Chapter08. 기본 타입, 컬렉션, 배열

8장에서 다루는 내용

- 원시 타입과 다른 기본 타입 및 자바 타입과의 관계
- 코틀린 컬렉션과 배열 및 이들의 널 가능성과 상호운용성

원시 타입과 기본 타입

- 자바와 달리 코틀린은 원시 타입과 래퍼 타입을 구분하지 않는다.

정수, 부동소수점 수, 문자, 불리언 값을 원시 타입으로 표현

- 자바는 원시 타입과 참조 타입을 구분한다.
    - 원시 타입의 변수에는 그 값이 직접 할당

    - 참조 타입의 변수에는 메모리상의 객체 위치가 들어간다.
- 원시 타입의 값을 더 효율적으로 저장하고 전달할 수 있지만, 그런 값에 대해 메서드를 호출하거나 컬렉션에 원시 타입 값을 담을 수 없다.
    - 따라서 자바는 참조 타입이 필요한 경우 래퍼타입으로 원시 타입 값을 감싸서 사용한다.
- 코틀린은 원시 타입과 래퍼 타입을 구분하지 않는다.
    - 그럼 항상 객체로 표현하는 것일까?
- 실행 시점에 숫자 타입은 가능한 한 가장 효율적인 방식으로 표현된다.
    - 대부분의 경우(변수, 프로퍼티, 파라미터, 반환 타입 등) 코틀린의 Int 타입은 자바 int 타입으로 컴파일 된다.
    - 이런 컴파일이 불가능한 경우는 컬렉션과 같은 제네릭 클래스를 사용하는 경우뿐이다. 그 때에는 래퍼 객체가 들어간다.
        - eg) Collection<Int>

양수를 표현하기 위해 모든 비트 범위 사용: 부호 없는 숫자 타입

- 양수를 표현하기 위해 모든 비트 범위를 사용하는 경우
    - 비트와 바이트 수준 작업
    - 비트맵 픽셀 조작
    - 파일에 담긴 바이트들이나 다른 2진 데이터 다룰 때
- 이런 경우 코틀린은 jvm의 일반적인 원시 타입을 확장해 부호 없는 타입을 제공한다.
- 다른 원시 타입과 마찬가지로 코틀린의 부호 없는 수도 필요할 때만 래핑된다.
- 명시적으로 전체 비트 범위가 필요한 경우가 아니라면 일반적으로 보통의 정수를 사용하고 음수 범위가 함수에 전달됐는지 검사하는 편이 낫다.
    - Q) 왜?
    - jvm 자체는 부호가 없는 수에 대한 원시 타입을 지정하고나 제공하지 않는다.
    - 코틀린은 이를 바꿀 수 없기 때문에 기존의 부호가 있는 원시 타입 위에 자체적인 추상화를 제공한다.
        - 인라인 클래스 사용

- 따라서 UInt 값은 실제로 Int이기 때문에 코틀린 컴파일러가 가능할 때마다 인라인 클래스를 프로퍼티로 대신하고나 래핑하는 등의 처리를 해준다.

Type	Size (bits)	Min value	Max value
UByte	8	0	0 ~ 255
UShort	16	0	0 ~ 65,535
UInt	32	0	0 ~ 4,294,967,295 (2^32 - 1)
ULong	64	0	0 ~ 18,446,744,073,709,551,615 (2^64 - 1)

널이 될 수 있는 기본 타입: Int? Boolean? 등

- null 참조를 자바의 참조 타입의 변수에만 대입할 수 있기 때문에 널이 될 수 있는 코틀린 타입은 자바 원시 타입으로 표현할 수 없다.
- 따라서 코틀린에서 널이 될 수 있는 원시 타입을 사용하면 그 타입은 자바의 래퍼 타입으로 컴파일된다.
- 위에서 말한대로 제네릭 클래스의 경우에도 래퍼 타입을 사용한다.
  - jvm에서 제네릭을 구현하는 방법 때문, jvm은 타입 인자로 원시 타입을 허용하지 않는다.
  - 따라서 자바, 코틀린 모두 제네릭 클래스는 항사 박스 타입 사용
- 원시 타입으로 이뤄진 대규모 컬렉션을 저장해야 한다면 서드 파티 라이브러리를 사용하거나 배열을 사용해야 한다.
  - 이클립스 컬렉션즈

수 변환

```
val i = 1
val l: Long i // 컴파일 오류

val i = 1
val l: Long = i.toLong()
```

