# 11장 도메인 특화 언어

도메인 특화 언어(Domain-Specific Language, DSL)는 특정 기능이나 영역을 위해 만들어진 언어를 뜻한다. 이런 언어는 소프트웨어를 개발하는 동안 다양한 작업에 활용하곤 한다. 예를 들어 소프트웨어 설정을 묘사하거나, 테스트 명세를 지정하거나, 작업흐름 규칙을 정의하거나, UI를 설계하거나, 데이터를 조작하는 등의 작업에 쓰인다. DSL의 주된 장점은 단순성이다. 자바 같은 범용 언어가 제공하는 저수준 기능에 의존하는 대신에 더 문제 영역에 적합한 기본 요소를 활용할 수 있기 때문에 문제를 문제가 속한 영역 자체의 용어로 기술할 수 있게 된다. 하지만 이런 접근 방식을 사용하면 DSL 코드를 범용 프로그램 내부에 내장시키기 어렵다. 그래서 보통은 DSL 프로그램을 호스트 언어 코드 외부에 저장하거나, 문자열 형태로 호스트 언어 코드에 내장시키곤 한다. 하지만 두 가지 방식 모두 컴파일 시간 검증과 IDE 코드 지원 기능 활용이 어려워진다는 문제가 있다.

하지만 코틀린은 이런 문제에 대한 해법을 제공한다. 이 장에서는 나머지 코틀린 코드와 매끄럽게 통합될 수 있는 DSL을 설계하고 싶을 때 도움이 될 수 있는 코틀린 기능을 몇 가지 소개한다. 아이디어는 코틀린으로 도메인 특화 언어를 닮은 특별한 API를 설계하는 것이다. 이렇게 하면 여러분의 코드가 다른 언어로 작성한 코드처럼 보이지만 여전히 올바른 코틀린 코드로 남아있을 수 있다. 즉, 이런 기능을 활용하면 강력한 타입 안전성 보장 등을 포함하는 컴파일 언어의 능력과 도메인 특화 언어를 사용한 접근 방식의 장점을 모두 취할 수 있다.

## 구조

* 연산자 오버로딩
* 위임 프로퍼티
* 고차 함수와 DSL

## 목표

코틀린의 고급 기능을 배우고, 이를 통해 영역 특화 언어 형태의 API를 설계할 수 있다.

## 연산자 오버로딩

연산자 오버로딩은 +, -, \*, / 등 코틀린 내장 연산자에 대해 새로운 의미를 부여할 수 있게 해주는 언어 기능이다. 앞에서 + 연산의 의미가 연산이 적용된 타입에 따라 달라진다는 사실을 이미 살펴봤다. +는 수 타입에 대해서는 덧셈 연산이지만, 문자열의 경우에는 연결 연산이고, 컬렉션의 경우에는 원소를 맨 뒤에 덧붙이는 연산이 되는 식이다. 이런 일이 벌어질 수 있는 이유는 +가 오버로딩되어 있어서 다양한 구현을 제공하기 때문이다.

코틀린에서 연산 식은 단지 함수 호출에 대한 편의 문법(syntatic sugar)일 뿐이다. 연산자를 구현하고 싶으면 정해진 관습을 따라 확장함수나 멤버 함수를 정의하고 operator 키워드를 붙이면 된다. 예를 들어 다음 정의는,

<코드>

operator fun String.times(n: Int) = repeat(n)

</코드>

\* 연산자(이 연산자는 times() 함수에 해당된다)를 String과 Int의 쌍에 대해 확장해 정의한다. 이로 인해 다음과 같은 코드를 작성할 수 있다.

<코드>

println(“abc” \* 3) // abcabcabc

</코드>

이런 식으로 연산자를 미리 정해진 몇 가지 함수를 통해 뒷받침하기 때문에, 항상 연산 식을 일반 함수 호출로 바꿀 수 있다. 예를 들어 바로 앞의 코드는 다음 코드와 같다.

<코드>

println(“abc”.times(3))

</코드>

덧셈 연산자 같은 내장 연산자까지도 이런 형태로 작성할 수 있다. 하지만 성능을 위해, 원시 타입의 경우에는 컴파일러가 덧셈, 뺄셈 등의 연산을 함수를 호출하지 않고 계산할 수 있게 최적화 해준다.

<코드>

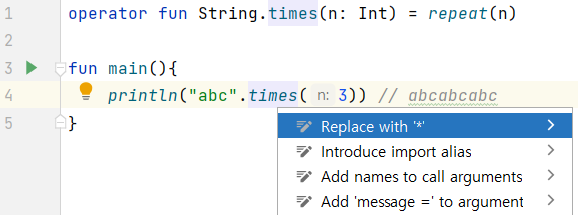
val x = 1.plus(2) // the same as 1 + 2

</코드>

<참고> IDE 팁

인텔리J 코틀린 플러그인은 연산자 함수를 명시적으로 호출한 부분을 그에 대응하는 단항/이항 연산식으로 바꿀 수 있다. 연산자 토큰이나 함수 이름에 캐럿을 위치시키고 [Alt]+[Enter]를 눌러서 적절한 변환을 택하면 된다. 그림 11-1은 times()를 이항 \*로 바꾸는 변환을 보여준다.

그림 11-1: 명시적인 호출을 연산자 형태로 바꾸기



다음 절에서는 코틀린 연산자와 관련한 관습에 대해 알아보고, 연산자 함수 구현 방법을 몇 가지 살펴보겠다.

### 단항 연산

오버로드할 수 있는 단항 연산자로는 전위 +나 -, ! 연산자가 있다. 컴파일러는 이런 연산자를 적절한 함수 호출로 펼쳐준다(표 11-1).

표 11-1: 단항 연산자 관습

|  |  |
| --- | --- |
| 식 | 의미 |
| +e | e.unaryPlus() |
| -e | e.unaryMinus() |
| !e | e.not() |

이런 함수를 인자 식의 타입에 대해 멤버 함수나 확장 함수로 정의할 수 있다. 단항 연산자에 해당하는 함수는 아무 파라미터도 받지 않고, 함수의 반환 타입이 전체 단항 연산의 반환 타입이 된다.

예를 들어 RGB와 RGB의 조합으로 표현할 수 있는 색을 표현하는 이넘 클래스를 생각해보자.

<코드>

enum class Color {  
 BLACK, RED, GREEN, BLUE, YELLOW, CYAN, MAGENTA, WHITE;  
}

</코드>

not() 관습을 사용하면 보색 관계를 ! 연산자로 표현할 수 있다.

<코드>

enum class Color {  
 BLACK, RED, GREEN, BLUE, YELLOW, CYAN, MAGENTA, WHITE;  
  
 operator fun not() = when (this) {  
 BLACK -> WHITE  
 RED -> CYAN  
 GREEN -> MAGENTA  
 BLUE -> YELLOW  
 YELLOW -> BLUE  
 CYAN -> RED  
 MAGENTA -> GREEN  
 WHITE -> BLACK  
 }  
}  
  
fun main() {  
 println(!Color.RED) // CYAN  
 println(!Color.CYAN) // RED  
}

</코드>

연산자 함수를 확장 함수로 정의함으로써 임의의 타입에 대한 연산자를 오버로딩할 수 있다. 예를 들면 다음과 같다.

<코드>

operator fun <T> ((T) -> Boolean).not(): (T) -> Boolean = { !this(it) }

</코드>

위 함수 정의를 사용하면 ! 연산자를 파라미터가 하나뿐인 술어에 적용할 수 있다.

<코드>

fun isShort(s: String) = s.length <= 4  
fun String.isUpperCase() = all { it.isUpperCase() }  
  
fun main() {  
 val data = listOf("abc", "abcde", "ABCDE", "aBcD", "ab")  
   
 println(data.count(::isShort)) // 3  
 println(data.count(!::isShort)) // 2  
 println(data.count(String::isUpperCase)) // 1  
 println(data.count(!String::isUpperCase)) // 4  
}

</코드>

### 증가와 감소

증가(++)와 감소(--) 연산자도 피연산자 타입에 대한 파라미터가 없는 inc()와 dec() 함수로 오버로드할 수 있다. 이런 함수의 반환 타입은 항상 증가 이후의 값이나 증가 이전의 값과 같은 타입이어야 한다. inc()/dec()가 사용되는 방법은 연산자가 전위 연산자 형태로 쓰이느냐 후위 연산자 형태로 쓰이느냐에 따라 달라진다. 예를 들어 무지개 색을 표현하는 이넘 클래스를 보자.

<코드>

enum class RainbowColor {  
 RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET;  
}

</코드>

위에 정의된 색의 순서에 따라 inc()와 dec()를 정의해보자. 맨 마지막 색과 맨 앞의 색의 경우에는 목록의 맨 뒤나 맨 앞으로 돌아간다. 즉, VIOLET의 다음은 RED이고, RED의 앞은 VIOLET이다.

<코드>

enum class RainbowColor {  
 RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET;  
 operator fun inc() = values[(ordinal + 1) % values.size]  
 operator fun dec() = values[(ordinal + values.size - 1) % values.size]  
 companion object {  
 private val values = enumValues<RainbowColor>()  
 }  
}

</코드>

이제 이 클래스의 증가/감소 연산이 어떻게 작동하는지 살펴보자. **2장 코틀린 언어 기초**에서 본 것처럼 ++/-- 연산의 후위 형태는 변수를 갱신하지만 갱신 이전의 값을 반환한다. 오버로드한 연산자의 경우에도 마찬가지다. 다음 코드를 생각해보자.

