# 5장. 고급 함수와 함수형 프로그래밍 활용하기

이 장에서는 함수와 프로퍼티의 고급 사용법에 대해 다룬다. 앞부분에서는 코틀린을 활용한 함수형 프로그래밍의 기초를 살펴본다. 고차함수(high order function)의 개념을 소개하고 람다(lambda)나 익명 함수, 호출 가능한 참조(callable reference)를 사용해 함숫값(functional value)을 만드는 방법을 보여주며, 인라인(inline) 함수를 통해 런타임 부가비용을 들이지 않으면서 함수형 프로그래밍을 하는 방법에 대해서도 설명한다. 뒷부분에서는 기존 타입을 (그 소스 코드를) 변경하지 않고도 새로운 기능을 추가할 수 있게 해주는 확장 함수와 프로퍼티에 대해 다룬다.

## 구조

* 람다와 고차 함수
* 함수 타입(functional type)
* 호출 가능 참조
* 인라인 함수
* 지역을 벗어나는(non-local) return을 활용한 제어 흐름
* 확장 함수와 프로퍼티
* 영역 함수

## 목표

함수형 프로그래밍을 돕는 고차 함수, 람다, 호출 가능 참조 등의 코틀린 언어기능에 대해 배우고, 기존 타입을 더 보완할 수 있는 확장 함수나 프로퍼티 사용법을 배운다.

## 코틀린을 활용한 함수형 프로그래밍

이 절에서는 함수형 패러다임을 지원하는 코틀린 언어 기능을 소개한다. 함수형 프로그래밍은 프로그램 코드를 불변 값을 변환하는 함수의 합성으로 구성할 수 있다는 아이디어를 바탕으로 한다. 함수형 언어는 함수를 일급 시민(first class) 값으로 취급한다. 이 말은 함수를 다른 일반적인 타입의 값과 똑같이 취급한다는 뜻이다. 특히 일급시민이라는 말에는 변수에 값을 대입하거나 변수에서 값을 읽을 수 있고, 함수에 값을 전달하거나 함수가 값을 반환할 수 있다는 뜻이 들어있다. 이런 성질은 함수인 값을 데이터와 마찬가지로 조작할 수 있는 고차 함수라는 함수를 정의할 수 있게 해주며, 코드 추상화와 합성(composition)이 더 쉽게 가능한 유연성을 제공해준다.

### 고차 함수

앞에서 이미 람다를 사용해 계산을 수행하는 예제를 몇 가지 살펴봤다. 예를 들어 배열 생성자는 람다를 받았다. 이때 이 람다는 주어진 인덱스에 따라 배열 원소 값을 계산한다.

<코드>

val squares = IntArray(5) { n -> n\*n } // 0, 1, 4, 9, 16

</코드>

이 절에서는 람다와 고차함수를 더 자세히 살펴본다.

어떤 정수 배열의 원소의 합계를 계산하는 함수를 정의하고 싶다고 하자.

<코드>

fun sum(numbers: IntArray): Int {  
 var result = numbers.firstOrNull()  
 ?: throw IllegalArgumentException(“Empty array”)  
   
 for (i in 1..numbers.lastIndex) result += numbers[i]  
   
 return result  
}  
  
fun main() {  
 println(sum(intArrayOf(1, 2, 3))) // 6  
}

</코드>

이 함수를 더 일반화해서 곱셈이나 최댓값/최솟값 같이 다양한 집계 함수를 사용하게 하려면 어떻게 해야 할까? 함수 자체의 기본적인 루프 로직은 그대로 두고 중간 값들을 함수의 파라미터로 추출한 다음, 일반화한 함수를 호출할 때 이 파라미터에 적당한 연산을 제공하면 된다.

<코드>

fun aggregate(numbers: IntArray, op: (Int, Int) -> Int): Int {  
 var result = numbers.firstOrNull()  
 ?: throw IllegalArgumentException("Empty array")  
   
 for (i in 1..numbers.lastIndex) result = op(result, numbers[i])  
   
 return result  
}  
  
fun sum(numbers: IntArray) =  
 aggregate(numbers, { result, op -> result + op })  
   
fun max(numbers: IntArray) =  
 aggregate(numbers, { result, op -> if (op > result) op else result })  
   
fun main() {  
 println(sum(intArrayOf(1, 2, 3))) // 6  
 println(sum(intArrayOf(1, 2, 3))) // 3  
}

</코드>

op 파라미터가 다른 파라미터와 다른 점은 이 파라미터를 표현하는 타입이 함수 타입인 (Int, Int) ->Int라는 점뿐이다. 이 말은 이 op를 함수처럼 호출할 수 있다는 뜻이다. 이 예제에서 op 파라미터는 Int 값을 한 쌍 받아서 Int를 결과로 내놓는 함수다.

sum()과 max()함수를 보면 aggregate호출하는 쪽에서는 함숫값[[1]](#footnote-1)을 표현하는 람다식을 인자로 넘긴다는 사실을 알 수 있다. 람다식은 기본적으로 단순한 형태의 문법을 사용해 정의하는 이름이 없는 지역 함수다. 예를 들어 다음 식을 보자.

<코드>

{ result, op -> result + op }

</코드>

result와 op는 함수 파라미터 역할을 하며 -> 다음에 오는 식은 결과를 계산하는 식이다. 이 경우 명시적인 return이 불필요하다. 그리고 컴파일러는 파라미터 타입을 문맥으로부터 추론해준다.

이제 이런 기능을 좀 더 자세히 살펴보자.

### 함수 타입

함수 타입은 함수처럼 쓰일 수 있는 값들을 표시하는 타입이다. 문법적으로 이런 타입은 함수 시그니처(signature)와 비슷하며, 다음과 같이 두 가지 부분으로 구성된다.

1. 괄호로 둘러싸인 파라미터 타입 목록은 함숫값에 전달될 데이터의 종류와 수를 정의한다.
2. 반환 타입은 함수 타입의 함숫값을 호출하면 돌려받게 되는 값의 타입을 정의한다.

반환값이 없는 함수라도 함수 타입에서는 반환 타입을 반드시 명시해야 한다. 따라서 이런 경우 Unit을 반환 타입으로 사용한다.

예를 들어 (Int, Int) ->Boolean이라는 타입은 인자로 정수를 한 쌍 받아서 결과로 Boolean 값을 계산하는 함수를 뜻한다. 함수 정의에서와 달리 함수 타입 표기에서는 인자 타입 목록과 반환 타입과 사이를 :가 아닌 ->로 구분한다.

함수 타입인 값도 op(result, numbers[i])와 같이 마치 일반 함수처럼 호출할 수 있다. 함숫값을 호출하는 다른 방법은 invoke() 메서드를 사용하는 것이다. invoke() 메서드도 함수 타입의 (괄호 안에 있는) 파라미터 목록과 똑같은 개수와 타입의 인자를 받는다.

<코드>

result = op.invoke(result, numbers[i])

</코드>

**자바와 코틀린의 차이**: 자바 8부터는 단일 추상 메서드(Single Abstract Method, SAM) 인터페이스를 문맥에 따라 적절히 함수 타입처럼 취급하기 때문에 람다식이나 메서드 참조로 SAM 인터페이스를 인스턴스화할 수 있다. 하지만 코틀린에서 함숫값은 항상 (P1, …, Pn) -> R 형태의 함수 타입에 속하기 때문에 임의의 SAM 인터페이스로 암시적으로 변환할 수가 없다. 따라서 다음 자바 코드는 올바른 코드이지만,

<코드>

import java.util.function.Consumer;  
  
public class Main {  
 public static void main(String[] args) {  
 Consumer<String> consume = s ->System.out.println(s);  
 consume.accept("Hello");  
 }  
}

</코드>

이와 비슷한 코틀린 코드는 컴파일되지 않는다.

<코드>

import java.util.function.Consumer  
  
fun main() {  
 // Error: type mismatch  
 val consume: Consumer<String> = { s ->println(s) }  
   
 consume.accept("Hello")  
}

</코드>

하지만 자바와의 상호 운용성을 위해 코틀린은 자바에 정의된 SAM 인터페이스 대신 코틀린 함수 타입을 쓸 수 있게 변환해준다. 이런 변환에 대해서는 **12장 자바 상호 운용성**에서 자세히 살펴본다.

코틀린 1.4부터는 코틀린 인터페이스 앞에 fun을 붙이면 코틀린 인터페이스를 SAM 인터페이스로 취급한다. 이 기능을 사용하면 마치 자바의 SAM 인터페이스와 마찬가지로 코틀린 인터페이스를 람다로 인스턴스화 할 수 있다. https://kotlinlang.org/docs/fun-interfaces.html 를 보라.

함수가 인자를 받지 않는 경우에는 함수 타입의 파라미터 목록에 빈 괄호를 사용한다.

<코드>

fun measureTime(action: () -> Unit): Long {  
 val start = System.nanoTime()  
   
 action()  
   
 return System.nanoTime() - start  
}

</코드>

파라미터 타입을 둘러싼 괄호는 필수이기 때문에 함수 타입이 파라미터를 하나만 받거나 전혀 받지 않는 경우에도 괄호를 꼭 쳐야 한다.

<코드>

val inc: (Int) -> Int = { n -> n + 1 } // Ok  
val dec: Int -> Int = { n -> n - 1 } // Error

</코드>

함수 타입의 값을 함수의 파라미터에만 사용할 수 있는 것도 아니다. 실제로는 이런 함수 타입을 다른 타입이 쓰일 수 있는 모든 장소에 사용할 수 있다. 예를 들어 함숫값을 변수에 저장할 수도 있다.

<코드>

fun main() {  
 val lessThan: (Int, Int) -> Boolean = { a, b -> a < b }  
 println(lessThan(1, 2)) // true  
}

</코드>

변수 타입을 생략하면 정보가 충분하지 못해 컴파일러가 람다 파라미터의 타입을 추론할 수 없다.

<코드>

val lessThan = { a, b -> a < b } // Error

</코드>

이런 경우 파라미터의 타입을 명시하면 된다.

<코드>

val lessThan = { a: Int, b: Int -> a < b } // Ok

</코드>

다른 타입과 마찬가지로 함수 타입도 널이 될 수 있는 타입으로 지정할 수 있다. 이럴 때는 함수 타입 전체를 괄호로 둘러싼 다음에 물음표를 붙인다.

<코드>

fun measureTime(action: (() -> Unit)?): Long {  
 val start = System.nanoTime()  
   
 action?.invoke()  
   
 returnSystem.nanoTime() - start  
}  
  
fun main() {  
 println(measureTime(null))  
}

</코드>

괄호로 함수 타입을 둘러싸지 않으면 물음표의 효과가 전혀 달라진다. () -> Unit?는 Unit? 타입의 값을 반환하는 함수를 표현하는 타입이라는 점에 유의하라.

함수 타입을 다른 함수 타입 안에 내포시켜서 고차 함수의 타입을 정의할 수 있다.

<코드>

fun main() {  
 val shifter: (Int) -> (Int) -> Int = { n -> { i ->i + n } }  
   
 val inc = shifter(1)  
   
 val dec = shifter(-1)  
   
 println(inc(10)) // 11  
 println(dec(10)) // 9  
}

</코드>

->는 오른쪽 결합(right associative)이다. 따라서 (Int) -> (Int) -> Int는 실제로 (Int) -> ((Int) -> Int)라고 해석된다. 즉, 이 타입은 Int 값을 인자로 받아서 함수를 반환하는 함수를 뜻하며, 이때 이런 타입의 함수가 반환하는 값의 타입은 Int를 받아서 다른 Int를 내놓는 함수 타입이 된다. 만약 Int를 받아서 Int를 내놓는 함수를 인자로 받아서 Int를 결과로 돌려주는 함수를 표현하고 싶다면 다음 예제처럼 괄호를 써야 한다.