- 코틀린과 자바의 큰 차이점 중 하나

- 코틀린은 한 타입의 수를 다른 타입의 수로 자동 변환하지 않는다.
    - 결과 타입이 허용하는 수의 범위가 원래 타입의 범위보다 넓은 경우조차도 자동 변환 불가능
- 코틀린은 모든 원시 타입(Boolean 제외)에 대하 변환 함수를 제공한다.
    - 양방향 변환 함수가 모두 제공된다.
    - eg) `Long.toInt()`

```
new Integer(42).equals(new Long(42)) // false
```

```
// 만약 코틀린에서 암시적 변환을 허용한다면
val x = 1
val list = listOf(1L, 2L, 3L)
x in list // false

// 명시적 변환
x.toLong() int list // true
```

- 개발자의 혼란을 피하고자 타입 변환을 명시하기로 결정
    - 특히 박스 타입을 비교하는 경우 문제가 많다.
- 숫자 리터럴을 사용할 때는 보통 변환 함수를 호출할 필요가 없다.
    - 직접 변환하지 않더라도 숫자 리터럴을 타입이 알려진 변수에 대입하거나 함수에게 인자로 넘기면 컴파일러가 필요한 변환을 자동으로 넣어준다.

```
fun printALong(l: Long) = println(l)

fun main() {
    val b: Byte = 1 // 상수 값은 적절한 타입으로 해석된다.
    val l = b + 1L // +는 Byte와 Long 인자로 받을 수 있다.
    printALong(42) // 42
}


fun main() {
```

```
    println(Int.MAX_VALUE + 1) // -2147483648, 오버플로

    println(Int.MIN_VALUE - 1) // 2147483647, 언더 플로
}
```

- 산술 연산자는 적당한 타입의 값을 받아들일 수 있도록 이미 오버로드돼 있다.
- 자바와 똑같이 숫자 연산 시 오버플로나 언더플로가 발생할 수 있다.

```
fun main() {
    println("42".toInt())
}
```

- 문자열을 원시 타입으로 변환하는 여러 함수를 제공한다.
- 변환에 실패하면 null을 돌려주는 함수들도 존재

Any와 Any?: 코틀린 타입 계층의 뿌리

- 자바에서 Object가 클래스 계층의 최상위 타입이듯 코틀린에서는 Any 타입이 모든 널이 될 수 없는 타입의 조상 타입
    - 하지만 자바는 참조 타입만 타입 계층에 포함되며, 원시 타입은 그런 계층에 들어있지 않다.
    - 래퍼 클래스 감싸야 하는 이유
- 코틀린에서는 Any가 Int등의 원시 타입을 포함한 모든 타입의 조상 타입

```
val answer: Any = 42 // Any가 참조 타입이기 때문에 42가 박싱된다.
```

- 코틀린에서도 원시 타입 값을 Any 타입의 변수에 대입하면 자동으로 값을 박싱한다.
- Any는 널이 될 수 없는 타입이다.
    - 널을 포함하는 모든 값을 대입할 변수를 선언하려면 Any?
- 내부에서 Any 타입은 Object에 대응한다.
    - Obejct → Any!(플랫폼 타입)

- Any → Objecy

Unit 타입: 코틀린의 void

- 자바의 void와 같은 기능
- 코틀린 함수의 반환타입이 Unit이고 그 함수가 제네릭 함수를 오버라이드하지 않는다면 내부에서 자바 void 함수로 컴파일된다.

```
interface Processor<T> {
    fun process(): T
}

class NoResultProcessor: Processor<Unit> {
    override fun process() { // Unit을 반환하지만 타입을 지정할 필요는 없다.
                                        // 여기서 명시적으로 return할 필요가 없다. 컴파일
    }
}
```

- 그럼 차이점은?
  - Unit은 모든 기능을 갖는 일반적인 타입
  - void와 달리 타입 인자로도 사용 가능
- 이 두 특성은 제네릭 파라미터를 반환하는 함수를 오버라이드허면서 반환 타입으로 Unit을 쓸때 유용하다.

Nothing 타입: 이 함수는 결코 반환되지 않는다.

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

fun main() {
    fail("Error occurred")
```

```
    val address = company.adress ?: fail("no address")
    println(address.city) // address가 널이 아님을 알 수 있음
}
```

- 코틀린에는 결코 성공적으로 값을 돌려주는 일이 없으므로 '반환값'이라는 개념 자체가 의미가 없는 함수가 일 부 존재한다.
- 그런 경우를 표현하고자 Nothing이라는 특별한 반환 타입 존재
- 컴파일러는 Nothing 반환 타입인 함수가 결코 정상 종료되지 않음을 알고 그 함수를 호출하는 코드를 분석할 때 사용한다.

컬렉션과 배열

- 자바와 코틀린 컬렉션 간의 관계에 대해 더 이야기해보자

널이 될수 있는 값의 컬렉션과 널이 될 수 있는 컬렉션

