# 8장 클래스 계층 이해하기

이 장에서는 **4장 클래스와 객체 다루기**와 **6장 특별한 클래스 사용하기**에서 소개한 코틀린 객체 지향 측면을 계속해서 다룬다. 클래스 상속 개념을 소개하고 하위 클래스를 정의하는 방법을 설명한다. 추상 클래스, 인터페이스, 클래스 위임을 통해 복잡한 클래스 계층구조를 설계하는 방법에 대해서도 살펴본다. 그리고 클래스 계층 구조를 제한하고 코틀린의 강력한 스마트 캐스팅을 사용할 수 있는 타입 검사를 가능하게 해주는 대수적 데이터 타입(algebraic data type, ADT) 개념을 구현하는 봉인된 클래스에 대해서도 알아본다.

## 구조

* 상속과 오버라이딩
* 타입 검사와 캐스트
* 추상 클래스
* 인터페이스
* 봉인된 클래스
* 위임

## 목표

코틀린에서 상속과 오버라이딩이 어떻게 작동하는지 이해하고, 코틀린 객체 지향 기능을 사용해 클래스 계층 구조를 구축하는 방법을 배운다.

## 상속

도메인 개념에 있는 is-a 관계(‘자동차는 교통수단이다’와 같이 A는 B의 일종이라는 관계)를 표현하기 위해 대부분의 객체 지향 언어는 상속(inheritance)이라는 개념을 사용한다. 클래스 A(하위 클래스(subclass)나 파생클래스(derived class)라고 부름)가 클래스 B(상위 클래스(superclass)나 기반 클래스(base class)라고 부름)를 상속하면 A의 모든 인스턴스는 자동으로 B의 인스턴스로 간주된다. 이로 인해 클래스 A는 B에 정의된 모든 멤버와 확장을 자동으로 얻는다. 이 관계는 추이적(transitive)이다. 즉, 클래스 B가 어떤 클래스 C를 상속하면 A는 C의 (간접적인) 하위 클래스가 된다.

자바와 마찬가지로 코틀린 클래스는 단일 상속만을 지원한다. 이 말은 어떤 클래스의 상위 클래스가 최대 하나뿐이라는 뜻이다. 어떤 클래스의 상위 클래스를 명시하지 않으면 컴파일러는 자동으로 이 클래스가 내장 클래스인 Any를 상속하는 것으로 가정한다. 따라서 프로그램의 모든 클래스는 잘 정의된 상속 트리를 구성하게 된다. 이런 트리를 클래스 계층(class hierarchy)이라고 부른다.

다음 절부터는 코틀린의 상속에 대한 기본적인 내용을 다룬다. 하위 클래스를 정의하는 방법, 상위 클래스의 멤버를 상속하고 오버라이드(override)하는 방법, Any 클래스를 통해 모든 클래스에 제공되는 공통 메서드 등에 대해 다룬다.

### 하위 클래스 선언

어떤 클래스를 상속하려면 클래스를 정의하면서 주 생성자 뒤에 :를 넣고 그 뒤에 상위 클래스가 될 클래스의 이름을 넣으면 된다.

<코드>

open class Vehicle {  
 var currentSpeed = 0  
   
 fun start() {  
 println(“I’m moving”)  
 }  
   
 fun stop() {  
 println(“Stopped”)  
 }  
}  
  
open class FlyingVehicle : Vehicle() {  
 fun takeOff() {  
 println(“Taking off”)  
 }  
 fun land() {  
 println(“Landed”)  
 }  
}  
  
class Aircraft(val seats: Int) : FlyingVehicle()

</코드>

**자바와 코틀린의 차이**: 자바와 달리 코틀린에서는 extends나 implements와 같은 특별한 키워드를 사용하지 않는다. 상속은 항상 콜론(:)으로 표시된다.

Vehicle과 FlyingVehicle의 하위 클래스를 정의하면서 이 두 클래스의 이름 뒤에 괄호를 붙인 이유는 상위 클래스 생성자를 호출하기 위해서다. 상위 클래스 초기화에 필요한 인자가 있으면 이 괄호 사이에 넣으면 된다.

Vehicle와 FlyingVehicle 근처에 open 키워드를 발견했을 것이다. 이 변경자는 해당 클래스가 상속에 대해 열려있다는 뜻이다. 즉, 해당 클래스를 상위 클래스로 지정할 수 있다는 말이다. 반면 Aircraft는 아무 변경자가 붙어있지 않으므로, 디폴트로 상속을 할 수 없는 (자바) final 클래스로 간주된다. 따라서 이런 클래스를 상속하면 컴파일러가 오류를 발생시킨다.

