# Chapter09. 연산자 오버로딩과 다른 관례

9장에서 다루는 내용
    관례
산술 연산자를 오버로드해서 임의의 클래스에 대한 연산을 더 편리하게 만들기
    plus, times, divide 등: 이항 산술 연산 오버로딩
    복합 대입 연산자 오버로딩: 연산을 적용한 다음에 그 결과를 바로 대입
    단항 연산자 오버로딩: 피연산자가 1개뿐인 연산자
비교 연산자를 오버로딩해서 객체들 사이의 관계를 쉽게 검사
    동등성 연산자: equals
    순서 연산자 : compareTo
**컬렉션과 범위에 대해 쓸 수 있는 관례**
    **인덱스로 원소에 접근: get과 set**
    in 관례 : 어떤 객체가 컬렉션에 들어 있는지 검사
    rangeTo와 rangeUntil 관례: 객체로부터 범위 만들기
    **for 루프를 위한 iterator 관례**
component 함수를 사용해 구조 분해 선언 제공
    구조 분해 선언과 루프
    _ 문자를 사용해 구조 분해 값 무시
위임 프로퍼티: 프로퍼티 접근자 로직 재활용
    위임 프로퍼티의 기본 문법과 내부 동작
    위임 프로퍼티 사용: by lazy()를 사용한 지연 초기화
    위임 프로퍼티 구현
    위임 프로퍼티는 커스텀 접근자가 있는 감춰진 프로퍼티로 변환된다.
    맵에 위임해서 동적으로 애트리뷰트 접근
    실전 프레임워크가 위임 프로퍼티를 활용하는 방법
요약

## 9장에서 다루는 내용

- 연산 오버로딩
- 관례: 여러 연산을 지원하기 위해 특별한 이름이 붙은 메서드
- 위임 프로퍼티

### 관례

- 코틀린에서는 직접 작성한 함수를 어떤 언어 기능이 호출함으로써 구현되는 경우가 있다.
  - for in 루프에 java.lang.Iterable 구현한 객체 사용
  - 자원을 사용하는 try 문에 java.lang.AutoCloseable 구현 객체 사용
- 특정 언어 기능과 미리 정해진 이름의 함수를 연결해주는 기법
  - plus 메서드
- 코틀린에서 자바와 달리 관례를 채택한 이유는 기존 자바 클래스를 코틀린 언어에 적용하기 위해
  - 코틀린쪽에서 자바 클래스가 새로운 인터페이스를 구현하게 만들 수 없다.
  - 하지만 확장 함수 메커니즘을 사용하면 기존 클래스에 새로운 메서드를 추가할 수 있다.

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

## 산술 연산자를 오버로드해서 임의의 클래스에 대한 연산을 더 편리하게 만들기

- 코틀린에서 관례를 사용하는 가장 단순한 예는 산술 연산자
- 자바에서는 오직 기본 타입에 대해서만 산술 연산자를 사용할 수 있다
  - 추가 ) String
- BigInteger 클래스를 사용하면  add 보단 산술연사자를 사용하는게 더 낫다.
  - 코틀린에서 가능하다.

### plus, times, divide 등: 이항 산술 연산 오버로딩

```
data class Point(val x: Int, val y: Int) {
    operator fun plus(other: Point): Point {
        return Point(x + other.x, y + other.y)
    }
}

operator fun Point.plus(other: Point): Point {
    return Point(x + other.x, y + other.y)
}


fun main() {
    val p1 = Point(10, 20)
    val p2 = Point(30, 40)
    println(p1 + p2)
    // Point(x=40, y=60)
}
```

- plus라는 이름의 연산자 함수를 정의한다.
    - 반드시 operator 키워드를 붙여야 한다.
- + 연산자는 내부적으로 plus 함수 호출로 변환된다.
- 멤버 함수 대신 확장 함수로 정의할 수도 있다.
- 오버로딩 가능한 이항 산술 연산자
    - times, *
    - div, /
    - mod, %
    - plus, +
    - minus, -
- 연산자 우선순위는 표준 숫자 타입에 대한 연산자 우선순위와 같다.

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

```
operator fun Point.times(scale: Double): Point {
    return Point((x * scale).toInt(), (y * scale).toInt())
}

fun main() {
    val p = Point(10, 20)
    println(p * 1.5)
    // Point(x=15, y=30)
}
```

- 연산자를 정의할 때 두 피연산자가 같은 타입일 필요는 없다.
    - 교환법칙 x

```
operator fun Char.times(count: Int): String {
    return toString().repeat(count)
}

fun main() {
    println('a' * 3)
    // aaa
}
```

- 연산자 함수의 반환 타입이 피연산자 중 하나와 일치해야만 하는 것도 아니다.
- operator 함수도 오버로딩할 수 있다.

```
fun main() {
    println(0x0F and 0xF0)
    // 0
    println(0x0F or 0xF0)
    // 255
    println(0x1 shl 4)
    // 16
}
```

- 비트 연산자에 대해 특별한 연산자 함수를 사용하지 않는다.
    - 대신에 중위 연산자 표기법을 지원하는 일반 함수를 사용한다.

