# 10장 애너테이션과 리플렉션

이 장에서는 두 가지 주요 주제를 다룬다. 첫 번째 부분은 코틀린 선언에 메타데이터를 엮어서 나중에 활용할 수 있게 해주는 애너테이션을 다룬다. 여러분 스스로 애너테이션을 정의하고 적용하는 방법을 설명하고, 코틀린 소스코드 컴파일에 영향을 끼치는 코틀린 내장 애너테이션에 대해 설명한다.

두 번째 부분은 리플렉션 API에 대해 다룬다. 리플렉션은 런타임에 코틀린 선언이 표현되는 방식에 따라 구성된 몇 가지 타입으로 이뤄진다. 리플렉션 객체를 얻는 방법, 리플렉션 객체의 애트리뷰트나 호출 가능한 객체를 사용하는 방법, 동적으로 함수나 프로퍼티를 호출하는 방법에 대해 설명한다.

## 구조

* 애너테이션 클래스를 정의하고 사용하는 방법
* 내장 애너테이션
* 클래스 리터럴과 호출 가능 참조
* 리플렉션 API

## 목표

코틀린 소스 코드에 애너테이션을 적용하는 방법과 여러분이 직접 애너테이션 클래스를 만드는 방법에 대해 배운다. 그리고 코틀린 리플렉션 API를 사용해 런타임에 코틀린 선언에 대한 정보를 얻고, 동적으로 함수와 프로퍼티를 호출하는 방법을 이해한다.

## 애너테이션

애너테이션은 커스텀 메타데이터를 정의하고 이 메타데이터를 소스 코드상의 선언, 식, 전체 파일 등의 요소에 엮는 방법을 제공한다. 자바 애너테이션과 마찬가지로 코틀린 애너테이션도 런타임에 접근할 수 있다. 여러 가지 프레임워크나 처리 도구들이 애너테이션을 광범위하게 사용하고 있다. 이런 도구들은 애너테이션을 사용해 코드에 필요한 정보를 채워 넣는다.

### 애너테이션 클래스 정의하고 사용하기

애너테이션을 사용하는 문법은 자바와 비슷하다. 가장 기본적인 경우는 어떤 선언의 앞쪽에 변경자 위치에 @가 붙은 애너테이션 이름을 놓는 것이다. 예를 들어 제이유닛(JUnit) 등의 테스트 프레임워크를 사용한다면 @Test라는 애너테이션을 통해 테스트 함수를 지정할 수 있다.

<코드>

class MyTestCase {  
 @Test  
 fun testOnePlusOne() {  
 assert(1 + 1 == 2)  
 }  
}

</코드>

**자바와 코틀린의 차이**: 자바와 달리 코틀린 애너테이션을 식에 적용할 수도 있다. 예를 들어 내장 @Suppress 애너테이션을 사용하면 소스 파일의 특정 식에 대한 컴파일러 경고를 끌 수 있다.

<코드>

val s = @Suppress(“UNCHECKED\_CAST”) objects as List<String>

</코드>

같은 구성 요소에 애너테이션을 여럿 붙이고 싶다면 각괄호로 애너테이션들을 감쌀 수 있다.

<코드>

@[Synchronized Strictfp] // @Synchronized @Strictfp와 같은 역할을 함  
fun main() { }

</코드>

애너테이션을 주 생성자에 적용하고 싶을 때는 명시적으로 주 생성자의 인자 목록 앞에 constructor 키워드를 붙여야 한다.

<코드>

class A @MyAnnotation constructor ()

</코드>

**4장 클래스와 객체 다루기**에서 주 생성자를 private로 정의하기 위해 이와 비슷한 구문을 사용했다.

애너테이션을 정의하기 위해서는 클래스 앞에 annotation이라는 변경자를 붙여야 한다.

<코드>

annotation class MyAnnotation  
  
@MyAnnotation fun annotatedFun() { }

</코드>

**자바와 코틀린의 차이**: 코틀린과 자바의 애너테이션 정의 차이에 유의하라. 자바 애너테이션은 인터페이스로 구성되지만 코틀린 애너테이션은 특별한 종류의 클래스로 구성된다.

일반 클래스와 달리 애너테이션 클래스에는 멤버나 부 생성자, 초기화 코드가 없다.

<코드>

annotation class MyAnnotation {  
 val text = “???” // Error  
}

</코드>

하지만 코틀린 1.3부터는 내포된 클래스, 인터페이스, 객체(동반 객체 포함)를 애너테이션 본문에 넣을 수 있다.

<코드>

annotation class MyAnnotation {  
 companion object {  
 val text = “???”  
 }  
}

</코드>

애너테이션에 커스텀 애트리뷰트를 추가하고 싶다면 생성자 파라미터를 통해야만 한다. 이와 같은 애너테이션을 사용할 때는 클래스 생성자를 호출할 때처럼 이런 파라미터에 실제 인자를 제공해야 한다.

<코드>

annotation class MyAnnotation(val text: String)  
  
@MyAnnotation(“Some useful info”) fun annotatedFun() { }

</코드>

애너테이션 파라미터를 항상 val로 선언해야 한다는 점에 유의하라.

**자바와 코틀린의 차이**: 자바 애너테이션에서는 애트리뷰트를 파라미터가 없는 메서드 형태로 지정해야 한다. 하지만 코틀린에서는 생성자 파라미터가 프로퍼티 역할을 함께 담당한다.

일반적인 생성자와 마찬가지로 디폴트 값이나 가변인자를 사용할 수도 있다.

<코드>

annotation class Dependency(var arg val componentNames: String)  
annotation class Component(val name: String = “Core”)  
  
@Component(“I/O”)  
class IO  
  
@Component(“Log”)  
@Dependency(“I/O”)  
class Logger  
  
@Component  
@Dependency(“I/O”, “Log”)  
class Main

</코드>

코틀린 애너테이션은 클래스의 일종이기는 하지만, 일반적인 클래스와 마찬가지 방식으로 이 클래스의 인스턴스를 만들 수는 없다.

<코드>

annotation class Component(val name: String = “Core”)  
  
val ioComponent = Component(“IO”) // Error

</코드>

앞에서 설명한 것처럼 @ 구문을 사용해야만 애너테이션 인스턴스를 생성할 수 있다. (런타임까지 유지되는 애너테이션인 경우) 실제 애너테이션 인스턴스를 얻기 위해서는 리플렉션 API를 사용할 수 있다. 리플렉션에 대해서는 나중에 다룰 것이다.

애너테이션 클래스에는 상위타입을 명시할 수도, 애너테이션 클래스를 상속하는 클래스를 정의할 수도 없다. 애너테이션은 Any 클래스와 빈 Annotation 인터페이스를 자동으로 상속하며, 이 둘은 모든 애너테이션 클래스의 공통 상위타입 역할을 한다.

애너테이션 인자는 컴파일 시에만 평가되기 때문에 애너테이션 인자에 임의의 식을 넣을 수는 없다. 더 나아가 컴파일러는 애너테이션 파라미터로 사용할 수 있는 타입의 종류를 다음과 같이 제한한다.

* Int, Boolean, Double 등 원시 타입
* String
* 이넘
* 다른 애너테이션
* 클래스 리터럴
* 위에 나열한 타입들로 이뤄진 배열

JVM에서는 애너테이션 애트리뷰트에 널을 저장할 수 없기 때문에 이와 같은 파라미터는 널이 될 수 없다.

