# 12장. 자바 상호 운용성

이 장에서는 자바와 코틀린 코드의 상호 운용성과 연관된 여러 가지 주제를 소개한다. 이런 주제들은 자바 코드와 코틀린 코드가 공존해야 하는 프로젝트에서 중요한 역할을 한다. JVM 상호 운용성이 좋기 때문에 코틀린 코드를 기존 프로젝트에 쉽게 추가할 수 있고, 자바로 구성된 프로젝트 코드를 주변 환경을 거의 변화시키지 않고도 점진적으로 코틀린으로 변환할 수 있다.

코틀린과 자바 타입이 서로 어떻게 매핑되는지 살펴보고, 코틀린 선언을 자바에서 보면 어떤 식으로 보이는지, 반대로 자바 선언을 코틀린에서 어떻게 바라보는지 설명한다. 그리고 자바/코틀린 상호 운용성을 제어할 때 도움이 되는 언어적 특징을 소개한다.

## 구조

* 자바 코드를 코틀린에서 사용하기
* 코틀린 코드를 자바에서 사용하기

## 목표

코틀린 선언과 타입이 어떻게 자바로 매핑되는지 배우고, 코틀린과 자바를 한 코드 기반에서 혼용하는 방법을 배운다.

## 자바 코드를 코틀린에서 사용하기

코틀린은 JVM을 주 대상으로 설계됐기 때문에 상당히 쉽게 자바 코드를 코틀린에서 사용할 수 있다. 주로 발생하는 문제들은 자바에 없는 코틀린 기능으로 인한 것이다. 예를 들어 자바는 널 안전성을 타입 시스템을 통해 제공하지 않지만 코틀린에서는 항상 어떤 타입이 널이 될 수 있는 타입인지 아닌지 지정해야 한다. 자바 타입은 보통 이런 정보가 부족하다. 이 절에서는 자바와 코틀린 양쪽에서 이런 문제를 어떻게 해결할 수 있는지 살펴본다.

### 자바 메서드와 필드

대부분의 경우 자바 메서드를 아무 문제 없이 코틀린 함수처럼 노출시킬 수 있다. 캡슐화되지 않은 자바 필드를 코틀린에서는 뻔한 접근자가 있는 프로퍼티처럼 쓸 수 있다. 하지만 각 언어의 구체적인 내용에 따라 생기는 약간의 뉘앙스 차이에 유의해야 한다.

#### Unit과 void

코틀린에는 반환 값이 없음을 나타내는 void 키워드가 없다. 따라서 자바 void 함수는 코틀린에서 Unit을 반환하는 함수로 보인다. 이런 함수를 호출하고 호출 결과를 저장하며(예: 변수에 대입) 컴파일러가 Unit 객체에 대한 참조를 생성해 저장해 줄 것이다.

#### 연산자 관습

Map.get() 같은 몇몇 자바 메서드는 코틀린의 연산자 관습을 만족한다. 이런 자바 메서드에는 operator 키워드가 붙어 있지 않지만, 코틀린에서는 이들을 마치 연산자 함수인 것처럼 연산자를 통해 사용할 수 있다. 예를 들어 자바 리플렉션 API의 Method 클래스에는 invoke() 메서드가 있기 때문에, 이 클래스를 마치 함수처럼 호출할 수 있다.

<코드>

val length = String::class.java.getDeclaredMethod("length")  
println(length("abcde")) // 5

</코드>

하지만 중위 호출 문법을 자바 메서드에 적용할 수는 없다.

#### 합성 프로퍼티

자바에는 합성 프로퍼티가 없고, 게터와 세터를 사용하는 일이 많다. 이로 인해 코틀린 컴파일러는 자바 게터나 세터를 일반적인 코틀린 프로퍼티처럼 쓸 수 있게 합성 프로퍼티를 노출시켜준다. (컴파일러가 합성 프로퍼티를 만들어줄 수 있으려면) 접근자는 다음 관습을 따라야만 한다.

* 게터는 파라미터가 없는 메서드여야 하며, 메서드 이름이 get으로 시작해야 한다
* 세터는 파라미터가 하나만 있는 메서드여야 하며, 메서드 이름이 set으로 시작해야 한다

예를 들어 다음 자바 클래스가 있다고 하자.

<코드>

public class Person {  
 private String name;  
 private int age;  
   
 public Person(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
   
 public String getName() {  
 return name;  
 }  
   
 public void setName(String name) {  
 this.name = name;  
 }  
   
 public int getAge() {  
 return age;  
 }  
   
 public void setAge(int age) {  
 this.age = age;  
 }  
}

</코드>

코틀린에서는 이 클래스의 인스턴스를 마치 name과 age라는 가변 프로퍼티가 정의된 것처럼 쓸 수 있다.

<코드>

fun main() {  
 val person = Person("John", 25)  
 person.name = "Harry"  
 person.age = 30  
   
 println("${person.name}, ${person.age}") // Harry, 30  
}

</코드>

이 관습은 게터 메서드만 있는 경우에도 동작한다. 이 경우 자동으로 만들어지는 프로퍼티는 불변 프로퍼티가 된다. 현재 코틀린은 쓰기 전용 필드를 제공하지 않기 때문에, 자바 클래스에 세터 메서드는 있지만 게터가 없는 경우에는 아무 프로퍼티도 노출되지 않는다.

다른 관습으로, 게터 이름이 is로 시작할 수도 있다. 이런 경우 합성 프로퍼티는 게터와 이름이 같다. Person 클래스를 변경해서 접근자가 있는 boolean 필드를 추가해 보자.

<코드>

public class Person {  
 ...  
   
 private boolean isEmployed;  
   
 public boolean isEmployed() {  
 return isEmployed;  
 }  
 public void setEmployed(boolean employed) {  
 isEmployed = employed;  
 }  
}

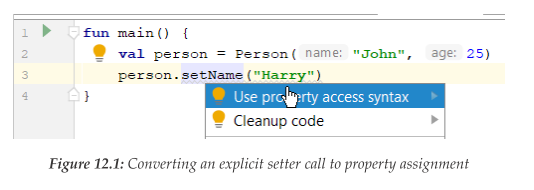
</코드>

코틀린 코드는 이 접근자를 isEmployed 프로퍼티로 사용할 수 있다.

<참고> IDE 팁

합성 프로퍼티 대신 일반 메서드를 사용할 수도 있다. 하지만 일반 메서드 사용은 불필요한 중복으로 간주된다. 기본적으로 인텔리J 코틀린 플러그인은 이런 호출을 감지해서 합성 프로퍼티 접근으로 변경하라고 제안한다(그림 12-1).

그림 12-1: 명시적인 세터 호출을 프로퍼티 대입문으로 변경하기



합성 프로퍼티 구문은 코틀린이 아닌 코드로 작성된 메서드에 대해서만 사용할 수 있다. 코틀린 소스 코드에서 get/set으로 선언한 메서드를 합성 프로퍼티로 사용할 수는 없다.