## 복합 대입 연산자 오버로딩: 연산을 적용한 다음에 그 결과를 바로 대입

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

operator fun Point.plus(other: Point): Point {
    return Point(x + other.x, y + other.y)
}

fun main() {
    var point = Point(1, 2)
    point += Point(3, 4)
    println(point)
    // Point(x=4, y=6)
}
```

- plus와 같은 연산자를 오버로딩하면 그와 관련 있는 복합 대입 연산자도 자동으로 함께 지원한다.
    - eg) +=, -=

```
fun main() {
    val numbers = mutableListOf<Int>()
    numbers += 42
    println(numbers[0])
    // 42
}



operator fun <T> MutableCollection<T>.plusAssign(element: T) {
  this.add(element)
}

val numbers = mutableList<Int>()
```

```
numbers += 42
numbers[0] // 42
```

- 경우에 따라 `+=` 연산이 객체에 대한 참조를 다른 참조로 바꾸기 보다 원래 객체의 내부 상태를 변경해야 하는 경우도 있다.
  - eg) 변경 가능한 컬렉션에 원소를 추가하는 경우
- 이런 경우 반환 타입이 `Unit`인 `plusAssign` 함수를 정의하면 코틀린은 `+=` 연산자에 그 함수를 사용한다.
- 하지만 어떤 클래스가 `plus`와 `plusAssign` 함수 모드를 정의하고 `+=`를 사용하는 경우 컴파일러는 오류를 발생시킨다.
  - 양쪽으로 컴파일 가능하기 때문
  - 하지만 동시에 정의하지 말자

```
fun main() {
    val list = mutableListOf(1, 2)
    list += 3
    val newList = list + listOf(4, 5)
    println(list)
    // [1, 2, 3]
    println(newList)
    // [1, 2, 3, 4, 5]
}
```

- +, - 는 항상 새로운 컬렉션 반환
- +=, = 항상 변경 가능한 컬렉션에 작용해 메모리에 있는 객체 상태 변화

## 단항 연산자 오버로딩: 피연산자가 1개뿐인 연산자

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

operator fun Point.unaryMinus(): Point {
    return Point(-x, -y)
}
```

```
fun main() {
    val p = Point(10, 20)
    println(-p)
    // Point(x=-10, y=-20)
}
```

- 단항 연산자를 오버로딩하는 절차도 이항 연산자와 같다.
- 단항 연산자 오버로딩 함수는 인자를 취하지 않는다.
- 오버로딩 가능한 단항 산술 연산자
- unaryPlus, +a
- unaryMinus, -a
- !a, not
- ++a, a++, inc
- —a, a—, dec

```
operator fun BigDecimal.inc() = this + BigDecimal.ONE

fun main() {
    var bd = BigDecimal.ZERO
    println(bd++)
    // 0
    println(bd)
    // 1
    println(++bd)
    // 2
}
```

- inc나 dec 함수를 정의해 증감 연산자를 오버로딩하는 경우 컴파일러는 일반적인 전위와 후위 증감 연산자와 같은 의미를 제공한다.

## 비교 연산자를 오버로딩해서 객체들 사이의 관계를 쉽게 검사

- `equals`나 `compareTo`를 호출해야 하는 자바와 달리 코틀린에서는 `==` 비교 연산자를 직접 사용할 수 있어서 코드가 더 간결하며 이해하기 쉽다.

## 동등성 연산자: equals

```
class Point(val x: Int, val y: Int) {
    override fun equals(other: Any?): Boolean {
        if (other === this) return true
        if (other !is Point) return false
        return other.x == x && other.y == y
    }
}

fun main() {
    println(Point(10, 20) == Point(10, 20))
    // true
    println(Point(10, 20) != Point(5, 5))
    // true
    println(null == Point(1, 2))
    // false
}
```

- 코틀린은 `==` 연산자 호출을 `equals` 메서드 호출로 컴파일한다는 것을 이미 배웠다.
  - 이것도 사실은 이전에 살펴봤던 경우와 동일하다.
  - `≠` 도 eqauls 호출로 컴파일된다.
- `==`와 `≠`는 내부에서 인자가 null인지 검사하므로 다른 연산과 달리 null이 될 수 있는 값에도 사용할 수 있다.
- `a == b`라는 비교를 처리할 때 `a`가 null인지 판단해서 null이 아닌 경우에만 `equals`를 호출한다.
  - `a`가 null이라면 `b`도 null인 경우에만 결과가 `true`이다.
- `equals`는 다른 연산자 오버로딩 관례와 달리 `Any`에 정의된 메서드이므로 `operator` 대신 `override`가 필요하다.
  - `Any`의 `equals`에 `operator`가 붙어있다.