다른 애너테이션을 인자로 사용하는 경우에는 @ 접두사를 안 붙여도 된다. 대신 일반 생성자 호출처럼 애너테이션을 써라. 앞의 예제를 다시 살펴보자.

<코드>

annotation class Dependency(vararg val componentNames: String)  
  
annotation class Component(  
 val name: String = “Core”,  
 val dependency: Dependency = Dependency()  
)  
  
@Component(“I/O”)  
class IO  
  
@Component(“Log”, Dependency(“I/O”))  
class Logger  
  
@Component(dependency = Dependency(“I/O”, “Log”))  
class Main

</코드>

애너테이션 파라미터로 vararg 대신 명시적인 배열 타입을 사용할 수도 있다. 이와 같은 애너테이션을 쓸 때는 표준 arrayOf() 함수를 통해 배열을 만들 수 있다.

<코드>

annotation class Dependency(val componentNames: Array<String>)  
  
@Component(dependency = Dependency(arrayOf(“I/O”, “Log”)))  
class Main

</코드>

코틀린 1.2부터는 애너테이션 인자로 각괄호([])를 사용해 더 간결하게 배열을 만들 수 있다.

<코드>

annotation class Dependency(val componentNames: Array<String>)  
  
@Component(dependency = Dependency([“I/O”, “Log”]))  
class Main

</코드>

각괄호를 사용한 배열 표현의 경우 현재는 애너테이션에서만 사용할 수 있다.

클래스 리터럴을 사용하면 KClass 타입의 리플렉션 객체로 클래스에 대한 표현을 얻을 수 있다. 이 KClass 타입은 자바 언어의 Class 타입에 해당하는 코틀린 클래스다. 클래스 이름 뒤에 ::class를 붙여서 클래스 리터럴을 만든다. 앞의 컴포넌트/의존관계 예제를 이름 대신 클래스 리터럴을 사용하는 형태로 바꾸자.

<코드>

import kotlin.reflect.KClass  
  
annotation class Dependency(vararg val componentClasses: KClass<\*>)  
  
annotation class Component(  
 val name: String = “Core”,  
 val dependency: Dependency = Dependency()  
)  
  
@Component(“I/O”)  
class IO  
  
@Component(“Log”, Dependency(IO::class))  
class Logger  
  
@Component(dependency = Dependency(IO::class, Logger::class))  
class Main

</코드>

**자바와 코틀린의 차이**: java.lang.Class의 인스턴스를 코틀린 애너테이션에 쓸 수도 있다. 하지만 JVM을 타깃으로 코틀린 클래스를 컴파일한 경우에는 자바 클래스로 자동 변환된다.

코틀린 소스 코드에서 여러 언어 요소가 함축되어 있는 선언에 대해 애너테이션이 붙을 수도 있다. 예를 들어 다음 클래스를 생각해 보자.

<코드>

class Person(val name: String)

</코드>

이 코드의 val name: String은 생성자 파라미터와 게터가 있는 클래스 프로퍼티, 그리고 프로퍼티 값을 저장하기 위한 뒷받침하는 필드 선언을 짧게 줄인 코드다. 이런 요소 각각에 대해 애너테이션을 붙일 수 있기 때문에, 코틀린에서는 애너테이션을 사용하는 시점에 어떤 대상에 대해 애너테이션을 붙이는지 지정할 수 있다.

이런 사용 지점 대상은 특별한 키워드를 통해 지정한다. 이 키워드는 애너테이션 이름 앞에 붙으며 :로 애너테이션 이름과 구분된다. 예를 들어 프로퍼티 게터에 대한 에너테이션을 붙이고 싶다면 get 키워드를 사용하면 된다.

<코드>

class Person(@get:A val name: String)

</코드>

대부분의 사용 지점 대상은 다양한 프로퍼티 구성 요소와 연관되어 있다. 최상위나 클래스 수준의 프로퍼티는 물론이고, 주 생성자의 val/var 파라미터에 대해서도 이런 대상을 지정할 수 있다.

* property: 프로퍼티 자체를 대상으로 한다
* field: 뒷받침하는 필드를 대상으로 한다(뒷받침하는 필드가 있는 프로퍼티에 대해서만 작동한다)
* get: 프로퍼티 게터를 대상으로 한다
* set: 프로퍼티 세터를 대상으로 한다
* param: 생성자 파라미터를 대상으로 한다(val/var가 붙은 파라미터만 대상으로 삼을 수 있다)
* setparam: 프로퍼티 세터의 파라미터를 대상으로 한다(가변 프로퍼티에만 사용할 수 있다)
* delegate: 위임 객체를 저장하는 필드를 대상으로 한다(위임 프로퍼티에만 사용할 수 있다. **11장 도메인 특화 언어**을 보라)

코드상에 명시적으로 프로퍼티 접근자가 없더라도, get/set 대상(바로 앞 예제의 val 파라미터를 보라)을 사용하면 접근자에 애너테이션을 붙일 수 있다. setparam 대상도 마찬가지다. 이 경우에는 세터의 파라미터에 대해 애너테이션을 직접 붙인 것과 마찬가지 역할을 한다.

사용 지점 대상을 사용하는 애너테이션을 [] 구문으로 묶을 수 있다. 이런 경우 대상을 모든 애너테이션에 적용하게 된다. 따라서 다음과 같은 정의는,

<코드>

class Person(@get:[A B] val name: String)

</코드>

다음과 같다.

<코드>

class Person(@get:A @get:B val name: String)

</코드>

receiver라는 대상을 사용하면 확장 함수나 프로퍼티의 수신 객체에 애너테이션을 붙일 수 있다.

<코드>

class Person(val firstName: String, val familyName: String)  
  
fun @receiver:A Person.fullName() = “$firstName $familyName”

</코드>

마지막으로 file이라는 대상을 사용해 전체 파일에 대해 애너테이션을 붙일 수 있다. 코틀린 파일 맨 앞, 다른 임포트나 패키지 지시자보다 더 앞에 이런 애너테이션을 붙일 수 있다.

<코드>

@file:JvmName(“MyClass”)  
fun main() {  
 println(“main() in MyClass”)  
}

</코드>

런타임에 최상위 함수와 프로퍼티가 들어가는 파일 퍼사드 클래스 안에 파일 애너테이션이 남는다. **12장 자바 상호 운용성**에서는 (앞의 @JvmName처럼) 퍼사드 클래스를 자바에서 볼 수 있게 할 수 있는 파일 수준의 애너테이션을 살펴본다.

이제 몇 가지 내장 애너테이션을 살펴보자. 이런 애너테이션들은 코틀린 코드 안에서 특별한 의미를 가진다.

## 내장 애너테이션

코틀린은 몇 가지 내장 애너테이션을 제공한다. 이들은 컴파일러 수준에서 특별한 의미를 가진다. 이런 애너테이션 중 일부는 애너테이션 클래스 자체에 적용 가능하며, 이런 애너테이션을 통해 대상 애너테이션의 사용 방법을 바꿀 수 있다. 이런 애너테이션 중 상당수는 자바 언어에 있는 비슷한 메타 애너테이션 역할을 한다.

@Retention 애너테이션은 애너테이션이 저장되고 유지되는 방식을 제어한다. 자바의 @Retention 인터페이스와 마찬가지로, 코틀린에서도 AnnotationRetention 이넘 클래스에 적용된 세 가지 중 한 가지 저장 방식을 지정할 수 있다.

