# 9장 제네릭스

이 장에서는 제네릭스에 대해 설명한다. 제네릭스는 여러분이 알지 못하는 타입의 데이터를 조작하는 코드를 작성할 수 있게 해주는 코틀린 타입 시스템이 제공하는 강력한 기능이다. 어떻게 제네릭 선언을 정의하고 사용하는지 살펴보고, 제네릭을 사용할 때 런타임에 타입 소거(type erasure)와 구체화(reification) 시 어떤 문제가 있는지 논의한 후, 같은 제네릭 타입에 대해 타입 파라미터 값이 달라지는 경우까지 하위 타입 관계를 확장함으로써 제네릭스의 유연성을 향상시켜주는 중요한 개념인 변성(variance)에 대해 설명한다. 이와 연관된 주제로 기존 타입에 대해 새로운 이름을 부여할 수 있는 타입 별명(type alias) 개념도 다룬다.

## 구조

* 제네릭 선언
* 타입 바운드(type bound)와 타입 제약(type constraint)
* 타입 소거와 구체화된 타입 파라미터
* 선언 지점 변성(declaration site variance)
* 프로젝션(projection)
* 타입 별명

## 목표

코틀린 제네릭스 선언의 기초를 배우고 자바 제네릭스와 코틀린 제네릭스가 어떤 차이가 있는지 살펴본다. 그리고 구체화한 타입 파라미터와 변성을 사용해 더 유연한 제네릭 API를 만드는 방법을 배운다.

## 타입 파라미터

앞 장에서 배열이나 여러 컬렉션 클래스와 같은 제네릭 타입이나 map(), filter(), sorted() 등의 제네릭 함수를 이미 살펴봤다. 이 절에서는 여러분의 코드를 더 일반화해서 더 유연하게 만드는 방법과 코틀린 타입 시스템의 고급 기능을 사용하는 방법을 설명한다.

### 제네릭 선언

어떤 선언을 제네릭 선언으로 만들려면 하나 이상의 타입 파라미터를 추가해야 한다. 이렇게 추가한 타입 파라미터를 선언 내부에서는 일반적인 타입 대신 사용할 수 있다. 선언을 사용할 때는(예를 들어 클래스 인스턴스를 만들거나 함수를 호출할 때는) 타입 파라미터를 대신할 실제 타입을 지정해야 한다.

<코드>

val map = HashMap<Int, String>()  
val list = arrayListOf<String>()

</코드>

컴파일러가 문맥에서 타입 인자의 타입을 추론할 수 있으면 타입 인자를 생략할 수 있는 경우도 있다.

<코드>

// map의 타입을 명시했기 때문에, HashMap 클래스의 타입 인자를 추론할 수 있음  
val map: Map<Int, String> = HashMap()

// arrayListOf()에 전달된 인자의 타입(모두 String)으로부터 타입 인자를 추론할 수 있음  
val list = arrayListOf(“abc”, “def”)

</코드>

**자바와 코틀린의 차이**: 코틀린과 자바의 제네릭 메서드에 타입 인자를 전달하는 방식의 차이에 유의하라. 자바에서는 Collections.<String>emptyList()처럼 점 바로 뒤에 각괄호를 사용해 타입을 전달하지만, 코틀린에서는 emptyList처럼 함수 이름 바로 뒤에 타입을 전달한다. 다만 자바에서는new ArrayList() 같은 스타일을 사용하고 코틀린에서는ArrayList() 같은 방식을 사용하기 때문에 클래스 생성자를 호출은 비슷하다.

자바도 클래스 생성자를 호출할 때 타입 추론을 지원하지만, 코틀린과 달리 다이아몬드 연산자(<>)를 추가해야 한다.

<코드>

Map<Int, String> map = new HashMap<>() // new HashMap()이 아님 !!!

</코드>

이유는 제네릭스가 추가된 자바 5 이전의 코드와 하위 호환성을 유지하기 위해서다.

이제 직접 제네릭 선언을 만들어보자.

어떤 주어진 타입의 값을 저장할 수 있는 트리를 표현하는 클래스를 정의하고 싶다고 하자.

<코드>

class TreeNode<T>(val data: T) {  
 private val \_children = arrayListOf<TreeNode<T>>()  
 var parent: TreeNode<T>? = null  
   
 private set  
   
 val children: List<TreeNode<T>>get() = \_children  
   
 fun addChild(data: T) = TreeNode(data).also {  
 \_children += it  
 it.parent = this  
 }  
   
 override fun toString() =  
 \_children.joinToString(prefix = "$data {", postfix = "}")  
}  
  
fun main() {  
 val root = TreeNode("Hello").apply {  
 addChild("World")  
 addChild("!!!")  
 }  
   
 println(root) // Hello {World {}, !!! {}}  
}

</코드>

클래스의 타입 파라미터를 각괄호 안에 써야 한다. 타입 파라미터는 클래스 이름 바로 뒤에 온다. 타임 파라미터 이름은 아무 이름이나 가능하지만, 관습적으로 T, U, V 등의 (짧은) 대문자를 사용한다. 클래스 안에서는 타입 파라미터를 변수나 프로퍼티, 함수의 타입이나 다른 제네릭 선언의 타입 인자로 쓸 수 있다.

**자바와 코틀린의 차이**: 제네릭 클래스나 인터페이스를 사용해 데이터의 타입을 지정할 때는 반드시 타입 인자를 명시해야 한다. TreeNode처럼 타입 파라미터를 지정하지 않은 로타입(raw type)을 사용할 수 있는 자바와 달리 코틀린에서는 이를 허용하지 않는다. 코틀린에서는 TreeNode<String>처럼 구체적인 타입을 지정하거나, TreeNode<U> 처럼 타입 인자로 받은 타입을 반드시 지정해야 한다.

제네릭 클래스 생성자를 호출할 때는 타입 인자가 불필요한 경우가 자주 있다. 대부분의 경우 컴파일러가 문맥에서 타입 인자를 추론해준다. 그래서 앞의 예제 코드에서 TreeNode(“Hello”)처럼 생성자를 호출할 수 있다. 다만 상위 클래스 생성자에 대한 위임 호출은 예외다. 예제 코드를 약간 바꿔보자.

<코드>

open class DataHolder<T>(val data: T)  
  
// 실제 타입을 상위 타입의 타입 인자로 넘김  
class StringDataHolder(data: String) : DataHolder<String>(data)  
  
// 타입 인자를 상위 타입의 타입 인자로 넘김  
class TreeNode<T>(data: T) : DataHolder<T>(data) { ... }

</코드>

컴파일러는 일반 생성자 호출과 달리 생성자 위임 호출의 타입 인자를 추론해 주지 못한다. 따라서 항상 위임 호출의 타입 인자를 명시해야 한다. 이 두 가지 경우를 비교해보자.

<코드>

// Error: DataHolder<String>를 명시해야 함  
class StringDataHolder(data: String) : DataHolder(data)  
  
// Ok: DataHolder<String>를 컴파일러가 추론함  
fun stringDataHolder(data: String) = DataHolder(data)

</코드>

타입 파라미터를 상속하지 않는다는 사실에 유의하라. 생성자 파라미터를 상위 타입 생성자의 인자로 전달하는 것과 비슷하게, 타입 파라미터를 상위 타입의 타입 인자로 전달해야 한다. 따라서 TreeNode의 T와 DataHolder의 T는 서로 다른 선언이다. 실제로 두 제네릭 타입 정의가 서로 다른 타입 파라미터 이름을 사용해도 문제가 없다.

<코드>

class TreeNode<U>(data: U) : DataHolder<U>(data) { ... }

</코드>

앞의 예제 코드에 있는 addChild()나 children 정의를 보면 알 수 있는 것처럼 제네릭 클래스에 정의된 함수와 프로퍼티에서 클래스의 타입 파라미터를 사용할 수 있다. 추가로 프로퍼티나 함수에 타입 파라미터를 추가하면 프로퍼티나 함수 자체를 제네릭으로 만들 수 있다.

