07. 다양한 클래스와 인터페이스 (297-347)

▼ 07-1 추상 클래스와 인터페이스

- 추상 클래스 : 선언 등의 대략적인 설계 명세와 공통의 기능을 구현나 클래스
  - 상속하는 하위 클래스에서 구체화해야함
  - 인터페이스와 비슷하지만 인터페이스와는 달리 프로퍼티에 상태 정보를 저장할 수 있음

추상 클래스

- 추상 클래스의 정의와 구현
  - abstract class Vehicle
  - 프로퍼티나 메서드도 abstract 선언 가능
  - 기본 프로퍼티나 메서드는 특정 초기화나 구현 필요 / abstract 필요 x
  - 추상 프로퍼티나 메서드가 있으면 추상 클래스가 되어야함
  - 추상 클래스, 메서드, 프로퍼티에서는 상속을 위해 open 키워드를 사용할 필요 없음
    - abstract 자체가 상속과 오버라이딩 허용

인터페이스

- 인터페이스에는 추상 메서드나 일반 메서드가 포함됨, 메서드에 구현 내용이 포함될 수 있음
- 프로퍼티는 선언만 가능 / 객체 생성 불가능

→ java도 8 이후 버전에서는 default 키워드를 통해 interface에 메소드에 구현 내용을 넣을 수 있음

- 인터페이스의 선언과 구현
  - 추상클래스와 다르게 abstract를 붙여주지 않다고 추상 프로퍼티와 추상 메서드가 지정됨
  - 코틀린은 상속(extends), 구현(implements) 둘다 콜론(:)을 통해 구현

- 게터를 구현한 프로퍼티
  - 인터페이스에서는 프로퍼티에 값을 저장할 수 없지만 val로 선언된 프로퍼티는 게터를 통해 필요한 내용을 구현할 수 있음

```
interface Pet{
  val msgTags: String //val 선언시 게터의 구현이 가능
    get() = "I'm your lovely pet!"
}
```

  - 초기화할 수는 없지만 반환값을 지정할 수는 있음

여러 인터페이스의 구현

- 동일한 이름의 메서드 실행할 때
  - super<인터페이스 이름>.메서드 이름()
- 인터페이스의 위임

```
interface A {
    fun functionA( ){}
}

interface B {
    fun functionB( ){}
}

class C(val a: A, val b: B) {
    fun functionC( ){
        a.functionA( )
        b.functionB( )
    }
}
```

```
class DelegatedC(a: A, b: B): A by a, B by b {
    fun functionC( ) {
        functionA( )
        functionB( )
    }
}
```

- 위임을 이용한 멤버 접근

커피 제조기 만들어 보기

▼ 07-2 데이터 클래스와 기타 클래스

- 데이터 클래스 : 자원의 낭비를 막고 데이터 저장에 초점을 맞추기 위해 사용

데이터 전달을 위한 데이터 클래스

- DTO(Data Transfer Object) / java에서는 POJO(Plain Old Java Object)
  - DTO는 구현로직을 가지고 있지 않고 순수한 데이터 객체를 표현
  - 코틀린에서는 DTO를 위해 데이터 클래스를 정의할 때 게터/세터, toString(), equeals() 같은 메서드를 자동으로 생성함
  - 내부적으로 자동으로 생성되는 메서드
    - 프로퍼티를 위한 게터/세터
    - 비교를 위한 equals()와 키 사용을 위한 hashCode()
    - 프로퍼티를 문자열로 변환해 순서대로 보여주는 toString()
    - 객체 복사를 위한 copy()
    - 프로퍼티에 상응하는 component1(), component2() 등
- 데이터 클래스 선언하기

```
data class Customer(var name: String, var email: String)
```

- 데이터 클래스는 다음 조건을 만족해야 함
  - 주 생성자는 최소한 하나의 매개변수르 가져야함
  - 주 생성자의 모든 매개변수는 val, var로 지정된 프로퍼티여야 한다.
  - 데이터 클래스는 abstract, open, sealed, inner 키워드를 사용할 수 없다.
- 필요하다면 부생성자나 init 블록을 넣을 수 있음

```
data class Customer(var name: String, var email: String) {
    var job: String = "Unknown"
    constructor(name: String, email: String, _job: String): this(name, email) {
        job = _job
    }
    init {
        // 간단한 로직은 여기에
    }
}
```