* SOURCE: 이 애너테이션은 컴파일 시점에만 존재하며 컴파일러의 바이너리 출력(JVM의 경우 바이트 코드가 저장된 클래스 파일)에는 저장되지 않는다.
* BINARY: 이 애너테이션은 컴파일러의 바이너리 출력에 저장되지만, 런타임에 리플렉션 API로 관찰할 수는 없다.
* RUNTIME: 이 애너테이션은 컴파일러의 바이너리 출력에 저장되며 런타임에 리플렉션 API를 통해 관찰할 수도 있다.

디폴트로 코틀린 애너테이션은 RUNTIME으로 유지 시점이 정의된다. 따라서 리플렉션 API에서 코틀린 애너테이션을 찾을 수 없을 염려는 없다. 하지만 현재는 식에 대해 붙은 애너테이션의 경우 런타임까지 유지되지 못한다는 점에 유의하라. 따라서 식에 붙는 애너테이션에 대해 BINARY나 RUNTIME을 지정하는 것은 금지되어 있다.

<코드>

@Target(AnnotationTarget.EXPRESSION)  
annotation class NeedToRefactor // Error: SOURCE만 허용함

</코드>

이런 경우에는 명시적으로 SOURCE 유지 시점을 지정해야 한다.

<코드>

@Target(AnnotationTarget.EXPRESSION)  
@Retention(AnnotationRetention.SOURCE)  
annotation class NeedToRefactor // Ok

</코드>

**자바와 코틀린의 차이**: 자바와 코틀린의 디폴트 유지 시점 차이를 기억하라. 자바에서는 디폴트가 RetentionPolicy.CLASS(이는 코틀린의 AnnotationRetention.BINARY)이며, 이 말은 명시적으로 RUNTIME으로 지정하지 않으면 자바 애너테이션을 리플렉션에서 관찰할 수 없다는 뜻이다.

@Repeatable은 애너테이션을 같은 언어 요소에 두 번 이상 반복 적용할 수 있다.

<코드>

@Repeatable  
@Retention(AnnotationRetention.SOURCE)  
annotation class Author(val name: String)  
  
@Author(“John”)  
@Author(“Harry”)  
class Services

</코드>

기본적으로 애너테이션을 반복 적용할 수는 없다. 반복 적용할 수 없는 애너테이션을 여러 번 적용하면 오류가 발생한다.

<코드>

@Deprecated(“Deprecated”)  
@Deprecated(“Even more deprecated”) // Error: 반복 불가능한 애너테이션  
class OldClass

</코드>

현재는 반복할 수 있는 애너테이션을 런타임까지 유지할 수 없다. 따라서 반복할 수 있는 애너테이션의 유지 시점을 꼭 SOURCE로 명시해야 한다.

@MustBeDocumented specifies that the annotation must be included into the documentation, since it is considered to be a part of the public API. This annotation plays the same role as Java’s @Documented and is supported by Dokka, the standard Kotlin documentation engine (the way @Documented is supported by Javadoc tool).

@Target은 애너테이션을 어떤 언어 요소에 붙일 수 있는지 지정한다. AnnotationTarget 이넘에 정의된 다음 상수들을 vararg로 지정하면 된다.

* CLASS: 클래스, 인터페이스, 객체에 붙일 수 있다(애너테이션 클래스도 포함).
* ANNOTATION\_CLASS: 애너테이션 클래스에 붙일 수 있다
* TYPEALIAS: 타입 별명 정의에 붙일 수 있다
* PROPERTY: 주 생성자에 정의된 val/var 프로퍼티를 포함해, 프로퍼티에 붙일 수 있다(로컬 변수에는 붙일 수 없다)
* FIELD: 프로퍼티를 뒷받침하는 필드에 붙일 수 있다
* LOCAL\_VARIABLE: 지역 변수에 붙일 수 있다(파라미터는 제외)
* VALUE\_PARAMETER: 생성자, 함수, 프로퍼티 세터의 파라미터에 붙일 수 있다
* CONSTRUCTOR: 주 생성자나 부 생성자에 붙일 수 있다
* FUNCTION: 람다나 익명 함수를 포함해, 함수에 붙일 수 있다(하지만 생성자나 프로퍼티 접근자에는 붙일 수 없다)
* PROPERTY\_GETTER/PROPERTY\_SETTER: 프로퍼티 게터/프로퍼티 세터에 붙일 수 있다
* FILE: 파일에 붙일 수 있다
* TYPE: 타입 지정에 붙일 수 있다. 변수의 타입이나 함수의 파라미터 타입, 반환 타입 등을 포함한다
* EXPRESSION: 식에 붙일 수 있다

TYPE\_PARAMETER라는 상수를 현재는 지원하지 않지만, 미래에 대비해 예약되어 있다. 따라서 제네릭 선언의 타입 파라미터에 애너테이션을 적용할 수는 없다.

@Target을 지정하지 않으면 타입 별명, 타입 파라미터, 타입 지정, 식, 파일을 제외한 언어 요소에 애너테이션을 적용할 수 있다. 예를 들어 파일에 적용할 수 있는 애너테이션을 원한다면 이를 명시해야만 한다.

**자바와 코틀린의 차이**: AnnotationTarget 클래스는 JDK에 있는 ElementType 이넘과 비슷하다. 하지만 TYPE이라는 상수가 서로 다른 의미라는 점에 유의하라. 코틀린의 AnnotationTarget.TYPE은 타입 지정(자바의 ElementType.TYPE\_USAGE에 해당)을 뜻하지만, 자바의 ElementType.TYPE은 실제 클래스나 인터페이스 선언(코틀린의 AnnotationTarget.CLASS에 해당)을 뜻한다.

그리고 자바와 달리 코틀린은 패키지 수준의 애너테이션을 지원하지 않는다(따라서 ElementType.PACKAGE에 대응하는 상수는 없다). 하지만 소스 파일 수준에서 애터네이션을 정의할 수 있다. **12장 자바 상호 운용성**에서 자바-코틀린 상호 운용성을 세밀하게 조정하기 위해 파일 애너테이션을 사용하는 방법에 대해 살펴본다.

다음 애너테이션은 이름이 같은 자바 변경자와 같은 역할을 한다.

* @StrictFp: 부동소수점 연산의 정밀도를 제한해서 여러 다른 플랫폼간의 부동소수점 연산 이식성을 높여준다
* @Synchronzied: 애너테이션이 붙은 함수나 프로퍼티 접근자의 본문에 진입하기 전에 모니터(monitor)를 획득하고 본문 수행 후 모니터를 해제하게 한다
* @Volatile: 애너테이션이 붙은 뒷받침하는 필드를 변경한 내용을 즉시 다른 스레드에서 관찰할 수 있게 해준다
* @Transient: 애네테이션이 붙은 필드를 직렬화 메커니즘이 무시한다

@Synchronized와 @Volatile은 동시성 지원과 관련이 있기 때문에 **13장 동시성**에서 자세히 다룬다.

@Suppress 애너테이션을 사용하면 지정한 이름의 컴파일러 경고를 표시하지 않게 할 수 있다. 이 애너테이션은 식이나 파일을 포함하는 모든 대상에 붙일 수 있다. 예를 들어 여러분이 코드가 맞다고 확신한다면 이 애너테이션을 사용해 타입 캐스팅과 관련한 불필요한 경고를 막을 수 있다.

