# 2장. 코틀린 언어 기초

이 장에서는 코틀린 프로그램의 기본적인 문법 요소를 살펴보고, 변수를 정의하고 사용하는 법을 배운다. 수, 문자, 불(boolean) 값 등을 나타내는 코틀린 타입과 내장 연산을 이해하고, 문자열(string)이나 배열(array) 같은 복잡한 구조를 익힌다. 그 과정에서 자바 문법과 타입 시스템(type system)이 코틀린과 어떻게 다른지 살펴보고, 자바 구문과 타입을 코틀린으로 쉽게 바꿀 수 있다는 걸 알게 될 것이다.

## 구조

* 변수 정의
* 가변과 불변 변수
* 기본 식: 참조, 호출, 단항/이항 연산자
* 기본 타입과 연산: 수, 문자, 불 값
* 문자열 타입: 문자열 리터럴과 템플릿, 기본 문자열 연산
* 배열 타입: 배열 생성과 기본 연산

## 목표

기본 코틀린 타입을 이해하고 수, 불, 문자/문자열, 배열을 사용해 간단한 계산을 수행하는 프로그램을 작성하는 것이다.

## 기본 문법

이 장에서는 지역 변수(local variable)를 정의하는 방법과 참조, 함수 호출, 단항/이항 연산자 같은 코틀린 기본 식을 배운다.

## 주석

자바와 마찬가지로 코틀린은 세 가지 주석(comment)을 지원하며, 코드를 문서화할 때 사용한다.

* 한 줄짜리 주석: //로 시작하며 줄이 끝나면 주석도 끝난다.
* 여러 줄 주석: /\*로 시작하고 \*/로 끝난다.
* KDoc 여러 줄 주석: /\*\*로 시작하고 \*/로 끝난다.

KDoc 주석은 자바독(Javadoc)과 비슷한 리치 텍스트 문서(rich text documentation)를 생성하기 위해 사용한다.

<코드>

/\*  
여러 줄 주석  
/\* 주석 안에 내포된 주석 \*/  
\*/  
println(“Hello”) // 한 줄짜리 주석

</코드>

**자바와 코틀린의 차이**: 자바와 달리 코틀린에서는 여러 줄 주석을 여러 번 내포시킬 수 있다.

## 변수 정의하기

코틀린에서 변수를 정의하는 가장 간단한 형태는 다음과 같다.

<코드>

val timeInSeconds = 15

</코드>

이 정의를 이루는 요소들을 살펴보자.

* val 키워드(keyword): 값을 뜻하는 value에서 유래
* 변수 식별자(identifier): 새 변수에 이름을 부여하고, 나중에 이를 가리킬 때 사용한다
* 변수의 초깃값(initial value)을 정의하는 식: = 기호 뒤에 온다

**자바와 코틀린의 차이**: 변수 정의 뒤에 세미콜론(;)을 붙이지 않았음을 눈치챘는가? 실수가 아니다. 코틀린에서는 줄 끝에 세미콜론을 생략해도 된다. 실제로는 세미콜론을 쓰지 않는 스타일을 더 권장한다. 한 줄에 한 문장만 넣으면 실질적으로 코드에서 세미콜론을 거의 사용하지 않아도 된다.

<참고> IDE 팁

인텔리J는 불필요한 세미콜론을 경고함으로써 세미콜론을 사용하지 않는 코드 스타일을 권장한다.

</참고>

사용자에게 정수 두 개를 물어보고 두 수의 합을 표시하는 프로그램을 작성한다고 하자. 다음은 코틀린으로 작성한 프로그램이다.

<코드>

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

</코드>

이 코드가 하는 일을 더 자세히 살펴보자.

* readLine()은 프로그램에서 readLine을 실행하게 하는 호출 식(call expression)이다. readLine은 표준 입력(standard input)에서 한 줄을 읽어서 문자열로 반환해주는 표준 코틀린 함수다.
* !!는 널-아님 단언(not-null assertion)으로, readLine()의 결과가 널(null)인 경우 예외를 발생시킨다. 자바와 달리 코틀린은 어떤 타입이 널 값이 될 수 있는지를 추적하고, 널이 아닌 것이 확실하지 않은 값에 대해 toInt() 함수를 호출하지 못하게 막음으로써 널로 인한 오류를 방지한다. 여기서는 콘솔에서 값을 읽을 경우 readLine()이 널을 반환하지 않을게 확실하기 때문에 !!으로 널이 될 수 있을 가능성을 무시하게 만든다. **4장. 클래스와 객체 다루기**에서 널 가능성에 대해 더 자세히 다루겠다.
* readLine()의 결과에 대해 toInt() 함수를 호출한다. toInt()는 코틀린 String 클래스가 제공하는 메서드(method)로, 메서드가 호출된 대상 문자열을 정수 값으로 변환한다. 대상 문자열이 올바른 정수 값의 형태가 아니라면 toInt()는 런타임 오류(runtime error)를 내면서 프로그램을 종료시킨다. 지금은 모든 사용자 입력이 정상이라고 가정하고, 오류 처리는 다음 장까지 미뤄두자.
* toInt()를 호출한 결과를 같은 줄에서 정의 중인 a 변수에 저장한다.
* 마찬가지로 b라는 두 번째 변수를 정의하고, 사용자가 입력한 정수를 저장한다.
* 마지막으로 a + b로 두 수의 합을 계산한 다음, 결과를 println() 함수에 전달한다. println()은 인자로 받은 값을 표준 출력에 출력한다.

이 코드의 a와 b 같은 변수를 지역 변수라고 한다. 함수 안(이 경우는 main())에서 정의했기 때문이다. 이와 별도로 코틀린에서는 프로퍼티(property)라는 변수와 비슷한 값을 정의할 수 있다. 값을 그대로 읽거나 쓰는 변수와 달리, 프로퍼티에서는 값을 읽거나 쓸 때 계산을 수행할 수 있다. 예를 들어, 앞으로 보겠지만, 코틀린의 모든 문자열에는 문자열에 포함된 문자의 개수가 들어있는 length라는 프로퍼티가 있다.

자바에 익숙하다면 변수 타입을 지정하지 않았는데도 프로그램이 성공적으로 컴파일되고 실행된다는 사실을 알았을 것이다(그림 2-1). 이유는 타입 추론(type inference)이라는 기능 때문이다. 타입 추론은 대부분의 경우 컴파일러가 코드의 문맥에서 타입을 도출해주는 언어 기능이다. 여기서 컴파일러는 toInt() 함수가 Int 타입의 값을 반환한다는 사실을 알고 있고, 이 결과를 변수에 대입했기 때문에 변수 타입 또한 Int로 추론한다. 타입 추론 덕분에 코틀린은 강한 타입 지정(strongly typed) 언어인 동시에 사용자가 불필요한 타입 정보를 코드에 추가해서 코드가 지저분해지는 일을 막을 수 있다. 이 책의 여러 예를 통해 타입 추론이 얼마나 코틀린 프로그래밍을 단순화시켜 주는지 볼 수 있을 것이다.

**자바와 코틀린의 차이**: 자바도 자바 10부터 코틀린과 비슷한 지역 변수 타입 추론을 도입했다. 이 말은 이제 자바로 다음과 같은 코드를 작성할 수 있다는 뜻이다.

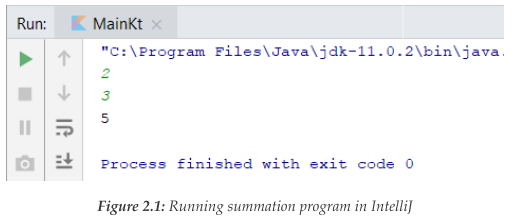
<코드>

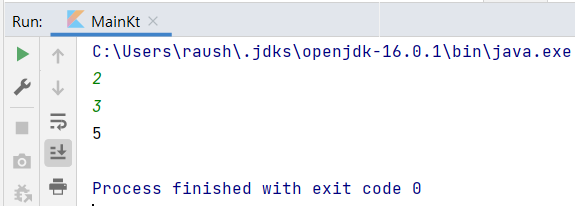
var text = “Hello”; // 자동으로 text의 타입을 String으로 추론한다

</코드>

하지만 코틀린은 지역 변수뿐 아니라 더 넓은 요소에 대해 타입 추론을 해준다. 이는 다음 장에서 살펴보겠다.

그림 2-1: 합계 프로그램을 인텔리J에서 실행하기





필요할 때는 타입을 명시해도 된다. 타입을 명시하려면 변수 이름 뒤에 콜론(:)을 표시하고, 그 뒤에 타입을 적는다.

<코드>

val n: Int = 100  
  
val text: String = “Hello!”

</코드>

이 경우 초깃값이 지정한 타입에 속해야 한다. 다음 코드는 컴파일 오류를 일으킨다.