데이터 클래스가 자동 생성하는 메서드

제공된 메서드	기능
equals( )	두 객체의 내용이 같은지 비교하는 연산자(고유 값은 다르지만 의미 값이 같을 때)
hashCode( )	객체를 구별하기 위한 고유한 정숫값 생성, 데이터 세트나 해시 테이블을 사용하기 위한 하나의 생성된 인덱스
copy( )	빌더 없이 특정 프로퍼티만 변경해서 객체 복사하기
toString( )	데이터 객체를 읽기 편한 문자열로 반환하기
componentN( )	객체의 선언부 구조를 분해하기 위해 프로퍼티에 상응하는 메서드

- hashCode()

```
val cus1 = Customer("Sean", "sean@mail.com")
val cus2 = Customer("Sean", "sean@mail.com")
...
println(cus1 == cus2) // 동등성 비교
println(cus1.equals(cus2)) // 위와 동일
println("${cus1.hashCode( )}, ${cus2.hashCode( )}")
```

- copy()

```
val cus3 = cus1.copy(name = "Alice") // name만 변경하고자 할 때
println(cus1.toString( ))
println(cus3.toString( ))
```

```
Customer(name=Sean, email=sean@mail.com)
Customer(name=Alice, email=sean@mail.com)
```

- 객체 디스트럭처링하기(Destructuring)
  - 객체가 가지고 있는 프로퍼티를 개별 변수로 분해하여 할당하는 것

```
val (name, email) = cus1
println("name = $name, email = $email")
```

  - 가져올 필요가 없으면 언더스코어(_)를 사용해 제외할 수 있음
  - 개별적으로 프로퍼티를 가져오기 위해 componentN() 메서드를 사용할 수 있음

```
val name2 = cus1.component1( )
val email2 = cus1.component2( )
println("name = $name2, email = $email2")
```

  - 반복문 사용

```
val cus1 = Customer("Sean", "sean@mail.com")
val cus2 = Customer("Sean", "sean@mail.com")
val bob = Customer("Bob", "bob@mail.com")
val erica = Customer("Erica", "erica@mail.com")

val customers = listOf(cus1, cus2, bob, erica) // 모든 객체를 컬렉션 List 목록으로 구성
...
for((name, email) in customers) { // 반복문을 사용해 모든 객체의 프로퍼티 분해
    println("name = $name, email = $email")
}
```

내부 클래스 기법

- 코틀린은 중첩 클래스, 이너 클래스 2가지의 내부 클래스 기법이 있음
  - 중첩 클래스 : 클래스 안에 또다른 클래스가 정의됨
  - 이너 클래스 : 클래스 내부에 있음 / 사용 방법이 조금 다름
  - 그 밖에도 지역 클래스와 익명 객체 방법으로도 내부 클래스 정의 가능

    → 너무 남용하면 클래스의 의존성이 커지고 코드가 읽기 어려워짐

자바와 코틀린의 내부 클래스 비교

자바	코틀린
정적 클래스(Static Class)	중첩 클래스(Nested Class): 객체 생성 없이 사용 가능
멤버 클래스(Member Class)	이너 클래스(Inner Class): 필드나 메서드와 연동하는 내부 클래스로 inner 키워드가 필요하다.
지역 클래스(Local Class)	지역 클래스(Local Class): 클래스의 선언이 블록 안에 있는 지역 클래스이다.
익명 클래스(Anonymous Class)	익명 객체(Anonymous Object): 이름이 없고 주로 일회용 객체를 사용하기 위해 object 키워드를 통해 선언된다.