<코드>

val strings = listOf<Any>(“1”, “2”, “3”)  
val numbers = listOf<Any>(1, 2, 3)  
  
// 경고 표시되지 않음  
val s = @Suppress(“UNCHECKED\_CAST”) (strings as List<String>)[0]  
  
// Unchecked cast 경고 표시됨  
val n = (numbers as List<Number>)[1]

</코드>

이 애너테이션은 자신이 적용된 요소 내부에 있는 모든 코드에 적용된다. 예를 들어 어떤 함수의 코드 안쪽에서 발생하는 모든 경고를 제거할 수도 있다.

<코드>

@Suppress(“UNCHECKED\_CAST”)  
fun main() {  
 val strings = listOf<Any>(“1”, “2”, “3”)  
 val numbers = listOf<Any>(1, 2, 3)  
 val s = (strings as List<String>)[0] // 경고 표시되지 않음  
 val n = (numbers as List<Number>)[1] // 경고 표시되지 않음  
   
 println(s + n) // 12  
}

</코드>

파일을 사용 지점 대상으로 지정하면 파일 전체에서 경고를 없앨 수도 있다.

<코드>

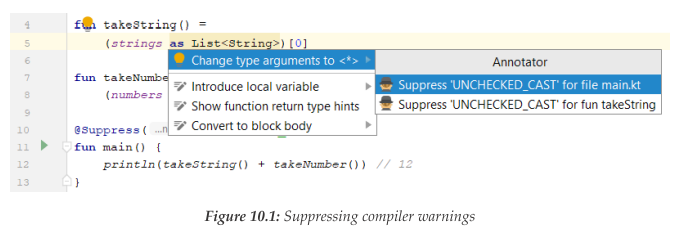
@file:Suppress(“UNCHECKED\_CAST”)  
val strings = listOf<Any>(“1”, “2”, “3”)  
val numbers = listOf<Any>(1, 2, 3)  
  
fun takeString() = (strings as List<String>)[0] // 경고 표시되지 않음  
fun takeNumber() = (numbers as List<Number>)[1] // 경고 표시되지 않음  
  
@Suppress(“UNCHECKED\_CAST”)  
fun main() {  
 println(takeString() + takeNumber()) // 12  
}

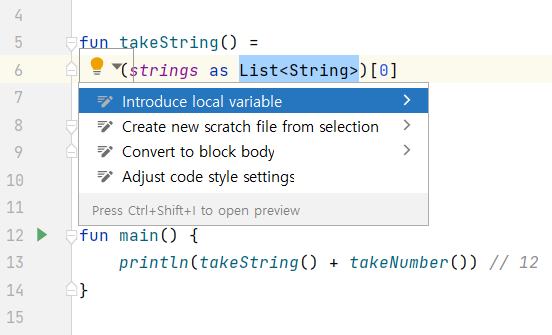
</코드>

<참고> IDE 팁

경고 이름을 찾아보거나 외울 필요는 없다. 인텔리J가 @Suppress 애너테이션을 자동으로 삽입할 수 있다. 경고가 발생할 수 있는 지점에 캐럿을 위치시킨 다음 [Alt]+[Enter]를 누른 후(그림 10-1), Annotator 하위 메뉴의 Suppress... 액션 중 하나를 선택하라. IDE가 코드를 검사해 보고하는 경고에 대해서도 같은 액션을 사용할 수 있다.

그림 10-1: 컴파일러 경고 없애기

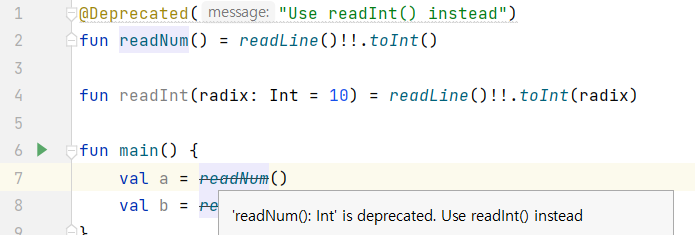




</참고>

다른 유용한 애너테이션인 @Deprecated는 자바의 같은 애너테이션과 같은 역할을 한다. 어떤 선언을 사용 금지 예정(deprecated)이라고 선언하면 이 선언을 사용하지 말라고 클라이언트 코드에게 권장한다. IDE에서 사용 금지 예정인 선언을 사용하면 취소선이 들어간 글자체로 표시된다(그림 10-2와 같음). @Deprecated를 사용할 때는 왜 이 선언이 사용 금지 예정인지 알려주고, 이 선언 대신 쓸 수 있는 대안을 알려주는 메시지를 추가하는 것이 일반적이다.

그림 10-2: 사용 금지 예정으로 지정된 선언



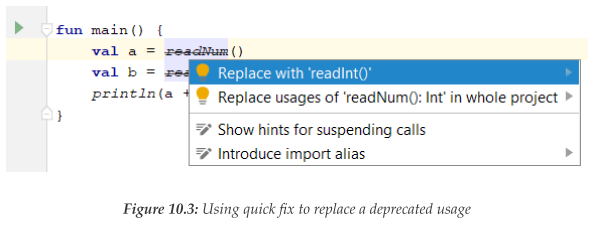
코틀린 @Deprecated는 자바에서 제공하지 않는 다른 기능도 제공한다. 이 기능을 사용하려면 사용 금지 예정 대상을 대신할 수 있는 식을 문자열로 지정해야 한다. 사용 금지 예정인 선언을 사용하는 코드에서 [Alt]+[Enter]를 눌러 퀵픽스를 시도하면 사용 금지된 식을 이 문자열에 해당하는 식으로 변경해준다(그림 10-3). 앞에서 본 readNum()을 readInt()로 대신하고 싶다고 하자. 이런 경우 다음과 같이 코드를 작성할 수 있다.

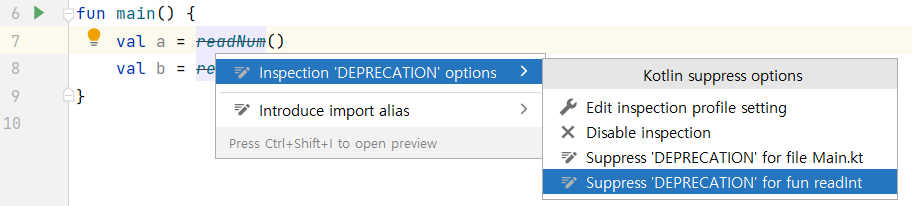
<코드>

@Deprecated(  
 “Use readInt() instead”, // 메시지  
 ReplaceWith(“readInt()”) // 대안  
)  
  
fun readNum() = readLine()!!.toInt()

</코드>

그림 10-3: 사용 금지 예정인 부분을 퀵픽스하기





ReplaceWith도 애너테이션이라는 점에 유의하라. 그래서 @Deprecated를 사용하면서 그 안에 RepalceWith를 지정할 수 있다. 하지만 @ReplaceWith를 단독으로 사용할 수는 없다. 이 애너테이션의 정의를 살펴보자.

<코드>

@Target()  
@Retention(BINARY)  
@MustBeDocumented  
public annotation class ReplaceWith(  
 val expression: String,  
 vararg val imports: String  
)

</코드>

코드를 보면 알 수 있듯이 이 애너테이션은 아무 대상도 지원하지 않는다. 따라서 @Deprecated와 같은 다른 애너테이션 내부에서만 이 애너테이션을 사용할 수 있다.

ReplaceWith가 추가로 받는 vararg 파라미터는 사용 금지 예정인 식을 대신할 때 필요한 임포트 문 목록을 지정하기 위한 것이다. 대체할 코드가 디폴트로 임포트되는 패키지나 사용 금지 예정인 식이 속한 패키지 안에 있지 않다면 이 파라미터가 유용하다.