<코드>

fun <T> TreeNode<T>.addChildren(vararg data: T) {  
 data.forEach { addChild(it) }  
}  
  
fun <T> TreeNode<T>.walkDepthFirst(action: (T) -> Unit) {  
 children.forEach { it.walkDepthFirst(action) }  
 action(data)  
}  
  
val <T> TreeNode<T>.depth: Int  
 get() = (children.asSequence().map { it.depth }.max() ?: 0) + 1  
  
fun main() {  
 val root = TreeNode("Hello").apply {  
 addChildren("World", "!!!")  
 }  
   
 println(root.depth) // 2  
}

</코드>

제네릭 클래스에서와 달리 프로퍼티나 함수를 제네릭으로 선언할 때는 타입 파라미터를 fun이나 val/var 바로 뒤에 위치시킨다는 점에 유의하라.

클래스 멤버 프로퍼티는 타입 파라미터를 가질 수 없고, 오직 확장 프로퍼티만 타입 파라미터를 가질 수 있다. 일반 멤버 프로퍼티에 대한 타입 파라미터를 허용하지 않는 이유는 이런 프로퍼티를 사용할 때 지정한 타입 인자에 따라 여러 값을 제공하는 일이 불가능하기 때문이다. 일반 프로퍼티는 근본적으로 한 값만 제공한다.

<코드>

var <T> root: TreeNode<T>? = null // Error: T를 수신 객체의 타입을 지정할 때만 쓸 수 있음

</코드>

같은 이유로 객체 선언에 타입 파라미터를 추가하는 것도 금지된다.

<코드>

object EmptyTree<T> // Error: 객체에 타입 파라미터를 지정할 수 없음

</코드>

프로퍼티 참조는 타입 인자를 지원하지 않는다. 따라서 제네릭 프로퍼티의 타입 파라미터는 수신 객체 타입으로부터 추론돼야 한다. 제네릭 (확장) 프로퍼티를 선언하면서 타입 파라미터를 사용하지 않는 경우에도 마찬가지 이유로 컴파일 시점 오류가 된다.

<코드>

// Error: 이 위치에 타입 인자를 명시할 수 없음  
val minDepth = TreeNode("").depth<String>  
  
// Error: T를 수신 객체의 타입에 사용하지 않음  
val<T> TreeNode<String>.upperCaseDataget() = data.toUpperCase()

</코드>

## 바운드와 제약

기본적으로 타입 인자로 들어갈 수 있는 타입에는 아무런 제약이 없다. 따라서 타입 파라미터들은 Any? 타입과 동의어인 것처럼 처리된다. 하지만 제네릭 클래스를 구현하면서 다뤄야 할 데이터의 타입에 좀 더 많은 정보가 필요한 경우가 많다. TreeNode 예제를 확장해서 모든 트리 노드에 저장된 값들의 평균을 계산하는 함수를 선언하고 싶다고 하자. 이런 종류의 연산은 수를 저장한 트리에만 적용할 수 있기 때문에 트리 원소가 Number 타입(또는 그 하위 타입)의 값이기를 바란다. 이런 특성을 표현하기 위해 타입 파라미터의 상위 바운드(upper bound)로 Number를 선언할 수 있다.

<코드>

fun <T : Number>TreeNode<T>.average(): Double {  
 var count = 0  
 var sum = 0.0  
 walkDepthFirst { // 깊이 우선으로 노드를 방문하면서 함수 수행  
 count++  
 sum += it.toDouble()  
 }  
 return sum/count  
}

</코드>

타입 파라미터에 상위 바운드가 있으면 컴파일러는 이 타입 파라미터에 공급된 타입 인자의 타입이 상위 바운드의 하위 타입인지 검사한다. 디폴트 상위 바운드는 Any?로 간주되므로 이를 명시할 필요는 없고, 이런 경우 이 타입 파라미터는 모든 코틀린 타입을 타입 인자로 받을 수 있다. Int와 Double이 Number의 하위 타입이기 때문에 다음 코드는 올바른 코드다.

<코드>

val intTree = TreeNode(1).apply {  
 addChild(2).addChild(3)  
 addChild(4).addChild(5)  
}  
println(intTree.average()) // 3.0  
  
val doubleTree = TreeNode(1.0).apply {  
 addChild(2.0)  
 addChild(3.0)  
}  
println(doubleTree.average()) // 2.0

</코드>

average()를 문자열 트리에 대해 호출하면 컴파일 오류가 발생한다.

<코드>

val stringTree = TreeNode("Hello").apply {  
 addChildren("World", "!!!")  
}  
println(stringTree.average()) // Error: String is not subtype of Number

</코드>

final 클래스를 상위 바운드로 사용하면 한 가지 타입만 지정할 수 있기 때문에, 이런 바운드는 쓸모가 없다. 따라서 이런 경우 컴파일러가 경고를 표시한다.

// 제네릭이 아닌 함수로 대신할 수 있다.  
// fun TreeNode<Int>.sum(): Int {...}  
fun <T : Int> TreeNode<T>.sum(): Int { // Warning  
 var sum = 0  
 walkDepthFirst{ sum += it }  
 return sum  
}

타입 파라미터 바운드로 타입 파라미터를 사용할 수도 있으며, 이런 경우를 재귀적 타입 파라미터라고 말한다. 예를 들어 트리 안에 Comparable 인터페이스의 인스턴스만 들어있다면, 최댓값이 들어있는 노드를 찾을 수 있다.

<코드>

fun <T : Comparable<T>> TreeNode<T>.maxNode(): TreeNode<T> {  
 val maxChild = children.maxBy { it.data } ?: return this  
   
 return if (data >= maxChild.data) this else maxChild  
}  
  
fun main() {  
 // Double은 Comparable<Double>의 하위 타입임  
 val doubleTree = TreeNode(1.0).apply {  
 addChild(2.0)  
 addChild(3.0)  
 }  
 println(doubleTree.maxNode().data) // 3.0  
   
 // String은 Comparable<String>의 하위 타입임  
 val stringTree = TreeNode("abc").apply {  
 addChildren("xyz", "def")  
 }  
 println(stringTree.maxNode().data) // xyz  
}

</코드>

바운드가 자신보다 앞에 있는 타입 파라미터를 가리킬 수도 있다. 이런 바운드를 사용해 트리 원소를 가변 리스트에 추가하는 함수를 정의할 수 있다.

<코드>

fun <T, U : T> TreeNode<U>.toList(list: MutableList<T>) {  
 walkDepthFirst{ list += it }  
}

</코드>

U가 T의 하위타입이기 때문에, 위 함수는 트리 원소의 타입보다 더 일반적인 타입의 리스트를 인자로 받을 수 있다. 예를 들어 Int 트리나 Double 트리에 있는 원소들을 (이 두 타입의 공통 상위 타입인) Number 타입의 리스트에 추가할 수 있다.

<코드>

fun main() {  
 val list = ArrayList<Number>()  
   
 TreeNode(1).apply {  
 addChild(2)  
 addChild(3)  
 }.toList(list)  
   
 TreeNode(1.0).apply {  
 addChild(2.0)  
 addChild(3.0)  
 }.toList(list)  
}

</코드>

**자바와 코틀린의 차이**: 코틀린 타입 파라미터의 상위 바운드는 자바의 상위 바운드와 비슷하다. 표현하는 방식이 자바에서는 T extends Number이지만 코틀린에서는 T: Number라는 점이 다르다.

아주 흔한 경우로 타입 파라미터를 널이 아닌 타입으로 제한하는 경우가 있다. 이를 위해서는 상위 바운드로 널이 될 수 없는 타입을 지정해야 한다.

<코드>

fun <T: Any>notNullTreeOf(data: T) = TreeNode(data)

</코드>

타입 파라미터 구문을 사용하면 상위 바운드를 하나만 지정할 수 있다. 하지만 한 타입 파라미터에 여러 제약을 가할 필요가 있을 때도 있다. 이럴 때는 좀 더 복잡한 타입 제약 구문을 통해 타입을 제한할 수 있다. 예를 들어 다음과 같은 인터페이스가 있다고 하자.

