



코틀린 언어 문서

번역: 최범균(<http://github.com/madvirus/kotlin-web-site>)
(주의: 역량부족으로 오류가 존재할 수 있음)

목차

시작하기	4
기본 구문	4
이디엄(Idioms)	10
코딩 규칙	15
기초	17
기본 타입	17
패키지	23
흐름 제어	24
리턴과 점프	28
클래스와 오브젝트	30
클래스와 상속	30
프로퍼티와 필드	36
인터페이스	40
가시성 제한자	42
확장	44
데이터 클래스	50
지네릭	52
지네릭 함수	56
지네릭 제약	56
중첩 클래스	58
Enum 클래스	59
오브젝트 식과 선언	61
위임	64
위임 프로퍼티	65
함수와 람다	69
함수	69
고차 함수와 람다	75
인라인 함수	80
기타	83

분리 선언	83
컬렉션	85
범위	87
타입 검사와 변환	90
This 식	92
동등성	93
연산자 오버로딩	94
Null 안전성	97
익셉션	100
애노테이션	102
리플렉션	107
타입-안전 빌더	110
동적 타입	115
레퍼런스	116
상호 운용	118
코틀린에서 자바 코드 호출하기	118
자바에서 코틀린 실행하기	126
도구	133
코틀린 코드 문서화	133
메이븐 사용하기	136
앤티 사용하기	139
그래들 사용하기	143
코틀린과 OSGi	147
FAQ	149
FAQ	149
자바와 비교	152
스칼라와 비교	153

시작하기

기본 구문

패키지 정의

소스 파일 최상단에 패키지를 지정한다.

```
package my.demo

import java.util.*

// ...
```

디렉토리와 패키지가 일치할 필요는 없다. 소스 파일은 아무 디렉토리에나 위치할 수 있다.

[패키지](#) 참고.

함수 정의

두 개의 `Int` 파라미터와 `Int` 리턴 타입을 갖는 함수:

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

식(expression) 몸체를 갖고 리턴 타입을 추론하는 함수:

```
fun sum(a: Int, b: Int) = a + b
```

의미있는 값을 리턴하지 않는 함수:

```
fun printSum(a: Int, b: Int): Unit {
    print(a + b)
}
```

`Unit` 리턴 타입은 생략할 수 있음:

```
fun printSum(a: Int, b: Int) {
    print(a + b)
}
```

[함수](#) 참고.

로컬 변수 정의

한 번만 할당하는 (읽기 전용) 로컬 변수:

```
val a: Int = 1
val b = 1 // `Int` 타입 추론
val c: Int // 값을 할당하지 않을 경우 타입 필요
c = 1 // 확정(definite) 할당
```

변경 가능 변수:

```
var x = 5 // `Int` 타입 추론
x += 1
```

[프로퍼티와 필드](#) 참고.

주석

자바와 자바스크립트처럼 코틀린도 라인(end-of-line) 주석과 블록 주석을 지원한다.

```
// 이것은 라인(end-of-line) 주석

/* 이 코드는 여러 줄에 걸치는
   블록 주석입니다. */
```

자바와 달리 코틀린은 블록 주석을 중첩할 수 있다.

문서화를 위한 주석 문법은 [코틀린 코드 문서화](#)를 참고한다.

문자열 템플릿 사용하기

```
fun main(args: Array<String>) {
    if (args.size == 0) return

    print("First argument: ${args[0]}")
}
```

[문자열 템플릿](#) 참고.

조건 식 사용하기

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

식에 `if` 사용하기:

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

[if-식](#) 참고.

nullable 값 사용과 `null` 검사

레퍼런스가 `null` 값을 가질 수 있으면 반드시 nullable하다고 표시해야 한다.

`str` 이 정수를 포함하지 않을 때 `null`을 리턴하는 코드 예:

```
fun parseInt(str: String): Int? {
    // ...
}
```

nullable 값을 리턴하는 함수 사용하기:

```
fun main(args: Array<String>) {
    if (args.size < 2) {
        print("Two integers expected")
        return
    }

    val x = parseInt(args[0])
    val y = parseInt(args[1])

    // 두 값이 null일 수 있으므로 `x * y`는 에러를 발생할 수 있다.
    if (x != null && y != null) {
        // null 검사 후에 x와 y를 non-nullable로 자동 변환
        print(x * y)
    }
}
```

또는

```
// ...
if (x == null) {
    print("Wrong number format in '${args[0]}')")
    return
}
if (y == null) {
    print("Wrong number format in '${args[1]}')")
    return
}

// null 검사 후에 x와 y를 non-nullable로 자동 변환
print(x * y)
```

[null-안전성](#) 참고.

타입 검사와 자동 변환 사용하기

`is` 연산자는 대상이 지정한 타입의 인스턴스인지 검사한다. 불변(immutable) 로컬 변수나 프로퍼티에 대해 특정 타입인지 검사하면 명시적으로 타입을 변환할 필요가 없다.

```
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // 이 코드 브랜치(branch)에서 `obj` 는 자동으로 `String` 으로 변환
        return obj.length
    }

    // 타입 검사 브랜치 밖에서 `obj` 는 여전히 `Any` 타입
    return null
}
```

또는

```
fun getStringLength(obj: Any): Int? {
    if (obj !is String)
        return null

    // `obj` 는 이 브랜치에서 자동으로 `String` 으로 변환
    return obj.length
}
```

또는 심지어 아래도 가능

```
fun getStringLength(obj: Any): Int? {
    // `&&` 의 우측에서 자동으로 `String` 으로 변환
    if (obj is String && obj.length > 0)
        return obj.length

    return null
}
```

[클래스와 타입 변환](#) 참고.

for 루프 사용하기

```
fun main(args: Array<String>) {  
    for (arg in args)  
        print(arg)  
}
```

또는

```
for (i in args.indices)  
    print(args[i])
```

[for 루프](#) 참고.

while 루프 사용하기

```
fun main(args: Array<String>) {  
    var i = 0  
    while (i < args.size)  
        print(args[i++])  
}
```

[while 루프](#) 참고.

when 식 사용하기

```
fun cases(obj: Any) {  
    when (obj) {  
        1          -> print("One")  
        "Hello"    -> print("Greeting")  
        is Long     -> print("Long")  
        !is String -> print("Not a string")  
        else       -> print("Unknown")  
    }  
}
```

[when 식](#) 참고.

범위(range) 사용하기

`in` 연산자를 사용해서 숫자가 특정 범위 안에 있는지 검사:

```
if (x in 1..y-1)  
    print("OK")
```

숫자가 범위 밖인지 검사:


```
if (x !in 0..array.lastIndex)
    print("Out")
```

범위에 속한 숫자를 반복(iteration):

```
for (x in 1..5)
    print(x)
```

[범위](#) 참고.

컬렉션 사용하기

컬렉션 반복:

```
for (name in names)
    println(name)
```

`in` 연산자를 사용해서 컬렉션이 객체를 포함하고 있는지 검사하기:

```
if (text in names) // names.contains(text) 호출
    print("Yes")
```

컬렉션을 필터링하고 변환(맵)할 때 람다식 사용하기:

```
names
    .filter { it.startsWith("A") }
    .sortedBy { it }
    .map { it.toUpperCase() }
    .forEach { print(it) }
```

[고차함수와 람다](#) 참고.

이디엄(Idioms)

코틀린에서 자주 사용하는 코딩 방식을 모은 것이다. 여러분도 자주 사용하는 코딩 방식이 있다면 풀 리퀘스트를 날려 공헌해보자.

DTO 만들기 (POJO/POCO)

```
data class Customer(val name: String, val email: String)
```

이 코드는 다음 기능을 가진 `Customer` 클래스를 제공한다.

- 모든 프로퍼티에 대한 getter (`var` 프로퍼티는 setter 포함)
- `equals()`
- `hashCode()`
- `toString()`
- `copy()`
- 모든 프로퍼티에 대해 `component1()`, `component2()`, ... ([데이터 클래스](#) 참고)

함수 파라미터의 기본 값

```
fun foo(a: Int = 0, b: String = "") { ... }
```

리스트 필터

```
val positives = list.filter { x -> x > 0 }
```

또는 다음과 같이 짧게:

```
val positives = list.filter { it > 0 }
```

문자열 인터플레이션(삽입)

```
println("Name $name")
```

인스턴스 검사

```
when (x) {  
    is Foo -> ...  
    is Bar -> ...  
    else   -> ...  
}
```

map/list pair 탐색

```
for ((k, v) in map) {
    println("$k -> $v")
}
```

k, v 에 아무 이름이나 붙여도 된다.

범위 사용

```
for (i in 1..100) { ... }
for (x in 2..10) { ... }
```

읽기 전용 리스트

```
val list = listOf("a", "b", "c")
```

읽기 전용 맵

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

맵 접근

```
println(map["key"])
map["key"] = value
```

지연(lazy) 프로퍼티

```
val p: String by lazy {
    // 문자열 계산
}
```

확장 함수

```
fun String.spaceToCamelCase() { ... }

"Convert this to camelcase".spaceToCamelCase()
```

싱글톤 생성

```
object Resource {
    val name = "Name"
}
```

if not null 단축 표현

```
val files = File("Test").listFiles()

println(files?.size)
```

if not null과 else 단축 표현

```
val files = File("Test").listFiles()

println(files?.size ?: "empty")
```

null이면 문장 실행하기

```
val data = ...
val email = data["email"] ?: throw IllegalStateException("Email is missing!")
```

null이 아니면 실행하기

```
val data = ...

data?.let {
    ... // null이 아니면 이 블록을 실행
}
```

when 문장에서 리턴하기

```
fun transform(color: String): Int {
    return when (color) {
        "Red" -> 0
        "Green" -> 1
        "Blue" -> 2
        else -> throw IllegalArgumentException("Invalid color param value")
    }
}
```

‘try/catch’ 식

```

fun test() {
    val result = try {
        count()
    } catch (e: ArithmeticException) {
        throw IllegalStateException(e)
    }

    // result로 작업
}

```

‘if’ 식

```

fun foo(param: Int) {
    val result = if (param == 1) {
        "one"
    } else if (param == 2) {
        "two"
    } else {
        "three"
    }
}

```

Unit을 리턴하는 메서드를 빌더(Builder) 스타일로 사용

```

fun arrayOfMinusOnes(size: Int): IntArray {
    return IntArray(size).apply { fill(-1) }
}

```

한 개 식을 갖는 함수

```

fun theAnswer() = 42

```

이 코드는 다음 코드와 동일하다.

```

fun theAnswer(): Int {
    return 42
}

```

코드를 더 짧게 하기 위해 이 코드를 다른 이디엄과 함께 사용할 수 있다. 다음은 **when** 식과 함께 사용한 예이다.

```

fun transform(color: String): Int = when (color) {
    "Red" -> 0
    "Green" -> 1
    "Blue" -> 2
    else -> throw IllegalArgumentException("Invalid color param value")
}

```

객체 인스턴스의 메서드 여러 번 호출하기 ('with')

```
class Turtle {
    fun penDown()
    fun penUp()
    fun turn(degrees: Double)
    fun forward(pixels: Double)
}

val myTurtle = Turtle()
with(myTurtle) { // 100 픽셀 정사각형 그리기
    penDown()
    for(i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

자바 7의 try-with-resources

```
val stream = Files.newInputStream(Paths.get("/some/file.txt"))
stream.buffered().reader().use { reader ->
    println(reader.readText())
}
```

지네릭 타입 정보가 필요한 지네릭 함수를 위한 간편 형식

```
// public final class Gson {
//     ...
//     public <T> T fromJson(JsonElement json, Class<T> classOfT) throws
//     JsonSyntaxException {
//         ...
//     }
// }

inline fun <reified T: Any> Gson.fromJson(json): T = this.fromJson(json,
T::class.java)
```

코딩 규칙

이 문서는 코틀린 언어를 위해 현재 사용 중인 코딩 스타일을 포함한다.

이름 규칙

애매하지 않다면, 다음 자바 코딩 규칙을 기본으로 사용한다.

- 이름에는 낙타표기법을 사용한다(이름에 밑줄을 넣지 않는다).
- 타입은 대문자로 시작한다.
- 메서드와 프로퍼티는 소문자로 시작한다.
- 들여쓰기에 공백 4개를 사용한다.
- `public` 함수는 코틀린 문서에 보이도록 문서화를 한다.

콜론

콜론이 타입과 상위 타입을 구분하면 콜론 전에 공백을 넣고, 콜론이 인스턴스와 타입을 구분하면 공백을 넣지 않는다.

```
interface Foo<out T : Any> : Bar {  
    fun foo(a: Int): T  
}
```

람다

람다식에서 종괄호 주변에 공백을 사용하고 파라미터와 몸체를 구분하는 -> 주변에도 공백을 사용한다. 가능하면 람다를 괄호 밖에 전달한다.

```
list.filter { it > 10 }.map { element -> element * 2 }
```

람다가 중첩되지 않고 짧으면 파라미터를 명시적으로 선언하는 대신 `it` 을 사용한다. 파라미터를 가진 중첩된 람다는 파라미터를 항상 명시적으로 지정한다.

Unit

함수가 `Unit`을 리턴하면 리턴 타입을 생략한다.

```
fun foo() { // ": Unit"을 생략함  
  
}
```

함수 대 프로퍼티

인자 없는 함수와 읽기 전용 프로퍼티는 서로 대신 사용할 수 있다. 비록 의미가 비슷하긴 하지만 몇 가지 규칙에 따라 둘 중 하나를 선택한다.

다음 알고리즘일 때 함수보다 프로퍼티를 선호한다.

- 익셉션을 던지지 않음

- $O(1)$ 복잡도를 가짐
- 계산 비용이 작다 (또는 처음 실행시 캐시한다)
- 호출과 같은 결과를 리턴한다

기초

기본 타입

코틀린은 모든 게 객체라서 변수에 대해 멤버 함수나 프로퍼티를 사용할 수 있다. 최적화 목적의 일부 타입을 제공하고 있지만 사용자에게는 일반 클래스와 같다. 이 절에서는 이 타입 중 숫자, 문자, 부울, 배열에 대해 설명한다.

숫자

코틀린은 자바와 유사한 방법으로 숫자를 다루지만 완전히 동일한 것은 아니다. 예를 들어, 숫자의 경우 큰 범위로의 자동 변환이 없고, 리터럴은 경우에 따라 약간 다르다.

코틀린은 숫자를 위해 다음의 내장 타입을 제공한다(자바와 유사하다):

타입	비트 길이
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

코틀린에서 문자는 숫자가 아니다.

리터럴 상수

정수 값에 대해 다음의 리터럴 상수가 존재한다.

- 십진법: `123`
 - Long은 대문자 'L'을 붙인다: `123L`
- 16진법: `0x0F`
- 이진법: `0b00001011`

주의: 8진법 리터럴은 지원하지 않는다.

코틀린은 또한 실수에 대해 다음 표기법을 지원한다.

- 기본은 Double: `123.5`, `123.5e10`
- `f` 나 `F` 를 붙여 Float 표현: `123.5f`

표현(Representation)

자바 플랫폼에서는 물리적으로 JVM 기본 타입으로 숫자를 보관한다. nullable 숫자 참조(예, `Int?`)가 필요하거나 지네릭과 연관이 있다면 박싱 타입을 사용한다.

숫자를 박싱하면 참조 동일성(identity)은 유지되지 않는 것에 주의해야 한다.

```
val a: Int = 10000
print(a === a) // 'true' 출력
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA === anotherBoxedA) // !!!'false' 출력!!!
```

반면에 값 동등성(equality)은 유지된다.

```
val a: Int = 10000
print(a == a) // 'true' 출력
val boxedA: Int? = a
val anotherBoxedA: Int? = a
print(boxedA == anotherBoxedA) // 'true' 출력
```

명시적 변환

표현이 다르기 때문에 작은 범위 타입은 큰 범위 타입의 하위타입이 아니다. 만약 하위타입이 된다면 다음과 같은 상황에서 문제가 된다.

```
// 가상의 코드로, 실제로는 컴파일되지 않는다.
val a: Int? = 1 // Int의 박싱 타입 (java.lang.Integer)
val b: Long? = a // Long의 박싱 타입 (java.lang.Long)을 만드는 자동 타입 변환
print(a == b) // 깜놀! Long의 equals()는 비교 대상이 Long인지 검사하기 때문에 "false"를 출력한다.
```

따라서, 동일성(identity)뿐만 아니라 모든 곳에서 동등성(equality)까지 잃게 될 수 있다.

이런 까닭에 작은 타입을 큰 타입으로 자동 변환하지 않는다. 이는 명시적 변환없이 `Byte` 값을 `Int` 변수에 할당할 수 없음을 의미한다.

```
val b: Byte = 1 // OK, 리터럴은 정적으로 검사한다.
val i: Int = b // 에러
```

큰 범위 숫자로 명시적으로 타입 변환을 할 수 있다.

```
val i: Int = b.toInt() // OK: 명시적으로 큰 범위 타입으로 변환
```

모든 숫자 타입은 다음 변환을 지원한다.

```
— toByte(): Byte
— toShort(): Short
— toInt(): Int
—
```

- `toLong(): Long`
- `toFloat(): Float`
- `toDouble(): Double`
- `toChar(): Char`

문맥에서 타입을 추론할 수 있고 수치 연산자를 각 타입 변환에 맞게 오버로딩했기 때문에, 자동 타입 변환이 없어도 거의 문제가 되지 않는다.

```
val l = 1L + 3 // Long + Int => Long
```

연산

코틀린은 숫자에 대한 표준 수치 연산을 지원한다. 이 연산자는 각 타입의 멤버로 정의되어 있다(컴파일러가 해당 명령어로 최적화한다). [연산자 오버로딩](#) 참고.

비트 연산의 경우 이를 위한 특수 문자를 사용하지 않고 중위 형식으로 호출할 수 있는 (이름을 가진) 함수가 존재한다. 다음은 예이다.

```
val x = (1 shl 2) and 0x000FF000
```

다음은 전체 비트 연산 목록이다(`Int` 와 `Long` 만 가능).

- `shl(bits)` – 부호가진 좌측 쉬프트 (자바의 `<<`)
- `shr(bits)` – 부호가진 우측 쉬프트 (자바의 `>>`)
- `ushr(bits)` – 부호없는 오른쪽 쉬프트 (자바의 `>>>`)
- `and(bits)` – and 비트연산
- `or(bits)` – or 비트연산
- `xor(bits)` – xor 비트연산
- `inv()` – 역 비트연산

문자

`Char` 타입으로 문자를 표현한다. 문자를 숫자로 다룰 수 없다.

```
fun check(c: Char) {
    if (c == 1) { // ERROR: 호환되지 않는 타입
        // ...
    }
}
```

문자 리터럴은 단일 따옴표 안에 위치한다: `'1'`. 특수 문자는 역슬래시를 이용해서 표시한다. 지원하는 이스케이프 시퀀스는 다음과 같다: `\t`, `\b`, `\n`, `\r`, `\'`, `\"`, `\\` 그리고 `\$`. 다른 문자를 인코딩하려면 유니코드 이스케이프 시퀀스 구문을 사용한다: `'\uFF00'`.

문자를 `Int` 숫자로 변환할 수 있다.

```
fun decimalDigitValue(c: Char): Int {
    if (c !in '0'..'9')
        throw IllegalArgumentException("Out of range")
    return c.toInt() - '0'.toInt() // 숫자로 변환
}
```

숫자처럼 nullable 참조가 필요하면 문자를 박싱한다. 박싱 연산은 동일성(identity)을 유지하지 않는다.

부울(Boolean)

`Boolean` 타입은 부울을 표현하며 `true`와 `false`의 두 값을 갖는다.

nullable 참조가 필요하면 `Boolean`을 박싱한다.

부울 타입의 내장 연산은 다음과 같다.

- `||` – lazy 논리 합(disjunction)
- `&&` – lazy 논리 곱(conjunction)
- `!` – 부정

배열

코틀린은 `Array` 클래스를 이용해서 배열을 표현한다. 이 클래스는 `get` 과 `set` 함수와 `size` 프로퍼티를 가지며 몇 개의 다른 유용한 멤버 함수가 있다(`get` , `set` 함수는 연산자 오버로딩 규칙에 따라 `[]` 로 바뀐다):

```
class Array<T> private constructor() {
    val size: Int
    fun get(index: Int): T
    fun set(index: Int, value: T): Unit

    fun iterator(): Iterator<T>
    // ...
}
```

`arrayOf()` 라이브러리 함수를 이용해서 배열을 생성한다. 이 함수에 값 목록을 제공한다. 예를 들어, `arrayOf(1, 2, 3)` 은 `[1, 2, 3]` 배열을 생성한다. `arrayOfNulls()` 라이브러리 함수를 사용하면 `null`을 값으로 갖는 지정한 크기의 배열을 생성한다.

팩토리 함수를 사용할 수도 있다. 팩토리 함수는 배열 크기와 배열의 각 인덱스에 대해 초기 값을 생성하는 함수를 인자로 받는다.

```
// ["0", "1", "4", "9", "16"]을 값으로 갖는 Array<String> 생성
val asc = Array(5, { i -> (i * i).toString() })
```

앞서 말했듯이 `[]` 연산은 `get()` 과 `set()` 함수 호출을 의미한다.

주의: 자바와 달리 코틀린의 배열은 무공변(invariant)이다. 이는 `Array<String>` 을 `Array<Any>` 에 할당할 수 없음을 의미하는데, 이는 런타임 실패를 가능한 방지한다(하지만 `Array<out Any>` 는 사용할 수 있는데 이에 대한 내용은 [타입 프로젝션\(Type Projections\)](#)을 참고한다).

코틀린은 기본 타입 배열을 위해 박싱 오버헤드가 없는 `ByteArray`, `ShortArray`, `IntArray` 등의 클래스를 제공한다. 이 클래스는 `Array` 클래스와 상속 관계를 갖지 않지만, 동일한 메서드와 프로퍼티를 갖는다. 또한 각 클래스를 위한 팩토리 함수가 존재한다.

```
val x: IntArray = intArrayOf(1, 2, 3)
x[0] = x[1] + x[2]
```

문자열

`String` 타입으로 문자열을 표현한다. 문자열은 불변(`immutable`)이다. 문자열의 요소는 문자이며, 각 요소는 인덱스 연산을 이용해서 접근할 수 있다: `s[i]`. `for`-루프를 통해 문자열을 이터레이션할 수 있다.

```
for (c in str) {
    println(c)
}
```

문자열 리터럴

코틀린은 두 종류의 문자열 리터럴을 갖는다. 하나는 이스케이프 문자를 가질 수 있는 이스케이프드(`escaped`) 문자열이고, 다른 하나는 뉴라인과 임의 텍스트를 가질 수 있는 `raw` 문자열이다. 이스케이프드 문자열은 자바 문자열과 매우 유사하다.

```
val s = "Hello, world!\n"
```

백슬래시를 사용해서 이스케이프 문자를 처리한다. 지원하는 이스케이프 시퀀스 목록은 앞서 [문자](#) 절을 참고한다.

`raw` 문자열은 세 따옴표(`"""`)로 구분한다. 이스케이프는 안 되고 뉴라인과 모든 다른 문자를 포함할 수 있다.

```
val text = """
    for (c in "foo")
        print(c)
    """
```

[trimMargin\(\)](#) 함수로 앞 공백을 제거할 수 있다.

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

기본적으로 `|` 문자를 가장자리 접두사로 사용하며, `trimMargin(">")` 와 같이 파라미터를 이용해서 접두사를 변경할 수 있다.

문자열 템플릿

문자열은 템플릿 표현식을 포함할 수 있다. 예를 들어, 코드의 일부를 평가해서 그 결과를 문자열에 연결할 수 있다. 템플릿 표현식은 달러 기호(`$`)로 시작하고 단순 이름을 포함한다.

```
val i = 10
val s = "i = $i" // 결과는 "i = 10"
```

또는 다음과 같이 중괄호 안에 위치한다.

```
val s = "abc"
val str = "$s.length is ${s.length}" // 결과는 "abc.length is 3"
```

템플릿은 raw 문자열과 이스케이프드 문자열에서 모두 지원한다. (역슬래시를 이용한 특수 문자 표기를 지원하지 않는) raw 문자열에서 `$` 문자를 표현하고 싶다면 다음 구문을 사용한다.

```
val price = """
${'$'}9.99
"""
```

패키지

소스 파일은 패키지 선언으로 시작한다.

```
package foo.bar

fun baz() {}

class Goo {}

// ...
```

클래스나 함수와 같은 소스 파일의 모든 내용은 선언한 패키지에 속한다. 위 예의 경우 `baz()`의 전체 이름은 `foo.bar.baz`이고 `Goo`의 전체 이름은 `foo.bar.Goo`이다.

패키지를 지정하지 않으면 파일의 모든 내용은 이름이 없는 “기본(default)” 패키지에 속한다.

임포트

기본 임포트 외에 각 파일은 자신의 `import` 디렉티브를 포함할 수 있다. 임포트 구문은 [문법](#)에서 설명한다.

다음은 한 개 이름을 임포트하는 예이다.

```
import foo.Bar // Bar에 전체 이름을 사용하지 않고 접근할 수 있다.
```

해당 범위의 모든 요소(패키지, 클래스, 오브젝트 등)에 접근할 수도 있다.

```
import foo.* // 'foo'의 모든 요소에 접근 가능
```

이름이 충돌하면 `as` 키워드로 로컬에서 사용할 이름을 변경해서 충돌을 피할 수 있다.

```
import foo.Bar // Bar로 접근
import bar.Bar as bBar // bBar는 'bar.Bar'를 의미
```

`import` 키워드는 클래스뿐만 아니라 다른 것도 임포트할 수 있다.

- 최상위 레벨 함수와 프로퍼티;
- [오브젝트 선언](#)의 함수와 프로퍼티;
- [열거형 상수](#)

자바와 달리 코틀린은 별도의 “`import static`” 구문을 지원하지 않는다. `import` 키워드를 사용해서 모든 선언을 임포트할 수 있다.

최상위 선언의 가시성

최상위 선언이 `private`이면 그것이 선언된 파일에 대해 `private`이다([가시성 제한자](#) 참고)

흐름 제어

If 식

코틀린에서 **if**는 식이며 값을 리턴한다. 삼항 연산자(condition ? then : else)는 없는데 그 이유는 **if**로 충분히 할 수 있기 때문이다.

```
// 구식 용법
var max = a
if (a < b)
    max = b

// else와 함께 사용
var max: Int
if (a > b)
    max = a
else
    max = b

// 식으로 사용
val max = if (a > b) a else b
```

if 브랜치는 블록이 될 수 있고 마지막 식이 블록의 값이 된다.

```
val max = if (a > b) {
    print("Choose a")
    a
}
else {
    print("Choose b")
    b
}
```

if를 문장이 아닌 식으로 사용할 경우 **else** 브랜치를 가지려면 식이 필요하다.

[if 문법](#) 참고.

When 식

when은 C와 유사한 언어의 **switch** 연산자를 대체한다. 가장 단순한 형태는 다음과 같다.

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> { // 블록 가능
        print("x is neither 1 nor 2")
    }
}
```


`when`은 충족하는 브랜치 조건이 나올 때까지 모든 브랜치를 순차적으로 찾는다. `when`은 식이나 문장으로 사용할 수 있다. 만약 식으로 사용하면 충족한 브랜치의 값이 전체 식의 값이 된다. 문장으로 사용하면 개별 브랜치의 값은 무시한다. (`if`처럼 각 브랜치는 블록일 수 있으며 블록의 마지막 식이 값이 된다.) `else` 브랜치는 모든 브랜치 조건이 맞지 않을 때 사용한다. `when`을 식으로 사용할 경우, 각 브랜치 조건이 모든 경우의 수를 다뤘다고 컴파일러에서 판명할 수 없으면, `else`를 필수로 사용해야 한다.

여러 경우를 동일하게 처리해야 할 경우 브랜치 조건을 콤마로 묶을 수 있다.

```
when (x) {  
  0, 1 -> print("x == 0 or x == 1")  
  else -> print("otherwise")  
}
```

브랜치 조건으로 (상수뿐만 아니라) 임의의 식을 사용할 수 있다.

```
when (x) {  
  parseInt(s) -> print("s encodes x")  
  else -> print("s does not encode x")  
}
```

`범위`나 컬렉션에 대해 `in`이나 `!in`을 사용해서 값을 검사할 수 있다.

```
when (x) {  
  in 1..10 -> print("x is in the range")  
  in validNumbers -> print("x is valid")  
  !in 10..20 -> print("x is outside the range")  
  else -> print("none of the above")  
}
```

`is`나 `!is`로 특정 타입의 값인지 검사할 수 있다. [스마트 변환](#) 덕에 추가 검사 없이 타입의 프로퍼티와 메서드에 접근할 수 있다.

```
val hasPrefix = when(x) {  
  is String -> x.startsWith("prefix")  
  else -> false  
}
```

`if-else if` 체인 대신 `when`을 사용할 수 있다. 인자를 제공하지 않으면 브랜치 조건은 단순 부울 식이며 각 조건이 `true`일 때 브랜치를 실행한다.

```
when {  
  x.isOdd() -> print("x is odd")  
  x.isEven() -> print("x is even")  
  else -> print("x is funny")  
}
```

[when 문법](#) 참고.

For 루트

`for` 루트는 이터레이터를 제공하는 모든 것에 대해 반복을 수행한다. 구문은 다음과 같다.

```
for (item in collection)
    print(item)
```

몸체는 블록일 수 있다.

```
for (item: Int in ints) {
    // ...
}
```

앞서 언급한 것처럼 `for`는 다음에 맞는 이터레이터(iterator)를 제공하는 모든 것을 반복한다.

- `iterator()` 멤버 함수나 확장 함수를 갖고, 이 함수의 리턴 타입이
 - `next()` 멤버 함수나 확장 함수를 갖고,
 - `Boolean`을 리턴하는 `hasNext()` 함수나 확장 함수를 갖는다.

이 세 함수는 `operator`로 표시해야 한다.

배열에 대한 `for` 루프는 이터레이터 객체를 생성하지 않는 인덱스 기반 루프로 컴파일된다.

