# 3장. 함수 정의하기

3장의 중심 주제는 함수(function)라는 개념이다. 함수의 기본적인 내부 구조를 살펴보고 중요한 요소, 즉 이름 붙은 인자(named argument), 디폴트(default) 값, vararg 스타일 함수 등을 살펴본다. 그리고 코틀린 언어의 명령형(imperative) 제어 구조도 소개할 것이다. 이 장에서는 if와 when을 사용해 두 가지 중 하나를 선택하거나 여러 가지 중 하나를 선택하는 방법을 보고, 여러 가지 형태의 반복문과 오류 처리 방법을 살펴본다. 이런 제어 구문이 자바(실제로는 명령형 패러다임을 사용하는 대부분의 언어)에서 사용하는 구문과 상당히 비슷하다는 점을 살펴보고, 자바 경험이 있는 개발자가 쉽게 코틀린에 적용할 수 있도록 몇 가지 중요한 차이점을 짚어줄 것이다. 추가로 관련 있는 여러 선언을 패키지(package)로 묶는 방법과 임포트 디렉티브(import directive)를 사용하는 패키지간 참조라는 흥미로운 주제에 대해 살펴보겠다.

## 구조

* 코틀린 함수의 구조: 정의와 호출 문법, 함수 파라미터에 대한 상세한 내용
* 제어 구조: 조건문, 루프, 오류 처리
* 코틀린 패키지 구조와 임포트

## 목표

조건문, 반복문, 오류 처리 제어 구조를 사용하는 코틀린 명령형 프로그래밍의 기초를 숙지하는 것이다. 그리고 함수와 패키지를 통해 코드를 조직화하는 수단을 배우는 것이다.

## 함수

자바 메서드처럼 코틀린 함수도 어떤 입력(파라미터(parameter)라고 부른다)을 받아서 자신을 호출한 코드 쪽에 출력값을 반환(return)할 수 있는, 재사용 가능한 코드 블록이다. 이 절에서는 함수의 정의와 내부를 살펴보자.

### 코틀린 함수의 구조

간단한 예부터 시작해보자. 반지름이 주어졌을 때 원의 넓이를 계산하는 함수를 정의해보겠다.

<코드>

import kotlin.math.PI  
  
fun circleArea(radius: Double): Double {  
 return PI\*radius\*radius  
}  
  
fun main() {  
 print("Enter radius: ")  
 val radius = readLine()!!.toDouble()  
 println("Circle area: ${circleArea(radius)}")  
}

</코드>

여기서 표준 상수 PI를 썼다. 이 값은 π의 근삿값을 표현한다. import 디렉티브를 사용해야 kotlin.math.PI라는 전체 이름이 아니라 PI라는 간단한 이름으로 이 값을 사용할 수 있다.

이제 circleArea 함수를 구성하는 요소를 자세히 살펴보자.

* (함수라는 뜻의 영어 단어 function에서 온) fun 키워드는 컴파일러에게 함수 정의가 뒤따라 온다는 사실을 알려준다.
* 변수 이름과 마찬가지로 아무 식별자나 함수 이름으로 쓸 수 있다(여기서는 circleArea를 썼다).
* 다음에는 괄호로 둘러싸여 있는 콤마(,)로 분리한 파라미터 목록이 온다. 파라미터 목록은 함수를 호출할 때 넘길 수 있는 데이터를 컴파일러에게 알려준다.
* 반환 타입(여기서는 Double)은 함수를 호출한 쪽에 돌려줄 반환값의 타입이다.
* 함수 본문(body)은 {}로 감싸있는 블록이며, 함수의 구현을 기술한다.

함수가 파라미터를 받지 않는다고 해도 파라미터를 감싸는 괄호는 꼭 있어야 한다는 점에 유의해야 한다. 예를 들면 다음과 같다.

<코드>

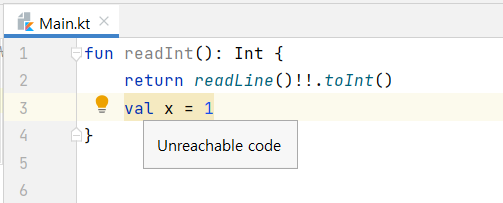
fun readInt(): Int {  
 return readLine()!!.toInt()  
}  
  
fun main() {  
 println(readInt())  
}

</코드>

자바와 마찬가지로 함수의 결괏값을 return문으로 지정한다. return문은 함수 실행을 끝내고 호출한 쪽에 제어를 돌려준다. return 문 다음에 위치한 코드는 실질적으로 죽어있는 코드이며, 결코 실행되지 않는다.

**자바와 코틀린의 차이**: 자바와 달리 코틀린에서는 도달할 수 없는 코드(unreachable code)가 오류가 아니다. 하지만 컴파일러는 경고를 표시하고 IDE는 해당 부분을 강조해주기 때문에 그림 3-1처럼 코드 중 일부가 죽은 코드임을 쉽게 알 수 있다.

그림 3-1: 도달할 수 없는 코드를 강조한 IDE



자바처럼 코틀린에도 블록문(block statement)이 있다. 블록문은 기본적으로 {}로 감싼 문의 그룹을 말한다. 문은 새줄문자로 분리하거나(이런 식으로 분리하는 쪽을 권장함) 세미콜론으로 분리하며, 순서대로 실행된다.

함수 본문을 작성할 때 우리는 이미 블록을 사용했다. 문법적으로 문장이 하나 들어갈 수 있는 위치에 여러 문장을 넣고 싶을 때면 언제든 블록을 사용할 수 있다. 예를 들어 루프의 몸통이나 조건문의 각 경우를 나타내는 문장 위치에 블록을 쓸 수 있다. 예를 들어 루프에서는 몸통이 단 한 문장으로 구성된 경우라도 {}로 둘러싸서 블록으로 코드를 작성하는 경우가 많다.

블록 안에는 지역 변수나 지역 함수 정의가 들어갈 수 있다. 이런 선언의 영역(scope)은 해당 함수/변수가 선언된 블록 내부로 제한된다.

파라미터 정의는 암시적으로 함수가 호출될 때 자동으로 인자 값으로 초기화되는 지역 변수로 취급된다.

**자바와 코틀린의 차이**: 자바 메서드 파라미터는 디폴트가 가변이라 함수 내부에서 변경하지 못하게 하려면 final을 지정해 불변값으로 바꿔야 하는데, 코틀린 함수 파라미터는 무조건 불변이다. 즉, 함수 본문에서 파라미터 값을 변경하면 컴파일 오류가 난다.

<코드>

fun increment(n: Int): Int {  
 return n++ // Error: can't change immutable variable  
}

</코드>

그리고 파라미터 앞에 val이나 var를 표시할 수 없다는 점에 유의하라. 이렇게 강제하는 이유는 파라미터에 대입하는 중에 실수를 저지르기 쉽고, 파라미터를 불변값으로 강제하는 편이 더 깔끔하고 이해하기 편한 코드를 만들어내기 때문이다.

코틀린은 값에 의한 호출(call-by-value) 의미론을 사용한다. 이 말은 파라미터 값에 호출하는 쪽의 인자를 복사한다는 뜻이다. 특히 호출 인자로 전달한 변수를 변경해도(예: 앞에서 본 원 넓이 예제의 main 함수 안에 있는 radius) 호출된 함수 내부의 파라미터 값에는 영향이 없다는 뜻이다. 하지만 파라미터가 참조(예: 배열 타입)라면 호출한 쪽의 데이터는 그대로 남아있고 이 데이터에 대한 참조만 복사된다. 따라서 파라미터 자체는 함수 내부에서 바뀔 수 없지만, 일반적으로 파라미터가 가리키는 데이터는 바뀔 수 있다. 예를 들어 다음 함수는 정수 배열을 가리키는 (불변) 참조를 받아서 첫 번째 원소를 변경하며, 그에 따라 호출하는 쪽의 데이터도 영향을 받는다.

<코드>

fun increment(a: IntArray): Int {  
 return ++a[0]  
}  
  
fun main() {  
 val a = intArrayOf(1, 2, 3)  
 println(increment(a)) // 2  
 println(a.contentToString()) // [2, 2, 3]  
}

</코드>

타입 지정을 생략해도 되는 변수와 달리 파라미터에는 항상 타입을 지정해야 한다. 컴파일러는 함수 정의에서 파라미터 타입을 추론하지 못한다[[1]](#footnote-1).

반면 반환 타입은 함수 파라미터에서 추론이 가능한 데도 여전히 명시해야 한다. 이런 결정은 함수에서 결괏값을 결정해 외부로 나가는 지점(즉, return으로 값을 반환하는 지점)이 여러 곳일 수 있는데, 함수 본문의 모든 반환 지점을 살펴보고 반환 타입을 알아내기 어려울 수 있기 때문이다. 이런 면에서 함수 정의에 있는 반환 타입은 일종의 문서화 역할을 하며, 함수 정의의 첫 줄만 보고도 함수가 만들어내는 값이 무엇인지 바로 알 수 있게 해준다.

하지만 경우에 따라 반환 타입을 생략할 수 있는데, 이런 예외적인 경우가 두 가지 있다. 첫 번째는 유닛(unit) 타입을 반환하는 경우다. 유닛은 자바 void에 해당하는 코틀린 타입으로, 함수가 의미 있는 반환값을 돌려주지 않는다는 뜻이다. 이런 함수가 반환하는 값은 Unit이라는 내장 타입에 속하는 Unit이라는 상수다. 함수 정의에서 반환값 타입을 지정하지 않으면 코틀린은 여러분이 Unit 함수를 정의한다고 가정한다. 즉, 다음 두 함수의 정의는 서로 동등하다.

<코드>

fun prompt(name: String) {  
 println("\*\*\*\*\* Hello, $name! \*\*\*\*\*")  
}  
  
fun prompt(name: String): Unit {  
 println("\*\*\*\*\* Hello, $name! \*\*\*\*\*")  
}

</코드>

이미 main() 예제에서 이런 예를 보았다. Unit 함수는 결과가 항상 Unit으로 동일하기 때문에 결과를 지정하는 return 문을 쓸 필요는 없다. 하지만 함수 본문의 끝에 도달하기 전에 함수 실행을 마치려면 return 문을 사용해 함수를 끝내야 한다(return Unit이라고 써도 정상적인 문장이지만 이 경우 Unit은 불필요한 중복일 뿐이다).

두 번째는 식이 본문인(expression-body) 함수다. 어떤 함수가 단일 식으로만 구현될 수 있다면 return 키워드와 블록을 만드는 중괄호({})를 생략하고 다음과 같은 형태로 함수를 작성해도 된다. 여기서 반환 타입과 함수 본문 식 사이에 =가 들어있다는 점에 유의하라.

<코드>

fun circleArea(radius: Double): Double = PI\*radius\*radius

</코드>

위와 같은 구문은 = 기호로 변수를 초기화하는 식을 지정했던 변수 정의와 비슷하다. 변수와 마찬가지로 식이 본문인 함수에서도 반환 타입을 생략할 수 있다.

<코드>

fun circleArea(radius: Double) = PI\*radius\*radius // 반환값이 Double로 추론됨

</코드>

식이 본문인 함수는 명시적인 반환 타입 정의를 쓰지 않아도 될 정도로 간단하다고 간주된다. 하지만 이 기능은 조심스럽게 사용해야 한다. 복잡하게 표현된 식이 본문인 함수는 일반적인 블록 구문을 사용해 가독성을 높여주는 편이 낫다.

블록이 본문인 함수를 정의할 때 {] 앞에 =를 넣으면 이 블록이 익명 함수를 기술하는 람다(lambda)로 해석되기 때문에 여러분이 원하는 결과를 얻을 수 없다. 예를 들어 다음 함수를 보자.

<코드>

fun circleArea(radius: Double) = { PI\*radius\*radius }

</코드>

위 정의는 미리 정해진 반지름에 해당하는 원의 넓이를 계산해주는 다른 함수를 반환하는 함수 정의에 해당하며, 올바른 정의다. 반면 비슷한 정의에서 블록 안에 return을 추가하면 컴파일 오류가 난다.

<코드>

fun circleArea(radius: Double) = {  
 return PI\*radius\*radius // error: 'return' is not allowed here  
 // error: type mismatch: inferred type is () -> [ERROR : Return not allowed] but Double was expected  
}

</코드>

이 오류는 타입 불일치로 인해 발생하며, 식이 본문인 함수 안에서 return문이 금지되기 때문이다(**5장 고급 함수와 함수형 프로그래밍 활용하기** 참고).

### 위치 기반 인자와 이름 붙은 인자

기본적으로 함수 호출 인자는 순서대로 파라미터에 전달된다. 첫 번째 인자는 첫 번째 파라미터, 두 번째 인자는 두 번째 파라미터 라는 식이다. 코틀린에서는 이런 방식의 인자 전달을 위치 기반 인자(positional argument)라고 한다.