<코드>

interface Named {  
 val name: String  
}  
  
interface Identified {  
 val id: Int  
}

</코드>

이름과 식별자가 모두 있는 객체 레지스트리를 정의하고 싶다고 가정하자.

<코드>

class Registry<T> where T : Named, T : Identified {  
 val items = ArrayList<T>()  
}

</코드>

where 절을 클래스 선언 본문 앞에 추가하고 바운드할 타입 목록을 표시한다.

이제 제네릭스의 문법을 맛보았으니 런타임에 제네릭스의 표현을 어떻게 처리할지에 대해 살펴보자.

## 타입 소거와 구체화

앞 예제에서는 타입 파라미터를 사용해 제네릭 선언 안의 변수, 프로퍼티, 함수 타입을 지정하는 것을 살펴봤다. 하지만 타입 파라미터가 항상 실제 타입을 대신할 수 있는 것은 아니다. 예를 들어 다음 코드를 살펴보자.

<코드>

fun <T>TreeNode<Any>.isInstanceOf(): Boolean =  
 data is T && children.all{ it.isInstanceOf<T>() } // Error

</코드>

의도는 주어진 트리의 노드나 자식의 노드가 모두 지정한 타입 T를 만족하는지 검사하는 것이다. 하지만 컴파일러는 data is T라는 식에 오류를 표시한다. 오류가 나는 이유는 타입 소거 때문이다.

자바에 익숙하다면 자바 제네릭스에도 비슷한 제약이 있다는 점을 떠올릴 것이다. 이런 제약은 자바 제네릭스가 자바 5부터 도입됐기 때문이다. 따라서 새 버전 자바 컴파일러와 가상머신은 기존(자바 5 이전) 코드와의 하위 호환성을 위해 기존 타입 표현 방식을 유지해야 했다. 그 결과 JVM에서 타입 인자에 대한 정보는 코드에서 지워지고(그래서 타입 소거라는 용어가 나왔다), 소스 코드에서 List<String>나 List<Number>와 같은 타입은 JVM에서 List라는 같은 타입으로 합쳐진다.

코틀린은 1.0 버전부터 제네릭스가 있었지만, JVM이 주요 플랫폼이었기 때문에 자바와 같은 타입 소거 문제가 생겼다. 런타임에 제네릭 코드는 파라미터 타입의 차이를 인식할 수 없고, 앞에서 본 data is T와 같은 검사는 기본적으로 아무 의미도 없다. isInstance() 함수가 런타임에 호출될 때 T가 어떤 타입을 뜻할지 알 방법이 없다. 마찬가지 이유로 제네릭 타입에 대해 is 연산자를 적용하는 것도 의미가 없다. 다만 이런 경우에는 컴파일러가 타입 인자와 타입 파라미터가 서로 일치하는지를 살펴보고 그에 따라 경고나 오류를 보고한다.

<코드>

val list = listOf(1, 2, 3) // List<Int>  
list is List<Number> // Warning: List<Int>는 List<Number>의 하위 타입임  
list is List<String> // Error: List<Int>는 List<String>의 하위 타입이 아님

</코드>

원소 타입에는 관심이 없고 어떤 값이 리스트인지만 확인하고 싶다면 어떨까? 코틀린의 제네릭 타입은 항상 타입 인자를 포함해야 하기 때문에 이런 경우에도 List만 쓸 수는 없다. 올바른 검사는 다음과 같다.

<코드>

list is List<\*>  
map is Map<\*, \*>

</코드>

여기서 \*는 기본적으로 알지 못하는 타입을 뜻하며, 타입 인자 하나를 대신한다. 이 구문은 실제로는 프로젝션(projection)이라는 특별한 경우에 속한다. 프로젝션에 대해서는 나중에 설명하겠다.

어떤 경우에는 컴파일러가 타입 검사가 맞는지 알 수 있는 정보를 충분히 가지고 있어서 경고/오류를 표시하지 않을 수도 있다. 다음 예제에서 타입 검사는 근본적으로 List<Int>나 Collection<Int> 같은 타입 파라미터의 구체적인 타입과 무관하게 List와 Collection 인터페이스의 관계에 따라 결정된다.

<코드>

val collection: Collection<Int> = setOf(1, 2, 3)  
  
if (collection is List<Int>) {  
 println("list")  
}

</코드>

어떤 값을 \*가 아닌 타입 인자가 붙은 제네릭 타입으로 캐스트하는 것이 허용된다. 하지만 이런 캐스트에는 위험이 따르기 때문에 항상 경고가 표시된다. 이런 식으로 타입을 캐스팅하면 제네릭스의 한계를 우회할 수는 있지만, 런타임까지 실제 타입 오류를 미루는 효과가 있다. 예를 들어 다음 두 식은 모두 경고가 표시되면서 컴파일되지만, 첫 번째 식은 정상적으로 끝나는 반면 두 번째 식은 예외가 발생한다.

<코드>

val n = (listOf(1, 2, 3) as List<Number>)[0] // OK  
val s = (listOf(1, 2, 3) as List<String>)[0] // Exception

</코드>

후자의 예외는 리스트 원소의 값(이 경우에는 Int)을 String 타입의 (정적으로 알려진) 변수에 대입하기 때문에 생긴다.

자바에서는 캐스트를 활용하거나 리플렉션(reflection)을 통해 타입 소거를 우회해야 한다. 두 방식 모두 단점이 있다. 캐스트는 문제를 (컴파일이 되도록) 덮어버려서 나중에 오류가 발생한다. 반대로 리플렉션 API를 사용하면 성능에 영향을 끼칠 수 있다. 코틀린은 이 두 가지 약점을 모두 극복할 수 있는 세 번째 해법을 제공한다.

구체화는 타입 파라미터 정보를 런타임까지 유지한다는 뜻이다. 어떻게 컴파일러가 타입 소거를 우회할 수 있을까? 해답은 인라인한 함수에 대해서만 구체화한 타입 파라미터를 쓸 수 있다는 데 있다. 함수 본문을 호출 위치로 인라인시키기 때문에, 컴파일러가 인라인된 함수에 제공되는 타입 인자의 실제 타입을 항상 알 수 있다.

파라미터를 실체화하기 위해서는 reified 키워드로 해당 타입 파라미터를 지정해야 한다. 이 기능을 활용해 isInstanceOf() 함수를 고쳐보자. 인라인 함수는 재귀 함수일 수 없기 때문에, 먼저 구현을 약간 고쳐 써야 한다.

<코드>

fun <T>TreeNode<T>.cancellableWalkDepthFirst(  
 onEach: (T) -> Boolean  
): Boolean {  
 val nodes = Stack<TreeNode<T>>()  
   
 nodes.push(this)  
   
 while (nodes.isNotEmpty()) {  
 val node = nodes.pop()  
 if (!onEach(node.data)) return false  
 node.children.forEach { nodes.push(it) }  
 }  
   
 return true  
}  
  
inline fun <reified T> TreeNode<\*>.isInstanceOf() =  
 cancellableWalkDepthFirst{ it is T }

</코드>

이 코드는 별도의 cancellableWalkDepthFirst()라는 인라인 되지 않는 함수로 트리 순회 로직을 추출해 사용한다. 예를 들어 이 함수를 다음과 같이 호출할 수 있다.

<코드>

fun main() {  
 val tree = TreeNode<Any>("abc").addChild("def").addChild(123)  
 println(tree.isInstanceOf<String>())  
}

</코드>

컴파일러는 isInstanceOf()를 인라이닝해서 T 대신 실제 타입인 String을 넣는다. 따라서 코드가 다음과 같아진다.