<코드>

class Airbus(seats: Int) : Aircraft(seats) // Error: Aircraft is final

</코드>

**자바와 코틀린의 차이**: 자바와 코틀린 클래스의 디폴트 동작 차이(상속에 대해 열려있는지 여부)에 주의하라.

자바에서는 모든 클래스가 디폴트로 열려있으며 상속을 금지하려면 final을 명시해야 한다. 하지만 코틀린에서는 디폴트가 final이다. 따라서 어떤 클래스를 상속할 수 있게 허용하려면 open을 붙여서 선언해야 한다.

그 동안의 (자바) 실전 경험을 살펴보면 설계 시 상속을 염두에 두지 않고 만들어진 클래스들은 깨지기 쉬운 기반 클래스(fragile base class) 문제를 야기하는 경우가 많다. 이 문제는 기반 클래스를 변경했는데 하위 클래스에서 올바르지 못한 동작이 발생하는 경우를 말한다. 따라서 상속할 수 있는 클래스를 주의 깊게 설계하고 기반 클래스가 하위 클래스들이 지키리라 가정하고 있는 내용을 명시적으로 문서화할 것을 적극 권장한다.

하위 클래스의 인스턴스는 상위 클래스의 인스턴스이기도 하다. 하위 클래스 인스턴스는 상위 클래스의 멤버를 모두 상속한다.

<코드>

val aircraft = Aircraft(100)  
val vehicle: Vehicle = aircraft // 상위 타입으로 암시적으로 변환  
vehicle.start() // Vehicle의 메서드 호출  
vehicle.stop() // Vehicle의 메서드 호출  
aircraft.start() // Vehicle의 메서드 호출  
aircraft.takeOff() // FlyingVehicle의 메서드 호출  
aircraft.land() // FlyingVehicle의 메서드 호출  
aircraft.stop() // Vehicle의 메서드 호출  
println(aircraft.seats) // Aircraft 자체 프로퍼티 접근

</코드>

어떤 종류의 클래스들은 상속을 제한적으로 지원한다. 예를 들어 데이터 클래스는 항상 final이며, open으로 선언할 수 없다.

<코드>

open data class Person(val name: String,val age: Int) // Error

</코드>

처음에는 데이터 클래스가 다른 클래스를 상속하는 것도 금지되어 있었다. 하지만 이 제한은 코틀린 1.1부터 사라졌다.

반면 현재 인라인 클래스는 다른 클래스를 상속할 수도 없고, 다른 클래스의 상위 클래스 역할을 할 수도 없다.

<코드>

class MyBase  
open inline class MyString(val value: String) // Error  
inline class MyStringInherited(val value: String): MyBase() // Error

</코드>

객체(동반 객체 포함)는 자유롭게 열린 클래스를 상속할 수 있다.

<코드>

open class Person(val name: String, val age: Int) {  
 companion object : Person("Unknown", 0)  
}  
  
object JohnDoe : Person("John Doe", 30)

</코드>

하지만 객체를 상속하거나 객체를 open으로 선언할 수는 없다. 모든 객체는 인스턴스가 단 하나뿐이기 때문이다.

상속이 제공하는 강력한 기능은 임의 다형성(ad-hoc polymorphism)이다. 임의 다형성은 상위 클래스 멤버의 여러 다른 구현을 하위 클래스에서 제공하고, 런타임에 실제 인스턴스가 속한 클래스에 따라 구현을 선택해주는 기능을 말한다. 코틀린에서는 상위 클래스의 멤버를 오버라이드(override)해서 임의 다형성을 달성할 수 있다. 다음 클래스들을 살펴보자.

<코드>

open class Vehicle {  
 open fun start() {  
 println("I’m moving")  
 }  
 fun stop() {  
 println("Stopped")  
 }  
}  
  
class Car : Vehicle() {  
 override fun start() {  
 println("I'm riding")  
 }  
}  
  
class Boat : Vehicle() {  
 override fun start() {  
 println("I'm sailing")  
 }  
}

</코드>

Vehicle 클래스는 start() 메서드의 공통 구현을 제공한다. 그리고 Vehicle을 상속한 클래스인 Car와 Boat는 start()를 오버라이드한다. 여기서 Vehicle 클래스의 start() 메서드를 open으로 지정했다는 점에 유의하라. 메서드를 open으로 지정하면 하위 클래스에서 오버라이드할 수 있다. 하위 클래스인 Car와 Boat에서는 상위 클래스의 메서드를 오버라이드해 구현하는 메서드 앞에 override를 붙여야 한다. Vehicle 타입에 대한 메서드 호출은 런타임에 그 인스턴스의 클래스가 무엇이냐에 따라 달라진다. 다음 코드를 실행하면,

<코드>

fun startAndStop(vehicle: Vehicle) {  
 vehicle.start()  
 vehicle.stop()  
}  
  
fun main() {  
 startAndStop(Car())  
 startAndStop(Boat())  
}

</코드>

다음과 같은 결과를 볼 수 있다.