<코드>

fun rectangleArea(width: Double, height: Double): Double {  
 return width\*height  
}  
  
fun main() {  
 val w = readLine()!!.toDouble()  
 val h = readLine()!!.toDouble()  
 println("Rectangle area: ${rectangleArea(w, h)}")  
}

</코드>

자바나 다른 여러 언어에서 위치 기반인자를 널리 쓰고 있다. 반면 코틀린은 이름 붙은 인자(named argument)라고 불리는 방식도 제공한다. 이름 붙은 인자는 위치가 아니라 파라미터의 이름을 명시함으로써 인자를 전달하는 방식이다. 예를 들어 rectangleArea() 호출을 다음과 같이 할 수도 있다.

<코드>

rectangleArea(width = w, height = h)

</코드>

심지어는 다음 코드처럼 호출해도 된다.

<코드>

rectangleArea(height = h, width = w)

</코드>

이름 붙은 인자를 사용하면 인자의 실제 순서는 중요하지 않다. 따라서 방금 본 두 함수 호출은 모두 rectangleArea(w, h)와 똑같은 뜻이다.

한 호출 안에서 위치 기반 인자와 이름 붙은 인자를 함께 사용할 수도 있다. 코틀린 1.4 이전에는 이름이 붙은 인자를 쓰기 시작하면 그 이후의 모든 인자에 이름을 붙여야 했다. 하지만 이 제약은 너무 심하기 때문에, 코틀린 1.4부터는 이름 붙은 인자를 중간에 섞어 쓸 수 있다. 다만 이 경우 원래 인자가 들어가야 할 위치에 이름 붙은 인자를 지정해야 정상 처리되며, 그렇지 않은 경우 위치 기반 인자의 타입이 어긋나거나, 이미 할당된 인자를 재할당하기 때문에 컴파일 오류가 발생한다. 예를 들어 문자열에서 두 문자를 서로 교환하는 함수를 생각해보자(물론 String은 불변값이기 때문에 원본 문자열의 내용은 바뀌지 않는다).

<코드>

fun swap(s: String, from: Int, to: Int): String {  
 val chars = s.toCharArray() // 배열로 변환  
 // 배열 원소 교환하기  
 val tmp = chars[from]  
 chars[from] = chars[to]  
 chars[to] = tmp  
 return chars.concatToString() // 문자열로 다시 변환(toString()을 쓰면 어떤지 볼것!)  
}  
  
fun main() {  
 println(swap("Hello", 1, 2)) // Hlelo  
 println(swap("Hello", from = 1, to = 2)) // Hlelo  
 println(swap("Hello", to = 3, from = 0)) // lelHo  
 println(swap("Hello", 1, to = 3)) // Hlleo  
 println(swap(from = 1, s = "Hello", to = 2)) // Hlelo  
 // 위치 기반 인자와 이름 붙은 인자를 혼용한 경우  
 println(swap(s = "Hello", 1, 2)) // 1.4 이전에는 컴파일 오류. 1.4부터는 정상  
 println(swap(s = "Hello", 1, to = 2)) // 1.4 이전에는 컴파일 오류. 1.4부터는 정상  
 println(swap(s = "Hello", 2, from = 1)) // error: an argument is already passed for this parameter  
 // error: no value passed for parameter 'to'  
 println(swap(1, 2, s = "Hello")) // the integer literal does not conform to the expected type String  
 // error: an argument is already passed for this parameter  
 //error: no value passed for parameter 'to'  
}

</코드>

### 오버로딩과 디폴트 값

자바 메서드와 마찬가지로 코틀린 함수도 오버로딩(overloading)할 수 있다. 이 말은 이름이 같은 함수를 여럿 작성할 수 있다는 뜻이다. 다만 컴파일러가 어떤 함수를 호출해야 할지 구분할 수 있도록 오버로딩한 함수의 파라미터 타입이 모두 달라야 한다. 예를 들어 다음 두 정의는 올바른 오버로딩을 구성한다.

<코드>

fun readInt() = readLine()!!.toInt()  
fun readInt(radix: Int) = readLine()!!.toInt(radix)

</코드>

하지만 다음 두 함수는 반환값만 다르기 때문에 컴파일 오류가 난다.

<코드>

fun plus(a: String, b: String) = a + b  
fun plus(a: String, b: String) = a.toInt() + b.toInt() // error: conflicting overloads: public final fun plus(a: String, b: String): String defined(이하생략)

</코드>

주어진 호출 식에 대해 실제 호출할 함수를 결정할 때 컴파일러는 자바 오버로딩 해소(overloading resolution) 규칙과 비슷한 다음 규칙을 따른다.

1. 파라미터의 개수와 타입을 기준으로 호출할 수 있는 모든 함수를 찾는다.
2. 덜 구체적인 함수를 제외시킨다. 1에서 선택한 후보 목록에서 어떤 함수의 파라미터 타입이 다른 함수의 파라미터 타입의 상위타입(supertype)인 경우 이 함수는 다른 함수보다 덜 구체적인 함수다. 덜 구체적인 함수가 모두 제외될 때까지 이 단계를 반복한다.
3. 후보가 하나로 압축되면 이 함수가 호출할 함수다. 후보가 둘 이상이면 컴파일 오류다.

다음 함수 정의를 살펴보자.

<코드>

fun mul(a: Int, b: Int) = a\*b // 1  
fun mul(a: Int, b: Int, c: Int) = a\*b\*c // 2  
fun mul(s: String, n: Int) = s.repeat(n) // 3  
fun mul(o: Any, n: Int) = Array(n) { o } // 4

</코드>

다음은 몇 가지 함수 호출의 오버로딩을 해결한 결과다.

<코드>

mul(1, 2) // Int가 Any의 하위타입이므로 1과 4중에 1을 선택  
mul(1, 2L) // error: none of the following functions can be called with the arguments supplied  
 // 오류: (Int, Long) 타입을 받을 수 있는 함수가 없음  
mul(1L, 2) // Long, Int 타입을 받을 수 있는 함수는 4번뿐이므로 4를 선택  
mul("0", 3) // String이 Any의 하위타입이기 때문에 3과 4중에 3을 선택

</코드>

덜 구체적인 함수로 간주될 것 같은 함수를 호출하고 싶은 경우에는, 다음 코드처럼 as 타입 캐스팅(type casting) 연산자를 사용해 인자를 상위타입으로, 명시적으로 변환해야 한다.

<코드>

mul("0" as Any, 3) // (Any, Int)를 받을 수 있는 함수는 4뿐이므로 4를 선택

</코드>

as 연산에 대해서는 **8장 클래스 계층 이해하기**에서 살펴본다. 8장에서는 코틀린의 하위타입 관계와 상속에 대해 자세히 다룰 것이다.

자바에서는 오버로딩한 메서드들이 똑같은 연산을 수행하는데 사용자가 함수 인자 중 일부를 생략해 미리 정해진 디폴트 값을 사용할 수 있게 하기 위해 메서드를 오버로딩해야 하는 경우가 있다. 이번 절 맨 앞에서 정의한 readInt() 함수를 살펴보라. 두 함수 모두 입력 문자열을 정수로 변환하는데 첫 번째 함수는 10진법으로 문자열을 파싱하지만, 두 번째 함수는 다양한 진법으로 문자열을 파싱한다는 점만 다르다. 실제 다음과 같이 두 번째 함수를 사용해 첫 번째 함수를 작성할 수도 있다.

<코드>

fun readInt() = readInt(10)

</코드>

코틀린에서는 경우에 따라 함수 오버로딩을 쓰지 않아도 된다. 더 우아한 해법인 디폴트 파라미터가 있기 때문이다. 파라미터 뒤에 변수 초기화 식을 추가하면 원하는 파라미터에 디폴트 값을 제공할 수 있다.

<코드>

fun readInt(radix: Int = 10) = readLine()!!.toInt(radix)

</코드>

이제 이 함수를 인자 없이 호출해도 되고 인자를 지정해 호출해도 된다.

<코드>

val decimalInt = readInt()  
val decimalInt2 = readInt(10)  
val hexInt = readInt(16)

</코드>

디폴트 파라미터 뒤에 디폴트가 지정되지 않은 파라미터가 있는 경우, 디폴트 파라미터를 지정하지 않고 이 함수를 호출하는 유일한 방법은 이름 붙은 인자를 사용하는 것뿐이다.

<코드>

fun restrictToRange(  
 from: Int = Int.MIN\_VALUE,  
 to: Int = Int.MAX\_VALUE,  
 what: Int  
): Int = Math.max(from, Math.min(to, what))  
  
fun main() {  
 println(restrictToRange(10, what = 1))  
}

</코드>

하지만 디폴트 값이 있는 파라미터를 함수 인자 목록 뒤쪽에 몰아두는 쪽이 더 좋은 코딩 스타일이다.

디폴트 값이 있으면 인자 개수가 가변적이어서 오버로딩 해소가 복잡해질 수 있다. 다음과 같은 정의가 있을 때,

<코드>

fun mul(a: Int, b: Int = 1) = a\*b // 1  
fun mul(a: Int, b: Long = 1L) = a\*b // 2  
fun mul(a: Int, b: Int, c: Int = 1) = a\*b\*c // 3

</코드>

다음 호출을 보자.

<코드>

mul(10) // 오류: 1과 2 사이에 어느 쪽을 호출할지 결정할 수 없음  
mul(10, 20) // 인자가 더 적기 때문에 1과 3 중에 1을 선택  
mul(10, 20, 30) // 적용 가능한 함수가 3번뿐이어서 3을 선택

</코드>

인자가 2개인 mul(10, 20) 호출에 대해서는 3번 함수가 덜 구체적인 함수다. 3번 함수는 3번째 파라미터로 디폴트 값이 있는 c를 추가해서 인자가 2개인 함수를 확장한 것이기 때문이다. 하지만 1번 함수 정의를 다음과 같이 바꾸면,

<코드>

fun mul(a: Number, b: Int = 1) = a\*b

</코드>

이 경우 Number가 Int의 상위타입이기 때문에 1번 함수가 3번 함수(에서 디폴트를 생략한 경우)보다 덜 구체적인 함수로 간주되어 mul(10, 20)이 세 번째 함수로 해소된다[[2]](#footnote-2).

### varargs

앞에서 인자의 개수가 정해지지 않은 arrayOf() 같은 함수를 살펴봤다. 여러분이 직접 작성한 함수에서도 이런 기능을 쓸 수 있다. 필요한 것은 파라미터 정의 앞에 vararg 변경자(modifier)를 붙이는 것뿐이다.

<코드>

fun printSorted(vararg items: Int) {  
 items.sort()  
 println(items.contentToString())  
}  
  
fun main() {  
 printSorted(6, 2, 10, 1) // [1, 2, 6, 10]  
}

</코드>

함수 내부에서는 파라미터를 적절한 배열 타입으로 사용할 수 있다. 예를 들어 printSorted() 내부에서 items는 IntArray이다.

또한, 스프레드(spread) 연산자인 \*를 사용하면 배열을 가변 인자 대신 넘길 수 있다.

<코드>

val numbers = intArrayOf(6, 2, 10, 1)  
printSorted(\*numbers)  
printSorted(numbers) // Error: passing IntArray instead of Int

</코드>

스프레드는 배열을 복사한다는 점에 유의하라. 파라미터 배열의 내용을 바꿔도 원본 원소에는 영향을 미치지 않는다.

<코드>

fun main() {  
 val a = intArrayOf(6, 2, 10, 1)  
 printSorted(\*a) // [1, 2, 6, 10]  
 println(a.contentToString()) // [6, 2, 10, 1]  
}

</코드>

하지만 이때 얕은(shallow) 복사가 이뤄진다. 즉, 배열 내부에 참조가 들어있는 경우에는 참조가 복사되기 때문에 참조가 가리키는 데이터가 호출하는 쪽과 함수 내부 배열에서 공유된다.

<코드>

fun change(vararg items: IntArray) {  
 items[0][0] = 100  
}  
  
fun main() {  
 val a = intArrayOf(1, 2, 3)  
 val b = intArrayOf(4, 5, 6)  
 change(a, b)  
 println(a.contentToString()) // [100, 2, 3]  
 println(b.contentToString()) // [4, 5, 6]  
}

</코드>

둘 이상을 vararg 파라미터로 선언하는 것은 금지된다. 하지만 vararg 파라미터에 콤마로 분리한 여러 인자와 스프레드를 섞어서 전달하는 건 괜찮다. 호출 시 이런 호출은 원래의 순서가 유지되는 단일 배열로 합쳐진다.

<코드>

printSorted(6, 1, \*intArrayOf(3, 8), 2) // 6,1,3,8,2 순서로 원소가 들어있는 배열이 전달되고, [1, 2, 3, 6, 8]이 반환됨

</코드>

vararg 파라미터가 맨 마지막에 있는 파라미터가 아니라면, vararg 파라미터 이후의 파라미터는 이름 붙은 인자로만 전달할 수 있다. 디폴트 값과 비슷하게 vararg 파라미터도 파라미터 목록의 맨 뒤에 위치시키는 게 좋은 코딩 스타일이다. vararg 파라미터를 이름 붙은 인자로 전달할 수는 없다. 단, 이름 붙은 인자에 스프레드를 사용해서 가변 인자를 전달할 수는 있다.