인덱스를 이용해서 배열이나 리스트를 반복하고 싶다면 다음 방법을 사용하면 된다.

```
for (i in array.indices)
    print(array[i])
```

이 “범위를 통한 반복”은 추가 객체를 생성하지 않는 코드로 최적화해서 컴파일된다.

대신 `withIndex` 라이브러리 함수로 처리할 수도 있다.

```
for ((index, value) in array.withIndex()) {
    println("the element at $index is $value")
}
```

[for 문법](#) 참고.

While 루프

`while`과 `do..while`은 예상하는대로 동작한다.

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // 여기서 y에 접근 가능!
```

[while 문법](#) 참고.

루프에서 break와 continue

코틀린은 루프에서 전통적인 `break`와 `continue`를 지원한다. [리턴과 점프](#)를 참고한다.

리턴과 점프

코틀린은 세 가지 구조적인(structural) 점프 연산자를 제공한다.

- **return**. 기본적으로 가장 가깝게 둘러싼 함수나 [임의 함수](#)에서 리턴한다.
- **break**. 가장 가깝게 둘러싼 루프를 끝낸다.
- **continue**. 가장 가깝게 둘러싼 루프의 다음으로 넘어간다.

Break와 Continue 라벨

코틀린의 모든 식은 **라벨**로 표식을 가질 수 있다. 라벨은 @ 기호 뒤에 구분자를 갖는다. 예를 들어 `abc@`, `fooBar@` 는 유효한 라벨이다([문법](#) 참고). 식에 라벨을 붙이려면 식 앞에 라벨을 위치시키면 된다.

```
loop@ for (i in 1..100) {  
    // ...  
}
```

이제 **break**나 **continue**를 라벨로 한정할 수 있다.

```
loop@ for (i in 1..100) {  
    for (j in 1..100) {  
        if (...)  
            break@loop  
    }  
}
```

라벨로 한정된 **break**는 그 라벨이 붙은 루프 이후로 실행 지점을 이동한다. **continue**는 루프의 다음 반복을 처리한다.

라벨에 리턴하기

코틀린은 함수 리터럴, 로컬 함수와 객체 식에서 함수를 중첩할 수 있다. 한정된 **return**은 바깥(outer) 함수에서 리턴할 수 있도록 한다. 가장 중요한 쓰임새는 람다 식에서 리턴하는 것이다. 다음 코드를 보자:

```
fun foo() {  
    ints.forEach {  
        if (it == 0) return  
        print(it)  
    }  
}
```

return 식은 가장 가깝게 둘러싼 함수인 `foo` 에서 리턴한다. (비-로컬(non-local) 리턴은 [인라인 함수](#)로 전달한 람다 식만 지원한다.) 만약 람다 식에서 리턴하고 싶다면 라벨을 붙여서 **return**을 한정해야 한다.

```
fun foo() {
    ints.forEach lit@ {
        if (it == 0) return@lit
        print(it)
    }
}
```

이제 이 return은 람다식에서 리턴한다. 종종 임의 라벨을 사용하는게 더 편리하다. 임의 라벨은 람다를 전달받은 함수와 같은 이름을 갖는다.

```
fun foo() {
    ints.forEach {
        if (it == 0) return@forEach
        print(it)
    }
}
```

다른 방법으로 람다 식 대신 [임의 함수](#)를 사용할 수 있다. 임의 함수에서 return 문은 임의 함수 자체에서 리턴한다.

```
fun foo() {
    ints.forEach(fun(value: Int) {
        if (value == 0) return
        print(value)
    })
}
```

값을 리턴할 때 파서는 한정한 리턴에 우선순위를 준다.

```
return@a 1
```

이는 “라벨 식 (@a 1) 을 리턴한다”가 아닌 “라벨 @a 에 1 을 리턴한다”를 의미한다.

클래스와 오브젝트

클래스와 상속

클래스

코틀린은 `class` 키워드를 사용해서 클래스를 선언한다.

```
class Invoice {  
}
```

클래스 선언은 클래스 이름, 클래스 헤더(타입 파라미터 지정, 주요 생성자 등)와 중괄호로 둘러 싼 클래스 몸체로 구성된다. 헤더와 몸체는 필수가 아니다. 클래스가 몸체를 갖지 않으면 중괄호를 생략할 수 있다.

```
class Empty
```

생성자

코틀린 클래스는 한 개의 **주요(primary) 생성자**와 한 개 이상의 **보조(secondary) 생성자**를 가질 수 있다. 주요 생성자는 클래스 헤더에 속한다. 클래스 헤더는 클래스 이름 뒤에(타입 파라미터가 있다면 그 뒤에) 위치한다.

```
class Person constructor(firstName: String) {  
}
```

주요 생성자가 애노테이션이나 가시성 제한자를 갖지 않으면 `constructor` 키워드를 생략할 수 있다.

```
class Person(firstName: String) {  
}
```

주요 생성자는 어떤 코드도 포함할 수 없다. 초기화 코드는 `init` 키워드를 이용한 **initializer 블록**에 위치할 수 있다.

```
class Customer(name: String) {  
    init {  
        logger.info("Customer initialized with value ${name}")  
    }  
}
```

주요 생성자의 파라미터는 initializer 블록과 클래스 몸체에 선언한 프로퍼티 initializer에서 사용할 수 있다.

```
class Customer(name: String) {
    val customerKey = name.toUpperCase()
}
```

코틀린은 주요 생성자에 프로퍼티를 선언하고 초기화할 수 있는 간결한 구문을 제공한다.

```
class Person(val firstName: String, val lastName: String, var age: Int) {
    // ...
}
```

일반 프로퍼티와 동일하게 주요 생성자에 선언한 프로퍼티도 변경 가능(**var**)하거나 읽기 전용(**val**)일 수 있다.

생성자가 애노테이션이나 가시성 제한자를 가질 경우 **constructor** 키워드가 필요하며 키워드 앞에 제한자가 위치하게 된다.

```
class Customer public @Inject constructor(name: String) { ... }
```

자세한 내용은 [가시성 제한자](#)를 참고한다.

보조 생성자(Secondary Constructors)

클래스는 **constructor**를 이용해서 **보조 생성자**를 선언할 수 있다.

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

클래스가 주요 생성자를 가질 경우, 각 보조 생성자는 직접 주요 생성자에 위임하거나 다른 보조 생성자를 통해 간접적으로 주요 생성자에 위임해야 한다. **this** 키워드를 이용해서 같은 클래스의 다른 생성자에 위임할 수 있다.

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

추상이 아닌 클래스가 어떤 생성자도 선언하지 않으면 인자를 갖지 않은 주요 생성자를 만든다. 클래스가 공개(**public**) 생성자를 갖지 않기를 원하면 기본 가시성이 아닌 다른 가시성을 갖는 빈(**empty**) 주요 생성자를 추가해야 한다.

```
class DontCreateMe private constructor () {
}
```

주의: JVM에서, 주요 생성자의 모든 파라미터가 기본 값을 가지면 컴파일러는 기본 값을 사용하는 파라미터 없는 생성자를 추가로 만든다. 이는 Jackson이나 JPA처럼 파라미터없는 생성자를 이용해서 클래스 인스턴스를 생성하는 라이브러리에 대해 코틀린을 사용하기 쉽게 만들어준다.

```
class Customer(val customerName: String = "")
```

클래스의 인스턴스 생성하기

클래스의 인스턴스를 생성하려면 일반 함수와 비슷하게 생성자를 호출한다.

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

코틀린은 `new`가 없음에 유의한다.

클래스 멤버

클래스는 다음을 갖는다.

- 생성자와 initializer 블록
- [함수](#)
- [프로퍼티](#)
- [중첩 클래스와 내부 클래스](#)
- [오브젝트 선언](#)

상속

코틀린의 모든 클래스는 공통의 상위클래스(superclass)인 `Any` 를 갖는다. `Any` 는 상위타입(supertype)을 지정하지 않은 클래스의 기본 상위타입이다.

```
class Example // 기본으로 Any를 상속한다
```

`Any` 는 `java.lang.Object` 가 아니다. 특히 `equals()`, `hashCode()`, `toString()` 외에 다른 멤버를 갖지 않는다. 보다 자세한 내용은 [자바 상호운용](#)을 참고한다.

상위타입을 직접 선언하려면 클래스 헤더에 콜론 뒤에 타입을 위치시킨다.

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

클래스가 주요 생성자를 가지면 주요 생성자의 파라미터를 이용해서 베이스 타입을 바로 초기화할 수 있다(그리고 반드시 초기화해야 한다).

클래스가 주요 생성자를 갖지 않으면 각 보조 생성자는 `super` 키워드를 사용해서 베이스 타입을 초기화하거나 그것을 하는 다른 생성자에 위임해야 한다. 위임받은 다른 보조 생성자는 베이스 타입의 다른 생성자를 호출할 수 있다.


```
class MyView : View {
    constructor(ctx: Context) : super(ctx) {
    }

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs) {
    }
}
```

클래스의 **open** 애노테이션은 자바의 **final**과 정반대이다. **open**은 다른 클래스가 이 클래스를 상속할 수 있도록 한다. 기본적으로 코틀린에서 모든 클래스는 **final**이다. 이는 [Effective Java](#)의 Item 17: *Design and document for inheritance or else prohibit it* 내용을 따른 것이다.

멤버 오버라이딩

앞서 언급한 것처럼 코틀린에서는 뭐든 명시적으로 만드는 것을 고수한다. 자바와 달리 코틀린은 오버라이딩 가능한 멤버(**열렸** **open**다고 함)와 오버라이드를 위해 명시적으로 애노테이션을 붙여야 한다.

```
open class Base {
    open fun v() {}
    fun nv() {}
}
class Derived() : Base() {
    override fun v() {}
}
```

`Derived.v()` 는 **override** 애노테이션이 필요하다. 붙이지 않으면 컴파일 에러가 발생한다. `Base.nv()` 와 같이 함수에 **open**을 붙이지 않으면 **override** 여부에 상관없이 하위클래스에서 동일 시그니처를 갖는 메서드를 선언할 수 없다. (**open** 애노테이션이 없는) **final** 클래스에는 **open** 멤버가 금지된다.

override를 갖는 멤버는 그 자체로 **open**이며 하위클래스에서 오버라이딩할 수 있다. 오버라이딩을 막고 싶다면 **final**을 사용하면 된다.

```
open class AnotherDerived() : Base() {
    final override fun v() {}
}
```

잠깐? 내가 만든 라이브러리를 어떻게 해킹하지?!

오버라이딩에 대해 클래스와 멤버가 기본적으로 **final**인 접근 방식은 한 가지 이슈가 있다. 그것은 바로 라이브러리 설계자가 오버라이딩하도록 의도하지 않은 어떤 메서드를 오버라이딩하거나 무언가 위험한(**nasty**) 해킹을 위해 라이브러리에 있는 것을 상속하기 어렵다는 점이다.

우리는 다음과 같은 이유로 이는 단점이 아니라 생각한다.

- 어쨌든 이런 류의 해킹은 허용하지 않는 것이 모범 사례이다
- 유사한 방식을 갖는 다른 언어(C++, C#)를 성공적으로 사용하고 있다
- 실제로 해킹을 원한다면 방법이 존재한다. 자바로 해킹 코드를 작성해서 코틀린에서 호출할 수 있고([자바 상호운용 참고](#)), **Aspect** 프레임워크로 해킹할 수 있다.

오버라이딩 규칙

코틀린에서 구현 상속은 다음 규칙을 따른다. 클래스가 바로 상위의 여러 상위클래스에서 같은 멤버 구현을 상속하면, 반드시 이 멤버를 오버라이딩하고 자신의 구현을 제공해야 한다(대개 상속받은 것 중의 하나를 사용하는 구현을 제공). 사용할 상위타입의 구현을 지정하려면 화살괄호에 상위타입 이름을 지정한 `super`를 사용한다(예, `super<Base>`).

```
open class A {
    open fun f() { print("A") }
    fun a() { print("a") }
}

interface B {
    fun f() { print("B") } // 인터페이스의 멤버는 기본적으로 'open'이다
    fun b() { print("b") }
}

class C() : A(), B {
    // 컴파일하려면 f()를 오버라이딩해야 한다.
    override fun f() {
        super<A>.f() // call to A.f()
        super<B>.f() // call to B.f()
    }
}
```

A와 B를 상속받는 것은 괜찮으며, a()와 b()에는 상속 관련 문제가 없다. 왜냐면 C는 이 두 함수에 대해 각각 한 개의 구현만 상속받기 때문이다. 하지만 f()의 경우 C가 두 개의 구현을 상속받기 때문에 모호함을 제거하기 위해 C에 f() 구현을 제공해야 한다.

추상 클래스

클래스와 멤버를 `abstract`로 선언할 수 있다. 추상 멤버는 구현을 갖지 않는다. 추상 클래스나 함수는 `open`을 붙일 필요가 없다.

추상이 아닌 `open` 멤버를 추상 멤버로 오버라이딩할 수 있다.

```
open class Base {
    open fun f() {}
}

abstract class Derived : Base() {
    override abstract fun f()
}
```

컴페니언 오브젝트(Companion Objects)

자바나 C#과 달리 코틀린의 클래스는 정적 메서드를 갖지 않는다. 많은 경우 그 대신 패키지 수준의 함수를 사용할 것을 권한다.

만약 클래스 인스턴스 없이 클래스의 내부에 접근해야 하는 함수를 작성하고 싶다면(예, 팩토리 메서드) 그 클래스 안에 [선언한 오브젝트](#)의 멤버로 함수를 작성할 수 있다.

좀 더 구체적으로 말하면 클래스 안에 [컴페니언 오브젝트](#)를 선언하면 클래스 이름만으로 자바/C#의 정적 메서드를 호출하는 것처럼 컴페니언 오브젝트의 멤버를 호출할 수 있다.

실드 클래스(Sealed Classes)

가질 수 있는 타입을 특정 타입 집합으로 제한하고 다른 타입은 가질 수 없도록 하고 싶을 때, 클래스 상속을 제한할 목적으로 실드 클래스를 사용한다. 실드 클래스는 어떤 의미에서 열거형 클래스의 확장이다. 열거 타입은 제한된 값 집합을 갖지만, 각 열거형 상수의 인스턴스는 오직 한 개만 존재한다. 반면에 실드 클래스의 하위클래스는 상태를 포함할 수 있는 여러 인스턴스를 가질 수 있다.

실드 클래스를 선언하려면 클래스 이름 앞에 `sealed` 제한자를 쓰면 된다. 실드 클래스는 하위 클래스를 가질 수 있지만 모든 하위 클래스는 실드 클래스 선언 자체에 중첩해야 한다.

```
sealed class Expr {  
    class Const(val number: Double) : Expr()  
    class Sum(val e1: Expr, val e2: Expr) : Expr()  
    object NotANumber : Expr()  
}
```

실드 클래스의 하위 클래스를 확장하는 클래스(indirect inheritors)는 어디든 위치할 수 있다. 실드 클래스 선언 내부에 위치할 필요는 없다.

실드 클래스의 최대 장점은 [when 식](#)과 함께 사용할 수 있다는 점이다. 모든 경우를 확실하게 다루고 있다면 `else` 절을 추가할 필요가 없다.

```
fun eval(expr: Expr): Double = when(expr) {  
    is Expr.Const -> expr.number  
    is Expr.Sum -> eval(expr.e1) + eval(expr.e2)  
    Expr.NotANumber -> Double.NaN  
    // 모든 경우를 다루기 때문에 `else` 절이 필요없다  
}
```

프로퍼티와 필드

프로퍼티 선언

코틀린 클래스는 프로퍼티를 가질 수 있다. **var** 키워드로 변경 가능 프로퍼티를 선언하고 **val** 키워드로 읽기 전용 프로퍼티를 선언한다.

```
public class Address {
    public var name: String = ...
    public var street: String = ...
    public var city: String = ...
    public var state: String? = ...
    public var zip: String = ...
}
```

프로퍼티를 사용하려면 자바 필드처럼 단순히 이름으로 참조하면 된다.

```
fun copyAddress(address: Address): Address {
    val result = Address() // 코틀린은 'new' 키워드가 없다
    result.name = address.name // accessor 실행
    result.street = address.street
    // ...
    return result
}
```

Getter와 Setter

프로퍼티 선언의 전체 구문은 다음과 같다.

```
var <propertyName>: <PropertyType> [= <property_initializer>]
    [<getter>]
    [<setter>]
```

initializer, getter, setter는 선택 사항이다. initializer나 오버라이딩한 베이스 클래스 멤버에서 타입을 유추할 수 있다면 프로퍼티 타입을 생략해도 된다.

예:

```
var allByDefault: Int? // 예러: initializer 필요, 기본 getter와 setter 포함
var initialized = 1 // Int 타입, 기본 getter와 setter 가짐
```

읽기 전용 프로퍼티 선언의 전체 구문은 변경 가능 프로퍼티와 두 가지가 다르다. **var** 대신에 **val** 로 시작하고 **setter**를 허용하지 않는다.

```
val simple: Int? // Int 타입과 기본 getter를 갖고, 생성자에서 초기화해야 한다
val inferredType = 1 // Int 타입과 기본 getter를 갖는다
```

프로퍼티 선언에서 바로 뒤에 일반 함수와 유사한 커스텀 accessor를 작성할 수 있다. 다음은 커스텀 getter의 예다.

```
val isEmpty: Boolean
    get() = this.size == 0
```

다음은 커스텀 setter의 예다.

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // 문자열을 파싱해서 다른 프로퍼티에 할당한다
    }
```

보통 setter의 파라미터 이름으로 `value`를 사용하지만 원하는 다른 이름을 사용해도 된다.

accessor의 기본 구현을 변경하지 않고 가시성을 변경하거나 애노테이션을 적용하고 싶다면 몸체 선언 없이 accessor를 정의할 수 있다.

```
var setterVisibility: String = "abc"
    private set // setter는 private이고 기본 구현을 갖는다

var setterWithAnnotation: Any? = null
    @Inject set // setter에 @Inject를 붙인다
```

Backing 필드

코틀린 클래스는 필드를 가질 수 없다. 하지만 커스텀 accessor를 사용하면 backing 필드가 필요할 때가 있다. 이를 위해 코틀린은 `field` 식별자를 사용해서 접근할 수 있는 backing 필드를 제공한다.

```
var counter = 0 // 초기화 값을 backing 필드에 직접 쓴다
    set(value) {
        if (value >= 0)
            field = value
    }
```

`field` 식별자는 프로퍼티 accessor에서만 사용할 수 있다.

accessor 몸체에서 backing 필드를 사용하면(또는 기본 구현을 사용하면) 컴파일러는 backing 필드를 생성한다. 그렇지 않으면 생성하지 않는다.

예를 들어 다음 경우 backing 필드가 없다.

```
val isEmpty: Boolean
    get() = this.size == 0
```

Backing 프로퍼티

“기본(implicit) backing 필드” 방식과 맞지 않는 것을 하고 싶다면 *backing 프로퍼티*로 대체할 수 있다.

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null)
            _table = HashMap() // 타입 파라미터를 유추
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

기본 getter와 setter로 private 프로퍼티에 접근하는 것을 최적화해서 함수 호출에 따른 오버헤드가 없기 때문에 모든 점에서 자바와 같다.

컴파일 타임 상수

컴파일 타임에 값을 알 수 있는 프로퍼티는 `const` 제한자를 이용해서 `_컴파일 타임 상수_`로 지정할 수 있다. 이 프로퍼티는 다음 요건을 충족해야 한다.

- 최상위 레벨 또는 오브젝트의 멤버
- `String` 이나 기본 타입 값으로 초기화
- 커스텀 getter 없음

이 프로퍼티는 애노테이션에서 사용할 수 있다.

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"

@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

초기화 지연(Late-Initialized) 프로퍼티

보통 non-null 타입을 갖는 프로퍼티를 선언하면 생성자에서 초기화해야 한다. 하지만 이게 늘 편한 것은 아니다. 예를 들어 의존 주입이나 단위 테스트의 셋업 메서드에서 프로퍼티를 초기화할 수 있다. 이 경우 생성자에 non-null initializer를 제공할 수 없지만 클래스 몸체 안에서 프로퍼티를 참조할 때 null 검사를 하고 싶지는 않을 것이다.

이 경우를 처리하기 위해 프로퍼티를 `lateinit` 로 제한할 수 있다.

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method() // null 검사 없이 직접 접근
    }
}
```

이 제한자는 (주요 생성자가 아닌) 클래스 몸체에 선언되고 커스텀 getter나 setter를 갖지 않는 `var` 프로퍼티에만 사용할 수 있다. 프로퍼티 타입은 non-null이어야 하고 기본 타입이면 안 된다.

초기화되기 전에 `lateinit` 프로퍼티에 접근하면 초기화되지 않은 프로퍼티에 접근했음을 정확하게 알려주기 위해 특수한 익셉션을 발생한다.

프로퍼티 오버라이딩

[멤버 오버라이딩](#) 참고

위임 프로퍼티

대부분 프로퍼티는 단순히 `backing` 필드에서 값을 읽거나 쓴다. 반면에 커스텀 `getter`와 `setter`를 갖는 프로퍼티는 프로퍼티의 행위를 구현할 수 있다. 프로퍼티 행위 구현에는 프로퍼티가 어떻게 작동하는지에 대한 어떤 공통 패턴이 있다. 키로 맵에서 읽기, 데이터 베이스 접근하기, 접근 시 리스너에 통지하기 등이 공통 패턴에 해당한다.

이런 공통 행위는 [위임 프로퍼티](#)를 사용하는 라이브러리로 구현할 수 있다.

인터페이스

코틀린 인터페이스는 자바 8과 매우 유사하다. 추상 메서드뿐만 아니라 메서드 구현을 가질 수 있다. 추상 클래스와의 차이점은 인터페이스는 상태를 가질 수 없다는 점이다. 프로퍼티를 가질 수는 있지만 추상이거나 accessor 구현을 제공해야 한다.

`interface` 키워드로 인터페이스를 정의한다.

```
interface MyInterface {  
    fun bar()  
    fun foo() {  
        // 몸체 선택  
    }  
}
```

인터페이스 구현

클래스나 오브젝트는 한 개 이상의 인터페이스를 구현할 수 있다.

```
class Child : MyInterface {  
    override fun bar() {  
        // 몸체  
    }  
}
```

인터페이스의 프로퍼티

인터페이스에 프로퍼티를 선언할 수 있다. 인터페이스에 선언한 프로퍼티는 추상이거나 accessor를 위한 구현을 제공해야 한다. 인터페이스의 프로퍼티는 backing 필드를 가질 수 없으므로 인터페이스에 선언한 accessor는 backing 필드를 참조할 수 없다.

```
interface MyInterface {  
    val property: Int // 추상  
  
    val propertyWithImplementation: String  
        get() = "foo"  
  
    fun foo() {  
        print(property)  
    }  
}  
  
class Child : MyInterface {  
    override val property: Int = 29  
}
```

오버라이딩 충돌 해결

상위타입 목록에 여러 타입을 지정하면 같은 메서드에 대해 한 개 이상의 구현을 상속받기도 한다. 다음 예를 보자.


```

interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }
}

```

*A*와 *B* 인터페이스가 *foo()*와 *bar()* 함수를 선언하고 있다. *foo()*는 두 인터페이스가 모두 구현하고 있고 *bar()*는 *B*만 구현하고 있다 (*A*에서 *bar()*를 *abstract*로 지정하지 않았는데 그 이유는 인터페이스에서는 함수에 몸체가 없으면 기본으로 추상이기 때문이다). 컨크리트 클래스 *C*가 *A*를 상속받으면 *C*는 명백하게 *bar()*를 오버라이딩해서 구현을 제공해야 한다. 그리고 *A*와 *B*를 상속받은 *D*는 *bar()*를 오버라이딩 할 필요가 없다. 왜냐면 한 개 구현만 상속받기 때문이다. 하지만 *foo()*는 두 개 구현을 상속받기 때문에 컴파일러는 둘 중 무엇을 선택해야 하는지 알 수 없다. 따라서 컴파일러는 *foo()*를 명시적으로 오버라이딩할 것을 강제한다.

가시성 제한자

클래스, 오브젝트, 인터페이스, 생성자, 함수, 프로퍼티와 프로퍼티의 setter는 _가시성 제한자_를 가질 수 있다(getter는 항상 프로퍼티와 동일한 가시성을 갖는다). 코틀린에는 `private`, `protected`, `internal`, `public` 의 네 개의 가시성 제한자가 존재한다. 명시적으로 제한자를 지정하지 않으면 기본적으로 `public` 가시성을 갖는다.

각 스코프의 선언별로 제한자에 대한 설명은 아래를 참고한다.

패키지

함수, 프로퍼티와 클래스, 오브젝트와 인터페이스는 “최상위 레벨”로 선언할 수 있다. 예를 들어 패키지 안에 바로 선언할 수 있다.

```
// 파일 이름: example.kt
package foo

fun baz() {}
class Bar {}
```

- 만약 가시성 제한자를 지정하지 않으면 기본으로 `public` 을 사용한다. 모든 곳에서 해당 선언에 접근할 수 있다.
- `private` 으로 지정하면 해당 선언을 포함한 파일에서만 접근할 수 있다.
- `internal` 로 지정하면 같은 모듈 안에서 접근할 수 있다.
- 최상위 레벨 선언은 `protected` 로 지정할 수 없다.

예제:

```
// 파일 이름: example.kt
package foo

private fun foo() {} // example.kt에서만 접근 가능

public var bar: Int = 5 // 프로퍼티는 모든 곳에서 접근 가능
    private set // setter는 example.kt에서만 접근 가능

internal val baz = 6 // 같은 모듈 안에서 접근 가능
```

클래스와 인터페이스

클래스 안에서 선언할 때:

- `private` 은 (클래스의 모든 멤버를 포함한) 클래스 안에서만 접근 가능하다.
- `protected` — `private` 과 동일 + 하위클래스에서 접근 가능하다.
- `internal` — 선언한 클래스에 접근할 수 있는 *모듈에 속한* 모든 클라이언트가 `internal` 멤버에 접근 가능하다.
- `public` — 선언한 클래스에 접근할 수 있는 모든 클라이언트가 `public` 멤버에 접근 가능하다.

자바 개발자 대상 주의: 코틀린에서 외부 클래스는 자신의 내부 클래스에 있는 `private` 멤버에 접근할 수 없다.

예제:

```

open class Outer {
    private val a = 1
    protected val b = 2
    internal val c = 3
    val d = 4 // 기본으로 public

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
    // a에 접근 불가
    // b, c 그리고 d에 접근 가능
    // Nested와 e에 접근 가능
}

class Unrelated(o: Outer) {
    // o.a, o.b에 접근 불가
    // o.c(같은 모듈)와 o.d에 접근 가능
    // Outer.Nested에 접근 불가, 그리고 Nested::e에도 접근 불가
}

```

생성자

클래스의 주요 생성자에 가시성을 지정하려면 다음 구문을 사용한다(**constructor** 키워드를 사용해야 한다):

```

class C private constructor(a: Int) { ... }

```

여기서 생성자는 **private**이다. 모든 생성자의 기본 가시성은 **public**이며 클래스에 접근 가능한 모든 곳에서 접근 가능하다(예를 들어 **internal** 클래스의 생성자는 오직 동일 모듈에서만 접근 가능하다).

로컬 선언

로컬에 선언한 변수, 함수, 클래스는 가시성 제한자를 가질 수 없다.

모듈

internal 가시성 제한자는 같은 모듈에서 멤버에 접근할 수 있다. 더 구체적으로 말하면 모듈은 함께 컴파일되는 코틀린 파일 집합이다.

- IntelliJ IDEA 모듈;
- Maven이나 Gradle 프로젝트;
- 한 Ant 태스크 실행 시 컴파일되는 파일 집합.

확장

C#이나 Gosu와 비슷하게 코틀린은 클래스를 확장하거나 데코레이터와 같은 디자인 패턴을 사용하지 않고 클래스에 새 기능을 확장하는 기능을 제공한다. `_확장(extension)_`이라 불리는 특수한 선언을 이용해서 확장할 수 있다. 코틀린은 `_확장 함수_`와 `_확장 프로퍼티_`를 지원한다.

확장 함수

확장 함수를 선언하려면 *리시버 타입(receiver type)*(확장할 타입)의 이름을 접두어로 사용해야 한다. 다음은 `MutableList<Int>`에 `swap` 함수를 추가한다.

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this'는 리스트에 해당한다
    this[index1] = this[index2]
    this[index2] = tmp
}
```

확장 함수에서 `this` 키워드는 리시버 객체(점 기호 앞에 전달되는 객체)에 해당한다. 이제 `MutableList<Int>`에 대해 다음과 같이 함수를 호출할 수 있다.

```
val l = mutableListOf(1, 2, 3)
l.swap(0, 2) // 'swap()'에서 'this'는 'l'의 값을 갖는다.
```

물론 이 함수는 모든 `MutableList<T>`에 의미가 있으므로 지네릭으로 만들 수 있다.

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this'는 리스트에 해당한다
    this[index1] = this[index2]
    this[index2] = tmp
}
```

함수 이름 앞에 리시버 타입 식에서 사용가능한 타입 파라미터를 선언한다. [지네릭 함수](#)를 참고한다.

정적인 확장 결정

확장은 실제로 확장할 클래스를 수정하지 않는다. 확장을 정의하면 클래스에 새 멤버를 추가하는 것이 아니라 단지 그 클래스의 인스턴스에 대해 점-기호로 호출할 수 있는 새로운 함수를 만드는 것뿐이다.

실행할 확장 함수는 **정적으로** 결정한다! 예를 들어 리시버 타입에 따라 버추얼하게 결정하지 않는다. 확장 함수를 호출하는 코드의 타입으로 호출할 확장 함수를 결정한다. 런타임에 그 식을 평가한 결과 타입으로 결정하지 않는다. 다음 예를 보자:

```

open class C

class D: C()

fun C.foo() = "c"

fun D.foo() = "d"

fun printFoo(c: C) {
    println(c.foo())
}

printFoo(D())

```

이 예는 “C”를 출력한다. 왜냐면 호출할 확장 함수를 선택할 때 `c` 파라미터의 선언 타입인 `C` 클래스만 사용하기 때문이다.

만약 클래스가 멤버 함수를 갖고 동일 리시버 타입을 갖는 동일 이름의 확장 함수가 있고 주어진 인자를 적용할 수 있다면 **항상 멤버가 이긴다**. 다음 예를 보자:

```

class C {
    fun foo() { println("member") }
}

fun C.foo() { println("extension") }

```

타입이 `C` 인 `c` 에 대해 `c.foo()` 를 호출하면 “extension”이 아닌 “member”를 출력한다..

Nullable 리시버

nullable 리시버 타입을 정의할 수도 있다. 이 확장은 비록 객체 변수가 null이어도 호출되며 확장 함수 몸체에서 `this == null` 로 검사할 수 있다. 이것이 코틀린에서 null 검사 없이 `toString()`을 호출할 수 있는 이유이다. 확장 함수 안에서 null 여부를 검사한다.