- 이처럼 상위 클래스의 메서드에 `operator`가 붙어있다면 하위 클래스의 메서드에서 붙이지 않아도 적용된다.

## 순서 연산자 : compareTo

```
class Person(
    val firstName: String, val lastName: String,
) : Comparable<Person> {

    override fun compareTo(other: Person): Int {
        return compareValuesBy(
            this, other,
            Person::lastName, Person::firstName
        )
    }
}

fun main() {
    val p1 = Person("Alice", "Smith")
    val p2 = Person("Bob", "Johnson")
    println(p1 < p2)
    // false
}
```

- <, >, ≤, ≥
- 자바에서 정렬이나 최댓값, 최솟값 등 값을 비교해야 하는 알고리즘에 사용할 클래스는 Comparable 인터페이스를 구현해야 한다.
- Comparable 인터페이스에 들어있는 `compareTo` 메서드는 한 객체와 다른 객체의 크기를 비교해 정수로 나타내준다.
- 자바에서는 이 메서드를 짧게 호출할 수 있는 방법이 없지만 코틀린은 Comparable 인터페이스 안에 있는 `compareTo` 메서드를 호출하는 관례를 제공한다.
    - 마찬가지로 기본 메서드에 operator 붙어있다.
- 두 객체를 비교하는 식은 compareTo의 결과를 0과 비교하는 코드로 컴파일된다.
    - a ≥ b

    - a.compareTo(b) ≥ 0
- 이렇게 하면 코드는 간결해지지만 필드를 직접 비교하는 것이 훨씬 더 빠르다는 것을 명심하자.
    - 언제나 처음에는 이해하기 쉽고 간결한 코드를 작성하고, 나중에 그 코드가 자주 호출됨에 따라 성능이 문제가 된다면 개선하자.

```
fun main() {
    println("abc" < "bac")
    // true
}
```

- Comparable 인터페이스를 구현하는 모든 자바 클래스를 코틀린에서는 간결한 연산자 구문으로 비교할 수 있다.
- eg) String

## 컬렉션과 범위에 대해 쓸 수 있는 관례

- 컬렉셔 사용시에 자주 사용하는 연산을 연산자 구문으로 사용할 수 있다.
    - 인덱스 접근 연산자: 컬렉션에서 인덱스를 사용해 원소를 읽거나 쓰기 연산
    - in 범위 연산자: 값이 컬렉션에 속해있는지 검사 연산,

### 인덱스로 원소에 접근: get과 set

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

operator fun Point.get(index: Int): Int {
    return when (index) {
        0 -> x
        1 -> y
        else ->
            throw IndexOutOfBoundsException("Invalid coordinate $index")
```

```
    }
}

fun main() {
    val p = Point(10, 20)
    println(p[1])
    // 20
}
```

```
data class MutablePoint(var x: Int, var y: Int)

operator fun MutablePoint.set(index: Int, value: Int) {
    when (index) {
        0 -> x = value
        1 -> y = value
        else ->
            throw IndexOutOfBoundsException("Invalid coordinate $index")
    }
}

fun main() {
    val p = MutablePoint(10, 20)
    p[1] = 42
    println(p)
    // MutablePoint(x=10, y=42)
}
```

- 코틀린에서는 인덱스 접근 연산자도 관례를 따른다.
- 인덱스 접근 연산자를 이용한 읽기 연산
    - get 연산자 메서드
- 인덱스 접근 연산자를 이용한 쓰기 연산
    - set 연산자 메서드
- operator + get / set으로 구현하면 된다.

- 인덱스는 맵이 String을 쓰는 것처럼 다양한 타입으로 구현 가능

- **[a,b]** 처럼 여러 인덱스 사용도 가능하다

## in 관례 : 어떤 객체가 컬렉션에 들어 있는지 검사

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

data class Rectangle(val upperLeft: Point, val lowerRight: Point)

operator fun Rectangle.contains(p: Point): Boolean {
    return p.x in upperLeft.x..<lowerRight.x &&
            p.y in upperLeft.y..<lowerRight.y
}

fun main() {
    val rect = Rectangle(Point(10, 20), Point(50, 50))
    println(Point(20, 30) in rect)
    // true
    println(Point(5, 5) in rect)
    // false
}
```