<코드>

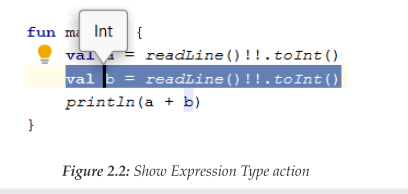
val n: Int = “Hello!” // Error: assigning String value to Int variable

</코드>

<참고>IDE 팁

인텔리J에서는 컴파일러가 식이나 값에 대해 추론한 타입을 볼 수 있다. 타입을 보려면 원하는 식을 편집창에서 선택하거나, 변수 이름에 캐럿(편집기 커서)을 위치시키고 [Ctrl] + [Shift] + [P]([Cmd] + [Shift] + [P])를 누른다.

그림 2-2: 식의 타입 보기



이를 바탕으로 간단한 액션으로 타입을 명시하거나 명시된 타입을 제거할 수 있다. 캐럿을 변수 식별자에 위치시키고 [Alt] + [Enter]를 누른 후, Specify Type explicitly(타입 명시)이나 Remove explicit type(명시된 타입 제거)를 선택하라(타입 제거는 식별자뿐 아니라 타입 지정에 위치시켜도 작동한다).

</참고>

초깃값을 생략하고 나중에 다른 문(statement)에서 변수를 초기화할 수도 있다. 이런 방식은 초깃값을 계산하는 식을 한 줄로 쓸 수 없는 경우 편리하다. 이 경우 변수 타입을 명시해야 한다.

<코드>

val text: String  
  
text = “Hello!”

</코드>

하지만 변수 값을 읽기 전에 변수를 초기화해야 한다는 사실에 주의하자. 변수를 사용하기 전에 변수를 초기화했는지 컴파일러가 확실히 알 수 없는 경우 컴파일 오류가 발생한다.

<코드>

val n: Int  
  
println(n + 1) // Error: variable n is not initialized

</코드>

## 식별자

식별자는 변수나 함수 등 프로그램에 정의된 대상에 붙은 이름이다. 코틀린 식별자는 두 가지로 구분된다. 첫 번째는 자바 식별자와 비슷하며 다음 규칙을 만족하는 임의의 문자열이다.

* 식별자는 오직 문자, 숫자, 밑줄 문자(\_)만 포함한다. 숫자로 식별자를 시작할 수는 없다.
* 밑줄로만 이뤄질 수도 있다. 하지만 이런 식별자(\_, \_\_, \_\_\_ 등)는 모두 미리 예약된 식별자이므로 일반적인 식별자로는 사용될 수 없다.
* 하드 키워드(hard keyword)를 식별자로 쓸 수는 없다.

하드 키워드(val, fun 등)는 코드 어디에 있든 키워드로 간주된다. 반면 소프트 키워드(import 등)는 특별한 문맥에서만 키워드로 간주되고, 그런 문맥이 아닌 경우에는 일반적인 식별자로 쓰일 수 있다. 하드 키워드와 소프트 키워드 목록은 Kotlin docs에서 볼 수 있다(kotlinlang.org/docs/reference/keyword-reference.html).

자바와 마찬가지로 문자나 숫자는 ASCII 문자, 숫자만을 의미하지 않고 유니코드 문자나 숫자도 포함한다. 하지만 영어 단어와 일반 숫자를 사용하는 쪽이 더 좋은 관습으로 여겨진다.

**자바와 코틀린의 차이**: 자바와 달리 코틀린 식별자에는 달러 기호($)를 쓸 수 없다.

두 번째 형태는 역작은따옴표(`)로 감싼 식별자로, 두 역작은따옴표 사이에는 빈 문자열을 제외한 아무 문자열이나 와도 된다.

<코드>

val `fun` = 1  
  
val `name with spaces` = 2

</코드>

이렇게 역작은따옴표 인용부호로 묶인 식별자 안에는 새줄문자(newline)나 역작은따옴표가 올 수는 없다. 또한, 플랫폼에 따른 식별자 기준을 만족해야 한다. 예를 들어 코틀린/JVM의 경우 JVM에 예약된 .;[ ]/<>:\는 역작은따옴표로 구분된 식별자 안에 올 수 없다.

가독성을 위해서 이런 기능을 가능한 피해야 한다. 이런 식별자를 허용하는 주된 이유는 자바에서 식별자로 사용할 수 있는 단어가 코틀린에서 키워드인데(예를 들어 fun은 코틀린에서는 키워드지만 자바에서는 아니다), 코틀린에서 자바에서 정의한 식별자를 가져와 써야 할 때가 있기 때문이다. 한 가지 다른 용례는 테스트케이스 메서드 이름을 정할 때이다. 이에 대해서는 **14장. 코틀린 테스팅**에서 살펴본다.

## 가변 변수

지금까지 살펴본 변수는 불변(immutable) 변수다. 불변 변수는 한번 초기화하면 다시는 값을 다시 대입할 수 없는 변수다. 이는 자바 final 변수와 비슷하다. 불변 변수를 사용하면 함수가 부수 효과(side effect)를 일으키지 못하고, 함수형 스타일 코드를 장려할 수 있으며, 이로 인해 코드에 대한 추론이 쉬워지기 때문에 가능하면 불변 변수를 많이 사용해야 한다.

하지만 필요한 경우 val 대신 var 키워드를 사용해 가변(mutable) 변수를 정의할 수 있다. 기본적인 문법은 불변 변수의 경우와 같다. 다만 원할 때 변수 값을 얼마든지 바꿀 수 있다는 점이 다르다. 변수 값을 바꿀 때 사용하는 = 연산을 대입(assignment)이라고 부른다. 대입 연산은 불변변수를 초기화할 때 사용했던 것이다.

<코드>

var sum = 1  
  
sum = sum + 2  
  
sum = sum + 3

</코드>

처음 변수에 값을 대입할 때 추론된 변수 타입은 변수가 불변이든 그렇지 않든 계속 유지된다는 점에 주의하자. 잘못된 타입의 값을 대입하면 컴파일 오류가 난다.

<코드>

var sum = 1  
  
sum = “Hello” // Error: assigning String value to Int variable

</코드>

추가로 코틀린은 복합대입연산(augmented assignment)이라는 대입과 +, -, \*, /, % 등의 이항(binary) 연산을 조합한 연산도 제공한다.

<코드>

var result = 3  
  
result \*= 10 // result = result \* 10  
  
result += 6 // result = result + 6

</코드>

이런 대입 연산은 복합대입연산의 대상이 되는 변수에 대해 이항 연산자가 쓰일 수 있을 때만 가능하다.

**자바와 코틀린의 차이**: 자바와 달리 코틀린 대입은 문(statement)이다. 따라서 아무 값도 돌려주지 않는다. 이로 인해 코틀린에서는 자바의 a=b=c와 같은 대입문 연쇄를 쓸 수 없다. 이런 식의 연쇄적인 대입문이 필요한 경우도 드물고, 쓰다가 실수를 하기도 쉽기 때문에 코틀린에서는 이런 대입을 금지한다. 복합대입연산도 마찬가지다.

변수 값을 변경할 수 있는 연산이 두 가지 더 있다. 바로 증가(++)와 감소(--) 연산이다. 이 연산의 가장 분명한 용례는 수 값을 1씩 증가시키거나 감소시키는 것이다. 자바와 마찬가지로 두 연산자를 전위(prefix)나 후위(postfix) 연산자로 쓸 수 있다.

<코드>

var a = 1  
println(a++) // a는 2, 1 출력됨  
println(++a) // a는 3, 3 출력됨  
println(--a) // a는 2, 2 출력됨  
println(a--) // a는 1, 2 출력됨

</코드>

위 예제를 보면 전위와 후위 연산이 모두 값을 변경하지만, 전위 연산의 결과는 변경된 이후의 새 값이고, 후위 연산의 결과는 변경되기 전 값이다.

## 식과 연산자

지금까지 사용한 코틀린 식을 다음과 같이 분류할 수 있다.

* 각 타입에 속하는 구체적인 값을 표현하는 리터럴(12, 3.56)
* 변수/프로퍼티 참조와 함수 호출(a, readLine(), "abc".length, "12".toInt())
* 전위와 후위 단항 연산(-a, ++b, c--)
* 이항 연산(a + b, 2 \* 3, x < 1)

모든 식은 정해진 타입이 있으며, 이 타입은 연산이 만들어내는 값의 범위와 값에 허용되는 연산을 결정한다. 예를 들어 1이라는 리터럴은 Int 타입이지만, readLine()!!이라는 식은 String 타입이다.

변수 참조와 함수 호출에 점(.)으로 구분된 수신객체(receiver) 연산을 포함시킬 수 있다는 점에 유의하라. 예를 들어 readLine()!!.toInt()는 readLine()!!이 만들어낸 값의 문맥에서 String에(이 타입은 readLine()!!의 결과 타입이다) 정의된 toInt()라는 함수를 호출한다.

