# 6장 특별한 클래스 사용하기

이번 장에서는 몇가지 일반적인 프로그래밍 패턴의 구현을 편하게 해주는 특별한 종류의 클래스에 대해 배운다. 이름을 말해보면, 미리 정해진 인스턴스만 허용하는 이넘(enum) 사용법에 대해 설명하고, 데이터를 data 클래스로 표현하는 방법을 배우며, 거의 런타임 부가비용이 없는 실험적인 경량 래퍼 클래스에 대해 배운다.

## 구조

* 이넘 클래스
* 데이터 클래스
* 인라인 클래스

## 목표

이넘이나 데이터 클래스 같은 다양한 클래스 유형을 사용해 일반적인 프로그래밍 과업을 수행하는 방법을 배운다. 인라인 클래스를 이해하고 부호가 없는 정수 타입을 예로 들어서 사용법을 익힌다.

## 이넘 클래스

이넘(enumeration을 줄인 enum의 발음을 음차) 클래스는 미리 정의된 상수들로 이뤄진 제한된 집합을 표현하는 특별한 클래스이다. 가장 단순한 형태는 enum class의 본문에 상수를 모아둔 형태다.

enum class WeekDay {  
 MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY  
}  
  
fun WeekDay.isWorkDay() =  
 this == WeekDay.SATURDAY || this == WeekDay.SUNDAY  
  
fun main() {  
 println(WeekDay.MONDAY.isWorkDay()) // false  
 println(WeekDay.SATURDAY.isWorkDay()) // true  
}

정수, 문자열 등과 비교할 때, 이넘을 사용하면 어떤 값이 가능한 범위 안에 들어가 있는지를 일일히 검사할 필요가 없기 때문에, 정해진 상수들로 이뤄진 집합을 타입 안전(type-safe)하게 다룰 수 있다. 컴파일러는 이넘 타입의 변수가 이넘 클래스에 정의된 값 중 하나를 사용하는지 검사해준다.

**자바와 코틀린의 차이**: 코틀린 이넘은 enum class라는 두 키워드의 조합으로 정의되지만 자바는 enum 만으로 정의된다. enum 키워드 자체는 소프트 키워드라서 다른 문맥에서는 식별자로 사용할 수 있다.

컴파일 시점 상수이기 때문에 이넘 값을 쓸때는 보통 대문자를 사용한다.

이넘은 특히 특정 타입에 속하는 미리 정해진 전역 상수를 정의한다는 점에서 객체 정의와도 약간 비슷하다. 객체와 마찬가지로 전역 상수로 사용할 수 있는 방법이 없는 위치에서 이넘을 정의할 수 없다. 예를 들어 이넘을 내부 클래스나 함수 본문에서 정의할 수 없다.

fun main() {  
 enum class Direction { NORTH, SOUTH, WEST, EAST } // Error  
}

### 빠뜨린 부분이 없는 when 식

다른 타입의 값과 마찬가지로 when을 사용하면 이넘 변수를 각각의 값과 비교할 수 있다. 하지만 이넘을 사용하면 한가지 장점이 있다. when 식에서 모든 이넘 상수를 다룬 경우에는 else 가지를 생략해도 된다.

enum class Direction {  
 NORTH, SOUTH, WEST, EAST  
}  
  
fun rotateClockWise(direction: Direction) = when (direction) {  
 Direction.NORTH -> Direction.EAST  
 Direction.EAST -> Direction.SOUTH  
 Direction.SOUTH -> Direction.WEST  
 Direction.WEST -> Direction.NORTH  
}

빠뜨린 부분이 없는 형태의 when 식을 사용하면 새 이넘 값을 추가하는 경우에 깨질 수 있는 코드가 생기는 일을 방지할 수 있다. 빠뜨린 부분이 없는 when을 사용하는 대신, else 가지를 추가했다고 가정해보자.

fun rotateClockWise(direction: Direction) = when (direction) {  
 Direction.NORTH -> Direction.EAST  
 Direction.EAST -> Direction.SOUTH  
 Direction.SOUTH -> Direction.WEST  
 Direction.WEST -> Direction.NORTH  
 else ->  
 throw IllegalArgumentException("Invalid direction: $direction")  
}

새로운 Direction 이넘 값을 추가할 때까지는 이 코드가 잘 작동한다. 하지만 새 값을 추가하면,

enum class Direction {  
 NORTH, SOUTH, WEST, EAST,  
 NORTH\_EAST, NORTH\_WEST, SOUTH\_EAST, SOUTH\_WEST  
}

rotateClockWise(Direction.NORTH\_EAST) 같은 호출이 예외를 발생시킨다. 하지만 else를 사용하지 않는 when 식을 사용하면 컴파일 시점에 컴파일러가 rotateClockWise() 함수의 when에 빠뜨린 부분이 있다고 경고하기 때문에 쉽게 이런 오류를 방지할 수 있다.