<코드>

var color = RainbowColor.INDIGO  
println(color++)

</코드>

실제 이 코드는 다음과 같은 효과를 나타낸다.

<코드>

var color = RainbowColor.INDIGO  
val \_oldColor = color  
color = color.inc()  
println(\_oldColor) // INDIGO

</코드>

전위 형태의 경우 증가/감소식의 결과가 갱신된 값과 똑같다. 다음 예는

<코드>

var color = RainbowColor.INDIGO  
println(color++)

</코드>

실제로 다음과 같이 변환된다.

<코드>

var color = RainbowColor.INDIGO  
color = color.inc()  
println(color) // VIOLET

</코드>

여기서 color = color.inc() 같은 대입문이 있다는 것은 다음 두 가지를 암시한다.

* 가변 변수에만 ++와 --를 적용할 수 있다.
* inc()/dec()의 반환 타입은 수신 객체 타입의 하위 타입 이어야 한다.

### 이항 연산

코틀린에서는 대부분의 이항 연산자를 오버로드할 수 있다. 단항 연산과 마찬가지로 정해진 이름의 연산자 함수를 정의하면 된다. 가장 큰 차이는 이항 연산자 함수는 대부분의 경우 왼쪽 피연산자를 수신 객체로, 오른쪽 피연산자를 일반적인 인자로 받는다는 점이다.

표 11-2는 산술 연산자, .., in, !in을 대신하는 함수의 관습적인 이름이다.

표 11-2: 이항 연산자 관습

|  |  |
| --- | --- |
| 식 | 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) |
| a .. b | a.rangeTo(b) |
| a in b | b.contains(a) |
| a !in b | !b.contains(a) |

처음에는 % 연산자에 해당하는 이름이 mod() 연산자 함수였지만, 현재는 rem()이 %를 대신한다. 현재도 mod() 관습을 사용할 수 있기는 하나, 사용 금지 예정이다.

예를 들어 기본 산술 연산을 지원하는 간단한 유리수(rational number) 시스템의 프로토타입을 생각해보자.

<코드>

import kotlin.math.abs  
  
class Rational private constructor(  
 val sign: Int,  
 val num: Int,  
 val den: Int  
) {  
 operator fun unaryMinus() = Rational(-sign, num, den)  
   
 operator fun plus(r: Rational): Rational {  
 val gcd = gcd(den, r.den)  
 val newDen = den/gcd\*r.den  
 val newNum = newDen/den\*num\*sign + newDen/r.den\*r.num\*r.sign  
 val newSign = newNum.sign()  
   
 return Rational(newSign, abs(newNum), newDen)  
 }  
   
 operator fun minus(r: Rational) = this + (-r)  
   
 operator fun times(r: Rational): Rational {  
 return of(sign\*r.sign\*num\*r.num, den\*r.den)  
 }  
   
 operator fun div(r: Rational): Rational {  
 return of(sign\*r.sign\*num\*r.den, den\*r.num)  
 }  
   
 override fun toString(): String {  
 return "${sign\*num}" + if (den != 1) "/$den" else ""  
 }  
   
 companion object {  
 private fun Int.sign() = when {  
 this > 0 -> 1  
 this < 0 -> -1  
 else -> 0  
 }  
   
 private tailrec fun gcd(a: Int, b: Int): Int {  
 return if (b == 0) a else gcd(b, a % b)  
 }  
   
 fun of(num: Int, den: Int = 1): Rational {  
 if (den == 0) throw ArithmeticException("Denominator is zero")  
   
 val sign = num.sign() \* den.sign()  
 val numAbs = abs(num)  
 val denAbs = abs(den)  
 val gcd = gcd(numAbs, denAbs)  
   
 return Rational(sign, numAbs/gcd, denAbs/gcd)  
 }  
 }  
}

</코드>

연산자 관습을 사용하면 Rational 인스턴스의 산술 연산을 편하게 사용할 수 있다.

<코드>

fun r(num: Int, den: Int = 1) = Rational.of(num, den)  
  
fun main() {  
 // 1/2 - 1/3  
 println(r(1, 2) - r(1, 3)) // 1/6  
   
 // 2/3 + (1/3)/2  
 println(r(2, 3) + r(1, 3)/r(2)) // 5/6  
   
 // 3/4 \* 8/9 / (2/3)  
 println(r(3, 4)\*r(8, 9)/r(2, 3)) // 1  
   
 // (1/10)\*2 - 2/6  
 println(r(1, 10)\*r(2) - r(2, 6)) // -2/15  
}

</코드>

일부 연산자 함수를 추가할 수도 있다. 이를 통해 Rational 객체와 Int 등의 다른 수를 혼용할 수 있다.

<코드>

operator fun Rational.plus(n: Int) = this + Rational.of(n)  
  
operator fun Int.plus(r: Rational) = r + this  
  
operator fun Rational.minus(n: Int) = this - Rational.of(n)  
  
operator fun Int.minus(r: Rational) = Rational.of(this) - r  
  
fun main() {  
 // -1/3 + 2  
 println(r(-1, 3) + 2) // 5/3  
  
 // 1 - (1/4)\*(1/2)  
 println(1 - r(1, 4)\*r(1, 2)) // 7/8  
}

</코드>

.. 연산자 사용을 보여주기 위해 두 유리수 사이의 구간을 표현하는 RationalRange 클래스를 정의해보자.

<코드>

class RationalRange(val from: Rational, val to: Rational) {  
 override fun toString() = "[$from, $to]"  
}

</코드>

이제 rangeTo() 함수를 정의할 수 있다. 이 함수는 RationalRange 인스턴스를 만들어준다.

<코드>

operator fun Rational.rangeTo(r: Rational) = RationalRange(this, r)  
  
fun main() {  
 println(r(1, 4)..r(1)) // [1/4, 1]  
}

</코드>

in/!in 연산은 contains() 연산자 함수에 의해 표현된다. 여기서 다른 이항 연산과 contains()의 인자 방향이 반대라는 사실에 유의하라. RationalRange 클래스를 주어진 수가 구간 안에 들어 있는지 검사할 수 있게 개선해보자.

<코드>

private fun Rational.isLessOrEqual(r: Rational): Boolean {  
 return num\*r.den <= r.num\*den  
}  
  
class RationalRange(val from: Rational, val to: Rational) {  
 override fun toString() = "[$from, $to]"  
   
 operator fun contains(r: Rational): Boolean {  
 return from.isLessOrEqual(r) &&r.isLessOrEqual(to)  
 }  
   
 operator fun contains(n: Int) = contains(r(n))  
}  
  
fun main() {  
 // 1/2 in [1/4, 1]  
 println(r(1, 2) in r(1, 4)..r(1)) // true  
   
 // 1 not in [5/4, 7/4]  
 println(1 !in r(5, 4)..r(7, 4)) // true  
}

</코드>

<나 > 같이 비교와 관련 있는 오버로드 가능 연산자도 있다. 이 연산자들은 서로 다른 함수에 대응하지 않는다. 대신 compareTo()라는 함수를 사용해 주어진 피연산자 타입에 대한 모든 비교 연산을 구현한다. compareTo() 함수는 비교 결과를 알려주는 Int 값을 반환한다. compareTo() 함수로부터 파생되는 모든 비교 연산을 표 11-3에 정리해 두었다.

표 11-3: 비교 연산자 관습

|  |  |
| --- | --- |
| 식 | 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 |

이제 앞의 예제에서 isLessThan() 함수를 없애고 더 일반적인 compareTo() 구현으로 바꿀 수 있다.

<코드>

operator fun Rational.compareTo(r: Rational): Int {  
 val left = num \* r.den  
 val right = r.num \* den  
   
 return when {  
 left < right -> -1  
 left > right -> 1  
 else -> 0  
 }  
}  
  
operator fun Rational.compareTo(n: Int) = compareTo(r(n))  
  
operator fun Int.compareTo(r: Rational) = -r.compareTo(this)  
  
class RationalRange(val from: Rational, val to: Rational) {  
 override fun toString() = "[$from, $to]"  
  
 operator fun contains(r: Rational) = r >= from && r <= to  
  
 operator fun contains(n: Int) = contains(r(n))  
}  
  
fun main() {  
 println(1 > r(1, 3)) // false  
 println(r(3/4) <= r(7/8)) // true  
}

</코드>

이제 이항 연산자 관습이 한 가지 남았는데, 이전에 다룬 동등성 관련 연산이 있다. ==나 !=를 사용하면 컴파일러는 equals() 함수를 호출해준다. 여기서 equals() 구현이 Any 클래스에 정의된 기반 구현을 상속하기 때문에 명시적인 operator 변경자를 붙이지 않아도 된다는 점에 유의하라. 같은 이유로 equals()를 항상 멤버 함수로 정의해야 한다. 확장함수로 ==나 != 연산자를 대신하는 equals()를 정의하더라도 항상 시그니처가 같은 멤버 함수가 확장 함수보다 우선되기 때문에 새로 정의한 확장 함수가 가려져 버린다.

코틀린에서는 &&와 ||를 오버로드 할 수 없다. 이들은 Boolean 값에 대해서만 지원되는 내장 연산이다. 참조 동등성 연산자인 ===나 !==도 마찬가지로 오버로드할 수 없는 내장 연산이다.

여러분이 원하는 이름으로 이항 연산을 정의하고 싶다면 어떻게 해야 할지 생각해보라. 코틀린은 프로그래머가 새 연산자를 정의하도록 허용하지는 않지만, 이름으로 쓰일 수 있는 일반적인 식별자를 중위 연산자로 사용하도록 허용한다. 이에 대해서는 다음 절에서 살펴보겠다.