```

fun Any?.toString(): String {
    if (this == null) return "null"
    // 검사 이후에 'this'를 non-null 타입으로 자동 변환하므로
    // 다음의 toString()은 Any 클래스의 멤버 함수를 사용한다.
    return toString()
}

```

확장 프로퍼티

함수와 유사하게 코틀린은 확장 프로퍼티를 지원한다.

```

val <T> List<T>.lastIndex: Int
    get() = size - 1

```

확장은 실제로 클래스에 멤버를 추가하지 않으므로 확장 프로퍼티가 [backing 필드](#)를 가질 방법은 없다. 이것이 **확장 프로퍼티에 대해 initializer를 허용하지 않는** 이유이다. 프로퍼티 기능은 명시적으로 제공하는 getter/setter로만 가능하다.

예제:

```
val Foo.bar = 1 // 에러: 확장 프로퍼티에 대한 initializer는 허용하지 않음
```

컴페니언 오브젝트 확장

클래스가 [컴페니언 오브젝트](#)를 가지면 컴페니언 오브젝트에 대해서 함수와 프로퍼티를 확장할 수 있다.

```
class MyClass {  
    companion object { } // "Companion"으로 불림  
}  
  
fun MyClass.Companion.foo() {  
    // ...  
}
```

컴페니언 오브젝트의 다른 멤버처럼 클래스 이름을 사용하여 확장을 호출할 수 있다.

```
MyClass.foo()
```

확장의 범위

대부분 패키지 하위의 최상위 레벨에 직접 확장을 정의한다.

```
package foo.bar  
  
fun Baz.goo() { ... }
```

패키지 밖에서 이런 확장을 사용하려면 사용 측에서 확장을 임포트해야 한다.

```
package com.example.usage  
  
import foo.bar.goo // "goo"의 모든 확장을 임포트  
// 또는  
import foo.bar.*   // "foo.bar"로부터 모두 임포트  
  
fun usage(baz: Baz) {  
    baz.goo()  
}
```

더 많은 정보는 [임포트](#)를 참고한다.

멤버로 확장을 선언하기

클래스 안에서 다른 클래스를 위한 확장을 선언할 수 있다. 그 확장 안에는 *리시버가 암묵적(implicit)으로* (한정자(qualifier) 없이 접근할 수 있는 오브젝트 멤버) 다수 존재한다. 확장을 선언한 클래스의 인스턴스를 `_디스패치(dispatch) 리시버_`라 부르고, 확장 메서드의 리시버 타입 인스턴스를 `_확장 리시버_`라고 부른다.

```

class D {
    fun bar() { ... }
}

class C {
    fun baz() { ... }

    fun D.foo() {
        bar()    // D.bar 호출
        baz()    // C.baz 호출
    }

    fun caller(d: D) {
        d.foo()  // 확장 함수 호출
    }
}

```

디스패치 리시버와 확장 리시버의 멤버 이름이 충돌하면 확장 리시버가 우선한다. 디스패치 리시버의 멤버를 참조하려면 [한정한 this 구문](#)을 사용하면 된다.

```

class C {
    fun D.foo() {
        toString()    // D.toString() 호출
        this@C.toString() // C.toString() 호출
    }
}

```

멤버로 선언한 확장을 `open` 으로 설정하면 하위클래스에서 오버라이딩할 수 있다. 이는 확장 함수를 디스패치 리시버 타입에 따라 선택한다는 것을 의미한다. 하지만 확장 리시버 타입은 정적이다.

```

open class D {
}

class D1 : D() {
}

open class C {
    open fun D.foo() {
        println("D.foo in C")
    }

    open fun D1.foo() {
        println("D1.foo in C")
    }

    fun caller(d: D) {
        d.foo()    // 확장 함수 호출
    }
}

class C1 : C() {
    override fun D.foo() {
        println("D.foo in C1")
    }

    override fun D1.foo() {
        println("D1.foo in C1")
    }
}

C().caller(D())    // "D.foo in C" 출력
C1().caller(D())   // "D.foo in C1" 출력 - 디스패치 리시버를 버추얼하게 선택
C().caller(D1())   // "D.foo in C" 출력 - 확장 리시버를 정적으로 선택

```

동기

자바에서는 `FileUtils`, `StringUtils` 처럼 “Utils”라는 이름을 갖는 클래스에 익숙하다. 잘 알려진 `java.util.Collections` 도 이에 속한다. 이런 유틸리티 클래스가 싫은 이유는 코드가 다음과 같은 모습을 띄기 때문이다.

```

// Java
Collections.swap(list, Collections.binarySearch(list, Collections.max(otherList)),
Collections.max(list))

```

클래스 이름이 항상 방해가 된다. 정적 임포트를 사용하면 다음 코드가 된다.

```

// Java
swap(list, binarySearch(list, max(otherList)), max(list))

```

조금 나아졌지만 IDE의 강력한 코드 완성 기능의 도움을 거의 받지 못한다. 다음 코드처럼 할 수 있다면 훨씬 나을 것이다.


```
// Java  
list.swap(list.binarySearch(otherList.max()), list.max())
```

하지만 `List` 클래스에 모든 가능한 메서드를 구현하는 것은 원치 않는다, 그렇지 않나? 이것이 확장이 우리를 돕는 지점이다.

데이터 클래스

종종 데이터만 갖고 다른 건 하지 않는 클래스를 만든다. 보통 그런 클래스의 표준 기능은 데이터로부터 기계적으로 만든다. 코틀린에서는 이를 `_데이터 클래스_`라 부르며 `data` 로 데이터 클래스를 지정한다.

```
data class User(val name: String, val age: Int)
```

컴파일러는 주요 생성자에 정의한 모든 프로퍼티로부터 다음의 멤버를 자동으로 생성한다.

- `equals()` / `hashCode()` 쌍
- `"User(name=John, age=42)"` 형식의 `toString()`
- 프로퍼티 선언 순서에 따라 프로퍼티별로 대응하는 [componentN\(\)](#) 함수
- `copy()` 함수 (아래 참고)

이 함수를 클래스 몸체에 정의하거나 베이스 타입에서 상속받을 경우 생성하지 않는다.

생성한 코드가 일관되고 의미있는 기능을 갖도록 하기 위해 데이터 클래스는 다음을 충족해야 한다.

- 주요 생성자는 최소 한 개 파라미터가 필요하다.
- 모든 주요 생성자 파라미터는 `val` 이나 `var` 로 지정해야 한다.
- 데이터 클래스는 추상, `open`, `실드`, 내부(`inner`)일 수 없다.
- 데이터 클래스는 다른 클래스를 확장할 수 없다(인터페이스 구현은 된다).

JVM의 경우, 생성한 클래스가 파라미터 없는 생성자를 필요로 하면 모든 프로퍼티에 대해 기본 값을 지정해야 한다([생성자](#) 참고).

```
data class User(val name: String = "", val age: Int = 0)
```

복사

객체를 복사할 때 종종 일부 프로퍼티만 변경하고 나머지는 그대로 유지하고 싶을 때가 있다. 이를 위해 `copy()` 함수를 생성한다. 앞서 `User` 클래스의 복사 구현은 다음과 같다.

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

다음과 같이 코드를 작성할 수 있다.

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

데이터 클래스와 분해 선언(Destructuring declarations)

데이터 클래스를 위해 생성한 `_컴포넌트 함수_`는 [분해 선언](#)에 데이터를 사용할 수 있도록 한다.

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // "Jane, 35 years of age" 출력
```

표준 데이터 클래스

표준 라이브러리는 `Pair` 와 `Triple` 을 제공한다. 많은 경우 이름을 갖는 데이터 클래스를 사용하는 것이 더 좋은 설계가 된다. 왜냐하면 프로퍼티를 위한 의미있는 이름을 제공함으로써 코드 가독성이 높아지기 때문이다.

지네릭

자바처럼 코틀린 클래스도 타입 파라미터를 가질 수 있다.

```
class Box<T>(t: T) {  
    var value = t  
}
```

이 클래스의 인스턴스를 생성하려면 타입 인자를 제공해야 한다.

```
val box: Box<Int> = Box<Int>(1)
```

생성자 인자나 다른 방법으로 파라미터를 유추할 수 있으면 타입 인자를 생략할 수 있다.

```
val box = Box(1) // 1은 Int 타입을 가지므로 컴파일러는 Box<Int>라고 알아낸다
```

가변(Variance)

자바 타입 시스템에서 가장 복잡한 부분 중 하나가 와일드카드 타입이다([자바 지네릭 FAQ](#) 참고). 코틀린에는 복잡한 와이드 카드가 없다. 대신 선언-위치 가변(declaration-site variance)과 타입 프로젝션(type projection)의 두 가지를 제공한다.

먼저 자바에서 미스테리한 와일드카드가 필요한 이유를 생각해보자. 이 문제를 [Effective Java, Item 28: Use bounded wildcards to increase API flexibility](#)에서 설명하고 있다. 첫째, 자바의 지네릭 타입은 **무공변(invariant)**이다. 이는 `List<String>`은 `List<Object>`의 하위타입이 **아님**을 의미한다. 왜 그랬을까? 만약 리스트가 **무공변(invariant)**이 아니면 자바 배열보다 나을 게 없다. 왜냐하면 다음 코드가 컴파일은 되지만 런타임에 익셉션이 발생하기 때문이다.

```
// Java  
List<String> strs = new ArrayList<String>();  
List<Object> objs = strs; // !!! 앞에서 언급한 문제 원인이 여기 있다. 자바는 이를 금지한다!  
objs.add(1); // 여기서 String의 리스트에 Integer를 넣는다  
String s = strs.get(0); // !!! ClassCastException: Integer를 String으로 변환할 수 없다
```

그래서 자바는 런타임 안정성을 보장하기 위해 이런 것을 금지했다. 하지만 이는 몇 가지 영향을 준다. 예를 들어 `Collection` 인터페이스의 `addAll()` 메서드를 생각해보자. 이 메서드의 시그니처는 무엇인가? 직관적으로 생각하면 다음과 같이 작성할 수 있다.

```
// Java  
interface Collection<E> ... {  
    void addAll(Collection<E> items);  
}
```

하지만 (완벽하게 안전한 코드임에도) 다음의 간단한 코드를 만들 수 없다.

```
// Java
void copyAll(Collection<Object> to, Collection<String> from) {
    to.addAll(from); // !!! addAll의 단순한 선언으로는 컴파일되지 않는다.
                    //      Collection<String>은 Collection<Object>의 하위 타입이 아니다
}
```

(우리는 이를 힘들게 배웠다. [Effective Java](#), Item 25: *Prefer lists to arrays*를 참고하자.)

이런 이유로 실제 `addAll()` 시그니처는 다음과 같다.

```
// Java
interface Collection<E> ... {
    void addAll(Collection<? extends E> items);
}
```

와일드카드 타입 인자 ? `extends E` 는 이 메서드가 `E` 자체가 아닌 `E` 의 *하위타입*의 객체 컬렉션을 허용한다는 것을 말한다. 이는 `items`에서 안전하게 `E` 로 읽을 수 있지만(이 컬렉션의 요소는 `E`의 하위클래스의 인스턴스이다), `E` 의 어떤 하위타입인지 모르기 때문에 `items`에 쓸 수 없다는 것을 의미한다. 이런 제한을 해소하기 위해 `Collection<String>` 이 `Collection<? extends Object>` 의 하위타입이 되도록 기능을 추가했다. “전문 용어”로 **extends-bound(upper bound)**를 갖는 와일드카드를 사용해서 타입을 **공변(convariant)**으로 만들었다.

이 트릭이 왜 작동하는지 이해하는데 있어 핵심은 다소 단순하다. 만약 컬렉션에서 아이템을 가져올 수만 있다면 `String` 컬렉션에서 `Object` 를 읽는 것은 괜찮다. 역으로 컬렉션에 항목을 넣을 수만 있다면 `Object` 컬렉션에 `String` 을 넣는 건 괜찮다. 자바에서 `List<? super String>` 가 `List<Object>` 의 상위타입이 된다.

후자를 **반공변(contravariance)**이라 부르며, `List<? super String>` 에 인자로 `String`을 받는 메서드만 호출할 수 있다(예를 들어 `add(String)` 이나 `set(int, String)` 을 호출할 수 있다). 반면에 `List<T>` 에서 `T` 를 리턴하는 어떤 것을 호출하면 `String` 이 아닌 `Object` 를 얻게 된다.

Joshua Blochs는 이 객체는 **Producer**에서만 읽을 수 있고 **Consumer**로만 쓸 수 있다고 했다. Joshua Blochs는 “*유연함을 최대한 얻으려면 producer나 consumer를 표현하는 입력 파라미터에 와일드카드 타입을 사용하라*”고 권하고 있으며, 다음과 같이 기억을 쉽게 할 수 있는 약자를 제시했다.

PECS는 Producer-Extends, Consumer-Super를 의미한다.

주의: producer 객체를 사용한다면, 예를 들어 `List<? extends Foo>` , 이 객체에 대해 `add()` 나 `set()` 을 호출하는 것을 허용하지 않는다. 하지만 이것이 이 객체가 **불변(immutable)**인 것을 의미하는 것은 아니다. 예를 들어, 리스트의 모든 항목을 삭제하기 위해 `clear()` 를 호출하는 것은 가능하다. 왜냐면, `clear()` 는 어떤 파라미터도 갖지 않기 때문이다. 와일드카드(또는 다른 종류의 가변variance)가 보장하는 것은 **타입 안정성**이다. 불변은 완전히 다른 얘기다.

선언-위치 가변(Declaration-site variance)

`Source<T>` 지네릭 인터페이스에 `T` 를 파라미터로 갖는 메서드는 없고 단지 `T` 를 리턴하는 메서드만 있다고 하자:

```
// Java
interface Source<T> {
    T nextT();
}
```

이때 `Source<Object>` 타입 변수에 `Source<String>` 인스턴스를 할당하는 것은 완전히 안전한 것이다. 여기엔 어떤 consumer 메서드도 호출하지 않는다. 하지만 자바는 이를 알지 못하기 때문에 이를 금지한다.

```
// Java
void demo(Source<String> strs) {
    Source<Object> objects = strs; // !!! 자바는 허용하지 않음
    // ...
}
```

이 코드의 문제를 고치려면 `Source<? extends Object>` 타입 객체를 선언해야 하는데 이는 다소 의미가 없다. 왜냐면 수정 전에도 동일한 메서드를 호출할 수 있고 더 복잡한 타입으로 수정해도 값을 추가하지 않기 때문이다. 하지만 컴파일러는 이를 알지 못한다.

코틀린은 컴파일러에 이런 류의 내용을 설명하는 방법이 존재한다. 이를 **선언-위치 가변(declaration-site variance)**이라고 부른다. 소스 코드의 **타입 파라미터** `T`에 애노테이션을 붙여서 `Source<T>`의 멤버가 **리턴(생성)**만 하고 **소비(consume)**하지 않는다고 할 수 있다. 이를 위해 **out** 제한자를 제공한다.

```
abstract class Source<out T> {
    abstract fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs // T는 out 파라미터이므로 OK
    // ...
}
```

일반 규칙: 클래스 `C`의 타입 파라미터 `T`를 **out**으로 선언하면 타입 파라미터는 오직 `C` 멤버의 **out-위치**에만 올 수 있다. 하지만 리턴에서 `C<Base>`는 안전하게 `C<Derived>`의 상위타입이 될 수 있다.

“전문 용어”로 클래스 `C`는 파라미터 `T`에 **공변(covariant)**한다 또는 `T`는 **공변(covariant)** 타입 파라미터라고 말한다. `C`를 `T`의 **consumer**가 아닌 `T`의 **producer**로 생각할 수 있다.

out 제한자는 **가변(variance) 애노테이션**이라 부르며, 타입 파라미터 선언 위치에 제공하기 때문에 **선언-위치 가변(declaration-site variance)**에 대한 것이다. 이는 자바가 타입을 사용할 때 와일드카드로 타입을 공변(covariant)하게 만드는 **사용-위치 가변(use-site variance)**인 것과 다르다.

out과 더불어 코틀린은 대체 가변(variance) 애노테이션인 **in**을 제공한다. **in**은 타입 파라미터를 **반공변(contravariant)**으로 만들어 준다. 이는 오직 consume만 될 수 있으며 produce 할 수 없다. 반공변(contravariant) 클래스의 좋은 예가 `Comparable`이다.

```
abstract class Comparable<in T> {
    abstract fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0은 Number의 상위 타입은 Double 타입을 갖는다
    // 그래서, Comparable<Double> 타입 변수를 x에 할당할 수 있다
    val y: Comparable<Double> = x // OK!
}
```

단어 **in**과 **out**은 자명하므로(이미 꽤 오랜 시간 C#에서 성공적으로 사용하고 있다) 위에서 언급한 기억하기 위한 PECS가 실제로 필요 없고 더 상위 목표를 위해 바꿀 수 있다고 생각한다.

실존주의 변환: Consumer in, Producer out! :-)

타입 프로젝션(Type projections)

사용-위치 가변(Use-site variance): 타입 프로젝션

타입 파라미터 **T**를 *out*으로 선언하는 것은 매우 편리하고 사용 위치에서 하위타입 관련 문제가 없다. 좋다. 그런데 클래스가 실제로 **T** 만 리턴하도록 제한이 있을 때 그것으로 못하는 건 무얼까? **Array**가 좋은 예이다.

```
class Array<T>(val size: Int) {
    fun get(index: Int): T { /* ... */ }
    fun set(index: Int, value: T) { /* ... */ }
}
```

이 클래스는 **T**에 대해 공변(covariant)도 반공변(contravariant)도 될 수 없다. 게다가 유연하지 못하게 강제한다. 다음 함수를 보자:

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

이 함수는 한 배열에서 다른 배열로 항목을 복사한다. 실제로 함수 실행을 시도해보자:

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3)
copy(ints, any) // 예러: expects (Array<Any>, Array<Any>)
```

여기서 익숙한 문제가 발생한다. **Array<T>**는 **T**에 대해 **무공변(invariant)**하므로 **Array<Int>**와 **Array<Any>**는 서로 상대방의 하위타입이 아니다. 왜 그럴까? **copy**는 나쁜 짓을 **할지 모르기** 때문이다. 예를 들어, **String**을 **from**에 **쓰려고** 시도하는데 실제로 **from**에 **Int** 배열을 전달했다면 나중에 **ClassCastException**이 발생할 수 있다.

여기서 원하는 것은 **copy()**가 그런 나쁜 짓을 하지 않는 것을 보장하는 것이다. 이 메서드가 **from**에 **쓰지** 못하게 하려면 다음과 같이 하면 된다.

```
fun copy(from: Array<out Any>, to: Array<Any>) {
    // ...
}
```

여기서 사용한 것을 **타입 프로젝션(type projection)**이라고 한다. 이 코드에서 **from**은 단순 배열이 아닌 **제한된(projected)** 배열이다. 오직 타입 파라미터 **T**를 리턴하는 메서드만 호출할 수 있다. 이 예의 경우 **get()**만 호출할 수 있다. 이것이 코틀린의 **사용-위치 가변(use-site variance)** 접근 방식이다. 자바의 **Array<? extends Object>**에 해당하지만 더 간단하다.

in을 이용해서 타입을 프로젝션할 수 있다.

```
fun fill(dest: Array<in String>, value: String) {
    // ...
}
```

`Array<in String>` 은 자바의 `Array<? super String>` 에 해당하며 `CharSequence` 의 배열이나 `Object` 의 배열을 `fill()` 함수에 전달할 수 있다.

스타-프로젝션

때때로 타입 인자에 대해 알지 못하지만 안전한 방법으로 인자를 사용하고 싶을 때가 있다. 여기서 안전한 방법은 지네릭 타입에 그런 프로젝션을 정의해서, 지네릭 타입의 모든 컨크리트 인스턴스가 그 프로젝트의 하위 타입이 되도록 하는 것이다.

코틀린은 이를 위해 **스타-프로젝션(star-projection)**이라 불리는 구문을 제공한다.

- `Foo<out T>` 에 대해, `T` 가 uppber bound `TUpper` 를 갖는 공변(covariant) 타입 파라미터면 `Foo<*>` 은 `Foo<out TUpper>` 와 같다. 이는 `T` 를 몰라도 안전하게 `Foo<*>` 에서 `TUpper` 값을 읽을 수 있다는 것을 의미한다.
- `Foo<in T>` 에 대해, `T` 가 반공변(contravariant) 타입 파라미터면 `Foo<*>` 는 `Foo<in Nothing>` 와 같다. 이는 `T` 를 모를 때 안전하게 `Foo<*>` 에 쓸 수 없다는 것을 의미한다.
- `Foo<T>` 에 대해, `T` 가 uppber bound `TUpper` 를 갖는 무공변(invariant) 타입 파라미터면, `Foo<*>` 는 값을 읽을 때는 `Foo<out TUpper>` 와 동일하고 값을 쓸 때는 `Foo<in Nothing>` 와 동일하다.

지네릭 타입이 여러 타입 파라미터를 가질 경우 각각 독립적으로 프로젝션할 수 있다. 예를 들어 `interface Function<in T, out U>` 타입을 정의하면 다음의 스타-프로젝션을 생각할 수 있다.

- `Function<*, String>` 은 `Function<in Nothing, String>` 을 의미한다.
- `Function<Int, *>` 은 `Function<Int, out Any?>` 를 의미한다.
- `Function<*, *>` 은 `Function<in Nothing, out Any?>` 을 의미한다.

주의: 스타-프로젝션은 자바의 `raw` 타입과 매우 유사하지만 안전하다.

지네릭 함수

클래스만 타입 파라미터를 가질 수 있는 것은 아니다. 함수도 가질 수 있다. 함수 이름 앞에 타입 파라미터를 위치시키면 된다.

```
fun <T> singletonList(item: T): List<T> {
    // ...
}

fun <T> T.basicToString(): String { // 확장 함수
    // ...
}
```

타입 파라미터를 호출 위치(call site)에서 명시적으로 전달하려면 함수 이름 뒤에 지정한다.

```
val l = singletonList<Int>(1)
```

지네릭 제약

주어진 타입 파라미터를 대체할 수 있는 모든 가능한 타입은 **지네릭 제약(constraints)**에 따라 제한된다.

Upper bounds

가장 일반적인 제약은 자바의 *extends* 키워드에 해당하는 **upper bound**이다.

```
fun <T : Comparable<T>> sort(list: List<T>) {  
    // ...  
}
```

콜론 뒤에 지정한 타입이 **upper bound**이다. `Comparable<T>`의 하위타입만 `T`를 대체할 수 있다. 다음 예를 보자.

```
sort(listOf(1, 2, 3)) // OK. Int는 Comparable<Int>의 하위타입이다.  
sort(listOf(HashMap<Int, String>())) // 에러: HashMap<Int, String>은  
Comparable<HashMap<Int, String>>의 하위타입이 아니다.
```

upper bound를 지정하지 않을 경우 기본 upper bound는 `Any?`이다. 화살괄호 안에 오직 한 개의 upper bound만 지정할 수 있다. 동일 타입 파라미터에 대해 한 개 이상의 upper bound가 필요하면 별도의 **where**-절을 사용해야 한다.

```
fun <T> cloneWhenGreater(list: List<T>, threshold: T): List<T>  
    where T : Comparable,  
           T : Cloneable {  
    return list.filter { it > threshold }.map { it.clone() }  
}
```

중첩 클래스

다른 클래스에 클래스를 중첩(nested)할 수 있다.

```
class Outer {  
    private val bar: Int = 1  
    class Nested {  
        fun foo() = 2  
    }  
}  
  
val demo = Outer.Nested().foo() // == 2
```

내부(Inner) 클래스

`inner`로 지정하면 외부(outer) 클래스의 멤버로 클래스에 접근할 수 있다. 내부 클래스는 외부 클래스 객체에 대한 레퍼런스를 갖는다.

```
class Outer {  
    private val bar: Int = 1  
    inner class Inner {  
        fun foo() = bar  
    }  
}  
  
val demo = Outer().Inner().foo() // == 1
```

내부 클래스에서 `this` 사용 시 모호함에 대한 내용은 [한정된 this 식](#)을 참고한다.

Enum 클래스

enum 클래스의 기본 용법은 타입에 안전한 열거형을 구현하는 것이다.

```
enum class Direction {  
    NORTH, SOUTH, WEST, EAST  
}
```

각 열거 상수는 객체이다. 열거 상수는 콤마로 구분한다.

초기화

각 열거 상수는 enum 클래스의 인스턴스이므로 초기화할 수 있다.

```
enum class Color(val rgb: Int) {  
    RED(0xFF0000),  
    GREEN(0x00FF00),  
    BLUE(0x0000FF)  
}
```

임의 클래스

열거 상수는 임의 클래스를 이용해서 개별 선언할 수 있다.

```
enum class ProtocolState {  
    WAITING {  
        override fun signal() = TALKING  
    },  
  
    TALKING {  
        override fun signal() = WAITING  
    };  
  
    abstract fun signal(): ProtocolState  
}
```

베이스 메서드를 오버라이딩하는 것 외에 상응하는 메서드를 가질 수 있다. enum 클래스가 멤버를 정의하면 자바처럼 세미콜론을 이용해서 멤버 정의와 enum 상수 정의를 구분해야 한다.

enum 상수로 작업하기

자바와 마찬가지로 enum 클래스는 정의한 enum 상수 목록을 구하고 이름으로 enum 상수에 접근할 수 있는 메서드를 제공하고 있다. 이 메서드의 시그니처는 다음과 같다(enum 클래스의 이름이 EnumClass 라고 가정):

```
EnumClass.valueOf(value: String): EnumClass  
EnumClass.values(): Array<EnumClass>
```

valueOf() 메서드는 지정한 이름에 해당하는 enum 상수가 없으면 IllegalArgumentException 을 발생한다.

모든 enum 상수는 상수의 이름과 enum 클래스에 정의된 순서를 구할 수 있는 프로퍼티를 갖는다.

```
val name: String  
val ordinal: Int
```

enum 상수는 또한 [Comparable](#) 인터페이스를 구현하고 있다. enum 클래스에 정의된 순서를 자연 정렬 순서로 사용한다.

오브젝트 식과 선언

때때로 상속한 클래스를 만들지 않고 어떤 클래스를 아주 약간 변경한 객체를 만들고 싶을 때가 있다. 자바에서는 이때 *임의 내부(inner) 클래스*를 사용한다. 코틀린은 이 개념을 *오브젝트 식*과 *오브젝트 선언*으로 약간 일반화했다.

오브젝트 식

특정 타입을 상속받은 임의 클래스의 객체를 생성할 때 다음과 같이 코드를 작성한다.

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }

    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
})
```

상위타입이 생성자를 가지면 알맞은 생성자 파라미터를 전달해야 한다. 상위 타입이 여러 개면 콜론 뒤에 콤마로 구분해서 지정한다.

```
open class A(x: Int) {
    public open val y: Int = x
}

interface B {...}

val ab = object : A(1), B {
    override val y = 15
}
```

만약 별도 상위타입이 없는 “단순히 객체”가 필요하다면 다음 코드를 사용할 수 있다.

```
val adHoc = object {
    var x: Int = 0
    var y: Int = 0
}
print(adHoc.x + adHoc.y)
```

자바의 임의 내부(inner) 클래스와 비슷하게 오브젝트 식의 코드는 외부 스코프의 변수에 접근할 수 있다(자바와 달리 final 변수로 제한되지 않는다).

```

fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // ...
}

```

오브젝트 선언

[싱글톤](#)은 매우 유용한 패턴이다. 코틀린은 (스칼라를 따라해서) 쉽게 싱글톤을 선언할 수 있도록 했다.

```

object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ...
}

```

이를 *오브젝트 선언*이라고 한다. **object** 키워드 뒤에 이름이 있으면 이는 `_식_`을 말하는 것이 아니다. 오브젝트는 변수에 할당할 수 없으며 이름으로 참조할 수 있다. 오브젝트는 상위타입을 가질 수 있다.

```

object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) {
        // ...
    }

    override fun mouseEntered(e: MouseEvent) {
        // ...
    }
}

```

주의: 오브젝트는 로컬일 수 없다(예를 들어 함수 안에 바로 중첩하는 것). 하지만 내부(inner)가 아닌 클래스나 다른 오브젝트 선언에 중첩할 수는 있다.

컴페니언 오브젝트

클래스 안의 오브젝트 선언은 **companion** 키워드를 붙일 수 있다.

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

컴페니언 오브젝트의 멤버는 클래스 이름을 한정자로 사용해서 간단히 호출할 수 있다.

```
val instance = MyClass.create()
```

컴페니언 오브젝트의 이름을 생략하면 `Companion` 을 이름으로 사용한다.

```
class MyClass {
    companion object {
    }
}

val x = MyClass.Companion
```

컴페니언 오브젝트의 멤버가 다른 언어의 정적 멤버처럼 보이긴 하지만 런타임에는 실제 객체의 인스턴스 멤버이다. 예를 들어 다음과 같이 인터페이스를 구현할 수 있다.

```
interface Factory<T> {
    fun create(): T
}

class MyClass {
    companion object : Factory<MyClass> {
        override fun create(): MyClass = MyClass()
    }
}
```

하지만 JVM에서 `@JvmStatic` 애노테이션을 사용하면 실제 정적 메서드와 필드로 생성된 컴페니언 오브젝트의 멤버를 가질 수 있다. 이에 대한 내용은 [자바 상호운용](#)을 참고한다.

오브젝트 식과 오브젝트 선언의 세만틱 차이

오브젝트 식과 오브젝트 선언은 한 가지 중요한 의미 차이가 있다.