<코드>

fun main() {  
 val evalAtZero: ((Int) -> (Int)) -> Int = { f -> f(0) }  
   
 println(evalAtZero { n -> n + 1 }) // 1  
 println(evalAtZero { n -> n - 1 }) // -1  
}

</코드>

함수 타입의 파라미터 목록에 파라미터 이름을 포함시킬 수도 있다. 이런 경우 파라미터 이름은 그냥 문서화를 위한 것이며 타입이 표현하는 함숫값에는 전혀 영향을 끼치지 못한다.

<코드>

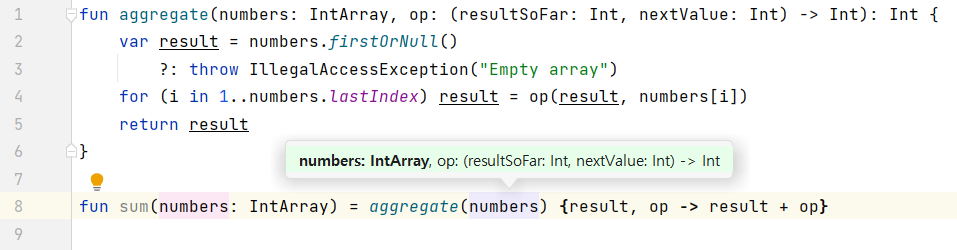
fun aggregate(  
 numbers: IntArray,  
 op: (resultSoFar: Int, nextValue: Int) -> Int  
): Int {...}

</코드>

<참고> IDE 팁

인텔리J IDEA에서 "Parameter Info" 기능을 통해 이런 파라미터를 볼 수 있다. 따라서 (함수 타입에 파라미터의 이름을 추가하면) 함숫값을 호출하는 부분에서 [Ctrl]+[P](또는 [Cmd]+[P])를 눌렀을 때 보이는 함수 시그니처에서 더 많은 힌트를 얻을 수 있다(그림 5-1을 보라).

그림 5-1: "Parameter Info"에서 함수 파라미터 이름을 표시



</참고>

### 람다와 익명 함수

함수형 타입의 구체적인 값을 어떻게 만들 수 있을까? 한 가지 방법은 함수를 묘사하되 이름을 지정하지는 않는 람다식을 사용하는 것이다. 예를 들어 앞에서 본 aggregate() 선언을 활용해 합계와 최댓값을 계산하는 함수를 만들어보자.

<코드>

fun sum(numbers: IntArray) =  
 aggregate(numbers, { result, op -> result + op })  
fun max(numbers: IntArray) =  
 aggregate(numbers, { result, op -> if (op > result) op else result })  
   
fun main() {  
 println(sum(intArrayOf(1, 2, 3))) // 6  
 println(sum(intArrayOf(1, 2, 3))) // 3  
}

</코드>

{ result, op -> result + op }라는 식을 람다식이고 부른다. 람다식 정의는 함수 정의와 비슷하게 다음과 같은 요소로 이뤄진다.

* 파라미터 목록: result, op
* 람다식의 몸통(본문)이 되는 식이나 문의 목록: result + op

함수 정의와 달리 반환 타입을 지정할 필요가 없다. 람다의 본문으로부터 반환 타입이 자동으로 추론된다. 그리고 람다 본문에서 맨 마지막에 있는 식이 람다의 결괏값이 된다.

람다의 파라미터 목록을 괄호로 둘러싸지 않는다는 점에 유의하라. 람다 정의에서 파라미터를 괄호로 감싸면 구조분해(destructuring) 선언이 된다. 이에 대해서는 **6장 특별한 클래스 사용하기**에서 살펴본다.

람다가 함수의 마지막 파라미터인 경우, 함수를 호출할 때 인자를 둘러싸는 괄호 밖에 이 람다를 위치시킬 수 있다. 코틀린에서는 이런 스타일의 코드를 권장한다. 이미 배열 생성자 코드와 지연 계산 프로퍼티에서 이런 방식으로 람다를 사용하는 예를 살펴봤다.

<코드>

fun sum(numbers: IntArray) =  
 aggregate(numbers) { result, op -> result + op }  
  
fun max(numbers: IntArray) =  
 aggregate(numbers) { result, op -> if (op > result) op else result }

</코드>

<참고> IDE 팁

인텔리J 코틀린 플러그인은 람다를 일반 인자 목록의 밖에 위치시킬 수 있는데 인자 목록 안에 포함시킨 경우 경고를 표시해주고, 자동으로 람다를 인자 목록 밖으로 빼는 코드 변경을 (프로그래머가 요청하면) 자동으로 수행해준다.

</참고>

람다에 인자가 없으면 화살표 기호(->)를 생략할 수 있다.

<코드>

fun measureTime(action: () -> Unit): Long {  
 val start = System.nanoTime()  
 action()  
 return System.nanoTime() - start  
}  
  
val time = measureTime{ 1 + 2 }

</코드>

코틀린은 인자가 하나밖에 없는 람다를 특별히 단순화해 사용할 수 있는 문법을 제공한다. 람다 인자가 하나인 경우, 파라미터 목록과 화살표 기호를 생략하고, 유일한 파라미터는 미리 정해진 it이라는 이름을 사용해 가리킬 수 있다.

<코드>

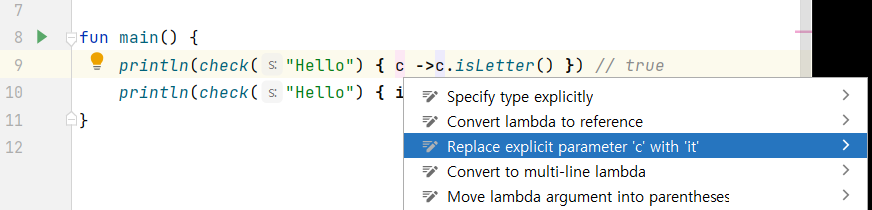
fun check(s: String, condition: (Char) -> Boolean): Boolean {  
 for (c in s) {  
 if (!condition(c)) return false  
 }  
 return true  
}  
  
fun main() {  
 println(check("Hello") { c ->c.isLetter() }) // true  
 println(check("Hello") { it.isLowerCase() }) // false  
}

</코드>

<참고> IDE 팁

인텔리J 코틀린 플러그인은 it을 사용하는 람다를 명시적인 파라미터를 사용하는 람다로 바꿔주거나 반대 방향으로 바꿔줄 수 있다. 캐럿을 파라미터를 참조하는 부분(또는 it을 사용한 부분)이나 파라미터 목록에 위치시키고 [Alt]+[Enter]를 누르면 된다(그림 5-2).

그림 5-2: 명시적 파라미터를 it으로 바꾸기



</참고>

코틀린 1.1부터는 람다의 파라미터 목록에서 사용하지 않는 람다 파라미터를 밑줄 기호(\_)로 지정할 수 있다.

<코드>

fun check(s: String, condition: (Int, Char) -> Boolean): Boolean {  
 for (i in s.indices) {  
 if (!condition(i, s[i])) return false  
 }  
 return true  
}  
  
fun main() {  
 println(check("Hello") { \_, c ->c.isLetter() }) // true  
 println(check("Hello") { i, c ->i == 0 || c.isLowerCase() }) // true  
}

</코드>

함숫값을 만드는 다른 방법은 익명 함수를 사용하는 것이다.

<코드>

fun sum(numbers: IntArray) =  
 aggregate(numbers, fun(result, op) = result + op)

</코드>

익명 함수는 일반 함수와 거의 문법이 똑같다. 몇 가지 차이점이 있는데, 다음과 같다.

* 익명 함수에는 이름을 지정하지 않는다. 따라서 fun 키워드 다음에 바로 파라미터 목록이 온다.
* 람다와 마찬가지로 문맥에서 파라미터 타입을 추론할 수 있으면 파라미터 타입을 지정하지 않아도 된다.
* 함수 정의와 달리, 익명 함수는 식이기 때문에 인자로 함수에 넘기거나 변수에 대입하는 등 일반 값처럼 쓸 수 있다(이는 객체 정의와 익명 객체 식의 관계와도 비슷하다).

람다와 달리 익명 함수에서는 반환 타입을 적을 수 있다. 반환 타입 관련해서는 함수 정의 시와 익명 함수 정의 시에 똑같은 규칙을 적용한다. 즉, 함수 본문이 식인 경우에는 반환 타입을 생략할 수 있고(생략 시 컴파일러가 타입을 추론할 수 있다), 함수 본문이 블록인 경우(특히 함수 반환 타입이 Unit 타입이 아닌 경우)에는 명시적으로 반환 타입을 지정해야 한다.

<코드>

fun sum(numbers: IntArray) =  
 aggregate(numbers, fun(result, op): Int { return result + op })

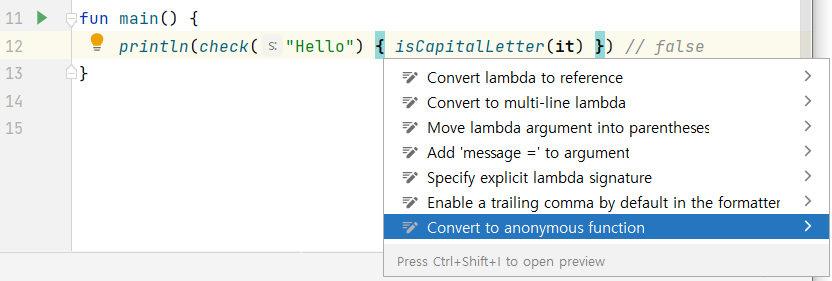
</코드>

람다와 달리 익명 함수를 인자 목록의 밖으로 내보낼 수는 없다.

<참고> IDE 팁

인텔리J 코틀린 플러그인은 람다와 익명 함수 사이를 자동으로 변환해주는 기능을 제공한다. 편집창의 캐럿을 람다를 여는 중괄호나 익명 함수의 fun에 위치시키고 [Alt]+[Enter]를 누르면 이 기능을 쓸 수 있다(그림 5-3)

그림 5-3: 람다 식을 익명 함수로 바꾸기



</참고>

지역 함수와 마찬가지로 람다나 익명 함수도 클로저, 또는 자신을 포함하는 외부 선언에 정의된 변수에 접근할 수 있다. 특히 람다나 익명 함수도 외부 영역의 가변 변수 값을 변경할 수 있다.

<코드>

fun forEach(a: IntArray, action: (Int) -> Unit) {  
 for (n in a) {  
 action(n)  
 }  
}  
  
fun main() {  
 var sum = 0  
 forEach(intArrayOf(1, 2, 3, 4)) {  
 sum += it  
 }  
 println(sum) // 10  
}

</코드>

**자바와 코틀린의 차이**: 반면, 자바 람다는 외부 변수의 값을 변경할 수 없다. 이는 **4장 클래스와 객체 다루기**에서 본 지역 클래스나 익명 객체에서 외부 변수를 변경하는 경우와 비슷하다.

### 호출 가능 참조

앞 절에서는 람다나 익명 함수를 사용해 함숫값을 만드는 방법을 살펴봤다. 하지만 이미 함수 정의가 있고, 이 함수정의를 함숫값처럼 고차함수에 넘기고 싶다면 어떻게 해야 할까? 물론 이를 다음과 같이 람다식으로 감싸서 전달할 수도 있다.