- in은 객체가 컬렉션에 들어있는지 검사한다.
  - 멤버쉽 검사
  - 대응하는 함수는 contains 이다.
- a in c
  - a가 파라미터 , c가 수신객체
  - c.contains(a) 로 변환된다.
- 10 until 20 : 10~19
- 10 .. 20: 10~20

## rangeTo와 rangeUntil 관례: 객체로부터 범위 만들기

```
operator fun <T: Comparable<T>> T.rangeTo(that: T): ClosedRange<T>

val now = LocalDate.now()
val vacation = now..now.plusDays(10)
println(now.plusWeeks(1) in vacation) // true
```

- .. 연산자는 rangeTo를 간략하게 표기한 방법이다.
  - start..end: sstart.rangeTo(end)
- Comparable 인터페이스를 구현하면 rangeTo가 이미 정의되어 있다.
- ..을 쓸땐 연산자 우선순위 때문에 괄호로 감싸주면 좋다
- rangeUntil
  - ..<
  - 열린 범위

## for 루프를 위한 iterator 관례

```
operator fun ClosedRange<LocalDate>.iterator(): Iterator<LocalDate> =
    object : Iterator<LocalDate> {
        var current = start

        override fun hasNext() =
            current <= endInclusive

        override fun next(): LocalDate {
            val thisDate = current
            current = current.plusDays(1)
            return thisDate
        }
    }

fun main() {
    val newYear = LocalDate.ofYearDay(2042, 1)
    val daysOff = newYear.minusDays(1)..newYear
    for (dayOff in daysOff) {
```

```
        println(dayOff)
    }
    // 2041-12-31
    // 2042-01-01
}
```

- for 루프 내에서 사용되는 in 은 iterator() 이름을 가진 함수를 호출하여 이터레이터를 얻는다.
- 그 이터레이터에 대해 hasNext와 next로 순회한다.

## component 함수를 사용해 구조 분해 선언 제공

```
val (a,b) = p
// val a = p.component1()
// val b = p.component2()
```

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

fun main() {
    val p = Point(10, 20)
    val (x, y) = p
    println(x)
    // 10
    println(y)
    // 20
}
```

- 내부에서 구조 분해 선언은 관례를 사용한다.
    - componentN 함수
    - N은 변수 위치에 따라 붙는 번호
- val (a, b) = p
    - val a = p.component1()
    - val b = p.component2()

- data class의 주 생성자에 들어있는 프로퍼티에 대해서는 컴파일러가 자동으로 componentN 함수를 만들어준다.
    - 일반 클래스인경우 operator 붙여서 구현하면 된다.

```
data class NameComponents(
    val name: String,
    val extension: String,
)

fun splitFilename(fullName: String): NameComponents {
    val (name, extension) = fullName.split('.', limit = 2)
    return NameComponents(name, extension)
}
```

- 코틀린 표준 라이브러리에서는 맨 앞의 다섯 원소에 대한 componentN 제공한다.
- Pair, Triple 클래스를 사용해도 좋지만 가독성과 코드의 표현력을 잃게 된다.

## 구조 분해 선언과 루프

```
fun printEntries(map: Map<String, String>) {
    for ((key, value) in map) {
        println("$key -> $value")
    }
}

fun main() {
    val map = mapOf("Oracle" to "Java", "JetBrains" to "Kotlin")
    printEntries(map)
    // Oracle -> Java
    // JetBrains -> Kotlin
}
```

- 함수 본문 내의 선언문 뿐만 아니라 변수 선언이 들어갈 수 있는 장소라면 어디든 구조 분해 선언을 사용할 수 있다.

- 람다가 data class나 map 같은 복합적인 값을 파라미털로 받을 때도 구조 분해 선언 사용 가능

### _ 문자를 사용해 구조 분해 값 무시

```
data class Person(
    val first: String,
    val last: String,
    val age: Int,
    val city: String
)

