# 4장. 클래스와 객체 다루기

이 장에서는 코틀린의 객체지향 프로그래밍을 맛보고 클래스를 사용해 우리들만의 타입을 정의하는 방법을 배운다. 클래스 인스턴스 초기화, 가시성을 사용해 구현 상세 사항을 감추는 법, 객체 선언을 통해 싱글턴을 구현하는 방법, 지연 계산, 지연 초기화, 커스텀 게터 세터를 통한 읽기 쓰기 기능 제어 등 여러 종류의 프로퍼티를 활용해 간단한 데이터 저장을 넘어서 다양한 기능을 제공하는 방법을 배운다. 이 장의 내용과 관련해 배워야 할 다른 내용으로는 타입의 널 가능성이 있다. 코틀린 컴파일러는 널이 될 수 있는 값과 그렇지 않은 값을 타입의 널 가능성을 활용해 구분한다.

## 구조

* 클래스 정의와 멤버
* 생성자
* 멤버 가시성
* 내포된 클래스와 지역 클래스
* 널이 될 수 있는 타입
* 단순하지 않은 프로퍼티를 사용하는 방법
* 객체와 동반객체

## 목표

이 장의 목표는 다음과 같다.

* 클래스와 객체를 사용하는 코틀린의 객체지향 프로그래밍의 기본을 익힌다.
* 널이 될 수 있는 값을 처리하는 법을 배운다.
* 여러 유형의 프로퍼티를 사용하는 방법을 이해한다.

## 클래스 정의하기

클래스 정의는 커스텀으로 정의된 연산들이 포함된 새로운 타입을 만들어준다. 자바나 C++ 같은 다른 객체 지향 프로그래밍 언어에 익숙하다면 클래스 정의도 익숙할 것이다. 이번 절에서는 기본적인 클래스 구조와 새로 할당된 인스턴스를 초기화하는 방법, 가시성 문제, 다른 클래스나 함수 본문 안에서 선언할 수 있는 특별한 유형의 클래스에 대해 살펴본다.

기본적으로 클래스 선언은 참조 타입(referential type)을 정의한다. 즉, 이런 참조 타입의 값은 특정 클래스 인스턴스(instance)의 실제 데이터 위치를 가리키는 참조다. 자바 인스턴스는 명시적으로 특별한 생성자 호출을 통해 생성되고, 프로그램 내에서 객체를 가리키는 모든 참조가 사라지면 가비지 컬렉터(garbage collector)에 의해 자동으로 해제된다. 코틀린 1.3부터 인라인 클래스(inline class)라는 개념이 도입됐다. 인라인 클래스를 사용하면 참조 타입이 아닌 타입을 정의할 수 있다. 이 주제에 대해서는 **6장 특별한 클래스 사용하기**에서 살펴본다.

### 클래스 내부 구조

자바와 마찬가지로 코틀린 클래스도 class 키워드 다음에 클래스 이름이 오고, 그 다음에 클래스 본문이 오는 형태로 정의된다. 클래스 본문은 멤버 정의가 들어있는 블록이다. 어떤 사람에 대한 정보를 저장하는 클래스를 정의해보자.

<코드>

class Person {  
 var firstName: String = ""  
 var familyName: String = ""  
 var age: Int = 0  
   
 fun fullName() = "$firstName $familyName"  
  
 fun showMe() {  
 println("${fullName()}: $age")  
 }  
}

</코드>

이 정의는 모든 Person 클래스의 인스턴스마다 firstName, familyName, age라는 프로퍼티와 fullName()과 showMe()라는 두 함수가 들어있음을 알려준다. 여러 가지 프로퍼티 유형 중 가장 단순한 것은 그냥 특정 클래스와 연관된 변수다. 자바 클래스 필드와 비슷하게 생각될 수도 있다. 더 일반적인 경우에는 프로퍼티에 어떤 계산이 포함될 수 있다. 이럴 경우 클래스 인스턴스 내부에 저장되는 대신 그때그때 계산되거나, 지연 계산되거나, 맵(map)에서 값을 얻어오는 등의 방식으로 프로퍼티의 값을 제공할 수 있다. 모든 프로퍼티에서 일반적으로 쓸 수 있는 기능에는 다음과 같이 마치 변수처럼 프로퍼티를 사용하는 참조 구문이 있다.

<코드>

fun showAge(p: Person) = println(p.age) // 프로퍼티를 읽기  
fun readAge(p: Person) {  
 p.age = readLine()!!.toInt() // 프로퍼티에 쓰기  
}

</코드>

프로퍼티는 어떤 클래스의 구체적인 인스턴스와 엮여있기 때문에, 이 인스턴스를 식으로 지정해야 한다(앞의 코드에서는 p가 이런 인스턴스다). 이런 인스턴스를 수신객체(receiver)라고 부르고, 수신객체는 프로퍼티에 접근할 때 사용해야 하는 객체를 지정한다. 멤버 함수의 경우에도 똑같이 수신 객체가 있고, 이런 경우 멤버 함수를 메서드(method)라고 부른다.

<코드>

fun showFullName(p: Person) = println(p.fullname()) // 메서드 호출하기

</코드>

수신객체를 모든 클래스 멤버에게 암시적으로 제공되는 사용 가능한 추가 변수라고 생각해도 된다. 클래스 내부에서는 this 식으로 수신 객체를 참조할 수 있다. 대부분의 경우 this를 디폴트로 가정하기 때문에 수신객체의 멤버 안에서 수신객체의 멤버를 참조할 때는 this를 생략해도 된다. 예를 들어 첫 번째 예제를 다음과 같이 쓸 수도 있다.

*<코드>*

*class Person {  
 var firstName: String = ""  
 var familyName: String = ""  
 var age: Int = 0  
   
 fun fullName() = "${this.firstName} ${this.familyName}"  
 fun showMe() {  
 println("${this.fullName()}: ${this.age}")  
 }  
}*

</코드>

하지만 때로는 this가 꼭 필요한 경우도 있다. 예를 들어 어떤 클래스의 프로퍼티와 메서드 파라미터 이름이 같은 경우, 이 둘을 구분하기 위해 프로퍼티 이름 앞에 this를 써야 한다.

<코드>

class Person {  
 var firstName: String = ""  
 var familyName: String = ""  
   
 fun setName(firstName: String, familyName: String) {  
 this.firstName = firstName  
 this.familyName = familyName  
 }  
}

</코드>

**자바와 코틀린의 차이**: 자바 필드와 달리, 코틀린에서는 클라이언트 코드를 바꾸지 않아도 원하는 대로 프로퍼티의 구현을 바꿀 수 있기 때문에(예를 들어 커스텀 게터나 세터를 추가해도 클라이언트 소스 코드를 바꿀 필요가 없다), 코틀린 프로퍼티는 캡슐화(encapsulation)에 위배되지 않는다. 즉, firstName라는 참조는 프로퍼티가 구현되는 방법과 무관하게 항상 올바른 참조로 남을 수 있다. 다음 절에서는 커스텀 프로퍼티를 정의하는 방법에 대해 자세히 살펴본다.

프로퍼티가 사용하는 내부 필드는 항상 캡슐화되어 있고 클래스 정의 밖에서는 이 내부 필드에 접근할 수 없다. 사실은 프로퍼티 정의 자체가 아니면 이런 내부 필드에 접근할 방법이 없다.

클래스 인스턴스의 프로퍼티나 메서드를 사용하기 위해서는 우선 인스턴스를 명시적으로 생성해야 한다. 일반 함수 호출과 똑같아 보이는 생성자 호출을 통해 인스턴스를 만들 수 있다. 일반 함수 호출과 생성자 호출의 차이는 함수 이름 대신 클래스 이름을 사용한다는 점 뿐이다.

<코드>

fun main() {  
 val person = Person() // Person 인스턴스 생성  
   
 person.firstName = "John"  
 person.familyName = "Doe"  
 person.age = 25  
   
 person.showMe() // John Doe: 25  
}

</코드>

생성자 호출을 사용하면 프로그램이 새 인스턴스에 대한 힙 메모리를 할당한 다음, 인스턴스의 상태를 초기화해주는 생성자 코드를 호출해준다. 앞의 예제에서는 아무 인자도 받지 않는 디폴트 생성자를 사용했다. 다음 절에서는 여러분 자신의 초기화 코드를 실행할 수 있는 커스텀 생성자를 작성하는 방법을 살펴본다.

기본적으로 코틀린 클래스는 공개(public) 가시성이다. 이 말은 코드의 어느 부분에서나 클래스를 사용할 수 있다는 뜻이다. 최상위 함수와 마찬가지로 최상위 클래스를 internal이나 private로 설정할 수 있다. 이렇게 지정하면 클래스의 가시성 범위를 클래스 정의가 들어있는 파일 내부나 컴파일 모듈 내부로 제한할 수 있다.

**자바와 코틀린의 차이**: 코틀린과 다르게, 자바에서는 기본적인 가시성이 클래스가 포함된 패키지 내부로 제한된다(이런 가시성을 패키지 내부공개(package private) 가시성이라고 부른다). 자바에서 어떤 정의를 아무데서나 쓸 수 있게 하려면 명시적으로 public 변경자를 정의 앞에 붙여야 한다.

코틀린에서는 소스 파일의 이름을 그 안에 들어있는 공개 클래스 이름과 똑같이 만들 필요가 없다. 한 파일 안에 여려 공개 클래스를 넣을 수도 있다. 그러나 만약 어떤 파일 안에 클래스가 하나만 들어있다면 보통은 클래스 이름과 파일 이름을 같게 한다. 하지만 코틀린에서는 이런 식으로 이름을 짓는 방식이 (자바와 달리) 엄격한 요구사항이 아니라 취향의 문제다.

클래스 프로퍼티는 지역 변수와 마찬가지로 불변일 수 있다. 하지만 이런 경우 초기화를 하는 동안 프로퍼티의 값을 지정할 수단이 있어야 한다. 그렇지 않다면 모든 클래스가 같은 프로퍼티 값을 사용하게 된다.

<코드>

class Person {  
 // 생성자로 초기화할 방법이 없으면 모든 클래스가   
 // firstName에 대해 같은 값을 사용하게 됨  
 val firstName = "John"  
}

</코드>

커스텀 생성자를 사용하면 이런 초기화가 가능하다. 다음 절의 주제가 바로 커스텀 생성자다.

## 생성자

생성자는 클래스 인스턴스를 초기화해주고 인스턴스 생성 시 호출되는 특별한 함수다. 다음 클래스를 보자.

<코드>

class Person(firstName: String, familyName: String) {  
 val fullName = "$firstName $familyName"  
}

</코드>

class 키워드 다음의 이름 뒤에 덧붙인 파라미터 목록을 살펴보라. 이 파라미터는 프로그램이 클래스의 인스턴스를 생성할 때 클래스에 전달된다. 이 파라미터를 사용해 프로퍼티를 초기화하고 다른 일을 수행할 수 있다.

<코드>

fun main() {  
 val person = Person("John", "Doe") // 새 Person 인스턴스 생성  
 println(person.fullName) // John Doe  
}

</코드>

**자바와 코틀린의 차이**: 코틀린에서는 생성자를 호출할 때 (자바의 new와 같은) 특별한 키워드를 사용하지 않는다.

클래스 헤더의 파라미터 목록을 주생성자(primary constructor) 선언이라고 부른다. 주생성자는 함수와 달리 본문이 하나가 아니다. 대신 주생성자는 클래스 정의 내에서 프로퍼티 초기화와 초기화 블록이 등장하는 순서대로 구성된다. 초기화 블록이란 init이라는 키워드가 앞에 붙은 블록이다. 이 블록 안에서 클래스 초기화 시 필요한 간단하지 않은 초기화 로직을 수행할 수 있다. 예를 들어 다음 클래스는 주생성자가 호출될 때마다 메시지를 표시한다.

<코드>

class Person(firstName: String, familyName: String) {  
 val fullName = "$firstName $familyName"  
   
 init {  
 println("Created new Person instance: $fullName")  
 }  
}

</코드>

클래스 안에 init 블록이 여럿 들어갈 수 있다. 이런 경우 각 블록은 프로퍼티 초기화와 함께 순서대로 실행된다.

초기화 블록에는 return 문이 들어가지 못한다는 점에 유의하라.

<코드>

class Person(firstName: String, familyName: String) {  
 val fullName = "$firstName $familyName"  
   
 init {  
 // error: 'return' is not allowed here  
 if (firstName.isEmpty() && familyName.isEmpty()) return   
 println("Created new Person instance: $fullName")  
 }  
}

</코드>

지금까지는 항상 프로퍼티 값을 프로퍼티 정의 시 초기화했다. 하지만 하나의 식으로 표현하기 어려운 복잡한 초기화 로직을 실행해야 프로퍼티를 초기화할 수 있는 경우도 있다. 이런 이유로 코틀린은 init 블록 안에서 프로퍼티를 초기화하는 것도 허용한다.