단항과 이항 연산마다 연산 순서를 결정하는 우선순위(precedence)가 정해져 있다. 예를 들어 2 + 3 \* 4라는 식에서는 3 \* 4를 먼저 계산한 후 그 결과에 2를 더해서 14를 얻는다. 이 순서를 바꾸려면 괄호를 사용한다. (2 + 3) \* 4는 5 \* 4와 같아서 20으로 계산된다. 이 장에서 살펴볼 연산자의 우선순위를 다음 표에 정리했다.

표 2-1: 연산자 우선순위

|  |  |  |  |
| --- | --- | --- | --- |
| 분류 | 연산자 | 예제 | 우선순위를 감안한 해석 |
| 후위 | ++ -- | a\*b++ | a\*(b++) |
|  |  | ++b-- | ++(b--) |
|  |  | a\*b.foo() | a\*(b.foo()) |
| 전위 | + - ++ -- ! | +a\*b | (+a)\*b |
|  |  | ++a\*b | (++a)\*b |
|  |  | !a || b | (!a) || b |
| 곱셈 | \*, /, % | a\*b + c | (a\*b) + c |
|  |  | a – b%c | a - (b\*c) |
| 덧셈 | +, - | a + b and c | (a + b) and 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) |
| 비교 | < > <= >= | a < b == b < c | (a < b) == (b < c) |
|  |  | a < b && b < c | (a < b) && (b < c) |
| 동등 | == != | a == b || b != c | (a == b) || (b != c) |
| 논리곱(conjunction) | && | a || b && c | a || (b && c) |
| 논리합(disjunction) | || | a && b || c | (a && b) || c |
| 대입 | = += -= \*= /= %= | a = b\*c | a = (b\*c) |
|  |  | a \*= a + b | a \*= (a + b) |

|  |  |  |
| --- | --- | --- |
| 분류 | 연산자 | 예제 |
| 후위 | ++ -- | a\*b++ // a\*(b++) |
|  |  | ++b-- // ++(b--) |
|  |  | a\*b.foo() // a\*(b.foo()) |
| 전위 | + - ++ -- ! | +a\*b // (+a)\*b |
|  |  | ++a\*b // (++a)\*b |
|  |  | !a || b // (!a) || b |
| 곱셈 | \*, /, % | a\*b + c // (a\*b) + c |
|  |  | a – b%c // a - (b\*c) |
| 덧셈 | +, - | a + b and c // (a + b) and 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) |
| 비교 | < > <= >= | a < b == b < c // (a < b) == (b < c) |
|  |  | a < b && b < c // (a < b) && (b < c) |
| 동등 | == != | a == b || b != c // (a == b) || (b != c) |
| 논리곱(conjunction) | && | a || b && c // a || (b && c) |
| 논리합(disjunction) | || | a && b || c // (a && b) || c |
| 대입 | = += -= \*= /= %= | a = b\*c // a = (b\*c) |
|  |  | a \*= a + b // a \*= (a + b) |

우선 순위가 같은 이항 연산자는 왼쪽에서 오른쪽으로 순서대로 계산된다.

<코드>

a.foo().bar() // (a.foo()).bar()  
a \* b % c // (a \* b) % c  
(a == 1) or (b < 1) and (c > 1) // ((a == 1) or (b < 1)) and (c > 1)

</코드>

이 책의 뒷부분에서 이 표를 더 세분화하는 연산을 그때그때 소개하겠다.

## 기본 타입

이 절에서는 수, 문자, 불 값 등을 표현하는 코틀린 타입을 살펴본다. 자바에 익숙하다면 이런 타입을 자바 원시 타입(primitive type)이라고 생각해도 되지만, 이 대응이 완벽하지는 않다. 자바에서는int와 같은 원시 타입(원시타입의 값은 메서드의 스택 영역에 저장될 수 있다)과 String 같이 클래스를 기반으로 하는 참조 타입(이런 참조 타입의 값은 동적으로 힙에 할당된 메모리를 가리키는 참조일 뿐이다) 사이에 명확한 구분이 있었다. 코틀린에서는 똑같은 타입(예: Int)이 문맥에 따라 원시 타입과 참조 타입을 가리키기 때문에, 이런 구분이 약간 모호하다. 자바에는 원시 타입을 감싸는 특별한 박싱 타입(boxing type)이 있지만 코틀린은 필요할 때 암시적으로 박싱을 수행한다.

**자바와 코틀린의 차이**: 자바와 달리 모든 코틀린 타입은 근본적으로 어떤 클래스 정의를 기반으로 만들어진다. 이 말은 Int와 같이 원시 타입과 비슷한 타입들도 메서드와 프로퍼티를 제공한다는 뜻이다. 예를 들어 1.5.toInt()라는 식으로 1.5에 대해 Double 타입에 정의된 toInt() 메서드를 호출할 수 있다. 이 메서드는 Double 값을 Int 값으로 변환해준다.

타입을 하위타입(subtype)이라는 개념으로 계층화할 수 있다. A타입이 B타입의 하위타입이라는 말은 근본적으로 B 타입의 값이 쓰일 수 있는 모든 문맥에 A 타입의 값을 넣어도 아무 문제가 없다는 뜻이다. 예를 들어 널을 허용하지 않는 모든 코틀린 타입은 Any라는 내장 타입의 직간접적인 하위타입이다. 따라서 다음 코드는 1이라는 값을 박싱하게 만든다.

<코드>

val n: Any = 1 // Ok: Int는 Any의 하위타임

</코드>

### 정수 타입

코틀린에는 정수를 표현하는 네 가지 기본 타입이 있다(표 2-2).

표 2-2: 정수 타입

|  |  |  |  |
| --- | --- | --- | --- |
| 이름 | 크기(바이트) | 범위 | 대응하는 자바 타입 |
| Byte | 1 | -128 .. 127 | byte |
| Short | 2 | -32768 .. 32767 | short |
| Int | 4 | -231 .. 231-1 | int |
| Long | 8 | -263 .. 263-1 | long |

어떤 정수 타입 값을 표현하는 가장 간단한 리터럴은 십진수다.

<코드>

val n = 12345

</코드>

코틀린 1.1부터는 자바 7+와 마찬가지로 수 리터럴에 \_를 넣어서 가독성을 높일 수 있다. 리터럴이 아주 큰 수를 나타낼 때 \_가 유용하다.

<코드>

val n = 34\_721\_189

</코드>

리터럴 자체의 타입은 그 수를 2진수로 표현했을 때의 크기에 따라 Int일 수도 Long일 수도 있다. 하지만 범위만 맞으면 수 리터럴을 크기가 더 작은 타입의 변수에 대입해도 된다.

<코드>

val one: Byte = 1 // OK  
val tooBigForShort: Short = 100\_000 // Error: too big for Short   
val million = 1\_000\_000 // OK: Int로 타입이 추론됨  
val tooBigForInt: Int = 10\_000\_000\_000 // Error: too big for Int   
val tenBillions = 10\_000\_000\_000 // OK: Long으로 타입이 추론됨  
val tooBigForLong = 10\_000\_000\_000\_000\_000\_000 // Error: too big for Long

</코드>

리터럴에 L이나 l을 접두사로 붙이면 Long 타입이 된다.

<코드>

val hundredLong = 100L // OK: Long으로 타입이 추론됨  
val hundredInt: Int = 100L // Error: assigning Long to Int

</코드>

앞에 0b(2진수)나 0x(16진수)를 붙여서 2진수나 16진수로 수 리터럴을 작성할 수도 있다.

<코드>

val bin = 0b10101 // 21  
val hex = 0xF9 // 249

</코드>

수 리터럴의 경우 0을 표현하는 경우가 아니라면 맨 앞에 0이 올 수 없음에 유의하라. 일부 프로그래밍 언어(자바도 포함)는 0으로 시작하는 수 리터럴을 8진수로 사용하는데, 8진수가 쓸모 있는 경우는 드물고 잘못 사용하기도 쉽기 때문에 코틀린은 이를 지원하지 않는다. 이로 인해 8진수 표기법에 익숙한 개발자들의 혼동을 막기 위해 0이 앞에 붙은 수 리터럴이 금지됐다.

<코드>

val zero = 0 // OK  
val zeroOne = 01 // Error

</코드>

-10과 같은 음수는 기술적으로는 리터럴이 아니다. 음수는 단항 음수 연산자(-)를 리터럴에 적용한 식이다[[1]](#footnote-1).

<코드>

val neg = -10  
val negHex = -0xFF

</코드>

각 정수 타입에는 최솟값(MIN\_VALUE)과 최댓값(MAX\_VALUE)을 포함하는 상수 정의가 들어있다. 이런 상수를 사용하려면 앞에 타입 이름을 붙여야 한다.

<코드>

Short.MIN\_VALUE // -32768  
Short.MAX\_VALUE // 32767  
Int.MAX\_VALUE + 1 // -2147483648(정수 오버플로(overflow))

</코드>

### 부동소수점 수

자바와 마찬가지로 코틀린도 IEEE 754 부동소수점 수(floating point number)를 따르는 Float와 Double을 제공한다. 이 둘은 각각 자바의 float와 double에 대응한다.