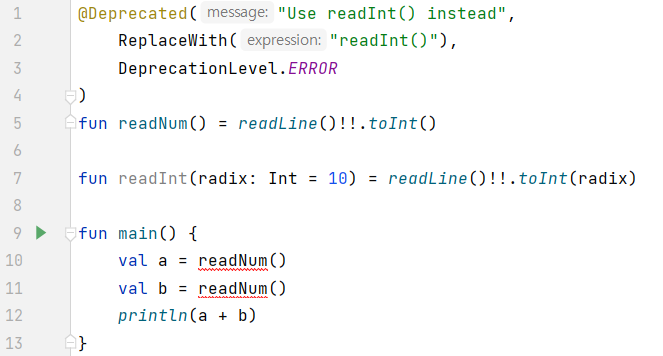
다른 기능으로, 사용 금지 예정의 심각성을 지정할 수도 있다. 이때 DeprecationLevel 이넘을 사용한다.

* WARNING: 사용 금지 예정이 붙은 선언을 사용하면 경고를 표시한다. 이 동작이 디폴트다
* ERROR: 사용 금지 예정이 붙은 선언을 사용하면 컴파일 오류로 처리한다
* HIDDEN: 사용 금지 예정이 붙은 선언에 점근하지 못하게 막는다

심각성 수준을 지정함으로써 사용 금지 정책을 유연하게 설정할 수 있다. 팀 개발 시 이런 유연성이 중요하다. 처음에는 디폴트 수준에서 선언을 사용 금지 예정으로 지정해서 기존의 용례에 대해 경고가 표시되게 한다. 이를 통해 기존 코드를 컴파일하지 못하는 일 없이 다른 개발자들이 기존 코드를 변경할 수 있는 기회를 제공한다. 그 후 사용 금지 예정 수준을 ERROR로 올린다. 이제부터는 사용 금지된 코드를 새로 도입할 수 없게 된다. 누구도 기존 방식을 사용하지 않는다고 확신한 다음에는 안전하게 코드 기반(code base)에서 해당 선언을 제거할 수 있다.

그림 10-4는 readNum() 함수의 심각성을 ERROR로 정의한 경우를 보여준다.

그림 10-4: ERROR 수준으로 정의한 사용 금지 예정 선언



자바/코틀린 상호 운용성을 조정하기 위해 @Throws, @JvmName, @NotNull 등의 애너테이션을 사용한다. 이에 대해서는 **12장 자바 상호 운용성**에서 다룰 것이다.

## 리플렉션

리플렉션 API는 클래스, 함수, 프로퍼티의 런타임 표현에 접근할 수 있게 해주는 타입, 함수, 프로퍼티 모음이다. 여러분이 작성한 프로그램이 컴파일 시점에 알 수 없는 클래스를 다뤄야 하는데, 어떤 정해진 공통의 계약을 준수해야만 하는 경우에 리플렉션이 유용하다. 예를 들어 플러그인으로 클래스를 동적으로 적재할 수 있고, 멤버들의 시그니처를 알면 이렇게 적재한 클래스에 속한 멤버를 호출할 수 있다.

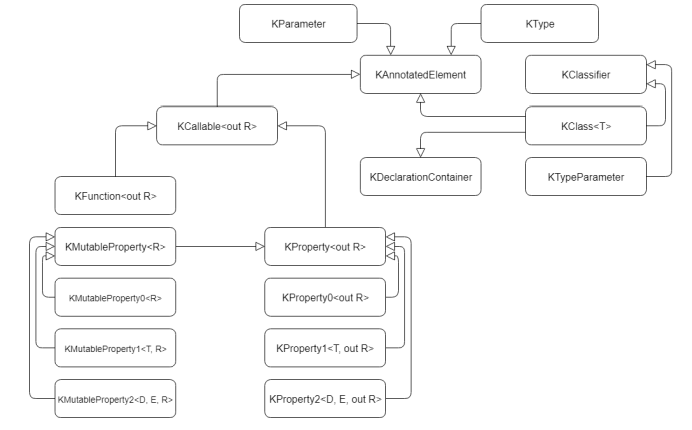
이제부터는 코틀린 리플렉션 API를 구성하는 여러 요소와 각각의 사용법에 대해 설명한다.

**자바와 코틀린의 차이**: 코틀린 리플렉션 자체로는 완결성이 없음에 유의하라. 클래스 검색이나 적재와 같은 몇몇 기능은 자바 리플렉션 API가 제공하는 기능을 사용해야 한다. 코드에서 코틀린에 관련된 측면(프로퍼티나 객체 등)을 처리할 때는 코틀린 API를 사용하면 런타임에 훨씬 더 깔끔하고 코틀린답게 해당 기능에 접근할 수 있다.

### 리플렉션 API 개요

리플렉션 관련 클래스는 kotlin.reflect 패키지에 들어있고, 크게 두 가지 그룹으로 나눌 수 있다. 호출 가능 그룹은 프로퍼티와 함수(생성자 포함)를 표현하며, 지정자 그룹은 클래스나 타입 파라미터의 런타임 표현을 제공한다. 그림 10-5는 기본적인 리플렉션 타입들을 보여준다.

그림 10-5: 기본적인 리플렉션 타입들



모든 리플렉션 타입은 KAnnotatedElement의 자손이다. KAnnotatedElement는 함수, 프로퍼티, 클래스 등 구체적인 언어 요소에 정의된 애너테이션에 접근하는 기능을 제공한다. KAnnotatedElement에는 프로퍼티가 하나뿐이다. 이 프로퍼티는 애너테이션 인스턴스로 이뤄진 리스트다.

<코드>

public val annotations: List<Annotation>

</코드>

앞에서 본 @Component/@Dependency 예제를 다시 살펴보자.

<코드>

import kotlin.reflect.KClass  
  
annotation class Dependency(vararg val componentClasses: KClass<\*>)  
  
annotation class Component(  
 val name: String = “Core”,  
 val dependency: Dependency = Dependency()  
)  
  
@Component(“I/O”)  
class IO  
  
@Component(“Log”, Dependency(IO::class))  
class Logger  
  
@Component(dependency = Dependency(IO::class, Logger::class))  
class Main

</코드>

이제 Main 클래스와 연관된 애너테이션을 가져오고 싶다. 클래스 리터럴의 annotations 프로퍼티를 통해 정보를 얻을 수 있다.

<코드>

fun main() {  
 val component = Main::class.annotations  
 .filterIsInstance<Component>()  
 .firstOrNull() ?: return  
   
 println(“Component name: ${component.name}”)  
   
 val depText = component.dependency.componentClasses  
 .joinToString { it.simpleName ?: “” }  
   
 println(“Dependencies: $depText”)  
}

</코드>

이 코드를 실행하면 다음이 출력된다.

<코드>

Component name: Core  
Dependencies: IO, Logger

</코드>

이제 리플렉션 API에서 지정자와 호출가능에 관련한 여러 구체적인 타입을 살펴보자.

### 지정자와 타입

코틀린 리플렉션에서 지정자는 타입을 정의하는 선언을 뜻한다. 이런 선언은 KClassifier 인터페이스에 의해 표현되며, 이 인터페이스에는 두 가지 구체적인 변종이 있다.

* KClass<T>는 컴파일 시점에 T 타입인 클래스나 인터페이스, 객체 선언을 런타임에 표현한다.
* KTypeParameter는 어떤 제네릭 선언의 타입 파라미터를 표현한다.