## 플랫폼 타입

자바가 널이 될 수 있는 타입과 그렇지 않은 타입을 구분하지 않기 때문에, 일반적으로 코틀린 컴파일러는 자바 코드에서 비롯된 객체가 널인지 여부에 대해 아무런 가정도 할 수 없다. 하지만 이런 값을 널이 될 수 있는 타입으로 노출시키면 코틀린 코드에서 널 검사를 일일이 수행해야만 하기 때문에 그리 실용적이지 않다. 따라서 코틀린 컴파일러는 자바 코드가 노출하는 타입에 대한 널 안전성 검사를 완화시켜서, 자바 타입을 명확한 널 가능성이 지정되지 않은 타입인 것처럼 취급한다. 코틀린에서, 자바 코드로부터 비롯된 객체는 플랫폼 타입(platform type)이라는 특별한 타입에 속한다. 플랫폼 타입은 널이 될 수 있는 타입이기도 하고 널이 될 수 없는 타입이기도 한 타입이다. 이런 타입에 대한 타입 안전성 보증은 기본적으로 자바와 동일하다. 플랫폼 타입의 값을 널이 될 수 있는 문맥에서 사용해도 되고 널이 될 수 없는 문맥에서 사용해도 된다. 하지만 널이 될 수 없는 문맥에서 플랫폼 타입을 사용할 경우 런타임에 NPE(NullPointerException)가 발생할 수도 있다.

다음 자바 클래스를 보자.

<코드>

public class Person {  
 private String name;  
 private int age;  
   
 public Person(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
   
 public String getName() { return name; }  
 public void setName(String name) { this.name = name; }  
 public int getAge() { return age; }  
 public void setAge(int age) { this.age = age; }  
}

</코드>

코틀린 코드에서 이 클래스를 사용해보자.

<코드>

fun main() {  
 val person = Person("John", 25)  
 println(person.name.length) // 4  
}

</코드>

이 코드에서 person.name은 컴파일러가 널 가능성을 판단할 수 없는 플랫폼 타입이다. 그래서 코드는 컴파일이 잘 되지만, 프로그램이 length에 접근하는 부분에 대한 널 검사는 런타임으로 미뤄진다. 이 코드를 다음 예제처럼 바꾸면 프로그램이 여전히 컴파일되지만, 런타임에 실패하게 된다.

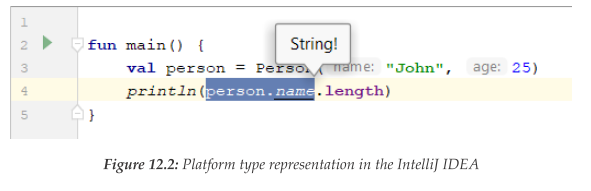
<코드>

fun main() {  
 val person = Person(null, 25)  
 println(person.name.length) // Exception  
}

</코드>

플랫폼 타입을 코틀린 소스 코드에서 명시할 수는 없다는 점에 유의하라. 컴파일러만 플랫폼 타입을 구성할 수 있다. 하지만 인텔리J IDEA 플러그인을 통해 플랫폼 타입을 살펴볼 수 있다. 예를 들어 "Show expression type" 액션([Ctrl]+[Shift]+[P]나 [Cmd]+[Shift]+[P])을 person.name 식에서 실행해보면 String!이라는 타입을 볼 수 있다(그림 12-2). 이 표기는 표시된 타입이 String?일 수도 있고 String일 수도 있다는 뜻이다.

그림 12-2: 인텔리J IDEA의 플랫폼 타입 표현



플랫폼 타입의 식을 변수에 대입하거나, 명시적인 타입을 지정하지 않고 함수에서 반환하면 플랫폼 타입이 전파된다. 예를 들어 다음 코드를 보자.

<코드>

import java.math.BigInteger  
  
// BigInteger! 반환 타입  
fun Int.toBigInt() = BigInteger.valueOf(toLong())  
  
val num = 123.toBigInt() // BigInteger! 타입

</코드>

타입을 명시적으로 지정하면 플랫폼 타입을 널이 될 수 있는 타입이나 널이 될 수 없는 타입으로 강제로 지정할 수 있다.

<코드>

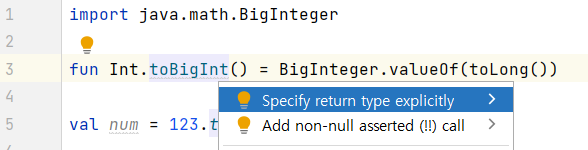
import java.math.BigInteger  
  
// BigInteger(널이 될 수 없음) 반환 타입  
fun Int.toBigInt(): BigInteger = BigInteger.valueOf(toLong())  
  
val num = 123.toBigInt() // BigInteger (널이 될 수 없음) 타입

</코드>

<참고> IDE 팁

인텔리J 코틀린 플러그인은 플랫폼 타입이 암시적으로 전파되는 경우를 찾아서 타입을 명시적으로 지정하거나 널 아님 단언(!!)을 사용하라고 경고해준다(그림 12-3).

그림 12-3: 플랫폼 타입 전파를 없애기



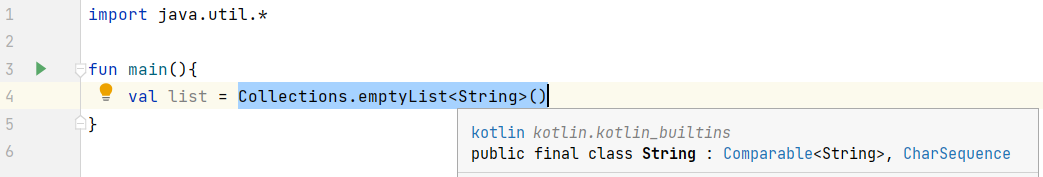
</참고>

플랫폼 타입을 널이 아닌 타입으로 강제로 변환하면 컴파일러가 단언문을 자동으로 생성해준다. 이를 통해 런타임에 널 값이 계속 전파되어 나중에 해당 값에 접근하는 시점에 NPE가 발생하지 않고 플랫폼 값을 널이 될 수 없는 값으로 대입하거나 변환하는 순간에 프로그램이 실패하게 된다.

코틀린은 자바 컬렉션 타입을 표현할 때도 플랫폼 타입을 쓴다. 그 이유는 널이 될 수 있는 타입의 경우와 비슷하다. 코틀린과 달리 자바는 가변 컬렉션과 불변 컬렉션을 구분하지 않는다. 따라서 코틀린에서 List, Set, Map 등의 표준 자바 컬렉션 인스턴스는 불변으로 생각될 수도 가변으로 생각될 수도 있다. IDE는 (Mutable)이라는 접두사를 붙여서 이런 타입을 나타낸다(그림 12-4).

그림 12-4: 변경 가능한 플랫폼 타입





## 널 가능성 애너테이션

자바 세계에서 널 안전성을 보장하는 일반적인 방법은 특별한 애너테이션을 쓰는 것이다. 인텔리J IDEA 같은 현대적인 개발 환경은 이런 애너테이션을 참조해서 널 가능성에 대한 계약을 잠재적으로 위반하는 코드를 경고해준다. 코틀린 컴파일러도 이런 애너테이션 중 일부를 지원한다. 이 경우 해당 자바 타입은 코틀린에서 (자바 코드에 붙은 애너테이션에 따라) 널이 될 수 있거나 널이 될 수 없는 타입 중 하나로 정해지며, 플랫폼 타입으로 지정되지 않는다. 예를 들어 앞에서 살펴본 Person 클래스 예제에 애너테이션을 붙일 수도 있다.