<코드>

fun check(s: String, condition: (Char) -> Boolean): Boolean {  
 for (c in s) {  
 if (!condition(c)) return false  
 }  
   
 return true  
}  
  
fun isCapitalLetter(c: Char) = c.isUpperCase() && c.isLetter()  
  
fun main() {  
 println(check("Hello") { c -> isCapitalLetter(c) }) // false  
 // 또는   
 println(check("Hello") { isCapitalLetter(it) }) // false  
}

</코드>

하지만 코틀린에는 이미 존재하는 함수 정의를 함수 타입의 식으로 사용할 수 있는 더 단순한 방법이 있다. 호출 가능 참조(callable reference)를 사용하면 된다.

<코드>

fun main() {  
 println(check("Hello", ::isCapitalLetter)) // false  
}

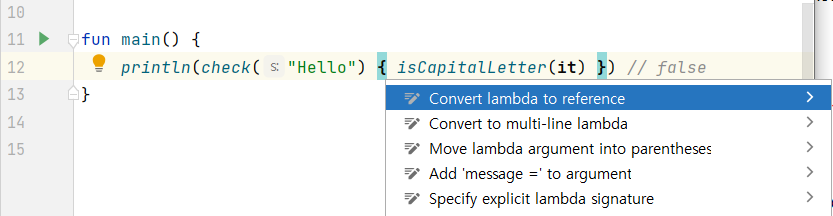
</코드>

::isCapitalLetter라는 식은 이 식이 가리키는 isCapitalLetter() 함수와 같은 동작을 하는 함숫값을 표현해준다.

<참고> IDE 팁

인텔리J 코틀린 플러그인은 람다식을 (변환이 가능한 경우에) 호출 가능 참조로 바꾸거나, 역으로 바꾸는(호출 가능 참조를 람다로 바꾸는 건 항상 가능함) 기능을 제공한다. [Alt]+[Enter] 메뉴에서 이 기능을 사용할 수 있다(그림 5-4).

그림 5-4: 람다를 호출 가능 참조로 바꾸기



</참고>

가장 간단한 형태의 호출 가능 참조는 최상위나 지역 함수를 가리키는 참조다. 이런 함수를 가리키는 참조를 만들려면 함수 이름 앞에 ::를 붙이면 된다.

<코드>

fun evalAtZero(f: (Int) -> Int) = f(0)  
  
fun inc(n: Int) = n + 1  
  
fun main() {  
 fun dec(n: Int) = n - 1  
 println(evalAtZero(::inc)) // 1  
 println(evalAtZero(::dec)) // -1  
}

</코드>

호출 가능 참조를 만들 때는 함수 이름을 간단한 이름으로만 써야 한다. 따라서 다른 패키지에 들어있는 함수의 호출 가능 참조를 만들려면 먼저 함수를 임포트해야 한다.

::를 클래스 이름 앞에 적용하면 클래스의 생성자에 대한 호출 가능 참조를 얻는다.

<코드>

class Person(valfirstName: String, valfamilyName: String)  
  
fun main() {  
 val createPerson= ::Person  
 createPerson("John", "Doe")  
}

</코드>

코틀린 1.1부터는 바인딩된 호출 가능 참조(bound callable reference)라는 :: 사용법이 도입됐다. 주어진 클래스 인스턴스의 문맥 안에서 멤버 함수를 호출하고 싶을 때 바인딩된 호출 가능 참조를 사용한다.

<코드>

class Person(val firstName: String, val familyName: String) {  
 fun hasNameOf(name: String) = name.equals(firstName,   
 ignoreCase = true)  
}  
  
fun main() {  
 val isJohn = Person("John", "Doe")::hasNameOf  
   
 println(isJohn("JOHN")) // true  
 println(isJohn("Jake")) // false  
}

</코드>

특정 인스턴스와 바인딩하지 않고 멤버 함수를 가리키는 다른 형태의 호출 가능 참조도 있다. 이런 참조를 **수신 객체가 있는 호출 가능 참조** 절에서 다룬다.

호출 가능 참조 자체는 오버로딩된 함수를 구분할 수 없다는 점에 유의하라. 오버로드된 함수 중 어떤 함수를 참조할지 명확히 하려면 컴파일러에게(물론 사람에게도) 타입을 지정해줘야 한다.

<코드>

fun max(a: Int, b: Int) = if (a > b) a else b  
fun max(a: Double, b: Double) = if (a > b) a else b  
  
val f: (Int, Int) -> Int = ::max // Ok  
val g = ::max // Error: ambiguous reference

</코드>

호출 가능 참조에서 구체적인 함수 시그니처를 지정할 수 있는 기능이 나중에 코틀린에 추가될 지도 모른다. 이런 이유로 향후 괄호를 사용해 호출 가능 참조를 더 세분화할 수 있는 여지를 남기기 위해, 호출 가능 참조 뒤에 괄호를 바로 붙이는 문법은 현재 예약되어 있다. 호출 가능 참조를 직접 호출하고 싶다면 참조 전체를 괄호로 둘러싼 다음에 인자를 지정해야 한다.

fun max(a: Int, b: Int) = if (a > b) a else b  
  
fun main() {  
 println((::max)(1, 2)) // 2  
 println(::max(1, 2)) // Error: this syntax is reserved for future use  
}

코틀린 프로퍼티에 대한 호출 가능 참조를 만들 수도 있다. 이런 참조 자체는 실제로는 함숫값이 아니고, 프로퍼티 정보를 담고 있는 리플렉션(reflection) 객체다. 이 객체의 getter 프로퍼티를 사용하면 게터 함수에 해당하는 함숫값에 접근할 수 있다. var 선언의 경우에는 리플렉션 객체의 setter 프로퍼티를 통해 세터 함수에 접근할 수 있다.

<코드>

class Person(var firstName: String, var familyName: String)  
  
fun main() {  
 val person = Person(“John”, “Doe”)  
 val readName = person::firstName.getter // 게터 참조  
 val writeFamily = person::familyName.setter // 세터 참조  
   
 println(readName()) // John  
 writeFamily(“Smith”)  
 println(person.familyName) // Smith  
}

</코드>

현재는 지역 변수에 대한 호출 가능 참조를 지원하지 않는다. 하지만 코틀린의 이후 버전에는 이에 대한 지원이 포함될 수도 있다.

**자바와 코틀린의 차이**: 자바를 잘 아는 독자라면 코틀린 호출 가능 참조와 자바8의 메서도 참조 사이의 유사성을 눈치챈 사람도 있을 것이다. 둘의 의미는 매우 비슷하지만, 몇 가지 중요한 차이가 있다. 첫째, 호출 가능 참조는 자바에는 없는 종류의 선언을 코틀린이 지원하기 때문에 자바의 메서드 참조보다 종류가 더 많다. 둘째, 코틀린의 호출 가능 참조는 일급 시민 식이지만, 자바 메서드 참조는 함수형 인터페이스 내에서만 의미가 있다. 즉, 자바의 메서드 참조에는 정해진 타입이 없다. 게다가 호출 가능 참조는 단순히 함숫값만이 아니라 런타임에 함수나 프로퍼티의 애트리뷰트를 얻을 때 사용할 수 있는 리플렉션 객체이기도 한다. **10장 애너테이션과 리플렉션**에서 리플렉션 API에 대해 더 자세히 다룬다.

### 인라인 함수와 프로퍼티

고차 함수와 함숫값을 사용하면 함수가 객체로 표현되기 때문에 성능상 부가비용이 생긴다. 더 나아가 익명 함수나 람다가 외부 영역의 변수를 참조하면 고차 함수에 함숫값을 넘길 때마다 이런 외부 영역의 변수를 포획할 수 있는 구조도 만들어서 넘겨야 한다. 함숫값을 호출할 때는 컴파일러가 함숫값의 정적인 타입을 알 수 없기 때문에 동적으로 가상 호출을 사용해 어떤 함수 구현을 사용할지 디스패치해야 한다.

하지만 코틀린은 함숫값을 사용할 때 발생하는 런타임 비용을 줄일 수 있는 해법을 제공한다. 기본적인 아이디어는 함숫값을 사용하는 고차함수를 호출하는 부분을 해당 함수의 본문으로 대체하는 인라인(inline) 기법을 쓰는 것이다. 인라인될 수 있는 함수를 구별하기 위해 프로그래머는 inline 변경자를 함수 앞에 붙여야 한다.

예를 들어 정수 배열에서 주어진 술어(predicate)를 만족하는 값을 찾는 함수가 있다고 하자.

<코드>

inline fun indexOf(numbers: IntArray, condition: (Int) -> Boolean): Int {  
 for (i in numbers.indices) {  
 if (condition(numbers[i])) return i  
 }  
   
 return -1  
}  
  
fun main() {  
 println(indexOf(intArrayOf(4, 3, 2, 1)) { it < 3 }) // 2  
}

</코드>

indexOf() 함수가 인라인됐기 때문에 컴파일러는 함수 호출을 함수의 본문으로 대체한다. 이 말은 main() 함수가 다음 코드와 같이 번역된다는 뜻이다.

<코드>

fun main() {  
 val numbers = intArrayOf(4, 3, 2, 1)  
 var index = -1  
  
 for (i in numbers.indices) {  
 if (numbers[i] < 3) {  
 index = i  
 break  
 }  
 }  
   
 println(index)  
}

</코드>

인라인 함수를 쓰면 컴파일된 코드의 크기가 커지지만, 지혜롭게 사용하면 성능을 크게 높일 수 있다. 특히, 대상 함수가 상대적으로 작은 경우 성능이 크게 향상된다. **7장 컬렉션과 I/O 자세히 알아보기**에서 볼 코틀린 표준 라이브러리가 제공하는 여러 고차함수 중에 상당수가 실제로 인라인 함수다.

함수 인라이닝을 지원하는 몇몇 프로그래밍 언어(C++ 등)와 달리 코틀린의 inline 변경자는 컴파일러가 상황에 따라 무시해도 되는 최적화를 위한 힌트가 아니다. inline이 붙은 코틀린 함수는 가능하면 항상 인라인이 되며, 인라인이 불가능한 경우에는 컴파일 오류로 간주된다.

앞의 예제는 inline 변경자가 붙은 함수뿐 아니라 이 함수의 파라미터로 전달되는 함숫값도 인라인된다는 사실을 보여준다. 이로 인해 이런 인라인 함수에 대한 조작이 제한된다. 예를 들어 인라인 함수는 실행 시점에 별도의 존재가 아니기 때문에 변수에 저장되거나, 인라인 함수가 아닌 함수에 전달될 수가 없다. 인라인이 될 수 있는 람다를 사용해 할 수 있는 일은 람다를 호출하거나 다른 인라인 함수에 인라인이 되도록 넘기는 두 가지뿐이다.

<코드>

var lastAction: () -> Unit = {}  
  
inline fun runAndMemorize(action: () -> Unit) {  
 action()  
 lastAction = action // Error  
}

</코드>

마찬가지 이유로 인라인 함수가 널이 될 수 있는 함수 타입의 인자를 받을 수도 없다.

<코드>

inline fun forEach(a: IntArray, action: ((Int) -> Unit)?) { // Error  
 if (action == null) return  
 for (n in a) action(n)  
}

</코드>

이런 경우 특정 람다를 인라이닝하지 말라고 파라미터 앞에 noinline 변경자를 붙일 수 있다.