## 중위 연산

중위 연산으로 쓸 수 있는 to나 until 같은 연산을 앞에서 이미 본 적이 있다.

<코드>

val pair1 = 1 to 2 // 중위 호출  
val pair2 = 1.to(2) // 일반적인 호출

</코드>

이런 호출을 가능하게 하려면 함수 앞에 infix 변경자를 붙여야 한다. 이항 연산자와 마찬가지로 infix를 붙일 수 있는 함수는 파라미터가 하나인 멤버나 확장 함수여야 한다. 예를 들어 표준 to 함수 구현은 다음과 같다.

<코드>

infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)

</코드>

앞에서 살펴본 술어 함수를 조금 바꿔서 술어의 논리곱(conjunction)과 논리합(disjunction)을 표현하는 중위 연산을 정의하자.

<코드>

infix fun <T> ((T) -> Boolean).and(  
 other: (T) -> Boolean  
): (T) -> Boolean {  
 return { this(it) && other(it) }  
}  
  
infix fun <T> ((T) -> Boolean).or(  
 other: (T) -> Boolean  
): (T) -> Boolean {  
 return { this(it) || other(it) }  
}

</코드>

이제 함수 리터럴을 좀 더 간결하게 조합해 새로운 술어를 만들 수 있다.

<코드>

fun main() {  
 val data = listOf("abc", "abcde", "ABCDE", "aBcD", "ab")  
   
 println(data.count(::isShort and String::isUpperCase)) // 0  
 println(data.count(::isShort or String::isUpperCase)) // 4  
 println(data.count(!::isShort or String::isUpperCase)) // 2  
 println(data.count(!(::isShort and String::isUpperCase))) // 5  
}

</코드>

이렇게 정의한 모든 중위 연산의 우선순위가 같다는 점에 유의하라. 예를 들어 바로 앞 예제에 있는 and와 or를 사용하는 복잡한 연산은 내장 ||와 && 연산자를 사용한 비슷한 의미의 식과는 다른 방식으로 파싱된다. 예를 들어 다음 식은,

<코드>

!::isShort or String::isEmpty and String::isUpperCase

</코드>

다음과 같은 의미다.

<코드>

(!::isShort or String::isEmpty) and String::isUpperCase

</코드>

반면 다음 불 식은,

<코드>

!s.isShort() || s.isEmpty() &&s.isUpperCase()

</코드>

다음 식과 의미가 같다.

<코드>

!s.isShort() || (s.isEmpty() &&s.isUpperCase())

</코드>

이유는 &&의 우선순위가 ||보다 더 높기 때문이다.

### 대입

또 다른 이항 연산 그룹으로 +=와 같은 복합 대입 연산이 있다. **7장 컬렉션과 I/O 자세히 알아보기**에서 가변 컬렉션과 불변 컬렉션에 따라 이런 연산의 동작이 달라진다고 설명했다. 정리하면 +=를 불변 컬렉션 타입의 변수에 적용하면 새로운 컬렉션 객체가 생기고 이 객체를 변수에 대입해 변수 값이 바뀐다. 따라서 이런 경우에는 변수가 가변이어야 한다.

<코드>

var numbers = listOf(1, 2, 3)  
numbers += 4  
println(numbers) // [1, 2, 3, 4]

</코드>

하지만 +=을 가변 컬렉션에 적용하면 컬렉션의 내용은 바뀌지만 객체 자체의 정체성은 바뀌지 않는다.

<코드>

val numbers = mutableListOf(1, 2, 3)  
numbers += 4  
println(numbers) // [1, 2, 3, 4]

</코드>

가변 컬렉션을 가변 변수에 넣으면 컴파일러가 어떤 관습을 따라야 할지 결정할 수 없기 때문에 += 연산에서 오류가 발생한다.

<코드>

var numbers = mutableListOf(1, 2, 3)  
// 변수에 저장된 참조를 바꿔야 할까 객체 내부를 바꿔야 할까?  
numbers += 4 // Error  
println(numbers)

</코드>

임의의 타입에 대해 이 두 가지 관습을 적용할 수 있다. 다만 관련된 연산자 함수를 오버로드해야 한다. 복합 대입문의 동작은 다음과 같은 요소에 따라 달라진다(표 11-4).

* 복합 대입문에 대응하는 이항 연산자 함수가 이미 있는지 여부: 예를 들어 +=의 경우 plus(), -=의 경우 minus() 등이 있는지에 따라
* 커스텀 대입 함수가 있는지 여부: +=의 경우 plusAssign()이 있는지, -=의 경우 minusAssign()이 있는지 등에 따라
* 복합 대입 연산자의 왼쪽 피연산자에 대한 가변 대입이 가능한지 여부에 따라

표 11-4: 대입 연산자 관습

|  |  |  |
| --- | --- | --- |
| 식 | 의미 | 의미 |
| a += b | a = a.plus(b) | a.plusAssign(b) |
| a -= b | a = a.minus(b) | a.minusAssign(b) |
| a \*= b | a = a.times(b) | a.timesAssign(b) |
| a /= b | a = a.div(b) | a.divAssign(b) |
| a %= b | a = a.rem(b) | a.remAssign(b) |

있을 수 있는 경우를 생각해보자. 왼쪽 피연산자에 대응하는 연산자(예: plus())가 있지만 커스텀 대입 연산자 함수는 없는 경우, 복합 대입 연산은 단순한 연산식으로 변환된다. 불변 컬렉션이나 원시 타입에 대해서도 이런 일이 발생한다. 우리가 만든 Rational 객체는 이미 +나 -등을 지원하기 때문에 이 객체에 대해 복합 대입문을 거저 사용할 수 있다.

<코드>

var r = r(1, 2) // ½  
  
// r = r + r(1, 3) 와 같음  
r += r(1, 3) // 1/2 + 1/3  
  
println(r) // 5/6

</코드>

이때 대입 연산자의 왼쪽 피연산자는 반드시 가변 변수여야 한다는 점에 유의하라.

대입 연산자의 왼쪽 피연산자에 대한 커스텀 대입 함수만 있는 경우(예: plusAssign()은 있지만 plus()는 없는 경우), 복합 대입문이 이 커스텀 대입 함수 호출로 바뀐다. **9장 제네릭스**에서 소개한 TreeNode 클래스의 API를 약간 변경해 다시 살펴보면서 이 관습에 대해 알아보자.

<코드>

class TreeNode<T>(val data: T) {  
 private val \_children = arrayListOf<TreeNode<T>>()  
   
 var parent: TreeNode<T>? = null  
 private set  
   
 operator fun plusAssign(data: T) {  
 val node = TreeNode(data)  
 \_children += node  
 node.parent = this  
 }  
 operator fun minusAssign(data: T) {  
 val index = \_children.indexOfFirst { it.data == data }  
 if (index < 0) return  
 val node = \_children.removeAt(index)  
 node.parent = null  
 }  
   
 override fun toString() =  
 \_children.joinToString(prefix = "$data {", postfix = "}")  
 }

</코드>

이제 TreeNode 인스턴스에 대해 +=와 -= 연산자를 사용해서 트리 원소를 추가하고 제거해보자.

<코드>

val tree = TreeNode("root")  
tree += "child 1"  
tree += "child 2"  
println(tree) // root {child 1 {}, child 2 {}}  
  
tree -= "child 2"  
println(tree) // root {child 1 {}}

</코드>

커스텀 대입 함수의 반환 타입은 Unit이어야 한다는 점에 유의하라.

왼쪽 피연산자가 커스텀 대입 함수와 간단한 이항 연산을 함께 제공하는 경우에는 왼쪽 피연산자가 불변인지 가변인지에 따라 처리 방식이 달라진다.

* 왼쪽 피연산자가 불변 변수라면 간단한 대입을 사용할 수 없기 때문에, 컴파일러는 커스텀 대입 함수를 선택한다.
* 왼쪽 피연산자가 가변 변수라면 a+=b가 a = a + b를 뜻하는지 a.plusAssign(b)를 뜻하는지 모호하기 때문에, 컴파일러는 오류를 보고한다.

코틀린 가변 컬렉션 클래스인 리스트나 집합이 방금 설명한 동작을 잘 보여준다. 이들은 모두 불변 컬렉션에서 상속받은 plus()/minus() 함수가 있고, 자체적으로 plusAssign()/minusAssign()도 정의하고 있다.

### 호출과 인덱스로 원소 찾기

호출 관습을 사용하면 값을 함수처럼 호출 식에서 사용할 수 있다. 필요한 파라미터와 함께 invoke() 함수를 정의하면 된다. 함수 타입의 값은 자동으로 invoke() 멤버가 생긴다. 하지만 원한다면 임의의 타입에 대해 invoke()를 정의할 수 있다. 예를 들어 다음 함수를 생각해 보자.

<코드>

operator fun <K, V> Map<K, V>.invoke(key: K) = get(key)

</코드>

이렇게 정의하면, 키를 넣으면 값이 나오는 함수처럼 Map 인스턴스를 사용할 수 있다.

<코드>

val map = mapOf("I" to 1, "V" to 5, "X" to 10)  
println(map("V")) // 5  
println(map("L")) // null

</코드>

유용한 용례로, invoke() 함수를 동반 객체에 넣어서 동반 객체를 팩터리로 만드는 방법을 들 수 있다. 예를 들어 Rational 함수를 확장해 필터리를 보충할 수 있다.