- 오브젝트 선언은 최초에 접근할 때까지 초기화를 (**lazily**) 미룬다.
- 오브젝트 식은 사용할 때 **즉시** 실행(초기화)된다.

위임

클래스 위임

[위임 패턴](#)은 상속의 좋은 대안임이 증명되었다. 코틀린은 장식 코드(boilerplate code) 없이 언어 자체에서 위임 패턴을 지원한다. 아래에서 `Derived` 클래스는 `Base` 인터페이스에서 상속받은 모든 public 메서드를 지정한 객체로 위임할 수 있다.

```
interface Base {  
    fun print()  
}  
  
class BaseImpl(val x: Int) : Base {  
    override fun print() { print(x) }  
}  
  
class Derived(b: Base) : Base by b  
  
fun main() {  
    val b = BaseImpl(10)  
    Derived(b).print() // 10 출력  
}
```

`Derived`의 상위타입 목록의 `by`-절은 `Derived` 객체 내부에 `b`를 저장하고 컴파일러가 `Base`의 모든 메서드에 대해 `b`로 위임하는 메서드를 `Derived`에 생성한다는 것을 나타낸다.

위임 프로퍼티

필요할 때 수동으로 기능을 구현할 수 있지만, 한번에 다 구현하고 라이브러리에 넣는 것이 매우 좋은 그런 종류의 프로퍼티가 존재한다. 다음 프로퍼티가 이런 종류에 속한다.

- lazy 프로퍼티: 최초로 접근할 때 값을 계산
- observable 프로퍼티: 이 프로퍼티가 바뀔 때 리스너에 통지
- 별도 필드가 아닌 맵에 저장한 프로퍼티

이를 포함한 여러 경우를 처리하기 위해 코틀린은 _위임 프로퍼티(Delegated Properties)_를 지원한다.

```
class Example {  
    var p: String by Delegate()  
}
```

위임 프로퍼티 구문은 `val/var <property name>: <Type> by <expression>` 이다. `by` 뒤에 식이 _대리 객체 (delegate)_이며, 프로퍼티에 대한 `get()` 과 `set()` 을 대리 객체의 `getValue()` 와 `setValue()` 메서드로 위임한다. 프로퍼티 대리 객체는 인터페이스를 아무렇게나 구현하면 안 되고, `getValue()` 함수를 (그리고 `var`의 경우 `setValue()` 함수를) 제공해야 한다. 다음 예를 보자:

```
class Delegate {  
    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {  
        return "$thisRef, thank you for delegating '${property.name}' to me!"  
    }  
  
    operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {  
        println("$value has been assigned to '${property.name}' in $thisRef.")  
    }  
}
```

`Delegate` 인스턴스에 위임하는 프로퍼티 `p` 에서 값을 읽으면 `Delegate` 의 `getValue()` 함수를 실행한다. 이 메서드의 첫 번째 파라미터는 `p` 를 포함한 객체이고 두 번째 파라미터는 `p` 자체에 대한 설명을 포함한다(예를 들어, 프로퍼티의 이름을 구할 수 있다). 다음은 예이다.

```
val e = Example()  
println(e.p)
```

이 코드는 다음을 출력한다.

Example@33a17727, thank you for delegating 'p' to me!

비슷하게 `p` 에 할당하면 `setValue()` 함수를 호출한다. 처음 두 파라미터는 동일하고 세 번째 파라미터는 할당할 값을 갖는다.

```
e.p = "NEW"
```

이 코드는 다음을 출력한다.

NEW has been assigned to 'p' in Example@33a17727.

프로퍼티 위임 객체 조건

위임 객체의 조건을 아래 요약했다.

읽기 전용 프로퍼티(예, `val`)에 대해 위임 객체는 다음 파라미터를 갖는 이름이 `getValue` 인 함수를 제공해야 한다.

- 리시버 — `_프로퍼티 소유자_`와 같거나 상위타입(확장 프로퍼티의 경우 — 확장한 타입)
- 메타데이터 — `KProperty<*>` 타입이거나 그 상위타입

이 함수는 프로퍼티와 같은 타입을 (또는 그 하위타입을) 리턴해야 한다.

수정 가능 프로퍼티(`var`)에 대해 위임 객체는 이름이 `setValue` 이고 *추가로* 다음 파라미터를 갖는 함수를 제공해야 한다.

- receiver — `getValue()` 와 동일
- metadata — `getValue()` 와 동일
- new value — 프로퍼티와 같거나 상위타입이어야 함

`getValue()` 와 `setValue()` 함수는 위임 클래스의 멤버 함수나 확장 함수로 제공할 수 있다. 확장 함수의 경우 원래 이 함수를 제공하지 않는 객체에 프로퍼티를 위임해야 할 때 유용하다. 두 함수 모두 `operator` 키워드로 지정해야 한다.

표준 위임

코틀린 표준 라이브러리는 몇 가지 유용한 종류의 위임 객체를 위한 팩토리 메서드를 제공한다.

Lazy

`lazy()` 는 람다를 파라미터로 받고 `Lazy<T>` 인스턴스를 리턴하는 함수이다. `Lazy`는 `lazy` 프로퍼티를 구현하기 위한 위임 객체이다. 처음 `get()` 을 호출하면 `lazy()` 에 전달한 람다를 실행하고 그 결과를 저장하며 이후 `get()` 호출에 대해선 저장한 결과를 리턴한다.

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}

fun main(args: Array<String>) {
    println(lazyValue)
    println(lazyValue)
}
```

`lazy` 프로퍼티 연산은 기본적으로 **동기화**된다. 한 스레드만 값을 계산할 수 있으며 모든 스레드는 같은 값을 보게 된다. 만약 초기화 과정을 동기화할 필요가 없다면 `lazy()` 함수의 파라미터에 `LazyThreadSafetyMode.PUBLICATION` 을 전달해서 여러 스레드가 동시에 실행할 수 있도록 할 수 있다. 그리고 항상 한 스레드에서만 초기화하는 것을 보장하려면 `LazyThreadSafetyMode.NONE` 모드를 사용하면 된다. 이 모드는 스레드 안정성을 보장하기 위한 오버헤드를 발생하지 않는다.

Observable

`Delegates.observable()` 은 초깃값과 수정에 대한 핸들러를 인자로 갖는다. 프로퍼티에 값을 할당할 때마다 (할당 완료 후에) 핸들러를 호출한다. 핸들러는 할당 대상 프로퍼티, 이전 값, 새로운 값을 파라미터로 갖는다.

```
import kotlin.properties.Delegates

class User {
    var name: String by Delegates.observable("<no name>") {
        prop, old, new ->
        println("$old -> $new")
    }
}

fun main(args: Array<String>) {
    val user = User()
    user.name = "first"
    user.name = "second"
}
```

이 예는 다음을 출력한다.

```
<no name> -> first
first -> second
```

만약 할당 과정 중간에 끼어들어 할당을 “거부(veto)”하고 싶다면 `observable()` 대신에 `vetoable()` 을 사용하면 된다. 프로퍼티에 새 값을 할당하기 전에 `vetoable` 에 전달한 핸들러를 호출한다.

맵에 프로퍼티 저장하기

프로퍼티의 값을 맵에 저장하는 것은 일반적이다. 이는 JSON 파싱이나 다른 “동적” 작업을 할 때 흔한 일이다. 이 경우 위임 프로퍼티로 위임 객체 대신 맵 인스턴스 자체를 사용할 수 있다.

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int by map
}
```

이 예제에서 생성자는 맵을 받는다.

```
val user = User(mapOf(
    "name" to "John Doe",
    "age" to 25
))
```

위임 프로퍼티는 이 맵에서 값을 읽어온다(문자열 키 — 프로퍼티의 이름):

```
println(user.name) // "John Doe" 출력
println(user.age)  // 25 출력
```

읽기 전용 `Map` 대신에 `MutableMap` 을 사용하면 `var` 프로퍼티에도 동작한다.

```
class MutableUser(val map: MutableMap<String, Any?>) {  
    var name: String by map  
    var age: Int by map  
}
```

함수와 람다

함수

함수 선언

`fun`을 사용해서 함수를 선언한다.

```
fun double(x: Int): Int {  
    }  
}
```

함수 사용

전통적인 방식으로 함수를 호출한다.

```
val result = double(2)
```

멤버 함수를 호출할 때에는 점 부호를 사용한다.

```
Sample().foo() // Sample 클래스의 인스턴스를 생성하고 foo 호출
```

중위 표현

다음의 경우 중위 표현(Infix notation)을 사용해서 함수를 호출할 수도 있다.

- 멤버 함수이거나 [확장 함수](#)일 때
- 파라미터를 한 개 가질 때
- `infix` 키워드로 지정했을 때

```
// Int에 대한 확장 함수
infix fun Int.shl(x: Int): Int {
    ...
}

// 중위 표현을 사용해서 확장 함수 호출

1 shl 2

// 다음 코드와 같음

1.shl(2)
```

파라미터

함수 파라미터는 *name: type*과 같은 파스칼 표기법을 사용해서 정의한다. 파라미터는 콤마로 구분한다. 각 파라미터는 반드시 타입을 지정해야 한다.

```
fun powerOf(number: Int, exponent: Int) {
    ...
}
```

기본 인자

함수 파라미터는 기본 값을 가질 수 있다. 해당 인자를 생략하면 이 기본 값을 사용한다. 이는 다른 언어 대비 오버로딩을 줄일 수 있도록 해 준다.

```
fun read(b: Array<Byte>, off: Int = 0, len: Int = b.size()) {
    ...
}
```

기본 값은 타입 뒤에 `=`와 값을 사용해서 지정한다.

이름(Named) 인자

함수 파라미터는 함수를 호출할 때 이름을 가질 수 있다. 파라미터 개수가 많거나 기본 값을 가진 경우 이름을 사용하면 매우 편리하다.

다음 함수를 보자.

```
fun reformat(str: String,
    normalizeCase: Boolean = true,
    upperCaseFirstLetter: Boolean = true,
    divideByCamelHumps: Boolean = false,
    wordSeparator: Char = ' ') {
    ...
}
```

기본 인자를 사용하면 이 함수를 다음과 같이 호출할 수 있다.

```
reformat(str)
```

하지만 기본 값을 사용하지 않고 호출하면 다음과 같은 코드가 된다.

```
reformat(str, true, true, false, '_')
```

이름 인자를 사용하면 코드 가독성을 높일 수 있다.

```
reformat(str,
  normalizeCase = true,
  upperCaseFirstLetter = true,
  divideByCamelHumps = false,
  wordSeparator = '_'
)
```

모든 인자가 필요하지 않으면 더 간결해진다.

```
reformat(str, wordSeparator = '_')
```

이름 인자 구문은 자바 함수를 호출할 때는 사용할 수 없다는 점에 유의하자. 왜냐하면 자바 바이트코드는 함수 파라미터의 이름을 유지하지 않기 때문이다.

Unit 리턴 함수

어떤 값도 리턴하지 않는 함수의 리턴 타입은 `Unit` 이다. `Unit` 타입의 값은 `Unit` 한 개만 존재한다. 이 값을 명시적으로 리턴할 필요는 없다.

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
    else
        println("Hi there!")
    // `return Unit` 또는 `return` 생략 가능
}
```

`Unit` 리턴 타입 선언도 생략할 수 있다. 위 코드는 다음과 동일하다.

```
fun printHello(name: String?) {
    ...
}
```

단일 식 함수

함수가 단일 식을 리턴하면 중괄호를 생략할 수 있고 `=` 부호 뒤에 몸체를 지정할 수 있다.

```
fun double(x: Int): Int = x * 2
```

이 경우 컴파일러가 리턴 타입을 유추할 수 있으므로 리턴 타입 지정을 생략할 수 있다.

```
fun double(x: Int) = x * 2
```

리턴 타입 지정

블록 몸체를 갖는 함수는 `Unit` 을 리턴하는 것이 아니라면 반드시 리턴 타입을 지정해야 한다. (생략할 수 있는 경우 참고) 블록 몸체를 가진 함수는 몸체에 제어 흐름이 복잡할 수 있고 코드를 읽는 사람 입장에서 (또는 컴파일러 입장에서도) 리턴 타입이 명확하지 않을 수 있기 때문에 리턴 타입을 유추하지 않는다.

가변 인자 (Varargs)

함수 파라미터를 (보통 마지막 파라미터를) `vararg` 제한자로 지정할 수 있다.

```
fun <T> asList(vararg ts: T): List<T> {  
    val result = ArrayList<T>()  
    for (t in ts) // ts는 Array  
        result.add(t)  
    return result  
}
```

가변 인자를 사용하면 함수에 인자 개수를 가변적으로 전달할 수 있다.

```
val list = asList(1, 2, 3)
```

`T` 타입의 `vararg` 파라미터는 함수 안에서 `T` 배열로 접근할 수 있다. 예의 경우 `ts` 변수는 `Array<out T>` 타입을 갖는다.

오직 한 개 파라미터만 `vararg` 로 지정할 수 있다. 만약 `vararg` 파라미터가 마지막 파라미터가 아니면 그 뒤 값은 이름 인자 구문을 이용해서 전달한다. 파라미터가 함수 타입이면 괄호 밖에 람다를 전달하는 방식을 사용할 수 있다.

`vararg` 함수를 호출할 때 `asList(1, 2, 3)` 처럼 인자를 한 개씩 전달할 수 있다. 만약 배열을 함수에 가변 인자로 전달하고 싶다면 **펼침 연산자**(배열 앞에 붙이는 `*`)를 사용하면 된다.

```
val a = arrayOf(1, 2, 3)  
val list = asList(-1, 0, *a, 4)
```

함수 범위

코틀린은 파일에서 함수를 최상위 레벨로 선언할 수 있다. 자바, C#이나 스칼라와 같은 언어처럼 함수를 갖는 클래스를 만들 필요가 없다. 최상위 레벨 함수뿐만 아니라 로컬 함수, 멤버 함수, 확장 함수로도 선언할 수 있다.

로컬 함수

코틀린은 로컬 함수를 지원한다. 다음과 같이 다른 함수 안에 함수를 선언할 수 있다.


```

fun dfs(graph: Graph) {
    fun dfs(current: Vertex, visited: Set<Vertex>) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v, visited)
    }

    dfs(graph.vertices[0], HashSet())
}

```

로컬 함수는 다른 함수의 로컬 변수에 접근 가능하므로(클로저) 위 코드에서 *visited*를 로컬 변수로 바꿀 수 있다.

```

fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}

```

멤버 함수

멤버 함수는 클래스나 오브젝트 안에 정의한 함수이다.

```

class Sample() {
    fun foo() { print("Foo") }
}

```

멤버 함수를 호출할 때는 점 부호를 사용한다.

```

Sample().foo() // Sample 클래스의 인스턴스를 생성하고 foo 호출

```

클래스와 멤버 오버라이딩에 대한 내용은 [클래스](#)와 [상속](#)을 참고한다.

지네릭 함수

함수는 함수 이름 앞에 화살괄호를 사용해서 지네릭 파라미터를 지정할 수 있다.

```

fun <T> singletonList(item: T): List<T> {
    // ...
}

```

지네릭 함수에 대한 내용은 [지네릭](#)을 참고한다.

인라인 함수

인라인 함수는 [여기](#)에서 설명한다.

확장 함수

확장 함수는 [여기](#)에서 설명한다.

고차 함수와 람다

고차 함수와 람다는 [여기](#)에서 설명한다.

꼬리 재귀 함수

코틀린은 [꼬리 재귀](#)로 알려진 함수형 프로그래밍 스타일을 지원한다. 꼬리 재귀는 재귀 함수를 스택오버플로우 걱정이 없는 루프로 바꾸는 알고리즘을 허용한다. 함수에 `tailrec` 제한자를 붙이고 컴파일러가 재귀를 최적화할 수 있는 요건을 충족하면 빠르고 효율적인 루프 기반 버전으로 바꾼다.

```
tailrec fun findFixPoint(x: Double = 1.0): Double
    = if (x == Math.cos(x)) x else findFixPoint(Math.cos(x))
```

이 코드는 수학의 상수값인 코사인의 고정소수점을 계산한다. 이 코드는 1.0에서 시작해서 더 이상 값이 바뀌지 않을 때까지 `Math.cos`를 반복해서 호출한다. 결과는 0.7390851332151607이다. 이 코드는 전통적인 방식의 다음 코드와 같다.

```
private fun findFixPoint(): Double {
    var x = 1.0
    while (true) {
        val y = Math.cos(x)
        if (x == y) return y
        x = y
    }
}
```

`tailrec` 제한자가 가능하려면 함수는 반드시 마지막에 자신을 호출해야 한다. 재귀 호출 뒤에 다른 코드가 있다면 꼬리 재귀를 사용할 수 없다. `try/catch/finally` 블록에서는 사용할 수 없다. 현재 재귀 호출은 JVM 기반에서만 지원한다.

고차 함수와 람다

고차 함수

고차 함수는 파라미터로 함수를 받거나 함수를 리턴하는 함수이다. 좋은 예가 `lock()` 이다. 이 함수는 락 오브젝트와 함수를 받아 락을 구하고 함수를 실행하고 락을 해제한다.

```
fun <T> lock(lock: Lock, body: () -> T): T {
    lock.lock()
    try {
        return body()
    }
    finally {
        lock.unlock()
    }
}
```

위 코드를 살펴보자. `body` 는 [함수 타입](#)인 `() -> T` 를 갖는다. `body`에는 파라미터가 없고 `T` 타입 값을 리턴하는 함수를 전달해야 한다. `lock` 으로 보호하는 동안 `try` 블록에서 `body` 함수를 실행하고 그 결과를 `lock()` 함수의 결과로 리턴한다.

`lock()` 함수를 호출하려면 인자로 다른 함수를 전달하면 된다([함수 레퍼런스](#) 참고).

```
fun toBeSynchronized() = sharedResource.operation()

val result = lock(lock, ::toBeSynchronized)
```

다른 간편한 방법은 [람다 식](#)을 전달하는 것이다.

```
val result = lock(lock, { sharedResource.operation() })
```

람다 식은 [뒤에서](#) 설명하지만, 이 절을 이해하는데 필요한 람다 식에 대한 내용을 아래 요약했다.

- 람다 식은 항상 중괄호로 둘러 쓴다.
- `->` 전에 파라미터를(존재하면) 선언하고(파라미터 타입은 생략 가능),
- `->` 뒤에 몸체가 온다(존재하면).

코틀린에서 함수의 마지막 파라미터로 함수를 전달하면 괄호 밖에 파라미터를 지정할 수 있다.

```
lock (lock) {
    sharedResource.operation()
}
```

고차 함수의 다른 예는 `map()` 이다.

```
fun <T, R> List<T>.map(transform: (T) -> R): List<R> {
    val result = arrayListOf<R>()
    for (item in this)
        result.add(transform(item))
    return result
}
```

이 함수는 다음과 같이 호출할 수 있다.

```
val doubled = ints.map { it -> it * 2 }
```

함수를 호출할 때 람다가 유일한 인자일 경우 함수 호출시 사용하는 괄호를 완전히 생략할 수 있다.

다른 유용한 규칙으로, 함수 리터럴의 파라미터가 한 개면 파라미터 선언을 (-> 포함해서) 생략할 수 있고 파라미터 이름은 `it` 이 된다.

```
ints.map { it * 2 }
```

이 규칙은 [LINQ-스타일](#)로 코드를 작성할 수 있도록 해 준다.

```
strings.filter { it.length == 5 }.sortBy { it }.map { it.toUpperCase() }
```

인라인 함수

때로는 [인라인 함수](#)를 사용하면 고차 함수의 성능을 향상할 수 있는 이점이 있다.

람다 식과 임의 함수

람다 식이나 임의 함수는 “함수 리터럴”로 함수 선언 없이 식으로 바로 전달할 수 있다. 다음 예를 보자:

```
max(strings, { a, b -> a.length() < b.length() })
```

`max` 함수는 두 번째 인자로 함수 값을 받는 고차 함수이다. 두 번째 인자는 그 자체가 함수인 함수 리터럴 식이다. 함수로서 이 식은 다음과 동일하다.

```
fun compare(a: String, b: String): Boolean = a.length() < b.length()
```

함수 타입

함수가 다른 함수를 파라미터로 받으려면 그 파라미터를 함수로 지정해야 한다. 예를 들어 위에서 보여준 `max` 함수는 다음과 같이 정의한다.

```
fun <T> max(collection: Collection<T>, less: (T, T) -> Boolean): T? {
    var max: T? = null
    for (it in collection)
        if (max == null || less(max, it))
            max = it
    return max
}
```

`less` 파라미터 타입은 `(T, T) -> Boolean` 함수 타입이다. 이 함수는 두 개의 `T` 타입 파라미터를 갖고 `Boolean` 타입을 리턴하며 첫 번째 파라미터가 두 번째 파라미터보다 작으면 `true`를 리턴한다.

코드에서 네 번째 줄을 보면 `less` 를 함수로 사용한다. 함수를 호출할 때 두 개의 `T` 타입 인자를 전달하고 있다.

함수 타입은 위와 같이 작성하거나, 각 파라미터에 의미를 문서화하고 싶다면 네임드 파라미터를 사용해서 작성한다.

```
val compare: (x: T, y: T) -> Int = ...
```

람다 식 구문

람다 식(함수 타입 리터럴)의 완전한 구문은 다음과 같다.

```
val sum = { x: Int, y: Int -> x + y }
```

람다 식은 항상 괄호로 둘러 싼다. 완전한 구문 형식에서는 괄호 안에 파라미터 선언이 위치하며 타입 지정을 생략할 수 있다. `->` 부호 다음에 몸체가 위치한다. 생략 가능한 것을 모두 생략하면 다음과 같이 작성할 수 있다.

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
```

람다 식은 파라미터를 한 개만 갖는 경우가 빈번하다. 코틀린이 그 시그니처를 알아낼 수 있으면 파라미터 선언을 생략할 수 있으며, 코틀린이 자동으로 `it` 이름의 파라미터를 선언한다.

```
ints.filter { it > 0 } // 이 리터럴의 타입은 '(it: Int) -> Boolean'이다.
```

함수의 마지막 파라미터가 함수면 인자 목록을 갖는 괄호 밖에 람다 식을 전달할 수 있다. [callSuffix](#) 문법을 참고한다.

임의 함수

위 람다 식 구문에서 빠진 게 하나 있는데 그것은 바로 리턴 타입을 지정하는 방법이다. 많은 경우 리턴 타입을 자동으로 유추하기 때문에 지정하지 않아도 된다. 하지만 명시적으로 지정하고 싶다면 *임의 함수* 구문을 사용할 수 있다.

```
fun(x: Int, y: Int): Int = x + y
```

임의 함수는 이름이 없다는 것을 제외하면 일반 함수 선언과 비슷하다. 몸체는 (위 코드처럼) 식이거나 블록일 수 있다.

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

일반 함수와 동일한 방법으로 파라미터와 리턴 타입을 지정한다. 문맥에서 파라미터 타입을 유추할 수 있으면 타입을 생략할 수 있다.

```
ints.filter(fun(item) = item > 0)
```

임의 함수의 리턴 타입은 일반 함수와 동일하게 유추한다. 식 몸체를 가진 임의 함수는 리턴 타입을 자동으로 유추한다. 블록 몸체를 가진 임의 함수는 명시적으로 리턴 타입을 지정해야 한다(아니면 `Unit` 으로 가정한다).

임의 함수 파라미터는 항상 괄호 안에 전달해야 한다. 괄호 밖에 함수를 위치시킬 수 있는 약식 구문은 람다 식만 가능하다.

람다 식과 임의 함수의 또 다른 차이점은 [비-로컬 리턴](#)의 동작에 있다. 라벨 없는 `return` 문은 항상 `fun` 키워드로 선언한 함수에서 리턴한다. 이는 람다 식 안에서 `return`을 사용하면 둘러 쓴 함수에서 리턴하는 반면에 임의 함수 안에서 `return`을 사용하면 임의 함수 자체에서 리턴하는 것을 의미한다.

클로저

람다 식이나 임의 함수 (또는 [로컬 함수](#)) 그리고 [오브젝트 식](#)은 그것의 `_클로저`(즉 외부 범위에 선언한 변수)_에 접근할 수 있다. 자바와 달리 클로저로 캡처한 변수를 수정할 수 있다.

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
}
print(sum)
```

리시버를 갖는 함수 리터럴

코틀린은 함수 리터럴을 실행할 때 `_리시버 객체_`를 지정할 수 있다. 함수 리터럴 몸체 안에서 추가 한정자 없이 리시버 객체의 메서드를 호출할 수 있다. 함수 몸체 안에서 리시버 객체의 멤버에 접근할 수 있는 것은 확장 함수와 유사하다. 이를 사용하는 가장 중요한 한 가지 예가 [타입-안전 그루브-스타일 빌더](#)이다.

이런 함수 리터럴 타입은 리시버를 갖는 함수 타입이다.

```
sum : Int.(other: Int) -> Int
```

이제 리시버 객체의 함수처럼 함수 리터럴을 호출할 수 있다.

```
1.sum(2)
```

임의 함수 구문을 사용하면 함수 리터럴에 직접 리시버 타입을 지정할 수 있다. 이는 리시버를 갖는 함수 타입 변수를 선언하고 이를 나중에 사용해야 할 때 유용하다.

```
val sum = fun Int.(other: Int): Int = this + other
```

문맥에서 리시버 타입을 유추할 수 있다면 람다 식을 리시버를 가진 함수 리터럴로 사용할 수 있다.

```
class HTML {  
    fun body() { ... }  
}  
  
fun html(init: HTML.() -> Unit): HTML {  
    val html = HTML() // 리시버 객체 생성  
    html.init()        // 람다에 리시버 객체를 전달  
    return html  
}  
  
html {                // 리시버를 가진 람다 시작  
    body()            // 리시버 객체의 메서드 호출  
}
```

인라인 함수

[고차 함수](#)를 사용하면 런타임에 어느 정도 불이익이 발생한다. 각 함수는 객체이고 함수 몸체에서 접근하는 변수를 캡처한다. (함수 객체와 클래스를 위한)메모리 할당과 버추얼 호출로 런타임에 부하가 발생한다.

하지만 많은 경우 람다식을 인라인해서 이런 부하를 제거할 수 있다. 위의 `lock()` 함수가 이런 상황의 좋은 예이다. `lock()` 함수는 호출 위치에 쉽게 인라인할 수 있다. 다음을 보자.

```
lock(1) { foo() }
```

파라미터를 위한 함수 객체를 생성하고 호출을 생성하는 대신 컴파일러는 다음 코드를 만들어낼 수 있다.

```
1.lock()
try {
    foo()
}
finally {
    1.unlock()
}
```

이게 우리가 애초에 원한 것이다. 그렇지 않나?

컴파일러가 이렇게 할 수 있으려면 `lock()` 함수에 `inline` 제한자를 붙이면 된다.

```
inline fun lock<T>(lock: Lock, body: () -> T): T {
    // ...
}
```

`inline` 제한자는 함수 자체와 함수에 전달하는 람다에 영향을 준다. 이것 모두 호출 위치에 인라인 된다.

인라인을 하면 생성된 코드가 증가할 수 있다. 하지만 합리적으로 잘 활용하면(큰 함수는 인라인하지 않는 식으로) 성능에서 (특히 루프의 “megamorphic” 호출 위치는 더 많은) 보상을 받게 된다.

noinline

인라인 함수에 전달된 람다 중 일부만 인라인 되길 원하면 `noinline` 제한자로 함수 파라미터를 지정하면 된다.

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) {
    // ...
}
```

인라인 가능 람다는 인라인 함수 안에서만 호출할 수 있고 인라인 가능 인자로만 전달할 수 있다. 하지만, `noinline` 에 전달한 람다는 필드에 저장하거나 인자로 전달하는 등 원하는 방식으로 처리할 수 있다.

인라인 함수에 인라인 가능한 함수 파라미터가 없고 [reified 타입 파라미터](#)가 없으면, 컴파일러는 그 인라인 함수가 이점이 없다는 경고를 발생한다(인라인이 필요하다고 확신하면 이 경고를 무시해도 된다).

비-로컬 리턴

코틀린에서는 이름을 가진 함수나 임의 함수에서 나가려면 한정하지 않은 일반 `return` 만 사용할 수 있다. 이는 람다에서 나가려면 [라벨](#)을 사용해야 한다는 것을 의미한다. 람다 안에서는 단순 `return` 은 허용하지 않는데 그 이유는 람다는 둘러싼 함수를 리턴할 수 없기 때문이다.

```
fun foo() {
    ordinaryFunction {
        return // 에러: `foo`를 여기서 리턴할 수 없다
    }
}
```

만약 람다를 전달한 함수가 인라인되면 리턴도 같이 인라인된다. 그래서 다음을 허용한다.

```
fun foo() {
    inlineFunction {
        return // OK: 람다를 인라인한다
    }
}
```

이 리턴(람다에 위치하지만 둘러싼 함수를 나가는)을 *비-로컬* 리턴이라 부른다. 이런 종류의 리턴을 인라인 함수가 둘러싼 루프에서 사용하곤 한다.

```
fun hasZeros(ints: List<Int>): Boolean {
    ints.forEach {
        if (it == 0) return true // hasZeros에서 리턴
    }
    return false
}
```

일부 인라인 함수는 파라미터로 전달받은 람다를 호출할 때 함수 몸체에서 직접 호출하지 않고 다른 실행 컨텍스트를 통해(예, 로컬 객체나 중첩 함수) 호출해야 할 때가 있다. 이 경우 람다 안에서 비-로컬 흐름을 제어할 수 없다. 이를 지정하려면 람다 파라미터에 `crossinline` 제한자를 붙이면 된다.

```
inline fun f(crossinline body: () -> Unit) {
    val f = object: Runnable {
        override fun run() = body()
    }
    // ...
}
```

인라인된 람다에서 `break`와 `continue`는 아직 사용할 수 없는데, 앞으로 지원할 계획이다.