<코드>

inline fun forEach(a: IntArray, noinline action: ((Int) -> Unit)?) {  
 if (action == null) return  
 for (n in a) action(n)  
}

</코드>

어떤 함수에 인라이닝할 수 있는 파라미터가 없다면 이 함수를 호출한 지점을 함수 본문으로 대치해도 런타임에 크게 이득이 없기 때문에, 보통 이런 함수는 인라이닝할 만한 가치가 없는 것으로 여겨진다. 이 때문에 코틀린 컴파일러는 이런 함수에 대해 경고를 표시한다.

공개 인라인 함수에게 비공개 멤버를 넘기려고 하면 어떻게 될까? 인라인 함수의 본문이 호출 지점을 대신하게 되므로, 이런 경우에는 외부에서 캡슐화를 깰 수 있게 된다. 비공개 코드가 외부로 노출되는 일을 방지하기 위해 코틀린은 인라인 함수에 비공개 멤버를 전달하는 것을 금지한다.

<코드>

class Person(private valfirstName: String,

private valfamilyName: String) {

inline fun sendMessage(message: () -> String) {

println("$firstName $familyName: ${message()}") // Error

}

}

</코드>

만약 sendMessage() 함수나 이 함수를 포함하는 클래스를 private 변경자로 지정한다면, sendMessage()의 본문에 있는 비공개 멤버에 대한 참조도 Person 클래스의 외부로 누출될 수 없다.

코틀린 버전 1.1부터는 프로퍼티 접근자를 인라이닝하도록 허용한다. 이 기능을 사용하면 함수 호출을 없애기 때문에 프로퍼티를 읽거나 쓰는 성능을 향상시킬 수 있다. 다음 코드에서는 fullName 게터에 대한 모든 호출이 인라이닝된다.

<코드>

class Person(var firstName: String, var familyName: String) {  
 var fullName  
 inline get() = "$firstName $familyName" // inline 게터  
 set(value) { ... } // inline이 아닌 세터  
}

</코드>

개별 접근자를 인라이닝하는 것 외에 프로퍼티 자체에 inline 변경자를 붙일 수도 있다. 이렇게 하면 컴파일러가 게터와 (프로퍼티가 가변 프로퍼티인 경우) 세터를 모두 인라닝해준다.

<코드>

class Person(var firstName: String, var familyName: String) {  
 inline var fullName // inline 게터와 세터  
 get() = "$firstName $familyName"  
 set(value) { ... }  
}

</코드>

프로퍼티에 대한 인라이닝은 뒷받침하는 필드가 없는 프로퍼티에 대해서만 가능하다. 그리고 함수와 비슷하게 프로퍼티가 공개 프로퍼티인 경우, 프로퍼티의 게터나 세터 안에서 비공개 선언을 참조하면 인라이닝이 불가능하다는 점도 기억하라.

<코드>

class Person(private val firstName: String,  
 private val familyName: String) {  
  
 inline var age = 0 // Error: property has a backing field  
 // Error: firstName and familyName are private  
 inline valfullNameget() = "$firstName $familyName"  
}

</코드>

## 비 지역적 제어 흐름

고차 함수를 사용하면 return문 등 일반적인 제어 흐름을 깨는 명령을 사용할 때 문제가 생긴다. 예를 들어 다음 코드를 살펴보자.

<코드>

fun forEach(a: IntArray, action: (Int) -> Unit) {  
 for (n in a) action(n)  
}  
  
fun main() {  
 forEach(intArrayOf(1, 2, 3, 4)) {  
 if (it < 2 || it > 3) return  
 println(it) // Error  
 }  
}

</코드>

프로그램의 의도는 어떤 범위 안에 들어있지 않은 수를 출력하기 전에 람다를 return을 사용해 종료하는 것이다. 하지만 이 코드는 컴파일되지 않는다. return문은 디폴트로 자신을 둘러싸고 있는 fun, get, set으로 정의된 가장 안쪽 함수로부터 제어 흐름을 반환시킨다. 따라서 앞의 예제는 실제로는 main 함수로부터 반환을 시도하는 코드가 된다. 이런 문을 비 지역적 return이라고 부른다. JVM에서는 람다가 효율적으로 자신을 둘러싸고 있는 함수를 반환시킬 방법이 없기 때문에, 이런 문을 금지한다. 이런 경우를 해결하는 방법은 람다 대신 익명 함수를 사용하는 것이다.

<코드>

fun main() {  
 forEach(intArrayOf(1, 2, 3, 4), fun(it: Int) {  
 if (it < 2 || it > 3) return  
 println(it)  
 })  
}

</코드>

람다 자체로부터 제어 흐름을 반환하고 싶다면 break나 continue에 대해 레이블을 사용했던 것처럼, return 문에 문맥 이름을 추가해야 한다. 일반적으로 함수 리터럴 식에 이름을 붙여서 문맥 이름을 만들 수 있다. 예를 들어 아래 코드는 myFun이라는 레이블을 action 변수 초기화 앞부분에 붙인다.

<코드>

val action: (Int) -> Unit = myFun@ {  
 if (it < 2 || it > 3) return@myFun  
 println(it)  
}

</코드>

하지만 람다를 고차 함수의 인자로 넘기는 경우에는 레이블을 명시적으로 선언하지 않아도 함수 이름을 문맥으로 사용할 수 있다.

<코드>

forEach(intArrayOf(1, 2, 3, 4)) {  
 if (it < 2 || it > 3) return@forEach  
 println(it)  
}

</코드>

이러한 한정시킨(qualified) return을 일반 함수에서도 사용할 수 있다. 불필요한 중복이긴 하지만, 함수 이름을 레이블처럼 쓸 수 있다.

<코드>

fun main(args: Array<String>) {  
 if (args.isEmpty()) return@main  
 println(args[0])  
}

</코드>

람다가 인라인될 때는 인라인된 코드를 둘러싸고 있는 함수에서 반환할 때 return 문을 사용할 수 있다. 고차 함수가 인라인 함수인 경우, 고차 함수를 호출하는 코드를 고차함수 본문과 람다 본문으로 대체하기 때문이 이런 동작이 가능하다. 다음 코드의 return문은 main()에서 반환되는 것으로 취급된다.

<코드>

inline fun forEach(a: IntArray, action: (Int) -> Unit) { ... }  
  
fun main() {  
 forEach(intArrayOf(1, 2, 3, 4)) {  
 if (it < 2 || it > 3) return // main에서 반환됨  
 println(it)  
 }  
}

</코드>

고차 함수가 인라인이 될 수 있는 람다를 받는데, 이 고차 함수의 본문에서 람다를 직접 호출하지는 않고 지역 함수나 지역 클래스의 메서드 등의 다른 문맥에서 간접적으로 호출할 수도 있다. 이런 경우에도 람다를 인라이닝할 수는 있지만, 인라이닝을 한 이후 람다에서 사용하는 return문이 고차 함수를 호출하는 쪽의 함수를 반환시킬 수는 없다. 인라이닝을 했음에도 불구하고 람다의 return과 람다를 실행해주는 함수가 서로 다른 실행 스택 프레임을 차지하기 때문이다. 그렇기 때문에 이런 식으로 함수 파라미터를 호출하는 일은 디폴트로 금지되어 있다.

<코드>

private inline fun forEach(a: IntArray, action: (Int) -> Unit) = object {  
 fun run() {  
 for (n in a) {  
 action(n) // Error  
 }  
 }  
}

</코드>

이런 호출을 허용하려면 함수형 파라미터 앞에 crossinline 변경자를 붙여야 한다. 이 변경자는 함숫값을 인라인시키도록 남겨두는 대신 람다 안에서 비 지역 return을 사용하지 못하게 막는 역할을 한다.

<코드>

private inline fun forEach(  
 a: IntArray, crossinline action: (Int) -> Unit  
) = object {  
 fun run() {  
 for (n in a) {  
 action(n) // Ok  
 }  
 }  
}  
  
fun main() {  
 forEach(intArrayOf(1, 2, 3, 4)){  
 if (it < 2 || it > 3) return // Error  
 println(it)  
 }  
}

</코드>

break나 continue를 쓸 때도 비 지역적 제어 흐름을 만들어낼 수 있다. 이런 경우 비 지역적 break나 continue는 람다를 둘러싼 루프를 대상으로 제어 흐름을 변경하게 된다. 현재는 람다가 인라인되더라도 이런 비 지역적 break나 continue를 사용하지는 못한다. 하지만 나중에는 코틀린에 이런 기능이 추가될 예정이다.

<코드>

while (true) {  
 forEach(intArrayOf(1, 2, 3, 4)) {  
 if (it < 2 || it > 3) break // Error  
 println(it)  
 }  
}

</코드>

## 확장

실무에서는 기존 클래스를 확장해야 하는 경우가 자주 있다. 프로그램이 진화함에 따라 개발자가 클래스에 새 함수나 프로퍼티를 추가해서 해당 클래스가 제공하는 API를 확장하고 싶을 수 있다. 하지만 대상 클래스가 다른 라이브러리에 들어있어서 (소스 코드에 대한 소유권이 없어서) 코드를 변경 할 수 없거나 클래스를 변경하는 비용이 너무 들어서 코드를 변경하지 못할 수도 있다. 게다가 한 클래스 안에 속한 메서드를 모드 함께 사용하지 않는데 모든 가능한 메서드를 다 넣어두면 실용적이지 못할 수도 있다. 이런 경우 여러 프로그램 모듈에 함께 사용되는 메서드를 서로 분리해 유지하는 편이 나을 수도 있다.

자바에서는 이런 추가 메서드를 별도의 유틸리티 클래스로 묶곤 한다. 일반적인 예로 Collection 인터페이스의 기능을 확장하는 여러 메서드가 들어있는 java.util.Arrays와 java.util.Collections를 들 수 있다. 이런 클래스의 문제는 불필요한 준비 코드가 너무 많이 필요하다는 점이다. 예를 들어 자바에서 유틸리티 메서드를 사용하는 전형적인 코드는 다음과 비슷하다.

<코드>

int index = Collections.indexOfSubList(  
 Arrays.asList("b", "c", "a"),  
 Arrays.asList("a", "b")  
)

</코드>

소스 코드가 지저분하다는 점 외에도, 이런 호출을 사용할 때는 인텔리J나 이클립스 같은 주요 IDE에서 자동완성을 사용할 수 없다는 단점이 있다.

그래서 코틀린은 마치 멤버인 것처럼 쓸 수 있는 함수나 프로퍼티를 클래스 밖에서 선언할 수 있게 해주는 확장이라는 기능을 제공한다. 확장을 사용하면 기존 클래스를 변경하지 않아도 새로운 기능으로 기존 클래스를 확장할 수 있어서 개방/폐쇄(open/closed) 디자인 원칙을 지원할 수 있다.

### 확장 함수

확장 함수는 어떤 클래스의 멤버인 것처럼 호출할 수 있는 (그러나 실제로는 멤버가 아닌) 함수를 뜻한다. 이런 함수를 정의할 때는 함수를 호출할 때 사용할 수신 객체의 클래스 이름을 먼저 표시하고, 점을 추가한 다음에 함수 이름을 표시한다. 예를 들어 String 타입에 문자열의 길이를 지정한 길이 이하로 제한하는 함수를 추가해서 확장하고 싶다고 하자. 가능한 구현은 다음과 같다.