**자바와 코틀린의 차이**: case 절에서 이넘 값을 간단한 이름으로 사용해야만 하는 자바의 switch 문과 달리 코틀린의 when에서는 이넘 상수를 임포트하지 않고 긴 이름으로 사용해도 된다. 앞에서 본 rotateClockWise() 함수를 이와 비슷한 역할을 하는 자바 메서드와 비교해보자.

public Direction rotateClockWise(Direction d) {  
 switch (d) {  
 case NORTH: return Direction.EAST;  
 case EAST: return Direction.SOUTH;  
 case SOUTH: return Direction.WEST;  
 case WEST: return Direction.NORTH;  
 }  
 throw new IllegalArgumentException("Unknown value: " + d);  
}

파일 앞에서 이넘 상수를 임포트하면 한정시키지 않은 짧은 이름으로 이넘 상수를 사용할 수 있다.

import Direction.\*  
  
enum class Direction {  
 NORTH, SOUTH, WEST, EAST  
}  
  
fun rotateClockWise(direction: Direction) = when (direction) {  
 NORTH -> EAST  
 EAST -> SOUTH  
 SOUTH -> WEST  
 WEST -> NORTH  
}

내부적으로 빠진 부분이 없는 when 식에는 NoWhenBranchMatchedException 타입의 예외를 던지는 else 가지가 암시적으로 추가된다.

IDE 팁

인텔리J 코틀린 플러그인은 불필요한 else 가지를 감지해서, 빠진 부분이 없는 when식으로 만들 수 있는 경우 해당 else가지를 제거하라고 제안한다(그림 6.1).

그림 6.1: 빠진 부분이 없는 when 식에 있는 불필요한 else 가지

### 커스텀 멤버가 있는 이넘 정의하기

다른 클래스와 마찬가지로 이넘 클래스도 멤버를 포함할 수 있다. 그 외에도 원한다면 앞의 예제에서 본 것처럼 이넘에도 확장 함수나 프로퍼티를 붙일 수 있다.

이넘 클래스도 일반 클래스에 허용되는 정의를 포함할 수 있다. 이런 정의에는 함수, 프로퍼티, 주 생성자와 부 생성자, 초기화 블럭, 내부/내포 클래스, 객체(동반 내포 객체와 일반 내포 객체) 등이 포함된다. 이넘 클래스에 정의된 이런 요소들은 반드시 이넘 상수 목록 뒤에 와야 한다. 이런 경우 상수 목록과 다른 부분을 구분하기 위해 상수 목록을 세미콜론으로 끝내야 한다(코틀린에서 세미콜론을 생략할 수 없는 몇 안되는 드문 경우 중 하나가 이 경우다). 모든 이넘 클래스 상수에서 이넘 클래스의 본문에 정의된 멤버를 사용할 수 있다.

enum class WeekDay {  
 MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;  
   
 val lowerCaseName get() = name.toLowerCase()  
 fun isWorkDay() = this == SATURDAY || this == SUNDAY  
}  
  
fun main() {  
 println(WeekDay.MONDAY.isWorkDay()) // false  
 println(WeekDay.WEDNESDAY.lowerCaseName) // wednesday  
}

이넘 클래스에 생성자가 있으면 각 이넘 상수의 정의 뒤에도 적절한 생성자 호출을 추가해야 한다.

enum class RainbowColor(val isCold: Boolean) {  
 RED(false), ORANGE(false), YELLOW(false),  
 GREEN(true), BLUE(true), INDIGO(true), VIOLET(true);  
   
 val isWarm get() = !isCold  
}  
  
fun main() {  
 println(RainbowColor.BLUE.isCold) // true  
 println(RainbowColor.RED.isWarm) // true  
}

이넘 상수에도 자신만의 정의가 들어있는 본문이 포함될 수 있다. 하지만 이런 상수에 의해 생겨나는 익명 타입은 (*4장. 클래스와 객체 다루기*에서 이미 익명 타입에 대해 설명했다) 코드 밖으로 노출되지 않는다. 따라서 이넘 상수의 본문에 정의된 멤버를 해당 본문이 아닌 부분에서 접근할 수는 없다. 다음 코드는 이런 아이디어를 보여준다.

enum class WeekDay {  
 MONDAY { fun startWork() = println(“Work week started”) },  
 TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY  
}  
  
fun main() = WeekDay.MONDAY.startWork() // Error

일반적으로 이넘 클래스 자체나 상위 타입에 들어있는 가상 메서드를 구현하는 경우에 이런 식의 상수별 멤버 정의가 유용하다. 이에 대한 예제는 *8장. 클래스 계층 이해하기*으로 미뤄두자.

현재로써는 이넘 상수의 본문에 정의된 내포된 클래스들은 반드시 내부 클래스여야만 한다는 점을 기억하라.

### 이넘 클래스의 공통 멤버 사용하기

코틀린의 모든 이넘 클래스는 암시적으로 kotlin.Enum 클래스의 하위 타입이다. kotlin.Enum 클래스는 모든 이넘 클래스가 사용할 수 있는 몇가지 공통 함수와 프로퍼티를 제공한다. 몇가지 API 차이를 제외하면 이런 공통 멤버들은 자바 java.lang.Enum에 있는 상응하는 멤버들과 꽤 비슷하다. 실제, JVM에서 코틀린 이넘은 자바 Enum을 통해 표현된다.