<코드>

operator fun Rational.Companion.invoke(num: Int, den: Int = 1) =  
 of(num, den)

</코드>

이제 클래스 이름을 참조해 Rational 인스턴스를 만들 수 있다.

<코드>

val r = Rational(1, 2)

</코드>

이 코드는 마치 직접적인 생성자 호출처럼 보인다. 하지만 사실은 다음과 같은 호출 경로를 거친다.

invoke() → of() → Rational의 비공개 생성자라

문자열, 배열, 리스트 등의 여러 객체에 대해 인덱스 연산자 []를 적용할 수 있는 것도 비슷한 관습 덕분이다. 인덱스 연산자 호출에 대한 내부적인 번역은 인덱스 연산자 식이 값으로 쓰이느냐 대입의 왼쪽에 있느냐에 따라 달라진다. 전자의 경우 컴파일러는 데이터 읽기 접근을 가정하기 때문에 인덱스 연산자를 get() 함수 호출로 변환하고 인자에 인덱스를 넣어준다.

<코드>

val array = arrayOf(1, 2, 3)  
println(array[0]) // println(array.get(0))와 같음

</코드>

하지만 인덱스 식이 대입 연산의 왼쪽에 있으면 컴파일러는 이를 set() 함수 호출로 만들어준다. 이 경우에는 인덱스를 첫 번째 인자로, 설정하려는 값을 두 번째 인자로 set()에 전달한다.

<코드>

val array = arrayOf(1, 2, 3)  
array[0] = 10 // array.set(0, 10)과 같음

</코드>

인덱스가 꼭 정수일 필요도 없고, 어떤 타입의 값이든 문제 없다. 예를 들어 맵의 인덱스 연산은 키 값을 인자로 받는다.

예를 들어 TreeNode 클래스에 자식에 접근할 수 있는 get()과 set() 연산자를 추가해보자.

<코드>

class TreeNode<T>(var data: T) {  
 private val \_children = arrayListOf<TreeNode<T>>()  
   
 var parent: TreeNode<T>? = null  
 private set  
   
 operator fun plusAssign(data: T) {  
 val node = TreeNode(data)  
 \_children += node  
 node.parent = this  
 }  
   
 operator fun minusAssign(data: T) {  
 val index = \_children.indexOfFirst { it.data == data }  
 if (index < 0) return  
 val node = \_children.removeAt(index)  
 node.parent = null  
 }  
   
 operator fun get(index: Int) = \_children[index]  
   
 operator fun set(index: Int, node: TreeNode<T>) {  
 node.parent?.\_children?.remove(node)  
 node.parent = this  
 \_children[index].parent = null  
 \_children[index] = node  
 }  
}  
  
fun main() {  
 val root = TreeNode("Root")  
   
 root += "Child 1"  
 root += "Child 2"  
 println(root[1].data) // Child 2  
   
 root[0] = TreeNode("Child 3")  
 println(root[0].data) // Child 3  
}

</코드>

더 복잡한 경우는 인덱스 연산자를 복합 대입 연산자에 사용하는 경우다. 이때 만들어지는 결과 코드는 왼쪽 피연산자가 속한 타입의 대입 연산자의 의미에 따라 달라지는데, 보통 이 (왼쪽 피연산자의) 타입은 get() 연산자 함수의 결과 타입과 같다. 예를 들어 plusAssign() 함수가 없는 Rational 객체의 배열에 대한 코드를 생각해보자. 다음 코드는,

<코드>

val array = arrayOf(r(1, 2), r(2, 3))  
array[0] += Rational(1, 3)

</코드>

다음을 의미하거나,

<코드>

val array = arrayOf(r(1, 2), r(2, 3))  
array[0] = array[0] + r(1, 3)

</코드>

모든 연산을 함수 호출로 바꾸면 다음을 의미한다.

<코드>

val array = arrayOf(r(1, 2), r(2, 3))  
array.set(0, array.get(0) + r(1, 3))

</코드>

그러나 plus()는 없지만 plusAssign() 함수가 정의되어 있는 TreeNode 객체의 배열을 사용하는 경우 다음 코드는,

<코드>

val array = arrayOf(TreeNode(“Root 1”), TreeNode(“Root 2”))  
array[0] += TreeNode(“Child 1”)

</코드>

다음과 같이 변환된다.

<코드>

val array = arrayOf(TreeNode(“Root 1”), TreeNode(“Root 2”))  
array.get(0).plusAssign(TreeNode(“Child 1”))

</코드>

어느 경우든 get() 함수가 필요하다는 점에 주의하라.

### 구조분해

데이터 클래스 인스턴스로부터 한번에 여러 프로퍼티를 읽어서 여러 가지 변수에 대입해주는 구조분해 선언에 대해 이미 살펴봤다. 연산자 오버로딩을 사용하면 임의의 타입에 대해 구조분해를 제공할 수 있다. 여러분이 해야 할 일은 파라미터가 없는 componentN() 이라는 이름의 컴포넌트 함수를 멤버 함수나 확장 함수로 정의하는 것뿐이다. 여기서 N은 1부터 시작하는 정수다. 이 함수를 정의하고 나면, 구조분해 선언의 각 변수는 각각의 순서에 따라 (1부터 정해지는) componentN() 함수의 반환 값과 타입으로 정해진다.

이 관습에 대해 알아보겠다. 앞에서 살펴본 RationalRange 클래스에 대해 컴포넌트 함수를 정의하자.

<코드>

operator fun RationalRange.component1() = from  
operator fun RationalRange.component2() = to

</코드>

이제 RationalRange 인스턴스에 대해 구조분해를 사용할 수 있다.

<코드>

fun main() {  
 val (from, to) = r(1, 3)..r(1, 2)  
   
 println(from) // 1/3  
 println(to) // 1/2  
}

</코드>

데이터 클래스도 이런 측면에서는 예외가 아니다. 단지 여러분이 직접 컴포넌트 함수를 명시하지 않아도 컴파일러가 알아서 함수들을 생성해주는 것뿐이다. 코틀린 표준 라이브러리에는 몇 가지 확장 컴포넌트 함수가 들어있다. 예를 들어 이런 함수로 맵 원소 타입(키/값 쌍)을 구조분해할 수 있다.

<코드>

val map = mapOf("I" to 1, "V" to 5, "X" to 10)  
for ((key, value) in map) {  
 println("$key = $value")  
}

</코드>

또는 리스트나 배열에서 몇 가지 원소를 추출할 수 있다.

<코드>

val numbers = listOf(10, 20, 30, 40, 50)  
val (a, b, c) = numbers  
println("$a, $b, $c") // 10, 20, 30

</코드>

### 이터레이션

**3장 함수 정의하기**에서 for 루프에 대해 설명했다. 문자열이나 범위, 컬렉션 등의 객체에 for 루프를 적용할 수 있다. 이런 식으로 for 루프를 적용할 수 있는 타입들의 공통점은 바로 iterator() 함수가 들어있다는 점이다. iterator() 함수는 Iterator 타입의 이터레이터 인스턴스를 반환한다. 여러분도 원하는 타입에 대해 iterator() 함수를 멤버나 확장으로 정의하면 for 루프를 사용할 수 있다.

예를 들어 앞에서 본 TreeNode 클래스에 대한 이터레이션을 지원해보자.

<코드>

operator fun <T>TreeNode<T>.iterator() = children.iterator()

</코드>

이제 children 멤버를 명시적으로 참조하지 않아도 TreeNode 인스턴스를 for 루프에 사용해서 자식 노드를 다룰 수 있다.

<코드>

fun main() {  
 val content = TreeNode("Title").apply {  
 addChild("Topic 1").apply {  
 addChild("Topic 1.1")  
 addChild("Topic 1.2")  
 }  
 addChild("Topic 2")  
 addChild("Topic 3")  
 }  
 for (item in content) {  
 println(item.data)  
 }  
}

</코드>

이 코드를 실행하면 다음과 같은 출력을 볼 수 있다.

<코드>

Topic 1  
Topic 2  
Topic 3

</코드>

이것으로 코틀린 연산자 오버로딩에 대한 설명을 마친다. 다음 절에서는 위임 메커니즘에 대해 논의할 것이다. 위임을 사용하면 새로운 종류의 프로퍼티를 코틀린 코드에 도입할 수 있다.

## 위임 프로퍼티

위임 프로퍼티를 사용하면 간단한 문법적인 장식 뒤에 커스텀 프로퍼티 접근 로직을 구현할 수 있다. 이미 lazy 위임을 통해 최초 접근 시까지 프로퍼티 계산을 지연시키는 경우를 살펴봤다.

<코드>

val result by lazy { 1 + 2 }

</코드>

프로퍼티 위임을 알아두면 사용하기 편한 API와 DSL을 설계할 때 도움이 된다.

앞 절에서 설명한 연산자와 마찬가지로 위임 프로퍼티 구현도 몇 가지 관습에 따라 이뤄진다. 이 관습을 통해 여러분은 프로퍼티를 읽고 쓰는 방법을 정의할 수 있고, 위임 객체 자체의 생성을 제어할 수 있다. 이 절에서는 이런 관습을 자세히 다루고, 코틀린 표준 라이브러리가 제공하는 위임 기능에 대해서도 설명하겠다.

### 표준 위임들

코틀린 표준 라이브러리에는 다양한 용례를 지원하는 바로 써먹을 수 있는 몇 가지 위임 구현이 들어있다. **4장 클래스와 객체 다루기**에서 이런 위임의 예로 지연 계산 프로퍼티를 표현하는 위임을 이미 살펴봤다.