- 코틀린 이너 클래스는 inner 키워드가 필요함
- 중첩 클래스 : 기본적으로 정적(static) 클래스처럼 다뤄짐 → 객체 생성 없이 접근 가능

```
class Outer {
    val ov = 5
    class Nested {
        val nv = 10
        fun greeting( ) = "[Nested] Hello ! $nv" // 외부의 ov에는 접근 불가
    }
    fun outside( ) {
        val msg = Nested( ).greeting( ) // 객체 생성 없이 중첩 클래스의 메서드 접근
        println("[Outer]: $msg, ${Nested( ).nv}") // 중첩 클래스의 프로퍼티 접근
    }
}

fun main( ) {
   // static처럼 객체 생성 없이 사용
    val output = Outer.Nested( ).greeting( )
    println(output)

   // Outer.outside( )  // 오류! 외부 클래스의 경우는 객체를 생성해야 함
    val outer = Outer( )
    outer.outside( )
}
```

  - 중첩 클래스는 바로 바깥 클래스의 맴버에 접근할 수 없음

- 바깥 클래스가 컴패니언 객체를 가지면 접근 가능

```
class Outer {
    class Nested {
        ...
        fun accessOuter( ) { // 컴패니언 객체는 접근할 수 있음
            println(country)
            getSomething( )
        }
    }
    companion object { // 컴패니언 객체는 static처럼 접근 가능
        const val country = "Korea"
        fun getSomething( ) = println("Get something...")
    }
}
```

- 이너 클래스
  - 클래스 안에 이너 클래스를 정의 하면 바깥 클래스의 맴버들에 접근할 수 있음
    - private 맴버도 접근 가능

```
class Smartphone(val model: String) {
    private val cpu = "Exynos"

    inner class ExternalStorage(val size: Int) {
       fun getInfo( ) = "${model}: Installed on $cpu with ${size}Gb" // 바깥 클래스의 프로퍼티 접근
    }
}

fun main( ) {
    val mySdcard = Smartphone("S7").ExternalStorage(32)
    println(mySdcard.getInfo( ))
}
```

- 지역 클래스
  - 특정 메서드 블록이나 init 블록 같이 블록 범위에서만 유효한 클래스

```
...
class Smartphone(val model: String) {
    private val cpu = "Exynos"
...
    fun powerOn( ): String {
        class Led(val color: String) { // 지역 클래스 선언
            fun blink( ): String = "Blinking $color on $model" // 외부의 프로퍼티는 접근 가능
```

```
        }
        val powerStatus = Led("Red") // 여기에서 지역 클래스가 사용됨
        return powerStatus.blink( )
    } // powerOn( ) 블록 끝
}

fun main( ) {
...
    val myphone = Smartphone("Note9")
    myphone.ExternalStorage(128)
    println(myphone.powerOn( ))
}
```

- 익명 객체

```
fun powerOn( ): String {
    class Led(val color: String) {
        fun blink( ): String = "Blinking $color on $model"
    }
    val powerStatus = Led("Red")
    val powerSwitch = object : Switcher { // ② 익명 객체를 사용해 Switcher의 on( )을 구현
        override fun on( ): String {
            return powerStatus.blink( )
        }
    } // 익명(object) 객체 블록의 끝
    return powerSwitch.on( ) // 익명 객체의 메서드 사용
}
```

실드 클래스와 열거형 클래스

- 실드 클래스 : 미리 만들어 놓은 자료형들을 묶어서 제공
- 실드 클래스

- sealed 키워드를 class와 함께 사용
- 추상 클래스와 같기 때문에 객체를 만들수는 없음
- 생성자도 기본적으로 private, private 이 아닌 생성자 허용하지 않음

```
// 실드 클래스를 선언하는 첫 번째 방법
sealed class Result {
    open class Success(val message: String): Result( )
    class Error(val code: Int, val message: String): Result( )
}

class Status: Result( ) // 실드 클래스 상속은 같은 파일에서만 가능
class Inside: Result.Success("Status") // 내부 클래스 상속
```

- 실드 클래스 자체를 상속할 때는 같은 파일에서만 가능

```
// 실드 클래스를 선언하는 두 번째 방법
sealed class Result

open class Success(val message: String): Result( )
class Error(val code: Int, val message: String): Result( )

class Status: Result( )
class Inside: Success("Status")
```

- 실드 클래스는 특정 객체 자료형에 따라 when문과 is에 의해 선택적으로 실행할 수 있음
- 필요한 경우의 수를 직접 지정할 수 있음(else 없이도 사용가능)