Reified 타입 파라미터

때때로 파라미터로 전달한 타입에 접근해야 할 때가 있다.

```
fun <T> TreeNode.findParentOfType(clazz: Class<T>): T? {
    var p = parent
    while (p != null && !clazz.isInstance(p)) {
        p = p?.parent
    }
    @Suppress("UNCHECKED_CAST")
    return p as T
}
```

이 코드는 노드가 특정 타입을 가졌는지 확인하기 위해 트리를 탐색하고 리플렉션을 사용한다. 모두 좋은데 호출하는 코드가 이쁘지 (pretty ^^) 않다.

```
myTree.findParentOfType(MyTreeNodeType::class.java)
```

실제로는 이 함수에 단순히 타입을 전달해서 다음과 같이 호출하길 원하는 것이다.

```
myTree.findParentOfType<MyTreeNodeType>()
```

이렇게 할 수 있도록 인라인 함수는 *reified 타입 파라미터*를 지원한다. 이를 사용한 코드는 다음과 같다.

```
inline fun <reified T> TreeNode.findParentOfType(): T? {
    var p = parent
    while (p != null && p !is T) {
        p = p?.parent
    }
    return p as T
}
```

타입 파라미터에 `reified` 제한자를 적용하면 마치 클래스처럼 타입 파라미터에 접근할 수 있다. 인라인 함수이므로 리플렉션이 필요 없고 `!is` 나 `as` 와 같은 일반 연산자가 동작한다. 또한 앞서 언급한 `myTree.findParentOfType<MyTreeNodeType>()` 처럼 호출할 수 있다:

`reified` 타입 파라미터에 리플렉션을 사용할 수 있다. 많은 경우 리플렉션이 필요 없긴 하지만 말이다.

```
inline fun <reified T> membersOf() = T::class.members

fun main(s: Array<String>) {
    println(membersOf<StringBuilder>().joinToString("\n"))
}
```

(인라인이 아닌) 일반 함수는 `reified` 파라미터를 가질 수 없다. 런타임 표현을 갖지 않는 타입(`reified` 타입 파라미터가 아니거나 `Nothing` 과 같은 가공 타입)은 `reified` 타입 파라미터를 위한 인자로 사용할 수 없다.

저수준 설명은 [스펙 문서](#) 참고한다.

기타

분리 선언

다음 코드처럼 객체의 값을 여러 변수로 분리하면 편리할 때가 있다.

```
val (name, age) = person
```

이 구문을 `_분리 선언(destructuring declaration)_`이라고 부른다. 분리 선언은 한 번에 여러 변수를 생성한다. 위 코드는 `name` 과 `age` 두 변수를 선언하고 각 변수를 독립적으로 사용할 수 있다.

```
println(name)
println(age)
```

분리 선언은 다음 코드로 컴파일된다.

```
val name = person.component1()
val age = person.component2()
```

`component1()` 과 `component2()` 함수는 코틀린에서 폭넓게 사용하는 `_원칙(principle of conventions)_`을 적용한 다른 예이다(`+` 와 `*` 같은 연산자, `for`-루프 등을 참고). 분리 선언의 우측에 위치한 것은 필요한 개수의 `component` 함수를 갖고 있으면 된다. 물론 `component3()` 과 `component4()` 등 두 개 이상도 가능하다.

분리 선언에서 사용하려면 `componentN()` 함수에 `operator` 키워드를 적용해야 한다.

`for`-루프에도 분리 선언이 가능하다.

```
for ((a, b) in collection) { ... }
```

변수 `a` 와 `b` 는 컬렉션의 `component1()` 와 `component2()` 를 호출한 결과를 값으로 갖는다.

예제: 함수에서 두 값 리턴하기

함수에서 두 값을 리턴해야 한다고 가정해보자. 예를 들어 결과 객체와 어떤 종류의 상태 값을 리턴해야 한다. 이를 하는 간단한 방법은 [데이터 클래스](#)를 만들고 그 인스턴스를 리턴하는 것이다.

```
data class Result(val result: Int, val status: Status)
fun function(...): Result {
    // 계산

    return Result(result, status)
}

// 이제, 이 함수 사용하기:
val (result, status) = function(...)
```

데이터 클래스는 `componentN()` 함수를 자동으로 선언하므로 분리 선언이 가능하다.

주의: 위 코드에서 표준 클래스 `Pair` 와 `Pair<Int, Status>` 를 리턴하는 `function()` 을 사용할 수도 있다. 하지만 데이터에 알맞은 이름을 붙이는게 보통 더 좋다.

예제: 분리 선언과 맵

다음은 아마도 맵을 탐색하는 가장 좋은 방법일 것이다.

```
for ((key, value) in map) {
    // 키와 값으로 무언가를 한다
}
```

이게 되려면 다음을 충족해야 한다.

- `iterator()` 함수를 제공해서 맵을 값 시퀀스로 제공해야 한다.
- 각 요소를 `component1()` 과 `component2()` 함수를 제공하는 페어로 제공해야 한다.

사실 표준 라이브러리가 이 확장을 제공한다.

```
operator fun <K, V> Map<K, V>.iterator(): Iterator<Map.Entry<K, V>> =
    entrySet().iterator()
operator fun <K, V> Map.Entry<K, V>.component1() = getKey()
operator fun <K, V> Map.Entry<K, V>.component2() = getValue()
```

따라서 맵을(또한 데이터 클래스의 인스턴스 컬렉션을) 사용하는 `for`-루프에서 자유롭게 분리 선언을 사용할 수 있다.

컬렉션

많은 다른 언어와 달리 코틀린은 (리스트, 집합, 맵 등 컬렉션에 대해) 변경가능 컬렉션과 변경불가 컬렉션을 구분한다. 언제 컬렉션을 수정할 수 있는지 정확하게 제어하는 것은 버그를 제거하고 좋은 API를 설계하는데 도움이 된다.

처음부터 변경 가능 컬렉션의 읽기 전용 `_뷰_`와 실제 변경 불가인 컬렉션의 차이점을 이해하는 것이 중요하다. 둘 다 만들기는 쉽지만, 타입 시스템은 둘의 차이를 표현하지 않으므로 (차이가 중요하다면) 직접 컬렉션이 둘 중 무엇인지 추적해야 한다.

코틀린의 `List<out T>` 타입은 `size` 나 `get` 과 같은 읽기 전용 연산을 제공하는 인터페이스이다. 자바와 비슷하게 `Iterable<T>` 의 하위타입인 `Collection<T>` 를 상속한다. 리스트를 변경할 수 있는 메서드는 `MutableList<T>` 리스트에 정의되어 있다. 집합과 맵도 동일한 방식으로 `Set<out T>/MutableSet<T>` 와 `Map<K, out V>/MutableMap<K, V>` 로 구분되어 있다.

리스트와 집합의 기본적인 사용법은 아래와 같다.

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)           // "[1, 2, 3]" 출력
numbers.add(4)
println(readOnlyView)      // "[1, 2, 3, 4]" 출력
readOnlyView.clear()       // -> 컴파일되지 않음

val strings = hashSetOf("a", "b", "c", "c")
assert(strings.size == 3)
```

코틀린은 리스트나 집합을 만들기 위한 구문 요소가 없다. `listOf()`, `mutableListOf()`, `setOf()`, `mutableSetOf()` 와 같은 표준 라이브러리의 메서드를 사용해서 생성한다.

성능이 중요하지 않은 코드는 맵을 생성할 때 `mapOf(a to b, c to d)` [이디엄](#)을 사용할 수 있다.

위 코드에서 `readOnlyView` 변수는 `numbers`와 같은 리스트를 참조하고 있는데, 참조한 리스트가 바뀌면 함께 바뀐다는 점에 유의하자. 리스트에 대한 유일한 참조가 읽기 전용 변수라면 완전한 불변 컬렉션을 사용할 것을 고려해보자. 불변 컬렉션을 만드는 간단한 방법은 다음과 같다.

```
val items = listOf(1, 2, 3)
```

현재 `listOf` 메서드는 배열 리스트를 이용해서 구현했다. 이 메서드는 불변 리스트를 생성하므로 이 점을 활용해서 향후에 메모리를 더 효율적으로 사용하는 완전한 불변 컬렉션 타입을 리턴하도록 구현할 것이다.

읽기 전용 타입은 [공변\(covariant\)](#)이다. 이는 `Rectangle`이 `Shape`를 상속받은 경우, `List<Rectangle>`를 `List<Shape>`에 할당할 수 있다는 것을 뜻한다. 변경가능 컬렉션은 런타임에 실패가 발생할 수 있기 때문에 이를 허용하지 않는다.

특정 시점에 컬렉션의 스냅샷을 호출자에 리턴하는데 (원본 컬렉션을 변경해도) 리턴한 컬렉션은 바뀌지 않는 것을 원할 때가 있다.

```
class Controller {
    private val _items = mutableListOf<String>()
    val items: List<String> get() = _items.toList()
}
```

`toList` 확장 메서드는 리스트 항목을 복사하기 때문에 (원본을 변경해도) 리턴된 리스트가 절대로 바뀌지 않음을 보장한다.

리스트와 집합에 대해 익숙해지면 좋은 몇 가지 유용한 확장 함수가 있다.

```
val items = listOf(1, 2, 3, 4)
items.first == 1
items.last == 4
items.filter { it % 2 == 0 } // [2, 4] 리턴
rwList.requireNotNulls()
if (rwList.none { it > 6 }) println("No items above 6")
val item = rwList.firstOrNull()
```

... 또한, sort, zip, fold, reduce와 같은 유틸리티도 존재한다.

맵도 동일 패턴을 따른다. 다음과 같이 쉽게 생성하고 사용할 수 있다.

```
val readWriteMap = hashMapOf("foo" to 1, "bar" to 2)
println(readWriteMap["foo"])
val snapshot: Map<String, Int> = HashMap(readWriteMap)
```

범위

범위(Range) 식은 연산자 형식인 “..”를 갖는 `rangeTo` 함수로 구성된다. 이 식은 `in`이나 `!in`과 함께 쓰인다. 모든 `Comparable` 타입에 대해 범위를 정의할 수 있으며 기본 정수 타입은 최적화한 구현을 제공한다. 다음은 범위 사용 예이다.

```
if (i in 1..10) { // 1 <= i && i <= 10와 동일
    println(i)
}
```

정수 타입 범위(`IntRange`, `LongRange`, `CharRange`)는 특수 기능-이터레이션 가능한-을 제공한다. 컴파일러가 이를 자바의 인덱스 기반 `for`-루프와 동일하게 바꿔서 추가 오버헤드가 없다.

```
for (i in 1..4) print(i) // "1234" 출력

for (i in 4..1) print(i) // 아무것도 출력하지 않음
```

숫자를 역으로 이터레이션하고 싶다면? 간단하다. 표준 라이브러리에 있는 `downTo()` 함수를 사용하면 된다.

```
for (i in 4 downTo 1) print(i) // "4321" 출력
```

1씩 증가/감소가 아닌 지정한 단계만큼 숫자를 이터레이션하는 것은? 물론 가능하다. `step()` 함수를 쓰면 된다.

```
for (i in 1..4 step 2) print(i) // "13" 출력

for (i in 4 downTo 1 step 2) print(i) // "42" 출력
```

동작 방식

범위는 라이브러리의 공통 인터페이스인 `ClosedRange<T>` 를 구현한다.

`ClosedRange<T>` 는 `Comparable` 타입에 대한 수학적 의미의 닫힌 구간을 뜻한다. 이는 범위에 포함되는 `start` 와 `endInclusive` 의 두 끝지점을 갖는다. 주요 오퍼레이션인 `contains` 는 보통 `in` / `!in` 연산자 형식으로 사용한다.

정수 타입 프로그레션(`IntProgression`, `LongProgression`, `CharProgression`)은 숫자 진행을 뜻한다. 프로그레션은 `first` 요소, `last` 요소, 0이 아닌 `increment` 로 정의한다. 첫 번째 요소는 `first` 이고, 다음 요소는 이전 요소에 `increment` 를 더한 값이다. `last` 요소는 프로그레이션이 비어(empty) 있지 않으면 항상 도달한다.

프로그레션은 `Iterable<N>` 의 하위 타입으로 `N` 에는 `Int`, `Long`, `Char` 가 올 수 있으며, `for`-루프나 `map`, `filter` 와 같은 함수에서 사용할 수 있다. 프로그레션에 대한 이터레이션은 자바/자바스크립에서 다음의 인덱스 기반 `for`-루프와 동일하다.

```
for (int i = first; i != last; i += increment) {
    // ...
}
```

정수 타입에서, `..` 연산자는 `ClosedRange<T>` 와 `*Progression` 을 모두 구현한 객체를 생성한다. 예를 들어, `IntRange` 는 `ClosedRange<Int>` 를 구현하고 `IntProgression` 을 확장해서, `IntProgression` 에 정의된 모든 오퍼레이션을 `IntRange` 에서 사용 가능하다. `downTo()` 와 `setp()` 함수 결과는 항상 `*Progression` 이다.

프로그래션은 컴페니언 객체에 정의한 `fromClosedRange` 함수로 생성한다.

```
IntProgression.fromClosedRange(start, end, increment)
```

양수 `increment` 기준으로 `end` 값보다 크지 않은 최댓값을 또는 음수 `increment` 에 대해 `end` 값보다 작지 않은 최솟값을 찾기 위해 `(last - first) % increment == 0` 와 같은 식을 사용해서 프로그래션의 `last` 요소를 계산한다.

유틸리티 함수

`rangeTo()`

정수 타입 `rangeTo()` 연산자는 단순히 `*Range` 클래스의 생성자를 호출한다.

```
class Int {  
    //...  
    operator fun rangeTo(other: Long): LongRange = LongRange(this, other)  
    //...  
    operator fun rangeTo(other: Int): IntRange = IntRange(this, other)  
    //...  
}
```

실수형 숫자(`Double` , `Float`)는 `rangeTo` 연산자를 정의하지 않고 지네릭 `Comparable` 타입을 위해 표준 라이브러리가 제공하는 연산자를 대신 사용한다.

```
public operator fun <T: Comparable<T>> T.rangeTo(that: T): ClosedRange<T>
```

이 함수가 리턴하는 범위는 이터레이션할 수 없다.

`downTo()`

`downTo()` 는 정수 타입 쌍을 위한 확장 함수이다. 다음은 두 가지 예이다.

```
fun Long.downTo(other: Int): LongProgression {  
    return LongProgression.fromClosedRange(this, other, -1.0)  
}  
  
fun Byte.downTo(other: Int): IntProgression {  
    return IntProgression.fromClosedRange(this, other, -1)  
}
```

`reversed()`

`reversed()` 는 각 `*Progression` 클래스를 위한 확장 함수이다. 모든 확장 함수는 역순 프로그래션을 리턴한다.

```
fun IntProgression.reversed(): IntProgression {  
    return IntProgression.fromClosedRange(last, first, -increment)  
}
```


step()

step() 은 *Progression 클래스를 위한 확장 함수이다. 모든 확장 함수는 수정한 step 값(함수 파라미터)을 가진 프로그레션을 리턴한다. step 값은 항상 양수여야 한다. 따라서 이 함수는 이터레이션의 방향을 절대 바꾸지 않는다.

```
fun IntProgression.step(step: Int): IntProgression {
    if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
    return IntProgression.fromClosedRange(first, last, if (increment > 0) step else
    -step)
}

fun CharProgression.step(step: Int): CharProgression {
    if (step <= 0) throw IllegalArgumentException("Step must be positive, was: $step")
    return CharProgression.fromClosedRange(first, last, step)
}
```

(last - first) % increment == 0 규칙을 유지하기 위해 리턴한 프로그레션의 last 값은 원래 프로그레션의 last 외 다를 수 있다. 다음은 예이다.

```
(1..12 step 2).last == 11 // [1, 3, 5, 7, 9, 11] 값을 가진 프로그레션
(1..12 step 3).last == 10 // [1, 4, 7, 10] 값을 가진 프로그레션
(1..12 step 4).last == 9  // [1, 5, 9] 값을 가진 프로그레션
```

타입 검사와 변환

is와 !is 연산자

`is` 와 `!is` 연산자를 사용하면 런타임에 객체가 주어진 타입인지 아닌지 검사할 수 있다.

```
if (obj is String) {  
    print(obj.length)  
}  
  
if (obj !is String) { // !(obj is String)와 동일  
    print("Not a String")  
}  
else {  
    print(obj.length)  
}
```

스마트 변환

코틀린은 컴파일러가 불변 값에 대해 `is` -검사를 추적해서 필요하면 자동으로 (안전한) 변환을 추가하기 때문에, 많은 경우 명시적인 변환 연산을 사용할 필요가 없다.

```
fun demo(x: Any) {  
    if (x is String) {  
        print(x.length) // x를 자동으로 String으로 변환한다  
    }  
}
```

컴파일러가 똑똑하기 때문에 타입 비일치 검사에서 리턴하면 안전하게 변환할 수 있다는 것을 안다.

```
if (x !is String) return  
print(x.length) // x를 자동으로 String으로 변환한다
```

`&&` 와 `||` 의 우측에서도 안전하게 변환한다.

```
// `||`의 우측에 대해 x를 자동으로 문자열로 변환  
if (x !is String || x.length == 0) return  
  
// `&&`의 우측에 대해 x를 자동으로 문자열로 변환  
if (x is String && x.length > 0)  
    print(x.length) // x를 자동으로 String으로 변환한다
```

스마트 변환은 [when-식](#)과 [while-루프](#)에도 동작한다.

```
when (x) {
  is Int -> print(x + 1)
  is String -> print(x.length + 1)
  is IntArray -> print(x.sum())
}
```

컴파일러가 타입 검사와 사용 사이에 변수가 바뀌지 않는다는 것을 보장할 수 없으면 스마트 변환을 할 수 없다. 더 구체적으로 스마트 변환은 다음 규칙에 따라 가능하다.

- **val** 로컬 변수 - 항상 가능하다.
- **val** 프로퍼티 - 프로퍼티가 `private`이나 `internal`이거나 또는 프로퍼티를 선언한 같은 모듈에서 검사를 수행한 경우. `open` 프로퍼티나 커스텀 `getter`를 가진 프로퍼티에는 스마트 변환을 적용할 수 없다.
- **var** 로컬 변수 - 검사와 사용 사이에 변수가 바뀌지 않고 그것을 수정하는 람다에 캡처되지 않는 경우 가능하다.
- **var** 프로퍼티 - (변수를 언제 어디서든 수정할 수 있으므로) 절대 안 된다.

“안전하지 않은” 변환 연산자

보통 변환 연산자는 변환을 할 수 없으면 익셉션을 발생한다. 그래서 이 변환을 *안전하지 않다*고 부른다. 안전하지 않은 변환은 **as** 중위 연산자로 한다([연산자 우선순위](#) 참고):

```
val x: String = y as String
```

`String` 타입은 [nullable](#)이 아니므로 `null`을 `String`으로 변환할 수 없다. 예를 들어 `y`가 `null`이면 이 코드는 익셉션이 발생한다. 자바 변환 세만틱과 맞추기 위해 다음과 같이 변환 피연산자로 `nullable` 타입을 사용해야 한다.

```
val x: String? = y as String?
```

“안전한” (nullable) 변환 연산자

익셉션 발생을 피하려면 *안전하게* **as?** 변환 연산자를 사용할 수 있다. 이 연산자는 실패시 `null`을 리턴한다.

```
val x: String? = y as? String
```

as?의 우측 피연산자가 non-null 타입 `String` 임에도 불구하고 변환 결과가 `nullable` 임에 주목하자.

This 식

현재 `_리시버_`를 표시할 때 `this` 식을 사용한다.

- [클래스](#)의 멤버에서 `this`는 그 클래스의 현재 객체를 참조한다.
- [확장 함수](#)나 [리시버를 지정한 함수 리터럴](#)에서 `this`는 점의 왼쪽 편에 전달한 *리시버* 파라미터를 나타낸다.

`this`에 한정자가 없으면 `_가장 안쪽을 둘러싼 스코프_`를 참조한다. 다른 스코프에서 `this`를 참조하려면 `_라벨 한정자_`를 사용한다.

한정한 `this`

외부 스코프([클래스](#)나 [확장 함수](#) 또는 라벨이 붙은 [리시버를 사용하는 함수 리터럴](#))에서 `this`에 접근하려면 `this@label` 을 사용한다. `@label` 은 `this`로 접근하려는 스코프에 대한 [라벨](#)이다.

```
class A { // implicit label @A
  inner class B { // implicit label @B
    fun Int.foo() { // implicit label @foo
      val a = this@A // A의 this
      val b = this@B // B의 this

      val c = this // foo()의 리시버인 Int
      val c1 = this@foo // foo()의 리시버인 Int

      val funLit = lambda@ fun String.() {
        val d = this // funLit의 리시버
      }

      val funLit2 = { s: String ->
        // 둘러싼 람다 식이 리시버를 갖지 않으므로
        // foo()의 리시버
        val d1 = this
      }
    }
  }
}
```

동등성

코틀린에는 두 가지 동등성이 있다.

- 참조 동등성(Referential equality) (두 레퍼런스가 같은 객체를 참고)
- 구조적 동등성(Structural equality) (`equals()` 로 검사)

참조 동등성

참조 동등성은 `===` 오퍼레이션으로 검사한다(역은 `!==`). `a === b` 는 `a` 와 `b` 가 같은 객체를 참조하는 경우에만 `true`이다.

구조적 동등성

구조적 동등성은 `==` 오퍼레이션으로 검사한다(역은 `!=`). 규약에 따라 `a == b` 와 같은 식은 다음으로 변환된다.

```
a?.equals(b) ?: (b === null)
```

이 코드는 `a` 가 `null` 이 아니면 `equals(Any?)` 함수를 호출하고 그렇지 않으면(`a` 가 `null` 이면) `b` 가 `null` 을 참조하는지 검사한다.

`null` 을 직접 비교할 때 코드 최적화 포인트는 없다. `a == null` 을 자동으로 `a === null` 로 변환해준다.

연산자 오버로딩

코틀린은 작성한 타입에 대해 미리 정의한 연산자의 구현을 제공할 수 있다. 이들 연산자는 (+ 나 * 와 같은) 고정된 부호를 가지며 [우선순위](#)가 고정되어 있다. 연산자를 구현하려면 고정된 이름을 가진 [멤버 함수](#)나 [확장 함수](#)를 제공하면 된다. 대응하는 타입은 이항 연산자의 경우 좌측 피연산자 타입이고 단항 연산자는 인자 타입이다. 연산자를 오버로딩하는 함수는 `operator` 제한자로 지정해야 한다.

규칙

다른 연산자를 위해 연산자 오버로딩을 규정하는 규칙을 설명한다.

단항 오퍼레이션

식	변환
<code>+a</code>	<code>a.unaryPlus()</code>
<code>-a</code>	<code>a.unaryMinus()</code>
<code>!a</code>	<code>a.not()</code>


예를 들어 컴파일러는 `+a` 식을 다음 절차에 따라 처리한다.

- `a`의 타입을 결정한다. 타입을 `T`라고 하자.
- 리시버 `T`에서 파라미터를 갖지 않고 `operator` 제한자로 지정한 `unaryPlus()` 함수(멤버 함수나 확장 함수)를 찾는다.
- 함수가 없거나 모호하면 컴파일 에러를 낸다.
- 함수가 존재하고 리턴 타입이 `R`이면 `+a` 식은 `R` 타입을 갖는다.

이 오퍼레이션과 다른 오퍼레이션은 [기본 타입](#)에 맞게 최적화되므로 오프레이션을 위한 함수 호출에 따른 오버헤드가 없다.

식	변환
<code>a++</code>	<code>a.inc()</code> + 아래 참고
<code>a--</code>	<code>a.dec()</code> + 아래 참고

이 오퍼레이션은 리시버를 변경하면서 (선택적으로) 값을 리턴해야 한다.

 **`inc()/dec()`에서 리시버 객체를 변경하면 안 된다.**
“리시버를 변경한다”는 것은 리시버 객체가 아닌 `_리시버-변수_`를 의미한다.

컴파일러는 다음 과정에 따라 `a++`와 같은 `후위` 연산자를 해석한다.

- `a`의 타입을 결정한다. 타입을 `T`라고 해 보자.
- 리시버 타입 `T`에서 `operator` 제한자를 갖고 파라미터가 없는 `inc()` 함수를 찾는다.
- 함수가 `R` 타입을 리턴하면 `R`은 `T`의 하위타입이어야 한다.

식의 계산 결과는 다음과 같다.

- `a`의 초기 값을 임시 저장소인 `a0`에 보관한다,
- `a.inc()`의 결과를 `a`에 할당한다.

— 식의 결과로 `a0` 를 리턴한다.

`a--` 처리 과정도 완전히 동일하다.

`++a` 와 `--a` 와 같은 *전위* 식도 같은 방식으로 동작한다. 결과는 다음과 같다.

— `a.inc()` 의 결과를 `a` 에 할당한다,

— 식의 결과로 `a` 의 새 값을 리턴한다.

이항 오퍼레이션

식	변환
<code>a + b</code>	<code>a.plus(b)</code>
<code>a - b</code>	<code>a.minus(b)</code>
<code>a * b</code>	<code>a.times(b)</code>
<code>a / b</code>	<code>a.div(b)</code>
<code>a % b</code>	<code>a.mod(b)</code>
<code>a..b</code>	<code>a.rangeTo(b)</code>

이 표의 오퍼레이션에 대해 컴파일러는 단순히 *변환* 칼럼의 식을 실행한다.

Expression	Translated to
<code>a in b</code>	<code>b.contains(a)</code>
<code>a !in b</code>	<code>!b.contains(a)</code>

`in` 과 `!in` 의 경우 절차는 같고 인자의 순서가 반대이다.

심볼	변환
<code>a[i]</code>	<code>a.get(i)</code>
<code>a[i, j]</code>	<code>a.get(i, j)</code>
<code>a[i_1, ..., i_n]</code>	<code>a.get(i_1, ..., i_n)</code>
<code>a[i] = b</code>	<code>a.set(i, b)</code>
<code>a[i, j] = b</code>	<code>a.set(i, j, b)</code>
<code>a[i_1, ..., i_n] = b</code>	<code>a.set(i_1, ..., i_n, b)</code>

대괄호는 해당 개수만큼 인자를 가진 `get` 과 `set` 메서드 호출로 변환된다.

심볼	변환
<code>a()</code>	<code>a.invoke()</code>
<code>a(i)</code>	<code>a.invoke(i)</code>
<code>a(i, j)</code>	<code>a.invoke(i, j)</code>
<code>a(i_1, ..., i_n)</code>	<code>a.invoke(i_1, ..., i_n)</code>

괄호는 해당 개수의 인자를 갖는 `invoke` 호출로 바뀐다.

식	변환
<code>a += b</code>	<code>a.plusAssign(b)</code>
<code>a -= b</code>	<code>a.minusAssign(b)</code>
<code>a *= b</code>	<code>a.timesAssign(b)</code>
<code>a /= b</code>	<code>a.divAssign(b)</code>
<code>a %= b</code>	<code>a.modAssign(b)</code>

`a += b` 와 같은 할당 오퍼레이션의 경우 컴파일러가 다음 과정을 수행한다.

- 우측 칼럼의 함수를 사용할 수 있으면
 - 해당 이항 함수(`plusAssign()` 의 경우 `plus()`)가 존재하면 에러를 발생한다(모호함)
 - 리턴 타입이 `Unit` 인지 확인하고 아니면 에러를 발생한다.
 - `a.plusAssign(b)` 코드를 생성한다.
- 그렇지 않으면, `a = a + b` 코드 생성을 시도한다(이는 타입 검사를 포함한다. `a + b` 의 타입은 `a` 의 하위타입이어야 한다).

주의: 할당은 코틀린에서 식이 *아니다*.

식	변환
<code>a == b</code>	<code>a?.equals(b) ?: b === null</code>
<code>a != b</code>	<code>!(a?.equals(b) ?: b === null)</code>

노트: `===` 와 `!==` (동일성 검사)는 오버로딩할 수 없으므로 어떤 규칙도 존재하지 않는다

`==` 오퍼레이션은 특수하다. `null` 을 걸러내어 `null == null` 이 `true` 가 되는 복잡한 식으로 바뀐다.

심볼	변환
<code>a > b</code>	<code>a.compareTo(b) > 0</code>
<code>a < b</code>	<code>a.compareTo(b) < 0</code>
<code>a >= b</code>	<code>a.compareTo(b) >= 0</code>
<code>a <= b</code>	<code>a.compareTo(b) <= 0</code>

모든 비교는 `compareTo` 에 대한 호출로 바뀐다. `compareTo` 함수는 `Int` 를 리턴해야 한다.

네임드 함수에 대한 중위 호출

[중위 함수 호출](#)을 사용해서 커스텀 중위 오퍼레이션을 흉내낼 수 있다.

Null 안전성

Nullable 타입과 non-null 타입

코틀린 타입 시스템은 [The Billion Dollar Mistake](#)라고 불리는 null 참조 위험을 제거하는데 목적이 있다.

자바를 포함한 많은 프로그래밍 언어에서 가장 흔한 위험 중 하나가 null 레퍼런스의 멤버에 접근해서 발생하는 null 참조 익셉션이다. 자바에서는 `NullPointerException` 또는 줄여서 `NPE`가 이에 해당한다.

코틀린 타입 시스템은 코드에서 `NullPointerException` 을 제거하려고 노력했다. 오직 다음 경우만 `NPE`가 발생한다.

- 직접 `throw NullPointerException()` 을 실행
- 아래 설명할 `!!` 연산자 사용
- 외부 자바 코드에서 발생
- 초기화에 관한 데이터 불일치 존재(생성자에서 초기화하지 않은 `this`를 어딘가에서 사용)

코틀린 타입 시스템은 `null`을 가질 수 있는 레퍼런스(nullable 레퍼런스)와 가질 수 없는 레퍼런스(non-null 레퍼런스)를 구분한다. 예를 들어 일반 `String` 타입 변수는 `null`을 가질 수 없다.

```
var a: String = "abc"
a = null // 컴파일 에러
```

`null`을 가지려면 `String?` 로 쓴 nullable `String`을 변수로 선언해야 한다.

```
var b: String? = "abc"
b = null // ok
```

`a`의 프로퍼티에 접근하거나 메서드를 호출할 때 `NPE`가 발생하지 않음을 보장할 수 있으며 다음 코드가 안전하다고 할 수 있다.

```
val l = a.length
```

하지만 `b`의 프로퍼티에 접근하면 안전하지 않으므로 컴파일러는 에러를 발생한다.