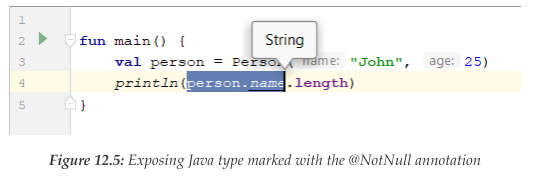
<코드>

import org.jetbrains.annotations.NotNull;  
  
public class Person {  
 @NotNull private String name;  
 private int age;  
   
 public Person(@NotNull String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
   
 @NotNull public String getName() { return name; }  
 public void setName(@NotNull String name) { this.name = name; }  
 public int getAge() { return age; }  
 public void setAge(int age) { this.age = age; }  
}

</코드>

이 코드에 대응하는 타입을 코틀린 코드에서 살펴보면 그림 12-5처럼 바뀐다.

그림 12-5: @NotNull 애너테이션이 붙은 자바 타입 노출하기



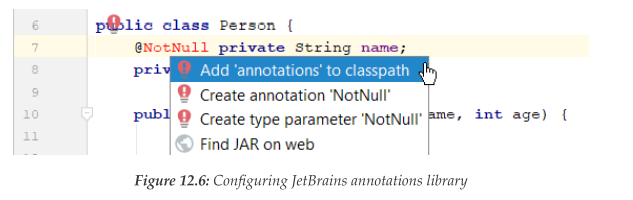
코틀린 컴파일러가 지원하는 널 가능성 애너테이션으로는 (더 완전한 목록을 kotlinlang.org의 코틀린 문서에서 볼 수 있다) 다음과 같은 것이 있다.

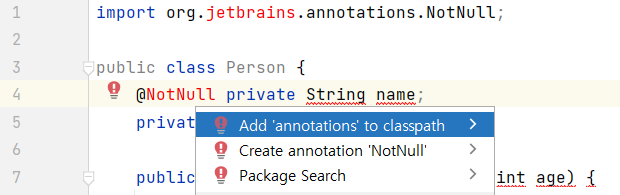
* 젯브레인즈의 @Nullable과 @NotNull (org.jetbrains.annotations 패키지)
* 안드로이드 SDK의 여러 가지 @Nullable과 @NotNull 변종들
* @Nonnull 등의 JSR-350 널 가능성 애너테이션(javax.annotation 패키지)

<참고> IDE 팁

젯브레인즈 애너테이션 라이브러리는 자동으로 프로젝트 의존관계에 추가되지 않지만 쉽게 설정에 추가할 수 있다. @Nullable/@NotNull 애너테이션을 사용할 수 없다면 사용할 수 없는 애너테이션 이름에 캐럿을 위치시키고 [Alt]+[Enter]를 눌러서 그림 12-6처럼 Add 'annotations' to the classpath 액션을 선택하라.

그림 12-6: 젯브레인즈 annotations 라이브러리 설정하기





</참고>

여러분이 사용하는 널 가능성 애너테이션이 ElementType.TYPE\_USE 대상을 지정하면, 자바 8부터 제네릭 선언의 타입 파라미터에 대해 이 애너테이션을 붙일 수 있다. 예를 들어 젯브레인즈의 @Nullable과 @NotNull 애너테이션은 버전 15부터 이 대상을 지원하기 때문에 다음과 같이 코드를 작성할 수 있다.

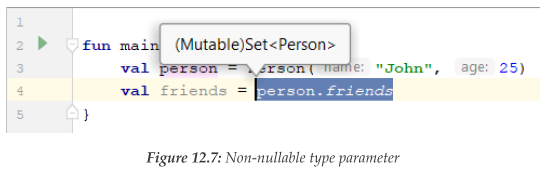
<코드>

public class Person {  
 ...  
 @NotNull private Set<@NotNull Person> friends = new HashSet<>();  
 @NotNull public Set<@NotNull Person>getFriends() { return friends; }  
}

</코드>

이 경우 코틀린에서 getFriends() 메서드의 반환 타입은 (Mutable)Set<Person> 처럼 보인다.

그림 12-7: 널이 될 수 없는 타입 파라미터



</참고>

타입 파라미터에 애너테이션이 붙지 않으면 코틀린 컴파일러는 플랫폼 타입을 사용한다. 따라서 다음 코틀린 예제 코드의 person.friends는 (Mutable)Set<Person!>이 된다.

<코드>

public class Person {  
 ...  
 @NotNull private Set<Person> friends = new HashSet<>();  
 @NotNull public Set<Person>getFriends() { return friends; }  
}

</코드>

## 자바/코틀린 타입 매핑

자바와 코틀린에서 비슷한 의미를 가지는 타입도 있다. 예를 들어 자바 원시 타입인 int, boolean 등은 코틀린의 내장 타입인 Int, Boolean 등에 해당하며, java.util.List 타입은 (Mutable)List에 해당한다. 코틀린 컴파일러는 JVM 플랫폼에서 실행되도록 컴파일하는 경우, 자바 코드에 선언된 내용을 코틀린 코드에서 사용하거나 코틀린 코드에 선언된 타입을 자바 코드에서 사용할 때 서로 상응하는 타입을 상호 변환해준다. 이 절에서는 기본적인 자바/코틀린 타입 매핑 규칙에 대해 알아보자.

우선 자바 원시 타입이나 원시 타입에 상응하는 박싱 타입은 코틀린의 기본 타입에 대응한다.

|  |  |
| --- | --- |
| 자바 타입 | 코틀린 타입 |
| byte/Byte | Byte |
| short/Short | Short |
| int/Integer | Int |
| long/Long | Long |
| char/Character | Char |
| float/Float | Float |
| double/Double | Double |

이 매핑은 역방향으로도 적용된다. JVM에서 기본 코틀린 타입의 값은 값이 사용되는 방식에 따라 JVM 원시 타입이거나 원시 타입에 상응하는 박싱 타입이다. 예를 들어 Int? 같은 타입은 자바의 int에서는 널을 표현할 방법이 없기 때문에 java.lang.Integer 인스턴스로 표현된다.

java.lang 패키지에 들어있는 원시 타입이 아닌 내장 클래스 중 일부도 kotlin 패키지에 있는 상응하는 코틀린 클래스로 매핑된다(역방향도 성립함). 이런 경우에는 자바와 코틀린 클래스 이름이 같다. 유일한 차이는 코틀린 Any로 매핑되는 자바 Object 뿐이다.

* Object
* Cloneable
* Comparable
* Enum
* Annotation
* CharSequence
* String
* Number
* Throwable

코틀린으로 매핑된 자바 클래스의 정적 멤버(예: Long.valueOf())를 코틀린쪽 동반 객체에서 직접 접근할 수 없다는 점에 유의하라. 정적 멤버를 사용하려면 해당 자바 클래스의 전체 이름을 언급해야 한다.

<코드>

val n = java.lang.Long.bitCount(1234)

</코드>

코틀린 표준 컬렉션 타입(불변과 가변 모두)은 java.util 패키지에 있는 상응하는 컬렉션 타입으로 매핑된다. 하지만 자바에서 코틀린 방향으로의 매핑은 앞에서 설명한 것처럼 플랫폼 타입을 만들어낸다. 자바 컬렉션은 불변과 가변 구현이 같은 API를 사용하기 때문이다. 매핑되는 타입은 다음과 같다.

* Iterable/Iterator/ListIterator
* Collection
* Set
* List
* Map/Map.Entry

제네릭 타입의 매핑은 양 언어 제네릭 구문의 차이 때문에 단순하지만은 않은 변환이 필요하다.