- 열거형 클래스
  - 여러 개의 상수를 선언하고 열거된 값을 조건에 따라 선택할 수 있는 특수한 클래스
  - 실드 클래스처럼 다양한 자료형을 다루지는 못함

```
enum class 클래스 이름 [(생성자)] {
    상수1[(값)], 상수2[(값)], 상수3[(값)], ...
    [; 프로퍼티 혹은 메서드]
}
```

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
```

- 각 상수는 Direction 클래스의 객체로 취급되고 쉼표(,)로 구분함

```
enum class DayOfWeek(val num: Int) {
  MONDAY(1), TUESDAY(2), WEDNESDAY(3), THURSDAY(4),
  FRIDAY(5), SATURDAY(6), SUNDAY(7)
}
```

```
val day = DayOfWeek.SATURDAY // SATURDAY의 값 읽기
when(day.num) {
    1, 2, 3, 4, 5 -> println("Weekday")
    6, 7 -> println("Weekend!")
}
```

- 필요한 경우 메서드를 포함할 수 있는데 세미콜론(;)을 통해 열거한 상수 객체를 구분함

```
enum class Color(val r: Int, val g: Int, val b: Int) {
    RED(255, 0, 0), ORANGE(255, 165, 0),
    YELLOW(255, 255, 0), GREEN(0, 255, 0), BLUE(0, 0, 255),
    INDIGO(75, 0, 130), VIOLET(238, 130, 238); // 세미콜론으로 끝을 알림

    fun rgb( ) = (r * 256 + g) * 256 + b // 메서드를 포함할 수 있음
}

fun main(args: Array) {
    println(Color.BLUE.rgb( ))
}
```

```
fun getColor(color: Color) = when (color) {
    Color.RED -> color.name // 이름 가져오기
    Color.ORANGE -> color.ordinal // 순서 번호: 1
    Color.YELLOW -> color.toString( ) // 문자열 변환
    Color.GREEN -> color // 기본값(문자열)
    Color.BLUE -> color.r // r값: 0
    Color.INDIGO -> color.g
    Color.VIOLET -> color.rgb( ) // 메서드 연산 결과
}

fun main( ) {
 ...
    println(getColor(Color.BLUE))
}
```

  - 열거형 클래스의 각 상수는 객체로 취급되므로 몇가지 기본적인 맴버를 제공함
  - name : 상수 이름 자체를 반환 / toString() : 이름 반환 / ordinal : 0부터 시작하는 순서

- 인터페이스를 통한 열거형 클래스 구현하기

```
interface Score {
    fun getScore( ): Int
}

enum class MemberType(var prio: String) : Score { // Score를 구현할 열거형 클래스
    NORMAL("Thrid") {
        override fun getScore( ): Int  = 100 // 구현된 메서드
    },
    SILVER("Second") {
        override fun getScore( ): Int  = 500
    },
    GOLD("First") {
        override fun getScore( ): Int  = 1500
    }
```

- values() 맴버 메서드를 사용해 내용을 출력할 수 있음

```
fun main( ) {
    println(MemberType.NORMAL.getScore( ))
    println(MemberType.GOLD)
    println(MemberType.valueOf("SILVER"))
    println(MemberType.SILVER.prio)

    for (grade in MemberType.values( )) { // 모든 값을 가져오는 반복문
        println("grade.name = ${grade.name}, prio = ${grade.prio}")
    }
}
```

```
grade.name = NORMAL, prio = Thrid
grade.name = SILVER, prio = Second
grade.name = GOLD, prio = First
```

애노테이션 클래스

- 애노테이션 클래스
  - 코드에 부가 정보를 추가하는 역할 / @ 기호와 함께 나타내는 표기법
  - 주로 컴파일러나 프로그램 실행 시간에서 사전 처리를 위해 사용
  - ex) @Test, @JvmStatic
- 애노테이션 선언하기
  - annotation class 애노테이션 이름
  - 어노테이션 정의 속성

- **@Target:** 애노테이션이 지정되어 사용할 종류(클래스, 함수, 프로퍼티 등)를 정의
- **@Retention:** 애노테이션을 컴파일된 클래스 파일에 저장할 것인지 실행 시간에 반영할 것인지 결정
- **@Repeatable:** 애노테이션을 같은 요소에 여러 번 사용 가능하게 할지를 결정
- **@MustBeDocumented:** 애노테이션이 API의 일부분으로 문서화하기 위해 사용

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE) // 애노테이션의 처리 방법 - SOURCE: 컴파일 시간에 제거됨
@MustBeDocumented
annotation class Fancy
```