<코드>

printSorted(items = \*intArrayOf(1, 2, 3))  
printSorted(items = 1, 2, 3) // Error: assigning single elements to varargs in named form is forbidden   
 // Error: the integer literal does not conform to the expected type IntArray

</코드>

디폴트 값이 있는 파라미터와 vararg를 섞어 쓰는 건 어렵다. 디폴트를 vararg보다 앞에 두면 vararg 파라미터에 첫 번째로 전달돼야 하는 값이 디폴트가 지정된 파라미터에게 전달될 값으로 간주된다. 이를 피하려면 vararg 파라미터를 이름 붙은 인자와 스프레드를 사용해 전달해야 한다. 하지만 이런 식으로 전달하는 코드는 원래 vararg를 도입했던 목적에 위배된다.

<코드>

fun printSorted(prefix: String = "", vararg items: Int) { }  
  
fun main() {  
 printSorted(6, 2, 10, 1) // Error: 6 is taken as value of prefix  
 printSorted(items = \*intArrayOf(6, 2, 10, 1)) // 정상  
}

</코드>

반면 vararg 파라미터 뒤에 디폴트 파라미터가 있는 경우에는, 디폴트 파라미터를 이름 붙은 인자로 호출해야 사용할 수 있다.

<코드>

fun printSorted(vararg items: Int, prefix: String = "") { }  
  
fun main() {  
 printSorted(6, 2, 10, 1, "!") // Error: "" is taken as part of vararg  
 printSorted(6, 2, 10, 1, prefix = "!") // 정상  
}

</코드>

vararg는 오버로드 해소에도 영향을 미친다. 다른 요소가 모두 같다면 vararg 파라미터가 있는 함수는 동일한 타입의 파라미터 수가 고정되어 있는 함수보다 덜 구체적인 함수로 간주된다. 예를 들어 다음과 같이 인자가 3개 있을 때 컴파일러는 2번 함수를 선택한다.

<코드>

fun printSorted(vararg items: Int) { } // 1  
  
fun printSorted(a: Int, b: Int, c: Int) { } // 2  
  
fun main() {  
 printSorted(1, 2, 3) // 2번 함수가 가변인자 함수가 아니므로 2번을 선택  
 printSorted(1, 2) // 적용할 수 있는 함수가 1번밖에 없으므로 1번을 선택  
}

</코드>

### 함수의 영역과 가시성

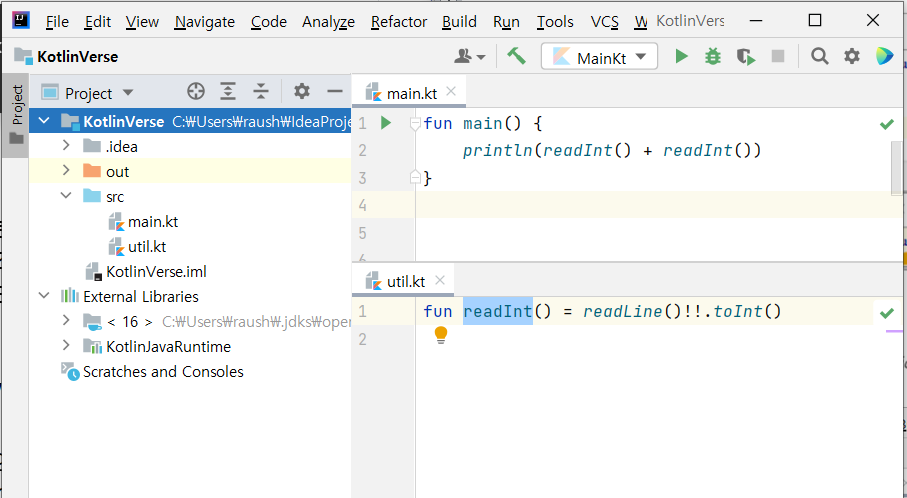
코틀린 함수는 정의된 위치에 따라 세 가지로 구분할 수 있다.

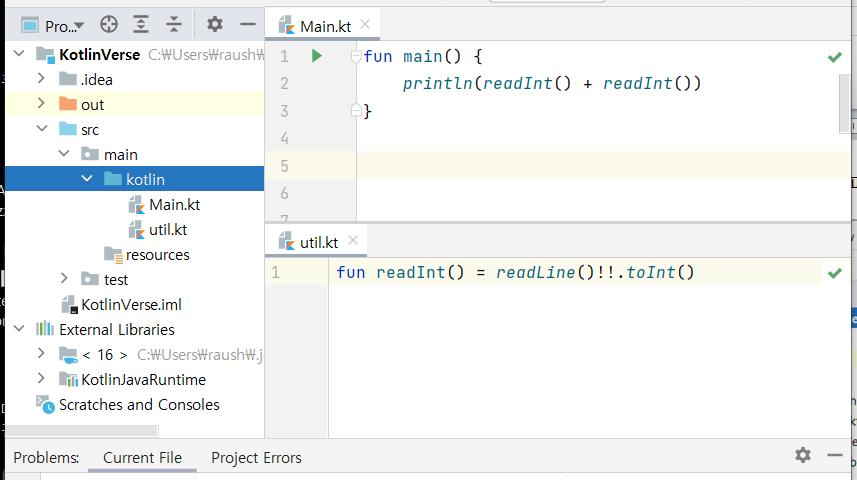
* 파일에 직접 선언된 최상위 함수
* 어떤 타입 내부에 선언된 멤버 함수
* 다른 함수 안에 선언된 지역 함수

이 장에서는 최상위 함수와 지역 함수를 다루고 멤버 함수는 **4장 클래스와 객체 다루기**에서 코틀린 클래스에 대해 설명하면서 다룰 것이다.

지금까지는 main()과 같은 최상위 함수만 선언했다. 디폴트로 최상위 함수는 공개(public) 함수다. 즉, 디폴트로 선언된 최상위 함수는 함수가 정의된 파일 내부뿐 아니라 프로젝트 어디에서나 쓰일 수 있다. 예를 들어 main.kt와 util.kt라는 두 코틀린 파일을 같은 디렉터리 안에 만들어 보자. main.kt에 정의된 main() 함수가 util.kt에 정의된 readInt() 함수를 호출하는 모습을 그림 3-2에서 볼 수 있다.

그림 3.2: 어떤 파일에 정의된 공개 함수를 다른 파일에서 호출하기

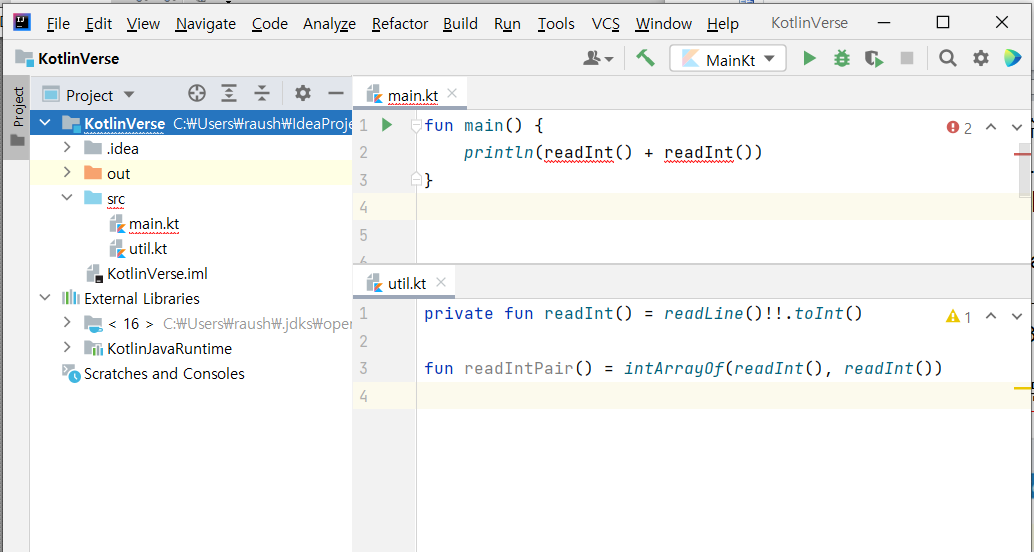


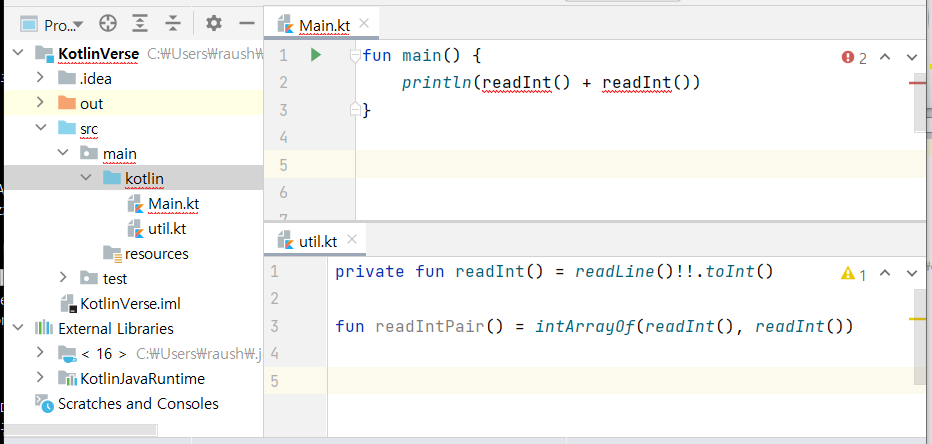


경우에 따라 프로젝트의 나머지 부분으로부터 구현 상세 내용을 숨겨 보호하고 싶을 때가 있다. 이렇게 하면 함수의 영역을 줄일 수 있고, 코드에서 함수가 쓰일 수 있는 위치를 제한할 수 있다. 이를 위해 최상위 함수 정의 앞에 private나 internal이라는 키워드를 붙일 수 있다. 이런 키워드를 가시성 변경자(visibility modifier)라고 부른다.

최상위 함수를 비공개(private)로 정의하면 함수가 정의된 파일 안에서만 해당 함수를 볼 수 있다. 예를 들어 readInt()를 비공개로 선언하면, util.kt 안에서는 이 함수를 쓸 수 있지만 main.kt에서는 이 함수를 쓸 수 없다(그림 3-3).

그림 3.3: 어떤 파일에 정의된 비공개 함수를 다른 파일에서 호출하기





internal 변경자를 적용하면 함수가 적용된 모듈 내부에서만 함수를 사용할 수 있게 제한한다. 코틀린에서 모듈은 기본적으로 함께 컴파일되는 파일 전부를 뜻한다. '함께 컴파일된다'라는 말의 구체적인 의미는 프로젝트에서 모듈 내의 파일을 모으는 빌드 시스템에 따라 달라진다. 하지만 인텔리J IDEA의 경우, 이는 한 IDE 모듈을 뜻한다. 따라서 어떤 함수를 내부(internal) 함수로 선언하면 같은 모듈에 속한 다른 파일에서는 이 함수를 사용할 수 있지만, 같은 프로젝트의 다른 모듈에서는 사용할 수 없다.

<참고> IDE 팁

**1장 강력하고 실용적인 코틀린**에서 설명했던 새 프로젝트 마법사와 비슷한 방식으로 **File > New > Module 마법사**를 사용하면 별도의 모듈을 만들 수 있다.

</참고>

public(공개)이라는 변경자도 있다. 하지만 최상위 함수는 디폴트로 공개 가시성이기 때문에 이 변경자는 불필요한 중복일 뿐이다.

지역 변수처럼 함수 내부에 로컬 함수를 정의할 수 있다. 이 함수의 영역은 함수를 감싸는 블록으로 한정된다.

<코드>

fun main() {  
 fun readInt() = readLine()!!.toInt()  
 println(readInt() + readInt())  
}  
  
fun readIntPair() = intArrayOf(readInt(), readInt()) // error: unresolved reference: readInt

</코드>

지역 함수는 자신을 둘러싼 함수, 블록에 선언된 변수나 함수에 접근할 수 있다. 이런 변수에는 지역 함수를 둘러싸고 있는 함수의 파라미터도 포함된다.

<코드>

fun main(args: Array<String>) {  
   
 fun swap(i: Int, j: Int): String {  
 val chars = args[0].toCharArray()  
 val tmp = chars[i]  
 chars[i] = chars[j]  
 chars[j] = tmp  
 return chars.concatToString()  
 }  
   
 println(swap(0, args[0].lastIndex))  
}

</코드>

지역 함수와 변수에는 가시성 변경자를 붙일 수 없다는 점에 유의하라.

**자바와 코틀린의 차이**: 일반적으로 자바 언어와 JVM에서 모든 메서드는 어떤 클래스 내부에 들어 있어야 한다. 따라서 어떻게 코틀린 최상위 함수나 로컬 함수를 JVM 플랫폼에서 컴파일할 수 있는지 궁금할 것이다. **1장 강력하고 실용적인 코틀린**에서 JVM이 볼 때 최상위 main() 함수는 코틀린 파일마다 자동으로 만들어지는 특별한 파사드 클래스의 정적 멤버라는 사실을 설명했다. 지역 함수의 경우에도 코틀린 컴파일러는 (최상위 함수와) 비슷한 트릭을 써서 지역 함수를 멤버로 가지고 지역 함수를 둘러싼 영역의 변수나 파라미터 목록 등의 문맥을 포획해주는 특별한 클래스(이를 자바의 지역 클래스와 비교할 수 있다)를 선언한다. 이 말은 지역 함수를 호출할 때마다 이런 특별한 클래스 객체를 생성하는 부가비용이 든다는 뜻이다. **5장 고급 함수와 함수형 프로그래밍 활용하기**에서 람다에 대해 설명할 때 이 주제에 대해 다시 다루겠다.