부동소수점 수 리터럴 중 가장 단순한 형태는 십진 소수 형태로, 정수 부분과 소수 부분을 나눠 소수점(.) 찍어 놓았다.

<코드>

val pi = 3.14  
val one = 1.0

</코드>

정수 부분이 비어있는 경우 정수 부분을 0으로 간주한다. 하지만 소수점을 남기면서 소수 부분을 생략할 수는 없다.

<코드>

val quarter = .25 // 0.25  
val one = 1. // Error: Expecting an element  
val two = 2 // 오류는 아니지만, 정수 리터럴임

</코드>

코틀린은 과학적 표기법(scientific notation) 리터럴을 허용한다. 과학적 표기법에서는 e나 E뒤에 10을 몇 번 거듭제곱하는지를 알려주는 숫자가 온다.

<코드>

val pi = 0.314E1 // 3.14 = 0.314\*10  
val pi100 = 0.314E3 // 314.0 = 0.314\*1000  
val piOver100 = 3.14E-2 // 0.0314 = 3.14/100  
val thousand = 1E3 // 1000.0 = 1\*1000

</코드>

과학적 표기법에서는 소수 부분(소수점 포함)을 생략할 수 있다.

**자바와 코틀린의 차이**: 자바 6+와 달리 코틀린에서는 Double이나 Float의 16진 리터럴을 지원하지 않는다.

디폴트로 부동소수점 리터럴은 Double 타입이다. f나 F를 리터럴 뒤에 붙이면 Float 타입이 된다(f나 F를 붙이는 경우 소수점을 포함한 소수 부분을 생략할 수 있다).

<코드>

val pi = 3.14f  
val one = 1f

</코드>

**자바와 코틀린의 차이**: 자바에서는 D나 d를 부동소수점 리터럴 뒤에 붙여서 강제로 double 타입으로 만들 수 있다(예: 1.25d). 하지만 코틀린에서는 이런 접미사를 허용하지 않으며, 부동소수점 리터럴의 타입은 디폴트로 Double이다.

Float 리터럴이 Double 타입으로 자동 변환되지 않는다는 점에 유의하자. 다음 코드는 컴파일 오류를 낸다.

<코드>

val pi: Double = 3.14f // Error

</코드>

Float와 Double도 각 타입의 특별한 값을 표현하는 몇 가지 상수를 제공한다.

* MIN\_VALUE, MAX\_VALUE: 각 타입에서 표현할 수 있는 가장 작은 유한값과 가장 큰 유한값
* NEGATIVE\_INFINITY, POSITIVE\_INFINITY: 음의 무한대와 양의 무한대. 각 타입이 표현할 수 있는 가장 작은 값과 가장 큰 값
* NaN: 0/0의 결과처럼 숫자가 아닌 값을 의미

<코드>

println(Float.MIN\_VALUE) // 1.4E-45  
println(Double.MAX\_VALUE) // 1.7976931348623157E308  
println(Double.POSITIVE\_INFINITY) // Infinity  
println(1.0/Double.NEGATIVE\_INFINITY) // -0.0  
println(2 - Double.POSITIVE\_INFINITY) // -Infinity  
println(3 \* Float.NaN) // NaN

</코드>

### 산술 연산

모든 수 타입은 기본 산술 연산을 지원한다.

표 2-3: 산술 연산

|  |  |  |
| --- | --- | --- |
| 연산 | 뜻 | 예제 |
| +(단항) | 원래 값과 같은 값 | +2 // 2 |
| -(단항) | 원래 값의 부호를 반전한 값 | -2 // -2 |
| + | 덧셈 | 2 + 3 // 5 |
| 2.5 + 3.2 // 5.7 |
| - | 뺄셈 | 1 – 3 // -2 |
| 3.4 – 1.8 // 1.6 |
| \* | 곱셈 | 3 \* 4 // 12 |
| 3.5 \* 1.5 // 5.25 |
| / | 나눗셈 | 7/4 // 1 |
| -7/4 // -1 |
| 7/(-4) // -1 |
| (-7)/(-4) // 1 |
| 6.5/2.5 // 2.6 |
| -6.5/2.5 // -2.6 |
| 6.5/(-2.5) // -2.6 |
| (-6.5)/(-2.5) // 2.6 |
| % | 나머지 | 7%4 // 3 |
| -7%4 // -3 |
| 7%(-4) // 3 |
| (-7)%(-4) // -3 |
| 6.5%2.5 // 1.5 |
| -6.5%2.5 // -1.5 |
| 6.5%(-2.5) // 1.5 |
| (-6.5)%(-2.5) // -1.5 |

산술 연산의 동작은 자바와 같다. 정수 나눗셈 연산자가 항상 0쪽으로 내림한 결과를 내놓는 반면, 나머지 연산의 결과는 항상 피제수와 같은 부호라는 사실에 유의하자. 부동소수점 수 연산은 IEEE 754 명세를 따른다.

참고로, 코틀린 1.5부터 표준 라이브러리에 정수 floorDiv()와 mod() 메서드가 추가됐다. floorDiv()는 정수 나눗셈(/) 연산과 마찬가지로 한 수로 다른 수로 나눈 몫을 계산하고, mod() 메서드는 정수 나머지(%) 연산과 마찬가지로 한 수를 다른 수로 나눈 나머지를 계산한다. 어떤 정수 a 를 0이 아닌 다른 정수 b로 나누면, floorDiv()는 몫을 더 작은 정수로 내림하고, mod()는 a와 a.floorDiv(b)\*b의 차이를 반환한다. 이렇게 계산한 나머지 값의 부호는 항상 b의 부호와 같다.

<코드>

println(7.floorDiv(4)) // 1

println((-7).floorDiv(4)) // -2

println(7.floorDiv(-4)) // -2

println((-7).floorDiv(-4)) // 1

println(7.mod(4)) // 3

println((-7).mod(4)) // 1

println(7.mod(-4)) // -1

println((-7).mod(-4)) // -3

</코드>

수 타입은 값을 1씩 증가/감소시키는 ++/-- 연산도 지원한다.

단항 +/- 연산의 결과는 인자들의 타입과 같다. 다만 Byte와 Short의 경우에는 Int를 내놓는다.

<코드>

val byte: Byte = 1  
val int = 1  
val long = 1L  
val float = 1.5f  
val double = 1.5  
-byte // -1: Int  
-int // -1: Int  
-long // -1: Long  
-float // -1.5: Float  
-double // -1.5: Double

</코드>

각 이항 산술 연산마다 모든 가능한 수 타입 간의 연산을 지원하기 위한 변종이 함께 제공된다. 수 타입이 6가지이기 때문에 각 연산마다 6 \* 6 = 36가지 버전이 있다는 뜻이다. 이로 인해 산술 연산에서 명시적인 타입 변환을 수행하지 않아도 여러 수 타입을 혼용할 수 있다. 이러한 산술 연산의 결과는 인자 중 더 큰 의미를 지니는 타입이 된다.

<코드>

Double > Float > Long > Int > Short > Byte

</코드>

대부분의 타입에서 이는 해당 타입이 표현할 수 있는 값의 범위가 더 넓다는 뜻이다. 하지만 항상 그렇지는 않다. 분명한 예로 Long에서 Float로 변환되는 경우를 들 수 있다. 이 경우 일부 유효자리 숫자를 잃고 정밀도(precision)가 떨어지게 된다.

이로부터 다음을 알 수 있다.

<코드>

byte + byte // 2: Byte  
int + byte // 2: Int  
int + int // 2: Int  
int + long // 2: Long  
long + double // 2.5: Double  
float + double // 3.0: Double  
float + int // 2.5: Float  
long + double // 2.5: Double

</코드>

### 비트 연산

Int와 Long은 비트 수준의 연산을 지원한다.

표2.4: 비트 연산

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 연산 | 뜻 | 예제 | 결과 | 해당하는 자바 연산 |
| shl | 왼쪽 시프트(shift) | 13 shl 2  (-13) shl 2 | 52: 0...00110100  -52: 1...11001100 | << |
| shr | 오른쪽 시프트 | 13 shr 2  (-13) shr 2 | 3: 0...00000011  -4: 1...11111100 | << |
| ushr | 부호없는 오른쪽 시프트 | 13 ushr 2  (-13) ushr 2 | 3: 0...00000011  1073741820:  001...111100 | >>> |
| and | 비트 곱(AND) | 13 and 19  -13 and 19 | 1: 0...00000001  19: 0...00010011 | & |
| or | 비트 합(OR) | 13 or 19  -13 or 19 | 31: 0...00011111  -13: 1...11110011 | | |
| xor | 비트 배타합(XOR) | 13 xor 19  -13 xor 19 | 30: 0...00011110  -32: 1...11100000 | ^ |
| inv | 비트 반전(inversion) | 13.inv()  (-13).inv() | -14: 1...11110010  12: 0...00001100 | ~ |
| 비고: 각 수의 2진 표현은 다음과 같다.  13: 0...00001101  -13: 1...11110011  19: 0...00010011 | | | | |

inv는 이항 연산이나 단항 연산이 아니라 점 표기법으로 호출하는 메서드일 뿐이라는 점에 유의하라.