<코드>

class Person(fullName: String) {  
 val firstName: String  
 val familyName: String  
 init {  
 val names = fullName.split(" ")  
 if (names.size != 2) {  
 throw IllegalArgumentException("Invalid name: $fullName")  
 }  
 firstName = names[0]  
 familyName = names[1]  
 }  
}  
  
fun main() {  
 val person = Person("John Doe")  
 println(person.firstName) // John  
}

</코드>

이 예제에서 init 블록은 fullName을 공백으로 분리된 부분 문자열의 배열로 나눈 다음에 이를 사용해 firstName과 familyName 프로퍼티를 초기화한다.

컴파일러는 모든 프로퍼티가 확실히 초기화되는지 확인한다. 컴파일러가 주생성자의 모든 실행 경로가 모든 멤버 프로퍼티를 초기화하거나, (일부 프로퍼티가 초기화가 안 되는 경우) 예외를 발생시키는지 확인할 수 없다면 다음과 같은 오류가 발생한다.

<코드>

class Person(fullName: String) {  
 // error: property must be initialized or be abstract  
 val firstName: String  
 val familyName: String  
 init {  
 val names = fullName.split(" ")  
 if (names.size == 2) {  
 firstName = names[0]  
 familyName = names[1]  
 }  
 }  
}

</코드>

주생성자 파라미터를 프로퍼티 초기화나 init 블록 밖에서 사용할 수는 없다. 예를 들어 멤버 함수 내부에서는 firstName을 사용할 수 없기 때문에 다음 코드는 잘못된 코드다.

<코드>

class Person(firstName: String, familyName: String) {  
 val fullName = "$firstName $familyName"  
 fun printFirstName() {  
 println(firstName) // Error: first name is not available here  
 }  
}

</코드>

이에 대한 해법은 생성자 파라미터의 값을 저장할 멤버 프로퍼티를 정의하는 것이다.

<코드>

class Person(firstName: String, familyName: String) {  
 val firstName = firstName // firstName은 생성자 파라미터를 가리킴  
 val fullName = "$firstName $familyName"  
 fun printFirstName() {  
 println(firstName) // Ok: 여기서 firstName은 멤버 프로퍼티를 가리킴  
 }  
}

</코드>

하지만 코틀린은 간단하게 생성자 파라미터의 값을 멤버 프로퍼티로 만들 수 있는 방법을 제공한다.

<코드>

class Person(val firstName: String, familyName: String) {  
 // firstName은 생성자 파라미터를 가리킴  
 val fullName = "$firstName $familyName"  
   
 fun printFirstName() {  
 println(firstName) // firstName은 멤버 프로퍼티를 가리킴  
 }  
}  
  
fun main() {  
 val person = Person("John", "Doe")  
 println(person.firstName) // firstName은 프로퍼티를 가리킴  
}

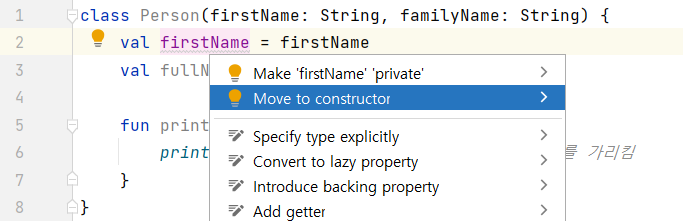
</코드>

기본적으로 생성자 파라미터 앞에 val이나 var 키워드를 덧붙이면 자동으로 해당 생성자 파라미터로 초기화되는 (생성자 파라미터와 이름이 같은) 프로퍼티를 정의한다. 이때 파라미터 이름을 프로퍼티 초기화나 init 블록 안에서 참조하면 생성자 파라미터를 가리키고, 다른 위치에서 참조하면 프로퍼티를 가리키게 된다.

<참고> IDE 팁

인텔리J 코틀린 플러그인은 멤버 프로퍼티의 값을 생성자 파라미터의 값으로 초기화하는 경우에, 해당 프로퍼티를 val/var 파라미터로 변환해준다(그림 4-1).

그림 4-1: 프로퍼티를 생성자 val/var 파라미터로 변환하기



</참고>

val/var 파라미터를 사용하면 단순하지 않은 멤버가 포함되지만 본문은 비어있는 클래스를 정의할 수 있다.

<코드>

class Person(val firstName: String, val familyName: String = "") {  
}

</코드>

이런 경우 코틀린에서는 본문을 아예 생략할 수 있다. 실제 인텔리J 코틀린 플러그인은 이런 코딩 스타일을 권장한다.

<코드>

class Person(val firstName: String, val familyName: String = "")

</코드>

함수와 마찬가지로 디폴트 값과 vararg를 생성자 파라미터에 사용할 수 있다.

<코드>

class Person(val firstName: String, val familyName: String = "") {  
 fun fullName() = "$firstName $familyName"  
}  
  
class Room(vararg val persons: Person) {  
 fun showNames() {  
 for (person in persons) println(person.fullName())  
 }  
}  
  
fun main() {  
 val room = Room(Person("John"), Person("Jane", "Smith"))  
 room.showNames()  
}

</코드>

여러 생성자를 사용해 클래스 인스턴스를 서로 다른 방법으로 초기화하고 싶을 때도 있다. 이런 경우도 대부분은 디폴트 파라미터를 사용하는 주생성자로 해결할 수 있지만, 경우에 따라 주생성자만으로는 충분하지 않을 수도 있다. 코틀린에서는 이런 문제를 부생성자(secondary constructor)를 사용해 해결할 수 있다. 부생성자 문법은 fun 함수이름 대신 constructor 키워드를 사용한다는 점을 제외하면 함수 정의 문법과 비슷하다.

<코드>

class Person {  
 val firstName: String  
 val familyName: String  
   
 constructor(firstName: String, familyName: String) {  
 this.firstName = firstName  
 this.familyName = familyName  
 }  
   
 constructor(fullName: String) {  
 val names = fullName.split(" ")  
 if (names.size != 2) {  
 throw IllegalArgumentException("Invalid name: $fullName")  
 }  
 firstName = names[0]  
 familyName = names[1]  
 }  
}

</코드>

부생성자에 반환 타입을 지정할 수는 없지만, 기본적으로 부생성자는 Unit 타입 값을 반환하는 함수와 마찬가지 형태다. 특히 (init 블록과 달리) 부생성자 안에서는 return을 사용할 수 있다.

클래스에 주생성자를 선언하지 않은 경우, 모든 부생성자는 자신의 본문을 실행하기 전에 프로퍼티 초기화와 init 블록을 실행한다. 이렇게 하면 어떤 부생성자를 호출하든지 공통적인 초기화 코드가 정확히 한번만 실행되게 보장할 수 있다.

다른 방법으로는 부생성자가 생성자 위임 호출을 사용해 다른 부생성자를 호출하는 방법이 있다.

<코드>

class Person {  
 val fullName: String  
 constructor(firstName: String, familyName: String):  
 this("$firstName $familyName")  
 constructor(fullName: String) {  
 this.fullName = fullName  
 }  
}

</코드>

생성자 파라미터 목록 뒤에 콜론(:)을 넣고 그 뒤에 일반 함수를 호출하는 것처럼 코드를 작성하되 함수 이름 대신 this를 사용하면 생성자 위임 호출이 된다.

클래스에 주생성자가 있다면, (부생성자가 있는 경우) 모든 부생성자는 주생성자에게 위임을 하거나 다른 부생성자에게 위임을 해야 한다. 예를 들어 앞의 예제에서 두 번째 생성자를 주생성자로 바꿀 수 있다.

<코드>

class Person(val fullName: String) {  
 constructor(firstName: String, familyName: String):  
 this("$firstName $familyName")  
}

</코드>

부생성자의 파라미터 목록에는 val/var 키워드를 쓸 수 없다는 점에 유의하자.

<코드>

class Person {  
 // error: 'val' on secondary constructor parameter is not allowed   
 constructor(val fullName: String)   
}

</코드>

부생성자와 클래스 상속이 결합되는 경우, 상위클래스 생성자를 호출하는 것과 관련된 다른 문제가 있다. 이에 대해서는 **8장 클래스 계층 이해하기**에서 살펴본다.

## 멤버 가시성

클래스 멤버마다 가시성을 다르게 지정할 수 있다. 즉, 각각 어떤 영역에서 쓰일 수 있는지 결정할 수 있다. 가시성을 사용해 구현과 관련한 세부 사항을 캡슐화해서 외부 코드로부터 구현 세부 사항을 격리시킬 수 있기 때문에, 가시성 지정은 클래스 정의 시 아주 중요한 부분이다. 코틀린에서는 클래스 멤버의 가시성을 다음과 같은 변경자 키워드로 지정할 수 있다.

* public(공개): 멤버를 어디서나 볼 수 있다. 디폴트 가시성이 바로 public이다. 따라서 명시적으로 public을 표기할 필요가 없다.
* internal(모듈 내부): 멤버를 멤버가 속한 클래스가 포함된 컴파일 모듈에 내부에서만 볼 수 있다.
* protected(보호): 멤버를 멤버가 속한 클래스와 멤버가 속한 클래스의 모든 하위 클래스 안에서 볼 수 있다. 이에 대해서는 **8장 클래스 계층 이해하기**에서 상속을 다룰 때 자세히 설명한다.
* private(비공개): 멤버를 멤버가 속한 클래스 내부에서만 볼 수 있다.

각 변경자의 뜻은 최상위 함수나 프로퍼티에서 살펴본 내용과 비슷하다.

**자바와 코틀린의 차이**: 자바의 기본 가시성은 패키지-전용(package parivate)으로, 어떤 멤버를 이 멤버가 속한 클래스가 들어있는 패키지 내부 어디서나 볼 수 있다. 자바에서 클래스 멤버를 공개 멤버로 정의하고 싶으면 명시적으로 public 변경자를 붙여야 한다. 반대로 코틀린에서는 클래스 멤버(실제로는 지역 선언이 아닌 모든 선언)의 디폴트 가시성이 공개 가시성이다. 또한, 코틀린에는 자바의 패키지-전용에 해당하는 가시성이 없다는 점에 유의하라[[1]](#footnote-1).

다음 코드에서 firstName과 familyName 프로퍼티는 private이기 때문에 main() 함수에서 이 둘을 볼 수 없다. 반면 fullName()은 public이다.

<코드>

class Person(private val firstName: String,  
 private val familyName: String) {  
 fun fullName() = "$firstName $familyName"  
}  
  
fun main() {  
 val person = Person("John", "Doe")  
 // error: cannot access 'firstName': it is private in 'Person'  
 println(person.firstName)   
 println(person.fullName()) // Ok  
}

</코드>

함수와 프로퍼티, 주생성자, 부생성자에 대해 가시성 변경자를 지원한다. 함수와 프로퍼티, 주/부 생성자는 모두 클래스 본문에 정의되거나 주생성자 파라미터로 정의된다. 주생성자의 가시성을 지정하려면 constructor 키워드를 꼭 명시해야 한다.

<코드>

class Empty private constructor() {  
 fun showMe() = println("Empty")  
}  
  
fun main() {  
 // error: cannot access '<init>': it is private in 'Empty'  
 Empty().showMe()  
}

</코드>

Empty 클래스의 유일한 생성자가 private이기 때문에 이 클래스를 클래스 본문 외부에서 인스턴스화 할 수 없다는 점에 유의하라. 이어서 **객체**에 대해 살펴볼 텐데 동반객체(companion object)에서 팩토리 메서드(factory method)를 제공할 때 생성자를 감추는 방법을 어떻게 함께 조합할지 알아보자.

### 내포된 클래스

함수, 프로퍼티, 생성자 외에도, 코틀린 클래스는 다른 클래스를 멤버로 가질 수 있다. 이런 클래스를 내포된(nested) 클래스라고 부른다. 예를 하나 살펴보자.

<코드>

class Person (val id: Id, val age: Int) {  
 class Id(val firstName: String, val familyName: String)  
 fun showMe() = println("${id.firstName} ${id.familyName}, $age")  
}  
  
fun main() {  
 val id = Person.Id("John", "Doe")  
 val person = Person(id, 25)  
 person.showMe()  
}

</코드>

내포된 클래스를 둘러싸고 있는 클래스의 본문 밖에서는 Person.Id처럼 내포된 클래스 이름 앞에 바깥쪽 클래스의 이름을 덧붙여야만 내포된 클래스를 참조할 수 있다는 점에 유의하라.