val (first, last, age) = p
val (first, _, age) = p
```

- 처음부터 3가지요소만 분해하여 뒤쪽의 구조 분해 선언을 제거할 수 있다.
- last 선언을 없애려면 _ 문자 사용
- 구조분해는 데이터 클래스의 프로퍼티 순서에 의존하므로 예상치 못한 이슈가 발생할 수 있다.
    - 결국 작은 컨테이너 클래스나 변경될 가능성이 아주 적은 클래스에 대해서만 유용하다는 것

## 위임 프로퍼티: 프로퍼티 접근자 로직 재활용

- 위임 프로퍼티를 사용하면 값을 뒷받침하는 필드에 단순히 저장하는 것보다 더 복잡한 방식으로 작동하는 프로퍼티를 접근자 로직을 매번 재구현할 필요 없이 쉽게 구현할 수 있다.
    - eg) 프로퍼티는 위임을 사용해 자신의 값을 필드가 아니라 데이터베이스 테이블이나 브라우저 세션, 맵 등에 저장할 수 있다.
- 위임은 객체가 직접 작업을 수행하지 않고 다른 도우미 객체가 그 작업을 처리하도록 맡기는 디자인 패턴을 말한다.
    - 이때 작업을 처리하는 도우미 객체를 위임 객체라고 한다.

- 이러한 위임 패턴을 프로퍼티에 적용해서 접근자 기능을 도우미 객체가 수행하도록 위임한다.
    - 코틀린 언어가 제공하는 기능을 활용할 수 있다.

## 위임 프로퍼티의 기본 문법과 내부 동작

```
var p: Type by Delegate()
```

- p 프로퍼티는 접근자 로직을 다른 객체에 위임
- 여기에서는 Delegate 클래스의 인스턴스를 위임 객체로 사용
- by 뒤에 있는 식을 계산해 위임에 쓰일 객체를 얻는다.
- 프로퍼티 위임 객체가 따라야 하는 관례를 따르는 모든 객체를 위임에 사용할 수 있다.

```
class Foo {
    var p: Type by Deletegate()
}
```

```
class Foo {

        private val delegate = Deletate() // 위임 객체

        var p: Type
            set(value: Type) = delegate.setValue(/* ...*/, value)
            get() = delegate.getValue(/*...*/)

}


val foo = Foo()
val oldValue = foo.p // delegate.getValue()
foo.p = newValue = // delegate.setValue(newValue)
```

- 컴파일러는 숨겨진 도우미 프로퍼티를 만들고 그 프로퍼티를 위임 객체의 인스턴스로 초기화한다.
- p 프로퍼티는 바로 그 위임 객체에게 자신의 작업을 위임한다.
    - p 프로퍼티를 위해 컴파일러가 생성한 접근자는 delegate의 setValue와 getValue 메서드를 호출한다.
- 프로퍼티 위임 관례에 따라 Delegate 클래스는 getValue와 setValue 메서드를 제공해야 하며, 변경 가능한 프로퍼티만 setValue를 사용한다.
    - 추가로 위임 객체는 선택적으로 provideDelegate 함수 구현을 제공할 수도 있다.
    - 이 함수는 최초 생성 시 검증 로직을 수행하거나 위임이 인스턴스화되는 방식을 변경할 수 있다.
    - 이런 함수들은 멤버로 구현할 수도, 확장함수로 구현할 수도 있다.

## 위임 프로퍼티 사용: by lazy()를 사용한 지연 초기화

- 지연 초기화는 객체의 일부분을 초기화하지 않고 남겨뒀다가 실제로 그 부분의 값이 필요할 경우 초기화할 때 흔히 쓰이는 패턴
- 초기화 과정에 자원을 많이 사용하거나 객체를 사용할 때마다 꼭 초기화하지 않아도 되는 프로퍼티에 대해 지연 초기화 패턴을 사용할 수 있다.

```
class Email { /*...*/ }

fun loadEmails(person: Person): List<Email> {
    println("Load emails for ${person.name}")
    return listOf(/*...*/)
}
```

- 이메일은 데이터베이스에 들어있고 불러오려면 아주 오랜 시간이 걸린다고 가정
- 따라서 이메일 프로퍼티의 값을 최초로 사용할 때 단 한 번만 이메일을 데이터베이스에서 가져오고 싶은 요구사항
    - `loadEmails()` : db에서 이메일을 가져오는 함수

```
class Person(val name: String) {
    private var _emails: List<Email>? = null

    val emails: List<Email>
        get() {
            if (_emails == null) { // 최초 접근 시 이메일 가져온다.
                _emails = loadEmails(this)
            }
            return _emails!!
        }
}

fun main() {
    val p = Person("Alice")
    p.emails
    // Load emails for Alice
    p.emails
}
```

- 이메일을 불러오기 전에는 null을 저장하고 불러온 다음에는 이메일 리스트를 저장하는 `_emails` 프로퍼티 추가해서 지연 초기화를 구현한 클래스
- emails 프로퍼티 자체는 커스텀 접근자 사용
- 여기서는 `backing property` 라는 기법을 사용
  - `_emails` : 값을 저장하는 비공개 프로퍼티
  - `emails` : _emails 프로퍼티의 읽기 연산 제공하는 공개 프로퍼티
  - 이들의 이름은 간단한 관례를 사용한다.
- 하지만 이런 코드를 만드는 일은 성가시고 많아지면 유지보수 측면에서 좋지 않다.
  - 구현이 스레드 세이프가 아니기 때문
  - 두 스레드가 모두 emails 프로퍼티에 접근할 때 비용이 많이 드는 loadEmails 함수가 2번 호출되는 것을 막는 장치가 없다.
  - 자원 낭비뿐만 아니라 애플리케이션 상태의 일관성이 사라질 수 있다.
- 코틀린은 더 나은 해법인 위임 프로퍼티를 제공한다.