코틀린 1.1부터는 Byte와 Short에도 or, xor, inv를 호출할 수 있다.

**자바와 코틀린의 차이**: 자바에 익숙하다면 비트 연산자 &, |, ^, ~, <<, >>, >>>를 알 텐데, 코틀린은 이런 연산자를 지원하지 않는다. 대신 이에 대응하는 and, or, xor, inv, shl, shr, ushr을 사용해야 한다.

## 문자 타입 Char

Char 타입은 유니코드 한 글자를 표현하며 16비트다[[2]](#footnote-2). 이 타입의 리터럴은 작은 따옴표(') 사이에 문자를 넣으면 된다.

<코드>

val z = 'z'  
val alpha = 'α'

</코드>

새줄 문자와 같은 특수 문자를 위해 코틀린은 이스케이프(escape)를 제공한다. \t는 탭(tab), \b는 백스페이스(backspace), \n은 새줄(newline), \r은 캐리지 리턴(carriage return), \’는 작은 따옴표(single quote), \”는 큰 따옴표(double quote), \\은 역슬래시(backslash), \$는 달러표시(dollar sign)다.

<코드>

val quote = '\''  
val newLine = '\n'

</코드>

\u 다음에 4자리 16진수를 넣는 시퀀스를 사용해 임의의 유니코드 문자를 문자 리터럴에 넣을 수도 있다.

<코드>

val pi = '\u03C0' // π

</코드>

내부에서 Char 값은 그냥 문자 코드일 뿐이지만, 코틀린은 Char 자체를 수 타입으로 취급하지는 않는다. 하지만 유니코드 문자 집합 내에서의 산술 연산을 몇 가지 허용한다. 다음은 Char로 할 수 있는 연산이다.

* +/- 연산자를 사용해 문자에 수를 더하거나 뺄 수 있다. 더하거나 뺀 수만큼 코드포인트가 이동한 새 문자를 반환한다.
* 두 문자를 빼면 두 문자의 코드포인트 사이의 거리를 얻을 수 있다.
* 문자를 ++나 --로 증가시키거나 감소시킬 수 있다(의미는 +나 -로 1을 더하거나 뺄 때와 같다).

몇 가지 예를 살펴보자.

<코드>

var a = 'a'  
var h = 'h'  
/\* 'a'보다 뒤 5번째 글자 \*/ println(a + 5) // f  
/\* 'a'보다 앞 5번째 글자 \*/ println(a – 5) // \  
/\* 'a'와 'h' 사이의 거리 \*/ println(h – a) // 7  
/\* 'h' 바로 앞 글자 \*/ println(--h) // g  
/\* 'a' 바로 뒤 글자 \*/ println(++a) // b

</코드>

**자바와 코틀린의 차이**: 자바에서는 문자(char)에 대한 산술 연산 결과가 암시적으로 정수로 변환된다. 반면 코틀린에서 Char에 대한 연산은(두 문자의 차이는 제외) Char를 결과로 돌려준다.

### 수 변환

각 수 타입마다 값을 다른 수 타입으로 변환하는 연산이 정의되어 있다. 연산 이름은 변환하려는 목적 타입을 알기 쉬운 이름으로 되어 있다. 예를 들어 toByte(), toShort(), toInt(), toLong(), toFloat(), toDouble(), toChar() 등이다. Char 값에 대해서도 같은 범위의 연산을 제공한다.

**자바와 코틀린의 차이**: 자바와 달리 코틀린에서는 범위가 큰 타입이 사용돼야 하는 문맥에 범위가 작은 타입을 쓸 수 없다. 예를 들어 Int 값을 Long 변수에 대입할 수 없다. 다음 코드는 컴파일 오류를 발생시킨다.

<코드>

val n = 100 // Int  
val l: Long = n // Error: can’t assign Int to Long

</코드>

이렇게 하게 된 이유는 앞에서 언급한 암시적인 박싱 때문이다. 일반적인 Int 값(또는 다른 수 타입의 값)이 꼭 원시 타입의 값으로 표현된다는 보장이 없다. 따라서 방금 본 코드와 같은 더 큰 범위의 타입으로 변환하는 경우 다른 박싱한 타입의 값을 만들어낼 수 있는 가능성이 생기고, 이로 인해 동등성(equality) 요구 조건을 만족시키지 못하게 되면서 미묘한 오류를 발생시킬 수 있다. 위 코드를 올바른 코드로 인정하면 다음 연산이 false를 출력할 것이다.

<코드>

println(l == n)

</코드>

하지만 이런 결과를 프로그래머가 예상하기는 힘들다. 자바에서도 박싱한 타입과 관련해 비슷한 문제가 있다.

<코드>

// 자바 코드  
Integer n = 100;  
Long l = n; // Error: can’t assign Integer to Long

</코드>

정수 타입 사이의 변환은 대상 타입이 더 큰 범위를 담는 타입인 경우 손실이 없이 수행된다. 그렇지 않은 경우 MSB(2진수로 표현했을 때 상위 비트 쪽)를 잘라내고 나머지를 대상 타입의 값으로 변환한다. Char 타입을 다른 타입의 값으로 바꾸거나 다른 타입의 값을 Char 타입으로 바꾸는 경우도 그렇다.

<코드>

val n = 945  
println(n.toByte()) // -79  
println(n.toShort()) // 945  
println(n.toChar()) // α  
println(n.toLong()) // 945

</코드>

부동소수점 수 타입과 관련된 변환의 경우, 일반적으로 대상 타입과 무관하게 정밀도를 잃을 수 있다. 예를 들어 Long을 Float로 변환하면 LSB쪽(2진수로 표현했을 때 하위 비트 쪽)을 잃어버릴 수 있다. 부동소수점 수를 정수로 변환하는 연산은 기본적으로 0쪽으로 어림(rounding to zere[[3]](#footnote-3))하는 연산이다.

<코드>

println(2.5.toInt()) // 2  
println((-2.5).toInt()) // -2  
println(1\_000\_000\_000\_000.toFloat().toLong()) // 999999995904

</코드>

### 불 타입과 논리 연산

코틀린은 참(true)이나 거짓(false) 중 하나로 판명되는 불(Boolean) 타입과 논리 연산을 제공한다.

<코드>

val hasErrors = false;  
val testPassed = true;

</코드>

자바와 마찬가지로 코틀린 Boolean도 수 타입과는 다른 타입이며, 암시적으로든 toInt() 등의 명시적인 내장 연산을 써서든 수로 변환할 수 없다(반대로 수를 Boolean 타입으로 변환할 수도 없다). 개발자는 비교 연산자나 조건식(if나 when등을 뜻하며, 나중에 설명한다)을 사용해 불이 아닌 값에서 불 값을 만들어야 한다.

불이 지원하는 연산은 다음과 같다.

* !: 논리 부정
* or, and, xor: 즉시 계산(eager) 방식의 논리합, 논리곱, 논리배타합
* ||, &&: 지연계산(lazy) 방식의 논리합, 논리곱

지연 계산 연산자(||, &&)는 자바의 동일 연산자와 똑같은 의미를 제공한다. ||의 왼쪽 피연산자가 참이면 오른쪽 피연산자를 계산하지 않는다. &&의 왼쪽 피연산자가 거짓이면 오른쪽 피연산자를 계산하지 않는다. 이런 방식의 계산은 왼쪽 피연산자에 부수 효과가 포함된 경우 쓸모가 있을 수 있다.

**자바와 코틀린의 차이**: 자바와 달리 코틀린은 &와 | 연산자를 제공하지 않는다. and와 or가 각각 &와 |를 대신한다.

동등성/비동등성 연산자 ==와 !=를 사용한 예를 살펴보자(이 두 연산자에 대해서는 다음 절에서 자세히 다룬다).

<코드>

println((x == 1) or (y == 1)) // true  
println((x == 0) || (y == 0)) // false  
println((x == 1) and (y != 1)) // true  
println((x == 1) and (y == 1)) // false  
println((x == 1) xor (y == 1)) // true  
println((x == 1) xor (y != 1)) // false  
println(x == 1 || y/(x - 1) != 1) // true  
println(x != 1 && y/(x - 1) != 1) // false

</코드>

마지막 두 예에서는 지연 계산이 필수적이다. x==1일 때 오른쪽 연산을 계산하면 0으로 나누는 오류가 발생하기 때문이다.

지연 계산과 즉시 계산 논리곱과 논리합의 차이를 기억하라. 즉시 계산인 and, or, xor은 이름이 붙은 중위 연산자와 같은 우선순위이기 때문에 && 연산이나 || 연산보다 더 우선하여 계산된다. 예를 들어 다음 식은,

<코드>

a || b and c or d && e

</코드>

다음과 같이 계산된다.