다른 멤버와 마찬가지로 내포된 클래스에도 여러 가지 가시성을 지정할 수 있다. 내포된 클래스도 자신을 둘러싼 클래스의 멤버이기 때문에 자신을 포함하는 클래스의 비공개 선언에 접근할 수 있다.

<코드>

class Person (private val id: Id, private val age: Int) {  
 class Id(private val firstName: String,  
 private val familyName: String) {

fun nameSake(person: Person) = person.id.firstName == firstName

}  
  
 // error: cannot access 'familyName': it is private in 'Id'  
 fun showMe() = println("${id.firstName} ${id.familyName}, $age") }

</코드>

**자바와 코틀린의 차이**: 자바와 달리 바깥쪽 클래스는 자신에게 내포된 클래스의 비공개 멤버에 접근할 수 없다(앞의 예제 코드의 오류를 보라).

내포된 클래스에 inner를 붙이면 자신을 둘러싼 외부 클래스의 현재 인스턴스에 접근할 수 있다.

<코드>

class Person(val firstName: String, val familyName: String) {  
 inner class Possession(val description: String) {  
 fun showOwner() = println(fullName())  
 }  
 private fun fullName() = "$firstName $familyName"  
}  
  
fun main() {  
 val person = Person("John", "Doe")  
 // Possession 생성자 호출  
 val wallet = person.Possession("Wallet")  
 wallet.showOwner() // John Doe  
}

</코드>

여기서 내부(inner[[2]](#footnote-2)) 클래스 생성자를 호출할 때 person.Possession("Wallet") 처럼 외부 클래스 인스턴스를 지정해야 한다는 점에 유의하라. 다른 멤버들과 마찬가지로 내부 클래스를 가리킬 때도 this를 생략해도 된다.

<코드>

class Person(val firstName: String, val familyName: String) {  
 inner class Possession(val description: String) {  
 fun showOwner() = println(fullName())  
 }  
  
 // this.Possession("Wallet")와 같음  
 val myWallet = Possession("Wallet")  
  
 fun fullName() = "$firstName $familyName"  
}

</코드>

일반적으로 this는 항상 가장 내부의 클래스 인스턴스를 가리킨다. 따라서 내부 클래스 본문에서 this는 내부 클래스 자신을 가리킨다. 내부 클래스 본문에서 외부 클래스 인스턴스를 가리켜야 한다면 한정시킨 this(qualified this) 식을 사용해야 한다.

<코드>

class Person(val firstName: String, val familyName: String) {  
 inner class Possession(val description: String) {  
 fun getOwner() = this@Person  
 }  
}

</코드>

한정시킨 this 식에서 @ 기호 다음에 오는 식별자는 외부 클래스의 이름이다.

**자바와 코틀린의 차이**: 코틀린의 내포 클래스와 자바의 내포 클래스는 아주 비슷하다. 주된 차이는 코틀린 내부 클래스 앞에 inner 변경자가 붙는다는 점이다. 자바 클래스는 디폴트로 내부 클래스이며, 내부 클래스가 외부 클래스 인스턴스와 연관되기를 원하지 않으면 명시적으로 static을 붙여야 한다. 반면 코틀린의 (inner가 없는) 내포 클래스는 외부 클래스 인스턴스와 연관되지 않는다. 따라서 다음 코틀린 코드는,

<코드>

class Outer {  
 inner class Inner  
   
 class Nested  
}

</코드>

다음 자바 코드와 같다.

<코드>

class Outer {  
 public class Inner {  
   
 }  
   
 public class class Nested {  
   
 }  
}

</코드>

### 지역 클래스

자바처럼 코틀린에서도 함수 본문에서 클래스를 정의할 수 있다. 이런 지역 클래스는 자신을 둘러싼 코드 블록 안에서만 쓰일 수 있다.

<코드>

fun main() {  
 class Point(val x: Int, val y: Int) {  
 fun shift(dx: Int, dy: Int): Point = Point(x + dx, y + dy)  
 override fun toString() = "($x, $y)"  
 }  
 val p = Point(10, 10)  
 println(p.shift(-1, 3)) // (9, 13)  
}  
  
fun foo() {  
 println(Point(0, 0)) // error: unresolved reference: Point  
}

</코드>

지역 함수와 비슷하게 코틀린 지역 클래스도 자신을 둘러싼 코드의 선언에 접근할 수 있다. 특히 지역 클래스는 클래스 본문 안에서 자신이 접근할 수 있는 값을 포획(capture)할 수 있고, 심지어는 변경할 수도 있다.

<코드>

fun main() {  
 var x = 1  
   
 class Counter {  
 fun increment() {  
 x++  
 }  
 }  
   
 Counter().increment()  
   
 println(x) // 2  
}

</코드>

**자바와 코틀린의 차이**: 코틀린과 달리 자바에서는 포획한 변수의 값을 변경할 수 없다. 더 나아가 익명 클래스 내부에서 포획한 변수를 사용할 경우에는 명시적으로 final로 선언해야만 한다. 하지만 코틀린이 제공하는 포획 변수를 변경하는 기능은 그에 따른 비용을 수반한다. 익명 객체와 이 객체를 둘러싸고 있는 코드 사이에 변수를 공유하기 위해서 코틀린 컴파일러는 (공유되는) 값을 특별한 래퍼(wrapper) 객체로 둘러싼다. 위 Counter 예제를 컴파일한 바이트 코드에 해당하는 자바 코드는 다음과 같다.

<코드>

import kotlin.jvm.internal.Ref.IntRef;  
  
class MainKt {  
 public static void main(String[] args) {  
 final IntRef x = new IntRef(); // 래퍼 생성  
 x.element = 1;  
   
 final class Counter {  
 public final void increment() {  
 x.element++; // 공유된 데이터 변경하기  
 }  
 }  
   
 (new Counter()).increment();  
   
 System.out.println(x.element); // 공유된 데이터 읽기  
 }  
}

</코드>

불변 변수는 값이 바뀌지 않기 때문에 래퍼가 필요 없어서 이런 부가비용이 없다.

내포된 클래스와 달리 지역 클래스에는 가시성 변경자를 붙일 수 없다. 지역 클래스의 영역은 항상 자신을 둘러싼 블록으로 제한된다.

지역 클래스도 함수, 프로퍼티, 생성자, 내포된 클래스 등 다른 클래스가 포함할 수 있는 모든 멤버를 포함할 수 있다. 하지만 내포된 클래스는 반드시 inner 클래스여야만 한다.

<코드>

fun main(args: Array<String>) {  
 class Foo {  
 val length = args.size  
 inner class Bar {  
 val firstArg = args.firstOrNull()  
 }  
 }  
}

</코드>

지역 클래스 안에 내부 클래스가 아닌 내포 클래스를 허용하지 않는 이유는 무엇일까? 지역 클래스는 자신을 둘러싼 지역적인 상태(이 예제에서는 args)에 접근할 수 있다. 하지만, 내포 클래스의 특성상 지역 클래스 안에 있는 내포 클래스는 자신의 외부 클래스에서 사용할 수 있는 상태에 접근할 수 없는데, 구문 영역(lexical scope)에 따른 변수 가시성 규칙이 지역 클래스 안의 내포 클래스에서만 동작하지 않으면 얼핏 혼동을 야기하기 쉽다.

## 널 가능성

자바와 마찬가지로 코틀린의 참조 값에는 아무것도 참조하지 않는 경우를 나타내는 특별한 null(널)이라는 값이 있다. 이 참조는 그 어떤 할당된 객체도 가리키지 않는 참조를 뜻한다. 널은 다른 참조와 비슷하게 동작하지 않는다. 자바에서는 모든 참조 타입의 변수에 널을 대입할 수 있지만, 이때 이 참조 타입(하지만 값은 널임)에 정의된 메서드나 프로퍼티를 사용하려고 하면 NullPointerException(줄여서 NPE라고 부름)이 발생한다. 이 오류가 최악인 이유는 컴파일러가 정적인 타입 정보만으로는 이런 오류를 잡아낼 수 없어서 런타임에 프로그램을 실행해봐야 이 오류를 찾을 수 있기 때문이다[[3]](#footnote-3).

코틀린 타입 시스템에는 널 값이 될 수 있는 참조 타입과 널 값이 될 수 없는 참조 타입을 확실히 구분해주는 큰 장점이 있다. 이 기능은 널 발생 여부를 컴파일 시점으로 옮겨주기 때문에 악명 높은 NullPointerException 예외를 상당 부분 막을 수 있다.

이 절에서는 널이 될 수 있는 값을 표현할 때 쓰이는 타입에 대해 설명하고, 널을 다룰 때 사용하는 기본적인 연산에 대해 다룬다. 또 **12장 자바 상호 운용성**에서 자바와 코틀린을 상호 운용할 때 생기는 널 가능성 문제에 대해 다룰 것이다.

### 널이 될 수 있는 타입

코틀린 타입 시스템의 중요한 특징은 널 값을 포함하는 타입과 그렇지 않은 타입을 구분하는 능력이다. 자바에서 모든 참조 타입은 널이 될 수 있는 타입으로 간주된다. 즉, 컴파일러는 어떤 참조 타입의 변수가 null이 아닌 값만 포함한다는 사실을 보장하지 못한다.

하지만 코틀린에서는 기본적으로 모든 참조 타입은 널이 될 수 없는 타입이다. 따라서 String 같은 타입에 null 값을 대입할 수 없다. 주어진 문자열에 문자(유니코드에서 문자로 분류되는 코드포인트들)만 들어있는지 검사하는 다음 함수를 살펴보자.

<코드>

fun isLetterString(s: String): Boolean {  
 if (s.isEmpty()) return false  
 for (ch in s) {  
 if (!ch.isLetter()) return false  
 }  
 return true  
}

</코드>

null을 s 파라미터에 넘기면 컴파일 오류가 발생한다.

<코드>

fun main() {  
 println(isLetterString("abc")) // Ok  
 // error: null can not be a value of a non-null type String  
 println(isLetterString(null))   
}

</코드>

이유는 두 번째 호출의 인자가 널이 될 수 있는 타입의 값이지만, String은 널을 받을 수 있는 타입이 아니기 때문이다. 따라서 이런 호출은 금지된다. isLetterString()에게 널이 전달되지 않는다는 사실을 보장하므로, 함수 자체에서는 널에 대한 검사를 추가로 수행할 필요가 없고 파라미터를 역참조(dereference)할 때 NPE가 발생하지 걱정할 필요가 없다. 코틀린 컴파일러는 이런 오류를 컴파일 시점에 방지해준다.

**자바와 코틀린의 차이**: 이와 반대로 컴파일러 관점에서 볼 때 자바에서 null을 다음 함수에 전달하는 것도 완전히 합법적이다. 하지만 런타임에 NullPointerException이 발생한다.

<코드>

class Test {  
 static booleanisLetterString(String s) {  
 for (int i = 0; i<s.length; i++) {  
 if (!Character.isLetter(s.charAt(i))) return false;  
 }  
 return true;  
 }  
 public static void main(String[] args) {  
 // 컴파일은 되지만 런타임에 예외가 발생한다  
 System.out.println(isEmpty(null))  
 }  
}

</코드>

코틀린에서 널이 될 수도 있는 값을 받는 함수를 작성하려면 어떻게 해야 할까? 이런 경우 파라미터 타입 뒤에 물음표(?)를 붙여서 타입을 널이 될 수 있는 타입으로 지정해야 한다.

<코드>

fun isBooleanString(s: String?) = s == "false" || s == "true"

</코드>

코틀린에서 String? 같은 타입은 널이 될 수 있는 타입(nullable type)이라고 불린다. 타입 시스템 용어에서 모든 널이 될 수 있는 타입은 원래 타입(?가 붙지 않은 타입)의 상위타입이며, 원래 타입에 속하는 모든 값으로 이뤄진 집합을 null로 확장한 집합이 값의 집합이 된다. 이 말은 특히 널이 될 수 있는 타입의 변수에 항상 널이 될 수 없는 타입의 값을 대입할 수 있다는 뜻이다. 하지만 물론 반대로 널이 될 수 없는 타입의 변수에 널이 될 수 있는 타입의 값을 대입할 수는 없다.

<코드>

fun main() {  
 println(isBooleanString(null)) // Ok  
 val s: String? = "abc" // Ok  
 // error: type mismatch: inferred type is String? but String was expected   
 val ss: String = s  
}

</코드>

앞 예제의 마지막 대입문은 잘못된 문장이다. 변수 s가 런타임에 null 값을 저장할 수 없지만, 우리가 s의 타입을 널이 될 수 있는 타입으로 지정했기 때문에 컴파일러는 이 정적인 타입 정보만 사용해 보수적으로 판단할 수밖에 없다.