<코드>

val text by lazy { File("data.txt").readText() }

</코드>

사실 lazy() 함수에는 다중스레드 환경에서 지연 계산 프로퍼티의 동작을 미세하게 제어하기 위해 세 가지 다른 버전을 가지고 있다. 디폴트 lazy()는 스레드 안전한 구현을 만들어낸다. 이 구현은 동기화를 사용해 지연 계산된 값이 항상 한 가지 스레드에 의해서만 초기화되게 보장한다. 이 경우 위임 인스턴스가 동기화 객체 역할까지 수행한다.

필요하다면 다른 lazy() 버전을 사용해서 여러분이 원하는 동기화 객체를 지정할 수도 있다.

<코드>

private val lock = Any()  
val text by lazy(lock) { File("data.txt").readText() }

</코드>

그리고 LazyThreadSafetyMode 이넘 상수를 통해서 세 가지 기본 구현 중 하나를 선택할 수 있다.

* SYNCHRONIZED: 프로퍼티 접근을 동기화한다. 따라서 한번에 한 스레드만 프로퍼티 값을 초기화할 수 있다(이 구현이 디폴트다)
* PUBLICATION: 초기화 함수가 여러 번 호출될 수 있지만 가장 처음 도착하는 결과가 프로퍼티 값이 되도록 프로퍼티 접근을 동기화한다
* NONE: 프로퍼티 접근을 동기화하지 않는다. 이 방식을 선택하면 다중 스레드 환경에서 프로퍼티의 올바른 동작을 보장할 수 없다.

초기화 함수에 부수 효과가 있는 경우 SYNCHRONIZED와 PUBLICATION의 차이가 명확해진다. 예를 들어 다음과 같은 프로퍼티가 있다면,

<코드>

val myValue by lazy {  
 println(“Initializing myValue”)  
 123  
}

</코드>

메시지가 최대 한번만 출력된다. (디폴트인) SYNCHRONIZED 모드의 경우 초기화 함수가 여러 번 호출되지 않게 보장하기 때문이다. 하지만 스레드 안전성 모드를 PUBLICATION으로 바꾸면,

<코드>

val myValue by lazy(LazyThreadSafetyMode.PUBLICATION) {  
 println("Initializing myValue")  
 123  
}

</코드>

프로퍼티 값은 그대로지만 여러 스레드가 myValue 값을 초기화하려고 시도하면서 메시지가 여러 번 출력될 수 있다.

NONE 모드는 가장 빠르며, 초기화 코드가 한 스레드에서만 불린다고 확신할 수 있는 경우 유용하다. 일반적인 경우로는 lazy인 지역 변수를 들 수 있다.

<코드>

fun main() {  
 val x by lazy(LazyThreadSafetyMode.NONE) { 1 + 2 }  
 println(x) // 3  
}

</코드>

초기화 함수가 예외를 던지면 프로퍼티가 초기화되지 않는다는 사실에 유의하라. 이런 경우 이 프로퍼티에 다시 접근하려 시도하면 또 다시 초기화 함수가 호출된다.

kotlin.properties.Delegates의 멤버를 통해 몇 가지 표준 위임을 사용할 수 있다. notNull() 함수는 프로퍼티 초기화를 미루면서 널이 아닌 프로퍼티를 정의할 수 있게 해준다.

<코드>

import kotlin.properties.Delegates.notNull  
  
var text: String by notNull()  
fun readText() {  
 text = readLine()!!  
}  
  
fun main() {  
 readText()  
 println(text)  
}

</코드>

notNull()의 의미는 기본적으로 lateinit 프로퍼티와 같다. 내부에서는 null 값을 사용해 초기화되지 않은 프로퍼티를 표시하지만, notNull() 위임 프로퍼티를 초기화하지 않고 읽으면 java.lang.IllegalStateException이 발생한다. 보통은 lateinit쪽이 더 간결하고 성능도 좋기 때문에 notNull()보다는 lateinit 프로퍼티를 사용하는 편이 낫다. 예외는 원시 타입인 변수의 경우다. 이런 경우에는 lateinit을 적용할 수 없다.

<코드>

import kotlin.properties.Delegates.notNull  
  
var num: Int by notNull() // lateinit을 쓸 수 없음  
  
fun main() {  
 num = 10  
 println(num) // 10  
}

</코드>

observable() 함수를 사용하면 프로퍼티 값이 변경될 때 통지를 받을 수 있다. observable()는 초깃값과 람다를 인자로 받는다. 프로퍼티 값이 바뀔 때마다 람다가 호출된다.

<코드>

import kotlin.properties.Delegates.observable  
  
class Person(name: String, val age: Int) {  
 var name: String by observable(name) { property, old, new ->  
 println("Name changed: $old to $new")  
 }  
}  
  
fun main() {  
 val person = Person("John", 25)  
   
 person.name = "Harry" // Name changed: John to Harry  
 person.name = "Vincent" // Name changed: Harry to Vincent  
 person.name = "Vincent" // Name changed: Vincent to Vincent  
}

</코드>

새 값이 이전 값과 같더라도 통지가 온다는 점에 유의하라. 필요하다면 두 값이 같은지 람다가 직접 검사해야 한다.

vetoable() 함수도 비슷한 위임을 만든다. 이 함수는 초깃값과 Boolean을 반환하는 람다를 인자로 받는다. 프로퍼티 값을 변경하려고 시도할 때마다 값을 변경하기 직전에 이 람다가 호출되고, 람다가 true를 반환하면 실제 값 변경이 일어난다. 람다가 false를 반환하면 값이 바뀌지 않고 그대로 남는다.

<코드>

import kotlin.properties.Delegates.vetoable  
  
var password: String by vetoable("password") { property, old, new ->  
 if (new.length< 8) {  
 println("Password should be at least 8 characters long")  
 false  
 } else {  
 println("Password is Ok")  
 true  
 }  
}  
  
fun main() {  
 password = "pAsSwOrD" // Password is Ok  
 password = "qwerty" // Password should be at least 8 characters long 표시됨  
}

</코드>

observable()과 vetoable()가 제공하는 변경 전과 변경 후 통지를 함께 조합하고 싶다면 ObservableProperty를 상속해서 beforeChange()와 afterChange() 함수를 오버라이드하면 된다.

표준 라이브러리는 맵에 프로퍼티 값을 설정하고 읽어올 수 있는 위임 기능도 제공한다. 맵에 값을 저장하고 읽을 때는 프로퍼티 이름을 키로 사용한다. map 인스턴스를 위임 객체로 사용하면 이런 기능을 활용할 수 있다.

<코드>

class CartItem(data: Map<String, Any?>) {  
 val title: String by data  
 val price: Double by data  
 val quantity: Int by data  
}  
  
fun main() {  
 val item = Cartitem(mapOf(  
 "title" to "Laptop",  
 "price" to 999.9,  
 "quantity" to 1  
 ))  
   
 println(item.title) // Laptop  
 println(item.price) // 999.9  
 println(item.quantity) // 1  
}

</코드>

프로퍼티에 접근할 때는 맵에서 값을 가져와서 (프로퍼티의 타입에 따라) 원하는 타입으로 값을 캐스팅한다. Map 위임은 타입 안전성을 해칠 수도 있기 때문에 조심해서 사용해야 한다. 특히 맵에 저장된 값이 원하는 타입의 값이 아니면 프로퍼티 접근이 캐스트 예외와 함께 끝나버린다.

맵을 위임 객체로 사용할 수 있으므로, 변경 가능한 맵을 통해 가변 프로퍼티를 정의할 수도 있다.

<코드>

class CartItem(data: MutableMap<String, Any?>) {  
 var title: String by data  
 var price: Double by data  
 var quantity: Int by data  
}

</코드>

표준 위임만으로 충분하지 않다면 어떻게 해야 할까? 이런 때는 코틀린의 위임 관련 관습을 활용해 커스텀 위임을 구현할 수 있다. 이에 대해서는 다음 절에서 설명한다.

### 커스텀 위임 만들기

커스텀 위임을 만들려면 특별한 연산자 함수(또는 함수들)를 정의하는 타입이 필요하다. 이 함수들은 프로퍼티 값을 읽고 쓰는 방법을 구현한다. 읽기 함수의 이름은 getValue여야 하고, 다음 두 가지 파라미터를 받는다.

1. receiver: 수신 객체 값이 들어있고, 위임된 프로퍼티의 수신 객체와 같은 타입(또는 상위 타입)이어야 한다
2. property: 프로퍼티 선언을 표현하는 리플렉션이 들어있다. KProperty<\*>이거나 상위 타입이어야 한다

두 파라미터의 이름은 실제로는 중요하지 않고 타입만 중요하다. getValue() 함수의 반환 타입은 반드시 위임 프로퍼티의 타입과 같아야(또는 하위 타입이어야) 한다.

예를 들어 프로퍼티 값과 수신 객체를 연관시켜 기억하는 일종의 캐시 역할을 하는 위임을 만들어보자.