```
val l = b.length // 에러: 변수 'b'가 null일 수 있다
```

그래도 `b`의 프로퍼티에 접근하고 싶다면 몇 가지 할 수 있는 방법이 있다.

조건에서 `null`을 검사하기

첫 번째 방법은 `b`가 `null`인지 검사하고 두 상황을 각각 처리하는 것이다.

```
val l = if (b != null) b.length else -1
```

컴파일러는 검사 결과를 추적하며 `if` 안에서 `length`를 호출할 수 있도록 한다. 더 복잡한 조건도 지원한다.

```
if (b != null && b.length > 0)
    print("String of length ${b.length}")
else
    print("Empty string")
```

이 코드는 오직 `b` 가 불변(예, 검사와 사용 사이에 바뀌지 않는 로컬 변수 또는 backing 필드를 갖거나 오버라이딩할 수 없는 `val` 멤버)인 경우에만 동작한다. 왜냐하면 불변이 아니면 검사 이후에 `b` 를 `null`로 바꾸는 일이 벌어질 수 있기 때문이다.

안전한 호출

두 번째 방법은 안전 호출 연산자인 `?.` 를 사용하는 것이다.

```
b?.length
```

`b` 가 `null`이 아니면 `b.length` 를 리턴하고 아니면 `null`을 리턴한다. 이 식의 타입은 `Int?` 이다.

안전 호출은 연속할 때 유용하다. 예를 들어, `Employee` 타입 `bob`을 `Department`에 할당하거나 그렇지 않을 수 있고 또 다른 `Employee`를 `Department`의 head로 가질 수 있을 때, `Bob`의 department head가 존재하면 그 이름을 구하는 코드를 다음과 같이 작성할 수 있다.

```
bob?.department?.head?.name
```

프로퍼티 중 하나라도 `null`이면 이 체인은 `null`을 리턴한다.

엘비스 연산자

nullable 레퍼런스 `r` 이 있을 때, “`r` 이 not `null`이면 그것을 사용하고 아니면 non-`null` 값 `x` 를 사용”하는 코드는 다음과 같이 작성한다.

```
val l: Int = if (b != null) b.length else -1
```

완전한 `if`-식 대신 엘비스 연산자인 `?:` 를 사용해서 이를 표현할 수 있다.

```
val l = b?.length ?: -1
```

`?:` 의 왼쪽 식이 `null`이 아니면 엘비스 연산자는 그것을 리턴하고, 그렇지 않으면 오른쪽 식을 리턴한다. 우측 식은 왼쪽 식이 `null`인 경우에만 평가한다.

코틀린에서 `throw`와 `return`은 식이기 때문에 엘비스 연산자의 우측에 사용할 수 있다. 이는 함수 인자를 검사할 때 매우 유용하게 쓸 수 있다.

```
fun foo(node: Node): String? {
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // ...
}
```

!! 연산자

세 번째 방법은 NPE-추종자를 위한 것이다. `b!!` 라고 작성하면 `b` 가 non-null이면 값을 리턴하고(이 예에서는 `String`) `b` 가 null이면 NPE를 발생한다.

```
val l = b!!.length()
```

따라서 NPE를 원한다면 NPE를 사용할 수 있다. 하지만 NPE를 명시적으로 써야만 한다면 생각하지 못한 곳에서 NPE가 발생하면 안 된다.

안전한 변환

일반 변환은 객체가 대상 타입이 아니면 `ClassCastException` 을 발생한다. 다른 옵션은 변환에 실패할 때 `null`을 리턴하는 안전 변환을 사용하는 것이다.

```
val aInt: Int? = a as? Int
```

익셉션

익셉션 클래스

코틀린의 모든 익셉션 클래스는 `Throwable` 클래스의 자식이다. 모든 익셉션은 메시지, 스택 트레이스 그리고 선택적으로 원인을 갖는다.

익셉션 객체를 발생하려면 `throw`-식을 사용한다.

```
throw MyException("Hi There!")
```

익셉션을 잡으려면 `try`-식을 사용한다.

```
try {  
    // some code  
}  
catch (e: SomeException) {  
    // handler  
}  
finally {  
    // optional finally block  
}
```

`catch` 블록은 없거나 한 개 이상 존재할 수 있다. `finally` 블록은 생략할 수 있다. 하지만 최소한 한 개의 `catch` 블록이나 `finally` 블록이 있어야 한다.

`try`는 식이다

`try`는 식이며 리턴 값을 가질 수 있다.

```
val a: Int? = try { parseInt(input) } catch (e: NumberFormatException) { null }
```

`try` 블록의 마지막 식이나 `catch` 블록(또는 블록들)의 마지막 식을 `try`-식의 리턴 값으로 사용한다. `finally` 블록의 내용은 식 결과에 영향을 주지 않는다.

체크트 익셉션

코틀린에는 체크트 익셉션이 없다. 없는데는 여러 이유가 있는데 간단한 이유를 들어보겠다.

다음은 `StringBuilder` 클래스가 구현한 JDK 인터페이스이다.

```
Appendable append(CharSequence csq) throws IOException;
```

이 시그니처가 무엇을 말하나? 이는 문자열을 어딘가에 저장할 때마다(예를 들어 `StringBuilder`, 어떤 종류의 로그, 콘솔 등) `IOException` 을 캐치해야 한다. 왜냐면 IO를 수행할지도 모르기 때문이다(`Writer` 도 `Appendable` 을 구현하고 있다). 따라서 도처에서 다음과 같은 코드를 작성하게 된다.

```
try {  
    log.append(message)  
}  
catch (IOException e) {  
    // 안전해야 함  
}
```

이는 좋지 않다. [Effective Java](#), Item 65: *Don't ignore exceptions* 절을 보자.

Bruce Eckel는 [자바에 체크드 익셉션이 필요한가?](#)라고 말했다.

작은 프로그램 조사에서는 익셉션 규약이 개발자 생산성과 코드 품질을 높여준다는 결론을 이끌지만, 대형 소프트웨어 프로젝트에서의 경험은 오히려 생산성을 낮추고 코드 품질 상승에 거의 효과가 없음이 밝혀졌다.

이와 관련된 글:

- [Java's checked exceptions were a mistake](#) (Rod Waldhoff)
- [The Trouble with Checked Exceptions](#) (Anders Hejlsberg)

자바 상호운용

자바 상호운용에 대한 정보는 [자바 상호운용](#)의 익셉션 절을 참고하자.

애노테이션

애노테이션 선언

애노테이션은 코드에 메타데이터를 추가하는 방법이다. 애노테이션을 선언하려면 **annotation** 제한자를 클래스 앞에 넣으면 된다.

```
annotation class Fancy
```

애노테이션 클래스에 메타 애노테이션을 붙여서 애노테이션의 속성을 지정할 수 있다.

- **@Target**은 애노테이션을 할 수 있는 요소 종류를 지정한다(클래스, 함수, 프로퍼티, 식 등);
- **@Retention**은 애노테이션을 컴파일한 클래스에 보관할지 여부와 런타임에 리플렉션을 통해 접근할 수 있는지 여부를 지정한다(기본은 둘 다 true);
- **@Repeatable**은 같은 애노테이션을 한 요소에 여러 번 적용할 수 있는지 여부를 지정한다;
- **@MustBeDocumented**은 애노테이션이 공개 API에 속하는지 여부와 생성한 API 문서의 클래스나 메서드 시그너처에 포함해야 하는지 여부를 지정한다.

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,  
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)  
@Retention(AnnotationRetention.SOURCE)  
@MustBeDocumented  
public annotation class Fancy
```

용법

```
@Fancy class Foo {  
    @Fancy fun baz(@Fancy foo: Int): Int {  
        return (@Fancy 1)  
    }  
}
```

클래스의 주요 생성자에 애노테이션을 하고 싶으면 **constructor** 키워드를 생성자 선언에 추가하고 그 키워드 앞에 애노테이션을 넣으면 된다.

```
class Foo @Inject constructor(dependency: MyDependency) {  
    // ...  
}
```

프로퍼티 accessor에도 애노테이션 할 수 있다.

```
class Foo {  
    var x: MyDependency? = null  
    @Inject set  
}
```

생성자

애노테이션은 파라미터가 있는 생성자를 가질 수 있다.

```
annotation class Special(val why: String)

@Special("example") class Foo {}
```

허용하는 파라미터 타입은 다음과 같다.

- 자바의 기본 타입에 해당하는 타입(Int, Long 등)
- 문자열
- 클래스(Foo::class)
- 열거형
- 다른 애노테이션
- 위에 나열한 타입의 배열

애노테이션을 다른 애노테이션의 파라미터로 사용하면 @ 문자를 이름 앞에 붙이지 않는다.

```
public annotation class ReplaceWith(val expression: String)

public annotation class Deprecated(
    val message: String,
    val replaceWith: ReplaceWith = ReplaceWith(""))

@Deprecated("This function is deprecated, use === instead", ReplaceWith("this === other"))
```

람다

람다에서도 애노테이션을 사용할 수 있다. 람다의 몸체를 생성할 때 invoke() 메서드에 애노테이션을 적용한다. 동시성 제어를 위해 애노테이션을 사용하는 [Quasar](#)와 같은 프레임워크에서 이를 유용하게 쓰고 있다.

```
annotation class Suspending

val f = @Suspending { Fiber.sleep(10) }
```

사용-위치(Use-site) 대상 애노테이션

주요 생성자 파라미터나 프로퍼티는 여러 자바 요소를 생성할 수 있다. 따라서 이 코틀린 요소에 애노테이션을 달면 생성한 자바 바이트코드의 여러 위치에 애노테이션이 붙을 수 있다. 정확하게 애노테이션을 어느 자바 요소에 생성할지 지정하려면 다음 구문을 사용한다.

```
class Example(@field:Ann val foo,    // 자바 필드
              @get:Ann val bar,     // 자바 getter
              @param:Ann val quux)  // 자바 생성자 파라미터
```

같은 구문을 전체 파일을 애노테이션 할 때에 사용할 수 있다. 이를 하기 위해 패키지 디렉티브 전에 또는 기본 패키지면 모든 임포트 전에 파일 최상단에 대상이 file 인 애노테이션을 넣는다.

```
@file:JvmName("Foo")
```

```
package org.jetbrains.demo
```

여러 애노테이션을 같은 대상에 적용하고 싶다면 대상 뒤에 대괄호 안에 모든 애노테이션을 넣어서 대상을 반복하는 것을 피할 수 있다.

```
class Example {  
    @set:[Inject VisibleForTesting]  
    public var collaborator: Collaborator  
}
```

지원하는 전체 사용 위치(use-site) 대상은 다음과 같다.

- `file`
- `property` (이 대상을 갖는 애노테이션은 자바에는 보이지 않는다)
- `field`
- `get` (프로퍼티 getter)
- `set` (프로퍼티 setter)
- `receiver` (확장 함수나 프로퍼티의 리시버 파라미터)
- `param` (생성자 파라미터)
- `setparam` (프로퍼티 setter 파라미터)
- `delegate` (위임 프로퍼티를 위한 위임 인스턴스를 보관한 필드)

확장 함수의 리시버 파라미터를 애노테이션 하려면 다음 구문을 사용한다.

```
fun @receiver:Fancy String.myExtension() { }
```

사용 위치(use-site) 대상을 지정하지 않으면 사용할 애노테이션의 `@Target` 애노테이션에 따라 대상을 선택한다. 만약 여러 대상이 적용 가능하면 다음 목록에서 먼저 적용할 수 있는 대상을 선택한다.

- `param`
- `property`
- `field`

자바 애노테이션

자바 애노테이션은 코틀린과 100% 호환된다.


```
import org.junit.Test
import org.junit.Assert.*

class Tests {
    @Test fun simple() {
        assertEquals(42, getTheAnswer())
    }
}
```

자바로 작성한 애노테이션은 파라미터 순서가 없기 때문에 인자를 전달할 때 일반 함수 호출 구문을 사용할 수 없다. 대신 이름 인자 구문을 사용해야 한다.

```
// Java
public @interface Ann {
    int intValue();
    String stringValue();
}
```

```
// Kotlin
@Ann(intValue = 1, stringValue = "abc") class C
```

자바와 마찬가지로 `value` 파라미터는 특수 케이스다. 이름 없이 이 파라미터 값을 지정할 수 있다.

```
// Java
public @interface AnnWithValue {
    String value();
}
```

```
// Kotlin
@AnnWithValue("abc") class C
```

자바에서 `value` 인자가 배열 타입이면 코틀린의 `vararg` 파라미터가 된다.

```
// Java
public @interface AnnWithArrayValue {
    String[] value();
}
```

```
// Kotlin
@AnnWithArrayValue("abc", "foo", "bar") class C
```

애노테이션 인자로 클래스를 지정하고 싶다면 코틀린 클래스 ([KClass](#))를 사용한다. 코틀린 컴파일러는 `KClass`를 자동으로 자바 클래스로 변환하므로 자바 코드는 애노테이션과 인자를 볼 수 있다.

```
import kotlin.reflect.KClass

annotation class Ann(val arg1: KClass<*>, val arg2: KClass<out Any?>)

@Ann(String::class, Int::class) class MyClass
```

애노테이션 인스턴스의 값은 코틀린 코드에서 프로퍼티로 노출된다.

```
// Java
public @interface Ann {
    int value();
}
```

```
// Kotlin
fun foo(ann: Ann) {
    val i = ann.value
}
```

리플렉션

리플렉션은 런타임에 프로그램의 구조를 인트로스펙션(introspection)할 수 있는 언어와 라이브러리 기능 집합이다. 코틀린 언어에서 함수와 프로퍼티는 일급이고 그것을 인트로스펙션(예, 런타임에 함수나 프로퍼티 타입의 이름을 알아내는 것)하는 것은 단순히 함수형이나 리액티브 방식을 사용하는 것과 밀접하게 관련되어 있다.

⚠ 자바 플랫폼에서, 리플렉션 기능을 사용하는데 필요한 런타임 컴포넌트를 별도 JAR 파일(`kotlin-reflect.jar`)로 배포하고 있다. 이는 리플릭션 기능을 사용하지 않는 어플리케이션이 필요로 하는 런타임 라이브러리의 크기를 줄여준다. 리플렉션을 사용하려면 이 jar 파일을 프로젝트 클래스패스에 넣어야 한다.

클래스 레퍼런스

가장 기본적인 리플렉션 기능은 코틀린 클래스에 대한 런타임 레퍼런스를 얻는 것이다. 정적으로 아는 코틀린 클래스의 레퍼런스를 구하려면 *클래스 리터럴* 구문을 사용하면 된다.

```
val c = MyClass::class
```

이 레퍼런스의 값은 `KClass` 타입이다.

코틀린 클래스 레퍼런스는 자바 클래스 레퍼런스와 다르다. 자바 클래스 레퍼런스를 구하려면 `KClass` 인스턴스의 `.java` 프로퍼티를 사용해야 한다.

함수 레퍼런스

다음과 같은 이름을 가진 함수 선언이 있다고 하자:

```
fun isOdd(x: Int) = x % 2 != 0
```

쉽게 함수를 직접 호출할 수 있다(`isOdd(5)`). 또한, 함수를 다른 함수에 값으로 전달할 수도 있다. 이를 하려면 `::` 연산자를 사용한다.

```
val numbers = listOf(1, 2, 3)
println(numbers.filter(::isOdd)) // [1, 3] 출력
```

여기서 `::isOdd` 는 함수 타입 `(Int) -> Boolean` 의 값이다.

오버로딩한 함수에 대해서는 `::` 연산자를 사용할 수 없다. 향후에 오버로딩한 함수 중에서 선택할 수 있도록 파라미터 타입을 지정하는 구문을 제공할 계획이다.

클래스의 멤버나 확장 함수를 사용해야 한다면 클래스 이름을 명시한다. 예를 들어, `String::toCharArray` 는 `String` 을 위한 확장 함수인 `String.() -> CharArray` 를 제공한다.

예제: 함수 조합

다음 함수를 보자:

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { x -> f(g(x)) }
}
```

이는 전달한 두 함수를 조합해서 리턴한다(`compose(f, g) = f(g(*))`). 이제 이 함수에 호출할 수 있는 레퍼런스를 적용할 수 있다.

```
fun length(s: String) = s.size

val oddLength = compose(::isOdd, ::length)
val strings = listOf("a", "ab", "abc")

println(strings.filter(oddLength)) // "[a, abc]" 출력
```

프로퍼티 레퍼런스

코틀린에서 일급 객체인 프로퍼티에 접근할 때에도 `::` 연산자를 사용한다.

```
var x = 1

fun main(args: Array<String>) {
    println(::x.get()) // "1" 출력
    ::x.set(2)
    println(x)         // "2" 출력
}
```

`::x` 식은 `KProperty<Int>` 타입의 프로퍼티 객체를 구한다. 이 타입을 이용하면 `get()` 을 사용해서 값을 읽거나 `name` 프로퍼티를 이용해서 프로퍼티 이름을 구할 수 있다. 더 자세한 정보는 [KProperty 클래스 문서](#)를 참고한다.

`var y = 1` 과 같은 수정 가능 프로퍼티의 경우 `::y` 는 `KMutableProperty<Int>` 타입 값을 리턴한다. 이 타입은 `set()` 메서드를 갖고 있다.

파라미터가 없는 함수가 필요한 곳에 프로퍼티 레퍼런스를 사용할 수 있다.

```
val strs = listOf("a", "bc", "def")
println(strs.map(String::length)) // [1, 2, 3] 출력
```

클래스의 멤버인 프로퍼티에 접근할 때에는 클래스를 한정한다.

```
class A(val p: Int)

fun main(args: Array<String>) {
    val prop = A::p
    println(prop.get(A(1))) // prints "1"
}
```

확장 프로퍼티의 경우:

```

val String.lastChar: Char
    get() = this[size - 1]

fun main(args: Array<String>) {
    println(String::lastChar.get("abc")) // prints "c"
}

```

자바 리플렉션과의 상호 운용성

자바 플랫폼에서, 표준 라이브러리는 리플렉션 클래스를 위해 자바 리플렉션 객체와의 매핑을 제공하는 확장을 포함하고 있다 (`kotlin.reflect.jvm` 패키지 참고). 예를 들어 backing 필드나 코틀린 프로퍼티의 `getter`를 위한 자바 메서드를 찾고 싶다면 다음과 같은 코드를 사용할 수 있다.

```

import kotlin.reflect.jvm.*

class A(val p: Int)

fun main(args: Array<String>) {
    println(A::p.javaGetter) // "public final int A.getP()" 출력
    println(A::p.javaField)  // "private final int A.p" 출력
}

```

자바 클래스에 해당하는 코틀린 클래스를 구하려면 확장 프로퍼티로 `.kotlin` 을 사용한다.

```

fun getKClass(o: Any): KClass<Any> = o.javaClass.kotlin

```

생성자 레퍼런스

메서드나 프로퍼티처럼 생성자 레퍼런스를 구할 수 있다. 생성자와 같은 파라미터를 갖고 관련 타입 객체를 리턴하는 함수 타입이 필요한 곳에 생성자 레퍼런스를 사용할 수 있다. 다음 함수는 `::` 연산자와 클래스 이름을 사용해서 생성자 레퍼런스를 구한다. 이 함수는 파라미터를 갖지 않고 리턴 타입이 `Foo` 인 함수를 파라미터로 사용한다.

```

class Foo

fun function(factory : () -> Foo) {
    val x : Foo = factory()
}

```

`::Foo` 를 사용하면, 즉 `Foo` 클래스의 인자 없는 생성자 레퍼런스로, 다음처럼 간단히 생성자를 호출할 수 있다.

```

function(::Foo)

```

타입-안전 빌더

빌더 개념은 [그루비](#)/커뮤니티에서 더 유명하다. 빌더는 반쯤 선언적인 방법으로 데이터를 정의할 수 있도록 해 준다. 빌더의 좋은 예로 [XML 생성](#), [컴포넌트 배치](#), [3D 장면 묘사](#) 등이 있다.

많은 유스케이스를 위해 코틀린은 *타입-검사*(*type-check*) 빌더를 제공한다. 이 빌더는 그루비 자체에서 만든 동적-타입 구현보다 더 매력적으로 이런 예를 구현할 수 있게 해준다.

코틀린은 동적 타입 빌더도 지원한다.

타입-안전 빌더 예제

다음 코드를 보자:

```
import com.example.html.* // 아래 선언 참고

fun result(args: Array<String>) =
    html {
        head {
            title {+"XML encoding with Kotlin"}
        }
        body {
            h1 {+"XML encoding with Kotlin"}
            p {+"this format can be used as an alternative markup to XML"}

            // 애트리뷰트와 텍스트 콘텐츠를 가진 엘리먼트
            a(href = "http://kotlinlang.org") {+"Kotlin"}

            // 혼합 컨텍스트
            p {
                +"This is some"
                b {+"mixed"}
                +"text. For more see the"
                a(href = "http://kotlinlang.org") {+"Kotlin"}
                +"project"
            }
            p {+"some text"}

            // 생성된 컨텍스트
            p {
                for (arg in args)
                    +arg
            }
        }
    }
```

이 코드는 완전히 올바른 코틀린 코드이다. 이 코드를 브라우저에서 수정하고 실행해 볼 수 있다([여기](#)).

동작 방식

코틀린에서 타입-안전 빌더를 구현하는 기법을 차례대로 살펴보자. 먼저 만들고 싶은 모델을 정의해야 한다. 이 예제의 경우 HTML 태그의 모델을 정의할 필요가 있다. 몇 개 클래스로 쉽게 이 모델을 만들 수 있다. 예를 들어, `HTML` 클래스는 `<html>` 태그를 표현하며, `<head>` 와 `<body>` 를 자식으로 정의한다. (아래 이 클래스의 선언을 참고한다.)

이제 왜 다음과 같은 코드를 작성할 수 있는지 보자:

```
html {  
  // ...  
}
```

`html` 은 실제로 [람다식](#)을 인자로 받는 함수 호출이다. 이 함수는 다음과 같이 정의되어 있다.

```
fun html(init: HTML.() -> Unit): HTML {  
    val html = HTML()  
    html.init()  
    return html  
}
```

이 함수는 이름이 `init` 인 파라미터를 갖는다. 이 파라미터 자체도 함수이다. `init` 함수 타입은 `_리시버를 갖는 함수 타입_인 HTML.() -> Unit 이다. 이는 함수에 HTML 타입의 인스턴스(리시버)를 전달해야 하고 함수 안에서 그 인스턴스의 멤버를 호출할 수 있음을 의미한다. this 키워드로 리시버에 접근할 수 있다.`

```
html {  
    this.head { /* ... */ }  
    this.body { /* ... */ }  
}
```

(`head` 와 `body` 는 `html` 의 멤버 함수이다.)

여기서 보통 `this`를 생략할 수 있다. 이 코드는 이미 빌더와 같은 모양이다.

```
html {  
    head { /* ... */ }  
    body { /* ... */ }  
}
```

그러면 이 함수 호출은 무엇을 할까? 위에 정의한 `html` 함수의 몸체를 보자. 이 함수는 새로운 `HTML` 인스턴스를 생성하고, 인자로 전달받은 함수를 호출해서 생성한 인스턴스를 초기화하고(이 예제에서는 `HTML` 인스턴스의 `head` 와 `body` 를 호출한다), 그 인스턴스를 리턴한다. 이것이 정확하게 빌더가 해야 하는 것이다.

`HTML` 클래스의 `head` 와 `body` 함수는 `html` 과 비슷하게 정의한다. 유일한 차이점은 둘러싼 `HTML` 인스턴스의 `children` 컬렉션에 생성한 인스턴스를 추가하는 것이다.

```

fun head(init: Head.() -> Unit) : Head {
    val head = Head()
    head.init()
    children.add(head)
    return head
}

fun body(init: Body.() -> Unit) : Body {
    val body = Body()
    body.init()
    children.add(body)
    return body
}

```

실제 이 두 함수는 같은 것을 하므로 지네릭 버전인 `initTag` 를 만들 수 있다.

```

protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
    tag.init()
    children.add(tag)
    return tag
}

```

이제 두 함수가 매우 간단해진다.

```

fun head(init: Head.() -> Unit) = initTag(Head(), init)

fun body(init: Body.() -> Unit) = initTag(Body(), init)

```

`<head>` 와 `<body>` 태그를 생성할 때 두 함수를 사용할 수 있다.

여기서 논의하는 것 중 다른 하나는 태그 몸체에 텍스트를 추가하는 것이다. 앞서 예제에서 다음과 같이 추가했다.

```

html {
    head {
        title {+"XML encoding with Kotlin"}
    }
    // ...
}

```

기본적으로 단순히 태그 몸체에 문자열을 넣는데 그 앞에 `+` 가 있다. 따라서 이는 접두 `unaryPlus()` 오퍼레이션을 실행하는 함수 호출이다. 실제로 이 오퍼레이션을 `TagWithText` 추상 클래스(`Title` 의 부모)의 멤버인 `unaryPlus()` 확장 함수로 정의했다.

```

fun String.unaryPlus() {
    children.add(TextElement(this))
}

```

따라서, 여기서 접두문자 `+` 는 문자열을 `TextElement` 의 인스턴스로 감싸고 그 인스턴스를 `children` 컬렉션에 추가해서, 그것이 태그 트리에 알맞은 부분이 되도록 한다.

이 모든 것이 `com.example.html` 패키지에 정의되어 있는데 위 벌더 예제는 처음에 이 패키지를 임포트한다. 다음 절에서 이 패키지의 전체 정의를 읽을 수 있다.

com.example.html 패키지의 전체 정의

다음 코드는 `com.example.html` 패키지의 코드이다(위 예제에서 사용하는 요소만 포함). 이는 HTML 트리를 만든다. [확장 함수](#)와 [리시버를 가진 람다](#)를 많이 쓰고 있다.

```
package com.example.html

interface Element {
    fun render(builder: StringBuilder, indent: String)
}

class TextElement(val text: String) : Element {
    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent$text\n")
    }
}

abstract class Tag(val name: String) : Element {
    val children = arrayListOf<Element>()
    val attributes = hashMapOf<String, String>()

    protected fun <T : Element> initTag(tag: T, init: T.() -> Unit): T {
        tag.init()
        children.add(tag)
        return tag
    }

    override fun render(builder: StringBuilder, indent: String) {
        builder.append("$indent<$name${renderAttributes()}>\n")
        for (c in children) {
            c.render(builder, indent + " ")
        }
        builder.append("$indent</$name>\n")
    }

    private fun renderAttributes(): String? {
        val builder = StringBuilder()
        for (a in attributes.keys) {
            builder.append(" $a=\"${attributes[a]}\"")
        }
        return builder.toString()
    }

    override fun toString(): String {
        val builder = StringBuilder()
        render(builder, "")
        return builder.toString()
    }
}
```

```

abstract class TagWithText(name: String) : Tag(name) {
    operator fun String.unaryPlus() {
        children.add(TextElement(this))
    }
}

class HTML() : TagWithText("html") {
    fun head(init: Head.() -> Unit) = initTag(Head(), init)

    fun body(init: Body.() -> Unit) = initTag(Body(), init)
}

class Head() : TagWithText("head") {
    fun title(init: Title.() -> Unit) = initTag(Title(), init)
}

class Title() : TagWithText("title")

abstract class BodyTag(name: String) : TagWithText(name) {
    fun b(init: B.() -> Unit) = initTag(B(), init)
    fun p(init: P.() -> Unit) = initTag(P(), init)
    fun h1(init: H1.() -> Unit) = initTag(H1(), init)
    fun a(href: String, init: A.() -> Unit) {
        val a = initTag(A(), init)
        a.href = href
    }
}

class Body() : BodyTag("body")
class B() : BodyTag("b")
class P() : BodyTag("p")
class H1() : BodyTag("h1")

class A() : BodyTag("a") {
    public var href: String
        get() = attributes["href"]!!
        set(value) {
            attributes["href"] = value
        }
}

fun html(init: HTML.() -> Unit): HTML {
    val html = HTML()
    html.init()
    return html
}

```

동적 타입

⚠ 동적 타입은 JVM 대상 코드에서 지원하지 않는다

코틀린은 정적 타입 언어이지만, 자바스크립트 에코시스템과 같이 untyped거나 타입이 유연한 환경에서도 돌아야 한다. 이런 환경을 쉽게 처리하기 위해 언어에서 동적 타입을 사용할 수 있다.

```
val dyn: dynamic = ...
```

`dynamic` 타입은 기본적으로 코틀린의 타입 검사를 하지 않는다. - 이 타입의 값은 모든 변수에 할당하거나 파라미터로 어디든 전달할 수 있다. - `dynamic` 타입 변수에 모든 값을 할당할 수 있고, 함수의 `dynamic` 파라미터에 모든 값을 전달할 수 있다. - 이 값에는 `null` -검사를 할 수 없다.

`dynamic` 의 가장 특별한 기능은 `dynamic` 변수에 대해 파라미터를 갖는 함수나 모든 프로퍼티를 호출할 수 있다는 것이다.

```
dyn.whatever(1, "foo", dyn) // 어디에도 'whatever'가 정의되어 있지 않음
dyn.whatever(*arrayOf(1, 2, 3))
```

자바스크립트 플랫폼에서, 이 코드는 있는 그대로 컴파일된다. 즉 코틀린의 `dyn.whatever(1)` 코드가 생성한 자바스크립트 코드에서도 `dyn.whatever(1)` 가 된다.

동적 호출은 항상 결과로 `dynamic` 을 리턴하므로 자유롭게 호출을 연결할 수 있다.

```
dyn.foo().bar.baz()
```

동적 호출에 람다를 전달하면 기본적으로 모든 파라미터는 `dynamic` 타입을 갖는다.