<코드>

fun String.truncate(maxLength: Int): String {  
 return if (length <= maxLength) this else substring(0, maxLength)  
}

</코드>

일단 정의하고 나면 다른 String 클래스의 멤버와 마찬가지로 이 함수를 사용할 수 있다.

<코드>

fun main() {  
 println("Hello".truncate(10)) // Hello  
 println("Hello".truncate(3)) // Hel  
}

</코드>

일반 멤버와 비슷하게, 확장 함수의 본문 안에서 수신 객체를 this로 접근할 수 있다는 점에 유의하라. 앞의 예제에서 truncate 함수의 본문에서 substring을 호출할 때와 마찬가지로, this를 명시하지 않아도 암시적으로 this를 사용해 수신 객체의 멤버나 확장 함수에 접근할 수 있다.

하지만 이 점에 주목하라. 확장 함수 자체는 수신 객체가 속한 타입의 캡슐화를 깰 수 없다. 예를 들어 확장 함수는 클래스 밖에 정의된 함수이기 때문에 수신 객체가 속한 클래스의 비공개 멤버에 접근할 수가 없다.

<코드>

class Person(val name: String, private val age: Int)  
  
fun Person.showInfo() = println("$name, $age") // Error: can’t access age

</코드>

하지만 클래스 본문 안에서 확장 함수를 정의할 수도 있고, 이렇게 정의할 경우에는 확장 함수가 수신 객체의 멤버인 동시에 확장 함수가 된다. 따라서 이런 함수 안에서는 클래스 본문에 들어있는 다른 함수들과 마찬가지로 클래스의 비공개 멤버에 마음대로 접근할 수 있다.

<코드>

class Person(val name: String, private val age: Int) {  
 // Ok: age is accessible  
 fun Person.showInfo() = println("$name, $age")  
}

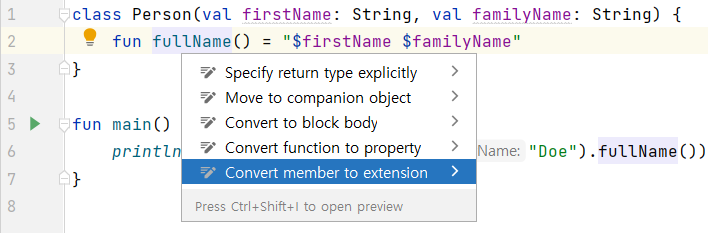
</코드>

이런 함수가 어떤 경우 도움이 되는지 이 장 뒷부분에서 살펴보자.

<참고> IDE 팁

인텔리J 코틀린 플러그인은 클래스 멤버를 확장으로 바꿀 수 있다. 멤버 이름에 캐럿을 위치시키고 [Alt]+[Enter]를 누르면 보이는 메뉴에서 Convert member to extension 액션을 선택하면 된다(그림 5-5).

그림 5-5: 멤버 함수를 확장으로 변환하기



</참고>

확장 함수는 일반 클래스 멤버와 비슷하게 바인딩된 호출 가능 참조 위치에 사용할 수도 있다.

<코드>

class Person(val name: String, val age: Int)  
  
fun Person.hasName(name: String) = name.equals(this.name,  
 ignoreCase = true)  
  
fun main() {  
 val f = Person("John", 25)::hasName  
 println(f("JOHN")) // true  
 println(f("JAKE")) // false  
}

</코드>

클래스 멤버와 확장의 시그니처가 같다면 어떻게 될까? 다음 코드를 살펴보라.

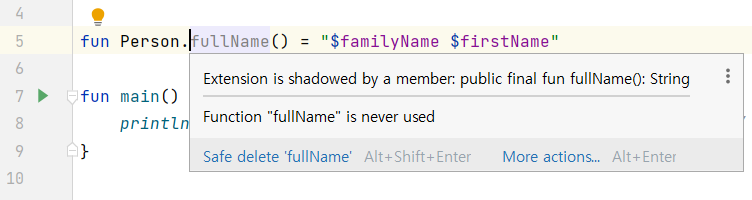
<코드>

class Person(valfirstName: String, valfamilyName: String) {  
 fun fullName() = "$firstName $familyName"  
}  
  
fun Person.fullName() = "$familyName $firstName"  
  
fun main() {  
 println(Person("John", "Doe").fullName()) // ???  
}

</코드>

이 경우 Person 클래스에 대해 정의된 fullName()함수가 둘 있다. 각각은 familyName을 첫 번째에 놓느냐 두 번째에 놓느냐에 따라 다르다. 이런 식으로 호출 지점에서 어떤 함수를 호출할지 모호할 때, 컴파일러는 항상 멤버 함수를 우선적으로 선택한다. 따라서 앞의 예제는 John Doe를 출력한다. 그리고 fullName()이라는 확장 함수가 Person클래스의 멤버에 의해 가려졌다(shadowing)는 경고가 발생한다. IDE도 적절히 코드를 강조해준다(그림 5-6을 보라).

그림 5-6: "확장이 가려짐" 경고



확장보다 멤버를 우선 선택함으로써 기존 클래스의 동작을 사고로 변경하지 못하게 방지해준다. 확장을 우선시하면 클래스 동작이 확장에 따라 변경되면서 찾기 어려운 오류가 발생할 수 있다. 예를 들어 다음과 같이 정의한 경우,

<코드>

package bad  
  
fun Person.fullName() = "$familyName $firstName"

</코드>

Person("John", "Doe").fullName() 호출의 의미는 다음 문장을 파일 안에 포함했느냐에 따라 달라져 버린다.

<코드>

import bad.fullName

</코드>

이런 우선순위는 내장된 JDK 클래스의 멤버를 보호하기 위한 것이기도 하다.

확장이 가려짐으로 인한 단점도 있다. 확장 함수를 먼저 정의하고, 확장과 똑같은 멤버를 클래스 안에 나중에 추가하면, 확장을 사용하던 모든 호출은 의미가 달라진다. 하지만 클래스의 멤버가 클래스에 제공하는 주된 API를 구성하고 확장 함수보다는 드물게 변경된다는 점에서 이런 단점을 받아들일 만하다. 그리고 이런 우선순위는 확장이 전혀 없는 자바 코드와의 상호 운용성을 단순화해준다.

지역 확장 함수를 정의할 수도 있다. 예를 들어 다른 확장 함수 안에 확장 함수를 내포시킬 수도 있다. 이런 경우 this 식은 가장 안쪽에 있는 함수의 수신 객체를 뜻한다. 바깥쪽 함수의 수신 객체를 참조하고 싶다면 한정시킨 this를 사용해 함수 이름을 명시해야 한다. 이런 성질은 확장 함수 본문에 정의된 지역 클래스나 익명 객체 안에서도 똑같다.

<코드>

private fun String.truncator(max: Int) = object {  
 val truncated  
 get() = if (length <= max) this@truncator else substring(0, max)  
   
 val original  
 get() = this@truncator  
}  
  
fun main() {  
 val truncator = "Hello".truncator(3)  
   
 println(truncator.original) // Hello  
 println(truncator.truncated) // Hel  
}

</코드>

문법은 내부 클래스에 대해 설명할 때 살펴본 한정시킨 this와 같다.

다른 패키지에 최상위 확장 함수가 정의된 경우, 확장 함수를 호출하기 전에 반드시 확장을 임포트해야만 한다. 예를 들면 다음과 같다.

<코드>

// util.kt  
package util  
  
fun String.truncate(maxLength: Int): String {  
 return if (length <= maxLength) this else substring(0, maxLength)  
}  
  
// main.kt  
package main  
  
import util.truncate  
  
fun main() {  
 println("Hello".truncate(3))  
}

</코드>

확장 함수를 전체 이름으로 호출할 수 없는 이유는 전체 이름에서 패키지 이름과 클래스 이름이 차지할 부분에 수신 객체 식이 오기 때문이다. 하지만 다음과 같이 확장 함수가 아닌 함수는

<코드>

fun truncate(s: String maxLength: Int): String {  
 return if (s.length<= maxLength) s else s.substring(0, maxLength)  
}

</코드>

임포트를 하지 않아도 util.truncate(“Hello”, 3)처럼 전체 이름으로 호출할 수 있다.

**자바와 코틀린의 차이**: JVM에서 확장 함수를 어떻게 표현하는지 궁금한 독자도 있을 것이다. 해답은 상당히 단순하다. 확장 함수는 수신 객체를 가리키는 파라미터가 추가된 정적 메서드로 컴파일된다. 앞에서 본 truncate() 함수를 컴파일한 바이트코드를 살펴보면 다음 자바 코드와 비슷한 내용임을 알 수 있다.

<코드>

public final class UtilKt {  
 public static String truncate(String s, int maxLength) {  
 return s.length() <= maxLength  
 ? s  
 : s.substring(0, maxLength)  
 }  
}

</코드>

이는 다음과 같은 비확장 코틀린 (최상위나 로컬) 함수에 해당한다.

<코드>

fun truncate(s: String, maxLength: Int) =  
 if (s.length<= maxLength) s else s.substring(0, maxLength)

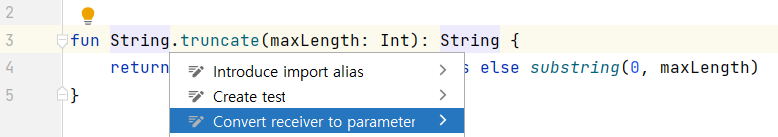
</코드>

즉, 확장 함수는 근본적으로 (특정 클래스 타입의 객체를 첫 번째 인자로 받는[[2]](#footnote-2)) **일반 함수를 마치 클래스 멤버인 것처럼 쓸 수 있게 해주는 편의 문법(syntatic sugar)일 뿐이다**.

<참고> IDE 팁

인텔리J 코틀린 플러그인은 확장 함수의 수신 객체를 명시적인 파라미터로 변환해서 자동으로 확장 함수를 비확장 함수로 바꿔주는 액션을 제공한다. 이런 변환을 하려면 케럿을 수신 객체 타입에 놓고 [Alt]+[Enter] 메뉴에서 Convert receiver to parameter를 선택하면 된다(그림 5-7). 그리고 반대 방향의 액션인 Convert parameter to receiver도 있다. 이 경우에는 캐럿을 해당 파라미터 타입에 놓고 [Alt]+[Enter]를 누르면 이 기능을 사용할 수 있다.

그림 5-7: "'Convert receiver to parameter'" 액션



멤버 함수나 프로퍼티와 달리 널이 될 수 있는 타입에 대해서도 확장을 정의할 수 있다는 점을 알아두면 좋다. 널이 될 수 있는 타입은 자체 멤버를 포함하지 않기 때문에 확장 함수를 바깥에서 정의함으로써 기능을 풍부하게 할 수 있다. 안전한 호출 연산자를 허용하지 않고도 이런 확장을 호출할 수 있다.

<코드>

// 널이 될 수 있는 수신 객체 타입  
fun String?.truncate(maxLength: Int): String {  
 if (this == null) return null  
 return if (length <= maxLength) this else substring(0, maxLength)  
}  
  
fun main() {  
 val s = readLine() // 널이 될 수 있는 문자열  
 println(s.truncate(3)) // 여기서 `?.`를 쓰지 않아도 된다.  
}

</코드>

확장 함수의 수신 객체가 널이 될 수 있는 타입인 경우, 널 값을 처리하는 책임은 확장 함수 쪽에 있다.

