kotlin은 이 함수가 만들어진 스코프를 바탕으로 클로저를 생성한다. 이 클로저는 해당 스코프에 있는 변수들을 레퍼런스로 받아 수정할 수 있으며, 이 점을 활용하여 람다 함수 바깥에 있는 b 변수를 a 변수의 값으로 할당한다.

할당이 끝난 뒤 마지막 변수 a에 값이 할당되는데, 위에서 설명한 바와 같이 b.also를 호출하는 시점에서 also의 반환값은 (Int)2로 결정되었으므로 람다 함수에서 진행한 값 변경과 관계없이 a 변수는 2로 할당된다.

4. Matrix multiplication

별첨된 코드 참고.