런타임에 널이 될 수 없는 값은 실제로 널이 될 수 있는 값과 차이가 없다. 둘 사이 구분은 컴파일 수준에서만 존재한다. 코틀린 컴파일러는 널이 될 수 없는 값을 표현하기 위해 어떤 래퍼(예를 들어 자바 8의 Optional 클래스 같은)도 사용하지 않는다. 따라서 런타임에는 아무 부가비용도 들지 않는다.

Int나 Boolean같은 원시 타입도 널이 될 수 있는 타입이 존재한다. 하지만 원시 타입의 널이 될 수 있는 타입은 항상 박싱한 값만 표현한다는 점을 명심하라.

<코드>

fun main() {  
 val n: Int = 1 // 원시 타입의 값  
 val x: Int? = 1 // 박싱한 타입의 값을 참조  
}

</코드>

가장 작은 널이 될 수 있는 타입은 Nothing?이다. 이 타입은 널 상수 이외의 어떤 값도 포함하지 않는다. 이 타입은 null 값 자체의 타입이며 다른 모든 널이 될 수 있는 타입의 하위 타입이다. 가장 큰 널이 될 수 있는 타입은 Any?이며 Any?는 코틀린 타입 시스템 전체에서 가장 큰 타입으로, 널이 될 수 있는 모든 타입과 널이 될 수 없는 모든 타입의 상위 타입이다.

널이 될 수 있는 타입은 원래 타입(널이 될 수 있는 타입에 해당하는 널이 될 수 없는 타입)에 들어있는 어떤 프로퍼티나 메서드도 제공하지 않는다. 이유는 멤버 함수 호출이나 프로퍼티를 읽는 등의 일반적인 연산이 null에서는 의미가 없기 때문이다. isLetterString()함수의 파라미터 타입을 String?으로 바꾸되 나머지 부분을 모두 그대로 두면 함수 본문의 모든 s 사용법이 잘못됐다고 컴파일 오류가 발생한다.

<코드>

fun isLetterString(s: String?): Boolean {  
 // error: only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?  
 if (s.isEmpty()) return false  
   
 // error: not nullable value required to call an 'iterator()' method on for-loop range  
 for (ch in s) {  
 if (!ch.isLetter()) return false  
 }  
 return true  
}

</코드>

String? 타입에 iterator() 메서드가 없기 때문에 for 루프를 사용해 널이 될 수 있는 문자열에 대한 반복을 수행할 수는 없다.

사실 널이 될 수 있는 타입은 코틀린의 확장 메커니즘을 활용해 자체적인 메서드와 프로퍼티를 제공한다. **5장 고급 함수와 함수형 프로그래밍 활용하기**에서 이 주제에 대해 더 자세히 다룬다. 한 가지 예는 문자열을 이어주는 연산(concatenation)이다. 이 연산은 String? 타입의 값에 대해서도 잘 작동한다[[4]](#footnote-4).

<코드>

fun exclaim(s: String?) {  
 println(s + "!")  
}  
  
fun main() {  
 exclaim(null) // null!  
}

</코드>

그렇다면 isLetterString() 같은 함수가 제대로 널이 될 수 있는 값을 처리하게 바꾸려면 어떻게 해야 할까? 이런 처리를 위해 코틀린은 여러 가지 방법을 제공한다. 이제부터 이 내용을 다뤄보겠다.

### 널 가능성과 스마트캐스트

널이 될 수 있는 값을 처리하는 가장 직접적인 방법은 해당 값을 조건문을 사용해 null과 비교하는 것이다.

<코드>

fun isLetterString(s: String?): Boolean {  
 if (s == null) return false  
   
 // s 는 여기서 널이 될 수 없다  
 if (s.isEmpty()) return false  
   
 for (ch in s) {  
 if (!ch.isLetter()) return false  
 }  
   
 return true  
}

</코드>

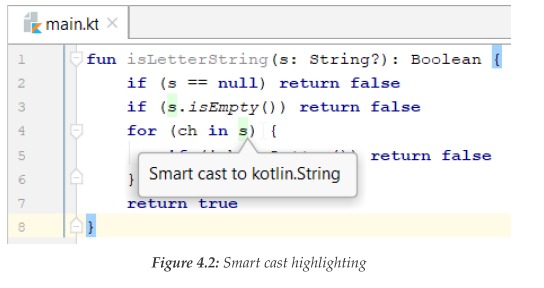
s 자체의 타입을 바꾸지는 않았지만 null에 대한 검사를 추가하면 코드가 어떤 이유로인지 컴파일이 된다. 스마트캐스트(smart cast)라고 불리는 코틀린 기능이 이런 일을 가능하게 해준다. 기본적으로 null에 대한 동등성 검사를 수행하면 컴파일러는 코드 흐름의 가지 중 한쪽에서는 대상 값이 확실히 널이고, 다른 가지에서는 널이 확실이 아니라는 사실을 알 수 있다. 그 후 컴파일러는 이 정보를 사용해 값 타입을 세분화해서 널이 될 수 있는 값을 널이 될 수 없는 값으로 타입 변환(cast)한다. 이런 기능을 스마트캐스트라고 부른다. 앞의 예제에서 컴파일러는 s == null이 참인 경우 해당 가지가 return 문으로 끝나기 때문에, s == null를 비교하는 비교문 다음의 문장은 s가 널일 경우 결코 실행되지 않는다는 사실을 알 수 있다. 그 결과 변수 s를 널이 될 수 없는 String 타입으로 가정하고 나머지 함수 본문을 실행한다.

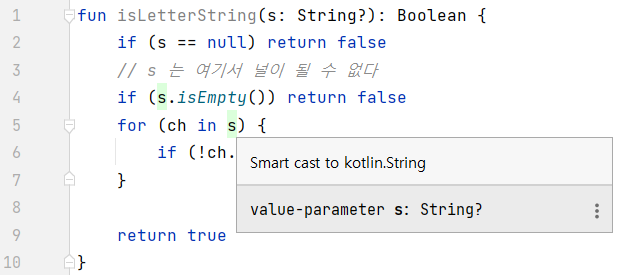
스마트캐스트는 널 가능성에만 제한되지 않는다. **8장 클래스 계층 이해하기**에서는 클래스 계층 구조 안에서 스마트캐스트를 통해 안전한 타입 캐스팅을 수행하는 방법을 살펴본다.

<참고> IDE 팁

인텔리J 플러그인은 스마트캐스트에 의해 영향을 받는 변수를 특별히 강조해준다. 이로 인해 여러분이 코드를 볼 때 이런 변수를 쉽게 구별할 수 있다. 그리고 참조 툴팁(tooltip)에서는 스마트캐스트에 의해 세분화된 타입을 보여주기까지 한다(그림 4-2).

그림 4-2: 스마트캐스트 강조





</참고>

스마트캐스트는 when이나 루프 같은 조건 검사가 들어가는 다른 문이나 식 안에서도 작동한다.

<코드>

fun describeNumber(n: Int?) = when (n) {  
 null -> "null"  
 // 아래 있는 가지에서 n은 널이 될 수 없다  
 in 0..10 -> "small"  
 in 11..100 -> "large"  
 else -> "out of range"  
}

</코드>

||나 && 연산의 오른쪽에서도 같은 일이 벌어진다.

<코드>

fun isSingleChar(s: String?) = s != null && s.length == 1

</코드>

스마트캐스트를 실행하려면 대상 변수의 값이 검사 지점과 사용 지점 사이에 변하지 않는다고 컴파일러가 확신할 수 있어야 한다. 특히 지금까지 살펴본 불변 지역 변수는 초기화 후 변경되지 않기 때문에 항상 제한 없이 스마트캐스트를 쓸 수 있다. 하지만 널 검사와 사용 지점 사이에 값이 변경되는 경우에는 스마트캐스트가 작동하지 않는다.

<코드>

var s = readLine() // String?  
if (s != null) {  
 s = readLine()  
 // 변수 값이 바뀌므로 스마트캐스트를 쓸 수 없음  
 // error: only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?  
 println(s.length)  
}

</코드>

(객체의) 가변 프로퍼티에 대해서는 절대 스마트 캐스트를 적용할 수 없다. 일반적으로 언제든 코드의 다른 부분에서 프로퍼티 값을 바꿀 수 있기 때문이다. **8장 클래스 계층 이해하기**에서 이런 규칙과 규칙의 예외에 대해 자세히 설명한다.

### 널 아님 단언 연산자

readLine() 함수와 관련해 !! 연산자를 이미 살펴봤다. !! 연산자는 널 아님 단언(not-null assertion)이라고도 부르는데, KotlinNullPointerException 예외(JVM에서 이 클래스는 악명 높은 NullPointerException의 하위 클래스다)를 발생시킬 수 있는 연산자다. 이 연산자가 붙은 식의 타입은 원래 타입의 널이 될 수 없는 버전이다. 기본적으로 널 아님 단언은 자바 프로그램의 널 관련 동작, 즉 널 값을 역참조하려 할 때 예외를 던지는 동작을 부활시킨다. 다음 예제는 이런 동작을 보여준다.

<코드>

val n = readLine()!!.toInt()

</코드>

일반적으로 널이 될 수 있는 값을 사용하려면 그냥 예외를 던지는 방식보다 더 타당한 응답을 제공해야 하기 때문에 이 연산자를 사용하지 말아야 한다. 하지만 이 연산자 사용을 정당화할 수 있는 경우가 있다. 예를 들어 다음 프로그램을 살펴보자.

<코드>

fun main() {  
 var name: String? = null  
   
 fun initialize() {  
 name = "John"  
 }  
   
 fun sayHello() {  
 println(name!!.uppercase[[5]](#footnote-5)())  
 }  
   
 initialize()  
 sayHello()  
}

</코드>

이 경우 이름에 널이 될 수 없는 값이 할당된 다음에 sayHello() 함수가 호출되기 때문에 널 아님 단언도 적절한 해법이다. 하지만 컴파일러는 이 사용이 안전하다고 인식할 수 없기 때문에 sayHello()안에서 변수 타입을 String으로 세분화하지 못한다. 따라서 널 아님 단언문을 사용해 컴파일러의 경고를 무시하는 것이 한 가지 해법이다. 하지만 이와 같은 경우라고 해도 널을 다룰 때 쓸 수 있는 덜 무딘 도구를 사용하거나, 코드 제어 흐름을 고쳐 써서 컴파일러가 스마트캐스트를 적용할 수 있게 하는 편이 더 낫다.

널 아님 단언문을 널이 될 수 없는 수신 객체에 대해 사용해도 오류로 간주되지 않는다. 하지만 이런 코드는 불필요하기 때문에 피해야 한다.

<참고> IDE 팁

인텔리J 코틀린 플러그인은 불필요한 !! 연산자를 강조하고 제거하라고 알려주는 코드 인스펙션(inspection)을 제공한다.

</참고>

다른 후위 연산자와 마찬가지로 널 아님 단언 연산자도 가장 높은 우선순위로 취급된다.

## 안전한 호출 연산자

널이 될 수 있는 타입의 값에 대해서는 그에 상응하는 널이 될 수 없는 타입의 값에 있는 메서드를 사용할 수 없다고 이미 설명했다. 하지만 특별한 안전한 호출 연산(safe call)을 사용하면 이런 제약을 피할 수 있다. 앞에서 본 예제를 다시 살펴보자.

<코드>

fun readInt() = readLine()!!.toInt()

</코드>

여러분의 프로그램이 콘솔을 표준 I/O로 사용하는 한 이 함수는 잘 작동한다. 하지만 프로그램이 파일을 표준 입력에 파이프(pipe)로 연결하면, 파일이 비어있는 경우 이 함수가 KotlinNullPointerException 예외를 발생시키면서 실패할 수 있다. 안전한 호출 연산자를 사용하면 다음 형태로 코드를 다시 작성할 수 있다.

<코드>

fun readInt() = readLine()?.toInt()

</코드>

앞의 코드는 기본적으로 다음 함수와 같다.

<코드>

fun readInt(): Int? {  
 val tmp = readLine()  
   
 return if (tmp != null) tmp.toInt() else null  
}

</코드>

즉, 안전한 호출 연산자는 수신 객체(왼쪽 피연산자)가 널이 아닌 경우 일반적인 함수 호출처럼 작동한다. 하지만 수신 객체가 널이면 안전한 호출 연산자는 호출을 수행하지 않고 그냥 널을 돌려준다. ||나 &&와 비슷하게 안전한 호출 연산도 지연 연산의 의미를 따른다. 다시 말해 수신 객체가 널이면 안전한 호출 연산자는 함수의 인자를 계산하지 않는다. 우선순위 면에서 ?. 연산자는 일반적인 함수 호출 연산자(.)와 같은 수준이다.

