11장 제네릭스

다루는 내용

- 제네릭 함수와 클래스를 정의하는 방법
- 타입 소거와 실체화된 타입 파라미터
- 선언 지점과 사용 지점 변성
- 타입 별명

제네릭 타입 파라미터

- 제네릭스를 사용하면 타입 파라미터를 받는 타입을 정의할 수 있다
- 제네릭 클래스에 구체적인 타입 인자를 넘기면 타입을 인스턴스화할 수 있다
    - 제네릭 타입의 인스턴스가 만들어질 때 이 타입 파라미터를 구체적인 타입 인자로 치환한다

코틀린 컴파일러는 보통 타입처럼 타입 파라미터도 추론할 수 있다

```
val authors = listOf("Seb", "Roman")
```

- 전달된 두 값이 문자열이기 때문에 컴파일러는 이 리스트가 List<String>임을 추론한다
- 빈 리스트를 만드는 경우엔 타입 인자를 추론할 근거가 없어서 직접 타입 인자를 명시해줘야 한다
    - 타입 인자를 명시할 대에는 변수의 타입을 지정해도 되고 함수의 타입 인자를 지정해도 된다

```
val readers: MutableList<String> = mutableListOf()
val readers = mutableListOf<String>()
```

자바의 경우 타입 인자가 없는 raw type을 허용하는 반면 코틀린은 컴파일러가 추론할 수 있어야 한다

- 자바는 뒤늦게 제네릭을 도입했기에 이전 버전과의 호환성을 위해 인자없이 제네릭 클래스 선언이 가능하다
- 코틀린은 처음부터 제네릭을 도입했기에 raw type을 허용하지 않고 제네릭의 타입 인자를 항상 정의해야 한다

제네릭 타입과 함께 동작하는 함수와 프로퍼티

어떤 특정 타입에 대한 것이 아닌 모든 타입을 다룰 수 있는 함수가 필요할 때 제네릭 함수를 작성한다

- 제네릭 함수는 그 자신이 타입 파라미터를 받기 때문에 호출할 때 반드시 구체적 타입으로 타입 인자를 넘겨야 한다
- 제공되는 컬렉션 라이브러리 함수는 대부분 제네릭 함수로 구성되어 있다

```
public fun <T> List<T>.slice(indices: IntRange): List<T> {
```

  - 제네릭 함수가 T를 타입 파라미터로 받기 때문에 임의의 원소 타입인 리스트에 쓰일 수 있다
  - 함수의 타입 파라미터 T가 수신 객체(`List<T>`)와 반환 타입(`List<T>`)에 모두 쓰인다

대부분 컴파일러가 제네릭 함수의 타입 인자를 추론할 수 있어서 지정할 필요가 없다

```
fun main() {
    val authors = listOf("Alice", "Bob", "Charlie")
    val readers = mutableListOf<String>("Bob", "Hadi")
    println(readers.filter { it !in authors })  // [Hadi]
}
```

- `filter` 함수의 경우 `(T) → Boolean` 타입의 함수를 파라미터로 받는다
- 컴파일러가 T가 `String`임을 추론하는 근거

1. `filter`가 `List<T>` 타입의 리스트에 대해 호출될 수 있다는 사실
2. 수신 객체인 reader의 실제 타입이 `List<String>`이라는 사실

타입 파리미터를 선언할 수 있는 위치

- 클래스나 인터페이스 안 정의된 메서드
- 최상위 함수
- 확장 함수
    - filter가 수신 객체 타입 List<T>와 파라미터 함수 타입 (T) → Boolean에 타입 파라미터 T를 사용하는 것처럼

제네릭 함수와 마찬가지로 제네릭 확장 프로퍼티를 선언할 수 있다

```
val <T> List<T>.penultimate: T  // 모든 List 타입에 이 제네릭 확장 프로퍼티 사용 가
    get() = this[size - 2]

fun main() {
    println(listOf(1, 2, 3, 4).penultimate)  // T가 Int로 추론된다
    // Output: 3
}
```

- 일반 프로퍼티는 타입 파라미터를 가질 수 없다
    - 클래스 프로퍼티에 여러 타입의 값을 저장할 수 없으므로 당연히 안 되는 것
    - 일반 프로퍼티를 제네릭하게 타입 파라미터를 사용하면 컴파일러가 에러를 표시한다

제네릭 클래스 선언법

- 자바와 마찬가지로 타입 파라미터를 넣은 `<>`를 클래스나 인터페이스 이름 뒤에 붙이면 된다

- 타입 파라미터를 이름 뒤에 붙이고 나면 클래스 본문 안에서 타입 파라미터를 다른 일반 타입처럼 사용할 수 있다

```
public interface List<out E> : Collection<E> { // List 인터페이스에 E 타입 파라미터
    public operator fun get(index: Int): E  // 인터페이스 안에서 E를 일반 타입처럼 사
}
```

