6장 코틀린 타입 시스템

신림프로그래머 최범균

- 널이 될 수 있는 타입
- 널 처리 구문
- 코틀린 원시 타입
- 코틀린 콜렉션과 자바 콜렉션

null 가능성: null이 될 수 있는 타입

코틀린은 널이 될 수 있는 타입과 널이 될 수 없는 타입을 명시적으로 지원

- null과 관련된 문제를 가능한 실행 시점에서 컴파일 시점으로 옮김

null이 될 수 있는 타입

- 타입 이름 뒤에 물음표를 붙임
- 프로퍼티나 변수에 null 가능 타입 지정 가능

```
fun strLen(s: String?) = if (s != null) s.length else 0

strLen(null)
strLen("10")
```

null 가능성: null이 될 수 없는 타입

null이 될 수 없는 타입

- 타입 뒤에 물음표가 없으면 null을 값으로 가질 수 없는 타입. null을 할당하면 컴파일 에러.

```
// s는 항상 String 인스턴스. null일 수 없음
fun strLen(s: String) = s.length

strLen(null) // 컴파일 에러: Null can not be a value of a non-null type String
strLen("10") // OK!

// null 불가 타입에, null 가능 타입 할당 불가!
val x: String? = null
val y: String = x // 컴파일 에러
```

null 가능 타입을 위한 연산자: 안전 호출 ?.

null 검사와 메서드 호출을 한 번의 연산으로 수행

- 호출하는 값이 null이 아니라면 일반 메서드 호출처럼 작동
- 호출하려는 값이 null이면 호출을 무시하고 null이 결과 값이 됨

```
val s: String? = ...
val s1: String? = s?.toUpperCase()
val s2: String? = if (s != null) s.toUpperCase() else null

class Employee(val name: String, val manager: Employee?)

val mname = employee.manager?.name
val country = company?.address?.country // 안전 호출 연쇄
```

null 가능 타입을 위한 연산자: 엘비스 연산자 ?:

좌항 값이 널이 아니면 좌항 값을 결과로 하고, 좌항 값이 널이면 우항 값을 결과로 함

```
fun Person.countryName(): String {
    return this.company?.address?.country ?: "Unknown"
}

val addr = person.company?.address ?: throw IllegalArgumentException("No address")
```

안전한 캐스트 as?

지정한 타입으로 캐스트하는데, 대상 타입으로 변환할 수 없으면 null을 리턴

```
fun equals(o: Any?): Boolean {
    // val other: Person? = o as? Person
    val other: Person = o as? Person ?: return false
    return ...
}
```

널 아님 단언: !!

어떤 값이든 널이 될 수 없는 타입으로 바꿈. 값 null이면 NPE 발생.

```
fun ignoreNulls(s: String?) {
    val notNull: String = s!! // s가 null이면 Kotlin NPE 발생
    ...
}
```

let 함수

- 자신의 수신 객체를 인자로 전달받은 람다에게 넘김
- 널 가능 타입을 널이 될 수 없는 타입은 인자로 받는 함수에 전달할 때 유용

```
fun sendEmail(email: String) { ... }

// let의 수신 객체 v는 not null
val email: String? = ...

email?.let { v: String -> sendEmail(v) }

// email?.let { sendEmail(it) }
// if (email != null) sendEmail(email)

// let의 수신 객체 v는 nullable
email.let { v: String? -> println("이메일 $v") }
```

lateinit 프로퍼티

- **널이 될 수 없는 프로퍼티**를 생성 시점이 아닌 나중에 초기화할 수 있게 함

```
class MyTest {
    private lateinit var myService: MyService

    @Before fun setUp() {
        myService = MyService()
    }
}
```

- 프로퍼티는 var여야 함
- lateinit 프로퍼티를 초기화 전에 접근하면 익셉션 발생

널 가능 타입의 확장

- 널 가능 타입에 확장을 정의하면 널이 될 수 있는 값에 대해 그 확장 함수를 호출할 수 있음
  - 확장 함수 안에서 this의 null 여부 검사하므로, 안전한 호출 필요 없음

```
fun CharSequence?.isNullOrBlank(): Boolean { ... }

val str: String? = ...
str.isNullOrBlank() // 안전안 호출(?.)을 하지 않아도 됨
```