  - @Rentention : 애노테이션의 처리방법 기술

- SOURCE : 컴파일 시간에 애노테이션이 제거 / BINARY : 클래스 파일에 포함되지만 리플렉션에 의해 나타나지 않음 / RUNTIME : 클래스 파일에 저장되고 리플렉션에 의해 나타남
- 리플렉션 : 프로그램을 실행할 때 프로그램의 특정 구조를 분석해 나타내는 기법

- 애노테이션 위치

```
@Fancy class MyClass {
    @Fancy fun myMethod(@Fancy myProperty: Int): Int {
        return (@Fancy 1)
    }
}
```

- 생성자에 어노테이션을 사용하면 constructor 생략 x
- 프로퍼티 게터/ 세터에도 사용가능

```
class Foo {
    var x: MyDependency? = null
        @Fancy set
}
```

- 애노테이션의 매개변수와 생성자
  - 애노테이션에 매개변수 지정하기

```
annotation class Special(val why: String) // 애노테이션 클래스의 정의
@Special("example") class Foo {} // 애노테이션에 매개변수를 지정
```

- 매개변수로 사용될 수 있는 자료형

- 자바의 기본형과 연동하는 자료형(Int형, Long형 등)
- 문자열
- 클래스(클래스 이름::class)
- 열거형
- 기타 애노테이션
- 위의 목록을 가지는 배열

- 애노테이션이 또 다른 애노테이션을 가지고 사용할 때는 @ 기호를 사용하지 않아도 됨

```
annotation class ReplaceWith(val expression: String) // 첫 번째 애노테이션 클래스 정의

annotation class Deprecated( // 두 번째 애노테이션 클래스 정의
    val message: String,
    val replaceWith: ReplaceWith = ReplaceWith(""))
...
// ReplaceWith는 @ 기호가 생략됨
@Deprecated("This function is deprecated, use === instead", ReplaceWith("this === other"))
```

- 애노테이션 인자로 특정 클래스가 필요하면 코틀린의 KClass를 사용해야함

  → 코틀린 컴파일러가 자동으로 자바 클래스로 변환함 → 이후에 자바 코드에서도 애노테이션 인자를 사용할 수 있음

```
import kotlin.reflect.KClass
annotation class Ann(val arg1: KClass<*>, val arg2: KClass<out Any>)
...
@Ann(String::class, Int::class) class MyClass
```

- 표준 애노테이션
  - @JvmName

```
@JvmName("filterStrings")
fun filter(list: List<String>): Unit

@JvmName("filterInts")
fun filter(list: List<Int>): Unit
```

  - filter()라는 이름을 자바에서 각각 filterStrings()와 filterInts()로 바꿔주는 것
  - 자바로 바뀐 것

```
public static final void filterStrings(java.util.List<java.lang.String>);
...
public static final void filterInts(java.util.List<java.lang.Integer>);
...
```

  - @JvmStatic : 자바의 정적 메서드를 생성할 수 있게 해줌
  - @Throw : 코틀린의 throw 구문이 자바에서도 포함되도록 함

```
class File(val path: String) {
    @Throws(FileNotFoundException::class)
    fun exists( ): Boolean {
        if (!Paths.get(path).toFile( ).exists( ))
        throw FileNotFoundException("$path does not exist")
        return true
    }
}
```

```
public void checkFile( ) throws FileNotFoundException {
    boolean exists = new File("somefile.txt").exists( );
    System.out.println("File exists");
}
```