<코드>

fun main() {  
 val tree = TreeNode<Any>("abc").addChild("def").addChild(123)  
 println(tree.cancellableWalkDepthFirst { it is String })  
}

</코드>

자바에서 사용했던 접근 방식과 달리 구체화한 타입 파라미터를 사용하는 해법은 안전하고(검사하지 않는 캐스트를 쓰지 않음) 빠르다(코드가 인라인됨). 하지만 인라인 함수를 사용하면 컴파일된 코드의 크기가 커지는 경향이 있다는 점을 조심해야 한다. 하지만 (앞의 코드에서 사용한 cancellableWalkDepthFirst()처럼) 코드에서 분량이 많은 부분을 별도의 인라인되지 않는 함수로 뽑아내면 코드 크기가 줄어드는 정도를 줄일 수 있다. 그리고 인라인 함수 안에서만 구체화한 타입 파라미터를 쓸 수 있기 때문에 구체화한 타입을 클래스나 프로퍼티와 함께 쓸 수는 없다.

구체화한 타입 파라미터도 한계가 있다. 이로 인해 구체화한 타입과 완전한 타입이 서로 구분된다. 현재는 구체화한 타입 파라미터를 통해 생성자를 호출하거나 동반 객체 멤버에 접근할 수 없다[[1]](#footnote-1).

<코드>

inline fun <reified T> factory() = T() // Error

</코드>

그리고 구체화한 타입 파라미터를 구체화하지 않은 타입 파라미터로 대신할 수는 없다.

<코드>

fun <T, U> TreeNode<\*>.isInstanceOfBoth() =  
 isInstanceOf<T>() && sInstanceOf<U>()

</코드>

이렇게 할 수 없는 이유 역시 타입 소거다. 이 코드에서 T나 U의 실제 타입을 알 수 없기 때문에 인라인된 instanceOf() 함수를 안전하게 호출할 방법을 찾을 수 없다.

이상으로 코틀린 제네릭스에 대한 기본적인 설명을 마친다. 이제 더 고급 주제인 변성(variance)을 다룰 것이다. 변성은 타입이 생산자/소비자 중 어떤 역할을 하느냐를 제어함으로써 더 유연하게 제네릭 타입을 선언할 수 있게 해준다.

## 변성

변성은 타입 파라미터가 달라질 때 제네릭 타입의 하위타입 관계가 어떻게 달라지는지를 설명하는 제네릭 타입의 한 측면이다. 앞 장에서 이미 변성이 다른 제네릭 타입의 예를 본 적이 있다. 예를 들어 배열과 가변 컬렉션은 타입 인자의 하위타입 관계를 유지하지 않는다. 예를 들어 String은 Any의 하위타입이지만, Array<String>은 Array<Any>의 하위타입으로 간주되지 않는다(그렇다고 Array<Any>가 Array<String>의 하위타입으로 간주되지도 않는다). 반대로 List나 Set 같은 불변 컬렉션의 경우, 타입 파라미터의 하위타입 관계가 컬렉션 타입에서도 유지된다. 예를 들어 List<String>은 List<Any>의 하위타입이다.

<코드>

val objects: List<Any> = listOf("a", "b", "c") // 맞음

</코드>

변성을 합리적으로 사용하면 타입 안전성을 해치지 않으면서 API 유연성을 향상시킬 수 있다. 이제부터는 변성의 의미와 코틀린 제네릭스에서 변성을 사용하는 방법에 대해 설명하겠다.

### 변성: 생산자와 소비자 구분

제네릭 클래스와 인터페이스의 타입 파라미터를 다른 타입 인자로 대치함으로써 무한히 많은 타입을 만들어 낼 수 있다. 디폴트로 어떤 제네릭 타입의 타입 인자를 서로 다른 타입으로 대치한 타입들은 서로 하위타입 관계가 없는 것으로 간주된다. 타입 인자들 사이에 하위타입 관계가 있는 경우에도 역시 서로 아무 관계도 없는 타입인 것으로 간주된다. 이런 경우 해당 제네릭 타입이 무공변(invariant)이라고 말한다(무공변이라는 말은 타입 파라미터의 하위타입 관계와 제네릭 타입의 하위타입 관계가 서로 무관하다는 뜻이다). 예를 들어 내장 Array 클래스나 가변 컬렉션 클래스, 또는 우리가 만든 TreeNode 클래스는 모두 무공변이다. 다음 예제는 TreeNode<String>이 TreeNode<Any>의 하위타입으로 간주되지 않는다는 사실을 보여준다.

<코드>

val node: TreeNode<Any> = TreeNode<String>("Hello") // Error

</코드>

반면 불변 컬렉션 같은 어떤 제네릭 타입들은 타입 인자 사이의 하위타입 관계가 그대로 제네릭 타입에서도 유지된다. 다음 절에서는 타입 인자의 하위타입 관계가 여러분이 만든 제네릭 타입의 하위타입 관계에 끼치는 영향을 제어하는 방법을 배울 것이다. 하지만 우선은 왜 어떤 제네릭 타입은 타입 인자 사이의 하위타입 관계를 그대로 유지하고 어떤 타입은 그렇지 못한지를 이해해야 한다.

이런 구분은 어떤 제네릭 타입이 자신의 타입 파라미터(T라고 하자)를 취급하는 방법에 달려있다. 모든 제네릭 타입은 세 가지로 나뉜다.

1. T 타입의 값을 반환하는 연산만 제공하고 T 타입의 값을 입력으로 받는 연산은 제공하지 않는 제네릭 타입인 생산자
2. T 타입의 값을 입력으로 받기만 하고 결코 T 타입의 값을 반환하지는 않는 제네릭 타입인 소비자
3. 위 두 가지 경우에 해당하지 않는 나머지 타입들

마지막 그룹에 있는 일반적인 타입(즉, 생산자도 소비자도 아닌 타입들)의 경우 타입 안전성을 깨지 않고는 하위타입 관계를 유지할 수 없다. 왜 이런 일이 벌어지는지 이해하기 위해, TreeNode 클래스를 예제로 살펴보자. 일단은 타입 파라미터의 하위타입 관계가 그대로 TreeNode에 타입 인자를 넣은 경우에도 성립한다고 가정하자. 하위타입 관계가 성립하므로 TreeNode<String>을 TreeNode<Any>에 대입할 수 있다. 다음 코드를 보자.

<코드>

val stringNode = TreeNode<String>(“Hello”)  
val anyNode: TreeNode<Any> = stringNode  
anyNode.addChild(123)  
val s = stringNode.children.first() // ???

</코드>

이제 문제가 명확히 보인다. Any 타입의 값을 TreeNode<Any>의 자식으로 추가할 수 있기 때문에 stringNode를 anyNode에 대입하고 나면 Int를 String 타입의 트리에 대입할 수 있게 된다. 이런 대입을 허용하면 stringNode.children.first()를 String으로 캐스트할 때 예외가 발생한다. 즉, TreeNode<String>의 자식으로 정수 값을 넣을 수 있으므로 TreeNode<String>의 계약을 위반할 수 있게 된다.

**자바와 코틀린의 차이**: 자바에 익숙하다면 배열 대입으로 인해 발생할 수 있는 악명 높은 ArrayStoreException 예외를 떠올릴 것이다. 실제로 코틀린 배열 타입이 자바 배열과 달리 원소의 하위타입 관계를 그대로 유지하지 않는 이유가 바로 이것이다.

A 타입을 B 타입의 하위타입이라고 간주한다는 말은 A 타입의 값을 B 타입의 값이 쓰일 수 있는 모든 문맥에서 사용할 수 있다는 뜻이다. 하지만 방금 본 경우는 이 말이 성립하지 않는 상황이다. TreeNode<Any> 타입의 값은 어떤 타입의 값이든 자식으로 받을 수 있는 능력이 있지만, TreeNode<String>은 String 타입의 값만 자식으로 받을 수 있다. 따라서 TreeNode<String>은 TreeNode<Any>의 하위타입이 될 수 없다.