```
fun loadEmails(person: Person): List<Email> {
    println("Load emails for ${person.name}")
    return listOf(/*...*/)
}

class Person(val name: String) {
    val emails by lazy { loadEmails(this) }
}
```

- 위임 프로퍼티는 데이터를 저장할 때 쓰이는 뒷받침하는 프로퍼티와 값이 오직 한 번만 초기화됨을 보장하는 게터 로직을 함께 캡슐화해준다.
- 예제와 같은 경우를 위한 위임 객체를 반환하는 표준 라이브러리 함수가 `lazy`
    - lazy 함수는 코틀린 관례에 맞는 시그니처의 getValue 메서드가 들어있는 객체를 반환한다.
    - 따라서 lazy를 by 키워드와 함께 사용해 위임 프로퍼티를 만들 수 있다.
    - lazy 함수의 인자는 값을 초기화할 때 호출할 람다다.
    - lazy 함수는 기본적으로 쓰레드 세이프하다.
        - 필요하면 사용할 락을 함수에 전달 가능
    - 다중 스레드 환경에서 사용하지 않을 프로퍼티를 위해 lazy 함수가 동기화를 생략하게 할 수도 있다.

## 위임 프로퍼티 구현

```
fun interface Observer {
    fun onChange(name: String, oldValue: Any?, newValue: Any?)
}

open class Observable {
    val observers = mutableListOf<Observer>()
    fun notifyObservers(propName: String, oldValue: Any?, newValue: Any?) {
        for (obs in observers) {
```

```
            obs.onChange(propName, oldValue, newValue)
        }
    }
}
```

- 어떤 객체의 프로퍼티가 바뀔 때마다 리스너에게 변경 통지를 보내고싶은 요구사항
    - observable
- 위임 프로퍼티 없이 구현해보자

```
class Person(val name: String, age: Int, salary: Int) : Observable() {
    var age: Int = age
        set(newValue) {
            val oldValue = field
            field = newValue
            notifyObservers(
                "age", oldValue, newValue
            )
        }

    var salary: Int = salary
        set(newValue) {
            val oldValue = field
            field = newValue
            notifyObservers(
                "salary", oldValue, newValue
            )
        }
}

fun main() {
    val p = Person("Seb", 28, 1000)
    p.observers += Observer { propName, oldValue, newValue ->
        println(
            """
            Property $propName changed from $oldValue to $newValue!
            """.trimIndent()
```

```
        )
    }
    p.age = 29
    // Property age changed from 28 to 29!
    p.salary = 1500
    // Property salary changed from 1000 to 1500!
}
```

- field 키워드를 통해 뒷받침하는 필드에 접근
- 세터 코드를 보면 중복이 많이 보인다.
    - 클래스 추출 해보자

```
class ObservableProperty(
    val propName: String,
    var propValue: Int,
    val observable: Observable,
) {
    fun getValue(): Int = propValue
    fun setValue(newValue: Int) {
        val oldValue = propValue
        propValue = newValue
        observable.notifyObservers(propName, oldValue, newValue)
    }
}

class Person(val name: String, age: Int, salary: Int) : Observable() {
    val _age = ObservableProperty("age", age, this)
    var age: Int
        get() = _age.getValue()
        set(newValue) {
            _age.setValue(newValue)
        }

    val _salary = ObservableProperty("salary", salary, this)
    var salary: Int
        get() = _salary.getValue()
```

```
        set(newValue) {
            _salary.setValue(newValue)
        }
}
```

- 코드가 코틀린의 위임의 실제 작동 방식과 비슷해졌다.
- 프로퍼티 값을 저장하고 그 값이 바뀌면 자동으로 변경 통지를 전달해주는 도우미 클래스를 만들었다.
- 하지만 아직도 각각의 프로퍼티마다 ObservableProperty를 만들고 게터와 세터에서 ObservableProperty에 작업을 위임하는 준비 코드가 상당 부분 필요
    - 코틀린의 위임 프로퍼티를 사용하자
    - 그러려면 관례에 맞게 수정해야한다.