> 처음에는 널이 될 수 없는 타입에 확장 함수를 정의하고
> 진짜 널 가능 타입에 대해 확장 함수가 필요하면 그때 추가

지네릭 타입 파라미터의 널 가능성

- 모든 타입 파라미터는 기본적으로 널 가능
- 널 가능 타입과 널 불가 타입 모두 타입 파라미터 인자로 사용 가능

```
fun <T> printHashCode(t: T) { // t는 Any?로 추론
    println(t?.hashCode()) // t.hashCode()는 컴파일 에러.
}
```

```
// 타입 파라미터의 상한 지정
fun <T: Any> printHashCode(t: T) { // t는 Any 타입으로 추론
    println(t.hashCode())
}
```

자바의 널 가능성 애노테이션

자바 코드에 널 가능성 애노테이션을 사용하면, 코틀린은 그 정보를 활용

- @Nullable String -> String?
- @NotNull String -> String

JSR-305 표준, 안드로이드, 젯브레인 애노테이션 등의 널 가능성 애노테이션 지원

플랫폼 타입

널 관련 정보를 알 수 없는 자바 타입

- 널 가능 타입으로 처리해도 되고 널 불가 타입으로 처리해도 됨
- 널 가능성을 코틀린 코드에서 알맞게 처리해야 함

```
public class Person {
    private final String name;

    public String getName() {
        return name;
    }
    ...
}
```

```
fun yellAt(p: Person) {
    // name의 플랫폼 타입을 널 불가 타입처럼 처리한 예
    // person.name 사용 전에 코틀린 컴파일러가 null 검사 코드 추가
    println(person.name.toUpperCase() + "!!!")
}
// person.name이 null일 때 익셉션 발생
Exception in thread "main" java.lang.IllegalStateException:
person.name must not be null
```

```
fun yellAtSafe(p: Person) {
    // name의 플랫폼 타입을 널 가능 타입처럼 처리한 예
    println( (person.name ?: "Anyone").toUpperCase() + "!!!")
}
```

자바 클래스를 코틀린에서 상속

- 메서드 파라미터와 리턴 타입을 널 가능 타입이나 널 불가 타입으로 선언할지 결정

```
interface StringProcessor {
    void process(String value);
}
```

```
class StringPrinter : StringProcessor {
    override fun process(value: String) { // 널 불가로 선언
        // 컴파일러가 널이 아님을 검사하는 단언 추가
        println(value)
    }
}

class NullableStringPrinter : StringProcessor {
    override fun process(value: String?) { // 널 가능으로 선언
        value.?let { println(it) }
    }
}
```

```
val sp: StringProcessor = StringPrinter()

// sp.process -> process(value: String!)
sp.process(null)

// Exception in thread "main" java.lang.IllegalArgumentException:
// Parameter specified as non-null is null:
// method chap06.StringPrinter.process, parameter value
```

원시 타입

- 실행 시점에 숫자 타입은 가능한 한 효율적인 방식으로 표현
  - 대부분의 경우 코틀린 숫자 타입은 자바 원시 타입으로 컴파일
- 널 가능 원시 타입은 랩퍼 타입으로 컴파일
  - 코틀린 Int? -> 자바 Integer
- 지네릭 클래스의 경우 래퍼 타입 사용

숫자 변환

- 한 숫자 타입을 다른 타입 숫자로 자동 변환하지 않음
  - 명시적으로 변환해야 함
- 숫자 리터럴은 컴파일러가 필요한 변환을 자동으로 처리
- 연산자를 각 타입에 대해 오버로딩하고 있음

```
val b: Byte = 1
val l = b + 1L
val l1: Long = 42 // Int 타입 리터럴을 Long으로 컴파일러가 변환
val i1: Int = 42
// val l2: Long = i1 // 컴파일 에러
val l2: Long = i1.toLong()
```

최상위 타입

- Any: Int 등 원시 타입을 포함한 모든 널 불가 타입의 조상 타입

```
val answer: Any = 42
```

- 내부에서 Any는 Object에 대응
  - 코틀린에서 Any를 사용하면 자바 바이트코드의 Object로 컴파일
- Any: toString, equals, hashCode 세 개의 메서드 제공
  - wait, notify 등은 제공하지 않음