* 자바의 extends 와일드카드는 코틀린 공변 프로젝션으로 변환된다. 예를 들어 TreeNode<? extends Person>은 TreeNode<out Person>으로 바뀐다
* 자바의 super 와일드카드는 코틀린 반공변 프로젝션으로 변환된다. 예를 들어 TreeNode<? super Person>은 TreeNode<in Person>으로 바뀐다
* 자바의 로우 타입(raw type)은 코틀린 스타 프로젝션으로 바뀐다. 예를 들어 TreeNode는 TreeNode<\*>이 된다

원시 타입으로 이뤄진 자바 배열(int[] 등)은 박싱/언박싱을 피하기 위해 상응하는 특화된 코틀린 배열 클래스(예: IntArray)로 매핑된다. 다른 모든 배열은 Array<(out)T>라는 특별한 플랫폼 타입의 배열 인스턴스로 변환된다(또는 Array<(out) String>! 같은 널이 될 수 있는 플랫폼 타입이 될 수도 있다). 특히 이런 변환으로 인해 상위 타입의 배열을 받으리라 예상하는 자바 메서드에게 하위 타입의 배열을 넘길 수 있다. 예를 들어 다음 코드는 Object[] 파라미터의 값으로 String으로 이뤄진 배열을 전달한다.

<코드>

import java.util.\*  
  
fun main() {  
 val strings = arrayOf("a", "b", "c")  
 println(Arrays.deepToString(strings))  
}

</코드>

이런 동작은 배열 타입이 공변적인 자바의 의미와 맞아 떨어진다. 하지만 코틀린 배열은 무공변이기 때문에, 코틀린 메서드에 대해서는 이런 기법을 사용할 수 없다. 원한다면 Array타입을 직접 Array<out Any>로 out 프로젝션 시켜야 한다.

### 단일 추상 메서드(SAM) 인터페이스

추상 메서드가 하나뿐인 자바 인터페이스가 있다면(이런 인터페이스를 SAM 인터페이스라고 줄여 부른다), 이 인터페이스는 기본적으로 코틀린 함수 타입처럼 작동한다. 이는 자동으로 람다와 적절한 SAM 타입 인스턴스 사이의 변환을 지원해주는 자바 8+와 비슷하다. 코틀린은 자바 SAM 인터페이스가 필요한 위치에 람다를 넘길 수 있게 지원한다. 이를 SAM 변환이라고 한다. 예를 들어 비동기 계산을 등록할 수 있는 JDK ExecutorService 클래스를 생각해보자. 이 클래스의 execute() 메서드는 Runnable 객체를 인자로 받는다.

<코드>

public interface Runnable {  
 public void run();  
}

</코드>

코틀린은 Runnable을 SAM 인터페이스로 판정하기 때문에, 코틀린 코드에서 execute() 메서드에게 람다를 넘길 수 있게 해준다.

<코드>

import java.util.concurrent.ScheduledThreadPoolExecutor  
  
fun main() {  
 val executor = ScheduledThreadPoolExecutor(5)  
   
 executor.execute {  
 println("Working on asynchronous task...")  
 }  
 executor.shutdown()  
}

</코드>

이런 호출을 지원하지 않는다면 더 번잡한 코드를 써야 할 것이다.

<코드>

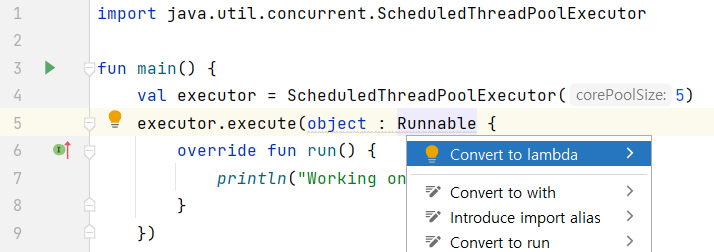
import java.util.concurrent.ScheduledThreadPoolExecutor  
  
fun main() {  
 val executor = ScheduledThreadPoolExecutor(5)  
   
 executor.execute(object : Runnable {  
 override fun run() {  
 println("Working on asynchronous task...")  
 }  
 })  
 executor.shutdown()  
}

</코드>

<참고> IDE 팁

인텔리J 코틀린 플러그인은 방금 본 것처럼 불필요한 객체 식을 자동으로 암시적인 SAM 변환을 사용하는 람다로 바꿔준다(그림 12-8).

그림 12-8: 객체 식을 람다로 변환하기



</참고>

때로는 컴파일러가 문맥을 충분히 파악할 수 없어서 적절한 변환을 찾지 못할 수도 있다. 예를 들어 자바 ExecutorService에는 미래에 수행할 계산을 표현하는 객체를 인자로 받는 submit() 메서드가 몇 가지 들어있다. 계산은 Runnable이나 Callable 인터페이스의 인스턴스일 수 있다. Callable은 다음과 같다.

<코드>

public interface Callable<V> {  
 V call() throws Exception;  
}

</코드>

Runnable과 Callable 모두 SAM 인터페이스이므로, 코틀린 코드에서 람다를 submit() 메서드에 전달하면 컴파일러는 Runnable을 선택한다. 왜냐하면 Runnable 타입이 가장 구체적인 시그니처를 가졌기 때문이다.

<코드>

import java.util.concurrent.ScheduledThreadPoolExecutor  
  
fun main() {  
 val executor = ScheduledThreadPoolExecutor(5)  
 // 암시적으로 Runnable으로 변환됨  
 val future = executor.submit { 1 + 2 }  
   
 println(future.get()) // null  
 executor.shutdown()  
}

</코드>

대신 Callable을 전달하고 싶다면 어떻게 해야 할까? 이 경우에는 대상 타입을 지정해서 변환을 명시해야 한다.

<코드>

import java.util.concurrent.Callable  
import java.util.concurrent.ScheduledThreadPoolExecutor  
fun main() {  
 val executor = ScheduledThreadPoolExecutor(5)  
 val future = executor.submit(Callable { 1 + 2 })  
   
 println(future.get()) // 3  
 executor.shutdown()  
}

</코드>

이런 식을 SAM 생성자라고 부른다.

SAM 변환은 인터페이스에 대해서만 적용되고, 메서드가 하나뿐인 클래스에는 적용될 수 없다. 그리고 자바와 달리 코틀린 인터페이스에 대해서도 SAM 변환을 쓸 수 없다. 코틀린에서는 보통 적절한 함수 타입을 사용할 수 있기 때문에 SAM 변환이 거의 불필요하지만, 가끔 자바의 SAM 변환을 코틀린에서 정의한 인터페이스로도 사용하고 싶을 때가 있다. 예를 들어 코틀린에서 정의한 인터페이스를 자바와 공유하는 경우 자바에서는 SAM 변환을 활용해 람다를 쓸 수 있는데, 코틀린에서는 객체 식을 써야 한다면 불편할 것이다. 코틀린 1.4부터는 함수형 인터페이스를 선언하면 코틀린에서도 인터페이스 인스턴스 대신 람다를 쓸 수 있다. 함수형 인터페이스는 추상 메서드가 하나뿐인 인터페이스 정의의 interface앞에 fun 키워드를 붙인 인터페이스다. 예를 들어 다음과 같은 인터페이스를 정의한다면,

<코드>

fun interface SomeCallback {  
 fun execute(arg: Any): Unit  
}

</코드>

다음과 같이 SAM 변환을 활용해 이 인터페이스의 인스턴스 대신 람다를 전달할 수 있다.