<코드>

import kotlin.reflect.KProperty  
  
class CachedProperty<in R, out T : Any>(val initializer: R.() -> T) {  
 private valcachedValues = HashMap<R, T>()  
   
 operator fun getValue(receiver: R, property: KProperty<\*>): T {  
 return cachedValues.getOrPut(receiver) { receiver.initializer() }  
 }  
}  
  
fun <R, T : Any> cached(initializer: R.() -> T) = CachedProperty(initializer)  
  
class Person(val firstName: String, val familyName: String)  
  
val Person.fullName: String by cached { "$firstName $familyName" }  
  
fun main() {  
 val johnDoe = Person("John", "Doe")  
 val harrySmith = Person("Harry", "Smith")  
   
 // johnDoe 수신 객체에 최초 접근. 값을 계산해 캐시에 담음  
 println(johnDoe.fullName)  
   
 // harrySmith 수신 객체에 최초 접근. 값을 계산해 캐시에 담음  
 println(harrySmith.fullName)  
   
 // johnDoe 수신 객체에 재접근. 캐시에서 값을 읽음  
 println(johnDoe.fullName)  
   
 // harrySmith 수신 객체에 재접근. 캐시에서 값을 읽음  
 println(harrySmith.fullName)  
}

</코드>

fullName은 최상위 확장 프로퍼티이기 때문에 위임도 전역 상태에 속하게 된다. 따라서 프로퍼티의 값은 수신 객체마다 단 한 번만 초기화된다(다중 스레드 문제는 여기서는 무시한다).

읽기 전용 커스텀 위임을 정의하고 싶다면 kotlin.properties 패키지의 ReadOnlyProperty 인터페이스를 사용하면 편리하다. 이 인터페이스는 getValue() 연산자를 추상화한 버전을 정의해준다. 따라서 여러분의 클래스에서 이 연산자를 구현해주면 된다.

<코드>

interface ReadOnlyProperty<in R, out T> {  
 operator fun getValue(thisRef: R, property: KProperty<\*>): T  
}

</코드>

var 프로퍼티에 해당하는 읽고 쓸 수 있는 프로퍼티의 경우, getValue() 외에도 프로퍼티에 값을 저장할 때 호출될 setValue() 함수를 정의해야 한다. 이 함수의 반환 타입은 Unit이어야 하며, 세 가지 파라미터를 받는다.

1. receiver: getValue()와 동일하다.
2. property: getValue()와 동일하다
3. newValue: 프로퍼티에 저장할 새 값이다. 프로퍼티 자체와 같은 타입(또는 상위타입)이어야 한다.

다음 예제는 lateinit 프로퍼티의 final 버전인 위임 클래스를 정의한다. 즉, 이 프로퍼티는 초기화를 단 한 번만 허용한다.

<코드>

import kotlin.reflect.KProperty  
  
class FinalLateinitProperty<in R, T : Any> {  
 private lateinit var value: T  
 operator fun getValue(receiver: R, property: KProperty<\*>): T {  
 return value  
 }  
 operator fun setValue(receiver: R,  
 property: KProperty<\*>,  
 newValue: T) {  
 if (this::value.isInitialized) throw IllegalStateException(  
 "Property ${property.name} is already initialized"  
 )  
 value = newValue  
 }  
}  
  
fun <R, T : Any> finalLateInit() = FinalLateinitProperty<R, T>()  
  
var message: String by finalLateInit()  
  
fun main() {  
 message = "Hello"  
 println(message) // Hello  
 message = "Bye" // Exception: Property message is already initialized  
}

</코드>

코틀린 표준 라이브러리에는 ReadOnlyProperty외에도 값을 변경할 수 있는 위임을 표현하는 인터페이스가 들어있다. 이 인터페이스의 이름은 ReadWriteProperty이다. 여러분이 정의할 위임 클래스에서 이 인터페이스를 구현할 수 있다.

<코드>

public interface ReadWriteProperty<in R, T> {  
 operator fun getValue(thisRef: R, property: KProperty<\*>): T  
 operator fun setValue(thisRef: R, property: KProperty<\*>, value: T)  
}

</코드>

getVersion()/setVersion() 함수를 멤버 함수로 정의할 수도 있지만 확장 함수로 정의해도 된다는 점을 기억하라. 확장 함수를 사용하면 어떤 객체나 일종의 위임 객체로 바꿀 수 있다. 특히 코틀린 라이브러리에서 Map/MutableMap 인스턴스를 사용하는 위임은 확장 함수로 정의되어 있다.

<코드>

inline operator fun <V, V1 : V> Map<in String, V>.getValue(  
 thisRef: Any?,  
 property: KProperty<\*>  
): V1 {...}

</코드>

코틀린 1.1부터는 provideDelegate() 함수를 통해 위임 인스턴스화를 제어할 수 있다. 기본적으로는 프로퍼티 선언 뒤의 by 키워드 다음에 오는 식을 통해 위임 인스턴스를 정의한다. 또는 일종의 위임 팩토리 역할을 하는 중간 인스턴스를 provideDelegate()를 통해 넘길 수도 있다. getValue()와 비슷하게, 이 함수는 프로퍼티 수신 객체와 리플렉션 객체를 파라미터로 받고, 프로퍼티 값 대신 실제 위임 객체를 돌려준다. 위임을 제대로 초기화하기 위해 프로퍼티의 메타데이터가 필요한 경우 이런 기능이 유용하다.

예를 들어 @NoCache라는 애너테이션을 도입하고 싶다고 하자. 이 애너테이션은 프로퍼티 캐시를 방지한다. 여기서는 이전에 정의한 CachedProperty 구현이 (이 애너테이션이 붙은 프로퍼티에) 실제 접근이 일어나는 시점까지 오류 발생을 미루지 않고, 프로퍼티를 초기화하는 단계에서 예외를 빨리 던지기를 바란다. 위임을 제공하는 위임 객체를 도입함으로써, 위임을 생성하기 전에 대상 프로퍼티를 검증해서 이런 기능을 구현할 수 있다.

<코드>

@Target(AnnotationTarget.PROPERTY)  
annotation class NoCache  
  
class CachedPropertyProvider<in R, out T : Any>(  
 val initializer: R.() -> T  
) {  
 operator fun provideDelegate(  
 receiver: R,  
 property: KProperty<\*>  
 ): CachedProperty<R, T> {  
 if (property.annotations.any{ it is NoCache }) {  
 throw IllegalStateException("${property.name} forbids caching")  
 }  
 return CachedProperty(initializer)  
 }  
}  
  
class CachedProperty<in R, out T : Any>(val initializer: R.() -> T) {  
 private valcachedValues = HashMap<R, T>()  
 operator fun getValue(receiver: R, property: KProperty<\*>): T {  
 return cachedValues.getOrPut(receiver) { receiver.initializer() }  
 }  
}  
  
fun <R, T : Any> cached(initializer: R.() -> T) =  
 CachedPropertyProvider(initializer)

</코드>

이제 @NoCache 애너테이션이 붙은 프로퍼티에 대해 캐시를 제공하는 위임을 적용하려고 시도하면 오류가 발생하면서 프로그램이 실패한다.

<코드>

class Person(valfirstName: String, valfamilyName: String)  
  
@NoCachevalPerson.fullName: String by cached {  
 if (this != null) "$firstName $familyName" else ""  
}  
  
fun main() {  
 val johnDoe = Person("John", "Doe")  
 println(johnDoe.fullName) // Exception  
}

</코드>

위임 접근자와 마찬가지로 provideDelegate()도 멤버 함수나 확장 함수로 정의할 수 있다.

### 위임 표현

위임 프로퍼티에 대한 설명을 마무리하면서 런타임에 위임이 어떻게 표현되고 이를 어떻게 접근할 수 있는지에 대해 살펴보겠다.

런타임에 위임은 별도의 필드에 저장된다. 반면 프로퍼티 자체에 대해서는 접근자가 자동으로 생성된다. 이 접근자는 위임에 있는 적절한 메서드를 호출한다. 예를 들어 다음 코드는,

<코드>

class Person(val firstName: String, val familyName: String) {  
 var age: Int by finalLateInit()  
}

</코드>

다음과 같은 효과를 나타내는 코드로 변환된다. 다만 코틀린 코드에서는 age$delegate 같은 위임 필드를 직접 명시적으로 사용할 수 없다.

<코드>

class Person(val firstName: String, val familyName: String) {  
 private val `age$delegate` = finalLateInit<Person, Int>()  
   
 var age: Int  
 get() = `age$delegate`.getValue(this, this::age)  
 set(value) {  
 `age$delegate`.setValue(this, this::age, value)  
 }  
}

</코드>

리플렉션 API를 사용하면 getDelegate() 멤버를 통해 얻을 수 있는 프로퍼티 객체를 통해 위임 값에 접근할 수 있다. 시그니처는 수신 객체의 수에 따라 달라진다. 예제를 하나 보자.

<코드>

import kotlin.reflect.jvm.isAccessible  
   
class Person(val firstName: String, val familyName: String) {  
 val fullName by lazy { "$firstName $familyName" }  
}  
  
fun main() {  
 val person = Person("John", "Doe")  
   
 // KProperty0: 모든 수신 객체가 엮여 있음  
 println(  
 person::fullName  
 .apply { isAccessible = true }  
 .getDelegate()!!::class.qualifiedName  
 ) // kotlin.SynchronizedLazyImpl  
   
 // KProperty1: 수신 객체가 하나 있음  
 println(  
 Person::fullName  
 .apply { isAccessible = true }  
 .getDelegate(person)!!::class.qualifiedName  
 ) // kotlin.SynchronizedLazyImpl  
}

</코드>

위임 인스턴스가 저장된 비공개 필드에 접근하기 위해서는 isAccessible = true를 사용해야만 한다.

확장으로 정의된 프로퍼티는 어떨까? 이런 경우 모든 수신 가능 객체에 대해 위임이 공유되며 getExtensionDelegate() 함수를 사용하면 특정 수신 객체를 지정하지 않아도 위임을 얻을 수 있다.