그렇다면 왜 List<T> 같은 불변 컬렉션 타입의 변성이 배열과 다를까? 이유는 불변 컬렉션에는 addChild() 같은 함수가 없기 때문이다. 불변 컬렉션은 T 타입의 값을 만들어내기만 하고 결코 소비하지 않는다. 따라서 List<Any>의 기본적인 계약은 Any 타입의 값을 돌려주는 것이다. 마찬가지로 List<String>의 계약은 String 타입의 값을 돌려주는 것이다. String이 Any의 하위타입이기 때문에 String 타입의 값을 돌려주는 List<String>은 자동적으로 Any 타입의 값을 돌려주는 능력도 가지게 된다. 이를 다른 말로 하면 컴파일러가 List<String>가 타입 안전성을 해치지 않고 List<Any> 대신 쓰일 수 있도록 허용할 수 있다는 말이다. 이런 경우를 제네릭 타입이 타입 인자에 대해 공변적(covariant)이라고 말한다. 코틀린에서 생산자 역할을 하는 타입은 모두 공변적이다.

Pair, Triple, Iterable, Iterator 등의 대부분의 내장 불변 타입은 공변적이다. 추가로 함수 타입은 반환 타입에 대해 공변적이다.

<코드>

val stringProducer: () -> String = { “Hello” }  
val anyProducer: () -> Any = stringProducer  
println(anyProducer()) // Hello

</코드>

하지만 공변성이 불변성과 같지 않다는 점에 유의하라. 공변성(타입 파라미터 T에 대한)은 단지 T를 입력으로 사용하지 못하게 방지할 뿐이다. 따라서 가변 타입을 공변적으로 만들 수도 있다. 예를 들어 특정 인덱스에 있는 원소를 삭제할 뿐 원소를 새로 추가할 수는 없는 리스트를 생각해보자.

<코드>

interface NonGrowingList<T> {  
 val size: Int  
 fun get(index: Int): Int  
 fun remove(index: Int)  
}

</코드>

이 타입은 분명 가변 타입이지만 공변적으로 동작한다. 예를 들어 NonGrowingList<String>은 NonGrowingList<Any>가 할 수 있는 일을 모두 할 수 있다.

역도 참이다. 즉, 불변 객체를 표현하는 타입이 공변적이지 않을 수도 있다. 예를 들어 다음을 보자.

<코드>

interface Set<T> {  
 fun contains(element: T): Boolean  
}

</코드>

위 타입은 불변적이지만 생산자가 아니기 때문에 T의 하위타입 관계를 유지하지 않는다. Set<Any>는 아무 값이나 입력을 받을 수 있지만 Set<String>은 문자열만 입력으로 받을 수 있다.

소비자와 같은 타입은 어떨까? 앞에서 논의한 내용에 비춰보면 소비자 역할을 하는 타입은 분명 타입 파라미터의 하위타입 관계를 유지해주지 못한다. 하지만 이런 타입은 타입 파라미터의 하위타입 관계를 역방향으로 유지해준다는 사실이 밝혀졌다. 이 말의 뜻을 이해하기 위해 앞에서 본 Set<T>의 T를 바꾼 두가지 타입인 Set<Int>와 Set<Number>를 살펴보자. Set<T>의 계약은 contains() 함수에 의해 T 타입인 원소를 처리할 수 있어야 한다는 것이다. Set<Number>는 아무 Number 값이나 처리할 수 있고, Set<Int>는 아무 Int 값이나 처리할 수 있다. 하지만 Int는 Number의 하위타입이다. 따라서 Set<Number>는 아무 Int나 처리할 수 있다. 즉, Set<Number>는 Set<Int>의 하위타입처럼 동작한다. 이런 경우를 반공변적(contravariant)이라고 말하며, 실제 코틀린에서는 T를 반공변적이라고 선언함으로써 이런 식의 하위타입 관계를 지정할 수 있다.

예를 들어 함수 타입은 인자 타입에 대해 반공변적이다.

<코드>

val anyConsumer: (Any) -> Unit = { println(it) }  
val stringConsumer: (String) -> Unit = anyConsumer  
stringConsumer("Hello") // Hello

</코드>

지금까지 어떤 주어진 제네릭 타입 X<T,...>이 타입 파라미터 T에 대해 다음과 같은 변성 중 하나임을 살펴봤다.

* X가 생산자 역할을 한다. 이 경우 T를 공변적으로 선언할 수 있고, A가 B의 하위타입이면 X<A>도 X<B>의 하위타입이 된다.
* X가 소비자 역할을 한다. 이 경우 T를 반공변적으로 선언할 수 있고, B가 A의 하위타입이면 X<A>가 X<B>의 하위타입이 된다.
* 나머지 경우, X는 T에 대해 무공변이다.

다음 절에서는 코틀린에서 변성을 표현하는 방법에 대해 살펴보자.

## 선언 지점 변성

두 가지 방법이 있다. 코틀린에서는 타입 파라미터의 변성을 선언 자체에 지정하거나, 타입 인자를 치환하는 사용 지점에서 지정할 수 있다. 이 절에서는 첫 번째 방식인 ‘선언 지점 변성’에 대해 다룬다.

디폴트로 타입 파라미터는 무공변으로 취급된다. 이 말은 제네릭 타입이 타입 파라미터의 하위타입 관계를 유지하지 않는다는 뜻이다(물론 타입 파라미터의 하위타입 관계의 역방향을 유지하지도 않는다). 예를 들어 List 타입을 단순화한 인터페이스와 배열 기반의 불변 구현을 생각해보자.

<코드>

interface List<T> {  
 val size: Int  
 fun get(index: Int): T  
}  
  
class ListByArray<T>(private vararg val items: T) : List<T> {  
 override val size: Int get() = items.size  
 override fun get(index: Int) = items[index]  
}

</코드>

이제 한 쌍의 리스트를 받아서 두 리스트의 모든 원소를 담은 리스트를 돌려주는 concat 함수를 만들자. 이 함수는 원소를 따로 저장하지 않고 원래의 두 리스트를 활용해 원소를 가져온다.

<코드>

fun <T>concat(list1: List<T>, list2: List<T>) = object : List<T> {  
 override val size: Int  
 get() = list1.size + list2.size  
   
 override fun get(index: Int): T {  
 return if (index < list1.size) {  
 list1.get(index)  
 } else {  
 list2.get(index - list1.size)  
 }  
 }  
}

</코드>

이제 모든 것이 잘 작동한다. 하지만 List<Number>와 List<Int>처럼 서로 연관이 있는 두 타입의 리스트를 합치려는 경우 문제가 생긴다.

<코드>

val numbers = ListByArray<Number>(1, 2.5, 3f)  
val integers = ListByArray(10, 30, 30)  
val result = concat(numbers, integers) // Error

</코드>

이유는 이 리스트가 타입 파라미터 T에 대해 무공변이기 때문이다. 이로 인해 List<Int>는 List<Number>의 하위타입이 아닌 것으로 간주된다(역방향의 하위타입 관계도 성립하지 않는다). 따라서 List<Number>를 원하는 함수의 인자로 List<Int>를 넘길 수 없다.

하지만 이런 제약은 너무 심하다. 리스트 인터페이스를 빠르게 훑어보면 이 타입이 실제로는 생산자 타입처럼 동작한다는 점을 알 수 있다. 이 인터페이스의 모든 연산은 T 타입의 값을 반환하기만 하지 입력으로 받지 않는다. 즉, 이 타입은 안전하게 T에 대해 공변적이 될 수 있다. 이를 위해 타입 파라미터 T 앞에 out이라는 키워드를 붙인다.

<코드>

interface List<out T> {  
 val size: Int  
 fun get(index: Int): T  
}

</코드>

이제 컴파일러가 List<Int>가 List<Number>의 하위타입이라는 사실을 이해하므로 concat() 호출은 예상대로 동작한다.

여기서 컴파일러가 다른 장소에서 파라미터를 공변적으로 선언할 수 있게 허용하지 않기 때문에, 생산자를 지정하는 부분이 중요하다. 가변 버전인 List를 살펴보자.

<코드>

interface MutableList<T> : List<T> {  
 fun set(index: Int, value: T)  
}

</코드>

MutableList의 T를 공변적으로 만들면 컴파일 오류가 발생한다.