<코드>

fun callWithString(arg: String, callback: SomeCallback) = callback.execute(arg)  
  
fun main() {  
 callWithString("World!"){println("Hello,$it")} // Hello, World!  
}

</코드>

## 자바를 코틀린으로 변환하는 변환기 사용하기

인텔리J 플러그인에는 자바 소스 파일을 동등한 코틀린 코드로 변환해주는 도구가 들어있다. 이 기능은 자바/코틀린 상호 운용성과 더불어 기존 자바 코드 기반을 점진적으로 코틀린으로 마이그레이션할 때 도움이 되는 기능이다.

파일을 변환하려면 [Ctrl]+[Alt]+[Shift]+[K]를 누르거나 Code 메뉴에서 Convert Java File to Kotlin File 액션을 선택하면 된다. IDE는 파일을 처리해서 코틀린으로 변환해주고 필요하면 외부에서 이 코틀린 코드를 사용하는 코드도 변경해준다.

또는 Project View 패널에서 여러 자바 파일을 선택한 다음에 같은 쇼트컷(shortcut)을 적용해서 한꺼번에 파일을 변경할 수도 있다.

자동 변환은 코틀린다운 코드를 생성해내도록 만들어졌다. 항상 이상적인 결과를 만들어내는 것은 아니지만, 기존 자바 코드 기반을 코틀린으로 마이그레이션할 때 이 도구로 시작하면 좋을 것이다.

## 코틀린 코드를 자바에서 사용하기

코틀린을 설계할 때 기존 자바 코드 기반과 부드럽게 상호 운용할 수 있어야 한다는 가이드라인이 있었다. 대부분의 코틀린 코드는 크게 신경 쓰지 않고 자바 코드와 통합할 수 있다. 하지만 자바에 직접적으로 대응하는 코틀린 기능이 없을 때도 있다. 이 절에서는 이런 기능을 설명하고, 자바 관점에서 코틀린 코드가 노출하는 내용을 미세하게 조정하는 방법에 대해 설명한다.

### 프로퍼티 접근

자바나 JVM에는 프로퍼티라는 개념이 없기 때문에, 코틀린 프로퍼티를 자바에서 직접 접근할 수는 없다. 하지만 컴파일된 JVM 바이트코드에서는 각 프로퍼티가 접근자 메서드로 표현되기 때문에 자바 클라이언트는 다른 일반적인 메서드와 마찬가지 방법으로 이 접근자 메서드를 통해 프로퍼티에 접근할 수 있다. 접근자의 시그니처는 다음 규칙에 따라 프로퍼티의 정의에서 만들어진다.

* 게터는 파라미터가 없는 메서드이며 게터의 반환 타입은 원래의 프로퍼티 타입과 같다. 게터의 이름은 프로퍼티 이름의 첫 번째 글자를 대문자로 바꾼 다음, 앞에 get을 붙여서 생성된다
* 세터는 새로운 값에 해당하는 파라미터를 하나만 받는 메서드다. 세터의 이름은 프로퍼티 이름의 첫 번째 글자를 대문자로 바꾼 다음, 앞에 set을 붙여서 생성된다

예를 들어 다음 코틀린 클래스를 살펴보자.

<코드>

class Person(var name: String, val age: Int)

</코드>

이 클래스는 (자바 관점에서 보면) 다음과 같다.

<코드>

public class Person {  
 @NotNull  
 public String getName() {...}  
 public void setName(@NotNull String value) {...}  
 public int getAge() {...}  
}

</코드>

따라서 자바 클라이언트 코드는 접근자 메서드를 통해 프로퍼티에 접근할 수 있다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 Person person = new Person("John", 25);  
 System.out.println(person.getAge()); // 25  
   
 person.setName("Harry");  
 System.out.println(person.getName()); // Harry  
 }  
}

</코드>

프로퍼티 이름이 is로 시작하는 경우 코틀린 컴파일러는 다른 명명 규칙을 사용한다.

* 게터 이름은 프로퍼티와 같다
* 세터 이름은 맨 앞의 is를 set으로 바꾼 이름이다

예를 들어 Person 클래스에 isEmployed 프로퍼티를 추가했다고 하자.

<코드>

class Person(var name: String, val age: Int, var isEmployed: Boolean)

</코드>

새 프로퍼티에 접근하는 자바 코드는 다음과 같다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 Person person = new Person("John", 25, false);  
 person.setEmployed(true);  
 System.out.println(person.isEmployed()); // true  
 }  
}

</코드>

이 관습은 이름만 살펴본다는 점에 유의하라. Boolean 타입과는 아무 관계가 없다(하지만 혼동을 막기 위해 is로 시작하는 이름은 오직 Boolean 프로퍼티에만 사용할 것을 강력히 권장한다).

코틀린 프로퍼티에 뒷받침하는 필드가 필요한 경우, 컴파일러가 접근자 메서드와 함께 필드도 만들어준다. 하지만 기본적으로 이 필드는 비공개이기 때문에 게터 세터 코드 밖에서 필드에 접근할 수는 없다. 경우에 따라 자바 클라이언트 쪽에 이 프로퍼티 필드를 노출시켜야 할 때가 있다. 이런 경우에는 @JvmField 애너테이션을 프로퍼티 앞에 넣으면 된다. 예를 들어 Person 클래스의 생성자 파라미터 앞에 @JvmField를 붙여보자.

<코드>

class Person(@JvmField var name: String, @JvmFieldval age: Int)

</코드>

이렇게 하면 자바 소스 코드에서 컴파일러가 생성해주는 필드에 접근할 수 있다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 Person person = new Person("John", 25);  
 System.out.println(person.age); // 25  
   
 person.name = "Harry";  
 System.out.println(person.name); // Harry  
 }  
}

</코드>

이 경우 접근자 메서드는 생성되지 않고 뒷받침하는 필드가 프로퍼티 자체와 동일한 가시성으로 만들어진다는 점에 유의하라. 프로퍼티 접근자가 뻔하지(게터는 필드 값을 그대로 돌려주기만 하고, 세터는 받은 값을 그대로 필드에 저장하기만 하며 다른 처리를 하지 않음) 않다면 @JvmField를 사용할 수 없다.

<코드>

class Person(valfirstName: String, valfamilyName: String) {  
 // Error: 프로퍼티에 커스텀 게터가 있음  
 @JvmFieldvalfullNameget() = “$firstName $familyName”  
}

</코드>

@JvmField를 추상 프로퍼티나 열린 프로퍼티에 적용할 수도 없다. 이 프로퍼티를 오버라이드하는 쪽에서 커스텀 접근자를 만들 수도 있기 때문이다.

<코드>

open class Person(val firstName: String, val familyName: String) {  
 // Error: property is open  
 @JvmField open val description: String  
 get() = “$firstName $familyName”  
}

</코드>

이름이 붙은 객체의 프로퍼티 대해 @JvmField를 적용하면 인스턴스 필드가 아니라 정적 필드를 만들어낸다. 예를 들어 다음 코틀린 객체를 보자.

<코드>

object Application {  
 @JvmFieldval name = "My Application"  
}

</코드>

자바 코드는 Application.name이라는 이름의 정적 필드를 통해 name 프로퍼티에 직접 접근할 수 있다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 System.out.println(Application.name);  
 }  
}

</코드>

const 변경자가 붙은 프로퍼티도 같다.

<코드>

object Application {  
 const val name = "My Application"  
}

</코드>

뒷받침하는 필드를 노출하는 다른 방법으로 lateinit 프로퍼티를 활용하는 방법이 있다.