### 확장 프로퍼티

코틀린에서는 확장 함수와 비슷하게 확장 프로퍼티를 정의할 수도 있다. 다른 일반 멤버 프로퍼티와 마찬가지 방법으로도 확장 프로퍼티에 접근할 수 있다. 문법도 비슷하다. 확장 프로퍼티를 정의하려면 프로퍼티 이름 앞에 수신 객체의 타입을 지정하면 된다. 다음 예제를 살펴보자.

<코드>

val IntRange.leftHalf: IntRange  
 get() = start..(start + endInclusive)/2  
  
fun main() {  
 println((1..3).leftHalf) // 1..2  
 println((3..6).leftHalf) // 3..4  
}

</코드>

이 코드는 IntRange에 대해 leftHalf라는 확장 프로퍼티를 정의한다. 이 확장 프로퍼티는 원래 범위의 왼쪽 절반을 계산한다.

멤버와 확장 프로퍼티의 결정적인 차이는 어떤 클래스의 인스턴스에 안정적으로 상태를 추가할 방법이 없기 때문에 확장 프로퍼티는 뒷받침하는 필드를 쓸 수 없다는 점이다. 이 말은 확장 프로퍼티를 초기화할 수도 없고, 접근자 안에서 field를 사용할 수도 없다는 뜻이다. 그리고 lateinit로 확장 프로퍼티를 정의할 수도 없다. lateinit은 뒷받침하는 필드에 의존하는 기능이기 때문이다. 마찬가지 이유로 확장 프로퍼티 정의에서는 항상 명시적인 게터를 정의해야 하고, 가변 프로퍼티인 경우에는 세터도 명시해야만 한다.

<코드>

val IntArray.midIndex  
 get() = lastIndex/2  
   
var IntArray.midValue  
 get() = this[midIndex]  
 set(value) {  
 this[midIndex] = value  
 }  
  
fun main() {  
 val numbers = IntArray(6) { it\*it } // 0, 1, 4, 9, 16, 25  
   
 println(numbers.midValue) // 4  
 numbers.midValue \*= 10  
 println(numbers.midValue) // 40  
}

</코드>

하지만 확장 프로퍼티에서도 위임을 사용할 수 있다. 다만 위임식이 프로퍼티의 수신 객체에 접근할 수 없다는 점에 유의하라. 따라서 일반적으로 다음과 같이 리시버 타입이 같으면 모두 똑같은 인스턴스를 얻기 때문에 lazy 프로퍼티를 확장으로 선언해도 사용상 그다지 이점이 없다.

<코드>

val String.message by lazy { “Hello” }  
  
fun main() {  
 println(“Hello”.message) // Hello  
 println(“Bye”.message) // Hello  
}

</코드>

인스턴스가 단 하나만 있기 때문에 객체 정의는 예외다.

<코드>

object Messages  
  
val Messages.HELLO by lazy { “Hello” }  
  
fun main() {  
 p rintln(Messages.HELLO)  
}

</코드>

일반적으로 프로퍼티의 수신 객체에 접근할 수 있는 위임을 만들 수 있다. 이에 대해서는 **11장 도메인 특화 언어**에서 다룬다.

## 동반 확장

**4장 클래스와 객체 다루기**에서 동반 객체에 대해 설명했다. 어떤 클래스의 동반 객체는 이 클래스에 내포된 객체 중에서 바깥 클래스의 이름을 통해 객체 멤버에 접근할 수 있는 특별한 객체다. 이런 유용한 성질이 확장의 경우에도 성립한다.

다음 예제에서는 IntRange 클래스의 동반 객체에 대한 확장 함수를 정의한다. 이 함수에 접근할 때는 클래스의 이름을 사용한다.

<코드>

fun IntRange.Companion.singletonRange(n: Int) = n..n  
  
fun main() {  
 println(IntRange.singletonRange(5)) // 5..5  
 println(IntRange.Companion.singletonRange(3)) // 3..3  
}

</코드>

물론 IntRange.Companion.singletonRange(3)처럼 완전한 동반 객체 이름을 사용해 이 함수를 호출할 수도 있다. 확장 프로퍼티도 마찬가지다.

<코드>

val String.Companion.HELLO  
 get() = "Hello"  
  
fun main() {  
 println(String.HELLO)  
 println(String.Companion.HELLO)  
}

</코드>

동반 객체가 비어있지 않은 경우에만 동반 객체에 대한 확장을 정의할 수 있다.

<코드>

class Person(valfirstName: String, valfamilyName: String) {  
 companion object  
}  
  
valPerson.Companion.UNKNOWN by lazy { Person(“John”, “Doe”) }

</코드>

반대로 Any에는 동반 객체가 존재하지 않기 때문에 Any의 동반 객체에 대한 확장을 정의할 수는 없다.

<코드>

// Error: Companion is undefined  
fun Any.Companion.sayHello() = println("Hello")

</코드>

### 람다와 수신 객체 지정 함수 타입

함수나 프로퍼티와 마찬가지로 코틀린에서는 람다나 익명 함수에 대해서도 확장 수신 객체를 활용할 수 있다. 이런 함숫값들은 수신 객체 지정 함수 타입(functional type with receiver)라는 특별한 타입으로 표현된다. aggregate() 예제에서 함수 타입 파라미터가 인자를 두 개 받는 대신 수신 객체를 받도록 다시 정의해보자.

<코드>

fun aggregate(numbers: IntArray, op: Int.(Int) -> Int): Int {  
 var result = numbers.firstOrNull()  
 ?: throw IllegalArgumentException("Empty array")  
   
 for (i in 1..numbers.lastIndex) result = result.op(numbers[i])  
   
 return result  
}  
  
fun sum(numbers: IntArray) = aggregate(numbers) { op -> this + op }

</코드>

파라미터 타입 목록 앞에 타입 이름과 점을 추가해서 수신 객체의 타입을 정의한다.

<코드>

Int.(Int) -> Int

</코드>

이렇게 정의한 경우 이 함수에 전달된 람다는 암시적으로 수신 객체를 가지게 된다. this를 사용해 이 객체에 접근할 수 있다.

<코드>

{ op -> this + op }

</코드>

마찬가지로 익명 함수에 대해서도 확장 함수 문법을 사용할 수 있다. 수산 객체 타입을 함수의 파라미터 목록 앞에 추가하면 된다.

<코드>

fun sum(numbers: IntArray) = aggregate(numbers, fun Int.(op: Int) = this + op)

</코드>

확장 함수 정의와 달리, 수신 객체가 있는 함숫값을 호출할 때는 수신 객체를 다른 모든 파라미터 앞에 첫 번째 파라미터로 넣어서 확장 함수가 아닌 일반 함수 형태로 호출할 수도 있다. 예를 들어 다음과 같이 작성할 수도 있다.

<코드>

fun aggregate(numbers: IntArray, op: Int.(Int) -> Int): Int {  
 var result = numbers.firstOrNull()  
 ?: throw IllegalArgumentException("Empty array")  
   
 for (i in 1..numbers.lastIndex) {  
 result = op(result, numbers[i]) // 비 확장 함수 호출  
 }  
   
 return result  
}

</코드>

기본적으로 함숫값이 리터럴이 아닌 경우에는 자유롭게 수신 객체가 있는 함수 타입의 값과 수신 객체가 첫 번째 파라미터인 일반 함수 타입의 값의 역할을 할 수 있다. 이렇게 쓸 수 있는 이유는 이 두 타입의 값이 런타임에는 똑같이 표현되기 때문이다.

<코드>

val min1: Int.(Int) -> Int = { if (this < it) this else it }  
val min2: (Int, Int) -> Int = min1  
val min3: Int.(Int) -> Int = min2

</코드>

수신 객체가 있는 함숫값을 확장 형태나 (수신 객체가 첫 번째 인자인) 비 확장 형태 중 어느 쪽으로나 호출할 수 있지만, 수신 객체가 없는 함숫값은 비 확장 형태로만 호출할 수 있다.

<코드>

fun main() {  
 val min1: Int.(Int) -> Int = { if (this < it) this else it }  
 val min2: (Int, Int) -> Int = min1  
 println(3.min1(2)) // Ok: calling min1 as extension  
 println(min1(1, 2)) // Ok: calling min1 as non-extension  
 println(3.min2(2)) // Error: Can’t call min2 as extension  
 println(min2(1, 2)) // Ok: Calling min2 as non-extension  
}

</코드>

수신 객체가 있는 람다는 DSL과 같은 API를 구축할 때 강력한 도구를 제공한다. 이 주제에 대해서는 **11장 도메인 특화 언어**에서 다룬다.

### 수신 객체가 있는 호출 가능한 참조

코틀린에서는 수신 객체가 있는 함숫값을 정의하는 호출 가능 참조를 만들 수 있다. 클래스 멤버를 바탕으로 이런 참조를 만들 수도 있고, 확장 선언을 바탕으로 이런 참조를 만들 수도 있다. 문법적으로 이들은 바인딩된 호출 가능 참조와 비슷하지만, 수신 객체를 계산하는 식 대신 수신 객체 타입이 앞에 붙는다는 점이 다르다.

<코드>

fun aggregate(numbers: IntArray, op: Int.(Int) -> Int): Int {  
 var result = numbers.firstOrNull()  
 ?: throw IllegalArgumentException("Empty array")  
   
 for (i in 1..numbers.lastIndex) result = result.op(numbers[i])  
   
 return result  
}  
  
fun Int.max(other: Int) = if (this > other) this else other  
  
fun main() {  
 val numbers = intArrayOf(1, 2, 3, 4)  
 println(aggregate(numbers, Int::plus)) // 10  
 println(aggregate(numbers, Int::max)) // 4  
}

</코드>

이 예제에서 Int::plus는 Int 내장 클래스의 plus() 멤버 함수(plus 멤버 함수는 +연산자와 같다)를 사용한 반면, Int::max는 같은 파일에 정의된 확장 함수를 사용해 정의됐다. 두 경우 모두 똑같은 구문을 사용한다.

앞 절에서 설명한 확장과 비확장 함수 타입 사이의 암시적 변환으로 인해, 수신 객체가 없는 호출 가능 참조를 수신 객체가 필요한 함수 타입의 값 대신 사용할 수도 있다.

예를 들어 Int.(Int) -> Int 타입의 파라미터를 받는 고차 함수에게 인자가 2개인 호출 가능 참조인 ::max를 전달해도 된다.

<코드>

fun aggregate(numbers: IntArray, op: Int.(Int) -> Int): Int {  
 var result = numbers.firstOrNull()  
 ?: throw IllegalArgumentException("Empty array")  
   
 for (i in 1..numbers.lastIndex) result = result.op(numbers[i])  
   
 return result  
}  
  
fun max(a: Int, b: Int) = if (a > b) a else b  
  
fun main() {  
 println(aggregate(intArrayOf(1, 2, 3, 4), ::max))  
}

</코드>

역방향의 대입도 참이다. 수신 객체가 지정된 호출 가능 참조를 수신 객체가 아닌 일반 함수 타입의 파라미터를 받는 함수에게 전달할 수도 있다. 다음 예제에서는 멤버에 대한 호출 가능 참조와 확장 함수에 대한 호출 가능 참조를 (Int, Int) -> Int 함수 타입의 파라미터 위치에 인자로 전달한다.