모든 이넘 값에는 ordinal와 name 이라는 한쌍의 프로퍼티가 들어있다. ordinal은 이넘 클래스 안에 정의된 이넘 값의 순서에 따른 인덱스이고 name은 이넘 값의 이름이다.

enum class Direction {  
 NORTH, SOUTH, WEST, EAST;  
}  
  
fun main() {  
 println(Direction.WEST.name) // WEST  
 println(Direction.WEST.ordinal) // 2  
}

특정 이넘 클래스의 값을 이넘 본문에 있는 정의의 위치를 사용해 서로 비교할 수 있다. 자바와 비슷하게 이넘 값의 동등성은 각각의 식별자에 따라 정해진다.

fun main() {  
 println(Direction.WEST == Direction.NORTH) // false  
 println(Direction.WEST != Direction.EAST) // true  
 println(Direction.EAST < Direction.NORTH) // false  
 println(Direction.SOUTH >= Direction.NORTH) // true  
}

이넘 값에 대한 비교는 ordinal 프로퍼티가 돌려주는 인덱스 값에 의해 정의된다.

**자바와 코틀린의 차이**: 자바와 코틀린 이넘이 모두 Comparable 인터페이스를 암시적으로 구현하기는 하지만, 자바에서는 이넘 값에 대해 <나 > 등의 연산을 적용할 수 없다.

각 이넘 클래스는 마치 동반 객체의 멤버처럼 호출할 수 있는 암시적인 메서드들을 제공한다. valueOf() 메서드는 이넘 값의 이름을 문자열로 넘기면 그에 해당하는 이넘 값을 돌려주거나 이름이 잘못된 경우 예외를 던진다.

fun main() {  
 println(Direction.valueOf("NORTH")) // NORTH  
 println(Direction.valueOf("NORTH\_EAST")) // Exception: Invalid name  
}

values() 메서드는 정의된 순서대로 모든 이넘 값이 들어있는 배열을 돌려준다. 이 메서드를 호출할 때마다 배열이 새로 생긴다는 점에 유의하라. 따라서 이 메서드에서 얻은 배열의 내용을 바꾸더라도 (또 values() 메서드를 호출해 얻은) 나머지 배열에는 영향이 없다.

enum class WeekDay {  
 MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY  
}  
  
private val weekDays = WeekDay.values()  
  
val WeekDay.nextDay get() = weekDays[(ordinal + 1) % weekDays.size]

코틀린 1.1부터는 values()나 valueOf() 대신, 제네릭(generic) 최상위 메서드인 enumValues()와 enumValueOf()를 사용할 수도 있다.

fun main() {  
 val weekDays = enumValues<WeekDay>()  
   
 println(weekDays[2]) // WEDNESDAY  
 println(enumValueOf<WeekDay>("THURSDAY")) // THURSDAY  
}

## 데이터 클래스

[지역변수가 아니어도 구조분해가 됩니다. 지역변수에서 지역을 뺍니다.]

코틀린은 데이터를 저장하기 위한 목적으로 주로 쓰이는 클래스를 선언할 수 있는 유용한 기능을 제공한다. 이런 기능을 데이터 클래스(data class)라고 부르며, 이 기능을 사용하면 컴파일러가 동등성 비교나 String으로 변환하는 등의 기본 연산에 대한 구현을 자동으로 생성해준다. 그리고 구조분해 선언(destructuring declaration)을 활용할 수도 있다. 구조분해를 사용하면 클래스의 프로퍼티를 간단한 한가지 언어 구성요소를 사용해 여러 변수에 나눠 넣을 수 있다. 이번 절에서는 데이터 클래스의 유용성에 대해 살펴본다.

### 데이터 클래스와 데이터 클래스에 대한 연산

다음 클래스를 생각해 보자.

class Person(val firstName: String,  
 val familyName: String,  
 val age: Int)

이 클래스의 두 인스턴스가 동등한지 비교하면 어떻게 될까? 자바와 마찬가지로 참조 타입의 값은 기본적으로 참조가 가리키는 객체의 정체성(identity)이 같으면 같은데, 이 말은 두 객체가 같은 객체면(즉 메모리에서 같은 위치에 있는 객체면) 같다는 말이다. 이때 인스턴스의 필드 값이 같은지는 고려 대상이 아니다.

fun main() {  
 val person1 = Person("John", "Doe", 25)  
 val person2 = Person("John", "Doe", 25)  
 val person3 = person1  
  
 println(person1 == person2) // false, 서로 다른 정체성  
 println(person1 == person3) // true, 같은 정체성  
}

클래스에 커스텀 동등성 비교가 필요하면 보통 equals() 메서드와 이와 연관된 hashCode() 메서드를 구현한다(*7장. 컬렉션과 I/O 자세히 알아보기*와 *8장. 클래스 계층 이해하기*에서 이에 대해 더 자세히 다룬다). 이 둘을 정의하면 HashMap등의 컬렉션에서도 클래스 인스턴스를 키로 사용할 수 있게 된다[[1]](#footnote-1). 데이터 클래스라고 불리는 특정 유형의 클래스에 대해서는 코틀린이 클래스의 프로퍼티 목록을 기반으로 이런 메서드를 자동으로 생성해준다. 앞의 예제를 조금 바꿔보자.

data class Person(val firstName: String,  
 val familyName: String,  
 val age: Int)  
  
fun main() {  
 val person1 = Person("John", "Doe", 25)  
 val person2 = Person("John", "Doe", 25)  
 val person3 = person1  
  
 println(person1 == person2) // true  
 println(person1 == person3) // true  
}

이제는 컴파일러가 주 생성자에 정의된 프로퍼티의 값을 서로 비교하는 동등성 비교 연산을 자동으로 생성해 주기 때문에, 두 비교가 모두 true를 반환한다. 마찬가지로 이 클래스의 hashCode() 메서드도 주 생성자에 정의된 프로퍼티의 값에 따라 결정된다.