<코드>

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

</코드>

이 경우 접근자와 뒷받침하는 필드가 모두 프로퍼티와 동일한 가시성을 가진다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 Person person = new Person("John", "Doe");  
 person.init();  
   
 // 필드에 직접 접근  
 System.out.println(person.fullName); // John Doe  
   
 // 접근자 호출  
 System.out.println(person.getFullName()); // John Doe  
 }  
}

</코드>

객체에서 lateinit은 @JvmField 애너테이션이 붙은 필드와 비슷한 정적 필드를 생성한다. 하지만 lateinit 프로퍼티의 접근자는 인스턴스 메서드로 남는다. 예를 들어 싱글턴 객체를 하나 정의해보자.

<코드>

object Application {  
 lateinit var name: String  
}

</코드>

다음 자바 코드는 lateinit 프로퍼티의 필드에 접근하는 경우와 접근자 메서드에 접근하는 경우의 차이를 보여준다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 // 접근자 호출(정적 접근이 아님)  
 Application.INSTANCE.setName(“Application1”);  
   
 // 직접 프로퍼티 접근(정적 접근)  
 Application.stdin = “Application2”  
 }  
}

</코드>

lateinit 프로퍼티에 @JvmField를 붙일 수 없다는 점에 유의하라.

## 필드 퍼사드와 최상위 선언

코틀린에서는 다른 선언 내부가 아니라 패키지 바로 아래에 두는 최상위 선언을 자주 사용한다. 자바와 JVM 플랫폼에서는 일반적으로 모든 메서드는 어떤 클래스에 속해야만 한다. 이런 요구사항을 만족시키기 위해 코틀린 컴파일러는 최상위 함수와 프로퍼티를 자동으로 생성된 파일 퍼사드(file facade)라는 클래스에 넣는다. 기본적으로 퍼사드 클래스 이름은 소스 코드 이름 뒤에 Kt를 더 붙인 이름이 된다. 예를 들어 다음 파일을 컴파일하면,

<코드>

// util.kt  
class Person(val firstName: String, val familyName: String)  
  
val Person.fullName  
 get() = "$firstName $familyName"  
   
fun readPerson(): Person? {  
 val fullName = readLine() ?: return null  
 val p = fullName.indexOf(' ')  
 return if (p >= 0) {  
 Person(fullName.substring(0, p), fullName.substring(p + 1))  
 } else {  
 Person(fullName, "")  
 }  
}

</코드>

다음과 같은 퍼사드 클래스가 만들어진다.

<코드>

public class UtilKt {  
 @NotNull  
 public static String getFullName(@NotNull Person person) {...}  
   
 @Nullable  
 public static Person readPerson() {...}  
}

</코드>

퍼사드 클래스에는 최상위에 선언된 클래스가 들어가지 않는다. 클래스는 JVM과 자바에서 최상위 수준에 존재할 수 있다.

퍼사드 클래스에 생성된 메서드가 정적 메서드이기 때문에 자바 코드에서 최상위 메서드에 접근할 때 퍼사드 클래스를 인스턴스화할 필요가 없다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 Person person = UtilKt.readPerson();  
 if (person == null) return;  
 System.out.println(UtilKt.getFullName(person));  
 }  
}

</코드>

코틀린 컴파일러는 생성된 퍼사드의 몇몇 세부 사항을 조정할 수 있게 허용한다. 우선 파일 수준의 @JvmName 애너테이션을 통해 퍼사드 클래스 이름을 지정할 수 있다.

<코드>

@file:JvmName("MyUtils")  
class Person(val firstName: String, val familyName: String)  
  
val Person.fullName  
 get() = "$firstName $familyName"

</코드>

이제 자바 클라이언트는 MyUtils라는 이름을 통해 이 파일에 선언된 최상위 함수와 프로퍼티에 접근할 수 있다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 Person person = new Person("John", "Doe");  
 System.out.println(MyUtils.getFullName(person));  
 }  
}

</코드>

또 다른 유용한 기능으로는 여러 파일에 있는 최상위 선언을 한 클래스로 모으는 것이다. 이렇게 하려면 합치려는 파일마다 @JvmMultifileClass라는 애너테이션을 붙이고, @JvmName으로 클래스 이름을 지정해야 한다. 그러면 코틀린 컴파일러가 자동으로 파사드 클래스 이름이 같은 파일들의 선언을 한데 모아준다. 예를 들어 조금 전에 본 예제의 모든 선언을 별도의 파일에 넣어보자.

<코드>

// Person.kt  
class Person(val firstName: String, val familyName: String)  
  
// utils1.kt  
@file:JvmMultifileClass  
@file:JvmName("MyUtils")  
  
val Person.fullName  
 get() = "$firstName $familyName"  
  
// utils2.kt  
@file:JvmMultifileClass  
@file:JvmName("MyUtils")  
  
fun readPerson(): Person? {  
 val fullName = readLine() ?: return null  
 val p = fullName.indexOf(' ')  
   
 return if (p >= 0) {  
 Person(fullName.substring(0, p), fullName.substring(p + 1))  
 } else {  
 Person(fullName, "")  
 }  
}

</코드>

@JvmMultifile와 @JvmName 덕분에 두 선언을 계속 MyUtil 클래스 안에서 찾을 수 있다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 Person person = MyUtils.readPerson();  
 if (person == null) return;  
 System.out.println(MyUtils.getFullName(person));  
 }  
}

</코드>

코틀린 코드에서는 퍼사드 클래스에 접근할 수 없고, 오직 JVM쪽 클라이언트에서만 퍼사드에 접근할 수 있다는 점에 유의하라.

## 객체와 정적 멤버

JVM에서 코틀린 객체 선언은 정적인 INSTANCE 필드가 있는 일반적인 클래스로 컴파일된다. 예를 들어 다음과 같은 코틀린 선언이 있다고 하자.

<코드>

object Application {  
 val name = “My Application”  
 fun exit() { }  
}

</코드>

자바 코드에서는 Application.INSTANCE 필드를 통해 이 객체의 멤버에 접근할 수 있다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 System.out.println(Application.INSTANCE.getName());  
 Application.INSTANCE.exit();  
 }  
}

</코드>

앞 절에서 @JvmField를 객체 프로퍼티에 적용하면 프로퍼티가 자바 입장에서 볼 때 정적 필드로 바뀐다고 설명했다. 때로는 객체 함수나 프로퍼티 접근자를 정적 메서드로 만들면 편리할 때가 있다. 이 경우 @JvmStatic 애너테이션을 사용한다.

<코드>

import java.io.InputStream  
  
object Application {  
 @JvmStatic var stdin: InputStream = System.`in`  
 @JvmStatic fun exit() { }  
}

</코드>

이제 자바 코드에서 구체적인 인스턴스를 지정하지 않아도 이런 함수나 프로퍼티에 접근할 수 있다.

<코드>

import java.io.ByteArrayInputStream;  
  
public class Main {  
 public static void main(String[] args) {  
 Application.setStdin(new ByteArrayInputStream("hello".getBytes()));  
 Application.exit();  
 }  
}

</코드>

### 노출된 선언 이름을 변경하기

@JvmName을 사용해 최상위 선언이 들어가는 퍼사드 클래스의 이름을 바꿀 수 있다는 점을 살펴봤다. 실제 @JvmName 애너테이션은 파일에만 적용할 수 있는 게 아니라, 함수나 프로퍼티 접근자에도 적용할 수 있다. 이를 통해 함수나 프로퍼티 접근자 이름을 변경할 수 있다.