**‘수신 객체가 널이 아닌 경우에는 의미 있는 일을 하고, 수신 객체가 널인 경우에는 널을 반환하라’**라는 패턴은 실전에서 꽤 많이 발생한다. 따라서 안전한 호출을 사용하면 불필요한 if 식과 임시 변수 사용을 줄여서 코드를 단순화시킬 수 있다. 한 가지 유용한 숙어는 안전한 호출 연산자를 연쇄시켜 다음과 같이 쓰는 것이다.

<코드>

println(readLine()?.toInt()?.toString(16))

</코드>

안전한 호출 연산자가 널을 반환할 수 있기 때문에 이 연산자의 타입은 이에 상응하는 안전하지 않은 연산자의 타입의 널이 될 수 있는 버전이 된다. 새로 만든 readInt() 함수를 호출하는 쪽에서도 이런 타입 변화를 염두에 둬야 한다.

<코드>

fun readInt() = readLine()?.toInt()  
  
fun main() {  
 val n = readInt() // Int?  
   
 if (n != null) {  
 println(n + 1)  
 } else {  
 println("No value")  
 }  
}

</코드>

널 아님 단언과 마찬가지로, 널이 될 수 없는 수신객체에 안전한 호출을 적용할 수도 있다. 이런 코드는 간단한 점 (.) 호출 연산과 똑같기 때문에 불필요한 중복에 불과하다.

<참고> IDE 팁

인텔리J 코틀린 플러그인은 자동으로 불필요한 ?. 연산자 사용을 강조해주고 일반 호출로 바꾸라고 제안해준다.

</참고>

### 엘비스 연산자

널이 될 수 있는 값을 다룰 때 유용한 연산자로 널 복합 연산자(null coalescing operator)인 ?:를 들 수 있다. 이 연산자를 사용하면 널을 대신할 디폴트 값을 지정할 수 있다. 엘비스 프레슬리(Elvis Persley)를 닮았기 때문에 널 복합 연산자를 보통은 엘비스 연산자라고 부른다. 다음 예제를 살펴보자.

<코드>

fun sayHello(name: String?) {  
 println("Hello, " + (name ?: "Unknown"))  
}  
  
fun main() {  
 sayHello("John") // Hello, John  
 sayHello(null) // Hello, Unknown  
}

</코드>

이 연산자의 결과는 왼쪽 피연산자가 널이 아닐 경우에는 왼쪽 피연산자의 값이고 왼쪽 피연산자가 널일 경우에는 오른쪽 피연산자의 값이다. 기본적으로 sayHello() 함수는 다음 코드와 같다.

<코드>

fun sayHello(name: String?) {  
 println("Hello, " + (if (name != null) name else "Unknown"))  
}

</코드>

안전한 연산과 엘비스 연산자를 조합해서 수신 객체가 널일 때의 디폴트 값을 지정하면 유용하다. 다음 코드는 프로그램의 표준 입력이 널을 반환할 경우 0을 n에 대입한다.

<코드>

val n = readLine()?.toInt() ?: 0

</코드>

더 간편한 패턴으로, return이나 throw 같은 제어 흐름을 깨는 코드를 엘비스 연산자 오른쪽에 넣는 방법이 있다. 이렇게 하면 이에 상응하는 if 식을 대신할 수 있다.

<코드>

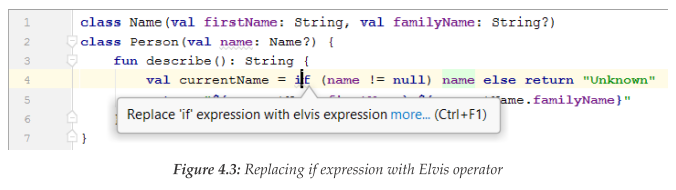
class Name(val firstName: String, val familyName: String?)  
  
class Person(val name: Name?) {  
 fun describe(): String {  
 val currentName = name ?: return "Unknown"  
 return "${currentName.firstName} ${currentName.familyName}"  
 }  
}  
  
fun main() {  
 println(Person(Name("John", "Doe")).describe()) // John Doe  
 println(Person(null).describe()) // Unknown  
}

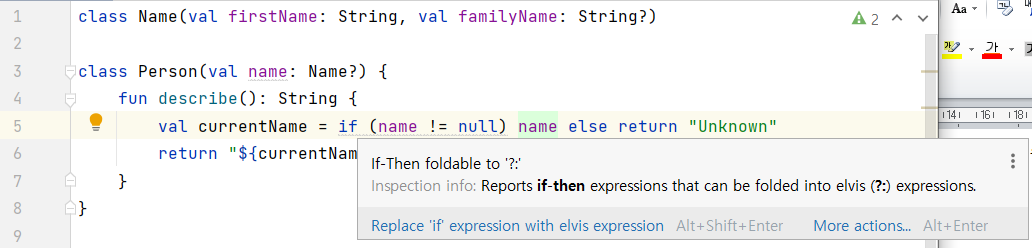
</코드>

<참고> IDE 팁

인텔리J 코틀린 플러그인은 널을 검사하는 if 식을 감지해서 엘비스 연산자로 바꿔주는 특별한 코드 인스펙션 기능을 제공한다(그림 4-3).

그림 4-3: 엘비스 연산자로 if 식 대신하기





</참고>

우선순위 면에서 엘비스 연산자는 or 등의 중위 연산자와 in, !in 사이에 위치한다. 특히 비교/동등성 연산자나 ||, &&, 대입보다 더 우선순위가 높다.

## 단순한 변수 이상인 프로퍼티

첫 번째 절에서 어떤 클래스 인스턴스나 파일로 인해 만들어지는 퍼사드(façade)와 묶인 변수라고 프로퍼티 개념을 설명했다. 하지만 일반적으로 코틀린 프로퍼티는 일반 변수를 넘어서, 프로퍼티 값을 읽거나 쓰는 법을 제어할 수 있는 훨씬 더 다양한 기능을 제공한다. 이번 절에서는 단순하지 않은 프로퍼티의 의미에 대해 자세히 살펴본다.

### 최상위 프로퍼티

클래스나 함수와 마찬가지로 최상위 수준에 프로퍼티를 정의할 수도 있다. 이런 경우 프로퍼티는 전역 변수나 상수와 비슷한 역할을 한다.

<코드>

val prefix = "Hello, " // 최상위 불변 프로퍼티  
  
fun main() {  
 val name = readLine() ?: return  
 println("$prefix$name")  
}

</코드>

이런 프로퍼티에 최상위 가시성(public/internal/private)을 지정할 수 있다. 그리고 임포트 디렉티브에서 최상위 프로퍼티를 임포트할 수도 있다.

<코드>

// util.kt  
package util  
  
val prefix = "Hello, "  
  
// main.kt  
package main  
  
import util.prefix  
  
fun main() {  
 val name = readLine() ?: return  
 println("$prefix$name")  
}

</코드>

## 늦은 초기화

클래스를 인스턴스화할 때 프로퍼티를 초기화해야 한다는 요구사항이 불필요하게 엄격할 때가 있다. 어떤 프로퍼티는 클래스 인스턴스가 생성된 뒤에, 그러나 해당 프로퍼티가 사용되는 시점보다는 이전에 초기화돼야 할 수도 있다. 예를 들어 단위 테스트를 준비하는 코드나 의존관계 주입에 의해 대입돼야 하는 프로퍼티가 이런 종류에 속한다. 이런 경우 생성자에서는 초기화가 되지 않은 상태라는 사실을 의미하는 디폴트 값을 대입하고(예: null) 실제 값을 필요할 때 대입할 수도 있다. 예를 들어 다음 코드를 생각해보자.

<코드>

import java.io.File  
  
class Content {  
 var text: String? = null  
   
 fun loadFile(file: File) {  
 text = file.readText()  
 }  
}  
  
fun getContentSize(content: Content) = content.text?.length ?: 0

</코드>

여기서 loadFile()은 다른 곳에서 호출되며 어떤 파일의 내용을 모두 문자열로 읽어온다고 가정하자. 이 예제의 단점은 실제 값이 항상 사용 전에 초기화되므로 절대 널이 될 수 없는 값이라는 사실을 알고 있음에도 불구하고 늘 널 가능성을 처리해야 한다는 점이다. 코틀린은 이런 패턴을 지원하는 lateinit 키워드를 제공한다. 앞의 예제에 lateinit를 적용해보자.

<코드>

import java.io.File  
  
class Content {  
 lateinit var text: String  
   
 fun loadFile(file: File) {  
 text = file.readText()  
 }  
}  
  
fun getContentSize(content: Content) = content.text.length

</코드>

lateinit 표시가 붙은 프로퍼티는 값을 읽으려고 시도할 때 프로그램이 프로퍼티가 초기화됐는지 검사해서 초기화되지 않은 경우 UninitializedPropertyAccessException를 던진다는 한 가지 차이를 제외하면 일반 프로퍼티와 같다. 이 특성은 때로 암시적인 !! 연산자와 비슷하다.

프로퍼티를 lateinit으로 만들기 위해서는 몇 가지 조건을 만족해야 한다. 첫째, 프로퍼티가 코드에서 변경될 수 있는 지점이 여러 곳일 수 있으므로 프로퍼티를 가변 프로퍼티(var)로 정의해야 한다. 둘째, 프로퍼티의 타입은 널이 아닌 타입이어야 하고 Int나 Boolean 같은 원시 값을 표현하는 타입이 아니어야 한다. 이유는 내부에서 lateinit 프로퍼티는 초기화되지 않은 상태를 표현하기 위해 null을 사용하는 널이 될 수 있는 값으로 표현되기 때문이다. 마지막으로, lateinit 프로퍼티를 정의하면서 초기화 식을 지정해 값을 바로 대입할 수 없다. 이런 대입을 허용하면 애초 lateinit을 지정하는 의미가 없기 때문이다.

코틀린 1.2부터 lateinit과 관련한 개선사항을 몇 가지 도입했다. 특히 이제는 최상위 프로퍼티와 지역 변수에서 늦은 초기화를 사용할 수 있게 됐다.

<코드>

lateinit var text: String  
  
fun readText() {  
 text = readLine()!!  
}  
  
fun main() {  
 readText()  
 println(text)  
}

</코드>

다른 개선으로는 lateinit 프로퍼티의 값을 읽기 전에 lateinit 프로퍼티가 설정됐는지를 알아보는 기능이 있다. 이 방법에 대해서는 **10장 애너테이션과 리플렉션**에서 코틀린 리플렉션 API를 다룰 때 살펴본다.

### 커스텀 접근자 사용하기

지금까지 살펴본 프로퍼티는 근본적으로 어떤 코틀린 클래스의 인스턴스나 어떤 파일 안의 문맥(이런 경우 JVM에서는 특별한 파사드 클래스 인스턴스로 파일 내 문맥을 저장한다) 내에 저장된 일반 변수처럼 작동했다. 하지만 코틀린 프로퍼티의 실제 능력은 변수와 함수의 동작을 한 선언 안에 조합할 수 있는 기능에 있다. 이런 기능은 커스텀 접근자(custom accessor)를 통해 이뤄진다. 커스텀 접근자는 프로퍼티 값을 읽거나 쓸 때 호출되는 특별한 함수다.

다음 예제는 프로퍼티 값을 읽을 때 사용하는 커스텀 게터(getter)를 정의한다.

<코드>

class Person(val firstName: String, val familyName: String) {  
 val fullName: String  
 get(): String {  
 return "$firstName $familyName"  
 }  
}

</코드>

게터는 프로퍼티 정의 끝에 붙으며 기본적으로 이름 대신 get이라는 키워드가 붙은 함수처럼 보인다. 하지만 이런 프로퍼티를 읽으면 프로그램이 자동으로 게터를 호출한다.

<코드>

fun main() {  
 val person = Person("John", "Doe")  
 println(person.fullName) // John Doe  
}

</코드>

함수와 비슷하게 접근자에도 식이 본문인 형태를 사용할 수 있다.

<코드>

val fullName: String  
 get() = "$firstName $familyName"

</코드>

게터에는 파라미터가 없다는 점을 기억하라. 반면 게터의 반환 타입은 (만약 반환 타입을 지정한다면) 프로퍼티의 타입과 같아야 한다.

<코드>

val fullName: Any  
 get(): String {   
 // error: getter return type must be equal to the type of the property, i.e. 'Any'  
 return "$firstName $familyName"  
 }

</코드>

코틀린 1.1부터는 프로퍼티와 게터 정의에서 프로퍼티의 타입을 생략하고 타입 추론에 의존하면 된다.