Unit 타입

- Unit 타입은 자바의 void와 유사
  - 관심을 가질 만한 내용을 리턴하지 않는 함수의 리턴 타입으로 Unit 사용
  - 리턴 타입 선언 없이 정의한 블록 몸체를 가진 함수는 Unit이 리턴 타입
- 함수 반환 타입이 Unit이고, 지네릭 함수를 재정의한 것이 아니면, 내부적으로 자바 void 함수로 컴파일
- Unit은 void와 달리 타입, Unit이라는 단일 값을 가짐
  - 리턴 타입이 Unit인 함수는 묵시적으로 Unit을 리턴

```
interface Processor<T> {
    fun process(): T
}
class NoResultProcessor: Processor<Unit> {
    override fun process() {
        // 컴파일러가 return Unit을 넣음
    }
}
```

Nothing 타입

- 함수가 정상적으로 끝나지 않는다는 사실을 표현

```
fun fail(message: String): Nothing {
    throw IllegalStateException(message)
}
```

- Nothing 타입은 아무 값도 포함하지 않음
- 함수의 리턴 타입이나 리턴 타입으로 쓰일 타입 파라미터로만 사용 가능

코틀린 콜렉션

인터페이스 분리

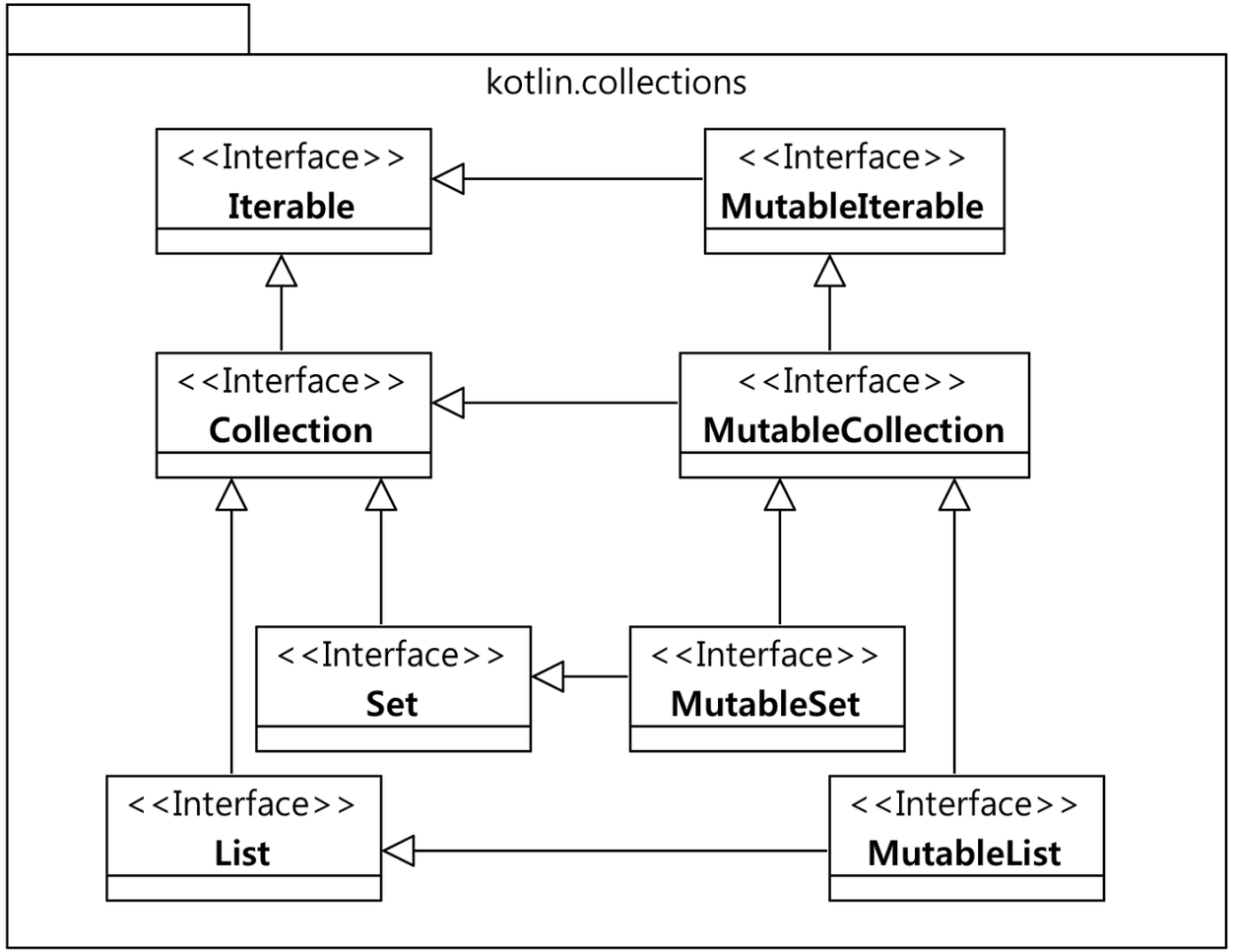
- kotlin.collections.Collection : 조회 기능만 제공
- kotlin.collections.MutableCollection : 수정 기능도 제공