```
dyn.foo {
    x -> x.bar() // x가 dynamic
}
```

기술적인 내용은 [스펙 문서](#)를 참고한다.

레퍼런스

문법

문법 정의를 그 형식에 맞게 개정하는 중이다. 그 때까지, [이전 사이트의 문법](#) 문서를 참고한다.

상호 운용

코틀린에서 자바 코드 호출하기

코틀린은 자바와의 상호운용성을 염두에 두고 설계했다. 특별한 노력 없이 코틀린에서 기존의 자바 코드를 호출할 수 있고, 또한 자바에서도 비교적 매끄럽게 코틀린 코드를 사용할 수 있다. 이 절에서는 코틀린에서 자바 코드를 호출하는 것에 대한 내용을 자세히 설명한다.

거의 모든 자바 코드를 별 문제없이 사용할 수 있다.

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 자바 컬렉션을 'for'-루프에서 사용:
    for (item in source)
        list.add(item)
    // 연산자 규칙도 작동:
    for (i in 0..source.size() - 1)
        list[i] = source[i] // get과 set 호출
}
```

Getter와 Setter

getter와 setter를 위한 자바 규칙을 따르는 메서드는(인자가 없고 이름이 `get` 으로 시작하는 메서드와 한 개 인자를 갖고 이름이 `set` 으로 시작하는 메서드) 코틀린에서 프로퍼티로 표현된다. 다음 예를 보자.

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) { // getFirstDayOfWeek() 호출
        calendar.firstDayOfWeek = Calendar.MONDAY // setFirstDayOfWeek() 호출
    }
}
```

자바 클래스가 setter만 가진 경우 코틀린에서 프로퍼티로 보이지 않는다. 왜냐하면 코틀린은 아직 쓰기 전용 프로퍼티를 지원하지 않기 때문이다.

void를 리턴하는 메서드

void를 리턴하는 자바 메서드를 코틀린에서 실행하면 Unit 을 리턴한다. 그 리턴 값을 사용하려고 하면 그 값을 코틀린 컴파일러는 미리 (Unit 임을) 알 수 있으므로 컴파일러가 호출 위치(call site)에 할당한다.

코틀린에서 키워드인 자바 식별자를 위한 이스케이프

in, object, is 등 코틀린 키워드 중에서 일부는 자바에서 올바른 식별자이다. 자바 라이브러리가 메서드 이름으로 코틀린 키워드를 사용하면 역따옴표(`) 문자로 메서드 이름을 이스케이프해서 호출할 수 있다.

```
foo.`is`(bar)
```

Null-안전성과 플랫폼 타입

자바에서 모든 레퍼런스는 null일 수 있는데 이는 자바에서 오는 객체는 코틀린의 엄격한 null-안전성을 쓸모없게 만든다. 코틀린은 자바 선언 타입을 플랫폼 타입으로 별도 처리한다. 이 타입에 대해서는 null-검사를 완화해서 그 타입에 대한 안전 보장 수준을 자바 정도로 맞춘다(아래 참고).

다음 예를 보자.

```
val list = ArrayList<String>() // non-null (생성자 결과)
list.add("Item")
val size = list.size() // non-null (int 타입)
val item = list[0] // 플랫폼 타입 유추 (일반 자바 객체)
```

플랫폼 타입 변수의 메서드를 호출하면 코틀린은 컴파일 시점에서 null 가능성 에러를 발생하지 않는다. 하지만 널포인트 익셉션이니 코틀린이 null 전파를 막기 위해 생성하는 assertion 때문에 런타임에 메서드 호출에 실패할 수 있다.

```
item.substring(1) // 컴파일은 허용, item == null 이면 익셉션이 발생할 수 있음
```

플랫폼 타입은 non-denotable로 이는 언어에서 직접 그 타입을 쓸 수 없음을 의미한다. 플랫폼 값을 코틀린 변수에 할당할 때, 타입 추론에 기대거나(위 예제에서 item 처럼 변수는 추론한 플랫폼 타입을 갖는다) 원하는 타입을 지정할 수 있다(둘 다 nullable과 non-null 타입을 허용한다):

```
val nullable: String? = item // 허용, 항상 동작
val notNull: String = item // 허용, 런타임에 실패할 수 있음
```

non-null 타입을 선택하면 컴파일러가 할당 시점에 assertion을 발생할 수 있다. 이는 코틀린 non-null 변수가 null을 갖는 것을 막아준다. 또한 non-null 값을 기대하는 코틀린 함수에 플랫폼 값을 전달하면 assertion을 발생한다. 이를 종합하면 컴파일러는 null 이 프로그램에 전파되는 것을 최대한 막는다(지네릭때문에 완전히 막지는 못한다).

플랫폼 타입을 위한 기호

위에서 말한 것처럼 프로그램에서 플랫폼 타입을 직접 지정할 수 없기에 언어에 플랫폼 타입을 위한 구문이 없다. 그럼에도 불구하고 컴파일러와 IDE는 플랫폼 타입을 표현해야 할 필요가 있기에(에러 메시지나 파라미터 정보 등) 다음 기호를 사용한다.

- T! 는 “T 또는 T?”를 의미한다,
- (Mutable)Collection<T>! 는 “T의 자바 컬렉션은 불변이거나 아닐 수 있고, nullable이거나 아닐 수 있다”를 의미한다,

`Array<(out) T>!` 는 “ T (또는 하위 타입)의 배열은 nullable이거나 아니다”를 의미한다,

Nullability 애노테이션

nullability 애노테이션을 가진 자바 타입은 플랫폼 타입이 아닌 실제 nullable이나 non-null 코틀린 타입으로 표현한다. 현재 컴파일러는 [JetBrains의 nullability 애노테이션](#)(`org.jetbrains.annotations` 패키지의 `@Nullable` 과 `@NotNull`)을 지원한다.

매핑한 타입

코틀린은 자바 타입을 특별하게 처리한다. 자바 타입을 그대로 로딩하지 않고 해당하는 코틀린 타입으로 매핑한다. 매핑은 컴파일 타임에만 일어나며 런타임 표현은 그대로 유지된다. 자바의 기본 데이터 타입은 해당하는 코틀린 타입으로 매핑된다([플랫폼 타입](#)을 유념한다).

자바 타입	코틀린 타입
byte	kotlin.Byte
short	kotlin.Short
int	kotlin.Int
long	kotlin.Long
char	kotlin.Char
float	kotlin.Float
double	kotlin.Double
boolean	kotlin.Boolean

기본 타입이 아닌 일부 내장 클래스도 매핑한다.

자바 타입	코틀린 타입
java.lang.Object	kotlin.Any!
java.lang.Cloneable	kotlin.Cloneable!
java.lang.Comparable	kotlin.Comparable!
java.lang.Enum	kotlin.Enum!
java.lang.Annotation	kotlin.Annotation!
java.lang.Deprecated	kotlin.Deprecated!
java.lang.Void	kotlin.Nothing!
java.lang.CharSequence	kotlin.CharSequence!
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!

코틀린에서 컬렉션 타입은 읽기 전용이거나 변경 가능할 수 있으므로 다음과 같이 자바 컬렉션을 매핑한다(이 표의 모든 코틀린 타입은 `kotlin` 패키지에 위치한다):

자바 타입	코틀린 읽기 전용 타입	코틀린 수정 가능 타입	로딩한 플랫폼 타입
Iterator<T>	Iterator<T>	MutableIterator<T>	(Mutable)Iterator<T>!

자바 타입	표현할 자바 타입	코틀린 수정 가능 타입	보장할 플랫폼 타입
Iterable<T> Collection<T>	Iterable<T> Collection<T>	MutableIterable<T> MutableCollection<T>	(Mutable)Iterable<T>! (Mutable)Collection<T>!
Set<T>	Set<T>	MutableSet<T>	(Mutable)Set<T>!
List<T>	List<T>	MutableList<T>	(Mutable)List<T>!
ListIterator<T>	ListIterator<T>	MutableListIterator<T>	(Mutable)ListIterator<T>!
Map<K, V>	Map<K, V>	MutableMap<K, V>	(Mutable)Map<K, V>!
Map.Entry<K, V>	Map.Entry<K, V>	MutableMap.MutableEntry<K, V>	(Mutable)Map. (Mutable)Entry<K, V>!

자바 배열은 [아래](#) 언급한 것처럼 매핑한다.

자바 타입	코틀린 타입
int[]	kotlin.IntArray!
String[]	kotlin.Array<(out) String>!

코틀린에서 자바 지네릭

코틀린의 지네릭은 자바와 약간 다르다([지네릭](#) 참고). 자바 타입을 코틀린에 임포트할 때 일부 변환이 발생한다.

- 자바 와일드 카드를 타입 프로젝션으로 변환
 - `Foo<? extends Bar>` 는 `Foo<out Bar!>!` 이 된다.
 - `Foo<? super Bar>` 는 `Foo<in Bar!>!` 이 된다.
- 자바의 raw 타입을 스타-프로젝션으로 변환
 - `List` 는 `List<*>!` , 즉 `List<out Any?>!` 이 된다.

자바처럼 코틀린도 런타임에 지네릭을 유지하지 않으므로 객체는 생성자에 전달한 실제 타입 파라미터에 대한 정보를 갖지 않는다. 즉 `ArrayList<Integer>()` 와 `ArrayList<Character>()` 는 구분되지 않는다. 이는 지네릭을 `is`-검사에 사용할 수 없게 만든다. 코틀린은 스타-프로젝션 지네릭 타입에 대한 `is`-검사만 허용한다.

```
if (a is List<Int>) // 에러: 실제 Int의 List인지 검사할 수 없다
// but
if (a is List<*>) // OK: List의 내용에 대해 보장하지 않는다
```

자바 배열

코틀린 배열은 자바와 달리 무공변(invariant)하다. 이는 코틀린에서 `Array<Any>` 에 `Array<String>` 를 할당할 수 없음을 의미하며, 가능한 런타임 실패를 막아준다. 또한 하위클래스 배열을 코틀린 메서드의 상위클래스 배열 파라미터에 전달하는 것도 막아준다. 자바 메서드는 `(Array<(out) String>!` 형식의 [플랫폼 타입](#)) 이를 허용한다.

자바 플랫폼에서는 박싱/언박싱 비용을 없애기 위해 배열에 기본 데이터타입을 사용한다. 코틀린은 이런 구현 상세를 감추므로, 자바 코드를 사용하려면 우회방법이 필요하다. 이를 위해 모든 기본 데이터타입의 배열을 위한 별도 클래스(`IntArray` , `DoubleArray` , `CharArray` 등)를 제공한다. 이 클래스는 `Array` 클래스와는 관련이 없으며 최대 성능을 위해 자바의 기본 데이터타입 배열로 컴파일된다.

int 배열을 받는 자바 메서드가 있다고 가정하자:

```
public class JavaArrayExample {

    public void removeIndices(int[] indices) {
        // code here...
    }
}
```

코틀린에서 기본 데이터타입의 배열을 전달하려면 다음과 같이 할 수 있다.

```
val javaObj = JavaArrayExample()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndices(array) // int[]를 메서드에 전달
```

JVM 바이트 코드로 컴파일할 때 컴파일러는 배열 접근을 최적화해서 추가 오버헤드를 없앤다.

```
val array = arrayOf(1, 2, 3, 4)
array[x] = array[x] * 2 // get()과 set()을 호출하지 않음
for (x in array) // Iterator를 생성하지 않음
    print(x)
```

인덱스로 접근할 때도 오버헤드가 발생하지 않는다.

```
for (i in array.indices) // Iterator를 생성하지 않음
    array[i] += 2
```

마지막으로 `in`-검사도 오버헤드가 없다.

```
if (i in array.indices) { // (i >= 0 && i < array.size)와 동일
    print(array[i])
}
```

자바 가변인자

가변 개수의 인자를 갖는 메서드를 사용하는 자바 클래스가 종종 있다.

```
public class JavaArrayExample {

    public void removeIndices(int... indices) {
        // code here...
    }
}
```

이 경우 `IntArray` 를 파라미터로 전달하려면 `spread` 연산자인 `*` 를 사용해야 한다.

```
val javaObj = JavaArray()
val array = intArrayOf(0, 1, 2, 3)
javaObj.removeIndicesVarArg(*array)
```

현재는 가변 인자 메서드에 `null`을 전달할 수 없다.

연산자

자바는 메서드를 연산자 구문으로 사용할 수 있는 방법이 없기 때문에, 코틀린은 올바른 이름과 시그니처를 가진 모든 자바 메서드를 연산자 오버로딩과 다른 규칙(`invoke()` 등)으로 사용할 수 있도록 한다. 자바 메서드를 중위 호출 구문을 이용해서 호출하는 것을 허용하지 않는다.

체크드 익셉션

코틀린의 모든 익셉션은 언체크드인데 이는 컴파일러가 익셉션 `catch`를 강제하지 않는다는 것을 의미한다. 따라서 체크드 익셉션을 선언한 자바 메서드를 호출할 때 코틀린은 익셉션 `catch`를 강제하지 않는다.

```
fun render(list: List<*>, to: Appendable) {
    for (item in list)
        to.append(item.toString()) // 자바는 여기에 IOException catch를 요구한다
}
```

Object 메서드

자바 타입을 코틀린에 임포트할 때 `java.lang.Object` 타입의 모든 레퍼런스는 `Any` 로 바뀐다. `Any` 는 플랫폼에 특화되어 있지 않기 때문에 멤버로 `toString()`, `hashCode()`, `equals()` 만 선언하고 있다. 따라서 `java.lang.Object` 의 다른 멤버를 사용할 수 있도록 코틀린은 [확장 함수](#)를 사용한다.

wait()/notify()

[Effective Java](#) Item 69는 `wait()` 와 `notify()` 보다 병렬 유틸리티를 쓰라고 제안하고 있다. `Any` 타입 레퍼런스에는 이 메서드를 사용할 수 없다. 만약 그 메서드를 실제로 호출해야 한다면 `java.lang.Object` 로 변환하면 된다.

```
(foo as java.lang.Object).wait()
```

getClass()

객체의 타입 정보를 구하려면 `javaClass` 확장 프로퍼티를 사용한다.

```
val fooClass = foo.javaClass
```

자바의 `Foo.class` 대신에 `Foo::class.java`를 사용한다.

```
val fooClass = Foo::class.java
```

clone()

`clone()` 을 오버라이드하려면, `kotlin.Cloneable` 을 확장해야 한다.

```
class Example : Cloneable {
    override fun clone(): Any { ... }
}
```

[Effective Java](#), Item 11: *Override clone judiciously*를 잊지 말자.

finalize()

`finalize()` 를 오버라이드하려면 `override` 키워드를 사용하지 않고 단순히 메서드를 선언만 하면 된다.

```
class C {
    protected fun finalize() {
        // finalization logic
    }
}
```

자바 규칙에 따라 `finalize()` 는 `private`이면 안 된다.

자바 클래스를 상속하기

코틀린 클래스의 상위타입으로 최대 한 개의 자바 클래스를(자바 인터페이스는 여러 개) 사용할 수 있다.

정적 멤버 접근

자바 클래스의 정적 멤버는 이 클래스의 “컴페니언 오브젝트”를 만든다. 값으로 “컴페니언 오브젝트”를 전달할 수 없지만 멤버에 직접 접근할 수는 있다.

```
if (Character.isLetter(a)) {
    // ...
}
```

자바 리플렉션

자바 리플렉션은 코틀린 클래스에 동작하며 반대로도 된다. 위에서 말한 것처럼 `java.lang.Class` 로 자바 리플렉션을 사용하려면 `instance.javaClass` 나 `ClassName::class.java` 를 사용할 수 있다.

이 외에 코틀린 프로퍼티를 위한 자바 `getter/setter` 메서드나 `backing 필드` 구하기, 자바 필드를 위한 `KProperty` , `KFunction` 을 위한 자바 메서드나 생성자 구하기 그리고 그 반대 기능을 지원한다.

SAM 변환

자바 8과 같이 코틀린은 SAM 변환을 지원한다. 이는 코틀린 함수 리터럴을 (인터페이스 파라미터 타입이 코틀린 함수의 파라미터 타입에 매칭되는 함수) 자동으로 한 개의 non-default 메서드를 갖는 자바 인터페이스 구현으로 변경한다는 것을 의미한다.

SAM 인터페이스의 인스턴스를 생성하려면 SAM 변환을 사용할 수 있다.

```
val runnable = Runnable { println("This runs in a runnable") }
```

...메서드 호출에서 SAM 변환:

```
val executor = ThreadPoolExecutor()  
// Java signature: void execute(Runnable command)  
executor.execute { println("This runs in a thread pool") }
```

자바 클래스가 함수형 인터페이스를 받는 메서드를 여러 개 가지면 람다를 특정 SAM 타입으로 변환하는 어댑터 함수를 호출해서 사용할 메서드를 선택할 수 있다. 컴파일러는 필요한 곳에 이 어댑터 함수를 생성한다.

```
executor.execute(Runnable { println("This runs in a thread pool") })
```

SAM 변환은 인터페이스에만 적용되며, 추상 클래스의 경우 추상 메서드 한 개만 가진 경우라도 적용되지 않는다.

또한 이 기능은 자바 상호운용에 대해서만 동작한다. 코틀린은 적당한 함수 타입을 갖고 있기 때문에 함수를 코틀린 인터페이스 구현으로 자동 변환하는 기능은 필요 없고 따라서 지원하지 않는다.

코틀린에서 JNI 사용하기

네이티브 (C나 C++) 코드로 구현한 함수를 선언하려면 `external` 제한자를 사용하면 된다.

```
external fun foo(x: Int): Double
```

이를 뺀 나머지는 자바와 정확하게 같은 방식으로 동작한다.

자바에서 코틀린 실행하기

자바에서 쉽게 코틀린 코드를 실행할 수 있다.

프로퍼티

프로퍼티 `getter`는 `get`-메서드로 `setter`는 `set`-메서드로 바뀐다.

패키지 수준 함수

`example.kt` 파일의 `org.foo.bar` 패키지에 선언한 함수와 프로퍼티는 `org.foo.bar.ExampleKt` 라 불리는 자바 클래스에 위치한다.

```
// example.kt
package demo

class Foo

fun bar() {
}
```

```
// Java
new demo.Foo();
demo.ExampleKt.bar();
```

`@JvmName` 애노테이션을 사용하면 생성할 자바 클래스의 이름을 바꿀 수 있다.

```
@file:JvmName("DemoUtils")

package demo

class Foo

fun bar() {
}
```

```
// Java
new demo.Foo();
demo.DemoUtils.bar();
```

(같은 패키지나 같은 이름 또는 같은 `@JvmName` 애노테이션을 사용해서) 생성할 자바 클래스 이름이 같은 파일이 여러 개인 경우 보통 에러가 발생한다. 하지만 컴파일러는 같은 이름을(파일 이름이나 `@JvmName` 값) 갖는 모든 파일에 선언한 모든 것을 포함하는 단일 자바 파사드 클래스를 만드는 기능을 제공한다. 파사드 생성을 활성화하려면 모든 파일에 `@JvmMultifileClass` 애노테이션을 사용하면 된다.

```
// oldutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun foo() {
}
```

```
// newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass

package demo

fun bar() {
}
```

```
// Java
demo.Utils.foo();
demo.Utils.bar();
```

인스턴스 필드

코틀린 프로퍼티를 자바의 필드로 노출하고 싶다면 `@JvmField` 애노테이션을 프로퍼티에 적용해야 한다. 필드는 해당하는 프로퍼티와 같은 가시성을 갖는다. 프로퍼티가 backing 필드를 갖거나, 가시성이 `private`이 아니거나, `open`, `override` 또는 `const` 제한자를 갖지 않거나, 위임 프로퍼티가 아니면 프로퍼티에 `@JvmField` 를 붙일 수 없다.

```
class C(id: String) {
    @JvmField val ID = id
}
```

```
// Java
class JavaClient {
    public String getID(C c) {
        return c.ID;
    }
}
```

[초기화 지연](#) 프로퍼티도 필드로 노출된다. 필드는 `lateinit` 프로퍼티의 `setter`와 같은 가시성을 갖는다.

정적 필드

네임드 오브젝트나 컴패니언 오브젝트에 선언한 코틀린 프로퍼티는 네임드 오브젝트나 컴패니언 오브젝트를 포함하는 클래스에서 정적 backing 필드를 갖는다.

보통 이 필드는 `private`이지만 다음 중 한 가지 방법으로 노출할 수 있다.

- `@JvmField` 애노테이션
- `lateinit` 제한자
- `const` 제한자

`@JvmField` 을 붙인 프로퍼티는 프로퍼티 자체와 같은 가시성을 갖는 정적 필드가 된다.

```
class Key(val value: Int) {
    companion object {
        @JvmField
        val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value }
    }
}
```

```
// Java
Key.COMPARATOR.compare(key1, key2);
// Key 클래스에 public static final 필드
```

오브젝트나 컴패니언 오브젝트의 [초기화 지연](#) 프로퍼티는 프로퍼티 setter와 같은 가시성을 갖는 정적 backing 필드를 갖는다.

```
object Singleton {
    lateinit var provider: Provider
}
```

```
// Java
Singleton.provider = new Provider();
// public static non-final field in Singleton class
```

(최상위 수준 또는 클래스에 있는) `const` 프로퍼티는 자바에서 정적 필드가 된다.

```
// file example.kt

object Obj {
    const val CONST = 1
}

class C {
    companion object {
        const val VERSION = 9
    }
}

const val MAX = 239
```

자바:


```
int c = Obj.CONST;
int d = ExampleKt.MAX;
int v = C.VERSION;
```

정적 메서드

앞서 언급했듯이 패키지 수준 함수에 대해 코틀린은 정적 메서드를 생성한다. 또한 코틀린은 네임드 오브젝트나 컴패니언 오브젝트에 정의한 함수에 `@JvmStatic` 을 붙이면 정적 메서드를 생성한다. 다음 예를 보자.

```
class C {
    companion object {
        @JvmStatic fun foo() {}
        fun bar() {}
    }
}
```

여기서 `foo()` 는 자바에서 정적이지만 `bar()` 는 아니다.

```
C.foo(); // 잘 동작
C.bar(); // 에러: 정적 메서드 아님
```

네임드 오브젝트도 같다.

```
object Obj {
    @JvmStatic fun foo() {}
    fun bar() {}
}
```

자바:

```
Obj.foo(); // 잘 동작
Obj.bar(); // 에러
Obj.INSTANCE.bar(); // 싱글톤 인스턴스를 통해서 호출
Obj.INSTANCE.foo(); // 역시 동작
```

`@JvmStatic` 애노테이션을 오브젝트 프로퍼티나 컴패니언 오브젝트에 적용할 수 있다. 이는 `getter`와 `setter` 메서드를 그 오브젝트나 컴패니언 오브젝트를 포함한 클래스의 정적 메서드로 만든다.

@JvmName으로 시그니처 충돌 처리하기

때때로 코틀린에서 네임드 함수가 JVM 바이트 코드 상의 이름과 달라야 할 때가 있다. 가장 두드러진 예는 *타입 제거(type erasure)* 때문에 발생한다.

```
fun List<String>.filterValid(): List<String>
fun List<Int>.filterValid(): List<Int>
```

이 두 함수는 함께 정의할 수 없다. 왜냐하면 JVM 시그니처가 `filterValid(Ljava/util/List;)Ljava/util/List;` 로 같기 때문이다. 코틀린에서 같은 이름을 갖는 함수를 정의하고 싶다면 두 함수 중 하나에 (또는 두 함수 모두에) `@JvmName` 의 인자로 다른 이름을 지정해야 한다.

```
fun List<String>.filterValid(): List<String>

@JvmName("filterValidInt")
fun List<Int>.filterValid(): List<Int>
```

코틀린에서는 같은 이름인 `filterValid` 로 접근할 수 있지만 자바에서는 이름이 `filterValid` 와 `filterValidInt` 가 된다.

함수 `getX()` 를 갖는 이름이 `x` 인 프로퍼티에 대해서도 같은 트릭을 사용할 수 있다.

```
val x: Int
    @JvmName("getX_prop")
    get() = 15

fun getX() = 10
```

오버로딩 생성

보통 기본 파라미터 값을 갖는 코틀린 메서드를 작성하면 자바에서는 모든 파라미터가 있는 전체 시그니처만 사용 가능하다. 만약 자바 코드에 오버로드한 여러 메서드를 노출하고 싶다면 `@JvmOverloads` 애노테이션을 사용할 수 있다.

```
@JvmOverloads fun f(a: String, b: Int = 0, c: String = "abc") {
    ...
}
```

기본 값을 가진 모든 파라미터에 대해 오버로드 메서드를 생성한다. 기본 값을 가진 파라미터와 그 오른쪽에 위치한 파라미터를 파라미터 목록에서 제거한다. 이 예는 다음 메서드를 생성한다.

```
// 자바
void f(String a, int b, String c) { }
void f(String a, int b) { }
void f(String a) { }
```

생성자와 정적 메서드에도 이 애노테이션을 적용할 수 있다. 인터페이스에 정의한 메서드를 포함해서 추상 메서드에는 사용할 수 없다.

[보조 생성자](#)에서 설명한 것처럼, 클래스가 모든 생성자 파라미터에 대해 기본 값을 가지면 인자 없는 `public` 생성자를 생성한다. 이는 `@JvmOverloads` 애노테이션을 지정하지 않아도 적용된다.

체크드 익셉션

코틀린에는 체크드 익셉션이 없다. 따라서 보통은 코틀린 함수의 자바 시그니처는 익셉션을 선언하지 않는다. 다음과 같은 코틀린 함수를 가졌다고 해보자:

```
// example.kt
package demo

fun foo() {
    throw IOException()
}
```

그리고 자바에서 위 함수를 호출할 때 익셉션을 catch하고 싶다고 하자.

```
// 자바
try {
    demo.Example.foo();
}
catch (IOException e) { // 예러: foo()는 throws 목록에 IOException을 선언하지 않았다
    // ...
}
```

foo() 가 IOException 을 선언하고 있지 않으므로 자바 컴파일러는 에러를 발생한다. 이 문제를 피하려면 코틀린 코드에 @Throws 애노테이션을 사용하면 된다.

```
@Throws(IOException::class)
fun foo() {
    throw IOException()
}
```

Null-안전성

자바에서 코틀린 함수를 호출할 때 non-null 파라미터에 null을 전달하는 것을 막을 수가 없다. 이런 이유로 코틀린은 non-null을 요구하는 모든 public 함수에 대해 런타임 검사를 추가한다. 이것이 자바 코드에서 즉시 NullPointerException 을 얻는 방법이다.

기변(Variant) 지네릭

코틀린이 [선언-위치 가변](#)을 사용하면, 자바 코드에서 이를 사용하는 두 가지 선택이 있다. 다음 클래스와 이 클래스를 사용하는 두 함수를 보자.

```
class Box<out T>(val value: T)

interface Base
class Derived : Base

fun boxDerived(value: Derived): Box<Derived> = Box(value)
fun unboxBase(box: Box<Base>): Base = box.value
```

이 함수를 자바로 변환하는 가장 단순한 방법은 다음과 같다.

```
Box<Derived> boxDerived(Derived value) { ... }
Base unboxBase(Box<Base> box) { ... }
```

문제는 코틀린에서는 `unboxBase(boxDerived("s"))` 가 가능하지만 자바에서는 가능하지 않다는 것이다. 왜냐면 자바에서 `Box` 클래스는 `T` 파라미터에 대해 *무공변(invariant)*이라서 `Box<Derived>` 가 `Box<Base>` 의 하위타입이 아니기 때문이다. 자바에서 이게 동작하도록 하려면 `unboxBase` 를 다음과 같이 정의해야 한다.

```
Base unboxBase(Box<? extends Base> box) { ... }
```

여기서 사용-위치 가변(use-site variance)을 이용해서 선언-위치 가변(declaration-site variance)을 흉내내기 위해 자바의 *와일드카드 타입(? extends Base)* 을 사용했다. 왜냐면 자바로는 이렇게밖에 못하기 때문이다.

코틀린 API를 자바에서 쓸 수 있게 하기 위해 파라미터로 쓰이는 `Box<Super>` 를 (공변으로 정의한 Box인) `Box<? extends Super>` 로 생성한다(또는 반공변으로 정의한 Foo인 `Foo<? super Bar>` 로 생성). 그것이 리턴 값이면 와일드카드를 생성하지 않는다. 그렇지 않을 경우 자바 클라이언트가 그것을 처리해야 하기 때문이다(그리고 이는 보통의 자바 코딩 방식과 반대된다). 따라서 위 예의 함수는 실제로 다음과 같이 번역된다.

```
// 리턴 타입 - 와일드카드 없음
Box<Derived> boxDerived(Derived value) { ... }

// 파라미터 - 와일드카드
Base unboxBase(Box<? extends Base> box) { ... }
```

노트: 인자 타입이 `final`이면, 보통 와일드카드를 생성해도 얻는게 없기 때문에 `Box<String>` 은 인자 위치에 상관없이 항상 `Box<String>` 이다.

기본적으로 와일드카드를 생성할 수 없는 곳에서 와일드카드가 필요하면 `@JvmWildcard` 애노테이션을 사용할 수 있다.

```
fun boxDerived(value: Derived): Box<@JvmWildcard Derived> = Box(value)
// 다음으로 번역
// Box<? extends Derived> boxDerived(Derived value) { ... }
```

반면에 와일드카드 생성이 필요 없으면 `@JvmSuppressWildcards` 를 사용한다.

```
fun unboxBase(box: Box<@JvmSuppressWildcards Base>): Base = box.value
// 다음으로 번역
// Base unboxBase(Box<Base> box) { ... }
```

노트: `@JvmSuppressWildcards` 을 개별 타입 인자에만 사용할 수 있는 것은 아니다. 함수나 클래스처럼 전체 선언에도 사용할 수 있다. 이는 선언 안의 모든 와일드카드를 제한한다.

Nothing 타입의 번역

`Nothing` 타입은 특별하다. 왜냐면 자바에 이와 딱들어맞는 요소가 없기 때문이다. 사실 `java.lang.Void` 를 포함한 모든 자바 레퍼런스 타입은 값으로 `null` 을 가질 수 있는데 `Nothing` 은 그것조차 안 된다. 따라서 자바에서 이 타입을 완벽하게 표현할 수는 없다. 이것이 코틀린이 `Nothing` 타입을 사용하는 인자에 raw 타입을 생성하는 이유이다.