<코드>

I'm riding  
Stopped  
I'm sailing  
Stopped

</코드>

반면 stop() 메서드는 final이다. 이 메서드 앞에 명시적으로 open이 표시되어 있지 않기 때문에 이 메서드를 오버라이드할 수는 없고 하위 클래스는 이 메서드를 단순히 상속하기만 한다.

**자바와 코틀린의 차이**: 코틀린과 자바 상속의 두 가지 중요한 차이를 설명할 필요가 있다. 우선, 코틀린 함수와 프로퍼티는 기본적으로 final이며 하위 클래스에서 오버라이드하게 허용하려면 open을 명시해야 한다. 하지만 자바에서 메서드나 프로퍼티는 암시적으로 open이기 때문에 오버라이드를 막으려면 final 변경자를 사용해 이 사실을 명시해야 한다. 둘째로, 코틀린에서는 멤버를 오버라이드 하는 경우에 override 키워드를 앞에 붙여야만 한다. 이 키워드를 붙이지 않으면 컴파일러가 오류를 발생시킨다. 반면 자바에서는 @Override 애너테이션을 붙이는 편을 권장하기는 하지만, 오버라이드하는 메서드에 꼭 표시를 하지 않아도 된다. 코틀린에서 오버라이드하는 멤버 앞에 override를 붙이게 강제하는 이유는 실수로 상위 클래스의 함수를 오버라이드하는 일을 막기 위해서다. 여러분이 어떤 멤버를 추가했는데 우연히 이 멤버의 이름이 상위 클래스 멤버 이름과 같아서 구현을 오버라이드해버린다면 런타임 시 프로그램의 동작이 예상과 달라지고 찾기 어려운 버그가 생길 수 있다.

멤버와 확장의 중요한 차이점에 주목하자. 클래스 멤버는 (final이 아니라면) 오버라이드할 수 있고, 그에 따라 런타임에 인스턴스의 구체적인 타입에 따라 어떤 구현이 호출될지 결정할 수 있지만, 확장은 항상 정적으로 호출할 대상이 결정된다. 즉, 컴파일러는 항상 정적으로 알려진 수신 객체 타입을 기반으로 호출할 확장을 선택한다. 다음 예제를 보자.

<코드>

open class Vehicle {  
 open fun start() {  
 println("I'm moving")  
 }  
}  
  
fun Vehicle.stop() {  
 println("Stopped moving")  
}  
  
class Car : Vehicle() {  
 override fun start() {  
 println("I'm riding")  
 }  
}  
  
fun Car.stop() {  
 println("Stopped riding")  
}  
  
fun main() {  
 val vehicle: Vehicle = Car()  
 vehicle.start() // I’m riding  
 vehicle.stop() // Stopped moving  
}

</코드>

프로그램이 Car 클래스에 정의된 start()를 호출한다는 사실은 명확하다. 이 메서드 호출은 vehicle 변수의 런타임 타입에 의해 동적으로 결정되기 때문이다(여기서는 Car). 반면 stop()은 vehicle 변수의 정적 타입(여기서는 Vehicle)에 의해 결정되기 때문에 Vechicle.stop()이 호출된다.

오버라이드를 하는 멤버의 시그니처가 상위 클래스의 (오버라이드 당하는) 멤버 시그니처와 일치해야 한다.

<코드>

open class Vehicle {  
 open fun start(speed: Int) {  
 println("I’m moving at $speed")  
 }  
}  
  
class Car : Vehicle() {  
 override fun start() { // Error: wrong signature  
 println("I'm riding")  
 }  
}

</코드>

하지만 반환 타입을 더 상위타입으로 바꿀 수 있다.

<코드>

open class Vehicle {  
 open fun start(): String? = null  
}  
  
open class Car : Vehicle() {  
 final override fun start() = "I'm riding a car"  
}

</코드>

오버라이드하는 멤버를 final로 선언하면 더 이상 하위 클래스가 이 멤버를 오버라이드할 수 없다.

<코드>

open class Vehicle {  
 open fun start() {  
 println("I’m moving")  
 }  
}  
  
open class Car : Vehicle() {  
 final override fun start() {  
 println("I'm riding a car")  
 }  
}  
  
class Bus : Car() {  
 override fun start() { // Error: start() is final in Car  
 println("I'm riding a bus")  
 }  
}

</코드>

프로퍼티도 오버라이드할 수 있다. 하위 클래스 본문에 구현을 넣는 방법 외에 주 생성자 파라미터로 오버라이드할 수도 있다.