현재 타입 별명을 표현하는 리플렉션 API가 없다는 점에 유의하라. 특히 여러분이 타입 별명에 애너테이션을 적용해도 런타임에 이를 얻을 수 없다는 뜻이다. 타입 별명에 대한 리플렉션 지원은 향후 코틀린에 추가될 예정이다.

KClassifier 자체에는 아무 멤버도 들어있지 않기 때문에, 클래스를 표현하는 KClass<T>와 타입 파라미터를 표현하는 KTypeParameter 각각의 세부 사항을 바로 살펴보자.

KClass 인스턴스를 얻는 방법은 두 가지다. 첫 번째는 애너테이션에 대해 설명할 때 이야기한 클래스 리터럴 구문을 사용하는 방법이다.

<코드>

println(String::class.isFinal) // true

</코드>

이 구문은 클래스뿐 아니라 구체화한 타입 파라미터도 지원한다. **9장 제네릭스**에서 제네릭 인라인 함수의 타입 파라미터를 구체화할 수 있다고 설명했다. 이 말은 컴파일러가 함수 호출 지점에 함수 본문을 인라인해주면서 이런 타입 파라미터의 타입을 실제 타입으로 대치해 준다는 뜻이다. 예를 들어 cast() 함수를 정의해 보자.

<코드>

inline fun <reified T> Any.cast() = this as? T

</코드>

이 함수를 다음과 같이 호출한다.

<코드>

val obj: Any = “Hello”  
println(obj.cast<String>())

</코드>

내부에서 컴파일러는 다음과 같은 코드를 생성한다.

<코드>

val obj: Any = “Hello”  
println(obj as? String)

</코드>

::class 구문을 사용하면 임의의 식의 결괏값에 대한 런타임 클래스를 얻을 수 있다.

<코드>

println((1 + 2)::class) // class kotlin.Int  
println(“abc”::class) // class kotlin.String

</코드>

KClass를 얻는 다른 방법은 kotlin 확장 프로퍼티를 사용해 java.lang.Class의 인스턴스를 KClass로 변환하는 것이다. 전체 이름을 가지고 클래스를 동적으로 찾을 때 이런 방법이 유용하다. 코틀린 리플렉션은 클래스를 검색하는 API를 제공하지 않기 때문에 플랫폼에 따라 적절한 클래스 검색 방식을 사용해야 한다.

<코드>

val stringClass = Class.forName(“java.lang.String”).kotlin  
println(stringClass.isInstance(“Hello”)) // true

</코드>

java 확장 프로퍼티를 사용하면 반대 방향으로도 변환이 가능하다.

<코드>

println(String::class.java) // class java.lang.String

</코드>

이제 KClass API를 살펴보자. KClass의 멤버 중 첫 번째 그룹은 대상 클래스에 어떤 변경자가 붙어있는지를 알아낼 수 있게 해준다.

<코드>

val isAbstract: Boolean  
val isCompanion: Boolean  
val isData: Boolean  
val isFinal: Boolean  
val isInner: Boolean  
val isOpen: Boolean  
val isSealed: Boolean

</코드>

이 그룹에 속한 visibility라는 프로퍼티는 KVisibility 이넘으로 클래스 선언의 가시성 수준을 돌려준다.

<코드>

enum class Kvisibility {  
 PUBLIC,  
 PROTECTED,  
 INTERNAL,  
 PRIVATE  
}

</코드>

코틀린 소스 코드에서 가시성을 표현할 수 없다면 visibility 값이 null이라는 점에 유의하라. 예를 들어 지역 클래스를 표현하는 KClass의 visibility는 null이다.

프로퍼티 중 다음 그룹은 클래스 이름을 제공한다.

<코드>

val simpleName: String?  
val qualifiedName: String?

</코드>

simpleName 프로퍼티는 소스 코드에서 사용되는 간단한 이름을 반환한다. 클래스 이름이 없다면(예: 객체 식을 표현하는 클래스) 결과는 null이다.

마찬가지로 qualifieName 프로퍼티를 사용하면 클래스의 전체 이름을 얻을 수 있다. 전체 이름에는 클래스가 포함된 패키지의 전체 경로가 들어간다. 클래스가 로컬 클래스이거나 로컬 클래스 안에 내포된 클래스인 경우, 최상위 경로에서 접근할 방법이 없기 때문에 이런 클래스에는 전체 이름이 없다. 따라서 이 프로퍼티는 null을 돌려준다. 소스 코드에서 이름이 없는 클래스의 경우도 마찬가지로 qualifieName이 null을 돌려준다.

jvmName 확장 프로퍼티를 사용하면 자바 관점에서 보는 클래스 전체 이름을 돌려준다. 이 이름은 qualifieName이 돌려주는 이름과 다를 수 있다. 일부 내장 코틀린 타입은 상응하는 JVM 표현이 없고 자바 클래스에만 의존하는 경우도 있다. 예를 들어 Any 클래스는 독립적인 자바 클래스로 존재하지 않고, JVM 환경에서는 java.lang.Object와 같다.

<코드>

println(Any::class.qualifiedName) // kotlin.Any  
println(Any::class.jvmName) // java.lang.Object

</코드>

isInstance() 함수는 주어진 객체가 이 함수의 수신 객체가 표현하는 클래스의 인스턴스인지 알려준다. 널이 될 수 없는 타입에 대해 이 함수를 호출하면 is 연산과 비슷하게 작동한다.

println(String::class.isInstance(“”)) // true  
println(String::class.isInstance(12)) // false  
println(String::class.isInstance(null)) // false

KClass에 속한 프로퍼티들이 속한 다음 그룹은 멤버 선언에 대한 접근을 제공한다.

* constructors: 주 생성자와 부 생성자들을 KFunction 타입의 인스턴스로 돌려준다
* members: KCallable 인스턴스로 표현되는 멤버 함수와 프로퍼티 표현의 컬렉션을 돌려준다. 이 컬렉션 안에는 상위 타입에서 상속한 모든 멤버도 함께 포함된다
* nestedClasses: 내포된 클래스와 객체들로 이뤄진 컬렉션이다. 동반 객체도 포함된다.
* typeParameters: KTypeParameter에 의해 표현되는 타입 파라미터로 이뤄진 리스트다(대상 클래스가 제네릭 타입이 아닌 경우 이 리스트는 빈 리스트다)

예를 들어 다음 코드는 리플렉션을 사용해 Person 클래스의 인스턴스를 만들고, 그 인스턴스의 fullName() 함수를 호출한다.

<코드>

class Person(valfirstName: String, val familyName: String) {  
 fun fullName(familyFirst: Boolean): String = if (familyFirst) {  
 “$familyName $firstName”  
 } else {  
 “$firstName $familyName”  
 }  
}  
  
fun main() {  
 val personClass = Class.forName(“Person”).kotlin  
 val person = personClass.constructors.first().call(“John”, “Doe”)  
 val fullNameFun = personClass.members.first { it.name == “fullName” }  
   
 println(fullNameFun.call(person, false)) // John Doe  
}

</코드>

KClass가 객체 선언을 표현하는 경우 constructors 프로퍼티는 항상 빈 컬렉션을 반환한다. 실제 인스턴스를 얻고 싶으면 objectInstance 프로퍼티를 사용해야 한다.

<코드>

object O {  
 val text = “Singleton”  
}  
  
fun main() {  
 println(O::class.objectInstance!!.text) // Singleton  
}

</코드>

KClass 인스턴스가 객체를 표현하지 않으면 objectInstance 프로퍼티도 null이다.