제네릭 클래스를 확장하거나 구현하는 클래스 정의하는 방법

- 기반 타입의 제네릭 파라미터에 대해 타입 인자를 지정한다
- 이 때 구체적인 타입을 넘길 수도 있고 타입 파라미터로 받은 타입을 넘길 수도 있다

```
class StringList: List<String> {    // String을 기반 타입 인자로 지정
    override fun get(index: Int): String  = TODO()  // 타입 인자 T를 구체적 타입 Stri
    // ..
}

class ArrayList<T>: List<T> {   // T는 기반 클래스의 타입 인자로 사용
    override fun get(index: Int): T = TODO() // 타입 파라미터로 받은 T를 사용
    // ..
}
```

- `ArrayList<T>`의 `T`와 `List<T>`의 `T`는 서로 다른 타입 파라미터라서 이름이 같을 필요가 없다
- `ArrayList<T>`를 선언할 때 타입 파라미터 `T`는 ArrayList 클래스의 자체 타입 파라미터이고, `List<T>`는 Kotlin 표준 라이브러리에서 정의된 제네릭 인터페이스이기 때문에 서로 다른 스코프의 타입 파라미터이다

클래스가 자신을 타입 인자로 참조할 수도 있다

- Comparable 인터페이스의 구현을 보면 자신을 타입 인자로 참조하는 패턴을 볼 수 있다

```
interface Comparable<T> {
    fun compareTo(other: T): Int
}
```

```
// String 클래스 예시
class String : Comparable<String> { // 제네릭 Comparable 인터페이스를 구현할 때
    override fun compareTo(other: String): Int = TODO()
}
```

타입 파라미터 제약

- 클래스나 함수에 사용할 수 있는 타입 인자를 제한할 수 있다
- 어떤 타입을 제네릭 타입의 타입 파라미터에 대한 상계로 지정하면 그 제네릭 타입을 인스턴스화할 때 사용하는 타입 인자는 반드시 그 상계로 지정한 타입이거나 그의 하위 타입이어야 한다
- 제약을 가하려면 타입 파라미터 이름 뒤에 콜론(:) 표시 후 상계 타입을 적으면 된다

```
// 상계 타입으로 Comparable<T> 제약
fun <T: Comparable<T>> max(first: T, second: T): T {
    return if (first > second) first else second // compareTo로 컴파일된다
}

fun main() {
    println(max("kotlin", "java"))

    // Argument type mismatch: actual type is 'Int', but 'Comparable<String> &
    println(max("kotlin", 42))  // 둘다 Comparable이긴 하지만 T를 어떤 타입으로 추론
}
```

둘 이상의 제약도 가능하다

```
fun <T> ensureTrailingPeriod(seq: T) where T : CharSequence, T : Appendabl
    if (seq.isNotEmpty() && seq.last() != '.') {
        seq.append('.')
```

```
    }
}
```

- 접근하는 연산 `endsWith`와 변경하는 연산 `append`을 모두 사용하기 위해 `CharSequence`와 `Appendable` 인터페이스를 모두 상계로 제약을 걸어둔다

Nullable한 타입 인자 제외시키기

제네릭 클래스나 함수를 정의한 뒤 그 타입을 인스턴스화할 때 타입 인자로 nullable한 타입을 지정해도 치환이 가능하다

- 타입 파라미터에 따로 ?로 nullable하다고 표시해주지 않아도 nullable한 타입으로 인스턴스화할 수 있다
- 그 이유는 아무런 상계를 정하지 않은 타입 파라미터의 경우 `Any?`를 상계로 정한 파라미터와 같기 때문이다

```
class Processor<T> {
    fun process(value: T) {
        value?.hashCode()   // value는 nullable하기에 safe call 필요
    }
}
```

항상 null이 될 수 없는 타입만 타입 인자로 받게 하려면 제약을 걸어줘야 한다

```
class Processor<T : Any> {
    fun process(value: T) {
        value.hashCode()   // value는 null이 될 수 없으므로 안전하게 사용 가능
    }
}
```

- 이렇게 하면 항상 `T`가 널이 될 수 없기에 `Processor<Stirng?>`같은 코드를 거부한다
- Any 뿐만 아니라 다른 non-null한 타입을 상계로 정해주면 타입 파라미터가 널이 아닌 타입으로 제약된다

제네릭 파라미터를 정의한 위치가 아닌 사용하는 지점에서 제약을 거는 방법도 제공한다

```
class KBox<T>: JBox<T> {
    override fun put(t: T & Any) {TODO()}
    override fun putIfNotull(t: T) {TODO()}
}
```

- 기존대로 제네릭 파라미터를 정의한 위치에 KBox<T : Any>로 제약을 걸었다면 기존 자바의 JBox의 putIfNotNull 메서드를 사용할 수 없다
- 상계가 필요할 때 타입 파라미터를 사용하는 지점에서 `T & 상계타입` 형태로 제약을 걸어줘 해결한다