프로퍼티 값의 비교도 equals() 메서드를 통한다는 점에 유의하라. 따라서 깊은 동등성(deep equality) 비교가 이뤄지느냐 여부는 프로퍼티의 타입으로 어떤 타입이 들어있느냐에 따라 달라진다. 다음 예를 보자.

data class Person(val firstName: String,  
 val familyName: String,  
 val age: Int)  
   
data class Mailbox(val address: String, val person: Person)  
  
fun main() {  
 val box1 = Mailbox("Unknown", Person("John", "Doe", 25))  
 val box2 = Mailbox("Unknown", Person("John", "Doe", 25))  
 println(box1 == box2) // true  
}

String, Person, MainBox는 모두 내용을 바탕으로 동등성을 구현하기 때문에 MailBox 인스턴스 비교는 address 프로퍼티(문자열) 비교와 person에 들어있는 Person 클래스의 동등성 비교에 따라 결정된다. 하지만 Person 클래스 앞에 있는 data 변경자를 없애면 Person 프로퍼티 비교가 객체의 정체성에 따라 결정되기 때문에 결과가 바뀐다.

class Person(val firstName: String,  
 val familyName: String,  
 val age: Int)  
   
data class Mailbox(val address: String, val person: Person)  
  
fun main() {  
 val box1 = Mailbox("Unknown", Person("John", "Doe", 25))  
 val box2 = Mailbox("Unknown", Person("John", "Doe", 25))  
 // false: 두 Person 인스턴스의 정체성이 다름  
 println(box1 == box2)  
}

비슷하게, hashCode() 메서드 구현도 주 생성자에 선언된 모든 프로퍼티의 해시코드에 의존해 계산한 해시 코드를 반환한다.

데이터 클래스는 equals()나 hashCode() 외에 toString() 메서드 구현도 생성해준다. 이 메서드는 클래스 인스턴스를 문자열로 변환해준다.

fun main() {  
 val person = Person("John", "Doe", 25)  
 println(person) // Person(firstName=John, familyName=Doe, age=25)  
}

주 생성자의 파라미터에서 선언한 프로퍼티만 equals()/hashCode()/toString() 메서드 구현에 쓰인다는 점에 유의하라. 다른 프로퍼티들은 이런 함수들의 생성에 영향을 끼치지 못한다.

data class Person(val firstName: String, val familyName: String) {  
 var age = 0  
}  
  
fun main() {  
 val person1 = Person("John", "Doe").apply { age = 25 }  
 val person2 = Person("John", "Doe").apply { age = 26 }  
 println(person1 == person2) // true  
}

모든 데이터 클래스는 암시적으로 copy() 함수를 제공한다. 이 함수를 사용하면 현재 인스턴스를 복사하면서 몇몇 프로퍼티를 변경할 수 있다. copy() 함수의 시그니처는 데이터 클래스의 주 생성자 시그니처와 같다. 하지만 각 프로퍼티마다 원본 객체의 프로퍼티 값이 디폴트 값으로 지정된다. 코드 가독성을 위해 copy() 함수를 호출할 때는 보통 이름 붙은 인자 구문을 사용한다.

fun Person.show() = println("$firstName $familyName: $age")  
  
fun main() {  
 val person = Person("John", "Doe", 25)  
   
 person.show() // John Doe: 25  
 person.copy().show() // John Doe: 25  
 person.copy(familyName = "Smith").show() // John Smith: 25  
 person.copy(age = 30, firstName = "Jane").show() // Jane Doe: 30  
}

쉽게 인스턴스를 복사할 수 있는 기능은 불변 데이터 구조 사용을 촉진한다. var 프로퍼티를 쓸 수도 있지만 보통은 데이터 클래스를 불변 클래스로 설계하는 편이 더 타당하다. 불변 데이터를 사용하면 코드에 대한 추론이 쉬워지고 실수를 덜 저지르게 된다. 특히 다중 스레드 환경에서 불변성이 더 도움이 된다. 그 외에도 객체를 맵의 키로 제대로 사용하기 위해서는 불변성이 필수 조건이다. 맵의 키 등에서 불변성을 위반하면 나중에 *7장. 컬렉션과 I/O 자세히 알아보기*에서 볼 수 있는 것 같은 예기치 않은 동작을 보게 될 수도 있다.

코틀린 표준 라이브러리에는 두가지 범용 데이터 클래스가 들어있다. 이들은 두 값(한 쌍, pair)이나 세 값(트리플렛, triplet)을 저장할 수 있는 데이터 클래스이다.

fun main() {  
 val pair = Pair(1, "two")  
   
 println(pair.first + 1) // 2  
 println("${pair.second}!") // two!  
   
 val triple = Triple("one", 2, false)  
   
 println("${triple.first}!") // one!  
 println(triple.second – 1) // 1  
 println(!triple.third) // true  
}

중위 연산자를 사용해 Pair 객체를 만들 수도 있다.

val pair = 1 to "two"  
  
println(pair.first + 1) // 2  
println("${pair.second}!") // two!

대부분의 경우 이같은 범용 데이터 클래스 보다는 커스텀 클래스를 사용하는 편이 더 낫다. 커스텀 클래스를 정의하면 클래스의 프로퍼티에 의미가 있는 이름을 부여할 수 있어서 코드 가독성을 향상시킬 수 있다.