## 패키지와 임포트

코틀린 패키지는 관련 있는 선언을 묶는 방법이다. 패키지는 이름이 있고, 다른 패키지를 포함할 수 있다. 코틀린의 패키지 개념은 자바의 패키지 개념과 비슷하지만 코틀린만의 특징도 있다. 지금부터 이런 특징을 자세히 살펴보겠다.

### 패키지와 디렉터리 구조

자바와 마찬가지로 코틀린 파일에서도 맨 앞에서 패키지 이름을 지정하면 파일에 있는 모든 최상위 선언을 지정한 패키지 내부에 넣을 수 있다. 패키지를 지정하지 않으면 컴파일러는 파일이 디폴트 최상위 패키지에 속한다고 가정한다. 디폴트 최상위 패키지는 이름이 없다.

패키지 디렉티브는 package 키워드로 시작하고 점(.)으로 구별된 식별자들로 이뤄진 패키지 전체 이름(qualified name)이 뒤에 온다. 기본적으로 이 전체 이름은 프로젝트의 전체 패키지 계층에서 루트 패키지로부터 지정한 패키지에 도달하기 위한 경로다. 예를 들어 다음 파일은,

<코드>

package foo.bar.util  
  
fun readInt(radix: Int = 10) = readLine()!!.toInt(radix)

</코드>

util이라는 패키지에 속하고, util은 다시 bar 패키지에 속하며, bar는 다시 foo 패키지에 속하고, foo 패키지는 루트 패키지에 속한다. 다음 파일은 패키지 계층에서 루트 패키지 바로 아래 있는 numberUtil에 속한다.

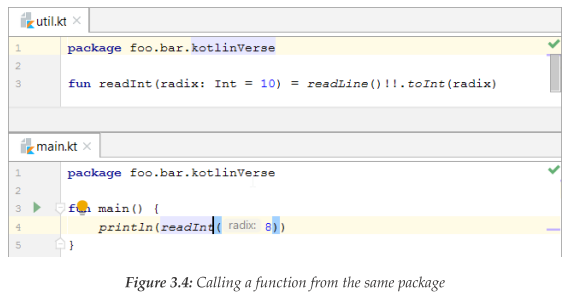
<코드>

package numberUtil  
fun readDouble() = readLine()!!.toDouble()

</코드>

같은 패키지 디렉티브를 사용하면 여러 파일을 같은 패키지에 넣을 수 있다. 이런 경우 패키지에는 각 파일의 모든 내용이 함께 포함된다.

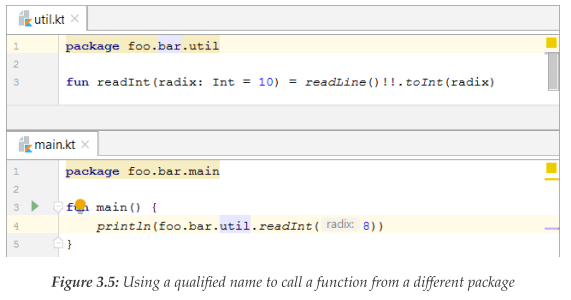
패키지를 이루는 최상위 선언에는 타입, 함수, 프로퍼티가 있다. 최상위 함수 정의에 대해서는 이미 살펴봤고, 타입이나 프로퍼티를 정의하는 방법은 4장에서 설명한다. 같은 패키지 안에서는 간단한 이름을 사용해 패키지 내에 있는 다른 정의를 참조할 수 있다. 지금까지 살펴본 예제에서는 선언들이 암시적으로 루트 패키지 안에 속해있다고 가정하고 (전체 이름을 사용하지 않고) 간단한 이름만으로 코딩을 진행해 왔다. 그림 3-4는 디폴트 패키지를 사용하지 않고 예제를 다시 구성해본 예다.

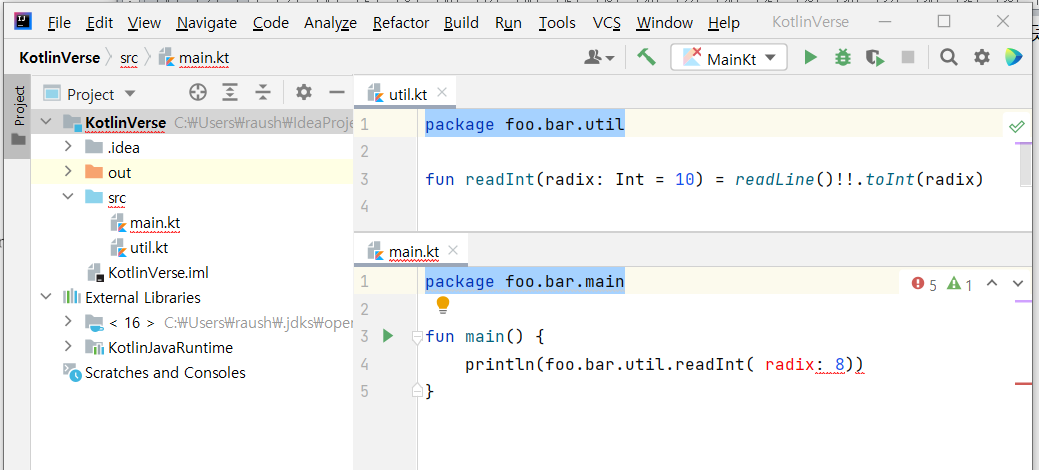
그림 3-4 같은 패키지 안에 있는 함수 호출하기[[3]](#footnote-3) 



선언이 다른 패키지에 들어있다면 어떻게 해야 할까? 이런 경우 전체 이름을 사용해 선언을 참조할 수 있다. 전체 이름은 기본적으로 간단한 이름 앞에 그 선언이 포함된 패키지의 전체 이름을 덧붙인 것이다(그림 3-5).

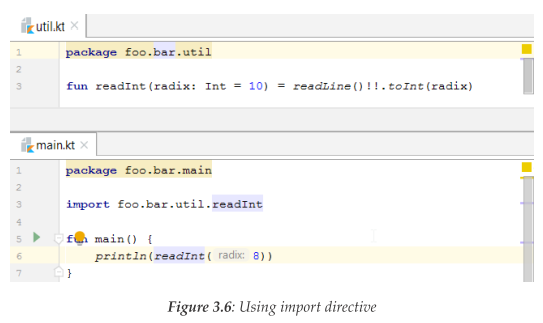
그림 3-5 전체 이름을 사용해 다른 패키지에 있는 함수 호출하기





일반적으로 이런 접근 방법을 사용하면 이름이 너무 길어서 코드를 읽기 어려워진다. 이 때문에 코틀린은 다른 메커니즘을 제공한다. 파일 맨 앞에서 선언의 전체 이름을 import 디렉티브를 사용해 임포트하면 간단한 이름을 사용해 해당 선언에 접근할 수 있다(그림 3-6).

그림 3-6: 임포트 디렉티브 사용법





<참고> IDE 팁

인텔리J 플러그인은 임포트와 관련한 지루한 작업을 대신해준다. 특히 여러분이 다른 패키지에 선언된 이름을 사용하려고 간단한 이름만 타이핑하면, IDE가 자동으로 해당 이름이 들어있는 패키지들을 보여주면서 임포트할 이름을 선택하는 팝업을 띄워준다. 또한, 사용하지 않은 임포트문을 강조해주고, Optimize Imports 명령을 사용해([Ctrl]+[Alt]+[O] 또는 [Cmd]+[Alt]+[O] 단축키를 사용해도 된다) 사용하지 않은 임포트를 제거하면서 모든 임포트를 정렬해준다.

</참고>

패키지 계층 구조는 소스 파일에 있는 패키지 디렉티브로부터 구성된 별도의 구조라는 점에 유의하라. 소스 파일 트리와 패키지 계층 구조가 일치할 수도 있지만 꼭 그럴 필요는 없다. 예를 들어 소스 파일은 모두 한 디렉터리 아래 있지만 각각이 서로 다른 패키지에 포함될 수도 있고, 한 패키지에 포함된 소스 파일들이 모두 서로 다른 디렉터리에 들어갈 수도 있다.

**자바와 코틀린의 차이**: 반면, 자바에서는 패키지 구조와 컴파일 대상 루트에 있는 소스 트리 디렉터리 구조가 같아야 한다. 둘의 경로가 다르면 컴파일 오류가 난다.

하지만 프로젝트의 여러 부분을 이동할 때 (쉽게 패키지로부터 파일 위치를 알 수 있어서) 좀 더 편하기 때문에 디렉터리와 패키지 구조를 동일하게 하는 편을 더 권장한다.

<참고> IDE 팁

기본적으로 인텔리J 플러그인은 패키지와 디렉터리 구조를 일치시키고, 그렇지 않은 경우 경고를 표시해준다. 그림 3-6의 패키지 디렉티브가 강조(회색 밑줄)됐다는 사실을 눈치챈 독자도 있을 것이다. 이유는 디렉터리 경로와 일치하지 않는 패키지를 지정했기 때문이다. 강조된 영역에서 [Alt] + [Enter]를 누르면 디렉티브를 변경하거나 파일을 임포트 디렉티브와 일치하는 디렉터리로 이동시킬 수 있다.

</참고>

### 임포트 디렉티브 사용하기

임포트 디렉티브를 사용하면 전체 이름을 사용하지 않아도 돼서 코드가 간단해진다는 사실을 앞에서 살펴봤다. 이번 절에서는 코틀린이 제공하는 임포트 디렉티브의 종류와 각각 자바의 비슷한 기능과 어떻게 다른지 살펴보겠다.

지금까지 예제에서 봤던 가장 단순한 임포트 형태는 전체 이름을 지정해 어떤 선언을 임포트하는 것이다.

<코드>

import java.lang.Math // JDK 클라스  
import foo.bar.util.readInt // 최상위 함수

</코드>

임포트 디렉티브가 클래스나 함수 등의 최상위 선언만 임포트할 수 있는 것은 아니다. 클래스 안에 내포된 클래스(nested class)나 다음 예제처럼 이넘 상수(enum constant) 등도 임포트할 수 있다.

<코드>

import kotlin.Int.Companion.MIN\_VALUE  
fun fromMin(steps: Int) = MIN\_VALUE + n // MIN\_VALUE를 간단한 이름으로 가리킴

</코드>

**4장 클래스와 객체 다루기**와 **6장 특별한 클래스 사용하기**에서 이런 경우를 더 자세히 살펴보겠다.

**자바와 코틀린의 차이**: 자바와 달리 코틀린은 타입 멤버를 임포트하는 별도의 import static 디렉티브가 없다. 코틀린의 모든 선언은 일반적인 임포트 디렉티브 구문을 사용해 임포트할 수 있다.

서로 다른 패키지에 있는 일부 선언의 이름이 똑같을 수도 있다. 이런 이름들을 한 파일에서 쓰려면 어떻게 해야 할까? 예를 들어 app.util.foo와 app.util.bar 패키지에 readInt() 함수가 들어있다고 하자. 이들을 모두 임포트하는 것은 도움이 되지 않는다.

<코드>

import app.util.foo.readInt  
import app.util.bar.readInt  
  
fun main() {  
 val n = readInt() // 오류: 두 가지 readInt() 중에 하나를 선택할 수 없음  
}

</코드>

이 둘을 구분하려면 항상 전체 이름을 사용해야 한다. 하지만 코틀린은 임포트 별명(alias)이라는 더 나은 해법을 제공한다. 이 기능을 사용하면 임포트한 선언에 새 이름을 부여할 수 있다. 그리고 이 새 이름은 임포트 디렉티브가 있는 파일 전체 영역에서 유효하다.

<코드>

import foo.readInt as fooReadInt  
import bar.readInt as barReadInt  
  
fun main() {  
 val n = fooReadInt()  
 val m = barReadInt()  
}

</코드>

또 다른 형태의 임포트로 어떤 영역에 속한 모든 선언을 한꺼번에 임포트할 수 있다. 전체 이름 뒤에 \*를 붙이면 된다. 다음과 같이 이 구문은 자바에서 같은 기능을 하는 임포트문과 꽤 비슷하다.

<코드>

import kotlin.math.\* // kotlin.math 패키지 안에 있는 모든 선언을 임포트

</코드>

이런 형태의 필요 시 임포트(import-on-demand) 기능은 구체적인 선언을 지정해 임포트하는 디렉티브보다 우선순위가 낮다. 앞의 readInt() 예제에서 한 임포트 디렉티브를 필요 시 임포트 디렉티브로 바꾸면 구체적인 임포트 문만 효과를 발휘한다.

<코드>

import app.util.foo.readInt  
import app.util.bar.\*  
  
fun main() {  
 val n = readInt() // 모호하지 않음. app.util.foo.readInt을 사용  
}

</코드>

## 조건문

조건문을 사용하면 어떤 조건의 값에 따라 둘 이상의 동작 중 하나를 수행할 수 있다. 코틀린에서는 자바의 if, switch와 비슷한 if, when이 조건문을 표현한다.