<코드>

fun aggregate(numbers: IntArray, op: (Int, Int) -> Int): Int {  
 var result = numbers.firstOrNull()  
 ?: throw IllegalArgumentException("Empty array")  
   
 for (i in 1..numbers.lastIndex) result = op(result, numbers[i])  
   
 return result  
}  
  
fun Int.max(other: Int) = if (this > other) this else other  
  
fun main() {  
 println(aggregate(intArrayOf(1, 2, 3, 4), Int::plus)) // 10  
 println(aggregate(intArrayOf(1, 2, 3, 4), Int::max)) // 4  
}

</코드>

클래스 멤버인 동시에 확장 함수로 정의된 함수에 대한 호출 가능 참조를 지원하지는 않는다. 이런 함수의 경우 타입::함수이름 형태로는 여러 수신 객체 타입을 지정할 방법이 없기 때문이다.

## 영역 함수

코틀린 표준 라이브러리에는 어쩐 식을 계산한 값을 문맥 내부에서 임시로 사용할 수 있도록 해주는 함수가 몇 가지 들어있다. 때로는 (식을 계산한 결과를 담을) 지역 변수를 명시적으로 선언하지 않고, 식의 값이 들어 있는 암시적인 영역을 정의해서 코드를 단순화할 수 있는 경우가 있다. 이런 함수를 보통 영역 함수(scope function)라고 부른다.

영역 함수가 기본적으로 하는 일은 여러분이 인자로 제공한 람다를 간단하게 실행해주는 것이다. 그 차이는 다음과 같은 몇 가지 관점의 조합에 있다.

* 문맥 식을 계산한 값을 영역 함수로 전달할 때, 수신 객체로 전달하는지(영역 함수가 확장 함수인 경우) 일반적인 함수 인자로 전달하는지(영역 함수가 일반 함수인 경우)
* 영역 함수의 람다 파라미터가 수신 객체 지정 람다(확장 람다)인지 아닌지
* 영역 함수가 반환하는 값이 람다의 결괏값인지 컨텍스트 식을 계산한 값인지

전체적으로는 run, let, with, apply, also라는 5가지 표준 영역 함수가 있다. 이 절에서는 이런 함수를 사용해 코드를 단순화하는 방법을 배운다. 모든 영역 함수는 인라인 함수이기 때문에 런타임 부가비용이 없다.

영역 함수를 조심해 사용해야 하며, 남용하면 오히려 코드 가독성이 나빠지고 실수를 하기도 쉬워진다는 점을 명심하라. 일반적으로 여러 영역 함수를 내포시키면 this나 it이 어떤 대상을 가리키는지 구분하기 어려워지기 때문에 영역 함수를 여러 겹으로 내포시켜 사용하지 않는 편을 권장한다.

### run과 with 함수

run() 함수는 확장 람다를 받는 확장 함수로 람다의 결과를 돌려준다. 기본적인 사용 패턴은 객체 상태를 설정한 다음, 이 객체를 대상으로 어떤 결과를 만들어내는 람다를 호출하는 것이다.

<코드>

class Address {  
 var zipCode: Int = 0  
 var city: String = “”  
 var street: String = “”  
 var house: String = “”  
   
 fun post(message: String): Boolean {  
 “Message for ${zipCode, $city, $street, $house}: $message”  
 return readLine() == “OK”  
 }  
}  
  
fun main() {  
 val isReceived = Address().run {  
 // Address 인스턴스를 this로 사용할 수 있다  
 zipCode = 123456  
 city = “London”  
 street = “Baker Street”  
 house = ”221b”  
 post(“Hello!”) // 반환값  
 }  
   
 if (!isReceived) {  
 println(“Message is not delivered”)  
 }  
}

</코드>

run 함수가 없으면 Address 인스턴스를 담을 변수를 추가해야 한다. 이로 인해 함수 본문의 나머지 부분에서도 이 변수에 접근할 수 있게 된다. 하지만 이 Address 인스턴스를 post()를 호출할 때 한번만 써야 한다면 함수의 나머지 부분에서 이 인스턴스에 마음대로 접근할 수 있는 것은 바람직하지 않다. run()과 같은 함수를 사용하면 지역 변수의 가시성을 좀 더 세밀하게 제어할 수 있다.

결과의 타입이 Unit일 수도 있다는 점에 유의하라.

<코드>

fun Address.showCityAddress() = println(“$street, $house”)  
  
fun main() {  
 Address().run {  
 zipCode = 123456  
 city = “London”  
 street = “Baker Street”  
 house = ”221b”  
 showCityAddress()  
 }  
}

</코드>

with() 함수는 run()과 상당히 비슷하다. 유일한 차이는 with()가 확장 함수 타입이 아니어서 문맥 식을 with의 첫 번째 인자로 전달해야 한다는 점뿐이다. 이 함수를 일반적으로 사용하는 경우는 문맥 식의 멤버 함수와 프로퍼티에 대한 호출을 묶어서 동일한 영역 내에서 실행하는 경우다.

<코드>

fun main() {  
 val message = with (Address(“London”, “Baker Street”, “221b”)) {  
 “Address: $city, $street, $house”  
 }  
 println(message)  
}

</코드>

앞의 예제에서는 this 인스턴스의 멤버를 전체 이름을 대신 간단한 이름만으로 사용할 수 있음을 보여줬다. 영역 함수를 사용하지 않으면 다음과 같이 코드를 작성해야 할 것이다.

<코드>

fun main() {  
 val addr = Address(“London”, “Baker Street”, “221b”)  
 val message = “Address: ${addr.city}, ${addr.street}, ${addr.house}”  
 println(message)  
}

</코드>

즉, 영역 함수가 없으면 변수를 추가로 도입하고 Address의 모든 멤버를 호출할 때마다 추가한 변수(여기서는 addr)를 매번 명시해야 했을 것이다.

### 문맥이 없는 run

코틀린 표준 라이브러리는 run()을 오버로딩한 함수도 제공한다. 이 함수는 문맥 식이 없고 람다의 값을 반환하기만 한다. 람다 자체에는 수신 객체도 없고, 파라미터도 없다.

주로 이 함수를 사용하는 경우는 어떤 식이 필요한 부분에서 블록을 사용하는 것이다. 다음 코드를 생각해 보자.

<코드>

class Address(val city: String, val street: String, val house: String) {  
 fun asText() = “$city, $street, $house”  
}  
  
fun main() {  
 val address = Address(“London”, “Baker Street”, “221b”)  
 println(address.asText())  
}

</코드>

표준 입력에서 주소의 각 요소를 읽어오려면 어떻게 해야 할까? 아마도 읽어온 여러 값에 대해 별도로 변수를 정의해야 할 것이다.

<코드>

fun main() {  
 val city = readLine() ?: return  
 val street = readLine() ?: return  
 val house = readLine() ?: return  
 val address = Address(city, street, house)  
 println(address.asText())  
}

</코드>

하지만 모든 변수가 main()의 다른 지역 변수(예: city)와 같은 영역 안에 있기 때문에 이 코드는 바람직하지 않다. 아마도 Address 인스턴스만을 처리하기 위한 영역을 따로 만들 수 있으면 더 타당할 것이다. 모든 변수를 인라이닝하는 것은 다음과 같이 알아보기 힘든 코드가 생기기 때문에 오히려 더 나쁜 선택이다.

<코드>

fun main() {  
 val address = Address(readLine() ?: return,  
 readLine() ?: return,  
 readLine() ?: return)  
 println(address.asText())  
}

</코드>

이 코드를 보자마자 각 readLine()이 어떤 의미인지 알아보기는 쉽지 않다. 코틀린에서 이런 경우를 처리하는 전형적인 방법은 run()을 사용하는 것이다.

<코드>

fun main() {  
 val address = run {  
 val city = readLine() ?: return  
 val street = readLine() ?: return  
 val house = readLine() ?: return  
 Address(city, street, house)  
 }  
   
 println(address.asText())  
}

</코드>

run은 인라인 함수이기 때문에 람다 내부에서 바깥쪽 함수의 제어를 반환시키기 위해 return을 사용해도 된다. 이로 인해 run이 마치 다른 일반적인 제어 구조와 똑같아 보인다.

여기서 블록을 사용하면 블록을 람다로 취급하기 때문에 제대로 작동하지 않는다. 그래서 표준 라이브러리에 run()이 추가됐다.

<코드>

fun main() {  
 val address = {  
 val city = readLine() ?: return // Error: return is not allowed  
 val street = readLine() ?: return // Error: return is not allowed  
 val house = readLine() ?: return // Error: return is not allowed  
 Address(city, street, house)  
 }  
   
 println(address.asText()) // Error: no asText() method  
}

</코드>

### let

let 함수는 run과 비슷하지만 확장 함수 타입의 람다를 받지 않고 인자가 하나뿐인 함수 타입의 람다를 받는다는 점이 다르다. 따라서 문맥 식의 값은 람다의 인자로 전달된다. let의 반환 값은 람다가 반환하는 값과 같다. 외부 영역에 새로운 변수를 도입하는 일을 피하고 싶을 때 주로 이 함수를 사용한다.

<코드>

class Address(val city: String, val street: String, val house: String) {  
 fun post(message: String) {}  
}  
  
fun main() {  
 Address(“London”, “Baker Street”, “221b”).let {  
 // 이 안에서는 it 파라미터를 통해 Address 인스턴스에 접근할 수 있음  
 println(“To city: ${it.city}”)  
 it.post(“Hello”)  
 }  
}

</코드>

다른 람다와 마찬가지로 모호함을 없애고 가독성을 높이기 위해 파라미터에 원하는 이름을 부여할 수도 있다.

<코드>

fun main() {  
 Address(“London”, “Baker Street”, “221b”).let { addr ->  
 // 이 안에서는 addr 파라미터를 통해 Address 인스턴스에 접근할 수 있음  
 println(“To city: ${addr.city}”)  
 addr.post(“Hello”)  
 }  
}

</코드>

let의 일반적인 사용법 중에는 널이 될 수 있는 값을 안전성 검사를 거쳐서 널이 될 수 없는 함수에 전달하는 용법이 있다. 앞 장에서 널이 될 수 있는 수신 객체에 대해 함수를 호출할 수 있게 해주는 안전한 호출 연산자에 대해 배웠다. 하지만 처리해야 하는 값을 무조건 널이 될 수 없는 값으로만 전달해야 한다면 어떻게 널이 될 수 있는 값을 전달할 수 있을까? 다음 예제를 살펴보자.

<코드>

fun readInt() = try {  
 readLine()?.toInt()  
} catch (e: NumberFormatException) {  
 null  
}  
  
fun main(args: Array<String>) {  
 val index = readInt()  
 val arg = if (index != null) args.getOrNull(index) else null  
 if (arg != null) {  
 println(arg)  
 }  
}

</코드>

getOrNull() 함수는 주어진 인덱스가 정상이라면 배열 원소를 반환하지만 그렇지 않으면 null을 반환한다. 이 함수의 파라미터가 널이 될 수 없는 타입이기 때문에 readInt()의 결과를 getOrNull()에 직접 전달할 수는 없다. 그래서 방금 본 예제에서는 if를 통해 readInt()의 결과를 스마트 캐스트해서 널이 아닌 값을 getOrNull()에 전달했다. 하지만 let을 사용하면 이 코드를 더 단순화시킬 수 있다.

<코드>

val arg = index?.let { args.getOrNull(it) }

</코드>

let 호출은 index가 널이 아닌 경우에만 호출되기 때문에 컴파일러는 람다 안에서 it 파라미터가 널이 될 수 없는 값임을 알 수 있다.

### apply/also 함수

apply() 함수는 확장 람다를 받는 확장 함수이며 자신의 수신 객체를 반환한다. 이 함수는 일반적으로 run()과 달리 반환 값을 만들어내지 않고 객체의 상태를 설정하는 경우에 사용한다.