실행 시점 제네릭스 동작

실행 시점의 타입 소거로 인한 제네릭 클래스의 한계

- JVM의 제네릭스는 보통 타입 소거를 사용해 구현되므로 실행 시점에 제네릭 클래스의 인스턴스에 타입 인자 정보가 들어있지 않다

```
val list1: List<String> = listOf("apple", "banana")
val list2: List<Int> = listOf(1, 2, 3)
```

- 컴파일러는 두 리스트를 서로 다른 타입으로 인식하지만 실행 시점이 되면 이 두 리스트를 완전히 동일한 List 객체로 본다
- 즉, 제네릭 클래스인 List의 타입 파라미터인 `String`, `Int`를 지워버리기 때문에 `is` 검사를 통해 타입 인자를 검사할 수가 없다
- 따라서 파라미터의 타입 인자에 따라 서로 다른 동작을 하도록 함수를 작성할 수가 없다
- 다만 타입 정보 크기가 줄어들어서 애플리케이션의 전체 메모리 사용량이 줄어든다는 제네릭 타입 소거의 장점이 있긴 하다

그렇다면 어떤 값이 맵이 아니라 리스트라는 사실을 어떻게 확인할 수 있을까?

- 타입 파라미터가 2개 이상이라면 모든 타입 파라미터에 스타 프로젝션 구문(*)을 포함시켜야 한다

- 인자를 알 수 없는 제네릭 타입을 표현할 때 스타 프로젝션을 사용한다

```
// List의 타입 인자를 검사할 수 없다
fun printList(l: List<Any>) {
    when (l) {
        is List<String> -> TODO() // Cannot check for instance of erased type 'Lis
        is List<Int> -> TODO()    // Cannot check for instance of erased type 'List<
    }
}

// 리스트라는 사실을 확인하기 위해 스타 프로젝션 사용
fun printList(l: List<Any>) {
    when(l) {
        is List<*> -> {
            for (item in l) {
                if (item is List<*>) {
                    printList(item)
                } else {
                    println(item)
                }
            }
        }
    }
}
```

제네릭 타입으로 타입 캐스팅 시 주의점

```
fun printSum(c: Collection<*>) {
    val intList = c as? List<Int> ?: throw IllegalArgumentException("List is not an
    println(intList.sum())
}

fun main() {
    printSum(listOf(1, 2, 3)) // output: 6
    printSum(listOf("a", "b", "c")) // throw IllegalArgumentException
}
```

- as 캐스팅 시 제네릭 타입을 사용할 때 타입 인자가 다른 타입으로 캐스팅해도 성공한다는 점을 주의하자
    - 컴파일러가 캐스팅 관련 경고를 해주긴 하지만 문제없이 컴파일된다
- 잘못된 타입의 원소가 들어간 리스트를 전달했을 때 컴파일을 잘 되고 실행 시점에 `sum`이 호출되는 도중에 `ClassCastException`이 발생한다

컴파일 시점에 타입 정보가 주어진 경우엔 `is` 검사를 허용한다

```
fun printlnSum(c: Collection<Int>) {
    when (c) {
        is List<Int> -> println("List sum: ${c.sum()}")
        is Set<Int> -> println("Set sum: ${c.sum()}")
    }
}
```

- 컴파일 시점에 컬렉션의 타입 정보가 주어져서 `is` 검사가 가능한 것이다

실체화된 타입 파라미터를 사용하면 타입 인자를 실행 시점에 사용할 수 있다

- 제네릭 타입의 타입 인자 정보는 실행시점에 지워져서 인스턴스화하거나 본문 구현 시 그 타입 인자를 알아낼 수가 없다

```
fun <T> isA(value: Any) = value is T
```

inline + reified 조합을 사용하면 타입 인자를 실행 시점에 유지할 수 있다

- 즉, `inline` 키워드로 호출 위치로 코드를 삽입할 때 `reified` 키워드를 통해 타입 파라미터 `T`의 구체적인 타입 정보를 소거 없이 그대로 코드에 반영할 수 있다

```
inline fun <reified T> isA(value: Any?) = value is T

fun main() {
```

```
    println(isA<String>("abc")) // true
    println(isA<String>(123)) // false
}
```

실체화된 타입 파라미터 활용 예제 - `filterIsInstance`

```
fun main() {
    val items = listOf("one", 2, "three")
    println(items.filterIsInstance<String>())
    // Output: [one, three]
}
```

- `filterIsInstance`는 인자로 받은 컬렉션에서 지정한 클래스의 인스턴스만을 모아 만든 리스트를 반환한다
- 타입 인자를 실행 시점에 알 수 있기 때문에 그 타입 인자를 사용해 타입이 일치하는 원소만 추려내는 것이다

```
inline fun <reified T> Iterable<*>.filterIsInstance(): List<T> {
    val destination = mutableListOf<T>()
    for (element in this) {
        if (element is T) {
            destination.add(element)
        }
    }
    return destination
}
```