주의

- 읽기 전용 콜렉션 인터페이스는 인터페이스만 읽기 전용이고, 실제 구현체는 수정 가능할 수 있음

코틀린 콜렉션과 자바

- 코틀린 콜렉션은 읽기 전용 인터페이스와 변경 가능 인터페이스로 제공
  - 실제 코틀린 콜렉션은 그에 상응하는 자바 컬렉션 인터페이스의 인스턴스
  - 따라서 코틀린과 자바 사이를 오갈 때 변화 필요 없음
- 실제 내부적으로는 자바 구현체 사용



생성 함수

타입	읽기 전용	변경 가능 타입
List	listOf	metableListOf, arrayListOf
Set	setOf	mutableSetOf, hashSetOf, linkedSetOf, sortedSetOf
Map	mapOf	mutableMapOf, hashMapOf, linkedMapOf, sortedMapOf

주의: 실제 사용되는 자바 컬렉션 인스턴스는 변경 가능하므로 주의

코틀린 콜렉션을 자바 메서드에 전달

- 코틀린 읽기 전용 콜렉션을 자바 메서드에 전달해도 자바에서는 수정 가능함에 주의

```
static List<String> uppercaseAll(List<String> items) { // 자바에서는 수정 가능
    for (int i = 0 ; i < items.size() ; i++) {
        items.set(i, items.get(i).toUpperCase());
    }
    return items;
}
```

```
fun printInUppercase(list: List<String>) { // 읽기 전용이지만
    println(uppercaseAll(list))
    println(list.first())
}
```

자바의 콜렉션 타입을 플랫폼 타입으로 다루기

- 자바의 콜렉션 타입은 읽기 전용이나 변경 가능으로 다룰 수 있음
- 시그너처에서 콜렉션 타입을 사용한 자바 메서드를 오버라이드할 경우 코틀린 타입 선택 필요
  - 선택 사항
    - 컬렉션이 널이 될 수 있는가?
    - 컬렉션의 원소가 널이 될 수 있는가?
    - 오버라이드하는 메서드가 컬렉션을 변경할 수 있는가?
  - 자바 인터페이스나 클래스가 어떤 맥락에서 사용되는지 정확히 알아야 함

배열

- Array 타입으로 배열 정의

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

- 배열 생성 방법
  - val num: Array = arrayOf(1, 2, 3, 4)
  - val nulls = arrayOfNulls(10) // Array<String?>
  - val nulls2 = Array(20) { i -> (i + 1).toString() }
- collection.toTypedArray()로 배열로 변경

원시 타입 전용 배열

- IntArray, CharArray, BooleanArray 등 원시 타입 전용 배열 제공
  - 자바 원시 타입 배열로 컴파일되어 성능에서 이점
- 배열 생성 방법
  - IntArray(5) // 길이가 5이고 기본 값으로 초기화
  - intArrayOf(1, 2, 3, 4)
  - IntArray(5) { i -> i * i }
- Array와 같은 타입은 toIntArray 등 함수를 이용해서 IntArray로 변환 가능

배열의 forEach

```
numbers.forEach { println(it) }

numbers.forEachIndexed { index, element -> println("Arg $index is : $element")}
```