```
class ObservableProperty(var propValue: Int, val observable: Observable) {
    operator fun getValue(thisRef: Any?, prop: KProperty<*>): Int = propValue

    operator fun setValue(thisRef: Any?, prop: KProperty<*>, newValue: Int) {
        val oldValue = propValue
        propValue = newValue
        observable.notifyObservers(prop.name, oldValue, newValue)
    }
}
```

- 코틀린 관례에 사용하는 다른 함수와 마찬가지로 getValue, setValue 함수에도 oeprator 변경자
- 두 함수는 파라미터를 2개 받는다.
    - `thisRef` : 설정하거나 읽을 프로퍼티가 들어있는 인스턴스
    - `prop` :프로퍼티를 표현하는 객체
- KProperty 인자를 통해 프로퍼티 이름을 전달받으므로 주 생성자에서는 name 프로퍼티를 없앤다.

```
class Person(val name: String, age: Int, salary: Int) : Observable() {
    val _age = ObservableProperty("age", age, this)
    var age: Int
        get() = _age.getValue()
        set(newValue) {
            _age.setValue(newValue)
        }

    val _salary = ObservableProperty("salary", salary, this)
    var salary: Int
        get() = _salary.getValue()
        set(newValue) {
            _salary.setValue(newValue)
        }
}

// 위임 프로퍼티
class Person(val name: String, age: Int, salary: Int) : Observable() {
    var age by ObservableProperty(age, this)
    var salary by ObservableProperty(salary, this)
}
```

- by 키워드를 사용해 위임 객체를 지정하여 코틀린 컴파일러가 자동으로 준비코드 처리
- ObservableProperty: 위임 객체
- 위임 객체를 감춰진 프로퍼티에 저장하고 주 객체의 프로퍼티를 읽거나 쓸 때마다 위임 객체의 getValue와 setValue를 호출해준다.

```
import kotlin.properties.Delegates
import kotlin.reflect.KProperty

class Person(val name: String, age: Int, salary: Int) : Observable() {
    private val onChange = {
            property: KProperty<*>, oldValue: Any?,
            newValue: Any?,
```

```
        →
        notifyObservers(property.name, oldValue, newValue)
    }

    var age by Delegates.observable(age, onChange)
    var salary by Delegates.observable(salary, onChange)
}
```

- 표준 라이브러리에 존재하는 Delegates.observable 사용하여 프로퍼티 변경 통지 구현
- by 오른쪽에 있는 식이 꼭 새 인스턴스를 만들 필요하는 없다.
  - 함수 호출, 다른 프로퍼티, 다른 식 등이 올 수 있다.
  - 다만 오른쪽 식을 계산한 결과인 객체는 컴파일러가 호출할 수 있는 올바른 타입의 getValue와 setValue를 반드시 제공해야한다.

## 위임 프로퍼티는 커스텀 접근자가 있는 감춰진 프로퍼티로 변환된다.

```
class C {
    var prop: Type by MyDelegate()

}

val c = C()
```

- MyDelegate 클래스의 인스턴스는 감춰진 프로퍼티에 저장된다.
  - 그 프로퍼티를 <delegate>라는 이름으로 불러 보자.
- 컴파일러는 프로퍼티를 표현하기 위해 Kproperty 타입의 객체를 사용
  - <property> 라고 불러 보자

```
class C {

    private val <delegate> = MyDelegate()
```

```
    var prop: Type
        get() = <delegate>.getValue(this, <property>)
        set(value: Type) = <delegate>.setValue(this, <property>, value)


}
```

- 컴파일러는 모든 프로퍼티 접근자 안에 getValue와 setValue 호출 코드를 생성해준다.
    - val x = c.prop → val x  = <delegate>.getValue(c, <property>)
    - c.prop = x → <delegate>.setValue(c, <property>, x)
- 프로퍼티 값이 저장될 장소를 바꿀 수도 있고 프로퍼티를 읽거나 쓸 때 벌어질 일을 변경할 수도 있다.
    - 그것도 간결한 코드로
- 표준 라이브러리가 제공하는 위임 프로퍼티를 사용하는 방법을 하나 더 보자

## 맵에 위임해서 동적으로 애트리뷰트 접근

```
class Person {
    private val _attributes = mutableMapOf<String, String>()

    fun setAttribute(attrName: String, value: String) {
        _attributes[attrName] = value
    }

    var name: String
        get() = _attributes["name"]!!
        set(value) {
            _attributes["name"] = value
        }
}

fun main() {
    val p = Person()
    val data = mapOf("name" to "Seb", "company" to "JetBrains")
    for ((attrName, value) in data)
```