지금까지 살펴본 자동으로 생성한 함수들 외에도, 데이터 클래스를 사용하면 인스턴스를 구성하는 프로퍼티들을 한 정의문 안에서 여러 변수에 나눠 추출할 수 있다. 다음 절에서는 구조 분해 선언을 사용해 이렇게 프로퍼티를 변수에 나눠 담는 방법을 알아본다.

### 구조분해 선언

다음 예제를 살펴보자.

import kotlin.random.Random  
  
data class Person(val firstName: String,  
 val familyName: String,  
 val age: Int)  
   
fun newPerson() = Person(readLine()!!,  
 readLine()!!,  
 Random.nextInt(100))  
   
fun main() {  
 val person = newPerson()  
 val firstName = person.firstName  
 val familyName = person.familyName  
 val age = person.age  
   
 if (age < 18) {  
 println("$firstName $familyName is under-age")  
 }  
}

이 코드는 Person 인스턴스의 각 프로퍼티를 추출한 다음 계산에 활용한다. 하지만 Person이 데이터 클래스이기 때문에 이를 각각의 프로퍼티에 대응하는 지역 변수를 정의하는 좀 더 간결한 구문으로 대신할 수 있다.

val (firstName, familyName, age) = person

이런 선언을 구조분해 선언이라고 하며, 변수 이름을 하나만 사용하는 대신 괄호로 감싼 식별자 목록으로 여러 변수를 한꺼번에 정의할 수 있게 해주는 일반화된 지역 변수 선언 구문이다. 각 이름은 별도의 변수 선언과 같고, = 다음에 적은 데이터 클래스 인스턴스의 프로퍼티에 해당한다.

여기서 각 변수에 어떤 프로퍼티가 매핑되는지는 데이터 클래스의 생성자에 있는 각 프로퍼티의 위치에 따라 결정되며, 선언하는 변수의 이름에 의해 결정되지 않는다는 점에 유의하라. 따라서 다음 코드는

val (firstName, familyName, age) = Person("John", "Doe", 25)  
  
println("$firstName $familyName: $age")

예상대로 John Doe: 25라는 문자열을 만들어내지만, 다음 코드는

val (familyName, firstName, age) = Person("John", "Doe", 25)  
  
println("$firstName $familyName: $age")

Doe John: 25를 만들어낸다.

[편집자께: 구조분해에서 변수 순서를 바꾸든 변수 이름을 바꾸든 결과는 똑같습니다. 그냥 변수와 프로퍼티 이름이 일치하게 바꾸라고 변경합니다.]

IDE 팁

방금 본 예제의 경우 구조분해에 사용한 변수 이름은 데이터 클래스 선언의 프로퍼티 이름과 일치하지만 순서만 달랐다. 인텔리J 코틀린 플러그인은 이런 경우 경고를 표시해서 잠재적인 버그를 줄여준다. 이런 경우에는 변수 이름과 프로퍼티 이름이 일치하게 구조분해 선언 부분을 변경하는 편을 권장한다(그림 6.2).

그림 6.2: 구조 분해 선언에서 변수 순서를 잘못 쓴 경우

구조분해 선언 전체는 타입이 없다. 하지만 필요하면 구조분해를 이루는 각 콤포넌트 변수에 타입을 표기할 수는 있다.

val (firstName, familyName: String, age) = Person("John", "Doe", 25)

구조분해 선언에 데이터 클래스의 프로퍼티 수보다 적은 수의 변수가 들어갈 수도 있다. 이런 경우 생성자의 뒷부분에 선언된 프로퍼티는 추출되지 않는다.

val (firstName, familyName) = Person("John", "Doe", 25)  
println("$firstName $familyName") // John Doe  
  
val (name) = Person("John", "Doe", 25)  
println(name) // John

시작 부분이나 중간 부분에서 몇가지 프로퍼티를 생략해야 한다면 어떻게 해야 할까? 코틀린 1.1부터는 람다의 사용하지 않는 파라미터와 비슷하게 사용하지 않는 부분을 \_로 대신할 수 있다.

val (\_, familyName) = Person("John", "Doe", 25)  
println(familyName) // Doe

val을 var로 바꾸면 변경할 수 있는 변수들을 얻을 수 있다.

var (firstName, familyName) = Person("John", "Doe", 25)  
firstName = firstName.toLowerCase()  
familyName = familyName.toLowerCase()  
println("$firstName $familyName") // john doe

[편집자께: val, var를 modifier로 보는건 좀 억지입니다.]

val/var 지정은 구조분해 선언의 모든 부분에 적용된다. 따라서 모든 변수를 가변 변수로 정의하거나 모든 변수를 불변 변수로 정의해야만 하며, 둘을 섞어서 정의할 수는 없다.

for 루프에서 구조분해를 사용할 수도 있다.

val pairs = arrayOf(1 to "one", 2 to "two", 3 to "three")  
  
for ((number, name) in pairs) {  
 println("$number: $name")  
}

코틀린 1.1부터는 람다 파라미터에 대해 구조분해를 사용할 수도 있다.

fun combine(person1: Person,  
 person2: Person,  
 folder: ((String, Person) -> String)): String {  
 return folder(folder("", person1), person2)  
}  
  
fun main() {  
 val p1 = Person("John", "Doe", 25)  
 val p2 = Person("Jane", "Doe", 26)  
   
 // 구조분해를 쓰지 않음  
 println(combine(p1, p2) { text, person -> "$text ${person.age}" })  
   
 // 구조분해를 씀  
 println(combine(p1, p2) { text, (firstName) -> "$text $firstName" })  
 println(combine(p1, p2) { text, (\_, familyName) -> "$text $familyName" })  
}

일반 람다 파라미터 목록과 달리 구조분해를 사용하는 파라미터를 괄호로 둘러싸야만 한다.