### if문으로 선택하기

if 문을 사용하면 불(boolean) 식의 결과에 따라 두 가지 대안 중 하나를 선택할 수 있다. 코틀린 if는 자바 if문과 비슷한 문법을 제공한다.

<코드>

fun max(a: Int, b: Int): Int {  
 if (a > b) return a  
 else return b  
}

</코드>

기본적으로 if문은 조건이 참일 때 첫 번째 문장을 실행하고, 조건이 거짓일 때 else 다음의 문장을 실행한다. 조건이 거짓일 때 아무 일도 하지 않아도 된다면 else 부분을 생략할 수 있다. if와 else 가지에 들어가는 문장은 모두 블록이 될 수 있다. 따라서 각 대안 안에서 여러 문장을 실행할 수도 있다.

<코드>

fun main(args: Array<String>) {  
 if (args.isNotEmpty()) {  
 val message = "Hello, ${args[0]}"  
 println(message)  
 } else {  
 println()  
 }  
}

</코드>

여기서 조건은 항상 Boolean 타입의 식이어야 한다는 점을 기억하자.

자바의 if문과 코틀린의 if문은 무엇이 다를까? 차이는 코틀린 if를 식으로 사용할 수 있다는 점이다. if 식을 사용하면 max 함수를 더 단순화할 수 있다.

<코드>

fun max(a: Int, b: Int) = if (a > b) a else b

</코드>

if의 두 가지 중 어느 쪽이 블록인 경우에도 마찬가지다. 이 경우 블록 맨 끝에 있는 식의 값이 블록 전체의 값이 된다.

<코드>

fun main() {  
 val s = readLine()!!  
 val i = s.indexOf("/")  
   
 // 10/3 같은 문자열을 /를 기준으로 10과 3으로 나눠서 나눗셈을 수행한다  
 val result = if (i>= 0) {  
 val a = s.substring(0, i).toInt()  
 val b = s.substring(i + 1).toInt()  
 (a/b).toString()  
 } else ""  
   
 println(result)  
}

</코드>

if문을 식으로 사용할 때는 양 가지가 모두 있어야 한다. else 가지가 없기 때문에 다음 코드는 컴파일되지 않는다.

<코드>

val max = if (a > b) a // error: 'if' must have both main and 'else' branches if used as an expression

</코드>

**자바와 코틀린의 차이**: 코틀린은 자바와 달리 3항 연산자(조건 ? 참일때\_식 : 거짓일때\_식)가 없다. 하지만 if를 식으로 쓸 수 있다는 점이 이 단점을 대부분 상쇄해준다.

if 식에서 return을 사용하면 편리한 경우가 있다. return 문은 존재하지 않는 값을 뜻하는 Nothing이라는 특별한 타입의 값으로 간주된다. 기본적으로 어떤 식이 Nothing 타입이라는 말은 프로그램의 순차적 제어 흐름이 그 부분에서 끝나되 어떤 잘 정의된 값에 도달하지 못한다는 뜻이다. return의 경우는 이 문장을 둘러싼 함수가 끝난다는 뜻이다. Nothing 타입은 모든 코틀린 타입의 하위타입으로 간주되기 때문에 식이 필요한 위치에 return을 사용해도 타입 오류가 발생하지 않는다. 어떤 전체 이름을 문자열로 받아, 간단한 이름 부분만 다른 이름으로 바꾸면 어떻게 되는지 알아보고 싶다고 하자. 이를 구현하면 다음과 같다.

<코드>

fun renamePackage(fullName: String, newName: String): String {  
 val i = fullName.lastIndexOf('.') // 마지막 . 위치를 찾음  
 val prefix = if (i>= 0) fullName.substring(0, i + 1) else return newName  
 return prefix + newName  
}  
  
fun main() {  
 println(renamePackage("foo.bar.old", "new")) // foo.bar.new  
}

</코드>

return e의 e 값은 return 식의 값이 아니라 함수의 반환값이라는 점에 유의하자. return 식 자체는 아무 값이 없고 Nothing 타입에 속한다. 그리고 Unit과 Nothing의 차이를 구분해야 한다. Nothing과 달리 Unit 타입에는 한 가지 인스턴스가 존재하는데, 이 인스턴스는 보통 유용한 값이 없다는 사실을 표현한다. 반면 Nothing은 아예 값이 없다는 사실을 표현한다.

### 범위, 진행, 연산

코틀린은 순서가 정해진 값 사이의 수열(interval)을 표현하는 몇 가지 타입을 제공한다. for 루프로 어떤 수 범위를 반복해야 할 때 이런 타입이 유용하다. 코틀린에서는 이런 타입을 범위(range)라고 부른다.

범위를 만드는 가장 간단한 방법은 수 값에 대해 .. 연산자를 사용하는 것이다.

<코드>

val chars = 'a'..'h' // 'a'부터 'h'까지의 모든 문자  
val twoDigits = 10..99 // 10부터 99까지의 모든 수  
val zero2One = 0.0..1.0 // 0부터 1까지의 모든 부동소수점 수

</코드>

in 연산을 사용하면 어떤 값이 범위 안에 들어 있는지 알 수 있다. 기본적으로는 한 쌍의 비교식을 조합한 것과 같다.

<코드>

val num = readLine()!!.toInt()  
println(num in 10..99) // num >= 10 && num <= 99

</코드>

이와 반대 연산인 !in도 있다. !in을 사용하면 !(a in b)를 더 간단하게 쓸 수 있다.

<코드>

println(num !in 10..99) // !(num in 10.99)

</코드>

실제로는 수 타입, Char, Boolean, String등 모든 비교 가능(comparable)한 타입에 대해 .. 연산을 쓸 수 있다. 기본적으로 <=와 >=를 쓸 수 있는 타입이라면 이 타입에 대해 ..를 사용해 범위를 만들 수 있다.

<코드>

println("def" in "abc".."xyz") // true  
println("zzz" in "abc".."xyz") // false

</코드>

.. 연산에 의해 만들어지는 범위는 닫혀있다(closed). 즉, 시작 값과 끝 값이 범위에 포함된다. 끝 값이 제외된, 반만 닫힌 범위(semi-closed range)를 만드는 다른 연산도 있다. 이 연산은 정수 타입에 대해서만 사용할 수 있고 끝 값보다 1 작은 값까지 들어있는 범위를 만들어낸다. 다음 예제에서 상계(upper bound)인 100은 범위에 포함되지 않는다.

<코드>

val twoDigits = 10 until 100 // 10..99와 같음. 100은 포함되지 않음

</코드>

내장 범위 연산에서 끝 값이 시작 값보다 확실히 더 작으면 빈 범위가 된다.

<코드>

println(5 in 5..5) // true  
println(5 in 5 until 5) // false  
println(5 in 10..1) // false

</코드>

일반적으로 주어진 타입에 대한 비교 연산이 제대로 작동하지 않는 경우 이런 특징이 성립하지 않는다. 특히 비교 연산이 추이적(transitive[[4]](#footnote-4))이지 않으면 a > b임에도 불구하고 x in a..b가 참이 될 수 있다.

범위와 연관이 있는 개념으로 진행(progression)이 있다. 진행은 정해진 간격(step)만큼 떨어져 있는 정수나 Char 값들로 이뤄진 시퀀스를 말한다. 정수나 Char에 대해 정의된 범위는 실제로는 간격이 1인 진행이라 할 수 있다. 하지만 일반적으로 진행이 범위보다 더 많은 옵션을 제공한다. 예를 들어 다음 코드처럼 downTo 연산을 사용하면 아래로 내려가는(descending) 진행을 만들 수 있다.

<코드>

println(5 in 10 downTo 1) // true  
println(5 in 1 downTo 10) // false: 빈 진행임

</코드>

그리고 진행의 간격을 지정할 수도 있다.

<코드>

1..10 step 3 // 1, 4, 7, 10  
15 downTo 9 step 2 // 15, 13, 11, 9

</코드>

진행의 간격은 양수여야 한다. 그래서 방금 본 예제처럼 downTo와 step으로 진행을 만들 때도 양수를 써야 한다.

진행의 원소는 시작 값에 간격을 연속적으로 더해서 만들어진다. 따라서 실제로는 끝 값이 진행에 속한 원소가 아닐 수도 있다. 이런 경우에는 자동으로 끝 값에 가장 가까운 값까지만 진행에 속하게 된다.

<코드>

1..12 step 3 // 1, 4, 7, 10: 1..10 step 3과 같음  
15 downTo 8 step 2 // 15, 13, 11, 9: 15 downTo 9 step 2와 같음

</코드>

범위를 사용하면 문자열이나 배열의 일부분을 뽑아낼 수 있다. subString() 함수가 닫힌 정수 범위를 받는 경우에는 두 인덱스 값을 받는 경우와 달리 끝값 위치에 있는 문자가 포함된다는 점에 유의하라.

<코드>

"Hello, World".substring(1..4) // ello  
"Hello, World".substring(1 until 4) // ell  
"Hello, World".substring(1, 4) // ell: substring(1 until 4)와 같음  
IntArray(10) { it\*it }.sliceArray(2..5) // 4, 9, 16, 25  
IntArray(10) { it\*it }.sliceArray(2 until 5) // 4, 9, 16

</코드>

범위와 진행 타입은 코틀린 표준 라이브러리에 IntRange, FloatRange, CharProgression, IntProgression 등으로 정의되어 있다. kotlin.ranges 패키지에 대한 문서<각주> <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/> </각주>를 찾아보면 범위나 진행에 관계된 모든 타입과 함수, 프로퍼티 목록을 볼 수 있다.

일반적으로 범위는 동적으로 할당되는 객체이기 때문에 비교 대신 범위를 사용하면 약간이 부가비용이 든다. 하지만 컴파일러는 꼭 필요할 때만 실제 객체를 만들어내기 위해 노력한다. 예를 들어 다음 프로그램을 실행하면 런타임에 IntRange 인스턴스를 생성하지 않고, 5를 입력한 값과 비교한다.

<코드>

fun main() {  
 val a = readLine()!!.toInt()  
 val b = readLine()!!.toInt()  
 println(5 in a..b)  
}

</코드>

따라서 성능 측면에서 볼 때 이 코드는 a <= 5 && 5 <= b라는 조건식과 같다. 범위나 진행을 주로 사용하는 다른 경우로는 for 루프를 들 수 있다.

<참고> IDE 팁

인텔리J 플러그인에는 코틀린 코드의 저수준 의미를 알아보고 싶을 때 도움이 되는 JVM 바이트코드 뷰어가 들어 있다. 이 뷰어를 열려면 IDE 메뉴에서 **Tools > Kotlin > Show Kotlin Bytecode**를 선택하라. 뷰어는 에디터에 있는 현재 파일의 바이트코드를 반영해 갱신되며, 현재 소스 코드에서 캐럿의 위치에 대응되는 바이트코드 부분을 자동으로 미리 선택해준다.

JVM 바이트코드에 익숙하지 않은 독자라면 **Decompile** 버튼을 클릭해서 바이트코드를 자바 코드로 변환할 수 있다. 코틀린 컴파일러가 생성한 바이트코드의 몇몇 세부 사항에 따라서는 디컴파일된 자바 코드가 이상해 보일 수도 있다. 하지만 그런 자바 코드라도 여전히 원래 코틀린 코드가 내부적으로 어떻게 동작하는지에 이해하기에는 충분하다.

</참고>

in/!in 연산을 지원하는 타입이 범위만 있는 것은 아니다. 문자열이나 배열 같이 다른 타입의 원소를 담는 컨테이너 종류의 타입이라면 보통 이 두 연산을 지원한다.

<코드>

val numbers = intArrayOf(3, 7, 2, 1)  
val text = "Hello!"  
println(2 in numbers) // true  
println(9 !in numbers) // true  
println(4 in numbers) // false  
println('a' in text) // false  
println('H' in text) // true  
println('h' !in text) // true

</코드>

우선순위로 보면 범위 연산인 ..는 덧셈과 중위 연산 사이에 속하며, in과 !in 연산은 중위와 비교 연산 사이에 속한다. 따라서 **2장 코틀린 언어 기초**에서 보았던 연산자 우선순위표를 다음과 같이 바꿔야 한다.

표 3-1 연산자 우선순위

|  |  |  |
| --- | --- | --- |
| 덧셈 | +, - | a + b and c // (a + b) and c |
| 범위 | .. | a..b step c // (a..b) step c |
|  |  | a in b..c // a in (b..c) |
| 중위 | 이름이 붙은 중위 연산자들 | a < b or b < c // (a < (b or b)) < c |
|  |  | a == b and b == c // (a == b) and (b == c) |
| 원소검사 | in !in | a < b in c // a < (b in c) |
|  |  | a !in b > c // (a !in b) > c |
| 비교 | < > <= >= | a < b == b < c // (a < b) == (b < c) |
|  |  | a < b && b < c // (a < b) && (b < c) |

until, downTo, step은 다른 이름이 붙은 중위 연산자(and, or 등)와 우선순위가 같다.