<코드>

a || (((b and c) or d) && e)

</코드>

의심스러운 경우에는 괄호를 써서 작성한 코드의 의미를 명확히 하는 편을 추천한다.

### 비교와 동등성

지금까지 언급한 모든 타입은 몇 가지 비교 연산을 제공한다. ==(같다), !=(같지 않다), <(보다 작다), <=(보다 작거나 같다), >(보다 크다),>=(보다 크거나 같다)가 비교 연산이다.

<코드>

val a = 1  
val b = 2  
println(a == 1 || b != 1) // true  
println(a >= 1 && b < 3) // true  
println(a < 1 || b < 1) // false  
println(a > b) // false

</코드>

일반적으로 동등성 연산인 ==와 !=를 모든 타입의 값에 적용할 수 있다. 하지만 수 타입이나 Char와 Boolean의 경우 예외가 있다. 다음 코드를 보라.

<코드>

val a = 1 // Int  
val b = 2L // Long  
println(a == b) // Error: comparing Int and Long  
println(a.toLong() == b) // Ok: 두 타입 모두 Long임

</코드>

기본적으로 이런 코틀린 타입은 두 인자가 모두 같은 타입일 때만 ==와 !=를 허용한다. 예를 들어 한 인자가 Int이고 다른 인자가 Long이면 ==를 적용할 수 없다. 값이 박싱되어 있는지에 따라 동등성 연산이 다른 결과를 낳을 수 있는데, 코틀린에서는 박싱이 암시적으로 진행되기 때문에 (자바 원시 타입에 해당하는 박싱이 될 수 있는 코틀린) 타입 사이의 동등성 연산을 허용하면 혼란을 야기할 수 있기 때문이다.

하지만 모든 수 타입의 값은 서로 <, <=, >, >=를 사용해 비교할 수 있다. 이는 수 타입 사이의 산술 연산이 가능한 모든 경우를 다룰 수 있도록 오버로딩(overloading)된 것과 마찬가지다. 따라서 다음과 같은 코드를 작성할 수 있다[[4]](#footnote-4).

<코드>

1 <= 2L || 3 > 4.5

</코드>

Char와 Boolean 값도 비교 연산을 지원한다. 하지만 같은 타입의 값과만 비교할 수 있다.

<코드>

false == true // false  
false < true // true  
false > 1 // Error: comparing Boolean and Int  
‘a’ < ‘b’ // true  
‘a’ > 0 // Error: comparing Int and Char

</코드>

false가 true보다 작다고 가정하고, Char는 문자의 코드값에 따라 비교된다는 점을 기억하라.

**자바와 코틀린의 차이**: 자바에서는 박싱된 값과 언박싱된 값이 서로 다른 타입으로 표현됐고(예: long과 Long), 원시 타입의 값을(char도 포함) 자유롭게 서로 ==, !=나 <, <=, >, >=로 비교할 수 있었다. 하지만, 자바의 불 값은 순서(대소관계)가 없기 때문에 두 불 값에 대해서는 동등성 비교만 적용할 수 있다.

부동소수점 타입 비교 연산은 IEEE 754 표준을 따른다. 특히 이 표준은 NaN 값을 특별히 취급한다.

println(Double.NaN == Double.NaN) // false  
println(Double.NaN != Double.NaN) // true  
println(Double.NaN<= Double.NaN) // false  
println(Double.NaN<Double.POSITIVE\_INFINITY) // false  
println(Double.NaN>Double.NEGATIVE\_INFINITY) // false

기본적으로 NaN은 그 어떤 값과도 같지 않다[[5]](#footnote-5). 특히 다른 NaN과도 같지 않고, 무한대를 포함한 다른 어떤 값보다 작지도 않고 크지도 않다.

하지만 이 규칙은 컴파일러가 정적으로(statically) 관심 대상의 타입이 부동소수점 타입이라는 사실을 알 때만 적용된다[[6]](#footnote-6). 더 일반적인, 예를 들어 수를 컬렉션(collection)에 넣는 것과 같은 경우에는 컴파일러가 동등성 연산자를 사용해 박싱한 타입을 처리하는 규칙을 사용한다. JVM에서 이는 원시 타입의 값을 감싼 Double과 Float 래퍼 타입의 인스턴스를 비교한다는 뜻이다.

<코드>

val set = sortedSetOf(Double.NaN, Double.NaN,Double.POSITIVE\_INFINITY, Double.NEGATIVE\_INFINITY,0.0)  
println(set) // [-Infinity, 0.0, Infinity, NaN]

</코드>

위 코드는 원소 타입의 자연적인 순서(natural ordering)로 정렬된 트리(JVM에서는 기본적으로 TreeSet이다)를 만들고 원소를 출력한다. 출력은 이런 경우에 다음과 같은 사실이 성립한다는 점을 보여준다.

* NaN은 자기 자신과 같다. 집합에 NaN이 단 하나만 추가됐기 때문이다.
* NaN은 Double에서 가장 큰 값으로 취급된다(심지어 양의 무한대보다도 크다).

앞으로 다룰 다른 장에서 동등성과 순서에 대해 더 자세히 살펴볼 것이다.

## 문자열

String 타입은 문자들로 이뤄진 문자열을 표현한다. 자바와 마찬가지로 코틀린 문자열도 불변이다. 따라서 String 객체를 만들고 나면 그 안의 문자를 변경할 수 없고 문자열을 읽기만 할 수 있으며, 문자를 바꾸고 싶으면 기존 문자열을 바탕으로 새로운 문자열을 만들어야 한다. 이 절에서는 새로운 문자열을 만드는 방법과 문자열을 조작하는 기본적인 방법을 살펴본다.

### 문자열 템플릿

문자열 리터럴을 정의하는 가장 간단한 방법은 앞에서 본 것처럼 자바와 마찬가지로 큰따옴표(")로 문자열을 감싸는 것이다.

<코드>

val hello = “Hello, world!”

</코드>

문자열에 새줄문자 같은 특수 문자가 들어가면 이스케이프 시퀀스를 사용해야 한다(Char 타입 절을 보라).

<코드>

val text = “Hello, world!\nThis is \”multiline\” string”  
println("\u03C0 \u2248 3.14") // π ≈ 3.14

</코드>

이런 리터럴은 기본적으로 자바 문자열과 같다. 코틀린은 이와 더불어 여러 가지 식에서 문자열을 합성해내는 훨씬 더 강력한 방법을 지원한다. 예를 들어 사용자를 환영하면서 현재 날짜와 시간을 표시하고 싶다고 하자.

<코드>

import java.util.Date[[7]](#footnote-7)  
fun main() {  
 val name = readLine()  
 println("Hello, $name!\n Today is ${Date()}")  
}

</코드>

기본적으로 ${}의 중괄호 사이에 넣기만 하면, 어떤 올바른 코틀린 식이든 문자열에 넣을 수 있다. 위 예제의 $name과 같이 이 식이 간단한 변수 참조인 경우에는 중괄호를 생략하고 달러 기호만 붙여도 된다. 이런 기능을 문자열 템플릿(string template)이라고 한다.

문자열 템플릿 안의 식은 어떤 값이든 될 수 있다. 이 값은 자동으로 모든 코틀린 타입이 제공하는 toString() 메서드를 통해 문자열로 변환된다.

이 프로그램을 실행하고 이름(예: John)을 입력하면 다음과 같이 출력된다.

<코드>

Hello, John!  
Today is Sat Sep 04 00:42:29 KST 2021

</코드>

결과는 여러분의 로케일(locale)에 따라 다르다.

첫 줄에서 사용한 import 문은 코드에서 JDK의 java.util.Date라는 클래스의 전체 이름을 사용하지 않고 Date라는 이름으로 참조할 수 있게 해준다. 다음 장에서 임포트와 패키지(package)에 대해 더 자세히 다루겠다.

한 가지 다른 문자열 유형으로 로우 문자열(raw string)이 있다. 로우 문자열을 사용하면 이스케이프 시퀀스를 사용하지 않고도 문자열을 작성할 수 있다. 이러한 리터럴은 큰따옴표 3개로 둘러 쌓여 있고, 새줄문자를 포함한 임의의 문자를 포함할 수 있다.

<코드>

val message = """  
 Hello, $name!  
 Today is ${Date()}  
""".trimIndent()

</코드>

trimIndent()는 여러 줄에 공통된 최소 들여쓰기(indent)를 제거해주는 표준 코틀린 함수다.

드물겠지만, 로우 문자열에 특수 문자를 추가하고 싶은 경우에는 ${}안에 특수 문자를 넣으면 된다.

<코드>

val message = """  
This is triple quote:’${"\"\"\""}’  
""".trimIndent()

</코드>

JVM으로 컴파일되는 애플리케이션에서 문자열은 JVM의 String 클래스로 표현된다.

### 기본 문자열 연산

모든 String 인스턴스는 문자열에 든 문자 수를 표현하는 length와 문자열의 마지막 문자 인덱스를 표현하는 lastIndex 프로퍼티를 제공한다.