<코드>

val fullName  
 get() = "$firstName $familyName" // 타입이 String으로 추론된다

</코드>

앞의 코드처럼 도입한 프로퍼티의 값은 매번 fullName 프로퍼티를 읽을 때마다 다시 계산된다. firstName, familyName과는 달리 fullName에는 뒷받침하는 필드(backing field)가 없기 때문에 클래스 인스턴스에서 전혀 메모리를 차지하지 않는다. 즉, 기본적으로 fullName는 프로퍼티 형태인 함수와 같다. 자바에서는 이런 경우 보통 getFullName()으로 게터 이름을 정한다. 뒷받침하는 필드와 관련한 규칙은 다음과 같다. 프로퍼티에 명시적으로 field를 사용하는 디폴트 접근자나 커스텀 접근자가 하나라도 있으면 뒷받침하는 필드가 생성된다. 불변 프로퍼티의 접근자는 읽기 접근자 하나뿐이기 때문에 앞 예제에서 fullName은 직접 뒷받침하는 필드인 field를 참조하지 않는다는 사실을 쉽게 알 수 있다. 따라서 fullName 프로퍼티에는 뒷받침하는 필드가 없다.

직접 뒷받침하는 필드에 접근하는 것은 어떨까? 프로퍼티가 어떤 저장된 값을 사용하지만 프로퍼티에 대한 접근을 커스텀화해야 할 경우 뒷받침하는 필드에 접근할 수 있으면 유용하다. 예를 들어 프로퍼티를 읽을 때마다 로그를 남기고 싶다면 다음과 같이 할 수 있다.

<코드>

class Person(val firstName: String, val familyName: String, age: Int) {  
 val age: Int = age  
 get(): Int {  
 println("Accessing age")  
 return field  
 }  
}

</코드>

뒷받침하는 필드 참조는 field라는 키워드를 사용하며 접근자의 본문 안에서만 유용하다.

프로퍼티에 뒷받침하는 필드가 없다면 필드를 초기화할 수 없다. 초기화는 기본적으로 클래스를 인스턴스화할 때 값을 뒷받침하는 필드에 직접 대입하는 것이기 때문이다. 그래서 앞의 앞 예제에서 본 (뒷받침하는 필드가 없이 게터만 있는) fullName을 초기화하지 않았다. 계산에 의해 값을 돌려주는 프로퍼티의 경우 뒷받침하는 필드가 필요하지 않다.

커스텀 게터가 있는 프로퍼티는 약간의 문법적인 차이에도 불구하고 파라미터가 없는 함수처럼 동작하기 때문에, 어떤 경우 함수를 사용하고 어떤 경우 프로퍼티를 사용할지에 대한 의문이 떠오를 수 있다. 공식 코틀린 코딩 관습은 값을 계산하는 과정에서 예외가 발생할 여지가 없거나, 값을 계산하는 비용이 충분히 싸거나, 값을 캐시해 두거나, 클래스 인스턴스의 상태가 바뀌기 전에는 여러 번 프로퍼티를 읽거나, 함수를 호출해도 항상 똑같은 결과를 내는 경우에는 함수보다 프로퍼티를 사용하는 쪽을 권장한다.

var로 정의하는 가변 프로퍼티에는 값을 읽기 위한 게터와 값을 설정하기 위한 세터(setter), 두 가지 접근자가 있다. 예제를 살펴보자.

<코드>

class Person(val firstName: String, val familyName: String) {  
 var age: Int? = null  
 set(value) {  
 if (value != null && value <= 0) {  
 throw IllegalArgumentException("Invalid age: $value")  
 }  
 field = value  
 }  
}  
  
fun main() {  
 val person = Person("John", "Doe")  
 person.age = 20 // 커스텀 세터를 호출  
 println(person.age) // 20 (커스텀 게터를 호출)  
}

</코드>

프로퍼트 세터의 파라미터는 단 하나이며 타입은 프로퍼티 자체의 타입과 같아야 한다. 보통은 파라미터 타입을 항상 미리 알 수 있기 때문에 세터에서는 파라미터 타입을 생략한다. 관습적으로 파라미터 이름을 value로 정하는 경우가 많지만 원하는 다른 이름을 붙여도 좋다.

프로퍼티를 초기화하면 값을 바로 뒷받침하는 필드에 쓰기 때문에 프로퍼티 초기화는 세터를 호출하지 않는다는 점에 유의하라.

가변 프로퍼티에는 두 가지 접근자가 있기 때문에 두 접근자를 모두 커스텀화하고 두 접근자가 모두 다 field 키워드를 통해 뒷받침하는 필드를 사용하지 않는 경우를 제외하면 항상 뒷받침하는 필드가 생긴다. 예를 들어 앞의 age 프로퍼티는 디폴트 게터를 사용하고 세터에서 field를 직접 언급하기 때문에 뒷받침하는 필드를 생성한다. 반면 다음 프로퍼티는 그렇지 않기 때문에 뒷받침하는 필드가 생기지 않는다.

<코드>

class Person(var firstName: String, var familyName: String) {  
  
 var fullName: String  
 get(): String = "$firstName $familyName"  
 set(value) {  
 val names = value.split(" ") // 공백으로 구분해 단어를 분리한다  
 if (names.size != 2) {  
 throw IllegalArgumentException("Invalid full name: '$value'")  
 }  
 firstName = names[0]  
 familyName = names[1]  
 }  
}

</코드>

프로퍼티 접근자에 별도로 가시성 변경자를 붙일 수도 있다. 여러분이 프로퍼티가 포함된 클래스 외부에서는 프로퍼티의 값을 변경하지 못하게 해서 바깥 세계에서 볼 때는 실질적으로 객체가 불변인 것처럼 여겨지게 하고 싶을 때 이런 빙식을 사용할 수 있다. 단순한 접근자 구현(뒷받침하는 필드를 바로 돌려주는 게터와 뒷받침하는 필드에 값을 바로 대입하는 세터)만 필요한 경우에는 그냥 get이나 set 키워드만 사용해서 게터와 세터를 정의할 수 있다.

<코드>

import java.util.Date  
  
class Person(name: String) {  
 var lastChanged: Date? = null  
 private set // Person 클래스 밖에서는 변경할 수 없다  
   
 var name: String = name  
 set(value) {  
 lastChanged = Date()  
 field = value  
 }  
}

</코드>

**자바와 코틀린의 차이**: JVM의 관점에서 볼 때 코틀린 프로퍼티는 일반적으로 비공개 프로퍼티에 의해 뒷받침되는 한두 개의 접근자나 메서드(getFullName()과 setFullName()처럼)에 대응된다. 이들 메서드 자체는 코틀린 코드에서는 호출할 수 없지만 자바 클래스에서는 이런 메서드를 호출할 수 있으며, 이런 점이 자바/코틀린 상호 운용성에서 가장 중요한 부분이다. **12장 자바 상호 운용성**에서 이 주제에 대해 더 자세히 다룬다. 반면 비공개 프로퍼티는 자신이 포함된 클래스 밖에서 사용할 수 없기 때문에 기본적으로 아무 접근자 메서드도 생성되지 않는다. 이런 비공개 프로퍼티에 대한 접근은 직접 필드에 접근함으로써 최적화된다.

lateinit 프로퍼티의 경우 항상 자동으로 접근자가 생성되기 때문에 프로그래머가 직접 커스텀 접근자를 정의할 수 없다. 그리고 주생성자 파라미터로 선언된 프로퍼티에 대한 접근자도 지원하지 않는다. 하지만 이 문제는 앞에서 본 age처럼, 일반적인 프로퍼티가 아닌 생성자 파라미터를 사용하고 클래스 본문 안에서 프로퍼티에 그 값을 대입함으로써 해결할 수 있다.

### 지연 계산 프로퍼티와 위임

앞 절에서는 lateinit 변경자를 사용해 지연 초기화를 구현하는 방법을 살펴봤다. 하지만 어떤 프로퍼티를 처음 읽을 때까지 그 값에 대한 계산을 미뤄두고 싶을 때가 자주 있다. 코틀린에서는 lazy 프로퍼티를 통해 이를 달성할 수 있다. 예를 살펴보자.

<코드>

import java.io.File  
  
val text by lazy {  
 File("data.txt").readText()  
}  
  
fun main() {  
 while (true) {  
 when (val command = readLine() ?: return) {  
 "print data" ->println(text)  
 "exit" -> return  
 }  
 }  
}

</코드>

앞 예제에서는 text 프로퍼티를 lazy로 정의했다. lazy 다음에 오는 블록 안에는 프로퍼티를 초기화하는 코드를 지정한다. main() 함수에서 사용자가 적절한 명령으로 프로퍼티 값을 읽기 전까지, 프로그램은 lazy 프로퍼티의 값을 계산하지 않는다. 초기화가 된 이후 프로퍼티의 값은 필드에 저장되고 그 이후로는 프로퍼티 값을 읽을 때마다 저장된 값을 읽게 된다. 예를 들어 간단한 초기화를 통해 다음과 같이 프로퍼티를 정의한 경우에는

<코드>

val text = File("data.txt").readText()

</코드>

프로그램이 시작될 때 바로 파일을 읽는다. 하지만 게터를 사용한 프로퍼티를 사용해 다음과 같이 정의하면,

<코드>

val text get() = File("data.txt").readText()

</코드>

프로그램이 프로퍼티 값을 읽을 때마다 파일을 매번 다시 읽어온다.

필요하면 프로퍼티 타입을 명시할 수도 있다.

<코드>

val text: String by lazy { File("data.txt").readText() }

</코드>

이 구문은 사실 프로퍼티 처리에 필요한 데이터를 모아서 유지하면서 읽기와 쓰기를 처리하는 위임 객체(delegate object)를 통해 프로퍼티를 구현하게 해주는 위임 프로퍼티(delegate property)라는 기능의 특별한 경우다. 위임 객체는 by라는 키워드 다음에 위치하며, 코틀린이 정한 규약을 만족하는 객체를 반환할 수 있는 임의의 식이 될 수 있다. 우리 예제에서 lazy {}는 코틀린의 내장 구성 요소가 아니라 표준 라이브러리 함수에 람다를 넘기는 식일 뿐이다(**2장 코틀린 언어 기초**에서 배열 인스턴스 생성에 대해 설명할 때 람다를 넘기는 표준 라이브러리 함수를 이미 살펴봤다).

코틀린이 기본 제공하는 몇 가지 위임 객체가 있다. 지연 계산을 활성화하는 lazy 외에도, 프로퍼티를 읽거나 쓸 때마다 리스너(listener)에게 통지해주는 위임이나 프로퍼티 값을 필드에 저장하는 대신 맵에 저장하는 위임 등이 기본으로 제공된다. 이 절에서는 지연 계산 프로퍼티라는 맥락에서 여러분에게 위임을 간단히 맛 보여주고, 종합적인 처리에 대한 논의는 **7장 컬렉션과 I/O 자세히 알아보기**와 **11장 도메인 특화 언어**에서 코틀린 표준 라이브러리에서 표준 위임에 대해 설명하고 여러분 자신의 위임 객체를 설계하는 방법에 대해 설명할 때까지 미뤄두겠다.

lateinit 프로퍼티와 달리 lazy 프로퍼티는 불변 프로퍼티가 아니다. lazy 프로퍼티는 일단 초기화된 다음에는 변경되지 않는다.

<코드>

// error: type 'Lazy<String>' has no method 'setValue(Chapter4, KProperty<\*>, String)' and thus it cannot serve as a delegate for var (read-write property)  
var text by lazy { "Hello" }

</코드>

디폴트로 lazy 프로퍼티는 스레드-안전(thread-safe)하다. 즉, 다중스레드 환경에서도 값을 한 스레드 안에서만 계산하기 때문에 lazy 프로퍼티에 접근하려는 모든 스레드는 궁극적으로 같은 값을 얻게 된다.

코틀린 1.1부터는 지역 변수에도 위임을 쓸 수 있게 됐다. 이 기능을 사용하면 함수 본문에서 지연 변수를 정의할 수 있다.

<코드>

fun longComputation(): Int {...}  
  
fun main(args: Array<String>) {  
 val data by lazy { longComputation() } // lazy 지역 변수  
 val name = args.firstOrNull() ?: return  
 println("$name: $data") // name이 널이 아닐 때만 data에 접근할 수 있음  
}

</코드>

지금 현재 위임 프로퍼티에 대해서는 스마트캐스트를 사용할 수 없다는 점에 유의하라. 위임은 구현이 다 다를 수 있기 때문에 커스텀 접근자로 정의된 프로퍼티처럼 다뤄진다. 그리고 이 말은 위임을 사용한 지역 변수의 경우에도 스마트 캐스트를 쓸 수 없다는 뜻이기도 하다.