현재는 지역 변수에서만 구조분해 선언을 사용할 수 있다. 클래스 본문이나 파일의 최상위에 구조분해를 사용할 수는 없다.

data class Person(val firstName: String,  
 val familyName: String,  
 val age: Int)  
   
val (firstName, familyName) = Person("John", "Doe", 25) // Error

아직은 구조분해 안에 다른 구조분해를 내포시킬 수 없다.

data class Person(val firstName: String,  
 val familyName: String,  
 val age: Int)  
   
data class Mailbox(val address: String, val person: Person)  
  
fun main() {  
 val (address, (firstName, familyName, age)) =  
 Mailbox("Unknown", Person("John", "Doe", 25)) // Error  
}

데이터 클래스는 선언하기만 하면 자동으로 구조분해를 지원하지만, 일반적으로 아무 코틀린 타입이나 구조분해를 사용할 수 있게 구현이 가능하다. *11장. 도메인 특화 언어*에서 연산자 오버로딩 관습을 사용해 구조분해를 지원하는 방법을 알아본다.

## 인라인 클래스

프로그래밍 실무에서는 래퍼(wrapper) 클래스를 만드는 일이 자주 있다. 무엇보다 래퍼 클래스는 잘 알려진 어댑터(adapter) 디자인 패턴의 핵심이기도 하다. 예를 들어 프로그램에서 통화(돈)라는 개념을 처리하고 싶다고 하자. 돈의 수량은 근본적으로 수에 불과하지만, 의미가 다른 통화를 구분없이 섞어서 사용하고 싶지는 않다. 따라서 어떤 래퍼 클래스와 유틸리티 함수를 도입한다.

class Dollar(val amount: Int) // amount의 단위는 센트  
class Euro(val amount: Int) // amount의 단위는 센트  
fun Dollar.toEuro() = ...  
fun Euro.toDollar() = ...

이런 접근 방법의 문제는 런타입 부가비용이 든다는 점이다. 새로운 종류의 통화 수량을 만들 때마다 래퍼 클래스를 생성해야 하기 때문에 이런 부가비용이 생긴다. 방금 본 예에서 처럼 감싸야 하는 대상이 원시 타입의 값이면 부가비용 문제가 더 커진다. 수 값을 직접 다룰 때는 전혀 객체를 생성할 필요가 없기 때문이다. 원시 타입 대신 래퍼 클래스를 사용하면 프로그램 성능이라는 댓가를 지불해야 한다.

이런 문제를 해결하기 위해 코틀린 1.3부터는 인라인 클래스(inline class)라고 부르는 새로운 종류의 클래스를 도입했다.

### 인라인 클래스 정의하기

인라인 클래스를 정의하려면 inline class를 클래스 이름 앞에 붙여야 한다.

inline class Dollar(val amount: Int) // amount의 단위는 센트  
inline class Euro(val amount: Int) // amount의 단위는 센트

인라인 클래스의 주 생성자에는 불변 프로퍼티를 하나만 선언해야 한다. 런타임에 클래스 인스턴스는 별도의 래퍼 객체를 생성하지 않고 이 프로퍼티의 값으로 표현된다. 그래서 인라인 클래스라는 이름이 붙었다. 런타임에는 호출지점을 함수 본문으로 인라이닝하는 인라인 함수처럼 인라인 클래스 객체를 사용하는 위치를 대신 인라인 클래스에 들어있는 값이 들어간다.

인라인 클래스도 자체 함수와 프로퍼티를 포함할 수 있다.

inline class Dollar(val amount: Int) {  
 fun add(d: Dollar) = Dollar(amount + d.amount)  
 val isDebt get() = amount < 0  
}  
  
fun main() {  
 println(Dollar(15).add(Dollar(20)).amount) // 35  
 println(Dollar(-100).isDebt) // true  
}

하지만 인라인 클래스의 프로퍼티는 상태를 포함할 수 없다. 이유는 주 생성자에 있는 프로퍼티와 함께 상태를 인라이닝해야 하는데, 현재 코틀린 컴파일러는 오직 한 프로퍼티만 인라이닝하도록 지원하기 때문이다. 이 말은 뒷받침하는 필드나 lateinit이나 (lazy를 포함하는) 위임 객체 프로퍼티를 사용할 수 없다는 뜻이다. 인라인 클래스 프로퍼티로는 앞 예제의 isDebt처럼 명시적인 접근자만 허용된다.

인라인 클래스의 본문에서 var 프로퍼티를 정의할 수도 있다. 하지만 인라인 클래스 내부에는 가변 상태가 없기 때문에, 보통은 이런 var 프로퍼티는 거의 아무 의미가 없다.

다른 제약으로는 초기화 블럭을 쓸 수 없다는 점을 들 수 있다. Dollar(15)와 같은 생성자 호출이 단순히 15라는 값을 언급하는 경우와 같아야 하기 때문에, 인라인 클래스는 런타임에 아무 커스텀 코드도 실행할 수 없다.