- 내부 구현을 보면 inline 및 reified 키워드로 타입 인자를 실행 시점까지 유지시키고 있다
    - 타입 파라미터가 구체적인 타입을 사용하므로 실행 시점에 벌어지는 타입 소거의 영향을 받지 않는다
- 실제 `filterIsInstance<Stirng>()` 호출 시 `element is String`처럼 구체적인 클래스를 참조하는 코드로 변환된다

클래스 참조를 실체화된 타입 파라미터로 대신하기

- java.lang.Class 타입 인자를 파라미터로 받는 API를 자주 사용한다
- `::class.java` 구문을 통해 코틀린 클래스에 대응하는 `java.lang.Class` 참조를 얻는다

```
val serviceImpl = ServiceLoader.load(Service::class.java)
```

이 때 클래스 참조 대신 타입 파라미터를 지정해 사용하면 더 간결하다

```
inline fun <reified T> loadService(): ServiceLoader<T> {
    return ServiceLoader.load(T::class.java)
}

val serviceImpl = loadService<Service>()
```

- 타입 파라미터 T를 reified로 지정하고 그 T에 대해 T::class로 대응되는 클래스를 가져온다

실체화된 타입 파라미터가 있는 접근자 정의

제네릭 타입에 대해 프로퍼티 접근자를 정의하는 경우 inline + reified로 타입 인자의 구체적인 클래스를 참조할 수 있다

```
inline val <reified T> T.canonical: String
    get() = T::class.java.canonicalName

fun main() {
    println(listOf(1, 2, 3).canonical) // Output: java.util.List
    println(1.canonical) // Output: java.lang.Integer
}
```

실체화된 타입 파라미터의 제약

실체화된 타입 파라미터 활용처

1. 타입 검사 및 캐스팅 (`is` , `as`)
2. 코틀린 리플렉션 API (`::class`)
3. 코틀린 타입에 대응되는 자바 클래스 참조 얻기 (`::class.java`)
4. 다른 함수 호출 시 타입 인자로 사용

실체화된 타입 파라미터를 사용할 수 없는 경우

1. 직접 인스턴스화(`T()`)
    - 타입 인자 T가 어떤 생성자를 가지는지 알 수 없기 때문
2. 동반 객체 메서드 호출(`T.Companion`)
    - T가 Companion Object를 가진다는 보장이 없어서 정적으로 참조할 수 없기 때문
3. `inline` + `reified` 함수 안에서 실체화되지 않은 타입 파라미터를 다시 넘기기
    a. 실체화되지 않은 타입 파라미터는 런타임에 존재하지 않으므로 넘길 수 없다
4. `inline` 없이 `reified` 사용

변성

- 변성은 기저 타입이 같고 타입 인자가 다른 여러 타입이 서로 어떤 관계가 있는지 설명하는 개념
- 변성을 잘 활용해야 타입 안전성을 보장하는 API를 만들 수 있다

변성은 인자를 함수에 넘겨도 안전한지 판단하게 해준다

- String은 Any를 확장하므로 Any 타입을 받는 함수에 String을 넘겨도 절대 안전하다
- 하지만 Any와 String이 List 인터페이스의 타입 잎나로 들어가는 경우 자신있게 안전하다고 할 수 없다

```
fun addAnswer(list: MutableList<Any>) {
    list.add(42)
}

fun main() {
    val strings = mutableListOf("abc", "def")
    addAnswer(strings) // Argument type mismatch: actual type is 'MutableList<
}
```

- 컴파일러가 MutableList<Any>가 필요한 곳에 Mutable<String>을 넘기면 안된다는 에러를 내준다
- 이처럼 리스트의 원소를 추가하거나 변경할 때 타입 불일치가 생길 수 있어 List<Any> 대신 List<String>을 넘길 수 없다
    - 다만 읽기 전용일 경우 원소의 추가나 변경이 없기에 넘겨줘도 안전하다

이렇게 제네릭 타입을 다른 타입으로 대체할 수 있는지 판단하는 기준이 바로 변경이고, 이 변성으로 코틀린의 타입 안전성을 지킬 수 있다

클래스 vs 타입

- 제네릭 클래스가 아닌 클래스에서는 클래스의 이름을 바로 타입으로 쓸 수 있다
- 같은 클래스가 적어도 둘 이상의 타입을 구성할 수 있다
    - `var x: String`
    - `var x: String?`