<코드>

fun main() {  
 val data by lazy { readLine() }  
   
 if (data != null) {  
 // error: smart cast to 'String' is impossible, because 'data' is a property that has open or custom getter  
 println("Length: ${data.length}")  
 }  
}

</코드>

lazy 프로퍼티와 lazy 지역 변수는 다르지 않다. 현재로써는 이 둘의 값이 초기화된 다음에 실제로 값이 바뀌지 않더라도 스마트 캐스트를 적용할 수 없다.

## 객체

이번 절에서는 객체 선언의 정의에 대해 알아본다. 코틀린에서 객체 선언은 클래스와 상수를 합한 것으로 객체 선언을 통해 싱글턴(singleton), 즉 인스턴스가 단 하나만 존재하는 클래스를 만들 수 있다. 그리고 자바 익명 클래스(anonymous class)와 비슷한 역할을 하는 객체 식에 대해서도 살펴본다.

### 객체 선언

코틀린은 어떤 클래스에 인스턴스가 오직 하나만 존재하게 보장하는 싱글턴 패턴을 내장하고 있다. 코틀린에서는 클래스와 비슷한 방법으로 싱글턴을 선언한다. 다만 class 대신 object라는 키워드를 사용한다.

<코드>

object Application {  
 val name = "My Application"  
   
 override fun toString() = name  
   
 fun exit() { }  
}

</코드>

이런 객체 선언은 클래스를 정의하는 동시에 클래스의 인스턴스를 정의하는 것이기도 하다. 예를 들어 다음 코드를 보자.

<코드>

fun describe(app: Application) = app.name // Application은 타입임  
  
fun main() {  
 println(Application) // Application은 값임  
}

</코드>

보통은 객체의 인스턴스는 단 하나뿐이기 때문에 인스턴스만 가리켜도 어떤 타입을 쓰는지 충분히 알 수 있다. 따라서 객체를 타입으로 사용해도 무의미하다.

객체 정의는 스레드-안전하다. 컴파일러는 실행되는 여러 스레드에서 싱글턴에 접근하더라도 오직 한 인스턴스만 공유되고 초기화 코드도 단 한번만 실행되도록 보장한다.

초기화는 싱글턴 클래스가 실제 로딩되는 시점까지 지연된다. 보통은 프로그램이 객체 인스턴스에 처음 접근할 때 초기화가 이뤄진다.

**자바와 코틀린의 차이**: 자바에서는 비공개 생성자와 정적(static) 상태를 조합한 일반적인 클래스 정의를 사용해 상글턴을 에뮬레이션한다. 이런 객체 선언은 세부 구현 내용에 따라 특성이 달라지며, 보통은 지연 계산 싱글턴과 미리 계산 싱글턴, 스레드-안전한 싱글턴과 스레드-안전하지 않은 싱글턴으로 구분할 수 있다. Application 객체의 JVM 바이트코드를 살펴보면 기본적으로 다음 자바 클래스와 같은 양의 코드를 볼 수 있다.

<코드>

public final class Application {  
 private static final String name = "My Application";  
   
 public static final Application INSTANCE;  
   
 private Application() { }  
   
 public final String getName() {  
 return name;  
 }  
   
 public final void exit() { }  
   
 static {  
 INSTANCE = new Application();  
 name = "My Application";  
 }  
}

</코드>

코틀린 코드에서는 INSTANCE 변수에 접근할 수 없다. 하지만 코틀린의 싱글턴을 참조하는 자바 클래스에서는 이 변수를 사용할 수 있다. **12장 자바 상호 운용성**에서 이 주제에 대해 더 자세히 다룬다.

클래스와 마찬가지로 객체 선언도 멤버 함수와 프로퍼티를 포함할 수 있고, 초기화 블록도 포함할 수 있다. 하지만 객체에는 주생성자나 부생성자가 없다. 객체 인스턴스는 항상 암시적으로 만들어지기 때문에 객체의 경우 생성자 호출이 아무 의미가 없다.

객체의 본문에 들어있는 클래스에는 inner가 붙을 수 없다. 내부 클래스의 인스턴스는 항상 바깥쪽 클래스의 인스턴스와 연관이 되는데, 객체 선언은 항상 인스턴스가 하나뿐이기 때문에 inner 변경자가 불필요해진다. 그래서 객체 안에 정의된 클래스에 대해서는 inner사용을 금지한다.

최상위 선언들과 마찬가지로, 객체의 멤버를 임포트해서 간단한 이름만 사용해 참조할 수 있다. 다음 예에서 Application 객체는 다른 파일에 정의되어 있다.

<코드>

import Application.exit  
  
fun main() {  
 println(Application.name) // 전체 이름을 사용  
 exit() // 간단한 이름을 사용  
}

</코드>

하지만 객체의 모든 멤버가 필요할 때 임포트문으로 임포트할 수는 없다.

<코드>

import Application.\* // Error

</코드>

이런 제약을 가하는 이유는 객체 정의 안에는 다른 클래스 정의와 같이 toString()이나 equals()와 같은 공통 메서드 정의가 들어있기 때문이다. 이로 인해 필요 시 임포트를 사용하면 이런 공통 메서드까지 임포트되어 문제가 생길 수 있다.

클래스와 마찬가지로 객체도 다른 클래스 안에 내포될 수 있고, 심지어는 다른 객체 안에 내포될 수도 있다. 이렇게 내포된 객체 선언도 싱글턴이며 전체 애플리케이션에서 인스턴스가 단 하나만 생긴다. 객체를 둘러싸고 있는 클래스마다 인스턴스를 별도로 만들어야 한다면 내부 클래스를 사용해야 한다. 하지만 객체를 함수 내부에 넣거나 지역 클래스나 내부 클래스 안에 넣을 수 없다. 이런 정의들은 어떤 외부 문맥에 의존하므로 싱글턴이 될 수 없기 때문이다. 객체 식(object expression)을 사용하면 지역 영역의 객체를 만들 수 있다. 이에 대해서는 조금 더 있다가 설명한다.

**자바와 코틀린의 차이**: 자바 세상에는 유틸리티 클래스(utility class)가 종종 존재한다. 유틸리티 클래스는 기본적으로 아무 인스턴스를 가지지 않고(보통은 비공개 생성자로 인스턴스 생성을 막는다) 관련 정적 메서드를 모아두는 역할만 하는 클래스다. 이런 패턴은 자바에서는 유용하지만 코틀린에서는 일반적으로 권장되지 않는 패턴이다. 무엇보다 코틀린 클래스에서는 정적 메서드를 정의할 수 없기 때문에 일반 클래스를 통해 자바와 동일한 방식의 유틸리티 클래스를 정의할 방법이 없다. 하지만 코틀린은 자바와 달리 최상위 선언을 패키지 안에 함께 모아둘 수 있기 때문에 불필요하게 유틸리티 클래스를 선언해야 할 필요가 없다.

### 동반객체

내포 클래스와 마찬가지로 내포 객체도 인스턴스가 생기면 자신을 둘러싼 클래스의 비공개 멤버에 접근할 수 있다. 이런 특성은, 예를 들어 팩토리 디자인 패턴을 쉽게 구현하는 경우 유용하게 활용할 수 있다. 생성자를 직접 사용하고 싶지 않을 때가 있다. 예를 들어 생성자를 사용하면 어떤 사전 검사 결과에 따라 널을 반환하거나 (같은 상위 타입에 속하는) 다른 타입의 객체를 반환할 수가 없다. 생성자는 항상 자신이 정의된 클래스의 객체를 반환하거나 예외를 던질 수만 있기 때문이다. 이를 해결하는 방법은 생성자를 비공개로 지정해서 클래스 외부에서 사용할 수 없게 한 다음, 내포된 객체에 팩토리 메서드 역할을 하는 함수를 정의하고 그 함수 안에서 필요에 따라 객체의 생성자를 호출하는 것이다.

<코드>

class Application private constructor(val name: String) {  
 object Factory {  
 fun create(args: Array<String>): Application? {  
 val name = args.firstOrNull() ?: return null  
 return Application(name)  
 }  
 }  
}  
  
fun main(args: Array<String>) {  
 // 직접 생성자를 호출하도록 허용하지 않음  
 // val app = Application(name)  
 val app = Application.Factory.create(args) ?: return  
 println("Application started: ${app.name}")  
}

</코드>

이런 경우 별도로 import Application.Factory.create로 팩토리 메서드를 임포트하지 않는 한 매번 내포된 객체의 이름을 지정해야 한다. 코틀린에서는 Factory 메서드를 동반객체(companion object)로 정의함으로써 이런 문제를 해결할 수 있다. 동반객체는 companion이라는 키워드를 덧붙인 내포된 객체다. 이 객체는 다른 내포된 객체와 마찬가지로 작동하지만 한 가지 예외가 있다. 동반객체의 멤버에 접근할 때는 동반객체의 이름을 사용하지 않고 동반객체가 들어있는 외부 클래스의 이름을 사용할 수 있다. 동반객체를 사용하면 앞에서 본 예제를 약간 더 간결하게 작성할 수 있다.

<코드>

class Application private constructor(val name: String) {  
 companion object Factory {  
 fun create(args: Array<String>): Application? {  
 val name = args.firstOrNull() ?: return null  
 return Application(name)  
 }  
 }  
}  
  
fun main(args: Array<String>) {  
 val app = Application.create(args) ?: return  
 println("Application started: ${app.name}")  
}

</코드>

불필요한 일이기는 하지만 여전히 동반객체의 이름을 사용해 동반객체 멤버에 접근할 수도 있다.

<코드>

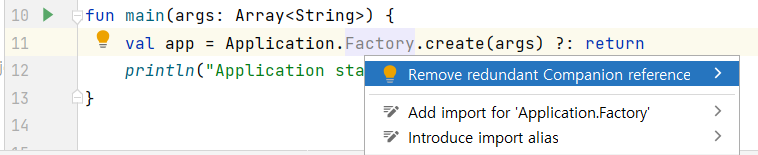
val app = Application.Factory.create(args) ?: return

</코드>

<참고> IDE 팁

인텔리J 코틀린 플러그인은 불필요하게 동반객체를 참조하는 경우 경고를 표시하고 코드에서 해당 참조를 제거하라고 알려준다(그림 4-4).

그림 4-4: 불필요한 동반객체 이름 참조



</참고>

동반객체의 경우 정의에서 이름을 아예 생략할 수도 있다. 이런 방식을 더 권장한다.

<코드>

class Application private constructor(val name: String) {  
 companion object {  
 fun create(args: Array<String>): Application? {  
 val name = args.firstOrNull() ?: return null  
 return Application(name)  
 }  
 }  
}

</코드>

동반객체 이름을 생략한 경우 컴파일러는 동반객체의 디폴트 이름을 Companion으로 가정한다.

동반객체의 멤버를 임포트하고 싶을 때는 객체 이름을 명시해야만 한다는 점에 유의하라.

<코드>

import Application.Companion.create // OK  
import Application.create // Error

</코드>

클래스에 동반객체가 둘 이상 있을 수는 없다.

<코드>

class Application {  
 companion object Factory  
 // error: only one companion object is allowed per class   
 companion object Utils   
}

</코드>

companion 변경자를 최상위 객체 앞에 붙이거나, 다른 객체에 내포된 객체 앞에 붙이는 것은 금지된다. 최상위 객체의 경우에는 동반객체를 연결할 클래스 정의가 없기 때문이고, 객체에 내포된 객체의 경우에는 companion을 붙이는 것이 불필요한 중복이기 때문이다.

**자바와 코틀린의 차이**: 코틀린의 동반객체를 자바의 정적 문맥과 대응하는 것처럼 생각할 수도 있다. 자바 정적 멤버와 마찬가지로 동반객체의 멤버도 외부 클래스와 똑같은 전역 상태를 공유하며 외부 클래스의 모든 멤버에 멤버 가시성과 관련 없이 접근할 수 있다. 하지만 중요한 차이는 코틀린 동반객체의 문맥은 객체 인스턴스라는 점이다. 이로 인해 자바의 정적 멤버(특히 정적 멤버 클래스)보다 코틀린 동반객체가 더 유연하다. 코틀린 동반객체는 다른 상위 타입을 상속할 수도 있고 일반 객체처럼 여기저기 전달될 수 있기 때문이다. **8장 클래스 계층 이해하기**와 **11장 도메인 특화 언어**에서 동반객체와 상속 및 코틀린 언어 관습을 조합해서 더 표현력이 좋은 코드를 만들어내는 방법을 살펴보겠다.

자바의 static 초기화 블록처럼 동반객체 안에서도 init 블록을 사용할 수 있다는 점도 알아두라.