### when 문과 여럿 중에 하나 선택하기

if문은 두 가지 가능성 중 하나를 선택하게 해준다. 이런 if 문을 여러 개 연쇄적으로 사용하면 관심 대상인 조건을 순차적으로 검사하면서 여럿 중 하나를 선택할 수 있다. 임의의 어떤 정수를 그에 대응하는 16진 숫자로 바꾸고 싶다고 하자.

<코드>

fun hexDigit(n: Int): Char {  
 if (n in 0..9) return '0' + n  
 else if (n in 10..15) return 'A' + n – 10  
 else return '?'  
}

</코드>

하지만 코틀린은 여러 대안 중 하나를 선택할 수 있는 더 간결한 대안인 when을 제공한다. 이 기능을 사용하면 앞의 예제를 다음과 같이 다시 쓸 수 있다.

<코드>

fun hexDigit(n: Int): Char {  
 when {  
 n in 0..9 -> return '0' + n  
 n in 10..15 -> return 'A' + n – 10  
 else -> return '?'  
 }  
}

</코드>

기본적으로 when 문은 when 키워드 다음에 블록이 온다. 블록 안에는 조건 -> 문 형태로 된 여러 가지, else -> 문 형태의 가지가 하나 있을 수 있다. 각 문장은 다음 규칙에 의해 작동한다. 프로그램은 코드에 쓰여져 있는 순서대로 조건을 검사해서 가장 처음으로 참으로 평가되는 조건을 찾는다. 이런 조건이 있으면 프로그램은 이 조건에 대응하는 문을 실행한다. 만약 모든 조건이 거짓이라면 else 부분을 실행한다(물론 else 가지가 없으면 그냥 when문이 끝난다).

<참고> IDE 팁

인텔리J 플러그인은 문맥에 따라 내포된 if와 when을 서로 변환해주는 기능을 제공한다. 이를 위해서는 if/when 키워드 위에 캐럿을 위치시키고 [Alt] + [Enter]를 누르고 if를 when으로 바꿔주는 "Replace if with when"이나 when을 if로 바꿔주는 "Replace when with if" 동작을 선택하면 된다.

</참고>

when 문도 if처럼 식으로 쓸 수 있다. 이때 모든 가능한 경우에 대한 값을 만들어내기 위해 else 가지를 꼭 포함시켜야 한다.

<코드>

fun hexDigit(n: Int) = when {  
 n in 0..9 -> '0' + n  
 n in 10..15 -> 'A' + n – 10  
 else -> '?'  
}

</코드>

**자바와 코틀린의 차이**: 코틀린 when은 여러 대안 사이에 하나를 선택한다는 점에서 자바 switch문과 비슷하다. 하지만 가장 중요한 차이점은 when에서는 임의의 조건을 검사할 수 있지만 switch에서는 주어진 식의 여러 가지 값 중 하나만 선택할 수 있다는 점이다. 추가로 자바의 switch 문은 폴스루(fall-through)라는 의미를 제공한다. 어떤 조건을 만족할 때 프로그램이 해당 조건에 대응하는 문을 실행하고, 명시적으로 break를 만날 때까지 그 이후의 모든 가지를 실행하는데, 이를 폴스루라고 한다. 코틀린 when은 조건을 만족하는 가지만 실행하고 절대 폴스루를 하지 않는다.

조건이 어떤 값에 대한 동등성이나 in 연산만 수행하는 경우, 이런 경우에 더 적합한 다른 형태의 when 문이 있다. 다음 함수를 보자.

<코드>

fun numberDescription(n: Int): String = when {  
 n == 0 -> "Zero"  
 n == 1 || n == 2 || n == 3 -> "Small"  
 n in 4..9 -> "Medium"  
 n in 10..100 -> "Large"  
 n !in Int.MIN\_VALUE until 0 -> "Negative"  
 else -> "Huge"  
}

</코드>

when의 조건은 한가지 값인 n을 대상으로 하는 동등성, in, !in 뿐이다. 모든 가지에서 연산의 대상이 n 뿐이기 때문에 이런 경우 다음과 같이 다시 쓸 수 있다.

<코드>

fun numberDescription(n: Int, max: Int = 100): String = when (n) {  
 0 -> "Zero"  
 1, 2, 3 -> "Small"  
 in 4..9 -> "Medium"  
 in 10..max -> "Large"  
 !in Int.MIN\_VALUE until 0 -> "Negative"  
 else -> "Huge"  
}

</코드>

<참고> IDE 팁

인텔리J 코틀린 플러그인은 한 형태의 when 식을 다른 형태로 변환하면서 연산 대상을 필요에 따라 넣어주거나 없애줄 수 있다. when 키워드에 캐럿을 위치시키고 [Alt] + [Enter]를 누른 후, when의 대상 값을 추가해주는 "Introduce ... as subject of when"이나 대상 값을 없애주는 "Eliminate argument of when"을 선택하면 된다.

</참고>

이러한 형태의 when 문은 when의 대상이 있다는 점으로부터 앞에서 본 when 식과 구분할 수 있다. 대상은 when 뒤에 오며 괄호로 둘러싸여 있다. when 문의 각 가지에는 in/!in이 오거나 임의의 식이 오거나 else 키워드가 올 수 있다(그리고 **8장 클래스 계층 이해하기**에서 다루게 될 is와 !is도 올 수 있다). when문을 실행하는 방법은 앞에서 본 첫 번째 형식의 when과 비슷하며 다음과 같다.

* 대상 식을 평가한다. 이렇게 계산한 값이 subj라고 하자.
* 프로그램은 최초로 참인 조건을 찾을 때까지 각 가지의 조건을 코드에 나온 순서대로 평가한다. 이때 in/!in 가지는 subj가 왼쪽 피연산자인 식으로 취급하고, 임의의 식 e는 subj = e로 취급한다.
* 참인 조건을 찾으면 그 가지의 문장을 실행한다. 참인 조건이 없으면 (else 가지가 있는 경우) else 가지의 문장을 실행한다.

대상이 있는 형태에서는 한 가지 안에 여러 조건을 콤마로 분리해 쓸 수 있다(위 예제의 1, 2, 3 -> "small" 절이 이런 경우다). 조건을 평가하는 과정에서 이런 조건은 각각의 조건을 논리 OR(||)한 것처럼 취급한다.

대상이 있는 형태의 가지의 식이 꼭 Boolean일 필요는 없다. 각 식은 적절한 연산(in이나 !in이나 ==)을 적용할 수 있는 한 어떤 타입이라도 좋다.

**자바와 코틀린의 차이**: 자바 12부터는 코틀린의 대상이 있는 when과 아주 비슷한 switch 식이 생겼다. 하지만 자바 switch 식에는 몇 가지 제약이 있는데, 특히 범위 검사(코틀린의 in/!in)를 지원하지 않고, 오직 정수, 이넘, 문자열 같은 몇 가지 타입에 대해서만 사용할 수 있다. 그리고 when 가지에서는 상수가 아닌 임의의 식을 사용해도 된다는 점에 유의하라.

코틀린 1.3부터는 다음과 같이 식의 대상을 변수에 연결(binding)할 수 있다.

<코드>

fun readHexDigit() = when(val n = readLine()!!.toInt()) { // n을 정의  
 in 0..9 -> '0' + n  
 in 10..15 -> 'A' + n – 10  
 else -> '?'  
}

</코드>

이때 정의한 변수는 when 블록 내부에서만 사용할 수 있고 var로 선언할 수는 없다.

## 루프

코틀린에서는 같은 명령 시퀀스를 주어진 데이터에 대해 수행하거나 주어진 조건이 만족될 때까지 수행하는 제어 구조를 세 가지 제공한다. while과 do-while 루프는 자바와 같은 구조이며, for 루프는 자바 for-each와 아주 비슷하다. 코틀린의 모든 루프는 식이 아니고 문이기 때문에 어떤 값으로 평가되지 않으며 부수 효과를 발생시킬 수만 있다.

### while과 do-while 루프

사용자가 입력한 정수의 합계를 구하고 싶다고 하자. 사용자가 입력한 값을 읽어서 0이면 루프를 끝내고 합계를 표시하기로 약속하자.

<코드>

fun main() {  
 var sum = 0  
 var num

do {  
 num = readLine()!!.toInt()  
 sum += num  
 } while (num != 0)  
   
 println("Sum: $sum")  
}

</코드>

do-while 루프는 다음 규칙에 따라 평가된다.

1. do와 while 키워드 사이에 있는 루프 몸통을 실행한다.
2. while 키워드 다음에 있는 조건을 평가한다. 이 값이 참이면 1단계로 가고, 이 값이 거짓이면 루프 문 다음에 있는 문을 실행한다.

루프 몸통을 실행한 다음에 조건을 검사하기 때문에 루프 몸통이 최소 한번은 실행된다는 사실에 유의하자.

while문의 경우는, 어떤 조건이 참인 동안 루프를 실행하지만 루프 몸통을 실행하기 전에 조건을 먼저 검사한다. 이 경우 처음부터 조건이 거짓이면 루프 몸통이 한번도 실행되지 않는다.

자, 다음과 같은 사용자가 값을 맞히는 프로그램을 작성한다고 해보자. 어떤 수를 미리 생성하고, 사용자가 추측하는 값을 입력 받아서 추측값과 생성한 값이 다르면 추측값에 대한 힌트(보다 크다, 보다 작다)를 제공하는 과정을 반복한다.

<코드>

import kotlin.random.\*  
  
fun main() {  
 val num = Random.nextInt(1, 101)  
 var guess = 0  
   
 while (guess != num) {  
 guess = readLine()!!.toInt()  
 if (guess < num) println("Too small")  
 else if (guess > num) println("Too big")  
 }  
 println("Right: it's $num")  
}

</코드>

표준 라이브러리의 Random.nextInt()를 사용해 수를 생성한다. 이 예제는 코틀린 while과 do-while 문을 사용하는 법이 자바와 같다는 사실을 잘 보여준다.

### for 루프와 이터러블

코틀린의 for 루프를 사용하면 컬렉션과 비슷하게 여러 값이 들어있을 수 있는 값에 대한 루프를 수행할 수 있다. 예를 들어 배열 원소에 대한 for 루프를 수행해서 원소의 합계를 구할 수 있다.

<코드>

fun main() {  
 val a = IntArray(10) { it\*it } // 0, 1, 4, 9, 16, …  
 var sum = 0  
   
 for (x in a) {  
 sum += x  
 }  
   
 println("Sum: $sum") // Sum: 285  
}

</코드>

루프는 다음 세 부분으로 이뤄진다.

1. 반복 대상을 담을 변수 정의(x)
2. 반복에 사용할 값들이 담겨있는 컨테이너를 계산하기 위한 식(a)
3. 루프 몸통에 해당하는 문({ sum += 1 }). 반복 시 이 몸통이 실행된다.

반복 변수는 루프 몸통 안쪽에서만 접근이 가능하며 매 루프마다 자동으로 새로운 값이 들어간다. 일반 변수와 달리 루프 변수에는 val이나 var를 붙이지 않는다는 점과 루프 변수는 자동으로 불변값이 된다는 사실에 유의하라. 즉, 루프 몸통에서 루프 변수를 변경할 수는 없다. 가장 단순한 루프 변수 정의는 간단한 식별자다. 원한다면 루프 변수의 타입을 지정할 수도 있지만 실전에서는 거의 쓸 일이 없다.

<코드>

for (x: Int in a) {  
 sum += x  
}

</코드>

**자바와 코틀린의 차이**: 코틀린의 for 루프는 Iterable 인스턴스에 대한 루프를 간편하게 작성하도록 해주는 자바 for-each 루프와 꽤 비슷하다. 배열, 리스트, 집합은 Iterable에 속하며, 사용자가 정의한 타입도 Iterable을 상속할 수 있다. 하지만 코틀린에는 선언, 초기화, 검사 및 갱신으로 구성되는 일반 자바 for 루프에 해당하는 언어 구조가 없다. 코틀린에서는 앞에서 살펴본 while 루프로 이런 루프를 표현하거나, 범위와 진행을 사용하는 for 루프를 통해 표현할 수 있다.

for 루프를 사용해 문자열의 각 문자에 대한 루프를 수행할 수도 있다. 예를 들어 이진 양수를 표현하는 문자열을 Int로 파싱하는 함수를 직접 작성한다고 가정하자.

<코드>

fun parseIntNumber(s: String, fallback: Int = -1): Int {  
 var num = 0  
   
 if (s.length !in 1..31) return fallback  
   
 for (c in s) {  
 if (c !in '0'..'1') return fallback  
 num = num\*2 + (c - '0')  
 }  
 return num  
}

</코드>

파싱할 문자열이 올바른 이진수 형태가 아니라면 이 함수는 fallback 값을 반환한다.

**자바와 코틀린의 차이**: 자바에서는 문자열의 각 문자에 대해 루프를 직접 수행할 수 없다. 따라서 인덱스를 사용한 루프를 돌거나, 문자열을 문자 배열로 바꾸는 등의 대안을 사용해야 한다.

수 범위에 대한 일반적인 루프를 어떻게 수행할 수 있을까? 이런 경우 앞 절에서 보여준 진행 과정을 사용해야 한다. 배열에 있는 원소 중에 인덱스가 짝수인 원소들만 2배로 변경하고 싶다고 하자.