<코드>

interface MutableList<out T> : List<T> {  
 fun set(index: Int, value: T) // Error: ‘in’위치에서 T를 사용함  
}

</코드>

T 타입의 값을 입력으로 받아서 소비자처럼 동작하는 set() 함수로 인해 이런 일이 발생한다. 기본적인 규칙은 다음과 같다.

어떤 타입 파라미터가 항상 out 위치에서 쓰이는 경우에만 이 타입 파라미터를 공변적으로 선언할 수 있다. out 위치는 기본적으로 값을 만들어내는 위치다. 프로퍼티나 함수의 반환값 타입이나 제네릭 타입의 공변적인 타입 파라미터 위치가 out 위치다. 예를 들어 다음 타입 정의의 경우 타입 파라미터 T가 항상 out 위치에만 사용되기 때문에 올바른 타입 정의다[[2]](#footnote-2).

<코드>

interface LazyList<out T> {  
 // 반환 타입으로 쓰임  
 fun get(index: Int): T  
  
 // 반환 타입의 out 타입 파라미터로 쓰임  
 fun subList(range: IntRange): LazyList<T>  
  
 // 함수 타입의 반환값 부분도 out 위치임  
 fun getUpTo(index: Int): () -> List<T>  
}

</코드>

마찬가지로 in 위치는 값을 함수 인자나 제네릭 타입의 반공변 타입 인자로 소비하는 경우를 뜻한다.

이런 검사에서 생성자 파라미터는 예외라는 점에 유의하라. 생성자는 제네릭 타입의 인스턴스가 존재하기 전에 호출되기 때문이다(생성자는 인스턴스를 생성하기 위해 최초 한번만 호출된다). 이로 인해 ListByArray 구현도 공변적이라고 선언할 수 있다.

<코드>

class ListByArray<out T>(private vararg val items: T) : List<T> { ... }

</코드>

out과 마찬가지로, 반공변인 타입 파라미터 앞에 in 키워드를 붙일 수 있다. 제네릭 타입이 소비자 역할을 할 때 타입 파라미터를 in으로 표시할 수 있다. 이 말은 타입 파라미터가 out 위치에 전혀 사용되지 않는다는 뜻이다. 예를 살펴보자.

<코드>

class Writer<in T> {  
 // 함수 인자로 쓰임  
 fun write(value: T) {  
 println(value)  
 }  
   
 // in 위치에 사용된 Iterable 제네릭 타입의 out 위치 인자로 T를 사용함   
 // 이런 경우 위치가 in 위치로 인정됨  
 fun writeList(values: Iterable<T>) {  
 values.forEach { println(it) }  
 }  
}  
  
fun main() {  
 val numberWriter = Writer<Number>()  
   
 // 맞음: Writer<Number>가 Int도 처리 가능  
 val integerWriter: Writer<Int> = numberWriter  
   
 integerWriter.write(100)  
}

</코드>

앞에서 본 예제의 TreeNode 클래스는 공변으로도 반공변으로도 정의할 수 없다. 타입 파라미터를 in 위치(예: addChild() 함수)와 out 위치(예: data나 children 프로퍼티)에 모두 사용하기 때문이다. 이런 경우에는 타입 파라미터를 디폴트인 무공변으로 지정하는 수밖에 없다. 하지만 모든 자식을 포함하는 트리의 복사본을 만들고 싶다면 어떻게 해야 할까? 이럴 경우 이 작업에 필요한 멤버 연산들은 data나 children 프로퍼티를 가져오는 것뿐이기 때문에 TreeNode인스턴스는 오직 생산자 역할만 수행할 것이다. 이럴 때 TreeNode를 공변적으로만 사용한다고 코틀린 컴파일러를 납득시킬 방법은 없을까? 물론 이런 방법이 있다. 이때 사용할 수 있는 코틀린 언어 도구가 바로 사용 지점 변성이다. 사용 지점 변성을 프로젝션이라고도 한다.

## 프로젝션을 사용한 사용 지점 변성

변성을 지정하는 다른 방법으로 제네릭 타입을 사용하는 위치에서 특정 타입 인자 앞에 in/out을 붙이는 방법이 있다. 이 방식을 프로젝션이라고 부르는데, 일반적으로는 무공변인 타입이지만 문맥에 따라 생산자나 소비자로만 쓰이는 경우에 유용하다.

기존 트리의 복사본을 다른 트리에 추가하는 함수를 구현한다고 하자. 먼저 무공변인 함수로부터 시작하자.

<코드>

fun <T> TreeNode<T>.addSubtree(node: TreeNode<T>): TreeNode<T> {  
 val newNode = addChild(node.data)  
 node.children.forEach { newNode.addSubtree (it) }  
 return newNode  
}

</코드>

이 함수는 잘 작동한다. 다만 두 트리의 원소 타입이 서로 같은 타입일 때만 작동한다.

fun main() {  
 val root = TreeNode("abc")  
 val subRoot = TreeNode("def")  
   
 root.addSubtree(subRoot)  
 println(root) // abc {def {}}  
}

하지만 Int 트리를 Number로 이뤄진 트리에 추가하고 싶은 경우에는 어떻게 해야 할까? Int가 Number의 하위타입이기 때문에, Int 타입이 들어있는 트리 노드를 Number 트리에 추가해도 트리가 원소 타입에 대해 가정하는 내용에 위배되지 않기 때문에, 이 연산은 잘 정의된 연산이다. 하지만 TreeNode<T>는 무공변 타입이므로 addSubtree() 함수에서 인자 타입과 수신 객체 타입이 똑같이 T라고 쓸 수 밖에 없었다. 이로 인해 컴파일러는 Int 트리를 Number 트리에 추가하는 연산을 허용하지 않는다.

<코드>

val root = TreeNode<Number>(123)  
val subRoot = TreeNode(456.7) // Error

</코드>

TreeNode<T> 타입에는 (data 프로퍼티처럼) T 타입의 값을 돌려주는 멤버와 (addChild() 함수처럼) T 타입의 값을 입력으로 사용하는 멤버가 모두 들어있기 때문에, TreeNode<T> 타입 자체는 무공변으로 남을 수 밖에 없다. 하지만 addSubtree() 함수 내부 맥락에서는 인자로 전달된 트리를 오직 소비자로만 사용한다. 이를 통해 필요한 타입 인자를 out으로 표시하면 우리가 원하는 목표를 달성할 수 있다.

<코드>

fun <T> TreeNode<T>.addSubtree(node: TreeNode<out T>): TreeNode<T> {  
 val newNode = addChild(node.data)  
   
 node.children.forEach { newNode.addSubtree(it) }  
 return newNode  
}  
  
fun main() {  
 val root = TreeNode<Number>(123)  
 val subRoot = TreeNode(456.7)  
   
 root.addSubtree(subRoot)  
   
 println(root) // 123 {456.7 {}}  
}

</코드>

또는 추가되는 트리의 원소를 표현하기 위해 첫 번째 타입에 의해 바운드되는 두 번째 타입 파라미터를 도입할 수도 있다.

<코드>

fun <T, U : T> TreeNode<T>.addSubtree(node: TreeNode<U>): TreeNode<T> {  
 val newNode = addChild(node.data)  
   
 node.children.forEach { newNode.addSubtree(it) }  
 return newNode  
}

</코드>

out 프로젝션을 사용하면 타입 파라미터를 추가할 필요 없이 문제를 더 간결하게 해결할 수 있다.

TreeNode<out T>를 프로젝션한 타입이라고 부른다. 프로젝션인 out T는 TreeNode의 실제 타입 인자를 알지는 못하지만 이 타입 인자가 T의 하위 타입이어야만 한다는 뜻이다. TreeNode<out T>를 TreeNode<T>에 속하지만 T에 대해 생산자 역할만 하는 연산만 노출시킨 타입이라고 생각할 수도 있다. 예를 들어 data, children, depth 등의 프로퍼티나 walkDepthFirst() 등의 함수는 입력으로 T 타입의 값을 받지 않기 때문에 생산자 역할만 한다고 할 수 있다. 이런 프로젝션 타입 내에 addChild()과 같은 연산이나 addChildren() 확장과 같이 소비자 역할을 하는 연산이 들어 있기는 하지만, 실제로 이런 연산을 사용할 수는 없다. 이런 연산을 사용하려고 시도하면 컴파일 오류가 난다.