하위 타입

- 어떤 타입 A의 값이 필요한 모든 장소에 어떤 타입 B의 값을 넣어도 아무 문제가 없을 경우 타입 B는 타입 A의 하위 타입이라고 한다
- 하위 타입이 중요한 이유는 컴파일러가 변수 대입이나 함수 인자 전달 시 매번 하위 타입 검사를 수행하기 때문이다

```
fun test(i: Int) {
    val n: Number = i     // Int는 Number의 하위타입이라서 컴파일 O

    fun f(s: String) { TODO() }
    f(i)    // Int는 String의 하위타입이 아니라서 컴파일 X
}
```

non-null 타입은 nullable 타입의 하위 타입이지만 둘 다 같은 클래스이다

```
val s: String = "abc"
val t: String? s
```

- 항상 널이 될 수 없는 타입의 값을 널이 될 수 있는 타입의 변수에 저장할 수 있지만 거꾸로는 불가능하다

제네릭 타입에서 하위 클래스와 하위 타입의 차이

- 타입 안정성을 지키기 위해 MutableList<Any>와 MutableList<String>은 서로 하위 타입도 상위 타입도 아닌 무공변이다.
    - A가 B의 하위 타입이더라도 제네릭의 타입 인자로 전달된 경우엔 서로 관계가 없다
- 이런 무공변이 없었다면 원소 추가 및 변경 시 타입 불일치가 발생하면서 타입 안정성이 사라질 것이다
    - 그래서 자바와 코틀린의 경우 기본적으로 제네릭 타입을 무공변으로 만든다
    - 다만 코틀린의 읽기 전용 클래스는 타입 안전이 유지되기 때문에 공변적이다

공변성은 하위 타입 관계를 유지한다

- 공변성은 A가 B의 하위 타입일 때 `List<A>`는 `List<B>`의 하위 타입 관계로 유지되어 안전하게 사용할 수 있다
- 코틀린 제네릭 클래스가 타입 파라미터에 대해 공변적임을 표시하려면 타입 파라미터 앞에 `out` 키워드를 붙여야 한다

```
interface Producer<out T> {
    fun produce(): T
}
```

- 타입 파라미터를 공변적으로 만들면 함수 정의에 사용한 파라미터 타입과 타입 인자의 타입이 정확히 일치하지 않아도 그 클래스의 인스턴스를 함수 인자나 반환값으로 사용할 수 있다

공변적인 클래스를 사용해 하위 타입 관계를 유지해

```
open class Animal {
    fun feed() { TODO() }
}

class Herd<T: Animal> {
    val size: Int get() = TODO()
    operator fun get(i: Int): T = TODO()
}

fun feedAll(animals: Herd<Animal>) {
    for (i in 0 until animals.size) {
        animals[i].feed()
    }
}

class Cat: Animal() {
    fun cleanLitter() { TODO() }
}

fun takeCareOfCats(cats: Herd<Cat>) {
    for (i in 0 until cats.size) {
        cats[i].cleanLitter()
```

```
    }
    feedAll(cats) // Argument type mismatch: actual type is 'Herd<Cat>', but 'H
}
```

- 하위 타입이 유지되지 않아서 `Herd<Cat>`이 `Herd<Animal>`의 하위 클래스가 아니라서 `feedAll()` 호출 시 타입 불일치 에러가 발생한다
- 강제 타입 캐스팅으로 해결할 수 있긴 하지만 좋은 방법은 아니므로 읽기 전용인 경우 공변적인 클래스로 만들어주는 것이 좋다

```
class Herd<T: Animal> {
    val size: Int get() = TODO()
    operator fun get(i: Int): T = TODO()
}
```

- `Herd<Cat>`을 `Herd<Animal>`처럼 다뤄도 안전하므로 out 키워드를 통해 하위 타입 관계를 유지하도록 공변적인 클래스로 지정한다
- 이러면 feedAll() 호출 시 `Herd<Cat>`이 `Herd<Animal>`의 하위 클래스로 취급되어 컴파일 오류 없이 안전하게 작동한다

공변적으로 만들면 안전하지 못한 클래스도 존재한다

- 모든 제네릭 타입이 공변적(out)으로 될 순 없다
  - T를 인자로 받아 그 타입의 값을 반환하는 메서드가 있는 경우 공변적 클래스가 될 수 없다
  - 이게 된다면 공변성으로 하위 타입 관계가 유지됐는데 런타임에 타입 안전성이 깨질 수 있다
- 타입 안전성을 보장하기 위해 공변적 파라미터는 항상 out 위치에만 있어야 하고 이를 컴파일러가 강제한다
  - 즉, T 타입의 값을 반환할 수는 있지만 소비할 수는 없다

클래스 멤버 선언 시 타입 파라미터를 사용할 수 있는 지점은 모두 인/아웃 위치로 나뉜다

위치		

아웃 위치 (`out`)	- 함수 반환 타입 - 읽기 전용 프로퍼티	`fun get(index: Int): T`
인 위치 (`in`)	- 함수 매개변수 - 쓰기 가능 프로퍼티	`fun set(value: T)` 외부에서 T를 소비