*2장. 코틀린 언어 기초*에서 프로그램 안에서 널이 될 수 있는 타입의 변수에 원시 값을 대입하는 등의 경우에는 암시적으로 원시 값을 박싱한다고 설명했다. 인라인 클래스도 마찬가지다. 최적화를 위해 컴파일러는 가능하면 박싱하지 않은 값을 사용하려고 한다. 하지만 박싱하지 않은 값을 사용할 수 없는 경우 컴파일러는 인라이닝 되지 않는 형태로 클래스를 사용하는 편을 택한다. 컴파일러의 동작을 잘 예측하려면, 인라인 클래스 인스턴스는 다른 어떤 타입의 값으로 캐스팅하지 않고도 정확히 상응하는 (프로퍼티) 타입의 값으로 사용될 수 있는 경우에는 인라인될 수 있다는 대략적인 규칙을 기억하라. 다음 예제를 살펴보자.

fun safeAmount(dollar: Dollar?) = dollar?.amount ?: 0  
  
fun main() {  
 println(Dollar(15).amount) // 인라이닝됨  
 println(Dollar(15)) // Any?로 사용되기 때문에 인라이닝 되지 않음  
 println(safeAmount(Dollar(15))) // Dollar?로 사용되기 때문에 인라이닝 되지 않음  
}

[편집자께: 공식 문서에서 베타라는 표현을 썼습니다. 그래서 베타로 바꿉니다.]

인라인 클래스가 아직 베타(beta) 단계라는 점을 알아둬야 한다. 현 시점(2021년 상반기) 기준으로 인라인 클래스의 언어 명세는 아직 확정되지 않았고 미래에 변할 가능성이 남아 있다. 이로 인해 코틀린 1.3에서 인라인 클래스를 사용하면 컴파일러가 경고를 표시한다. 특별한 컴파일러 옵션으로 -XXLanguage:+InlineClasses를 지정하면 이런 경고를 없앨 수 있다.

IDE 팁

인텔리J를 쓸때는 프로젝트에서 Alt + Enter(맥에서는 Opt + Enter)를 누르면 나오는 메뉴에서 그림 6.3처럼 적절한 액션을 선택해 인라인 클래스를 켜거나 끌 수 있다.

그림 6.3: 인텔리J 프로젝트에서 인라인 클래스 활성화하기

### 부호없는 정수

코틀린 버전 1.3부터는 코틀린에 내장된 부호 있는 타입을 기반으로 인라인 클래스로 작성된 부호 없는 정수 타입이 코틀린 표준 라이브러리에 추가됐다. 일반적으로, 이런 타입들도 인라인 클래스와 마찬가지로 아직 베타 단계이며 프로젝트에서 명시적으로 끄지 않으면 컴파일시 경고가 표시된다(그림 6.4를 보라).

그림 6.4: 인텔리J 프로젝트에서 부호없는 정수 지원 활성화하기

부호 없는 정수 타입의 이름은 상응하는 부호 있는 정수 타입의 앞에 U를 덧붙인 이름이다.

|  |  |  |
| --- | --- | --- |
| 타입 | 크기(바이트) | 범위 |
| UByte | 1 | 0 ~ 255 |
| UShort | 2 | 0 ~ 65535 |
| UInt | 4 | 0 ~ 232-1 |
| ULong | 8 | 0 ~ 264-1 |

부호 없는 값을 표현하려면 정수 리터럴 뒤에 u나 U 접두사를 붙인다. 리터럴의 타입은 리터럴이 대입될 변수의 타입에 따라 결정되지만 변수에 타입을 지정하지 않은 경우에는 크기에 따라 UInt나 ULong 중 하나로 결정된다.

val uByte: UByte = 1u // 명시적으로 UByte  
val uShort: UShort = 100u // 명시적으로 UShort  
val uInt = 1000u // 자동으로 UInt로 추론  
val uLong: ULong = 1000u // 명시적으로 ULong  
val uLong2 = 1000uL // L 접미사가 붙었기 때문에 명시적으로 ULong

부호가 있는 타입과 부호가 없는 타입은 서로 호환되지 않는다. 예를 들어, 부호가 없는 타입의 값을 부호가 있는 타입의 변수에 저장하거나 반대 방향으로 저장할 수 없다.

val long: Long = 1000uL // Error

부호가 있는 타입과 없는 타입의 값을 toXXX()를 통해 반대쪽 타입으로 변환할 수 있다.

println(1.toUByte()) // 1, Int -> UByte  
println((-100).toUShort()) // 65436, Int -> UShort  
println(200u.toByte()) // -56, UInt -> Byte  
println(1000uL.toInt()) // 1000, ULong -> Int

부호가 없는 타입의 API는 부호가 있는 타입의 API와 꽤 비슷하다. 특히 +, -, \*, /, % 연산자의 경우 부호가 없는 값을 가능한 모든 방식으로 짝을 지어 사용할 수 있다.

println(1u + 2u) // 3  
println(1u - 2u) // 4294967295  
println(3u \* 2u) // 6  
println(5u / 2u) // 2  
println(7u % 3u) // 1

하지만 부호가 없는 타입의 값과 부호가 있는 타입의 값을 섞어서 사용할 수는 없다.

println(1u + 2) // Error  
println(1 + 2u) // Error

그리고, 부호 있는 값과 달리, 부호 없는 정수는 단항 부호 반전(-) 연산자를 지원하지 않는다. 부호 없는 정수는 음수를 표현할 수 없으므로 당연하다.