<코드>

val Person.fullName: String by cached { “$firstName $familyName” }  
  
fun main() {  
 println(  
 Person::fullName  
 .apply { isAccessible = true }  
 .getExtensionDelegate()!!::class.qualifiedName  
 ) // CachedProperty  
}

</코드>

## 고차 함수와 DSL

이 절에서는 타입 안전한 빌드(type-safe builder)를 통해 DLS을 설계하는 방법을 알아본다. 이 작업에 새로운 지식은 필요하지 않다. 코틀린의 고차 함수에 대해 이미 배운 내용을 잘 활용하면 된다.

### 중위 함수를 사용해 플루언트 DSL 만들기

첫 번째 예제를 통해 중위 함수를 사용하는 플루언트(fluent) API를 만드는 방법을 알아본다. 컬렉션 데이터에 대한 질의에 사용할 수 있는 SQL과 비슷한 문법을 사용하는 간단한 DSL을 정의할 것이다.

즉, 다음과 같은 식으로 코드를 작성할 수 있게 허용하고 싶다(C#에 익숙한 독자는 LINQ와 비슷하다는 사실을 눈치챘을 것이다).

<코드>

val nums = listOf(2, 8, 9, 1, 3, 6, 5)  
val query = from(nums) where { it > 3 } select { it\*2 } orderBy { it }  
println(query.items.toList())

</코드>

기본적으로 우리가 만들 질의는 다음 요소로 구성된다.

1. from 절은 대상 컬렉션을 지정한다
2. 다음으로는 선택적으로 where 절이 온다. 이 절은 걸러낼 조건을 기술한다
3. 다음으로는 선택적으로 select 절이 온다. 이 절은 원래 데이터를 출력 값으로 매핑해준다
4. select 절이 있는 경우 원한다면 orderBy 절을 추가할 수도 있다. 이 절은 결과의 순서를 정할 때 사용할 키를 지정한다

이 같은 API를 어떻게 구현할 수 있을까? 먼저 질의의 중간 구조를 표현하는 몇 가지 클래스를 정의하자. 대부분의 중간 구조는 원래의 컬렉션이나 걸러낸 결과 등의 일종의 데이터 집합을 표현하므로, 원소의 시퀀스를 돌려주는 기능을 제공하는 공통 인터페이스를 정의할 것이다.

<코드>

interface ResultSet<out T> {  
 val items: Sequence<T>  
}

</코드>

이제 질의 구성 요소를 표현하는 클래스를 정의할 수 있다.

<코드>

class From<out T>(private val source: Iterable<T>) : ResultSet<T> {  
 override val items: Sequence<T>  
 get() = source.asSequence()  
}  
  
class Where<out T>(  
 private val from: ResultSet<T>,  
 private val condition: (T) -> Boolean  
) :ResultSet<T> {  
 override val items: Sequence<T>  
 get() = from.items.filter(condition)  
}  
  
class Select<out T, out U>(  
 private val from: ResultSet<T>,  
 private val output: (T) -> U  
) :ResultSet<U> {  
 override val items: Sequence<U>  
 get() = from.items.map(output)  
}  
  
class OrderBy<out T, in K : Comparable<K>>(  
 private val select: ResultSet<T>,  
 private valorderKey: (T) -> K  
) :ResultSet<T> {  
 override val items: Sequence<T>  
 get() = select.items.sortedBy(orderKey)  
}

</코드>

기본적인 구성 요소가 준비됐으므로, DSL의 요구사항에 맞게 이들을 엮어줄 중위 연산자 함수를 정의할 수 있다.

<코드>

// from 뒤에 where이 올 수 있음  
infix fun <T> From<T>.where(condition: (T) -> Boolean) =  
 Where(this, condition)  
   
// from이나 where 뒤에 select가 올 수 있음  
infix fun <T, U> From<T>.select(output: (T) -> U) =  
 Select(this, output)  
  
infix fun <T, U> Where<T>.select(output: (T) -> U) =  
 Select(this, output)  
   
// select 뒤에 orderBy가 올 수 있음  
infix fun <T, K : Comparable<K>> Select<\*, T>.orderBy(  
 orderKey: (T) -> K  
) = OrderBy(this, orderKey)

</코드>

마지막 조각은 from() 함수다. 이 함수는 질의를 시작한다.

<코드>

fun <T>from(source: Iterable<T>) = From(source)

</코드>

이제 원래의 예제는 다음과 같다.

<코드>

val nums = listOf(2, 8, 9, 1, 3, 6, 5)  
val query = from(nums) where { it > 3 } select { it\*2 } orderBy { it }  
println(query.items.toList())

</코드>

제대로 컴파일이 되고 다음과 같은 결과를 내놓는다.

<코드>

[10, 12, 16, 18]

</코드>

타입 안전성으로 인해 우리가 의도한 문법 구조에 맞지 않는 질의는 거부된다. 예를 들어 where 절이 하나만 있어야 하기 때문에 다음 코드는 컴파일이 되지 않는다.

<코드>

val query = from(nums) where { it> 3 } where { it < 10 }

</코드>

하지만 여러 where 절을 허용하고 싶다면 infix 함수를 하나 더 추가하면 된다.

<코드>

infix fun <T> Where<T>.where(condition: (T) -> Boolean) =  
 Where(this, condition)

</코드>

이제 구조 내에 다른 구조를 내포하는 좀 더 복잡한 예제를 살펴보자.

### 타입 안전한 빌더 사용하기

DSL을 설계할 때 일반적으로 필요한 기능으로, 도메인 객체가 다른 도메인 객체 안에 내포되는 계층적 구조를 표현하는 방법이 있다. 코틀린에는 빌더 함수와 확장 람다를 결합해 이런 계층적인 구조를 선언적인 방식으로 기술할 수 있게 도와주는 강력한 도구가 들어있다. 간단한 컴포넌트 레이아웃 DSL 예제를 통해 계층적 구조를 구현하는 방법을 알아보자.

목표는 다음과 같은 방식으로 프로그램의 UI를 기술할 수 있게 해주는 API다.

<코드>

fun main() {  
 val form = dialog("Send a message") {  
 borderLayout {  
 south = panel {  
 +button("Send")  
 +button("Cancel")  
 }  
 center = panel {  
 verticalBoxLayout {  
 +filler(0, 10)  
 +panel {  
 horizontalBoxLayout {  
 +filler(5, 0)  
 +label("Message: ")  
 +filler(10, 0)  
 +textArea("")  
 +filler(5, 0)  
 }  
 }  
 +filler(0, 10)  
 }  
 }  
 }  
 }  
 form.size = Dimension(300, 200)  
 form.isVisible = true  
}

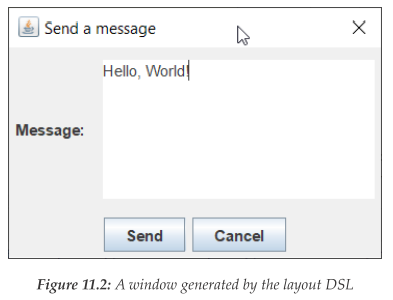
</코드>

기본적으로 다음을 제공하는 DSL을 만들고 싶다.

* UI 컴포넌트를 구성하는 계층적인 구조를 기술한다
* BorderLayout, BoxLayout 등의 표준 레이아웃 관리자를 지원한다
* 버튼, 텍스트 필드, 패널(panel), 윈도우(window) 등의 공통 컴포넌트를 생성하고 초기화할 수 있는 도우미 함수를 제공한다

그림 11-2는 방금 본 코드로부터 생성한 윈도우다.

그림 11-2: 레이아웃 DSL로 생성한 윈도우



이런 종류의 언어를 어떻게 구현할 수 있을까? 먼저 UI를 기술할 때 어떤 종류의 객체들이 필요한지 생각해보자.

* 버튼이나 텍스트 필드 등의 간단한 컴포넌트: 간단한 컴포넌트 내부에는 내포된 구조가 없다
* 패널이나 윈도우 등의 컨테이너: 컨테이너에는 레이아웃을 추가하고 내부에 + 연산을 통해 컴포넌트를 내포시켜야 한다
* 컨테이너의 자식 컴포넌트의 위치를 지정하기 위한 레이아웃: 레이아웃은 구체적인 배치를 결정한다. 예를 들어 보더(border) 레이아웃은 자식을 미리 정해진 영역(NORTH, SOUTH 등)에 연동시키지만, 박스 레이아웃은 컴포넌트를 추가한 순서대로 행이나 열로 배치하게 해준다.

button() 함수는 가장 단순한 부분이다. 아무런 추가 처리가 필요하지 않고, 단지 컴포넌트 생성자를 감싸주기만 한다.

<코드>

fun label(text: String) = JLabel(text)  
fun button(text: String) = JButton(text)  
fun textArea(text: String) = JTextArea(text)

</코드>

더 흥미로운 경우는 panel() 함수다. 이 함수는 내포된 컴포넌트 정의를 포함하는 람다를 인자로 받는다. 컨테이너의 상태를 유지하기 위해 ContainerBuilder 클래스를 도입한다. 이 ContainerBuilder 클래스를 통해 내포시킬 컴포넌트를 추가하고 레이아웃을 정의할 수 있다.

<코드>

class ContainerBuilder(private val container: Container) {  
 operator fun Component.unaryPlus() = apply { container.add(this) }  
   
 fun borderLayout(body: BorderLayoutBuilder.() -> Unit) {  
 BorderLayoutBuilder(container).body()  
 }  
   
 fun horizontalBoxLayout(body: BoxLayoutBuilder.() -> Unit) {  
 BoxLayoutBuilder(container, BoxLayout.LINE\_AXIS).body()  
 }  
   
 fun verticalBoxLayout(body: BoxLayoutBuilder.() -> Unit) {  
 BoxLayoutBuilder(container, BoxLayout.PAGE\_AXIS).body()  
 }  
}

</코드>  
이제 이 빌더를 사용해 panel()과 dialog() 함수를 정의할 수 있다.