이 기능의 주된 용도는 코틀린에서는 올바른 선언이지만 자바에서는 금지된 선언이 되는 시그니처 충돌을 막는 것이다. 다음 코틀린 코드를 보라.

<코드>

class Person(val firstName: String, val familyName: String)  
  
val Person.fullName  
 get() = "$firstName $familyName" // Error  
   
fun getFullName(person: Person): String { // Error  
 return "${person.familyName}, ${person.firstName}"  
}

</코드>

코틀린에서는 프로퍼티와 함수를 쉽게 구분할 수 있음에도 불구하고 이 코드는 컴파일 오류가 난다. JVM 상에서 프로퍼티와 함수가 똑같은 시그니처의 메서드를 만들어내고, 이로 인해 모호성이 생기기 때문에 컴파일러는 오류를 낼 수 밖에 없다.

<코드>

@NotNull  
public static String getFullName(@NotNull Person person) {...}

</코드>

@JvmName을 사용하면 충돌이 일어나는 이름을 바꿔서 문제를 해결할 수 있다.

<코드>

@JvmName(“getFullNameFamilyFirst”)  
fun getFullName(person: Person): String { // 정상  
 return "${person.familyName}, ${person.firstName}"  
}

</코드>

이제 자바 클라이언트는 이 함수를 getFullNameFamilyFirst이라는 이름으로 호출할 수 있지만, 코틀린 코드는 여전히 원래의 getFullName을 사용해야 한다.

마찬가지로 프로퍼티의 접근자에 대해 @JvmName 애너테이션을 붙이면 JVM의 접근자 이름을 지정할 수 있다.

<코드>

val Person.fullName  
 @JvmName("getFullNameFamilyLast")  
 get() = "$firstName $familyName"

</코드>

또는 프로퍼티 자체에 애너테이션을 붙일 수도 있다(이때 사용 지점 대상을 지정한다).

<코드>

@get:JvmName("getFullNameFamilyLast")  
val Person.fullName  
 get() = "$firstName $familyName"

</코드>

특히 @JvmName을 사용하면 프로퍼티 접근자에 적용되는 표준 명명 방식을 우회할 수 있다.

<코드>

class Person(@set:JvmName("changeName") var name: String, val age: Int)

</코드>

자바 코드에서 볼 때 Person 클래스에는 setName()이 아니라 changeName()이라는 메서드가 포함된다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 Person person = new Person("John", 25);  
 person.changeName("Harry");  
 System.out.println(person.getName());  
 }  
}

</코드>

@JvmName은 코틀린 함수 이름이 자바 키워드와 우연히 겹쳐서 자바 클라이언트가 이 함수를 쓸 수 없는 경우에도 유용하다. 예를 살펴보자.

<코드>

class Person(val firstName: String, val familyName: String) {  
 @JvmName("visit")  
 fun goto(person: Person) {  
 println("$this is visiting $person")  
 }  
   
 override fun toString() = "$firstName $familyName"  
}

</코드>

goto가 예약된 키워드이기 때문에 자바에서는 goto() 함수를 호출할 수 없다. 이 문제는 커스텀 JVM 이름을 지정하면 해결할 수 있다.

### 오버로딩한 메서드 생성하기

코틀린 함수에 디폴트 값이 지정된 경우, 함수 인자 중 일부를 생략할 수 있기 때문에 함수를 호출할 때 인자의 수가 달라질 수 있다.

<코드>

// util.kt  
fun restrictToRange(  
 what: Int,  
 from: Int = Int.MIN\_VALUE,  
 to: Int = Int.MAX\_VALUE  
): Int {  
 return Math.max(from, Math.min(to, what))  
}  
  
fun main() {  
 println(restrictToRange(100, 1, 10)) // 10  
 println(restrictToRange(100, 1)) // 100  
 println(restrictToRange(100)) // 100  
}

</코드>

하지만 자바에는 디폴트 값이라는 개념이 없기 때문에 이 예제의 함수는 다음과 같이 보인다.

<코드>

public int restrictToRange(int what, int from, int to) {...}

</코드>

결과적으로 자바 클라이언트는 디폴트 값을 사용하고 싶더라도 항상 모든 인자를 넘겨야만 한다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 System.out.println(UtilKt.restrictToRange(100, 1, 10));  
 System.out.println(UtilKt.restrictToRange(100, 1)); // Error  
 System.out.println(UtilKt.restrictToRange(100)); // Error  
 }  
}

</코드>

코틀린은 이에 대한 해법으로 @JvmOverloads 애너테이션을 제공한다.

<코드>

@JvmOverloads  
fun restrictToRange(  
 what: Int,  
 from: Int = Int.MIN\_VALUE,  
 to: Int = Int.MAX\_VALUE  
): Int {  
 return Math.max(from, Math.min(to, what))  
}

</코드>

@JvmOverloads를 적용하면 원래 코틀린 함수 외에 오버로드된 함수를 추가로 생성해준다.

* 오버로드된 첫 번째 함수는 마지막 파라미터를 제외한 나머지 인자를 받는 함수이며, 이 함수는 원래 함수의 마지막 파라미터를 디폴트 값으로 지정해준다.
* 오버로드된 두 번째 함수는 마지막 두 파라미터를 제외한 나머지 인자를 받는 함수이며, 이 함수는 원래 함수의 마지막 두 파라미터를 디폴트 값으로 지정해준다. 세 번째, 네 번째 함수도 이런 식으로 점점 파라미터 수를 줄여나가면서 디폴트 값을 적용해준다.
* 오버로드된 마지막 함수는 파라미터를 하나만 받고, 나머지 파라미터를 디폴트 값으로 적용해준다.

예를 들어 앞의 restrictToRange 함수는 자바에서 볼 때 세 가지 오버로드 함수가 생긴다.

<코드>

public int restrictToRange(int what, int from, int to) {...}  
public int restrictToRange(int what, int from) {…}  
public int restrictToRange(int what) {...}

</코드>

추가된 오버로드 함수들은 원래 함수를 호출하면서 생략된 파라미터에 대해 디폴트 값을 전달해준다. 이제 오버르드가 제대로 해결되므로 처음의 자바 코드에서 오류가 났던 함수 호출 부분이 모두 제대로 컴파일된다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 System.out.println(UtilKt.restrictToRange(100, 1, 10)); // 10  
 System.out.println(UtilKt.restrictToRange(100, 1)); // 100  
 System.out.println(UtilKt.restrictToRange(100)); // 100  
 }  
}

</코드>

@JvmOverloads 애너테이션에 의해 생성되는 오버로드된 함수들이 컴파일된 바이너리상에 존재하기는 하지만, 코틀린에서 이런 함수를 호출할 수는 없다는 점에 유의하라. 이런 오버로드된 함수는 오직 자바와 상호 운용하려는 목적으로 추가된 것이다.

## 예외 선언하기

**3장 함수 정의하기**에서 코틀린이 검사 예외와 비검사 예외를 구분하지 않는다고 설명했다. 함수나 프로퍼티는 예외의 유형과 무관하게 (throws 절 등의) 코드를 추가할 필요 없이 예외를 그냥 던지면 된다. 반면 자바에서는 명시적으로 함수 본문에서 처리하지 않고 외부로 던져지는 검사 예외 목록을 함수에 추가해야 한다. 이로 인해 자바 코드가 코틀린 선언을 호출하면서 코틀린 선언에서 발생하는 검사 예외를 처리하고 싶을 때 문제가 생긴다. 예를 들어 다음과 같은 코틀린 함수가 있다고 하자.