<코드>

"Hello!".length // 6  
"Hello!".lastIndex // 5(첫 번째 문자의 인덱스가 0이므로)

</코드>

시작인 0이 인덱스를 각괄호([]) 안에 넣는 연산자를 사용해 개별 문제에 접근할 수 있다. JVM에서 잘못된 인덱스를 넘기면 자바와 마찬가지로 StringIndexOutOfBoundsException 예외가 발생한다.

<코드>

val s = “Hello!”  
println(s[0]) // H  
println(s[1]) // e  
println(s[5]) // !  
println(s[10]) // 잘못된 인덱스

</코드>

+ 연산자를 사용해 두 문자열을 연결(concatenate)할 수 있다. toString()을 사용해 문자열로 변환될 수 있는 모든 값을 +의 두 번째 피연산자로 지정할 수 있다. 하지만 일반적으로 문자열 템플릿이 더 간결하므로 문자열 템플릿을 사용하는 것이 좋다.

<코드>

val s = “The sum is: ” + sum // "The sum is $sum"으로 대신할 수 있음

</코드>

문자열은 ==와 !=를 사용해 동등성을 비교할 수 있다. 이들 연산은 문자열의 내용을 비교하기 때문에, 서로 다른 두 객체 인스턴스를 비교해도 문자들의 순서와 길이가 같으면 같은 문자열로 간주한다.

<코드>

val s1 = “Hello!”  
val s2 = “Hel” + “lo!”  
println(s1 == s2) // true

</코드>

**자바와 코틀린의 차이**: 자바의 ==와 != 연산자는 참조 동등성(referential equality)을 비교하기 때문에 실제 문자열 내용을 비교하려면 equals() 메서드를 사용해야 한다. 코틀린에서는 ==가 기본적으로 equals()를 가리키는 편의 문법이기 때문에 ==를 사용하면 직접 equals()를 호출하므로, 따로 equals()를 호출할 필요가 없다. 널 가능성을 논의에서 배제하면, 위 코드는 자바의 s1.equals(s2)과 같다. 코틀린에서 참조 동등성을 쓰고 싶으면 어떻게 할까? ===와 !== 연산자를 사용하면 된다.

문자열은 사전식 순서(lexicographically)로 정렬된다. 따라서 <, >, <=, >= 같은 연산자를 사용해 문자열을 비교할 수 있다.

<코드>

println("abc" < "cba") // true  
println("123" > "34") // false

</코드>

문자열은 수 타입이나 불로 변환하는 toByte(), toShort(), toInt(), toLong(), toFloat(), toDouble(), toBoolean() 변환 함수를 제공한다. 문자열이 대상 타입으로 변환될 수 있게 형태가 잘 잡혀있지 않다면(well-formed) 이런 함수는 런타임 오류를 발생시킨다.

다음은 문자열이 제공하는 다른 유용한 함수들이다.

|  |  |  |
| --- | --- | --- |
| isEmpty  isNotEmpty | 문자열이 비어있는지 검사한다 | “Hello”.isEmpty() // false  “”.isEmpty() // true  “Hello”.isNotEmpty() // true |
| substring | 부분 문자열을 추출한다 | “Hello”.substring(2) // “llo”  “Hello”.substring(1, 3) // “el”  startsWith |
| endsWith | 접두사(prefix)나 접미사(suffix)인지 검사한다 | “Hello”.startsWith(“Hel”) // true  “Hello”.endsWith(“lo”) // true |
| indexOf | 인자로 받은 문자나 문자열이 수신객체인 문자열에 나타나는 첫번째 인덱스를 반환한다 | // 맨 앞부터 찾기  “abcabc”.indexOf(‘b’) // 1  “abcabc”.indexOf(“ca”) // 2  “abcabc”.indexOf(“cd”) // -1  // 주어진 인덱스부터 찾기  “abcabc”.indexOf(‘b’, 2) // 4  “abcabc”.indexOf(“ab”, 2) // 3` |

이 책 전체에서(특히 **7장 컬렉션과 I/O 자세히 알아보기**)에서 코틀린 문자열 API 예제를 더 살펴볼 것이다. 공식 문서에서 더 많은 정보를 얻을 수 있다(kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/index.html).

## 배열

배열은 내장된 코틀린 데이터 구조로, 미리 정해진 숫자만큼 같은 타입의 원소를 모아서 저장하고 각각을 인덱스로 참조할 수 있게 해준다. 코틀린 배열은 개념적으로 자바 배열과 비슷하며, 실제로 코틀린/JVM 애플리케이션에서는 자바 배열로 코틀린 배열을 표현한다. 이 절에서는 배열을 정의하고 배열에 저장된 데이터에 접근하는 방법을 살펴본다.

### 배열 정의하기

배열 구조를 구현하는 가장 일반적인 코틀린 타입은 Array<T>이다. 여기서 T는 원소의 타입을 뜻한다. **1장 강력하고 실용적인 코틀린** 예제에서는 프로그램에 전달된 인자가 저장되는 Array<String> 타입의 파라미터를 받는 main() 함수를 살펴봤다. 만들려는 배열의 크기를 미리 알 수 있다면 표준 함수 중 하나를 사용해 배열을 생성할 수 있다.

<코드>

val a = emptyArray<String>() // Array<String> (원소 0개)  
val b = arrayOf(“hello”, “world”) // Array<String> (원소 2개)  
val c = arrayOf(1, 4, 9) // Array<Int> (원소 3개)

</코드>

[편집자께: 저자가 Int를 써야 하는데 Integer를 썼습니다. Integer는 자바 타입입니다.]

각 함수는 제네릭(generic)하다. 이 말은 호출할 때 원소의 타입을 지정해야 한다는 뜻이다. 하지만 타입 추론 덕분에 두 번째와 세 번째 함수 호출의 타입을 인자들의 타입에서 알아낼 수 있다. 예를 들어 정수로 배열을 만들면 이 배열은 분명 Array<Int> 타입일 것이다. 하지만 첫 번째 호출에서는 컴파일러가 타입을 추론할 수 있는 단초가 없다. 따라서 꺽쇠(<>) 사이에 원소 타입을 명시해야 한다. 지금은 그냥 이런 문법을 그대로 받아들이고 제네릭 타입에 대한 논의는 **9장 제네릭스**에 가서 다루자.

인덱스로부터 원소를 만들어내는 방법을 기술하여 배열을 생성하는 더 유연한 방법도 있다. 다음 코드는 1부터 사용자가 입력한 값에 이르는 정수의 제곱으로 이뤄진 배열을 만든다.

<코드>

val size = readLine()!!.toInt()  
val squares = Array(size) { (it + 1)\*(it + 1) }

</코드>

중괄호({}) 안에 들어있는 언어 요소를 람다(lambda)라고 부른다. 람다는 인덱스를 기반으로 값을 계산하는 식을 정의한다. 이때 인덱스를 표현하는 변수로 자동으로 선언되는 it을 사용한다. 배열 인덱스는 0부터 배열크기-1 이기 때문에 이 배열에는 1, 4, 9 등의 값이 들어간다. 지금은 이런 구문을 있는 그대로 받아들이고, 나중에 **5장 고급 함수와 함수형 프로그래밍 활용하기**에서 이에 대해 자세히 알아보자.

Array<Int>를 사용하는 배열은 제대로 작동하지만 모든 수를 박싱하기 때문에 그다지 실용적이지 못한 해법이다. 이런 이유로 코틀린은 더 효율적인 ByteArray, ShortArray, IntArray, LongArray, FloatArray, DoubleArray, CharArray, BooleanArray라는 특화된 배열 타입을 제공한다. JVM에서 이런 배열 타입들은 int[], boolean []등의 원시 타입 배열로 표현된다. 이런 특화된 배열에도 arrayOf()나 Array()에 해당하는 함수가 함께 따라온다.

<코드>

val operations = charArrayOf(‘+’, ‘-’, ‘\*’, ‘/’, ’%’)  
val squares = IntArray(10) { (it + 1)\*(it + 1) }

</코드>

**자바와 코틀린의 차이**: 자바와 달리 코틀린에는 new 연산자가 없기 때문에 배열 인스턴스 생성이 일반 함수 호출처럼 보인다. 코틀린에서는 배열 원소를 명시적으로 초기화해야만 한다는 점에도 유의하라.

### 배열 사용하기

배열 타입은 문자열 타입과 꽤 비슷하다. 특히 size(문자열의 length에 해당)와 lastIndex 프로퍼티가 있다는 점과 인덱스 연산으로 원소에 접근할 수 있다는 점이 비슷하다. 잘못된 인덱스를 사용하면 런타임에 IndexOutOfBoundsException 예외가 발생한다.

<코드>

val squares = arrayOf(1, 4, 9, 16)  
squares.size // 4  
squares.lastIndex // 3  
squares[3] // 16  
squares[1] // 4

</코드>

하지만 문자열과 달리 배열에서는 원소를 변경할 수 있다.