<코드>

class Address {  
 var city: String = “”  
 var street: String = “”  
 var house: String = “”  
   
 fun post(message: String) { }  
}  
  
fun main() {  
 val message = readLine() ?: return  
   
 Address().apply {  
 city = “London”  
 street = “Baker Street”  
 house = ”221b”  
 }.post(message)  
}

</코드>

비슷한 함수로 also()가 있다. 이 함수는 apply()와 달리 인자가 하나 있는 람다를 파라미터로 받는다.

<코드>

fun main() {  
 val message = readLine() ?: return  
   
 Address().also {  
 it.city = “London”  
 it.street = “Baker Street”  
 it.house = ”221b”  
 }.post(message)  
}

</코드>

## 클래스 멤버인 확장

앞 절에서 확장 함수를 클래스 멤버로 선언할 수 있다고 설명했다. 이런 확장에 대해 더 자세히 살펴보자.

클래스 안에서 확장 함수나 프로퍼티를 선언하면 일반적인 멤버나 최상위 확장과 달리 이런 함수나 프로퍼티에는 수신 객체가 두 개 있다. 이때 확장 정의의 수신 객체 타입의 인스턴스를 확장 수신 객체(extension receiver)라고 부르고, 확장이 포함된 클래스 타입의 인스턴스를 디스패치 수신 객체(dispatch receiver)라고 부른다. 두 수신 객체를 가리킬 때는 this앞에 클래스 이름(디스패치 수신 객체의 경우)이나 확장 이름(확장 수신 객체의 경우)을 붙여서 한정시킨다. 일반적으로 한정을 시키지 않은 this는 this를 가장 가깝게 둘러싸는 선언의 수신 객체를 참조한다. 따라서 지역적인 클래스나 수신 객체 지정 람다나 내포된 확장 함수 선언이 있지 않는 한, 일반적으로 this는 this가 속한 확장 함수의 확장 수신 객체가 된다.

두 종류의 수신 객체를 모두 보여주는 예제를 살펴보자.

<코드>

class Address(val city: String, val street: String, val house: String)  
  
class Person(val firstName: String, val familyName: String) {  
 fun Address.post(message: String) {  
 // 암시적 this: 확장 수신 객체(Address)  
 val city = city  
 // 한정시키지 않은 this: 확장 수신 객체(Address)  
 val street = this.city  
 // 한정시킨 this: 확장 수신 객체(Address)  
 val house = this@post.house  
 // 암시적 this: 디스패치 수신 객체(Person)  
 val firstName = firstName  
 // 한정시킨 this: 디스패치 수신 객체(Person)  
 val familyName = this@Person.familyName  
   
 println("From $firstName, $familyName at $city, $street, $house:")  
 println(message)  
 }  
   
 fun test(address: Address) {  
 // 디스패치 수신 객체: 암시적  
 // 확장 수신 객체: 명시적  
 address.post("Hello")  
 }  
}

</코드>

test() 안에서 post() 함수를 호출하면 test()가 Person 의 멤버이기 때문에 디스패치 수신 객체가 자동으로 제공된다. 반면 확장 수신 객체는 address 식을 통해 명시적으로 전달된다.

비슷하게 Person 클래스 인스턴스가 다른 방식으로 공급되는 경우에도 post() 함수를 호출할 수도 있다. 예를 들어 확장 수신 객체나 외부 클래스의 인스턴스로 Person 타입의 값이 공급될 수 있다.

<코드>

class Address(val city: String, val street: String, val house: String)  
  
class Person(val firstName: String, val familyName: String) {  
 fun Address.post(message: String) { }  
 inner class Mailbox {  
 fun Person.testExt(address: Address) {  
 address.post("Hello")  
 }  
 }  
}  
  
fun Person.testExt(address: Address) {  
 address.post("Hello")  
}

</코드>

Address 타입의 수신 객체가 있다면 어떻게 될까? Address 클래스의 본문 안에서 post()를 호출하고 싶다고 하자.

<코드>

class Address(val city: String, val street: String, val house: String) {  
 fun test(person: Person) {  
 person.post(“Hello”) // Error: method post() is not defined  
 }  
}  
  
class Person(val firstName: String, val familyName: String) {  
 fun Address.post(message: String) { }  
}

</코드>

Person 타입의 디스패치 수신 객체가 현재 영역에 존재해야만 post()를 호출할 수 있으므로 이 예제에서는 post() 함수를 호출할 수가 없다. 이 문제를 해결하려면 post() 호출을 Person 수신 객체를 받는 확장 람다로 감싸야 한다.

<코드>

class Address(val city: String, val street: String, val house: String) {  
 fun test(person: Person) {  
 with(person) {  
 // 암시적 디스패치와 확장 수신 객체  
 post()  
 }  
 }  
}  
  
class Person(val firstName: String, valfamilyName: String) {  
 fun Address.post(message: String) { }  
}

</코드>

이 트릭을 Address나 Person 클래스, 혹은 이들의 확장 밖에서 post()를 호출할 때도 사용할 수 있다.

<코드>

class Address(val city: String, val street: String, val house: String)  
  
class Person(val firstName: String, val familyName: String) {  
 fun Address.post(message: String) { }  
}  
  
fun main() {  
 with(Person(“John”, “Watson”)) {  
 Address(“London”, “Baker Street”, “221b”).post(“Hello”)  
 }  
}

</코드>

이 예제들은 함수와 프로퍼티의 이중 수신 객체 규칙이 큰 혼란을 야기할 수도 있음을 보여준다. 이런 이유로 수신 객체의 영역을 자신이 포함된 선언 내부로 제한하는 쪽을 더 권장한다.

<코드>

class Address(val city: String, val street: String, val house: String)  
  
class Person(val firstName: String, val familyName: String) {  
 // Person 클래스 밖에서는 쓸 수 없음  
 private fun Address.post(message: String) { }  
 fun test(address: Address) = address.post("Hello")  
}

</코드>

혼란을 야기하고 오류가 생기기 쉬워서 피해야 한다고 언급할 만한 사용 예로는, 다음 코드처럼 디스패치와 확장 수신 객체가 똑같은 타입인 경우가 있다.

<코드>

class Address(val city: String, val street: String, val house: String) {  
 fun Address.post(message: String) { }  
}

</코드>

이중 수신 객체 멤버를 사용하는 흥미로운 예로는 객체(특히 동반 객체) 안에 선언된 확장 함수가 있다. 이런 확장을 최상위로 임포트해서 최상위 확장처럼 사용할 수 있다.

<코드>

import Person.Companion.parsePerson  
  
class Person(valfirstName: String, valfamilyName: String) {  
 companion object {  
 fun String.parsePerson(): Person? {  
 val names = split(" ")  
 return if (names.size == 2) Person(names[0], names[1]) else null  
 }  
 }  
}  
  
fun main() {  
 // Person.Companion 인스턴스가 암시적으로 공급됨  
 println("John Doe".parsePerson()?.firstName) // John  
}

</코드>

하지만 대부분의 경우 최상위 확장이 더 간단하고 읽기 좋은 코드를 만들어내므로, 최상위 확장을 우선 사용하는 편이 더 낫다.

## 결론

이 장에서 배운 내용을 정리하자. 함수 타입과 고차 함수를 사용해 코드 조각을 함수 형태로 추상화하고 합성하는 방법을 배웠다. 그리고 함숫값을 만드는 여러 가지 방법과 함수 인라이닝의 가능성을 배웠다. 마지막으로 기존 타입에 새로운 기능을 추가할 수 있는 확장 함수와 프로퍼티를 어디에 사용하는지 살펴봤다.

다음 장에서는 객체지향 프로그래밍으로 돌아가서 열거형(enumeration)이나 데이터 클래스 등의 일반적인 프로그래밍 패턴을 단순화시켜주는 특별한 클래스에 대해 논의한다.

## 문제

1. 고차 함수란 무엇인가?
2. 람다 식의 문법을 설명하라. 자바 람다와 코틀린 람다를 비교하라.
3. 함수 타입이란 무엇인가? 코틀린의 함수 타입과 자바의 함수형 인터페이스의 다른 점은 무엇인가?
4. 수신 객체가 있는 함수 타입과 수신 객체가 없는 함수 타입을 비교해 설명하라.
5. 람다와 익명 함수의 차이는 무엇인가? 람다보다 익명 함수를 쓰면 더 좋은 때는 언제인가?
6. 인라인 함수의 장단점을 설명하라. 인라인 함수에는 어떤 제약이 있는가?
7. 호출 가능 참조란 무엇인가? 호출 가능 참조의 여러 가지 형태에 대해 설명하라. 각각의 형태를 자바의 메서드 참조와 비교하라.
8. 람다와 익명 함수 안에서 return이 어떻게 작동하는지 설명하라. 한정된 return문이란 무엇인가?
9. 함수 타입의 파라미터의 디폴트 인라이닝 모드와 noinline, crossinline 모드를 서로 비교하라.
10. 확장 함수를 정의하는 방법은 무엇인가? 확장 함수는 확장이 적용된 함수를 변경하는가?
11. 동반 객체 확장을 사용하는 방법에 대해 설명하라.
12. 확장 프로퍼티에는 어떤 제약이 있는가?
13. 여러 가지 형태의 this 식에 대해 설명하라. 한정된 this를 언제 어떻게 사용하는가?
14. 클래스 내부에서 확장 함수를 선언하는 경우 주의해야 할 점에 대해 설명하라.
15. 영역 함수란 무엇인가? 어떤 일을 처리할 때 어떤 영역 함수를 사용해야 할지 판단하는 기준이 무엇인지 설명하라.

1. 옮긴이 - 영어 function value나 functional value는 함수가 반환하는 값(return value나 result)이 아니라 함수인 값을 뜻한다. 이 책에서는 이런 함수 역할을 하는 값을 함숫값이라고 번역했다. 함숫값은 실제 런타임에 힙 메모리상에 존재하는 함수처럼 호출할 수 있는 객체란 점에서 일반 함수와 약간 다르다. 다만 함숫값이나 일반 함수나 호출할 수 있고 호출 시 결과를 돌려준다는 점은 동일하다. [↑](#footnote-ref-1)
2. 옮긴이 - C++이나 파이썬을 보면 알 수 있지만(JVM은 invokevirtual등 객체지향에 특화된 메서드 호출 명령어가 있어서 약간 다르다. 하지만 JVM 스펙의 invokevitual 정의(https://docs.oracle.com/javase/specs/jvms/se14/html/jvms-6.html#jvms-6.5.invokevirtual)를 살펴보면 스택의 맨 위에 인스턴스에 대한 참조가 있다는 점에서 실제로 동작 측면에서는 이들 언어와 비슷하다고도 할 수 있다), 일반적으로 객체의 메서드도 실제로는 명시적으로 컴파일러가 수신 객체를 첫 번째 인자로 전달해주는 일반 함수에 지나지 않는다. C로 객체지향을 구현하는 경우 이런 식으로 메서드 맨 앞에 객체에 대한 포인터(또는 참조)를 두는 방식으로 메서드를 구현하는 경우가 대부분이다. 다만 상속 등을 고려해 인스턴스를 동적으로 디스패치하려면 가상 함수 테이블 등을 거쳐 간접적으로 런타임에 실제 호출할 함수를 결정해야 하지만, 확장 함수 같은 경우는 클래스 타입을 기준으로 항상 정적으로 함수가 결정되기 때문에 자바에서는 static 메서드로 컴파일된다. [↑](#footnote-ref-2)