```
fun readNumbers(text: String): List<Int?> {
    val result: MutableList<Int?> = mutableListOf<Int?>()
    for (line in text.lineSequence()) {
        val numberOrNull = line.toIntOrNull()
        result.add(numberOrNull)
    }
    return result
}

fun addValidNumbers(numbers: List<Int?>) {
    val validNumbers: List<Int> = numbers.filterNotNull()
    println("Sum of valid numbers: ${validNumbers.sum()}")
    println("Invalid numbers: ${numbers.size - validNumbers.size}")
}
```

읽기 전용과 변경 가능한 컬렉션

```
fun <T> copyElements(source: Collection<T>,
                     target: MutableCollection<T>) {
    for (item in source) {
        target.add(item)
    }
}

fun main() {
    val source: Collection<Int> = arrayListOf(3, 5, 7)
    val target: MutableCollection<Int> = arrayListOf(1)
    copyElements(source, target)
    println(target)

    val source2: Collection<Int> = arrayListOf(3, 5, 7)
    val target2: Collection<Int> = arrayListOf(1)
    copyElements(source2, target2) // 컴파일 오류
}
```

- 코틀린에서는 컬렉션 안에 데이터에 접근하는 인터페이스와 컬렉션 안의 데이터를 변경하는 인터페이스를 분리했다.
    - kotlin.collections.Collection
    - kotlin.collections.MutableCollection
    - 코틀린 컬렉션과 자바 컬렉션을 나누는 가장 중요한 특성 중 하나
- 실제 객체가 변경 가능한 컬렉션이더라 하더라도 target에 해당하는 인자로 읽기 전용 컬렉션 타입의 값을 넘길 수 없다.
- 핵심은 읽기 전용 컬렉션이더라도 꼭 변경 불가능한 컬렉션일 필요는 없다.
    - 읽기 전용 컬렉션이 항상 thread safe하지 않다는 점을 명시해야한다.
    - 실제로는 내부에서 변경 가능한 컬렉션을 가리킬 수 있기 때문
- 불변 컬렉션 라이브러리 참고
    - kotlinx.collections.immutable
- 코틀린은 모든 자바 컬렉션 인터페이스마다 읽기 전용 인터페이스와 변경 가능한 인터페이스라는 두가지 표현을 제공한다.

```
/* Java */
// CollectionUtils.java
public class CollectionUtils {
    public static List<String> uppercaseAll(List<String> items) {
        for (int i = 0; i < items.size(); i++) {
            items.set(i, items.get(i).toUpperCase());
        }
        return items;
    }
}
```

```
// Kotlin
// collections.kt
fun printlnUppercase(list: List<String>) {
    println(CollectionUtils.uppercaseAll(list))
    println(list.first())
}

fun main() {
    val list = listOf("a", "b", "c")
    printlnUppercase(list)
}
```

- 컬렉션을 메소드 인자로 넘겨야 한다면 따로 변환/복사 등의 추가 작업 없이 직접 컬렉션을 넘기면 된다.
    - 이런 성질로 인해 컬렉션의 변경 가능성과 관련해 중요한 문제가 생긴다.
    - 책임은 우리에게 있다.
    - 자바는 읽기 전용 컬렉션과 변경 가능 컬렉션을 구분하지 않기 때문
- 이런 함장은 널이 아닌 원소로 이뤄진 컬렉션 타입에도 발생한다.

자바에서 선언한 컬렉션은 코틀린에서 플랫폼 타입으로 보임

- 자바 코드에서 정의한 타입을 코틀린에서는 플랫폼 타입으로 본다.

    - 플랫폼 타입의 경우 코틀린 쪽에는 널 관련 정보가 없다.
    - 보통은 원하는 동작이 그냥 잘 수행될 가능성이 높으므로 실제 문제가 되지는 않는다.
- 컬렉션 타입이 시그니처에 들어간 자바 메소드 구현을 오버라이드하려는 경우 읽기 전용 컬렉션과 변경 가능 컬렉션의 차이가 문제가 된다. 이런 상황에서는 여러 가지를 선택해야 한다.
    - 컬렉션이 널이 될 수 있는가?
    - 컬렉션의 원소가 널이 될 수 있는가?
    - 오버라이드하는 메소드가 컬렉션을 변경할 수 있는가?

```
interface FileContentProcessor {
    void processContents(File path, byte[] binaryCOntents, List<String> textCo
}
```

- 이 인터페이스를 코틀린으로 구현하려면 다음을 선택해야 한다.
    - 일부 파일은 이진 파일이며 이진 파일 안의 내용은 텍스트로 표현할 수 없는 경우가 있으므로 리스트는 널이 될 수 있다.
    - 파일의 각 줄은 널일 수 없으므로 이 리스트의 원소는 널이 될 수 없다.
    - 이 리스트는 파일의 내용을 표현하며 그 내용을 바꿀 필요가 없으므로 읽기 전용이다.