<코드>

squares[2] = 100 // squares: 1, 4, 100, 16  
squares[3] += 9 // squares: 1, 4, 100, 25  
squares[0]-- // squares: 0, 4, 100, 25

</코드>

자바와 마찬가지로 배열 타입의 변수 자체에는 실제 데이터에 대한 참조를 저장한다. 이로 인해 배열 변수에 다른 배열을 대입하면 같은 데이터 집합을 함께 공유하게 된다.

<코드>

val numbers = squares  
numbers[0] = 1000 // 바뀐 데이터가 squares와 numbers에 공유됨  
println(squares[0]) // prints 1000

</코드>

원본과 별도로 배열을 만들고 싶다면 copyOf() 함수를 사용해야 한다. copyOf()는 필요 시 다른 크기의 배열을 만들어내기도 한다.

<코드>

val numbers = squares.copyOf()  
numbers[0] = 1000 // squares에는 영향이 없다  
squares.copyOf(2) // 뒤가 잘림: 1, 4  
squares.copyOf(5) // 부족한 부분에 0이 채워짐: 1, 4, 9, 16, 0

</코드>

배열 타입 변수에 타입이 다른 배열을 대입할 수 없다는 점을 기억하라. 다음 코드는 컴파일 오류를 발생시킨다.

<코드>

var a = arrayOf(1, 4, 9, 16)  
a = arrayOf(“one”, “two”) // Error: can’t assign Array<String> to Array<Int>

</코드>

**자바와 코틀린의 차이**: 자바에서는 상위타입의 배열에 하위타입의 배열을 대입할 수 있었다. 배열이 가변 데이터 구조이기 때문에 이런 대입은 런타임 시 문제를 발생시킬 수 있다.

<코드>

Object[] objects = new String[] { “one”, “two”, “three” };  
objects[0] = new Object(); // ArrayStoreException 예외가 발생함

</코드>

이런 이유로 코틀린 배열 타입은 (자신과 같은 타입을 제외하고) 모든 다른 배열 타입과 서로 하위타입 관계가 성립하지 않는다고 간주되며, 이에 따라 앞에서 본 것 같은 대입도 금지된다. 코틀린에서 String은 Any의 하위 타입이지만 Array<String>은 Array<Any>의 하위타입이 아니다.

<코드>

val strings = arrayOf(“one”, “two”, “three”)  
val objects: Array<Any> = strings // 예외

</코드>

사실 이는 **9장 제네릭스**에서 다룰 강력한 변성(variance)이라는 개념을 구체적으로 적용한 것일 뿐이다.

배열을 생성하고 나면 그 길이를 바꿀 수 없지만, + 연산을 사용해 원소를 추가한 새로운 배열을 만들 수는 있다.

<코드>

val b = intArrayOf(1, 2, 3) + 4 // 원소를 하나만 추가: 1, 2, 3, 4  
val c = intArrayOf(1, 2, 3) + intArrayOf(5, 6) // 다른 배열을 추가: 1, 2, 3, 5, 6

</코드>

문자열과 달리 배열에 대한 ==과 != 연산자는 원소 자체를 비교하지 않고 참조를 비교한다.

<코드>

intArrayOf(1, 2, 3) == intArrayOf(1, 2, 3) // false

</코드>

배열 내용을 비교하고 싶으면 contentEquals() 함수를 사용하라.

<코드>

intArrayOf(1, 2, 3).contentEquals(intArrayOf(1, 2, 3)) // true

</코드>

<참고>IDE 팁

인텔리J는 ==나 !=를 사용해 배열을 비교하는 경우를 contentEquals() 호출로 바꾸라는 경고를 표시해준다.

</참고>

배열을 사용할 때 몇 가지 표준 함수가 도움이 될 것이다.

|  |  |  |
| --- | --- | --- |
| isEmpty | 배열이 비었는지 검사 | intArrayOf(1, 2).isEmpty() // false |
| isNotEmpty | 배열에 원소가 있는지 검사 | intArrayOf(1, 2).isNotEmpty() // true |
| indexOf | 인자와 일치하는 최초의 배열 아이템의 인덱스를 반환 (일치하는 원소가 없으면 -1) | intArrayOf(1, 2, 3).indexOf(2) // 1  intArrayOf(1, 2, 3).indexOf(4) // -1 |

나중에 배열을 다루는 함수를 더 소개할 것이다. 특히 코틀린 컬렉션 API를 다루는 7장, 컬렉션과 IO 자세히 알아보기에서 이런 함수 대부분을 보게 된다.

## 결론

이 장에서는 코틀린을 처음 맛봤다. 변수와 타입 추론에 대해 배우고, 기본 타입에 대해 살펴봤으며 수나 문자, 불에 대한 기본 연산과 문자열이나 배열 같이 더 복잡한 형태의 데이터를 생성하고 조작하는 법을 살펴봤다. 그리고 코틀린의 설계가 어떻게 일반적인 프로그래밍 오류를 방지해주는지 살펴봤다. 토대를 구축했으니 이제 다음 단계로 나아가 보자. **3장. 함수 정의하기**에서는 코틀린의 제어 구조, 함수나 패키지를 사용해 코드를 구조화하는 방법을 살펴본다.

1. 옮긴이 - 이는 문법 구조에 대한 설명일 뿐이다. 예를 들어 -2를 컴파일러가 구문분석을 할 때는 -라는 단항연산자를 2라는 상수에 적용하는 식으로 해석하긴 하지만, 바이트코드를 생성할 때는 -2라는 음수 상수를 사용한다. [↑](#footnote-ref-1)
2. 옮긴이 - 최초 설계 당시 유니코드는 65535자면 전 세계 문자를 담기에 충분하리라고 생각하고 설계됐지만, 나중에 문자가 추가되면서 보조 문자(supplementary character)가 생겨났으며 이 문자들의 코드포인트는 U+1000부터 U+10FFFF라서 16비트 Char 타입에 들어갈 수 없다. 이로 인해 자바 5부터는 저수준에서는 32비트 int를 사용하는 식으로 API가 변경됐다. 다만, 기본 플레인(U+0000부터 U+FFFF)에 속하는 문자는 여전히 char 타입에 들어갈 수 있지만, 그렇지 못한 문자들은 char 하나만으로는 표현하지 못하는 경우가 생긴다. 이에 대해서는 https://www.oracle.com/technical-resources/articles/javase/supplementary.html 를 읽어보라(영문). [↑](#footnote-ref-2)
3. 옮긴이 - 실수를 정수로 만들 때 수직선상에서 0쪽으로 값이 이동한다고 생각하면 된다. 즉, 절대값이 감소하는 방향으로 변환이 이뤄진다. 예를 들어 1.5는 1로, -1.5는 -1로 바뀐다. 한편, 여기서 rounding은 10진수에서 벌어지지 않고 2진수에서 벌어진다는 점에 유의하라. 마지막 예제에서 999999995904가 나오는 이유가 바로 이 때문이다. [↑](#footnote-ref-3)
4. 옮긴이 - 혼합 연산을 허용하기는 하지만, 부동소수점 수 사이의 비교나 정수와 부동소수점 수 사이를 비교할 때는 조심해야 한다. 유효숫자 범위에 따라 미묘한 버그가 발생할 수 있다. 이에 대해 자세히 설명하는 것은 이 책의 범위를 벗어난다. 수치해석 관련 서적을 찾아보라. [↑](#footnote-ref-4)
5. 옮긴이 - 의미상으로 NaN은 수가 아니기 때문에 두 NaN이 같은 값인지를 비교할 수 없다. NaN==NaN이 항상 false가 되야 하는 이유를 또 다른 측면에서 기술적으로 보자면, 쓰기는 NaN이라고 마치 한 값인 것처럼 썼지만 NaN을 표현하는 비트 패턴은 다양하기 때문이다. 예를 들어 32비트 Float에서 Nan은 s111 1111 1xxx xxxx xxxx xxxx xxxx xxxx (s는 부호, x는 0 또는 1) 형태로 표현된다. Float.NaN은 이런 값 중 0x7fc00000로 인코딩되어 있지만, 실제 부동소수점 연산에서는 다양한 NaN이 발생할 수 있다. 그래서 어떤 값이 NaN인지를 NaN 상수와 직접 비교해 알아낼 수는 없고, isNaN() 함수를 사용해야 한다. [↑](#footnote-ref-5)
6. 옮긴이 - 정적으로라는 말은 실행하기 전의 상태(주로 소스코드나 컴파일된 바이트 코드)를 뜻한다. 반대로 동적으로(dynamically)라는 말은 프로그램을 실행하는 시점(즉, 런타임)을 뜻한다. [↑](#footnote-ref-6)
7. 옮긴이 -여기서는 문자열 변환 예제를 보여주기 위해 java.util.Date를 썼지만, 시간을 다룰 때는 항상 java.time 패키지에 있는 기능을 활용하는게 바람직하다. [↑](#footnote-ref-7)