println(-1u) // Error

부호 없는 값을 증가시키거나 감소시키거나, 복합대입연산자에 쓸 수 있다.

var uInt: UInt = 1u  
++uInt  
uInt -= 3u

그리고 비트 반전, AND, OR, XOR과 같은 기본적인 비트 연산을 지원한다.

val ua: UByte = 67u // 01000011  
val ub: UByte = 139u // 10001011  
println(ua.inv()) // 10111100: 188  
println(ua or ub) // 11001011: 203  
println(ua xor ub) // 11001000: 200  
println(ua and ub) // 00000011: 3

UInt와 ULong은 왼쪽과 오른쪽 비트 시프트를 지원한다.

val ua = 67u // 0..0001000011  
println(ua shr 2) // 0..0000010000: 16  
println(ua shl 2) // 0..0100001100: 268

여기서 비트 수를 UInt가 아니라 보통의 Int 타입의 값으로 지정한다는 점에 유의하라. 그리고 부호가 없는 정수에서는 shr 과 ushr이 똑같기 때문에 별도로 부호없는 오른쪽 시프트 연산인 ushr를 제공하지는 않는다는 점에 유의하라.

일반 정수와 마찬가지로 <, >, <=, >=, ==, != 연산을 사용해 부호없는 값을 비교할 수 있다.

println(1u < 2u) // true  
println(2u >= 3u) // false  
println(2u + 2u == 1u + 3u) // true

코틀린 표준 라이브러리는 부호 없는 정수 배열을 표현하는 UByteArray, UShortArray, UIntArray, ULongArray 보조 타입을 포함한다. 지금까지 봐온 배열을 만드는 방법과 비슷한 방식으로 부호 없는 타입의 값으로 이뤄진 배열을 만들 수 있다.

val uBytes = ubyteArrayOf(1u, 2u, 3u)  
val squares = UIntArray(10) { it\*it }

범위와 진행 타입에도 부호 없는 타입이 있다. .. 연산자는 물론 until과 downTo를 사용해서 이런 타입의 값을 만들 수 있다.

1u .. 10u // 1, 2, 3, 4, 5, 6, 7, 8, 9, 10  
1u .. 10u step 2 // 1, 3, 5, 7, 9  
1u until 10u // 1, 2, 3, 4, 5, 6, 7, 8, 9  
10u downTo 1u // 10, 9, 8, 7, 6, 5, 4, 3, 2, 1  
10u downTo 1u step 3 // 10, 7, 4, 1

## 결론

이번 장에서는 특정 프로그래밍 문제를 풀때 쓸 수 있는 몇가지 특별한 종류의 클래스에 대해 살펴봤다. 이넘을 사용해 공통 함수와 프로퍼티가 있는 한정된 값의 집합을 기술하는 방법을 살펴보고, 데이터를 간단히 저장할 수 있는 클래스를 간편하게 설계하는 데이터 클래스와 데이터 클래스의 프로퍼티를 추출하는 구조분해에 대해 배웠다. 마지막으로 경량 래퍼를 만들 때 사용하기 위해 코틀린 1.3에 도입된 인라인 클래스의 기능을 살펴보고, 코틀린 인라인 클래스를 바탕으로 만들어진 부호 없는 정수에 대해 살펴봤다.

다음 장에서는 코틀린 표준 라이브러리에 초점을 맞춘다. 특히 기본적인 컬렉션 타입에 대해 다루면서 배열과 문자열에 대해 좀 더 살펴보고, I/O와 네트워크 관련 기능 및 이와 관련한 여러 유용한 유틸리티 함수에 대해 살펴본다.

## 문제

1. 이넘 클래스란 무엇인가? 이넘 클래스에 내장된 연산은 어떤 것들이 있는가?
2. when 식에서 이넘 클래스를 사용하는 경우에 대해 자세히 설명하라.
3. 커스텀 함수나 프로퍼티가 있는 이넘 클래스를 어떻게 정의하는가?
4. 데이터 클래스란 무엇인가? 데이터 클래스에 대해 컴파일러가 자동으로 생성해주는 연산(함수)는 무엇이 있는가? 어떻게 데이터 클래스의 인스턴스를 복사할 수 있는가?
5. 구조분해 선언이란 무엇인가? 어떤 용도로 구조분해 선언을 사용할 수 있는가?
6. 인라인 클래스의 목적이 무엇인가? 어떤 클래스를 인라인으로 선언하기 위해 필요한 요구사항이 무엇인가?
7. 코틀린의 부호 없는 타입과 이런 타입이 제공하는 내장 연산에 대해 설명하라. 부호가 있는 정수와 비교할 때 부호가 없는 정수 타입이 다른 부분은 무엇인가?

1. 옮긴이 - 해시맵에 넣을 때 hashCode만 정의하면 되는게 아닌가 생각하는 독자도 있을지 모르겠다. hashCode()로는 해시맵의 버킷만 결정할수 있기 때문에, 같은 버킷 안에 들어가는 키 사이에 같은지 여부를 결정할 방법이 없으면 해시맵이 정상적으로 작동할 수 없다. 같은 버킷에 들어가는 키가 여럿인 경우 키에 해당하는 값을 결정하려면 equals로 키를 비교해봐야 한다. [↑](#footnote-ref-1)