```
// FileContentProcessor를 코틀린으로 구현한 모습
class FileIndexer : FileContentProcessor {
    override fun processContents(path: File,
        binaryContents: ByteArray?,
        textContents: List<String>?) {
            ...
    }
}
```

- 코틀린으로 구현

성능과 상호운용을 위해 객체의 배열이나 원시 타입의 배열을 만들기

- 코틀린 배열은 타입 파라미터를 받는 클래스다.
- 배열의 원소 타입은 바로 그 타입 파라미터에 의해 정해진다.
- 배열 생성 방법
  - arrayOf 함수에 원소를 넘기면 배열을 만들 수 있다.
  - arrayOfNulls 함수에 정수 값을 인자로 넘기면 모든 원소가 null이고 인자로 넘긴 값과 크기가 같은 배열을 만들 수 있다.
    - 물론 원소 타입이 널이 될 수 있는 타입인 경우에만 쓸 수 있다.
  - Array 생성자는 배열 크기와 람다를 인자로 받아서 람다를 호출해서 각 배열 원소를 초기화해준다.
    - arrayOf를 쓰지 않고 각 원소가 널이 아닌 배열을 만들어야 하는 경우 이 생성자를 사용한다.

```
fun main() {
    val strings = listOf("a", "b", "c")
    println("%s/%s/%s".format(*strings.toTypedArray()))
}
```

- 코틀린은 배열을 인자로 받는 자바 함수를 호출하거나 vararg 파라미터를 받는 코틀린 함수를 호출하기 위해 배열을 자주 선언.
  - 이 때 toTypedArray 메소드를 사용하면 컬렉션을 배열로 변경 가능
- 제네릭 타입처럼 배열 타입의 타입인자도 항상 객체 타입이 된다.
  - Array<Int>
- 박싱하지 않은 원시 타입의 배열이 필요하다면 그런 타입을 위한 특별한 배열 클래스를 사용해야 한다.
  - IntArray, ByteArray, CharArray, BooleanArray 등
  - int[], byte[], char[] 등으로 컴파일 된다.
- 원시 타입 배열 생성 방법

    - 각 배열 타입의 생성자는 size 인자를 받아서 해당 원시 타입의 디폴트 값(보통 0)으로 초기화된 size 크기의 배열을 반환한다.
    - 팩토리 함수는 여러 값을 가변 이자로 받아 그런 값이 들어간 배열을 반환한다.
    - 크기와 람다를 인자로 받는 다른 생성자를 사용한다.
- 박싱된 값이 들어있는 컬렉션이나 배열은 toIntArray 등의 변환 함수를 사용해 원시 타입의 배열로 변환 가능하다.

```
fun main(args: Array<String>) {
    args.forEachIndexed { index, element ->
        println("Argument $index is: $element")
    }
}
```

- 코틀린 표준 라이브러리는 배열 기본 연산에 더해 컬렉션에 사용할 수 있는 모든 확장 함수를 배열에도 제공
- 원시 타입인 원소로 이뤄진 배열에도 그런 확장 함수를 똑같이 사용할 수 있다.
    - 응답값은 리스트이다.

요약

- 기본적인 수를 표현하는 타입은 일반 클래스처럼 보이고 동작하지만 보통 자바의 원시 타입으로 컴파일된다.
- 코틀린의 부호 없는 수 클래스는, jvm에는 상응하는 타입이 없는데, 인라인 클래스를 통해 변환되며 원시 타입과 마찬가지 성능을 낸다.
- 널이 될 수 있는 원시 타입은 자바의 박싱된 원시 타입에 대응된다.
- Any 타입은 모든 다른 타입에 상위 타입이며 자바 Object 타입에 대응한다.
- Unit 타입은 void에 대응한다.
- Nothing 타입은 함수가 정상적으로 끝나지 않는다는 것을 나타내는 타입이다.
- 자바에서 온 타입은 코틀린에서 플랫폼 타입으로 취급된다.

    - 개발자는 이를 널 가능, 불가능 타입 모두 취급 가능하다.
- 코틀린은 컬렉션에 대해 표준 자바 클래스를 사용하지만 읽기 전용과 변경 가능한 컬렉션을 구분함으로써 컬렉션을 더 개선했다.
- 코틀린에서 자바 클래스를 확장하거나 자바 인터페이스를 구현해야 한다면 파라미터의 널 가능성과 변경 가능성을 주의 깊게 생각해야 한다.
- 코틀린에서도 배열을 사용할 수 있다.
    - 하지만 일반적으로는 컬렉션을 사용하는 쪽을 더 권장한다.
- 코틀린 Array는 일반적 제네릭 클래스처럼 보이지만 자바 배열로 컴파일된다.
- 원시 타입의 배열은 IntArray 등의 특별한 클래스에 의해 표현된다.