생성자 파라미터는 인이나 아웃 위치 어느 쪽도 아니다

```
class Herd<out T: Animal>(vararg animals: T) { TODO() }
```

- 타입 파라미터가 out이라도 그 타입을 생성자 파라미터 선언에 사용할 수 있다

하지만 val이나 var 키워드를 생성자 파라미터에 적는 경우 게터나 세터를 정의하는 것과 같다

```
class Herd<T: Animal>(val leadAnimal: T, vararg animals: T) { TODO() }
```

- 인 위치에 프로퍼티가 있기에 out 키워드를 사용할 수 없다
- 읽기 전용 프로퍼티의 경우 타입을 소비하지 않기 때문에 공변성과 충돌하지 않아서 아웃 위치이고, 변경 가능 프로퍼티는 생산/소비가 모두 되기에 아웃/인 위치 모두에 해당한다

변성 규칙은 외부에서 볼 수 있는 클래스 API만 적용된다

```
class Herd<out T: Animal>(private val leadAnimal: T, vararg animals: T) { TOD
```

- 변성 규칙은 클래스 외부의 사용자가 클래스를 잘못 사용하는 일을 막기 위한 것으로 클래스 내부 구현에는 적용되지 않는다
- 그래서 비공개 메서드 파라미터는 인도 아니고 아웃도 아닌 위치라서 T에 out 키워드를 사용할 수 있다

반공변성은 하위 타입 관계를 뒤집는다

반공변성은 제네릭 타입의 타입 파라미터 앞에 `in` 키워드를 붙여서 타입 인자의 하위 타입 관계가 제네릭 타입에서는 반대로 적용되도록 만드는 것이다

```
interface Comparator<in T> {
    fun compare(e1: T, e2: T): Int { TODO() }
}
```

- in 키워드는 그 키워드가 붙은 타입이 그 클래스의 메서드 안으로 전달돼 메서드에 의해 소비된다는 뜻이다
- 따라서 `T`가 `in` 위치에만 쓰인다고 명시해주는 것으로 `T` 타입의 값을 소비하기만 한다

이 때 반공변성이기에 타입 안전성을 유지하면서 Comparator<Cat>에 Comparator<Animal>를 사용할 수 있다

```
sealed class Animal

sealed class Cat : Animal()

val animalComparator: Comparator<Animal> = Comparator<Animal> { a, b →

// 반공변성으로 Comparator<Animal>을 Comparator<Cat>로 사용 가능
val catComparator: Comparator<Cat> = animalComparator
```

이게 가능한 이유는 Comparator<Animal>이 두 Animal을 비교할 능력이 있을 때 더 구체적인 Cat을 비교하는 것도 문제없이 할 수 있기 때문이다

- 즉, 보다 일반적인 타입의 Consumer는(`Comparator<Animal>`) 더 구체적인 타입의 Consumer(`Comparator<Cat>`) 자리에 안전하게 들어갈 수 있기 때문에 하위 타입 관계가 반대가 되어도 타입 안전성을 해치지 않는 것이다
- 즉, T를 `in` 키워드를 통해 반공변성으로 만들면 더 일반적인(상위의) 타입을 안전하게 사용할 수 있게 된다

Animal, Cat의 공변/반공변 관계 도식

```
Cat → Animal

// 공변성 (out)
Producer<Cat> → Producer<Animal>
```

```
// 반공변성 (in)
Consumer<Animal> → Consumer<Cat>
```

공변성 vs 반공변성

공변성	반공변성
Producer<out T>	Consumer<in T>
타입 인자의 하위 타입 관계가 제네릭 타입에서도 유지	타입 인자의 하위 타입 관계가 제네릭 타입에서 반전
Producer<Cat> → Producer<Animal>	Consumer<Animal> → Consumer<Cat>
타입 인자 T를 out 위치에서만 사용 가능	타입 인자 T를 in 위치에서만 사용 가능

- 무공변성의 경우 타입 인자의 하위 타입 관계와 완전 무관하다

어쩔 땐 공변적이면서 어쩔 땐 반공변적일 수 있다?

- `Function` 인터페이스의 경우 각 파라미터마다 다른 변성을 가질 수 있다

```
interface Function1<in P, out R> {
    operator fun invoke(p: P): R
}
```

- `P`의 경우 입력 파라미터로 `in` 키워드와 함께 반공변성
- `R`의 경우 반환 타입으로 `out` 키워드와 함께 공변성

따라서 `Funtion1`의 하위 타입 관계는 `(P) → R`일 때 첫 타입 인자의 하위 타입 관계와는 반대(반공변성)이고 두 번째 타입 인자의 하위 타입 관계와는 같다(공변성)

```
fun enumerateCats(f: (Cat) → Number) { TODO() }
fun Animal.getIndex(): Int { TODO() }

fun main() {
```

```
    enumerateCats(Animal::getIndex)
}
```

- 이 코드가 가능한 이유가 `Animal`은 `Cat`의 상위 타입인데 반공변성인 `in` 위치라서 관계가 반전되고, `Int`의 경우는 `Number`의 하위 타입인데 공변성인 `out` 위치라서 관계가 유지되서 가능한 것이다.