<코드>

open class Entity {  
 open val name: String get() = ""  
}  
  
class Person(override val name: String) : Entity()

</코드>

불변 프로퍼티를 가변 프로퍼티로 오버라이드할 수 있다[[1]](#footnote-1).

<코드>

open class Entity {  
 open val name: String get() = ""  
}  
  
class Person() : Entity() {  
 override var name: String = ""  
}

</코드>

자바와 마찬가지로 코틀린도 멤버의 영역을 하위 클래스의 영역으로만 제한하는 특별한 접근 변경자를 제공한다. 이런 멤버에는 protected라는 키워드가 붙는다.

<코드>

open class Vehicle {  
 protected open fun onStart() { }  
 fun start() {  
 println("Starting up...")  
 onStart()  
 }  
}  
  
class Car : Vehicle() {  
 override fun onStart() {  
 println("It's a car")  
 }  
}  
  
fun main() {  
 val car = Car()  
 car.start() // Ok  
 car.onStart() // Error: onStart is not available here  
}

</코드>

**자바와 코틀린 비교**: 코틀린과 자바의 protected 변경자의 차이에 대해 잘 알아둬야 한다. 두 언어 모두 하위 클래스에서 상위 클래스의 protected 멤버 접근을 허용하지만, 자바는 같은 패키지에 속한 아무 코드에서나 이런 멤버에 접근할 수 있다. 반면 코틀린에서는 이런 접근이 금지된다. 현재 코틀린에는 선언의 영역을 선언이 들어있는 패키지로 제한하는 변경자가 없다.

때때로 함수나 프로퍼티를 오버라이드한 버전이 원래 버전을 재사용해야 하는 경우가 있다. 이런 경우 멤버 참조 앞에 super 키워드를 붙이면 원래 버전을 참조할 수 있다(이 구문은 this를 사용하는 구문과 비슷하다. 다만 현재 클래스의 멤버가 아니라 상위 클래스에서 상속 받은 멤버에 접근한다는 점이 다르다).

<코드>

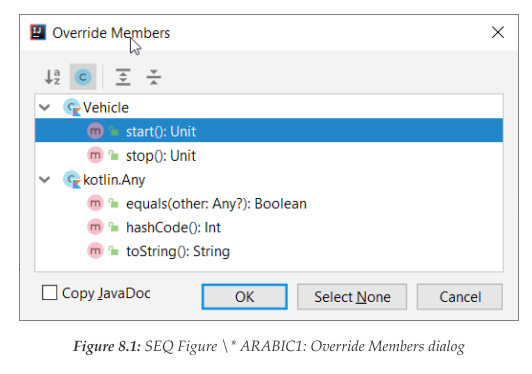
open class Vehicle {  
 open fun start(): String? = "I'm moving"  
}  
  
open class Car : Vehicle() {  
 override fun start() = super.start() + " in a car"  
}  
  
fun main() {  
 println(Car().start()) // I’m moving in a car  
}

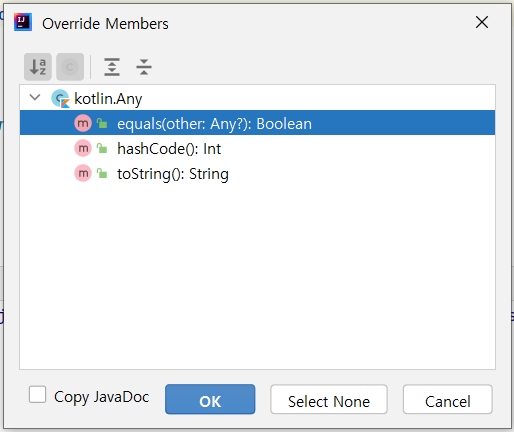
</코드>

<참고> IDE 팁

인텔리J 코틀린 플러그인은 오버라이드하는 멤버에 대한 골격 코드를 자동으로 생성해주는 기능이 있다. 이 기능을 사용하려면 클래스 본문 안에서 [Ctrl]+[O]나 [Cmd]+[O]를 눌러야 한다. 이 키를 누르면 IDE가 대화창을 띄우고, 그 대화창에서 오버라이드하고 싶은 상위 클래스 멤버를 선택할 수 있다(그림 8-1).