```
        p.setAttribute(attrName, value)
    println(p.name)
    // Seb
    p.name = "Sebastian"
    println(p.name)
    // Sebastian
}
```

- 자신의 프로퍼티를 동적으로 정의할 수 있는 객체를 만들 때 위임 프로퍼티를 활용 할 수 있다.
    - C#에서는 그런 객체를 확장 가능한 객체라고 부르기도 한다.
- eg) 연락처 관리 시스템에서 각 연락처별로 임의의 정보를 저장할 수 있게 허용하는 경우
    - 연락처에는 공통의 필수 정보와 사람 마다 다른 추가 정보가 있다.
- 위임 프로퍼티를 활용하도록 변경해보자

```
class Person {
    private val _attributes = mutableMapOf<String, String>()

    fun setAttribute(attrName: String, value: String) {
        _attributes[attrName] = value
    }

    var name: String by _attributes
}
```

- 이 코드가 작동하는 이유는 표준 라이브러리가 Map과 MutableMap 인터페이스에 대해 getValue와 setValue 확장 함수를 제공하기 때문
- getValue에서 맵에 프로퍼티 값을 저장할 때는 자동으로 프로퍼티 이름을 키로 활용한다.
- p.name
    - == `_attributes.getValue(p, prop)`

- == `_attributes[prop.name]`

## 실전 프레임워크가 위임 프로퍼티를 활용하는 방법

```
object Users : IdTable() { // 객체는 데이터베이스 테이블에 해당
    val name: Column<String> = varchar("name", length = 50).index() // 프로퍼티
    val age: Column<Int> = integer("age")

}

class User(id: EntityID) : Entity(id) { // User 인스턴스는 테이블에 들어있는 구체적인
    var name: String by Users.name // name 값은 데이터베이스에 저장된 사용자의 이
    var age: Int by Users.age
}
```

- Users 객체
  - 데이터베이스 테이블 표현, 싱글턴 객체로 선언
  - 객체의 프로퍼티는 테이블 칼럼 표현
- Entity class
  - db 칼럼을 엔티티의 속성 값으로 연결해주는 매핑이 있다.

```
operator fun <T> Column<T>.getValue(o: Entity, desc: KProperty<*>): T {
    // db에서 칼럼 값 가져오기
}

operator fun <T> Column<T>.setValue(o: Entity, desc: KProperty<*>, value: T
    // db의 값 변경하기
}
```

- Column
  - 프레임워크는 해당 클래스 안에 getValue와 setValue 메서드를 정의한다.
  - 이 두 메서드는 코틀린의 위임 객체 관례에 따른 시그니처 요구사항을 만족한다.
- 이 예제의 완전한 구현을 Exposed 프레임워크 소스코드에서 볼 수 있다.

## 요약

- 코틀린은 정해진 이름의 함수를 정의함으로써 표준적인 수학 연산을 오버로드할 수 있게 해준다.
    - 자신만의 연산자를 정의할 수는 없지만 중위 함수를 더 표현력이 좋은 대안으로 사용할 수 있다.
- 비교 연산자를 모든 객체에 사용할 수 있다.
    - 비교 연산자는 equals와 compareTo 메서드 호출로 변환된다.
- get, set, contains라는 함수를 정의하면 코틀린 컬렉션과 비슷하게 여러분 클래스의 인스턴스에 대해 []와 in 연산을 사용할 수 있다.
- 미리 정해진 관례를 따라 범위를 만들거나 컬렉션과 배열의 원소를 이터레이션할 수 있다.
- 구조 분해 선언을 통해 한 객체의 상태를 분해해서 여러 변수에 대입할 수 있다.
    - 함수가 여러 값을 한꺼번에 반환해야 하는 경우 유용하다.
    - 데이터 클래스에 대해 구조 분해를 거저 사용할 수 있지만, 자신의 클래스에 componentN 함수를 정의하면 구조 분해를 지원할 수 있다.
- 위임 프로퍼티를 통해 프로퍼티 값을 저장하거나 초기화하거나 읽거나 변경할 때 사용하는 로직을 재활용 할 수 있다.
    - 위임 프로퍼티는 프레임워크를 만들 때 아주 강력한 도구로 쓰인다.
- 표준 라이브러리 함수인 lazy를 통해 지연 초기화 프로퍼티를 쉽게 구현할 수 있다.
- Delegates.observable 함수를 사용하면 프로퍼티 변경을 관찰할 수 있는 옵저버를 쉽게 추가할 수 있다.
- 맵을 위임 객체로 사용하는 위임 프로퍼티를 통해 다양한 속성을 제공하는 객체를 유연하게 다룰 수 있다.