사용 지점 변성과 선언 지점 변성

- 자바는 타입 파라미터가 있는 타입을 사용할 때마다 그 타입 파라미터를 하위 타입이나 상위 타입 중 어떤 타입으로 대치할 수 있는지 명시하는 사용 지점 변성을 사용해야 한다
    - 그래서 `Function`같은 인터페이스 사용 시 호출하는 곳에서 매번 `Function<? super T, ? extends R>`처럼 와일드 카드로 명시해야 한다
- 반면 코틀린은 클래스를 선언하면서 변성을 지정하는 선언 지점 변성을 지원하기 때문에 사용하는 모든 장소에 변성을 적용할 수 있어 매우 간결하다
- 물론 자바처럼 사용 지점 변성을 지원해 클래스 안에서 선언 지점 변성을 사용할 수 없는 경우 특정 파라미터가 나타나는 지점에서 변성을 정할 수 있다

컬렉션 복사 함수 copyData

```
fun <T: R, R> copyData(
    source: MutableList<T>,
    destination: MutableList<R>
) {
    for (item in source) {
        destination.add(item)
    }
}
```

- 무공변성인 가변 컬렉션 `MutableList<T>`를 다른 `MutableList<T>`로 복사할 때 원본 컬렉션은 읽기만 하고 대상 컬렉션은 쓰기만 한다
- 원본 컬렉션이 대상 컬렉션에 복사될 수 있으려면 원본 리스트 원소 타입은 복사 대상 리스트 원소 타입의 하위 타입이어야 해서 `<T: R, R>`로 타입 파라미터를 선언한 상태이다

사용 지점 변성 + 타입 프로젝션으로 리팩토링

```
fun <T> copyData(
    source: MutableList<out T>,
    destination: MutableList<T>
) {
    for (item in source) {
        destination.add(item)
    }
}
```

- 원본 컬렉션에서 타입 인자 `T`를 읽을수만 있도록 `out` 키워드로 제한할 수 있다
    - 이 경우는 타입 파라미터의 선언 위치(제네릭 타입 선언)에서는 그냥 `<T>`로 적었지만 함수의 사용 시점에서 변성 변경자 `out`을 적용한 경우이다
- 이렇게 제네릭 타입의 특정 사용 시점에서만 변성 제약을 주면 타입 프로젝션이 일어나면서 `MutableList<T>`가 `T`를 생산만 할 수 있고 소비는 할 수 없는 제약이 걸린 타입으로 만들어준다

> 사실 이 예제의 경우 MutableList의 공변성 버전인 List가 있기에 타입 프로젝션을 굳이 쓸 필요는 없지만 공변적/무공변적 인터페이스가 나눠져 있지 않은 경우엔 프로젝션이 유용하다

스타 프로젝션 `*`

스타 프로젝션 `*`은 제네릭 타입 인자 정보가 없음을 표현할 때 `List<*>`와 같이 표현하는 것이다

- 즉 타입 인자를 시그니처에서 전혀 언급하지 않거나 데이터를 사용하기는 하지만 구체적인 타입은 신경쓰지 않을 때 스타 프로젝션을 쓴다

```
fun printFirst(list: List<*>) {
    if (list.isNotEmpty()) {
        println(list.first()) // 먼 타입인지 상관없이 그냥 출력만 할 때
    }
}
```

```
fun main() {
    printFirst(listOf("Seb", "Roman"))
}
```

특징 1) 스타 프로젝션이라고 해서 그 안에 아무 원소나 다 담을 수 있는 것은 아니다

- `MutableList<*>`는 어떤 T인지 모르는 `MutableList<T>`지 이게 `MutableList<Any?>`가 될 순 없다

특징 2) 타입을 알 수 없는 `MutableList<*>`와 같은 리스트를 만들 수 없다

- 타입을 알 수 없는 리스트를 만들 수는 없고 단지 그런 알 수 없는 타입의 리스트를 받아 사용할 수 있을 뿐이다
- 어쨋든 진짜 원소 타입은 알 수 없지만 그 원소 타입이 Any?의 하위 타입은 분명하기에 사용은 할 수 있게 된다

특징 3) 컴파일러는 스타 프로젝션을 기본적으로 아웃 프로젝션 타입으로 인식한다

- 꺼내올 땐 Any?로 꺼내올 수 있지만 마음대로 넣을 수는 없기 때문에 아웃 프로젝션 타입으로 인식하는 것이다

스타 프로젝션을 제네릭 타입 파라미터로 우회할 수 있다

```
fun <T> printFirst(list: List<T>) {
    if (list.isNotEmpty()) {
        println(list.first())
    }
}
```