### 객체 식

코틀린은 명시적인 선언 없이 객체를 바로 생성할 수 있는 특별한 식을 제공한다. 객체 식(object expression)은 자바 익명 클래스(anonymous class)와 아주 비슷하다. 다음 코드를 보자.

<코드>

fun main() {  
 fun midPoint(xRange: IntRange, yRange: IntRange) = object {  
 val x = (xRange.first + xRange.last)/2  
 val y = (yRange.first + yRange.last)/2  
 }  
   
 val midPoint = midPoint(1..5, 2..6)  
   
 println("${midPoint.x}, ${midPoint.y}") // (3, 4)  
}

</코드>

객체 식은 이름이 없는 객체 정의처럼 보인다. 그리고 객체 식도 식이기 때문에 앞의 예제처럼 객체 식이 만들어내는 값을 변수에 대입할 수 있다. 클래스나 객체 식과 달리 객체를 함수 안에 정의할 수는 없다.

<코드>

fun printMiddle(xRange: IntRange, yRange: IntRange) {  
 // error: named object 'MidPoint' is a singleton and cannot be local. Try to use anonymous object instead  
 object MidPoint {  
 val x = (xRange.first + xRange.last)/2  
 val y = (yRange.first + yRange.last)/2  
 }  
   
 println("${MidPoint.x}, ${MidPoint.y}")  
}

</코드>

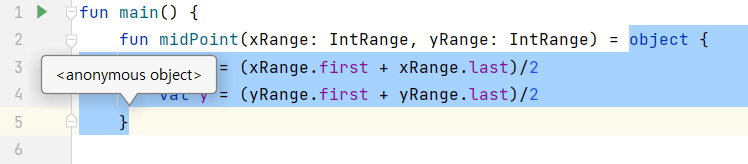
코틀린 설계자들이 이렇게 결정한 이유는 객체 선언이 싱글턴을 표현하지만 로컬 객체들은 외부 함수가 호출될 때마다 매번 다시 생성돼야 하기 때문이다.

midPoint() 함수가 반환하는 객체에 대해 명시적으로 타입을 지정하지 않았기 때문에, 이 함수의 반환 타입이 무엇인지 궁금할 것이다. 답은 객체 식 안에 정의된 모든 멤버들이 들어있는 클래스를 표현하는 익명 객체 타입(anonymous object type)이며, 이런 타입은 단 하나만 존재한다(즉, 멤버가 모두 완전히 똑같은 두 객체식이 있다고 해도, 둘의 타입은 서로 다르다). 코틀린 언어에서 이런 타입을 표현할 방법은 없다. 이 타입은 단지 코틀린 컴파일러가 객체 식의 타입을 표현하기 위해 사용하는 내부 표현일 뿐이다. 객체식이 만들어내는 객체도 다른 클래스 인스턴스와 마찬가지로 사용할 수 있다. 예를 들어 앞 예제의 println 호출에서처럼 객체 식에 정의된 멤버에 접근할 수 있다.

<참고> IDE 팁

Show Expression Type 액션([Ctrl]+[Shift]+[P]/[Cmd]-[Shift]-[P])을 사용해 객체 식의 타입을 살펴보면 인텔리J는 <anonymous object>라고 표시해준다(그림 4-5).

그림 4-5: 익명 객체 타입



</참고>

다음 예제는 객체 식을 본문으로 하는 함수의 반환 타입이 익명 객체 타입이며, 지역 변수나 프로퍼티의 타입도 마찬가지일 수 있음을 보여준다.

<코드>

fun main() {  
 val o = object { // 익명 객체 타입이 추론됨  
 val x = readLine()!!.toInt()  
 val y = readLine()!!.toInt()  
 }  
 println(o.x + o.y) // 여기서 o안의 x와 y에 접근할 수 있음  
}

</코드>

하지만 익명 객체 타입은 지역 선언이나 비공개 선언에만 전달될 수 있다. 예를 들어 midPoint 함수를 최상위 함수로 정의하면 객체 멤버에 접근할 때 컴파일 오류가 난다.

<코드>

fun midPoint(xRange: IntRange, yRange: IntRange) = object {  
 val x = (xRange.first + xRange.last)/2  
 val y = (yRange.first + yRange.last)/2  
}  
  
fun main() {  
 val midPoint = midPoint(1..5, 2..6)  
 // error: unresolved reference: x  
 // error: unresolved reference: y  
 println("${midPoint.x}, ${midPoint.y}")  
}

</코드>

여기서 midPoint() 함수의 타입은 객체 식에 해당하는 익명 객체 타입이 아니라, 객체 식에 지정된 상위타입이 된다. 하지만 예제 객체 식에는 상위 타입을 명시하지 않았기 때문에 Any를 상위 타입으로 가정한다. 그래서 midPoint.x 참조에서 x를 찾을 수 없는 것이다.

지역 함수나 클래스와 마찬가지로 객체식도 자신을 둘러싼 코드 영역의 변수를 포획할 수 있다. 이렇게 포획한 가변 변수를 객체 본문에서 변경할 수 있다. 컴파일러는 지역 클래스와 비슷하게 데이터를 공유하기 위해 필요한 래퍼를 생성해준다.

<코드>

fun main() {  
 var x = 1  
   
 val o = object {  
 fun change() {  
 x = 2  
 }  
 }  
   
 o.change()  
 println(x) // 2  
}

</코드>

지연 초기화되는 객체 선언과 달리 객체식이 만들어내는 객체는 객체 인스턴스가 생성된 직후 바로 초기화된다. 다음 코드를 보면 o 정의에서 객체가 생성되는 시점에 이 객체의 a 프로퍼티가 초기화되기 때문에 o.a에 접근하기 전에 x를 표시했음에도 불구하고 x의 값으로 2가 표시된다.

<코드>

fun main() {  
 var x = 1  
   
 val o = object {  
 val a = x++;  
 }  
   
 println("o.a = ${o.a}") // o.a = 1  
 println("x = $x") // x = 2  
}

</코드>

자바 익명 클래스와 달리 객체 식은 클래스 상속과 조합했을 때 더 강력해진다. 객체 식은 기존 클래스의 하위 클래스를 선언하지 않고도 기존 클래스를 약간만 변경해 기술하는 간결한 방법을 제공한다. 이에 대해서는 **8장 클래스 계층 이해하기**에서 살펴본다.

## 결론

이 장에서 배운 내용을 정리해보자. 코틀린 클래스를 사용해 커스텀 타입을 선언하고 사용하는 방법, 클래스를 제대로 초기화하는 방법, 싱글턴 객체 사용법을 배웠다. 그리고 읽기나 쓰기 동작을 원하는 대로 커스텀할 수 있는 여러 가지 프로퍼티 유형에 대해서도 배웠다. 우리는 이제 강력한 타입 널 가능성 메커니즘을 사용해 프로그램의 안전성을 더 향상시킬 수 있다.

나중에 코틀린의 객체지향 기능을 다시 살펴볼 것이다. 특히 **6장 특별한 클래스 사용하기**에서는 일반적인 프로그래밍 패턴을 처리하는 특별한 클래스에 대해 살펴보고, **8장 클래스 계층 이해하기**에서는 클래스 계층을 만드는 방법과 상속에 대해 살펴본다.

다음 장에서는 좀 다른 주제로 방향을 전환해서 코틀린 개발에 활용할 수 있는 다른 중요한 패러다임인 함수형 프로그래밍에 대해 배운다. 람다를 소개하고 고차 함수에 대해 논의한 후, 확장 함수와 확장 프로퍼티를 활용해 기존 타입에 새로운 기능을 추가하는 방법에 대해 알아보자.

## 문제

1. 코틀린 클래스의 기본적인 구조를 설명하라. 자바 클래스와 비교하면 어떤 차이가 있는가?
2. 주생성자란 무엇인가?
3. 부생성자란 무엇인가? 클래스에 포함시켜야 하는 생성자가 무엇인지, 부생성자가 필요할지 여부를 어떻게 결정할 수 있는가?
4. 코틀린이 지원하는 멤버 가시성은 무엇인가? 자바의 가시성과 어떤 차이가 있는가?
5. 내포된 클래스 중 내부 클래스와 비 내부 클래스의 차이는 무엇인가? 각각에 해당하는 자바 클래스와는 어떤 차이가 있는가 비교하라.
6. 함수 본문에서 클래스를 정의할 수 있는가? 정의할 수 있다면, 이렇게 정의한 클래스에는 어떤 제약이 있을까?
7. 지연 초기화 메커니즘의 요지는 무엇인가? 널이 될 수 있는 프로퍼티 대신 lateinit 프로퍼티를 사용할 경우 어떤 장점이 있는가?
8. 커스텀 프로퍼티 접근자란 무엇인가? 코틀린 접근자와 자바의 게터/세터를 비교하라.
9. 클래스를 사용하는 클라이언트 입장에서 볼 때 실질적으로 val과 같은 역할을 하는 읽기 전용 프로퍼티를 val을 쓰지 않고 만들 수 있는가? 반대로 쓸 수 만 있는 프로퍼티는 어떻게 만들 수 있을까?
10. lazy 프로퍼티를 사용해 지연 계산을 달성하는 방법은 무엇인가? lazy와 lateinit 프로퍼티를 비교해 보라.
11. 객체 선언이란 무엇인가? 코틀린 객체와 자바에서 일반적인 싱글턴 구현 패턴을 비교하라.
12. 클래스와 비교할 때 객체 선언은 어떤 제약이 있는가?
13. 일반 객체와 동반객체의 차이는 무엇인가?
14. 코틀린 동반객체와 자바의 static 내포 객체를 비교하라.
15. 자바의 익명 클래스에 해당하는 코틀린 기능은 무엇인가? 이런 코틀린 언어의 기능을 어떻게 사용할 수 있을까?

1. 옮긴이 - 자바에서는 패키지만 같으면 기본 가시성의 멤버를 볼 수 있기 때문에 .jar로 소스코드 없이 클래스파일만 배포한 경우에도 사용하는 쪽에서 패키지를 똑같이 선언해서 기본 가시성 멤버를 사용하는 일이 가능했고, 대부분의 클래스가 공개나 패키지 클래스이기 때문에 모듈 내부에서만 사용하는 클래스와 모듈 밖으로 노출시켜야 하는 클래스를 구분하기 힘들었다. 이런 문제를 해결하기 위해 자바 9에는 JigSaw 모듈 시스템이 도입됐다. 코틀린은 모듈 내부 가시성을 통해 모듈 안에서만 쓸 수 있는 선언과 그렇지 않은 선언을 구분할 수 있어서 이런 문제를 방지할 수 있다. [↑](#footnote-ref-1)
2. 옮긴이 - 내포된 클래스와 달리 내부 클래스에는 반드시 자신과 연관된 외부 클래스 인스턴스가 있어야 한다. 내부 클래스가 외부 클래스의 멤버에 접근할 수 있는 이유는 항상 자신과 연관된 외부 클래스 인스턴스가 있기 때문이다. 이를 내부 클래스 안에 컴파일러가 자동으로 정의해주는 외부 클래스 인스턴스를 가리키는 outer 같은 프로퍼티가 있는 것처럼 생각할 수 있다. [↑](#footnote-ref-2)
3. 옮긴이 - 한가지 더 심각한 문제는 null 참조가 생기는 시점과 실제 null 참조를 이용하는 시점이 서로 다를 수 있기 때문에, NPE가 발생한 지점에서 실제 NPE의 원인을 알아내기가 쉽지 않을 수도 있다는 점이다. 이런 문제를 해결할 때 도움이 될 수 있도록 JDK14부터는 좀 더 개선된 NPE 오류 메시지를 도입했다. https://openjdk.java.net/jeps/358를 살펴보라. [↑](#footnote-ref-3)
4. 옮긴이 - 다만, 이로 인해 null+null을 하면 NPE가 발생하지 않고 "nullnull"이라는 문자열이 반환된다. 또, null.toString()도 NPE를 발생시키지 않고 "null"이라는 문자열을 돌려준다. 이 두 가지 동작으로 인해 값을 문자열로 변환하는 과정에서 널 가능성을 실수로 빼먹어도 컴파일이 되는 경우가 있고 이로 인해 가끔 예기치 않은 버그가 생길 수 있다. [↑](#footnote-ref-4)
5. 옮긴이 - 코틀린 1.5부터 toUpperCase(), toLowerCase()등은 사용 금지 예고(deprecated)됐다. 대신 uppercase() 등을 써야 한다. uppercase()는 로케일과 무관하게 항상 똑같은 방식(영문 대소문자는 변환하지만 다른 문자는 그대로 두는 방식임)으로 대소문자 변환을 보장한다. [↑](#footnote-ref-5)