```
fun emptyList(): List<Nothing> = listOf()
// is translated to
// List emptyList() { ... }
```

도구

코틀린 코드 문서화

코틀린 코드를 문서화하기 위해 사용하는 언어를 **KDoc**이라 한다. 이것의 핵심은 블록 태그를 위한 Javadoc 구문(코틀린에 맞는 요소를 지원하기 위해 확장)과 인라인 마크업을 위해 마크다운을 조합한 것이다.

문서 생성하기

[Dokka](#)라는 도구를 이용해서 코틀린 문서를 생성한다. 사용법은 [Dokka README](#) 문서를 참고한다.

Dokka는 그래들, 메이븐, 앤트 플러그인을 갖고 있어서 빌드 과정에 문서 생성을 통합할 수 있다.

KDoc 구문

JavaDoc처럼 KDoc 주석도 `/**`로 시작해서 `*/`로 끝난다. 모든 주석 줄은 애스터리크(*)를 포함할 수 있지만 애스터리크는 주석 내용에 포함되진 않는다.

보통 주석 내용의 첫 번째 단락(첫 번째 빈 행이 올 때까지)에는 요소의 요약 설명을 넣고 자세한 내용은 그 뒤에 넣는다.

모든 블록 태그는 새 줄에서 `@` 문자로 시작한다.

다음은 KDoc을 이용해서 클래스를 문서화한 예이다.

```
/**
 * A group of *members*.
 *
 * This class has no useful logic; it's just a documentation example.
 *
 * @param T the type of a member in this group.
 * @property name the name of this group.
 * @constructor Creates an empty group.
 */
class Group<T>(val name: String) {
    /**
     * Adds a [member] to this group.
     * @return the new size of the group.
     */
    fun add(member: T): Int { ... }
}
```

블록 태그

현재 KDoc은 다음 블록 태그를 지원한다.

`@param <name>`

함수의 파라미터나 클래스의 타입 파라미터 값을 문서화한다. 설명과 파라미터 이름을 더 잘 구분하고 싶다면 파라미터 이름을 대괄호로 둘러쌀 수 있다. 그래서 다음 두 구문은 동일하다.

`@param name description.`
`@param[name] description.`

`@return`

함수의 리턴 값을 문서화한다.

`@constructor`

클래스의 주요 생성자를 문서화한다.

`@property <name>`

지정한 이름을 가진 클래스의 프로퍼티를 문서화한다. 주요 생성자에 선언한 (프로퍼티 정의 앞에 주석을 직접 넣는게 어색한) 프로퍼티를 문서화하는데 이 태그를 사용할 수 있다.

`@throws <class>, @exception <class>`

메서드가 발생할 수 있는 익셉션을 문서화한다. 코틀린에는 체크드 익셉션이 없기 때문에 발생 가능한 모든 익셉션을 문서화할 것이라고 기대하진 않는다. 하지만 클래스 사용자에게 익셉션에 대한 정보가 유용하면 이 태그를 사용하면 된다.

`@sample <identifier>`

지정한 이름을 가진 함수의 몸체를 현재 요소의 문서화 결과에 삽입한다. 해당 요소를 어떻게 사용하는지 예를 보여주기 위한 용도로 사용한다.

`@see <identifier>`

문서의 **See Also** 블록에 지정한 클래스나 메서드에 대한 링크를 추가한다.

`@author`

문서화 대상 요소의 작성자를 지정한다.

`@since`

문서화 대상 요소를 추가한 소프트웨어 버전을 지정한다.

`@suppress`

문서 생성 대상에서 요소를 제외한다. 모듈의 공식 API에 포함은 되지 않지만 외부에 노출해야 하는 요소를 문서화할 때 사용할 수 있다.

⚠️ KDoc은 @deprecated를 지원하지 않는다. 대신 @Deprecated를 사용해야 한다.

인라인 마크업

인라인 마크업을 위해 KDoc은 정규 [Markdown](#) 구문을 사용하며 코드에 있는 다른 요소에 링크하기 위한 약식 구문 지원을 확장했다.

요소에 링크하기

다른 요소(클래스, 메서드, 프로퍼티 또는 파라미터)에 링크하려면 단순히 대괄호 안에 이름을 넣으면 된다.

Use the method [foo] for this purpose.

링크에 커스텀 라벨을 지정하고 싶다면 마크다운의 레퍼런스-스타일 구문을 사용한다.

Use [this method][foo] for this purpose.

링크에 전체 이름을 사용할 수도 있다. JavaDoc과 달리 전체 이름은 항상 컴포넌트를 구분할 때 점 문자를 사용해야 하며, 이는 메서드 이름 전이라도 마찬가지다.

Use [kotlin.reflect.KClass.properties] to enumerate the properties of the class.

링크에 있는 이름을 해석할 때에는 문서화할 대상 안에서 사용할 이름과 같은 규칙을 사용한다. 특히, 이는 현재 파일에 이름을 임포트하면 KDoc 주석에서 그 이름을 사용할 때 완전한 이름을 사용할 필요가 없다는 것을 뜻한다.

KDoc은 링크에서 오버로딩한 멤버를 찾기 위한 별도 구문을 제공하지 않는다. 코틀린 문서 생성 도구는 같은 페이지에 있는 모든 오버로딩 함수를 문서화에 넣기 때문에, 오버로딩한 함수 중에서 링크 적용 대상을 식별할 필요가 없다.

메이븐 사용하기

플러그인과 버전

*kotlin-maven-plugin*은 코틀린 소스와 모듈을 컴파일한다. 현재는 메이븐 v3만 지원한다.

*kotlin.version*으로 사용할 코틀린 버전을 정의한다. 코틀린 릴리즈와 버전 간의 관계를 아래 표시켰다.

Milestone	Version
1.0.1 hotfix update 2	1.0.1-2
1.0.1 hotfix update	1.0.1-1
1.0.1	1.0.1
1.0 GA	1.0.0
Release Candidate	1.0.0-rc-1036
Beta 4	1.0.0-beta-4589
Beta 3	1.0.0-beta-3595
Beta 2	1.0.0-beta-2423
Beta	1.0.0-beta-1103
Beta Candidate	1.0.0-beta-1038
M14	0.14.449
M13	0.13.1514
M12.1	0.12.613
M12	0.12.200
M11.1	0.11.91.1
M11	0.11.91
M10.1	0.10.195
M10	0.10.4
M9	0.9.66
M8	0.8.11
M7	0.7.270
M6.2	0.6.1673
M6.1	0.6.602
M6	0.6.69
M5.3	0.5.998

의존

코틀린은 애플리케이션이 사용할 수 있는 방대한 표준 라이브러리를 갖고 있다. *pom* 파일에 다음 의존을 설정한다.


```

<dependencies>
  <dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-stdlib</artifactId>
    <version>${kotlin.version}</version>
  </dependency>
</dependencies>

```

코틀린 소스 코드만 컴파일하기

소스 코드를 컴파일하려면 태그에 소스 디렉토리를 지정한다.

```

<sourceDirectory>${project.basedir}/src/main/kotlin</sourceDirectory>
<testSourceDirectory>${project.basedir}/src/test/kotlin</testSourceDirectory>

```

코틀린 메이븐 플러그인에 소스 컴파일을 참조해야 한다.

```

<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>

  <executions>
    <execution>
      <id>compile</id>
      <goals> <goal>compile</goal> </goals>
    </execution>

    <execution>
      <id>test-compile</id>
      <goals> <goal>test-compile</goal> </goals>
    </execution>
  </executions>
</plugin>

```

코틀린과 자바 소스 컴파일하기

자바와 코틀린을 함께 사용하는 애플리케이션 코드를 컴파일하려면 자바 컴파일러 전에 코틀린 컴파일을 실행해야 한다. 메이븐에서는 maven-compiler-plugin 전에 kotlin-maven-plugin를 실행해야 함을 의미한다.

코틀린 컴파일을 실행하려면 코틀린 컴파일 과정을 이전 단계인 process-sources로 옮기면 된다(더 나은 방법을 알고 있다면 자유롭게 제안해 달라):

```
<plugin>
  <artifactId>kotlin-maven-plugin</artifactId>
  <groupId>org.jetbrains.kotlin</groupId>
  <version>${kotlin.version}</version>

  <executions>
    <execution>
      <id>compile</id>
      <phase>process-sources</phase>
      <goals> <goal>compile</goal> </goals>
    </execution>

    <execution>
      <id>test-compile</id>
      <phase>process-test-sources</phase>
      <goals> <goal>test-compile</goal> </goals>
    </execution>
  </executions>
</plugin>
```

OSGi

OSGi에 대한 지원은 [Kotlin OSGi 페이지](#)를 참고한다.

예제

모든 메이븐 프로젝트 예제는 [GitHub 리포지토리에 직접 다운로드](#)할 수 있다.

앤티 사용하기

앤티 태스크 얻기

코틀린은 세 개의 앤티 태스크를 제공한다.

- `kotlinc`: JVM 대상 한 코틀린 컴파일러
- `kotlin2js`: 자바스크립트 대상 코틀린 컴파일러
- `withKotlin`: 표준 `javac` 앤티 태스크를 사용할 때 코틀린 파일을 컴파일하는 태스크

[코틀린 컴파일러](#)의 `lib` 폴더에 위치한 `kotlin-ant.jar` 라이브러리에 이 태스크가 정의되어 있다.

JVM 대상으로 코틀린 소스만 컴파일하기

프로젝트가 코틀린 소스 코드로만 구성되어 있을 때 프로젝트를 컴파일하는 가장 쉬운 방법은 `kotlinc` 태스크를 사용하는 것이다.

```
<project name="Ant Task Test" default="build">
  <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
    classpath="${kotlin.lib}/kotlin-ant.jar"/>

  <target name="build">
    <kotlinc src="hello.kt" output="hello.jar"/>
  </target>
</project>
```

`${kotlin.lib}`는 코틀린 표준 컴파일러의 압축을 푼 폴더를 가리킨다.

JVM 대상으로 여러 루트를 갖는 코틀린 소스만 컴파일하기

프로젝트가 여러 소스 루트를 가지면 `src` 요소를 사용해서 경로를 정의한다.

```
<project name="Ant Task Test" default="build">
  <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
    classpath="${kotlin.lib}/kotlin-ant.jar"/>

  <target name="build">
    <kotlinc output="hello.jar">
      <src path="root1"/>
      <src path="root2"/>
    </kotlinc>
  </target>
</project>
```

JVM 대상으로 코틀린과 자바 소스가 함께 컴파일하기

프로젝트에 코틀린과 자바 소스 코드가 함께 있다면 `kotlinc`를 사용할 수도 있지만, 태스크 파라미터 중복을 피하기 위해 `withKotlin` 태스크를 사용할 것을 권한다.

```

<project name="Ant Task Test" default="build">
  <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
    classpath="${kotlin.lib}/kotlin-ant.jar"/>

  <target name="build">
    <delete dir="classes" failonerror="false"/>
    <mkdir dir="classes"/>
    <javac destdir="classes" includeAntRuntime="false" srcdir="src">
      <withKotlin/>
    </javac>
    <jar destfile="hello.jar">
      <fileset dir="classes"/>
    </jar>
  </target>
</project>

```

`<withKotlin>`에 추가 명령행 인자를 지정하려면 `<compilerArg>` 파라미터를 사용하면 된다. `kotlinc -help`로 사용할 수 있는 전체 인자 목록을 볼 수 있다. `moduleName` 애트리뷰트로 컴파일 할 모듈 이름을 지정할 수 있다.

```

<withKotlin moduleName="myModule">
  <compilerarg value="-no-stdlib"/>
</withKotlin>

```

자바 스크립트 대상으로 한 개 소스 폴더를 컴파일하기

```

<project name="Ant Task Test" default="build">
  <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
    classpath="${kotlin.lib}/kotlin-ant.jar"/>

  <target name="build">
    <kotlin2js src="root1" output="out.js"/>
  </target>
</project>

```

자바 스크립트 대상으로 prefix, postfix, 그리고 sourcemap 옵션 사용하기

```

<project name="Ant Task Test" default="build">
  <taskdef resource="org/jetbrains/kotlin/ant/antlib.xml"
    classpath="${kotlin.lib}/kotlin-ant.jar"/>

  <target name="build">
    <kotlin2js src="root1" output="out.js" outputPrefix="prefix"
      outputPostfix="postfix" sourcemap="true"/>
  </target>
</project>

```

자바스크립트 대상으로 한 개 소스 폴더와 meatInfo 옵션 사용하기

코틀린/자바스크립트 라이브러리로 변환 결과를 배포하길 원한다면 `metaInfo` 옵션을 유용하게 사용할 수 있다. `metaInfo` 옵션을 `true` 로 설정하면 컴파일할 때 바이너리 메타데이터를 가진 JS 파일을 추가로 생성한다. 변환 결과와 함께 이 파일을 배포해야 한다.

```
<project name="Ant Task Test" default="build">
  <typedef resource="org/jetbrains/kotlin/ant/antlib.xml"
  classpath="${kotlin.lib}/kotlin-ant.jar"/>

  <target name="build">
    <!-- 바이너리 디스크립터를 포함한 out.meta.js 파일을 생성한다. -->
    <kotlin2js src="root1" output="out.js" metaInfo="true"/>
  </target>
</project>
```

레퍼런스

요소와 애트리뷰트 전체 목록을 아래에 나열했다.

kotlinc와 kotlin2js의 위한 공통 애트리뷰트

이름	설명	필수	기본 값
src	컴파일할 코틀린 소스 파일 또는 디렉토리	Yes	
nowarn	모든 컴파일 경고를 무시함	No	false
noStdlib	클래스패스에 코틀린 표준 라이브러리를 포함하지 않음	No	false
failOnError	컴파일하는 동안 에러를 발견하면 빌드에 실패함	No	true

kotlinc 애트리뷰트

이름	설명	필수	기본 값
output	대상 디렉토리나 jar 파일 이름	Yes	
classpath	컴파일 클래스패스	No	
classpathref	컴파일 클래스패스 참조	No	
includeRuntime	output이 jar 파일일 때 코틀린 런타임 라이브러리를 jar에 포함할지 여부를 지정	No	true
moduleName	컴파일 할 모듈 이름	No	(대상을 지정했다면) 대상 이름 아니면 프로젝트 이름

kotlin2js 애트리뷰트

이름	설명	필수
output	대상 파일	Yes
library	라이브러리 파일 (kt, dir, jar)	No
outputPrefix	생성할 자바스크립트 파일에 사용할 접두사	No
outputSuffix	생성할 자바스크립트 파일에 사용할 접미사	No
sourcemap	sourcemap 파일을 생성할지 여부	No
metaInfo	바이너리 디스크립터를 가진 메타데이터 파일을 생성할지 여부	No

이름	설명	별명
----	----	----

일러가 생성한 코드가 메인 함수를 호출하는지

그래들 사용하기

그래들에서 코틀린을 빌드하려면 [kotlin-gradle 플러그인을 설정](#)하고 프로젝트에 [플러그인을 적용](#)하고 [kotlin-stdlib 의존을 추가](#)해야 한다. IntelliJ IDEA에서는 Tools

Kotlin Configure Kotlin를 사용하면 자동으로 처리한다.

플러그인과 버전

*kotlin-gradle-plugin*은 코틀린 소스와 모듈을 컴파일한다.

보통 사용할 코틀린 버전은 *kotlin_version* 프로퍼티로 정의한다.

```
buildscript {
    ext.kotlin_version = '<version to use>'

    repositories {
        mavenCentral()
    }

    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

코틀린 릴리즈와 버전 간의 관계를 아래 표시했다.

Milestone	Version
1.0.1 hotfix update 2	1.0.1-2
1.0.1 hotfix update	1.0.1-1
1.0.1	1.0.1
1.0 GA	1.0.0
Release Candidate	1.0.0-rc-1036
Beta 4	1.0.0-beta-4589
Beta 3	1.0.0-beta-3595
Beta 2	1.0.0-beta-2423
Beta	1.0.0-beta-1103
Beta Candidate	1.0.0-beta-1038
M14	0.14.449
M13	0.13.1514
M12.1	0.12.613
M12	0.12.200
M11.1	0.11.91.1
M11	0.11.91
M10.1	0.10.195
M10	0.10.4
M9	0.9.66
M8	0.8.11

Milestone	Version
M6.2	0.6.1673
M6.1	0.6.602
M6	0.6.69
M5.3	0.5.998

JVM 대상

JVM을 대상으로 하려면 kotlin 플러그인을 적용하면 된다.

```
apply plugin: "kotlin"
```

코틀린 소스와 자바 소스를 같은 폴더나 다른 폴더에 위치시킬 수 있다. 기본 규칙은 다른 폴더를 사용하는 것이다.

```
project
- src
  - main (root)
    - kotlin
    - java
```

기본 규칙을 사용하지 않을 경우 해당하는 *sourceSets* 프로퍼티를 수정해야 한다.

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
    main.java.srcDirs += 'src/main/myJava'
}
```

자바스크립트 대상

자바 스크립트가 대상이면 다른 플러그인을 사용한다.

```
apply plugin: "kotlin2js"
```

이 플러그인은 코틀린 파일만 처리하므로 (동일 프로젝트에 자바 파일을 포함하고 있다면) 코틀린과 자바 파일을 별도로 구분하는게 좋다. JVM도 대상으로 하면서 기본 규칙을 사용하지 않으면, *sourceSets*으로 소스 폴더를 지정해야 한다.

```
sourceSets {
    main.kotlin.srcDirs += 'src/main/myKotlin'
}
```

재사용 가능한 라이브러리를 만드려면 바이너리 디스크립터를 가진 JS 파일을 추가로 생성하기 위해 *kotlinOptions.metaInfo* 를 사용한다. 이 파일을 변환 결과와 함께 배포해야 한다.

```
compileKotlin2Js {
    kotlinOptions.metaInfo = true
}
```


안드로이드 대상

안드로이드의 그레들 모델은 일반적인 그레들과 약간 다르다. 따라서 코틀린으로 작성한 안드로이드 프로젝트를 빌드하려면 *kotlin* 대신 *kotlin-android* 플러그인을 사용해야 한다.

```
buildscript {  
    ...  
}  
apply plugin: 'com.android.application'  
apply plugin: 'kotlin-android'
```

안드로이드 스튜디오

안드로이드 스튜디오를 사용하면 android에 다음 코드를 추가해야 한다.

```
android {  
    ...  
  
    sourceSets {  
        main.java.srcDirs += 'src/main/kotlin'  
    }  
}
```

이 설정은 안드로이드 스튜디오가 코틀린 디렉토리를 소스 루트로 사용하게 한다. 따라서 프로젝트 모델을 IDE에 로딩할 때 올바르게 인식한다.

의존 설정

kotlin-gradle-plugin 의존과 함께 코틀린 표준 라이브러리에 대한 의존을 추가해야 한다.

```
buildscript {  
    ext.kotlin_version = '<사용할 버전>  
    repositories {  
        mavenCentral()  
    }  
    dependencies {  
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"  
    }  
}  
  
apply plugin: "kotlin" // 또는 자바 스크립트가 대상이면 apply plugin: "kotlin2js"  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"  
}
```

프로젝트에서 코틀린 리플렉션이나 테스트 기능을 사용하려면 해당 의존을 추가로 넣는다.

```
compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"
testCompile "org.jetbrains.kotlin:kotlin-test:$kotlin_version"
```

OSGi

OSGi 지원은 [Kotlin OSGi 페이지](#)를 참고한다.

예제

[코틀린 리포지토리](#)에 있는 예제:

- [코틀린](#)
- [자바와 코틀린 함께 사용](#)
- [안드로이드](#)
- [자바스크립트](#)

코틀린과 OSGi

코틀린 OSGi 지원 기능을 활성화하려면 일반 코틀린 라이브러리 대신 `kotlin-osgi-bundle` 를 포함하면 된다. `kotlin-osgi-bundle` 가 `kotlin-runtime`, `kotlin-stdlib` 그리고 `kotlin-reflect` 를 모두 포함하고 있으므로 이 세 의존을 제거한다. 또한 외부 코틀린 라이브러리를 포함한 경우 주의해야 한다. 대부분 일반 코틀린 의존은 OSGi를 지원하지 않으므로 사용하면 안 되고 프로젝트에서 제거해야 한다.

메이븐

메이븐 프로젝트에 코틀린 OSGi 번들을 포함하기:

```
<dependencies>
  <dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-osgi-bundle</artifactId>
    <version>${kotlin.version}</version>
  </dependency>
</dependencies>
```

외부 라이브러리에서 표준 라이브러리 제외하기(“* 제외”는 메이븐 3에서만 동작한다).

```
<dependency>
  <groupId>some.group.id</groupId>
  <artifactId>some.library</artifactId>
  <version>some.library.version</version>

  <exclusions>
    <exclusion>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>*</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Gradle

그레들 프로젝트에 `kotlin-osgi-bundle` 포함하기:

```
compile "org.jetbrains.kotlin:kotlin-osgi-bundle:$kotlinVersion"
```

의존성 전이로 포함되는 기본 코틀린 라이브러리를 제외하려면 다음 방법을 사용한다.

```
dependencies {
  compile (
    [group: 'some.group.id', name: 'some.library', version: 'someversion'],
    .....) {
    exclude group: 'org.jetbrains.kotlin'
  }
}
```

FAQ

왜 모든 코틀린 라이브러리에 필요한 매니페스트 옵션을 넣지 않았나?

매니페스트 옵션이 OSGi를 지원하는 가장 선호하는 방법이긴 하지만, 아쉽게도 “[패키지 분리](#)” 문제라 불리는 문제 때문에 할 수 없다. 이 문제는 쉽게 제거할 수 없고 아직 그렇게 큰 변경을 할 계획이 없다. `Require-Bundle` 피처가 있지만 이 역시 최고의 선택은 아니며 사용을 추천하지 않는다. 그래서 SOGi를 위한 별도 아티팩트를 만들기로 결정했다.

FAQ

FAQ

일반 질문

코틀린이 무엇인가?

코틀린은 정적 타입 언어로 JVM과 자바스크립트를 대상으로 한다. 산업에서 사용할 목적으로 만든 범용 언어이다.

코틀린은 OSS 언어이고 (JetBrains 소속이 아닌) 외부 기여자가 있지만, JetBrains에 있는 팀에서 만들고 있다.

왜 새로운 언어를 만들었나?

JetBrains에서 오랫동안 자바 플랫폼으로 개발을 진행했고 자바가 얼마나 좋은지 알고 있다. 하지만 자바 언어에는 일부 한계가 있고 하위호환 이슈 때문에 불가능하거나 고치기 힘든 문제가 있다는 것도 알고 있다. 자바가 오래 갈 거라는 건 알고 있다. 그래도 커뮤니티는 레거시 문제에서 벗어나 개발자가 절실하게 원하는 기능을 갖고 있는 JVM 대상 정적 타입 언어의 장점을 누려야 한다고 믿고 있다.

이 프로젝트의 주요 설계 목적은 다음과 같다.

- 자바와 호환되는 언어 만들기,
- 적어도 자바만큼 빠르게 컴파일해야 함,
- 자바보다 안전해야 함, 예를 들어 null pointer 값 참조와 같은 일반적인 문제를 정적으로 검사함,
- 변수 타입 추론, 고차 함수(클로저), 확장 함수, 믹스인, 일급 위임과 같은 걸 지원해서 자바보다 더 간결해야 함;
- 그리고 스칼라와 같은 가장 성숙한(mature) 경쟁 언어보다 방법을 더 간단하게 (위에서 언급한) 유용한 수준의 표현력을 유지하게 만든다.

라이선스는 어떻게 되나?

Kotlin은 OSS 언어로 Apache 2 OSS 라이선스를 따른다. IntelliJ 플러그인도 OSS이다.

GitHub에 호스팅하고 있으며 기여자를 행복하게 받아들이고 있다.

자바와 호환되나?

물론이다. 컴파일러는 자바 바이트 코드를 생성한다. 코틀린에서 자바를 호출할 수 있고 자바에서 코틀린을 호출할 수 있다. [자바 상 호운용](#)을 참고한다.

코틀린 코드를 실행하는데 필요한 최소 자바 버전은 어떻게 되나?

코틀린은 자바 6 또는 상위 버전과 호환되는 바이트코드를 생성한다. 안드로이드처럼 최근 버전이 자바 6을 지원하는 환경에서 코틀린을 사용할 수 있다.

지원하는 도구는 있나?

있다. Apache 2 라이선스 기반 OSS 프로젝트로 사용 가능한 IntelliJ IDEA 플러그인이 있다. IntelliJ IDEA의 [free OSS Community Edition](#)과 [Ultimate Edition](#) 두 버전에서 코틀린을 사용할 수 있다.

이클립스는 지원하나?

지원한다. [튜토리얼](#)에서 설치 과정을 참고한다.

단독 컴파일러가 있나?

있다. [GitHub 릴리즈 페이지](#)에서 단독 컴파일러와 다른 빌드 도구를 다운로드 할 수 있다.

코틀린은 함수형 언어인가?

코틀린은 객체 지향 언어이다. 추가로 고차 함수나 랴다 식 그리고 최상위 함수를 지원한다. 표준 코틀린 라이브러리에는 (map, flatMap, reduce 등) 함수형 언어에서 일반적인 요소가 상당히 많이 존재한다. 또한 함수형 언어가 무엇이고 코틀린은 그게 아니라고 할 만한 명확한 정의가 없다.

코틀린은 지네릭을 지원하나?

코틀린은 지네릭을 지원한다. 또한, 선언-위치 가변(declaration-site variance)과 사용-위치 가변(usage-site variance)을 지원한다. 코틀린에는 또한 와일드카드 타입이 없다. 인라인 함수는 reified 타입 파라미터를 지원한다.

세미콜린이 필요한가?

필요없다. 선택적으로 사용할 수 있다.

종괄호가 필요한가?

필요하다.

왜 타입 선언이 우측에 위치하나?

이 방식이 코드 가독성을 높여준다고 믿는다. 뿐만 아니라 일부 좋은 구문 기능을 가능하게 해준다. 예를 들어, 타입 애노테이션을 쉽게 뺄 수 있다(?). 스칼라가 이것이 문제가 없음을 거의 보여줬다.

우측에 위치한 타입 선언이 도구에 영향을 미치나?

영향 없다. 변수 이름 추천과 같은 기능을 여전히 구현할 수 있다.

코틀린은 확장 가능한가?

인라인 함수에서 애노테이션과 타입 로더까지 몇 가지 방법으로 코틀린을 확장할 계획이다.

DSL을 언어에 삽입할 수 있나?

있다. 코틀린은 이를 위해 연산자 오버로딩, 인라인 함수를 통한 커스텀 컨트롤 구조, 중위 함수 호출, 확장 함수, 애노테이션, 언어 인용과 같은 기능을 제공한다.

자바스크립트 몇 레벨의 ECMAScript를 지원하나?

현재 5 레벨을 지원한다.

자바 스크립트 백엔드는 모듈 시스템을 지원하나?

지원한다. CommonJS와 AMD를 지원할 계획이다.

자바와 비교

코틀린에 있는 몇 가지 자바 이슈

코틀린은 자바가 겪고 있는 여러 문제를 해결한다.

- [타입 시스템에서 null 참조를 제어](#)한다
- [raw 타입 없음](#)
- 코틀린의 배열은 [invariant](#)하다
- 자바의 SAM-변환과 달리 코틀린은 올바른 [function types](#)을 갖는다
- 와일드카드 없는 [Use-site variance](#)
- 코틀린은 체크드 [익셉션](#)이 없다

코틀린에 없고 자바에 있는 것

- [체크드 익셉션](#)
- 클래스가 아닌 [기본 타입](#)
- [정적 멤버](#)
- [비-private 필드](#)
- [와일드카드-타입](#)

자바에 없고 코틀린에 있는 것

- [람다 식](#) + [안리안 함수](#) = 훌륭한 커스텀 제어 구조
- [확장 함수](#)
- [Null-안전성](#)
- [스마트 타입변환](#)
- [문자열 템플릿](#)
- [프로퍼티](#)
- [주요 생성자](#)
- [필드-클래스 위임](#)
- [변수와 프로퍼티 타입을 위한 타입 추론](#)
- [싱글톤](#)
- [Declaration-site variance & Type projections](#)
- [Range 식](#)
- [연산자 오버로딩](#)
- [컴페니언 오브젝트](#)
- [데이터 클래스](#)
- [읽기 전용 컬렉션과 변경 가능 컬렉션 인터페이스 분리](#)

스칼라와 비교

코틀린 팀의 주요 목적은 실용적이고 생산적인 프로그래밍 언어를 만드는 것이다. 프로그래밍 언어 연구 용으로 앞서가는 최신 언어를 만드는 건 목적이 아니다. 이를 고려해서, 만약 스칼라에 만족한다면, 아마도 코틀린이 필요하지 않을 것이다.

코틀린에 없고 스칼라에 있는 것

- 암묵적 변환, 파라미터, etc
 - 스칼라는 때때로 디버거 없이 코드에서 무슨 일이 벌어지고 있는지 말하기 어렵다. 왜냐면 동작하기 위해 암묵적으로 너무 많은 것이 일어나기 때문이다.
 - 코틀린에서 타입에 함수를 추가하고 싶으면 [확장 함수](#)를 사용한다.
- 오버라이딩할 수 있는 타입 멤버
- 경로-의존 타입(Path-dependent types)
- 매크로
- Existential types
 - [타입 프로젝션](#)은 매우 특수한 경우이다
- 트레이트 초기화를 위한 복잡한 로직
 - [클래스와 상속](#) 참고
- 커스텀 오퍼레이션 심볼
 - [연산자 오버로딩](#) 참고
- XML 기본 지원
 - [Type-safe Groovy-style builders](#) 참고
- Structural types
- 값 타입(Value types)
 - [Project Valhalla](#)가 JDK에 일부로 릴리즈되면 지원할 계획이다.
- Yield 연산자
- 액터
 - JVM에 액터를 지원하기 위한 외부 프레임워크인 [Quasar](#)를 지원한다.
- 병렬 컬렉션
 - 코틀린은 유사한 기능을 제공하는 자바 8 스트림을 지원한다

스칼라에 없고 코틀린에 있는 것

- [오버헤드 없는 null-안정성](#)
 - 스칼라는 신택틱(syntactic)하고 런타임 래퍼인 Option을 갖는다.
- [스마트 변환](#)
- [코틀린의 인라인 함수는 non-로컬 점프를 쉽게할 수 있다](#)
- [일급 위인](#). 또한 외부 플러그인인 Autoproxy로 구현된다.
- [멤버 참조](#) (자바 8에서도 지원한다).