<코드>

fun panel(body: ContainerBuilder.() -> Unit) = JPanel().apply {  
ContainerBuilder(this).body()  
}  
  
fun dialog(  
 title: String,  
 body: ContainerBuilder.() -> Unit  
): JDialog = JDialog().apply {  
 this.title = title  
 pack()  
 defaultCloseOperation = JDialog.DISPOSE\_ON\_CLOSE  
 ContainerBuilder(contentPane).body()  
}

</코드>

코드를 보면 두 함수가 람다를 받고, 이 람다는 ContainerBuilder의 확장함수 역할을 한다는 점을 알 수 있다. 이렇게 하면 람다 내부에서 this 수신 객체로 ContainerBuilder가 암시적으로 지정되기 때문에 직접 ContainerBuilder의 멤버를 호출할 수 있다. 예를 들어 다음과 같은 코드는,

<코드>

panel {  
 horizontalBoxLayout {  
 +filler(5, 0)  
 ...  
 }  
}

</코드>

실제로는 다음과 같은 뜻이다.

<코드>

panel {  
 this.horizontalBoxLayout {  
 // BoxLayoutBuilder 타입의 디스패치 수신 객체를 암시적으로 가정한다  
 filler(5, 0).unaryPlus()  
 ...  
 }  
}

</코드>

레이아웃은 어떻게 처리할 수 있을까? 비슷한 방법으로 레이아웃 빌더를 만들 수 있다. 이때 여러 가지 레이아웃 사이의 API 차이를 감안해야 한다. 예를 들면 다음과 같다.

<코드>

class BoxLayoutBuilder(private val container: Container, direction: Int) {  
 init {  
 container.layout = BoxLayout(container, direction)  
 }  
   
 operator fun Component.unaryPlus() = apply { container.add(this) }  
   
 fun filler(width: Int, height: Int) =  
 Box.createRigidArea(Dimension(width, height))  
}

</코드>

패널 같은 컨테이너의 경우 자식을 순서대로 추가하고 싶기 때문에, 이를 위해 BoxLayoutBuilder에 unaryPlus()를 추가했다. BorderLayoutBuilder의 경우, 추가된 자식 컴포넌트를 유지했다가 컨테이너에 연결해주는 north, south, west 등의 프로퍼티를 설정해야 한다. 이 로직을 observable 위임을 사용해 별도의 로직으로 처리한다.

<코드>

fun constrained(  
 container: Container,  
 constraint: Any?  
) = observable<Component?>(null) { \_, \_, value ->  
 container.add(value, constraint)  
}  
  
class BorderLayoutBuilder(container: Container) {  
 init {  
 container.layout = BorderLayout()  
 }  
   
 var north by constrained(container, BorderLayout.NORTH)  
 var south by constrained(container, BorderLayout.SOUTH)  
 var west by constrained(container, BorderLayout.WEST)  
 var east by constrained(container, BorderLayout.EAST)  
 var center by constrained(container, BorderLayout.CENTER)  
}

</코드>

다양한 코틀린 DSL이 비슷한 방식으로 구현된다. 이 책의 뒷부분에서는 테스트 명세, 안드로이드 UI 기술, 웹 애플리케이션 처리 규칙, 타입 안전한 HTML 생성 등 여러 가지 공통 분야에 쓰이는 DSL에 대해 자세히 살펴볼 것이다. 지금은 빌더 함수의 영역을 어떻게 제어할 것인가에 대해 더 논의해보자.

### @DslMarker

앞 절에서 논의했던 것 같은 계층적인 DSL을 사용하는 경우 외부 블록의 멤버가 내부 영역에 누출된다는 사실을 눈치챘는가? 예를 들어 레이아웃 DSL에 다음과 같은 코드를 써도 문제가 없다.

<코드>

val myPanel = panel {  
 borderLayout {  
 borderLayout {  
 }  
 }  
}

</코드>

물론 이런 구조는 우리 의도를 벗어난다. 레이아웃 안에 다른 레이아웃을 포함시키고 싶지 않기 때문이다. 하지만 여기 보여준 코드는 여전히 올바른 코드이고, 이때 두 borderLayout() 함수 호출의 수신 객체는 가장 밖의 ContainerBuilder 객체로 똑같다. 문제는 각 수신 객체를 그 자신을 정의한 영역뿐 아니라 (확장 함수 내부에) 내포된 모든 영역에서 접근 가능할 수 있다는 데 있다. 수신 객체를 명시하면 코드가 다음과 같은 모습이 될 것이다.

<코드>

val myPanel = panel {  
 this@panel.borderLayout {  
 this@panel.borderLayout {  
 }  
 }  
}

</코드>

이 코드를 보면 두 수신 객체가 같다는 점을 명확히 알 수 있다.

따라서 암시적 수신 객체가 내포된 영역에 누출되는 것이 타입 안전성을 해치지는 못하지만, 원치 않는 코드를 작성할 수 있게 되므로 오류가 발생하기 쉽다. 이 문제는 특히 우리가 작성하는 코드가 내포된 확장 람다가 많이 있는 DSL인 경우 더 문제가 된다. 따라서 코틀린 1.1에서는 @DslMarker를 도입해서 DSL 설계자들이 암시적 수신 객체의 영역을 제한할 수 있게 허용했다.

@DslMarker는 여러분이 애너테이션을 작성할 때 쓸 수 있는 메타 에너테이션이다. 이 말은 @DslMarker가 특정 DSL을 구분하는 마커(marker) 역할을 하는 애너테이션을 정의할 때 쓰인다는 뜻이다. 이 목적으로 @LayoutDsl을 만들어보자.

<코드>

@DslMarker  
annotation class LayoutDsl

</코드>

이제 DSL 블록 안에서 수신 객체로 쓰일 수 있는 클래스에 @LayoutDsl 애너테이션을 붙일 수 있다. 우리가 만든 DSL에서는 ContainerBuilder, BorderLayoutBuilder, BoxLayoutBuilder가 수신 객체 역할을 할 수 있는 클래스다.

<코드>

@LayoutDsl  
class ContainerBuilder(private val container: Container) {...}  
  
@LayoutDsl  
class BorderLayoutBuilder(container: Container) {...}  
  
@LayoutDsl  
class BoxLayoutBuilder(private val container: Container, direction: Int)   
{...}

</코드>

대상 클래스에 상위 타입이 있다면 이 상위 타입에 마커를 붙일 수도 있다. DSL 마커는 자동으로 모든 하위 타입에도 영향을 끼친다.

이렇게 하고 나면 컴파일러가 어떤 클래스가 같은 DSL에 속하는지 구분할 수 있다. 따라서 컴파일러는 내포된 영역 내부에서 (더 밖에 있는) 같은 DSL에 상응하는 수신 객체를 사용하지 못하게 금지한다. 예를 들어 앞에서 보여준 이중 레이아웃 예제는 이제 컴파일 오류를 발생시킨다.

<코드>

val myPanel = panel {  
 borderLayout {  
 borderLayout{ // Error: DSL 영역 위반  
 }  
 }  
}

</코드>

여기서 @DslMarker는 암시적인 수신 객체의 누출만 막을 수 있다는 점에 유의하라. 필요한 경우에는 명시적인 this를 써서 외부 수신 객체를 가져와 사용할 수 있다.

<코드>

val myPanel = panel {  
 borderLayout {  
 this@panel.borderLayout{ // 올바른 코드  
 }  
 }  
}

</코드>

## 결론

이 장은 코틀린 프로그램에 내포 시킬 수 있는 도메인 특화 언어를 설계할 때 도움이 되는 고급 코틀린 언어 기능을 살펴봤다. DSL을 사용하면 코틀린 컴파일러가 제공하는 타입 안전성과 DSL이 주는 사용 편의를 함께 누릴 수 있다. 개발자가 오버로딩한 연산자를 정의할 수 있게 허용하는 언어 관습에 대해 살펴보고, 위임 프로퍼티의 표준 구현을 알아봤으며, 여러분이 직접 위임 프로퍼티를 작성하는 방법에 대해 설명했다. 마지막으로 함수형 프로그래밍과 타입 안전한 빌더가 계층적인 DSL을 설계할 때 얼마나 도움이 되는지 살펴봤다.

다음 장에서는 자바/코틀린 상호 운용성에 대해 다룬다. 코틀린 선언을 자바 코드에서 사용하는 방법과 자바 선언을 코틀린 코드에서 사용하는 방법에 대해 살펴보고, 코틀린 기반 API를 자바 클라이언트가 사용할 수 있게 미세 조정하는 방법과 코틀린 도구로 자바 코드를 코틀린 코드로 자동 변환해주는 방법을 알아보자.

## 문제

1. 연산자 오버로딩이란 무엇이고, 코틀린에서는 어떤 관습을 사용해 연산자 오버로딩을 사용하는가?
2. 표준 위임 구현에 대해 설명하라.
3. 프로퍼티 위임의 관습은 무엇인가? 커스텀 위임 구현의 예를 한 가지 들어보라.
4. 런타임에 (리플렉션을 써서) 위임 값에 접근하는 방법을 설명하라.
5. 고차 함수를 사용해 DSL을 설계하는 방법을 설명하라.
6. @DslMarker의 의미에 대해 설명하라.