<코드>

fun processOut(node: TreeNode<out Any>) {  
 node.addChild("xyz") // Error: addChild()는 프로젝션에 의해 제외됨  
}

</코드>

이와 비슷하게 in 프로젝션을 통해 타입을 소비자로만 사용하게 할 수도 있다. 예를 들어 다음과 같은 형태로 트리 추가 함수를 작성할 수도 있다.

<코드>

fun <T> TreeNode<T>.addTo(parent: TreeNode<in T>) {  
 val newNode = parent.addChild(data)  
   
 children.forEach { it.addTo(newNode) }  
}

</코드>

이번에는 추가할 노드들이 들어있는 원본 트리가 수신 객체이고, 노드를 추가할 대상 트리가 함수의 인자다. in 프로젝션으로 인해 TreeNode<T>에 대해 정의된 이 함수는 T의 상위타입이 들어있는 트리만 허용한다.

<코드>

fun main() {  
 val root = TreeNode<Number>(123)  
 val subRoot = TreeNode(456.7)  
  
 subRoot.addTo(root)  
   
 println(root) // 123 {456.7 {}}  
}

</코드>

**자바와 코틀린의 차이**: 코틀린 프로젝션은 근본적으로 자바의 extends/super 와일드카드와 같은 역할을 한다. 예를 들어 TreeNode<out Number>와 TreeNode<in Number>는 순서대로 자바의 TreeNode<? extends Number>와 TreeNode<? super Number>에 해당한다.

프로젝션을 사용하면 프로젝션이 적용된 타입 인자에 해당하는 선언 지점 변성이 의미가 없다는 점에 유의하라. 프로젝션이 타입 파라미터의 변성과 일치하면 프로젝션이 불필요하기 때문에 컴파일러가 경고를 표시한다. 반대로 프로젝션과 타입 파라미터의 변성이 일치하지 않으면 컴파일러는 컴파일 오류를 낸다. 다음 예제를 보자.

<코드>

interface Producer<out T>{  
 fun produce(): T  
}  
  
interface Consumer<in T> {  
 fun consume(value: T)  
}  
  
fun main() {  
 val inProducer: Producer<in String> // Error: 프로젝션과 선언 지점 변성이 충돌함  
 val outProducer: Producer<out String> // out이 불필요함  
 val inConsumer: Consumer<in String> // in이 불필요함  
 val outConsumer: Consumer<out String> // Error: 프로젝션과 선언 지점 변성이 충돌함  
}

</코드>

자바 와일드카드와 마찬가지로 프로젝션을 사용하면 타입을 생산자나 소비자로만 사용하라고 제약을 가할 수 있어서 무공변 타입을 더 유용하게 쓸 수 있다. 추가로 코틀린은 제네릭 타입 파라미터를 아무 타입으로나 치환할 수 있게 해주는 특별한 방법도 제공한다. 바로 스타 프로젝션이다.

## 스타 프로젝션

\*로 표시되는 스타 프로젝션은 타입 인자가 타입 파라미터의 바운드 안에서 아무 타입이나 될 수 있다는 사실을 표현한다. 코틀린 타입 파라미터는 상위 바운드만 허용하기 때문에 타입 인자에 스타 프로젝션을 사용하면 타입 인자가 해당 타입 파라미터를 제한하는 타입의 하위 타입 중 어떤 것이든 관계 없다는 뜻이다. 예를 살펴보자.

<코드>

// List의 원소 타입은 `Any?`에 의해 제한되므로 아무 리스트나 가능함  
val anyList: List<\*> = listOf(1, 2, 3)  
  
// 자기 자신과 비교가능한 아무 객체나 가능(T : Comparable<T> 바운드에 의해)  
val anyComparable: Comparable<\*> = "abcde"

</코드>

즉, 스타 프로젝션은 out 프로젝션을 타입 파라미터 바운드에 적용한 것과 같이 동작한다.

**자바와 코틀린의 차이**: 코틀린의 스타 프로젝션은 자바의 ? 와일드카드에 대응한다. 따라서 코틀린 TreeNode<\*>는 자바 TreeNode<?>에 해당한다.

타입 소거와 구체화에 대한 절에서 이미 스타 프로젝션을 한 타입을 타입 검사 연산에 쓸 수 있다는 사실을 살펴봤다.

<코드>

val any: Any = “”  
any is TreeNode<\*>

</코드>

TreeNode의 타입 파라미터는 Any?에 의해 바운드되므로 이를 명시적인 out 프로젝션을 써서 다음과 같이 쓸 수도 있다.

<코드>

any is TreeNode<out Any?> // Ok

</코드>

하지만 Any?를 다른 타입으로 치환하려고 하면 타입 소거로 인해 타입 체크가 불가능해지므로 컴파일러가 오류를 표시한다.

<코드>

any is TreeNode<out Number> // Error

</코드>

TreeNode<\*>와 TreeNode<Any?>처럼 \*를 사용하는 경우와 타입 파라미터 바운드를 비-프로젝션 타입으로 타입 파라미터에 사용하는 경우의 차이를 구분하는 것이 중요하다. TreeNode<Any>는 아무 타입의 값이나 노드 값으로 들어갈 수 있는 트리를 뜻하지만, TreeNode<\*>는 모든 노드가 어떤 공통 타입 T에 속하나 T가 어떤 타입인지 알려져 있지 않은 트리를 뜻한다. 이런 이유로 TreeNode 연산은 T 타입의 값을 소비하는 소비자로 작동한다. 다만 실제 타입을 알지는 못하기 때문에 어떤 값을 트리가 받아들일지도 알 수 없다. 이 말은 바로 앞에서 out 프로젝션의 뜻에 대해 논의했던 내용과 똑같다.

간단히 설명해 스타 프로젝션을 사용하면 타입 인자가 중요하지 않거나 알려져 있지 않은 제네릭 타입을 간결하게 표현할 수 있다.

만약 타입 파라미터에 바운드가 둘 이상 있다면 \*로 명시적인 out 프로젝션을 대신할 수가 없다는 점에도 유의하라. 코틀린 소스 코드에서 타입의 교집합을 명시할 방법이 없기 때문이다.

<코드>

interface Named {  
 val name: String  
}  
  
interface Identified {  
 val id: Int  
}  
  
class Registry<T> where T : Named, T : Identified  
  
// Registry의 타입 파라미터의 바운드는 Named와 Identified의 교집합이다  
var registry: Registry<\*>? = null

</코드>

\*와 명시적인 out의 다른 차이는 \*를 선언 지점 변성이 붙은 타입 파라미터를 대신할 때 쓸 수 있다는 점에 있다. 이런 경우 컴파일러가 오류나 경고를 표시하지 않는다.

<코드>

interface Consumer<in T> {  
 fun consume(value: T)  
}  
  
interface Producer<out T> {  
 fun produce(): T  
}  
  
fun main() {  
 val starProducer: Producer<\*> // Producer<Any?>와 같음  
 val starConsumer: Consumer<\*> // Consumer<Nothing>과 같음  
}

</코드>

반공변 위치(Consumer<\*>)에 적용한 스타 프로젝션은 Nothing 타입을 인자로 지정한 것과 같다. Nothing은 모든 타입의 하위타입이므로, consume() 함수에 아무 타입의 값이나 넣어도 문제가 없다.

## 타입 별명

마지막으로 제네릭스와 직접적인 연관은 없지만 복잡한 제네릭 타입을 다룰 때 도움이 될 수 있는 언어 기능을 하나 소개한다. 바로 타입 별명(type alias)이다.

코틀린 1.1에 타입 별명을 도입한 이유는 기존 타입의 이름을 대신할 수 있는 새 이름을 도입할 수 있게 하기 위해서였다. 주 목적은 제네릭 타입이나 함수 타입처럼 긴 이름을 짧게 부를 수 있게 해주는 것이다. 타입 별명 정의는 typealias 키워드로 이뤄지며, 그 다음에 별명이 오고 = 기호 다음에 실제 타입이 온다.