- @JvmOverloads : 크톨린에서 기본값을 적용한 인자에 함수르 모두 오버로딩 해줌

→ 표준 애노테이션 : 자바와 원활하게 연동하는 목적이있음

▼ 07-3 연산자 오버로딩

연산자의 우선순위

우선순위	분류	심볼
높음	접미사(Postfix)	++, --, ., ?., ?
	접두사(Prefix)	-, +, ++, --, !, 라벨 선언(이름@)
	오른쪽 형식(Type RHS)	:, as, as?
	배수(Multiplicative)	*, /, %
	첨가(Additive)	+, -
	범위(Range)	..
	중위 함수(Infix Function)	SimpleName
	엘비스(Elvis)	?:
	이름 검사(Name Checks)	in, !in, is, !is
	비교(Comparison)	<, >, <=, >=
	동등성(Equality)	==, !=
	결합(Conjunction)	&&
	분리(Disjunction)	‖
낮음	할당(Assignment)	=, +=, -=, *=, /=, %=

연산자의 작동 방식

- a.plus(b) : a+b를 내부적으로 호출
- 연산자 오버로딩
  - plus 연산자 오버로딩

```
class Point(var x: Int = 0, var y: Int = 10) {
    // plus( ) 함수의 연산자 오버로딩
    operator fun plus(p: Point) : Point {
        return Point(x + p.x, y + p.y)
    }
}

fun main( ) {
    val p1 = Point(3, -8)
    val p2 = Point(2, 9)

    var point = Point( )
    point = p1 + p2 // Point 객체의 + 연산이 가능하게 됨
    println("point = (${point.x}, ${point.y})")
}
```

- — 연산자 오버로딩

```
class Point(var x: Int = 0, var y: Int = 10) {
...
    operator fun dec( ) = Point(--x, --y)
}

fun main( ) {
...
    --point // -- 연산자
    println("point = (${point.x}, ${point.y})")
}
```

연산자의 종류

- 산술 연산자

표현식	의미
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b) (Kotlin 1.1부터) , a.mod(b) (지원 중단)
a..b	a.rangeTo(b)

- 호출 연산자(Invoke Operator) : 함수 호출을 돕는데 사용

```
class Manager {
    operator fun invoke(value: String) {
        println(value)
```

```
    }
}

fun main( ) {
    val manager = Manager( )
    manager("Do something for me!")  // manager.invoke("...") 형태로 호출되며 invoke가 생략됨
}
```

- 원래는 manager.invoke("…") 형태로 호출되어야 하지만 invoke를 생략하고 객체 이름만 작성해서 코드를 읽기가 수월해짐

```
val sum = { x: Int, y: Int -> x + y }
sum.invoke(3, 10)
sum(3, 10)
```

- 람다식에서는 기본적으로 invoke가 정의됨

- 인덱스 접근 연산자(Indexed Access Operator)
  - 게터/ 세터를 다루기 위한 대괄호([]) 연산자를 제공함

인덱스 접근 연산자의 의미

표현식	의미
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, ..., i_n] = b	a.set(i_1, ..., i_n, b)

- 단일 연산자

단일 연산자의 의미

표현식	의미
+a	a.unaryPlus( )
-a	a.unaryMinus( )
!a	a.not( )

```
data class Point(val x: Int, val y: Int)

operator fun Point.unaryMinus( ) = Point(-x, -y)

val point = Point(10, 20)
println(-point) // 단일 연산자에 의해 (-10, -20) 값을 바꿈
```

- 범위 연산자

```
if (i in 1..10) { // 1 <= i && i <= 10 와 동일
    println(i)
}
for (i in 1..4) print(i) // "1234" 출력
```

범위 연산자의 의미

표현식	의미
a in b	b.contains(a)
a !in b	!b.contains(a)

- 대입 연산자

대입 연산자의 의미

표현식	의미
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a %= b	a.remAssign(b), a.modAssign(b) (지원 중단)

  - +에 대응하는 plus()를 오버로딩하면 +=는 자동으로 구현됨 → plusAssign()을 따로 오버로딩 x도됨
  - 2개를 동시에 오버로딩하면 무엇으로할지 모호해지므로 오류 발생

- 동등성 연산자

동등성 연산자의 의미

표현식	의미
a == b	a?.equals(b) ?: (b === null)
a != b	!(a?.equals(b) ?: (b === null))

    - equals는 Any안에 operator 키워드가 붙어서 구현 → 하위 클래스에서는 override 함수를 사용해서 ==와 치완 가능
        - → equals는 확장 함수로 구현할 수 없음
        - ===과 ≠==는 오버로딩 x
- 비교 연산자

비교 연산자의 의미

표현식	의미
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0