<코드>

val a = IntArray(10) { it\*it } // 0, 1, 4, 9, 16, ...  
  
for (i in 0..a.lastIndex) { // 0, 1, 2, 3, ...  
 if (i % 2 == 0) { // 0, 2, 4, 6, ...  
 a[i] \*= 2  
 }  
}

</코드>

이 루프 간격을 사용한 진행으로 더 간편하게 작성할 수도 있다.

<코드>

for (i in 0..a.lastIndex step 2) { // 0, 2, 4, 6, ...  
 a[i] \*= 2  
}

</코드>

문자열과 배열에는 원소나 문자의 인덱스 범위를 제공하는 indices라는 프로퍼티가 들어있다.

<코드>

val a = IntArray(10) { it\*it } // 0, 1, 4, 9, 16, ...  
for (i in a.indices step 2) { // 0, 2, 4, 6, ...  
 a[i] \*= 2  
}

</코드>

for 루프의 진정한 아름다움은 컴파일러가 수 범위나 컬렉션 등 몇 가지에 대한 용례만 지원하지 않고 다양한 값에 대한 루프를 수행할 수 있는 통일된 메커니즘을 제공한다는 점에 있다. 어떤 컨테이너를 for 루프에 사용하기 위해서는 컨테이너가 iterator() 함수를 지원하기만 하면 된다. 이 iterator() 함수는 원소 값을 추출하는 기능을 제공하는 이터레이터(Iterator 타입)를 반환한다. 이에 대한 자세한 논의는 **7장 컬렉션과 I/O 자세히 알아보기**에서 알아보고, 지금은 여러 표준 코틀린 타입이 이런 이터레이터를 기본 제공한다는 점을 알면 된다. 그래서 진행, 배열, 문자열 등에 대해 for 루프를 쓸 수 있다. 그리고 나중에 보겠지만, 확장(extension) 메커니즘을 사용하면 원하는 어떤 타입이든 iterator() 함수를 덧붙일 수 있기 때문에 여러분이 원하는 모든 타입을 확장해서 for 루프로 반복을 하게 만들 수 있다.

**자바와 코틀린의 차이**: 자바 for-each 루프는 Iterable의 모든 하위 타입에 사용할 수 있다는 점에서 코틀린 for 문과 비슷하다. 하지만 컨테이너가 특정 타입일 필요가 없기 때문에 코틀린의 for 루프 관례가 더 유연하다. 단지 어떤 타입이 iterator()라는 함수를 제공하기만 하면 for 루프를 사용할 수 있다.

### 루프 제어 흐름 변경하기: break와 continue

때로는 일반적인 루프 흐름을 바꾸고 싶을 때가 있다. 예를 들어서 루프의 시작이나 끝 부분이 아니라 루프 중간에서 종료 조건을 검사하면 편리할 수 있다. 이런 경우를 위해 코틀린은 몇 가지 식을 제공한다.

* break는 즉시 루프를 종료시키고, 실행 흐름이 루프 바로 다음 문으로 이동하게 만든다.
* continue는 현재 루프 이터레이션을 마치고 조건 검사로 바로 진행하게 만든다.

즉, 이 두 문장은 자바의 동일한 문장과 같은 기능을 제공한다. 예를 들어 앞에서 본 숫자 추측하기 프로그램을 살펴보자. 다음과 같이 break를 써서 같은 기능을 하는 프로그램을 만들 수 있다.

<코드>

import kotlin.random.\*  
  
fun main() {  
 val num = Random.nextInt(1, 101)  
  
 while (true) {  
 val guess = readLine()!!.toInt()  
   
 if (guess < num) println("Too small")  
 else if (guess > num) println("Too big")  
 else break  
 }  
   
 println("Right: it's $num")  
}

</코드>

루프 몸통 중간에서 루프 종료 조건을 검사하기 때문에, 루프의 진입 조건이 불필요해졌다. 덕분에 guess 변수를 루프 내부로 옮길 수 있다.

**자바와 코틀린의 차이**: return과 마찬가지로 코틀린의 break, continue문도 Nothing 타입의 식으로 쓸 수 있다. 예를 들어 앞의 프로그램을 출력하기 전에 메시지를 계산하는 프로그램으로 다시 작성할 수도 있다.

<코드>

import kotlin.random.\*  
  
fun main() {  
 val num = Random.nextInt(1, 101)  
   
 while (true) {  
 val guess = readLine()!!.toInt()  
 val message =  
 if (guess < num) "Too small"  
 else if (guess > num) "Too big"  
 else break  
 println(message)  
 }  
   
 println("Right: it's $num")  
}

</코드>

하지만 이런 기능을 남용하면 안 된다. 실제 더 복잡한 코드에서 이런 기능을 사용하면 코드를 이해하기 어려워질 수 있다.

주어진 문자열에서 영어 문자들이 몇 번씩 나오는지 수를 세고 싶다고 하자. 다음 예제에서는 배열에 접근하기 전에 문자가 영어 알파벳인지 검사해서 그렇지 않은 경우에는 루프를 빠져 나오는 continue 식을 사용한다.

<코드>

fun countLetters(text: String): IntArray {  
 val counts = IntArray('z' - 'a' + 1)  
   
 for (char in text) {  
 val charLower = char.toLowerCase()  
 if (charLower !in 'a'..'z') continue  
 counts[charLower - 'a']++  
 }  
   
 return counts  
}

</코드>

**자바와 코틀린의 차이**: 자바에서는 switch에서 남은 가지의 실행을 막기 위해서 break를 사용하기도 한다. 하지만 코틀린 when 식은 페일스루를 사용하지 않기 때문에 코틀린 break는 자바 break와 같지 않다. 이로 인한 혼동(특히 자바를 사용하던 코틀린 개발자의 혼동)을 막기 위해 코틀린 1.4 이전에는 when 내부에서 continue나 break를 사용하는 것은 금지되어 있었다. 따라서 코틀린 1.4 미만 버전에서 수 추측 프로그램의 if 연쇄를 한 when 식으로 바꾼다면 컴파일러가 오류를 낼 것이다.

<코드>

val message = when {  
 guess < num -> "Too small"  
 guess > num -> "Too big"  
 else -> break // 1.4 이전에는 오류, 1.4부터는 정상  
}

</코드>

하지만, 코틀린 1.4부터는when 내에서 사용하는 break나 continue는 when을 둘러싸고 있는 가장 가까운(when의 바로 바깥) 루프로 제어가 이동한다. 한편, continue는 미래의 어느 시점에 나올 코틀린 버전에서 명시적인 페일스루에 사용하기 위해 예약되어 있다.

하지만 when을 둘러싼 가장 가까운 루프가 아니라 그보다 밖의 루프로 제어를 넘기거나, 코틀린 1.4 미만 버전을 사용할 때는 레이블이 붙은 break/continue를 사용해야 한다. 이에 대해 다음 절에서 살펴보자.

### 내포된 루프와 레이블

루프를 내포시켜 사용하는 경우 앞 절에서 본 간단한 break/continue 식은 가장 안쪽에 내포된 루프에만 적용된다. 경우에 따라 더 밖에 있는 루프의 제어 흐름을 변경하고 싶을 때가 있는데, 이를 위해 코틀린은 자바의 임의 레이블과 비슷하지만 약간 다른 문법의 레이블 기능을 제공한다.

어떤 정수 배열 안에 어떤 정해진 순서로 정수가 배열된 하위 배열이 있는지 찾는 함수를 작성하고 싶다고 하자. 이 함수는 문자열의 indexOf()와 비슷한 역할을 한다.

<코드>

fun indexOf(subarray: IntArray, array: IntArray): Int {  
 outerLoop@ for (i in array.indices) {  
 for (j in subarray.indices) {  
 if (subarray[j] != array[i + j]) continue@outerLoop  
 }  
 return i  
 }  
   
 return -1  
}

</코드>

여기서는 바깥쪽 루프에 레이블을 붙이고, 하위 배열의 오프셋을 찾는 과정에서 하위 배열과 배열의 원소가 일치하지 않는다는 사실을 발견하자 마자 continue@outerLoop를 사용해 바깥쪽 루프의 현재 이터레이션을 끝내고 다음 이터레이션을 시작한다. 이 지점에서 하위 배열의 나머지 원소를 비교할 필요가 없다는 사실을 알기 때문에 바깥쪽 배열의 다음 오프셋부터 하위 배열 비교를 다시 시작해야 한다.

코틀린에서는 어느 문장 앞에든 레이블을 붙일 수 있지만, break와 continue에는 구체적으로 loop 앞에 붙은 레이블만 사용할 수 있다. 그렇지 않으면 컴파일러가 오류를 보고한다. 레이블 이름은 배열이나 함수와 마찬가지로 임의의 식별자가 될 수 있다.

**자바와 코틀린의 차이**: 코틀린과 자바는 레이블 정의와 사용 문법이 다르다.

<코드>

loop@ while(true) break@loop // 코틀린  
loop: while(true) break loop // 자바

</코드>

레이블을 사용하면 (레이블을 사용하는 다른 경우도 있지만) break와 continue를 when 식 안에서 쓰면서 제어를 옮길 대상 루프를 지정할 수 있고, 다시 이 when 식을 루프 몸통 안에서 사용할 수 있다. 이로 인해 앞 절에서 살펴본 수 추측 프로그램을 다음과 같이 작성할 수 있다.

<코드>

import kotlin.random.\*  
  
fun main() {  
 val num = Random.nextInt(1, 101)  
   
 loop@ while (true) {  
 val guess = readLine()!!.toInt()  
   
 val message = when {  
 guess < num -> "Too small"  
 guess > num -> "Too big"  
 else ->break@loop // 정상  
 }  
 println(message)  
 }  
 println("Right: it's $num")  
}

</코드>

## 꼬리 재귀 함수

코틀린은 꼬리 재귀(tail recursive) 함수에 대한 최적화 컴파일을 지원한다. 이번에는 어떤 정수 배열에 대한 이진 검색(binary search)을 수행하는 함수를 작성해보자. 배열이 오름차순으로 정렬되어 있다고 가정하고, 검색 함수를 재귀적으로 작성하자.

<코드>

tailrec fun binIndexOf(  
 x: Int,  
 array: IntArray,  
 from: Int = 0,  
 to: Int = array.size  
): Int {  
 if (from == to) return -1  
 val midIndex = (from + to - 1) / 2  
 val mid = array[midIndex]  
 return when {  
 mid < x ->binIndexOf(x, array, midIndex + 1, to)  
 mid > x ->binIndexOf(x, array, from, midIndex)  
 else ->midIndex  
 }  
}

</코드>

이 정의는 이진 검색의 아이디어를 깔끔하게 보여준다. 하지만 일반적으로 비-재귀 버전과 비교해보면 성능상 부가비용이 약간 들고 스택 오버플로(stack overflow)가 발생할 가능성이 있다. 하지만 코틀린에서는 함수에 tailrec을 붙이면 컴파일러가 재귀 함수를 비재귀적인 코드로 자동으로 변환해준다. 그 결과 양쪽의 장점, 즉 재귀 함수의 간결함과 비재귀 루프의 성능만을 취할 수 있다. 즉, 앞에서 본 함수는 다음 코드와 같다.

<코드>

// 앞의 binIndexOf 함수를 코틀린 컴파일러가 컴파일한 결과는 이런 식으로 루프로 작동한다

fun binIndexOf(  
 x: Int,  
 array: IntArray,  
 from: Int = 0,  
 to: Int = array.size  
): Int {  
 var fromIndex = from  
 var toIndex = to  
   
 while (true) {  
 if (fromIndex == toIndex) return -1  
 val midIndex = (fromIndex + toIndex - 1) / 2  
 val mid = array[midIndex]  
   
 when {  
 mid < x ->fromIndex = midIndex + 1  
 mid > x ->toIndex = midIndex  
 else -> return midIndex  
 }  
 }  
}

</코드>

이런 변환을 적용하려면 함수가 재귀 호출 다음에 아무 동작도 수행하지 말아야 한다. 이 말이 바로 꼬리 재귀라는 용어가 뜻하는 바다. 함수에 tailrec을 붙였는데 꼬리 재귀가 아니라는 사실을 컴파일러가 발견하면, 컴파일러는 경고를 표시하고 함수를 일반적인 재귀적인 함수로 컴파일한다. 예를 들어 다음 합계 함수는 sum (array, from + 1, to)를 호출한 결과에 덧셈을 수행하기 때문에 꼬리재귀가 아니다.

<코드>

tailrec fun sum(array: IntArray, from: Int = 0, to: Int = array.size): Int {  
 // warning: a function is marked as tail-recursive but no tail calls are found   
 // warning: recursive call is not a tail call  
 return if (from < to) return array[from] + sum(array, from + 1, to) else 0  
}

</코드>

함수에 tailrec을 붙였는데 함수가 재귀 함수가 아닌 경우에도 컴파일러가 경고를 표시한다.