마지막으로 봉인된 클래스(isSealed == true)의 경우 sealedSubclasses 프로퍼티를 통해 직접적인 상속자로 이뤄진 리스트를 얻을 수 있다.

KClass에서 얻을 수 있는 다른 종류의 정보로 supertypes 프로퍼티를 통해 얻을 수 있는 KType 인스턴스의 리스트를 들 수 있다. 뒤에서 KType API에 대해 설명하므로 여기서는 간단한 예제만 살펴보자.

<코드>

open class GrandParent  
open class Parent :GrandParent()  
interface IParent  
class Child : Parent(), IParent  
  
fun main() {  
 println(Child::class.supertypes) // [Parent, IParent]  
}

</코드>

supertype 프로퍼티는 클래스가 직접 상속한 상위 타입만 돌려준다(따라서 예제의 출력에 GrandParent가 빠져 있다). 따라서 간접적인 상위 클래스도 모두 포함하고 싶다면 별도의 상속 그래프 순회를 수행해야 한다.

KClassifier의 변종 중 KClass외의 것으로 KTypeParameter 인터페이스로 표현되는 정보가 있다. KClass와 비교할 때 KTypeParameter는 상당히 단순하며 프로퍼티를 4개만 제공한다.

<코드>

val isReified: Boolean  
val name: String  
val upperBounds: List<KType>  
val variance: KVariance

</코드>

upperBounds는 KClass의 supertypes 프로퍼티와 비슷하게 상위 바운드 타입으로 이뤄진 리스트를 돌려준다. 모든 타입 파라미터에는 최소한 디폴트 바운드인 Any?가 바운드로 있기 때문에, 이 리스트는 절대로 빈 리스트가 될 수 없다. 그리고 타입 파라미터를 타입 제약에 사용하는 경우에는 바운드가 둘 이상일 수도 있다. 예를 살펴보자.

<코드>

interface MyMap<K : Any, out V>  
  
fun main() {  
 val parameters = MyMap::class.typeParameters  
 // K: [kotlin.Any], V: [kotlin.Any?]  
 println(parameters.joinToString { “${it.name}: ${it.upperBounds}” })  
}

</코드>

variance 프로퍼티는 KVariance 이넘으로 변성을 돌려준다. 이 이넘은 타입 파라미터의 선언 지점 변성 종류를 표현한다.

<코드>

enum class KVariance{ INVARIANT, IN, OUT }

</코드>

이제 코틀린 리플렉션이 KType 인터페이스를 통해 어떻게 타입을 표현하는지 살펴보자. 코틀린 타입은 다음과 같은 세 가지 성격을 지닌다.

* isMarkedNullable 프로퍼티가 제공하는 널 가능성. 예를 들면 이를 통해 List<String>와 List<String>?를 구분할 수 있다
* classifier 프로퍼티를 통해 제공하는 지정자. 지정자는 타입을 정의하는 클래스나 인터페이스나 객체 선언을 가리킨다. 예를 들어 List<String>에서 List 부분을 가리키는 리플렉션 요소가 지정자다
* 타입 프로퍼티에 전달된 실제 타입 인자 리스트. 예를 들어 List<String>이면 <String>, Map<Int, Boolean>이면 <Int, Boolean>이 타입 인자 리스트다

타입 인자가 타입 자체와 타입의 사용 지점 변성을 함께 포함하는 KTypeProjection 인터페이스에 의해 표현될 수도 있다.

<코드>

val type: kotlin.reflect.KType?  
val variance: kotlin.reflect.KVariance?

</코드>

스타 프로젝션의 경우 두 프로퍼티 모두 null을 돌려준다.

여기까지가 지정자 타입에 대한 소개다. 이제부터는 리플렉션 API에서 호출가능 부분에 대해 살펴보겠다.

### 호출가능

호출가능한(callable) 요소라는 개념은 어떤 결과를 얻기 위해 호출할 수 있는 함수나 프로퍼티를 함께 묶어준다. 리플렉션 API에서는 KCallable<out R>이라는 제네릭 인터페이스를 통해 호출가능한 요소를 표현한다. 여기서 R은 함수의 반환 타입이거나 프로퍼티의 타입에 해당한다.

KCallable 인스턴스를 얻는 방법으로는 **5장 고급 함수와 함수형 프로그래밍 활용하기**에서 설명한 호출 가능 참조를 사용하는 방식이 있다.

<코드>

fun combine(n: Int, s: String) = “$s$n”  
  
fun main() {  
 println(::combine.returnType) // kotlin.String  
}

</코드>

KClass 인스턴스로부터 멤버 프로퍼티나 함수를 얻을 수도 있다. 하지만 현재는 리플렉션 API에서 이런 방식을 통해 최상위 호출가능 인스턴스를 얻을 수는 없다.

이제 KCallable이 제공하는 멤버에 대해 살펴보자. KClass와 마찬가지로 어떤 변경자가 붙어있는지 알아낼 수 있는 프로퍼티들이 존재한다.

<코드>

val isAbstract: Boolean  
val isFinal: Boolean  
val isOpen: Boolean  
val isSuspend: Boolean  
val visibility: KVisibility?

</코드>

isSuspend에 해당하는 suspend 변경자에 대해서는 아직 설명하지 않았다. 이 변경자는 일시중단 가능한 계산(suspendable computation)을 지원하는 호출가능 객체에 사용된다. **13장. 동시성**에서 이 주제에 대해 자세히 다룰 것이다.

다음으로 프로퍼티나 함수의 시그니처를 표현하는 프로퍼티가 속한 그룹이 있다.

<코드>

val name: String  
val typeParameters: List<KTypeParameter>  
val parameters: List<KParameter>  
val returnType: KType

</코드>

멤버와 확장의 경우 첫 번째 파라미터는 수신 객체로 예약되어 있다는 점에 유의하라. 호출가능 요소가 멤버인 동시에 확장이라면 두 번째 파라미터도 다른 수신 객체로 예약되어 있다. 예를 살펴보자.

<코드>

import kotlin.reflect.KCallable  
  
val simpleVal = 1  
val Int.extValget() = this  
  
class A {  
 val Int.memberExtValget() = this  
}  
  
fun main() {  
 fun printParams(callable: KCallable<\*>) {  
 println(  
 callable.parameters.joinToString(prefix = “[“, postfix = “]”) {  
 it.type.toString()  
 }  
 )  
 }  
   
 // []  
 printParams(::simpleVal)  
   
 // [kotlin.Int]  
 printParams(Int::extVal)  
   
 // [A, kotlin.Int]  
 printParams(A::class.members.first { it.name == “memberExtVal” })  
}

</코드>

KParameter 인터페이스는 멤버나 확장 선언의 수신 객체나 함수/생성자의 파라미터에 대한 정보를 포함한다.

<코드>

val index: Int  
val isOptional: Boolean  
val isVararg: Boolean  
val name: String?  
val type: KType

</코드>

isOptional 프로퍼티는 파라미터에 디폴트 값이 있는지 여부를 돌려준다. 하지만 아직 리플렉션을 통해 디폴트 값 자체를 알 수는 없다. 파라미터 이름이 없거나, 소스 코드상에서 파라미터 이름을 표현할 수 없는 경우에는 name이 null일 수도 있다는 점에 유의하라. 예를 들어 수신 객체를 표현하는 파라미터의 경우 name이 null이다.

kind 프로퍼티는 KParameter 인스턴스가 일반적인 값에 해당하는지, 아니면 디스패치나 확장의 수신 객체인지를 알려준다. 이 프로퍼티는 KParameter.Kind 이넘에 정의된 상수 중 하나를 반환한다.