<코드>

typealiasIntPredicate = (Int) -> Boolean  
typealiasIntMap = HashMap<Int, Int>

</코드>

이제 이렇게 정의한 이름을 =의 오른쪽에 있는 타입 대신 쓸 수 있다.

<코드>

fun readFirst(filter: IntPredicate) =  
 generateSequence{ readLine()?.toIntOrNull() }.firstOrNull(filter)  
  
fun main() {  
 val map = IntMap().also {  
 it[1] = 2  
 it[2] = 3  
 }  
}

</코드>

다른 사용법으로는 내포된 클래스를 더 짧은 이름으로 가리키는 것이다.

<코드>

sealed class Status {  
 object Success : Status()  
 class Error(val message: String) : Status()  
}  
  
typealiasStSuccess = Status.Success  
typealiasStError = Status.Error

</코드>

타입 별명도 제네릭 타입처럼 타입 파라미터를 포함할 수 있다. 특히 제네릭 클래스와 비슷한 방식으로 타입 별명을 정의할 수 있다.

<코드>

typealias ThisPredicate<T> = T.() -> Boolean  
typealias MultiMap<K, V> = Map<K, Collection<V>>

</코드>

그리고 가시성을 사용해 타입 별명이 보이는 영역을 제한할 수도 있다.

<코드>

private typealias MyMap = Map<String, String> // 현재 파일 내부에서만 볼 수 있음

</코드>

현재(코틀린 1.3)는 타입 별명을 최상위에만 선언할 수 있다. 예를 들어 함수나 클래스 내부에서 타입 별명을 선언할 수는 없다.

<코드>

fun main() {  
 typealias A = Int // Error  
}

</코드>

또 다른 제약도 있다. 제네릭 타입 별명에 대해 제약이나 바운드를 선언할 수는 없다.

<코드>

typealias ComparableMap<K : Comparable<K>, V> = Map<K, V> // Error

</코드>

타입 별명이 새 타입을 도입하는 것이 아니고, 기존 타입을 가리키는 새로운 방법을 추가해주는 것뿐이라는 점을 기억하라. 이 말은 타입 별명과 원래의 타입을 자유롭게 바꿔 쓸 수 있다는 뜻이다.

<코드>

typealias A = Int  
  
fun main() {  
 val n = 1  
 val a: A = n  
 val b: Int = a  
}

</코드>

여러분도 이미 알고 있듯이 타입 별명이 기존 타입에 대한 새 이름을 제공하는 유일한 방법은 아니다. 따라서 비슷한 목적으로 사용할 수 있는 여러 언어 기능의 차이를 이해하면 유용하다.

예를 들어 임포트 별명을 사용하면 임포트 지시문의 일부분으로 별명을 도입할 수 있다. 임포트 별명은 함수나 프로퍼티에 대한 별명도 지원하지만, 제네릭 별명을 허용하지는 않는다. 게다가 임포트 별명의 영역은 항상 임포트 지시문이 포함된 파일 안으로 제한된다. 반면 공개(public) 타입 별명은 더 넓은 영역에 쓰일 수 있다.

제네릭 타입이나 함수 타입을 상속해서 새로운 타입 이름을 제공할 수도 있다. 이 방법을 사용하면 제네릭 타입도 정의할 수 있고, 새 이름의 가시성도 제어할 수 있다. 이 방법과 타입 별명의 가장 중요한 차이는 이런 방식의 정의는 원래 타입의 하위타입인 새로운 타입을 만들어낸다는 점에 있다. 따라서 이렇게 만든 이름은 원래 이름과 한 방향으로 밖에 호환되지 않는다.

<코드>

class MyMap<T> : HashMap<T, T>()  
  
fun main() {  
 val map: Map<String, String> = MyMap() // Ok, MyMap은 Map의 하위타입  
 val myMap: MyMap<String> = map // Error  
}

</코드>

게다가 final 클래스는 상속할 수 없지만 이런 클래스에 대한 별명은 만들 수 있다.

인라인 클래스도 원래 타입과 동일한 표현을 사용한다는 점에서는 타입 별명과 같다. 하지만 인라인 클래스는 원래 타입과 호환이 되지 않는 새로운 타입을 만들어낸다는 중요한 차이가 있다. 예를 들어 명시적인 변환을 사용하지 않으면 UInt 값을 Int 타입의 변수에 대입할 수는 없다(역방향 대입도 안 된다).

## 결론

이 장에서는 코틀린 코드에서 추상화를 설계하는 또 다른 도구인 제네릭스의 개념을 배웠다. 이제는 여러분 자신만의 제네릭 API를 설계할 수 있고, 구체화한 타입 파라미터나 변성 등의 고급 개념을 더 간결하고 효율적이며 타입 안전한 코드를 작성하기 위해 활용할 수 있어야 한다. 그리고 어떤 타입 이름을 대신할 수 있는 별명을 도입하게 해주는 타입 별명이라는 유용한 개념에 대해 소개했다. 타입 별명을 사용하면 복잡한 제네릭 타입이나 함수 타입을 더 쉽게 처리할 수 있다.

다음 장에서는 두 가지 서로 연관된 개념을 자세히 살펴본다. 첫 번째로 살펴볼 내용은 애너테이션(annotation)이다. 애너테이션을 사용하면 프로그램 요소에 여러 가지 메타데이터(metadata)를 지정할 수 있다. 코틀린에서는 애너테이션을 다른 기능들과 함께 사용해 코드의 상호 운용성을 세밀하게 조절한다. 이에 대해서는 **12장 자바 상호 운용성**에서 다룬다. **10장. 애너테이션과 리플렉션**에서 살펴볼 두 번째 주요 내용은 프로그램 내부 구조를 살펴보고 동적으로 코드를 호출할 수 있게 해주는 API를 제공하는 리플렉션(reflection)이다.

## 문제

1. 코틀린에서 제네릭 클래스, 함수, 프로퍼티를 정의하는 방법은 무엇인가?
2. 타입 파라미터에 대해 제약을 거는 방법을 설명하라. 자바와 코틀린의 타입 파라미터 제약 기능을 비교해 설명하라.
3. 타입 소거란 무엇인가? 타입 파라미터의 한계를 일반 타입과 비교해 설명하라.
4. 구체화한 타입 파라미터를 사용해 어떻게 타입 소거를 우회할 수 있는가? 구체화한 타입 파라미터의 한계는 무엇인가?
5. 변성이란 무엇인가? 제네릭 코드에서 변성이 중요한 이유는 무엇인가?
6. 코틀린에서 선언 지점 변성을 어떻게 사용하는지 설명하라.
7. 코틀린 사용 지점 변성과 자바 와일드카드를 비교해 설명하라.
8. 스타 프로젝션의 목적이 무엇인지 설명하라.
9. 타입 별명 구문에 대해 설명하라. 임포트 별명이나 상속과 같은 (기존 타입 이름에 대한 새로운 타입 이름을 소개하는) 언어 기능과 타입 별명을 비교하라.

1. 옮긴이 - 실제로는 제네릭 타입 파라미터를 생성자 호출이나 동반객체 접근에 사용할 수 없다. 따라서 이 제약은 구체화한 타입만의 제약은 아니다. [↑](#footnote-ref-1)
2. 옮긴이 - 함수 타입의 반환값으로 타입 파라미터 T가 직접 쓰인 경우에는 out 위치이지만, 함수의 반환 타입이 제네릭 타입이고 T가 이 타입의 인자로 쓰인 경우에는 제네릭 타입의 out 위치에 T가 쓰인 경우에만 공변적이다. 예를 들어 이 예제의 getUpTo가 배열을 반환하면 배열 타입은 인자에 대해 무공변이기 때문에 LazyList 정의에서 T를 공적으로 선언할 수 없다. (fun getUpTo(index: Int): () -> Array<T> // Error: () -> Array<T>에서 T가 무공변 위치에 사용됨) [↑](#footnote-ref-2)