<코드>

tailrec fun sum(a: Int, b: Int): Int {  
 return a + b // warning: a function is marked as tail-recursive but no tail calls are found   
}

</코드>

## 예외 처리

코틀린의 예외 처리(exception handling)는 자바의 접근 방법과 아주 비슷하다. 함수는 정상 종료, 즉 어떤 값을 반환(어쩌면 Unit 타입의 뻔한 값을 반환할 수도 있다)하거나, 비정상적으로 오류가 발생한 경우 예외를 던질(throw) 수 있다. 예외가 발생한 경우에는 함수를 호출한 쪽에서 예외를 잡아내거나(catch), (잡아내지 않는 경우) 함수 호출 스택의 위로 예외가 전달될 수 있다. 이제 예외 관련 제어 구조를 살펴보자.

### 예외 던지기

오류 조건을 신호로 보내려면 자바와 마찬가지로 throw 식에 예외 객체를 사용해야 한다. 앞에서 본 parseIntNumber() 함수를 문자열이 잘못된 경우 어떤 폴백 값을 돌려주는 대신 오류를 발생시키게 고쳐보자.

<코드>

fun parseIntNumber(s: String): Int {  
 var num = 0  
   
 if (s.length !in 1..31) throw NumberFormatException("Not a number: $s")  
   
 for (c in s) {  
 if (c !in '0'..'1') throw NumberFormatException("Not a number: $s")  
 num = num\*2 + (c - '0')  
 }  
   
 return num  
}

</코드>

**자바와 코틀린의 차이**: 자바와 달리 코틀린에서는 클래스 인스턴스를 생성(여기서는 예외)할 때 new 같은 특별한 구문을 사용하지 않는다. 코틀린에서는 NumberFormatException("Not a number: $s")처럼 생성자 호출도 일반 함수 호출과 똑같이 생겼다.

예외를 던질 때는 다음과 같은 일이 벌어진다.

* 프로그램은 예외를 잡아내는 핸들러(exception handler)를 찾는다. 예외와 일치하는 예외 핸들러가 있다면 예외 핸들러가 예외를 처리한다.
* 현재 함수 내부에서 핸들러를 찾을 수 없으면 함수 실행이 종료되고 함수가 스택에서 제거(pop) 된다. 그리고 호출한 쪽의 문맥 안에서 예외 핸들러 검색을 수행한다. 이런 경우를 예외를 호출자에게 전파(propagate)했다고 말한다.
* 프로그램 진입점에 이를 때까지 예외를 잡아내지 못하면 현재 스레드가 종료된다.

즉, 코틀린 예외 처리 단계는 기본적으로 자바와 같다.

**자바와 코틀린의 차이**: 코틀린에서는 앞에서 봤던 break나 continue와 같이 throws 식도 Nothing 타입의 식이다. 예를 들면 다음과 같다.

<코드>

fun sayHello(name: String) {  
 val message =  
 if (name.isNotEmpty()) "Hello, $name"  
 else throw IllegalArgumentException("Empty name")  
   
 println(message)  
}

</코드>

### try 문으로 예외 처리하기

코틀린에서 예외를 처리할 때는 기본적으로 자바와 똑같은 문법의 try 문을 사용한다. 입력으로 받은 문자열을 수로 파싱할 수 없는 경우 어떤 디폴트 값을 반환하는 다음 함수를 생각해보자.

<코드>

import java.lang.NumberFormatException  
  
fun readInt(default: Int): Int {  
 try {  
 return readLine()!!.toInt()  
 } catch (e: NumberFormatException) {  
 return default  
 }  
}

</코드>

예외가 발생할 수 있는 코드(여기서는 toInt() 호출)를 try 블록으로 감싼다. 방금 본 첫 번째 형태의 try 문에는 최소한 하나 이상 적절한 타입의 예외(여기서는 NumberFormatException)를 잡아내는 catch 블록이 있어야 한다. 처리할 예외는 예외 파라미터로 표현되며, catch 블록의 내부에서는 이 파라미터를 마음대로 쓸 수 있다. try 블록 내부의 코드가 예외를 던지면 코드 실행이 중단되고 프로그램은 예외를 처리할 수 있는 첫 번째 catch 블록으로 제어를 이동한다. 만약 예외와 일치하는 catch 블록이 없으면 예외가 전파된다.

**자바와 코틀린의 차이**: 자바 7부터는 catch (FooException | BarException e) {} 같은 구문을 사용해 한 캐치 블록 안에서 여러 예외를 처리할 수 있다. 코틀린에서는 이런 핸들러를 아직 지원하지 않는다.

catch 블록은 선언된 순서대로 예외 타입을 검사하기 때문에, 어떤 타입을 처리할 수 있는 catch 블록을 그 타입의 상위 타입을 처리할 수 있는 catch 블록보다 앞에 작성해야 한다. 그렇지 않으면 상위 타입을 잡아내는 핸들러가 하위 타입인 예외도 모두 잡아내 버린다. 다음 코드에서 NumberFormatException는 Exception의 하위타입이기 때문에 두 번째 catch 블록은 실제로는 죽어있는 코드다.

<코드>

import java.lang.NumberFormatException  
  
fun readInt(default: Int): Int {  
 try {  
 return readLine()!!.toInt()  
 } catch (e: Exception) {  
 return 0  
 } catch (e: NumberFormatException) {  
 return default // 죽은 코드  
 }  
}

</코드>

**자바와 코틀린의 차이**: 자바에서는 도착할 수 없는 코드를 금지하기 때문에 이런 식의 코드는 금지된다.

자바와 코틀린의 try 문에서 가장 크게 다른 점은 코틀린 try가 식이라는 점이다. 이 식의 값은 (예외가 발생하지 않은 경우) try 블록의 값이거나, 예외를 처리한 catch 블록의 값이 된다.

<코드>

import java.lang.NumberFormatException  
  
fun readInt(default: Int) = try {  
 readLine()!!.toInt()  
} catch (e: NumberFormatException) {  
 default  
}

</코드>

**자바와 코틀린의 차이**: 자바와 달리 코틀린에서는 검사 예외(checked exception)와 비검사 예외(unchecked exception)를 구분하지 않는다. 큰 프로젝트에서 발생할 수 있는 예외를 함수에 지정하도록 요구해도 실제로는 생산성이 저하되고 불필요하게 긴 준비 코드를 생성한다는 사실을 (자바를 사용한 경험으로부터) 알았기 때문이다.

try 문의 다른 형태는 finally 블록을 사용한다. finally 블록은 try 블록을 떠나기 전에 프로그램이 어떤 일을 수행하도록 만들어준다.

<코드>

import java.lang.NumberFormatException  
  
fun readInt(default: Int) = try {  
 readLine()!!.toInt()  
} finally {  
 println("Error")  
}

</코드>

이런 finally 블록은 try 블록 앞이나 내부에서 할당한 자원을 해제할 때 유용하다. 예를 들어 파일을 닫거나 네트워크 연결을 닫는 등의 일을 할 수 있다. 그리고 catch와 finally를 한 try 문 안에서 함께 사용할 수도 있다.

try 블록을 식으로 사용할 경우, finally 블록의 값은 전체 try 블록의 값에 영향을 끼치지 못한다는 점에 유의하라.

**자바와 코틀린의 차이**: 자바 7에 도입된 자원을 활용하는 try(try-with-resource)에 익숙한 독자도 있을 것이다. 이를 사용하면 파일 스트림이나 네트워크 연결 등의 자원을 자동으로 정리할 수 있다. 코틀린은 이런 목적에 쓸 수 있는 특별한 언어 구조를 제공하지는 않지만, 똑같은 작업을 수행할 수 있는 라이브러리 함수를 제공한다. 이에 대해서는 **7장 컬렉션과 I/O 자세히 알아보기**에서 더 자세히 살펴본다.

## 결론

이번 장에서 다룬 내용을 정리해보자. 우리는 명령형 프로그래밍의 알고리즘적인 근간을 만들어내는 기초 제어 구조에 대해 배웠다. 공통으로 쓰이는 코드 조각을 편리하게 재사용하기 위해 함수를 어떻게 정의하고 사용하는지 배웠다. 마지막으로 관련 있는 선언을 패키지로 묶어서 프로그램 구조를 잡는 방법에 대해 살펴봤다. 여러분은 이제 코틀린을 활용해 명령형 프로그래밍과 프로시저 중심의 프로그래밍(procedural programming[[5]](#footnote-5))을 작성할 때 필요한 모든 지식을 갖췄다.

다음 장에서는 객체 지향 프로그래밍으로 나아갈 것이다. 클래스와 객체를 정의하는 방법을 살펴보고, 클래스 초기화에 대해 이해하며, 프로퍼티를 선언하고 사용하는 방법을 배우고, 코틀린에서 null 값을 다루는 방법에 대해 알아볼 것이다.

## 문제

1. 식이 본문인 함수란 무엇인가? 블록이 본문인 함수 대신 식이 본문인 함수를 쓰면 더 좋은 경우는 어떤 경우인가?
2. 디폴트 파라미터와 함수 오버로딩 중 어느 쪽을 써야 할지를 어떻게 결정할 수 있을까?
3. 이름 붙은 인자를 사용할 경우의 장단점은 무엇인가?
4. 인자 개수가 가변적인 함수를 정의하는 방법은 무엇인가? 코틀린과 자바의 vararg 함수의 차이는?
5. Unit와 Nothing 타입을 어디에 사용하는가? 이들을 자바의 void와 비교해 설명하라. Nothing이나 Unit이 타입인 함수를 정의해 사용할 수 있는가?
6. return return 0과 같은 코드의 의미를 설명해보라. 이런 코드가 올바르지만 불필요한 중복이 있는 것으로 여겨지는 이유는 무엇인가?
7. return 문을 사용하지 않는 함수를 정의할 수 있는가?
8. 지역 함수란 무엇인가? 이런 함수를 자바에서는 어떻게 흉내 낼 수 있을까?
9. 공개(public)와 비공개(private) 최상위 함수는 어떤 차이가 있는가?
10. 패키지를 사용해 코드를 어떻게 여러 그룹으로 나눌 수 있는가? 자바와 코틀린 패키지의 가장 핵심적인 차이는 무엇인지 설명하라.
11. 임포트 별명이란 무엇인가? 자바의 정적 임포트와 비슷한 임포트를 코틀린에서는 어떻게 처리하는가?
12. if문/식은 어떤 일을 하는가? 각각을 자바의 if 문과 3항 조건 연산자(?:)와 비교해보라.
13. when 문을 처리하는 알고리즘을 설명하라. 자바 switch와 코틀린 when은 어떤 차이가 있는가?
14. 자바 for (int i = 0; i< 100; i++)와 같이 수를 세는 루프를 코틀린에서는 어떻게 구현하는가?
15. 코틀린이 제공하는 루프문에는 어떤 것이 있는가? while과 do...while의 차이는 무엇인가? 코틀린 for 루프를 사용해야 하는 이유는 무엇인가?
16. break와 continue를 사용해 루프의 제어 흐름을 어떻게 변경할 수 있는가?
17. 예외 처리 과정을 전체적으로 설명하라. 자바와 차이점은 무엇인가? 자바와 코틀린 try문의 차이에 대해 설명하라.

1. 옮긴이 - 모든 언어의 컴파일러가 파라미터 타입을 추론하지 못하는 것은 아니다. 특히 함수형 프로그래밍 언어의 컴파일러는 대부분 함수 정의의 파라미터 타입도 가장 일반화된 타입으로 추론해준다. 이런 차이는 채택하는 타입 추론 엔진이 달라서 생긴다. [↑](#footnote-ref-1)
2. 옮긴이 - 실제로는 Number에 times() 함수가 정의되어 있지 않기 때문에 fun mul(a: Number, b: Int = 1) = a\*b 라는 코드가 컴파일 되지 **않는다**. 이 예제를 오버로딩과 디폴트 인자 해소시 타입을 어떻게 고려하는지 알려주기 위한 사고실험으로 간주하라. [↑](#footnote-ref-2)
3. 옮긴이 - readInt(radix: 8)에서 흐리게 radix:라고 표시된 부분은 인텔리J의 파라미터 이름 힌트 기능에 의해 인텔리J가 표시해준 부분이다(직접 radix:를 입력하면 구문분석 오류가 난다). File>Settings>Editor>Inlay Hints>Kotlin에서 Show parameter hints 체크를 해제하면 파라미터 힌트가 사라진다. [↑](#footnote-ref-3)
4. 옮긴이 - 어떤 집합 A에 대한 릴레이션(relation) R이 있을 때 a R b와 b R c가 성립하면 a R c도 참인 경우, R이 추이적이라고 말한다. [↑](#footnote-ref-4)
5. 옮긴이 - "procedural programming"이라는 용어를 순차적 프로그래밍이라고 번역하는 경우가 자주 있지만 오해의 여지가 있다. 실제로 "procedural programming"은 함수 또는 프로시저 호출을 중심으로 코드를 작성하는 방법을 뜻하며 단순히 명령어를 순차적으로 나열하는 프로그래밍 기법을 뜻하지 않는다. 즉, 코드를 적절한 덩어리로 구분 지어 적당한 이름을 붙인 함수나 프로시저를 만들고, 이런 추상화를 바탕으로 더 상위의 함수를 작성해 나감으로써 좀 더 이해하기 쉽고 관리하기 쉬운 코드 기반을 만드는 프로시저 중심의 프로그래밍 패러다임이다. [↑](#footnote-ref-5)