- 여기서 `T`는 어떤 타입인지 알지 못하는 상태겠지만 실제로는 `List<T>`로 명시되어 있어 타입 시스템의 제약을 적용받고 동시에 값을 안전하게 꺼내 쓸 수 있게 된다
- 이게 가능한 이유는 T가 함수 내부에서 읽기 전용 out 위치에서만 사용되고 있기에 코틀린이 T를 어떤 타입이든 안전하게 사용할 수 있다고 판단하기 때문이다

스타 프로젝션 사용 시 주의점

- 스타 프로젝션을 사용해 타입 인자가 서로 다른 제네릭 클래스들을 담으면 나중에 꺼내 쓸 때 컴파일러는 어떤 타입에 해당하는 제네릭 클래스인지 몰라서 에러를 낸다
- 강제로 캐스팅해서 사용할 수는 있겠지만 좋은 방법은 아니다

강제로 캐스팅해서 사용하는 예제

```
import kotlin.reflect.KClass

interface FieldValidator<in T> {
    fun validate(input: T): Boolean
}

object DefaultStringValidator : FieldValidator<String> {
    override fun validate(input: String) = input.isNotBlank()
}

object DefaultIntValidator : FieldValidator<Int> {
    override fun validate(input: Int) = input >= 0
}

fun main() {
    val validators = mutableMapOf<KClass<*>, FieldValidator<*>>()
    validators[String::class] = DefaultStringValidator
    validators[Int::class] = DefaultIntValidator

    // 그냥 사용 시 컴파일러 에러
    validators[String::class]!!.validate("") // Argument type mismatch: actual typ

    // 강제 캐스팅 사용
    val stringValidator = validators[String::class]!! as FieldValidator<String> // u
    println(stringValidator.validate("Hello, World!")) // true
}
```

스타 프로젝션 없이 검증기 등록 및 접근 시 타입을 제대로 검사하도록 캡슐화

```
object Validators {
    private val validators = mutableMapOf<KClass<*>, FieldValidator<*>>()

    fun <T : Any> registerValidator(type: KClass<T>, validator: FieldValidator<T
        validators[type] = validator
    }

    @Suppress("UNCHECKED_CAST")
    fun <T : Any> getValidator(type: KClass<T>): FieldValidator<T> {
        return validators[type] as? FieldValidator<T>
            ?: throw IllegalArgumentException("No validator registered for type: $t
    }
}

fun main() {
    Validators.registerValidator(String::class, DefaultStringValidator)
    Validators.registerValidator(Int::class, DefaultIntValidator)

    val stringValidator = Validators.getValidator(String::class)
    println(stringValidator.validate("Hello")) // true

    val intValidator = Validators.getValidator(Int::class)
    println(intValidator.validate(-5)) // false
}
```

- 실제 등록과 조회를 제네릭 타입 T로 정확한 타입 검사를 통해 수행되기 때문에 항상 타입에 맞게 검증기를 등록하고 올바른 검증기를 돌려준다
- 안전하지 못한 캐스팅 로직을 전부 내부로 캡슐화해 외부에서 잘못 사용할 수 없도록 보장한다

타입 별명

- 타입 컬렉션의 목적이 무엇인지 떠올리기 힘들거나 여러 곳에서 매번 반복해 사용하는 것이 힘들 때 `typealias` 키워드로 별명을 지어줄 수 있다

```
typealias UserNameWithEmail = Pair<String, String>
typealias UserNameWithEmailList = List<UserNameWithEmail>

val users: UserNameWithEmailList = listOf(
    "Alice" to "alice@example.com",
    "Bob" to "bob@example.com"
)
```

- 하지만 익숙하지 않은 경우 이 타입 별명의 정의를 찾아다녀야 한다는 점도 생각해야 한다

인라인 클래스 vs 타입 별명

타입 별명은 말그대로 별명으로 컴파일 시점에 원래 타입으로 바꿔주기만 할 뿐 아무것도 추가로 보장해주지 않는다

```
typealias ValidatedInput = String

fun save(v: ValidatedInput): Unit = TODO()

fun main() {
    val input: ValidatedInput = "needs validating!"
    save(input)
}
```

반면 인라인 클래스의 타입은 다른 일반적인 타입과 동일하게 검사되기 때문에 타입 안전성이 있다

```
@JvmInline
value class ValidatedInput(val s: String)

fun save(v: ValidatedInput): Unit = TODO()

fun main() {
    val input = ValidatedInput("needs validating!")
```

```
    save(input)
}
```

- `save()` 사용 시 일반 `String`으론 컴파일이 되지 않고 반드시 래핑해야 되기 때문에 안전하다
- 여기에 `init` 블록까지 사용할 수 있기 때문에 유효성 검증까지 할 수 있다