<코드>

// util.kt  
fun loadData() = File("data.txt").readLines()

</코드>

이 함수를 자바 쪽에서 사용한다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 for (String line :UtilKt.loadData()) {  
 System.out.println(line);  
 }  
 }  
}

</코드>

data.txt 파일을 읽을 수 없으면 loadData()는 IOException을 던지고, 이 예외는 조용히 main() 스레드를 중단시킨다. main()에 예외 처리를 추가하려고 시도하면 다른 문제가 생긴다.

<코드>

import java.io.IOException;  
  
public class Main {  
 public static void main(String[] args) {  
 try {  
 for (String line :UtilKt.loadData()) {  
 System.out.println(line);  
 }  
 } catch (IOException e) { // Error  
 System.out.println("Can't load data");  
 }  
 }  
}

</코드>

자바는 try 블록 안의 코드에서 발생하는 것으로 선언되지 않은 검사 예외를 catch로 처리하는 것을 금지하므로 이 코드는 컴파일 오류가 난다. 문제는 자바 관점에서 loadData() 함수가 다음과 같이 보인다는 점이다.

<코드>

@notNull  
public List<String>loadData() {...}

</코드>

따라서 이 함수는 자신이 던지는 예외에 대해 아무 정보도 제공하지 않는다. 해결 방법은 @Throws 애너테이션을 사용해 예외 클래스를 지정하는 것이다.

<코드>

// util.kt  
@Throws(IOException::class)  
fun loadData() = File("data.txt").readLines()

</코드>

이제 자바 try-catch 블록에서 이 예외를 제대로 처리할 수 있다. 이 함수 또는 명시적으로 throws IOException이 붙은 함수를 예외 핸들러 밖에서 호출하면 예상대로 컴파일 오류가 난다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 // Error: Unhandled IOException  
 for (String line :UtilKt.loadData()) {  
 System.out.println(line);  
 }  
 }  
}

</코드>

코틀린 컴파일러는 기반 클래스의 멤버와 이 멤버를 오버라이딩한 자식 클래스의 멤버에 붙은 @Throws 애너테이션의 예외 선언의 일관성을 검증하지 않는다. 예를 들어 다음과 같이 써도 된다.

<코드>

import java.io.File  
import java.io.IOException  
  
abstract class Loader {  
 abstract fun loadData(): List<String>  
}  
  
class FileLoader(val path: String) : Loader() {  
 @Throws(IOException::class)  
 override fun loadData() = File(path).readLines()  
}

</코드>

하지만 자바 언어 명세가 오버라이드한 메서드에서 throws에 검사 예외를 추가하는 것을 금지하기 때문에 자바 소스 코드에서는 이런 선언을 사용할 수 없다.

## 인라인 함수

자바에는 인라인 함수가 없기 때문에 코틀린에서 inline 변경자가 붙은 함수는 일반 메서드로 자바 쪽에 노출된다. 자바 코드에서 이런 메서드를 호출할 수 있지만 이 경우 인라인 함수의 본문이 호출하는 자바 코드로 인라인되지는 않는다.

특별한 경우로 구체화한 타입 파라미터가 있는 제네릭 인라인 함수를 들 수 있다. 현재는 인라인을 사용하지 않고 타입 구체화를 구현할 방법이 없기 때문에 자바 코드에서 이런 함수를 호출하는 것이 불가능하다. 예를 들어 다음 cast() 함수는 자바 클라이언트에서 쓸모가 없다.

<코드>

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

</코드>

이 함수는 퍼사드 클래스의 비공개 멤버로 노출된다. 따라서 외부에서는 이 함수를 호출할 방법이 없다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 UtilKt.<Integer>cast(""); // Error: cast is private  
 }  
}

</코드>

## 타입 별명

코틀린 타입 별명을 자바 코드에서 쓸 수는 없다. 자바에서 볼 때 타입 별명을 참조하는 선언은 모두 원래 타입을 가리키는 것으로 보인다. 예를 들어 JVM 관점에서 보면 다음 Person 정의는 Name이라는 별명을 String으로 치환한 Person 클래스로 보인다.

<코드>

typealias Name = String  
class Person(val firstName: Name, val familyName: Name)

</코드>

자바 코드에서 Person 클래스의 인스턴스를 생성하고 사용하면 이 사실을 쉽게 알 수 있다.

<코드>

public class Main {  
 public static void main(String[] args) {  
 Person person = new Person("John", "Doe");  
 System.out.println(person.getFamilyName()); // Doe  
 }  
}

</코드>

## 결론

이 장에서는 코틀린과 자바 코드를 공통 코드 기반 내에서 엮는 방법에 대해 설명했다. 코틀린과 자바 선언이 서로에게 어떤 식으로 노출되는지 살펴봤다. 노출 방식의 차이로 인해 자바 선언을 코틀린 코드에서 사용하려고 하거나, 코틀린 선언을 자바 코드에서 사용하려고 하면 여러 가지 공통적인 문제가 발생하는데, 이러한 문제를 해결하는 기본적인 해법을 알아보고 JVM 플랫폼에서 자바와 코틀린의 상호 운용성을 세밀하게 조정하는 방법에 대해 설명했다.

다음 장에서는 동시성 애플리케이션에 초점을 맞춘다. 자바 동시성 기본 요소들을 코틀린에서 사용하는 방법을 살펴보고, 일시 중단 가능한 계산을 프로그래밍할 수 있게 해주는 강력한 언어 기능인 코루틴의 여러 측면을 다룬다.

## 문제

1. 합성 프로퍼티란 무엇인가? 자바 접근자 메서드를 코틀린에서 사용할 때 적용되는 규칙은 무엇인가?
2. 플랫폼 타입이란 무엇인가? 코틀린이 지원하는 플랫폼 타입의 종류를 설명하라.
3. 자바 코드의 널 가능성 애너테이션이 코틀린 타입에 어떤 영향을 끼치는가?
4. 코틀린 타입이 자바 타입으로 매핑되는 방식과 자바 타입이 코틀린 타입으로 매핑되는 방식에 대해 설명하라.
5. 코틀린의 SAM 변환과 SAM 생성자에 대해 설명하라. 코틀린의 함수형 인터페이스에 대해 설명하라.
6. 코틀린 프로퍼티를 자바 코드에서 어떻게 접근할 수 있는가?
7. 자바 코드 쪽에서 코틀린 프로퍼티를 뒷받침하는 필드에 접근할 수 있는 경우는 언제인가?
8. 파일 퍼사드란 무엇인가? 코틀린 최상위 함수와 프로퍼티를 자바에서 사용하는 방법에 대해 설명하라.
9. 여러 코틀린 파일에 있는 최상위 함수와 프로퍼티 선언을 한 파사드 클래스로 합치는 방법은 무엇인가?
10. @JvmName 애너테이션 사용법을 설명하라.
11. 코틀린 객체 선언의 인스턴스가 자바 코드에 어떤 식으로 노출되는지 설명하라.
12. 코틀린 객체 멤버를 자바에서 정적 멤버로 접근할 수 있게 하려면 어떻게 해야 하는가?
13. @JvmOverloads 애너테이션은 어떤 효과가 있는가?
14. 코틀린 함수에서 발생할 수 있는 검증 오류를 어떻게 기술할 수 있는가?