* INSTANCE: 멤버 선언의 디스패치 수신 객체
* EXTENSION\_RECEIVER: 확장 선언의 확장 수신 객체
* VALUE: 일반적인 값

KCallable에는 이 호출가능 요소가 표현하는 호출 가능한 선언을 동적으로 호출할 수 있게 해주는 call() 멤버 함수가 들어있다.

<코드>

fun call(vararg args: Any?): R

</코드>

함수로부터 만들어진 호출가능 요소인 경우 call()은 함수를 호출한다. 호출 가능 요소가 프로퍼티라면 게터가 호출된다. call()을 사용해 생성자나 멤버 함수를 호출하는 예제는 이미 살펴봤다. 여기서는 프로퍼티 예제를 살펴보자. 생성자 호출 예제와 똑같이 Person 클래스를 사용한다.

<코드>

fun main() {  
 val person = Person(“John”, “Doe”)  
 val personClass = person::class  
 val firstName = personClass.members.first { it.name == “firstName” }  
   
 println(firstName.call(person)) // John  
}

</코드>

호출 가능한 요소를 호출하는 다른 방법으로 callBy() 함수가 있다. 이 함수를 사용하면 맵 형태로 인자를 넘길 수 있다.

<코드>

fun callBy(args: Map<KParameter, Any?>): R

</코드>

이제 더 구체적인 호출 가능 요소를 살펴보자. KProperty 인터페이스는 프로퍼티에만 있는 변경자를 검사하는 프로퍼티를 추가 제공한다.

<코드>

val isConst: Boolean  
val isLateinit: Boolean

</코드>

프로퍼티 게터를 KFunction 타입의 인스턴스를 통해 접근할 수도 있다.

<코드>

val myValue = 1  
  
fun main() {  
 println(::myValue.getter()) // 1  
}

</코드>

KMutableProperty는 KProperty에 세터를 추가해준다.

<코드>

var myValue = 1  
  
fun main() {  
 ::myValue.setter(2)  
 println(myValue) // 2  
}

</코드>

KProperty에도 KProperty0, KProperty1, KProperty2이라는 하위 타입이 있다. 이들은 각각 순서대로 수신 객체가 없는 경우, 수신 객체가 하나인 경우(디스패치이거나 확장인 경우), 수신 객체가 둘인 경우(멤버인 확장의 경우)를 표현한다. 이런 하위 타입은 자신이 속한 유형에 따라 다른 함수 타입으로 게터 타입을 세분화해준다. 이로 인해 방금 봤던 예제처럼 ::myValue.getter를 함수로 쓸 수 있다. 비슷한 하위 타입을 통해 KMutableProperty도 역시 세터에 대한 세분화한 타입을 제공해준다.

마지막으로 다룰 리플렉션 타입은 KFunction이다. 이름에서 예상할 수 있는 것처럼 이 타입은 함수나 생성자를 표현한다. 이 인터페이스에 추가된 멤버는 모두 함수에 적용 가능한 변경자 검사를 위한 프로퍼티들 뿐이다.

<코드>

val isInfix: Boolean  
val isInline: Boolean  
val isOperator: Boolean  
val isSuspend: Boolean

</코드>

isInfix와 isOperator 검사는 연산자 함수와 관련이 있다. 이에 대해서는 **11장 도메인 특화 언어**에서 자세히 다룬다.

KFunction 자체는 다양한 인자 개수를 지원해야 하기 때문에 아무 함수 타입도 상속하지 않는다는 점에 유의하라. 하지만 일부 함수 타입은 좀 더 구체적인 KFunction의 하위 타입을 통해 구현이 될 수 있다. 이런 하위 타입의 예로 KProperty0/KProperty1/KProperty2에 정의된 접근자에 대해 앞에서 이미 살펴봤다. 또 다른 중요한 점으로 호출 가능한 참조가 항상 적절한 함수 타입을 준수한다는 점을 들 수 있다. 예를 살펴보자.

<코드>

import kotlin.reflect.KFunction2  
  
fun combine(n: Int, s: String) = “$s$n”  
  
fun main() {  
 val f: KFunction2<Int, String, String> = ::combine  
 println(f(1, “2”)) // 12  
}

</코드>

이 예제의 호출 가능 참조는 KFunction2<Int, String, String>이며, 이는 (Int, String) -> String의 하위 타입이다. 하지만 KProperty0나 다른 비슷한 타입들과 달리 KFunction0/KFunction1/... 등의 타입은 컴파일 시점에만 존재한다. 런타임에 이들은 합성 클래스에 의해 표현되며, 이는 람다를 표현하기 위해 합성 객체를 사용하는 것과도 비슷하다.

한 가지 더 주목할 점은 리플렉션을 통하면 가시성이 제한된 호출 가능 요소에 접근할 수도 있다는 점이다. 경우에 따라 비공개 함수를 리플렉션을 통해 호출할 필요가 있을 때가 있다. 자바에서 이런 시도를 하면 예외가 발생한다. 따라서 이럴 때는 미리 setAccessible(true)를 호출해서 해당 요소에 접근할 수 있게 설정해야 한다. 코틀린에서는 isAccessible 프로퍼티를 같은 목적에 사용한다.

<코드>

import kotlin.reflect.KProperty1  
import kotlin.reflect.jvm.isAccessible  
  
class SecretHolder(private val secret: String)  
  
fun main() {  
 val secretHolder = SecretHolder("Secret")  
 val secretProperty = secretHolder::class.members  
 .first { it.name == "secret" } as KProperty1<SecretHolder, String>  
   
 secretProperty.isAccessible = true  
 println(secretProperty.get(secretHolder))  
}

</코드>

## 결론

이 장에서는 애너테이션과 리플렉션에 대해 살펴봤다. 코틀린 코드에 애너테이션을 붙이고 런타임까지 메타데이터를 유지하는 방법을 살펴봤다. 주요 내장 애너테이션을 살펴보고 여러분이 직접 애너테이션 클래스를 작성하는 방법에 대해서도 설명했다. 그리고 코틀린 리플렉션 API에 대해 소개했다. 이제는 지정자나 호출가능 리플렉션 객체에 있는 애트리뷰트에 접근하는 방법을 알았으므로, 이를 동적으로 활용할 수 있다.

다음 장에서는 자신의 API를 영역 특화 언어처럼 설계하는 방법을 살펴보겠다. 이를 통해 여러분의 코드 기반에 선언적인 프로그래밍을 가미할 수 있다.

## 문제

1. 새로운 애너테이션을 어떻게 정의하는가? 코틀린 애너테이션 구문과 자바 구문을 비교하라.
2. 코틀린 코드에 애너테이션을 어떻게 붙일 수 있는가?
3. 코틀린이 제공하는 내장 애너테이션에 대해 설명하라.
4. 애너테이션의 사용 지점 대상은 무엇인가? (애너테이션 정의의) @Target 메타 에너테이션과 사용 지점 대상 사이에는 어떤 연관이 있는가?
5. 코틀린 리플렉션 API를 구성하는 기본 타입들은 무엇인가?
6. 클래스 리터럴과 호출 가능 참조 구문에 대해 설명하라.
7. KClass API에 대해 설명하라. KClass와 자바의 Class 인스턴스를 어떻게 상호 변환하는가?
8. KCallable API에 대해 설명하라.