그림 8-1: 멤버 오버라이드 대화창





</참고>

### 하위 클래스 초기화

**4장 클래스와 객체 다루기**에서는 생성자를 사용해 어떤 클래스의 인스턴스 상태를 초기화하는 방법을 배웠다. 프로그램은 하위 클래스의 인스턴스를 생성하는 동안에 상위 클래스에 정의된 초기화 코드를 호출해야 한다. 상위 클래스에서 초기화하는 상태가 하위 클래스 코드가 사용할 환경에 되기 때문에 항상 상위 클래스 초기화를 먼저 시행해야 한다. 코틀린에서는 이 순서가 자동으로 지켜진다. 여러분의 프로그램이 어떤 클래스 A의 인스턴스를 생성하려고 시도하면 A는 자신의 생성자를 처리하기 전에 상위 클래스 생성자를 호출하고, 다시 이 상위 클래스 생성자는 자신의 상위 클래스 생성자를 호출하며, 이런 호출이 최상위(즉 Any 클래스) 클래스에 이를 때까지 연쇄적으로 일어난다. 그 결과 처음에는 Any의 초기화 코드가 실행되고, 그 다음에는 Any를 바로 상속한 선조 클래스의 초기화 코드가 실행되며, 이런 초기화 코드 실행은 A 초기화 코드가 실행되면서 끝난다. 초기화 순서를 보여주는 예제를 살펴보자.

<코드>

open class Vehicle {  
 init {  
 println("Initializing Vehicle")  
 }  
}  
  
open class Car : Vehicle() {  
 init {  
 println("Initializing Car")  
 }  
}  
  
class Truck : Car() {  
 init {  
 println("Initializing Truck")  
 }  
}  
  
fun main() {  
 Truck()  
}

</코드>

실행하면 이 코드는 다음을 출력한다.

<코드>

Initializing Vehicle  
Initializing Car  
Initializing Truck

</코드>

이 결과는 상위 클래스로부터 하위 클래스 순서로 초기화가 진행된다는 사실을 확인해준다.

앞에서 상위 클래스 이름 뒤에 있는 괄호가 생성자 호출을 구성한다고 설명했다. 지금까지 사용한 모든 상위 클래스는 디폴트 생성자를 사용했기 때문에 예제에서 인자를 상위 클래스 생성자에 넘기지 않았다. 하지만 데이터를 상위 클래스 생성자에게 전달해야 한다면 어떻게 해야 할까? 가장 간단한 경우는 상위 클래스에 생성자가 하나뿐인 경우다.

<코드>

open class Person(val name: String, val age: Int)  
  
class Student(name: String, age: Int, val university: String) :  
 Person(name, age)

fun main() {  
 Student("Euan Reynolds", 25, "MIT")  
}

</코드>

이 예제에서 Student 클래스의 주 생성자는 위임 호출(delegating call)이라고 하는 Person(firstName, familyName, age) 호출을 사용해 자신의 파라미터 중 세 가지를 Person 상위 클래스에게 넘긴다.

일반 생성자 호출과 마찬가지로 위임 호출도 주 생성자나 부 생성자에 모두 적용할 수 있다.

<코드>

open class Person {  
 val name: String  
 val age: Int  
   
 constructor(name: String, age: Int) {  
 this.name = name  
 this.age = age  
 }  
}  
  
class Student(name: String, age: Int, val university: String) :  
 Person(name, age)

</코드>

Student 클래스에서 부 생성자를 사용하고 싶다면 어떻게 해야 할까? 이 경우에는 위임 호출을 생성자 시그니처 바로 뒤에 위치시켜야 한다.

<코드>

open class Person(val name: String, val age: Int)  
  
class Student : Person {  
 val university: String  
 constructor(name: String, age: Int, university: String) :  
 super(name, age) {  
 this.university = university  
 }  
}

</코드>

super 키워드는 컴파일러에게 부 생성자가 상위 클래스의 생성자를 위임 호출한다는 사실을 컴파일러에게 알려준다. 이 구문은 같은 클래스의 다른 생성자를 호출하는 this 키워드 사용법과 비슷하다(**4장 클래스와 객체 다루기**를 보라). 주 생성자를 호출하는 경우와 비교할 때 다른 차이는 수퍼클래스 이름 다음에 괄호가 없다는 점이다. 즉, Person() 대신 Person을 사용한다. 이렇게 하는 이유는 우리가 정의하고 있는 클래스에 주 생성자가 없어서 부 생성자에서 위임 호출을 해야만 하기 때문이다.

**자바와 코틀린의 차이**: 자바와 달리 생성자간의 호출(생성자가 같은 클래스에 속했든 상위 클래스에 속했든 관계 없이)이 생성자 본문에 들어가는 일은 결코 없다. 코틀린에서는 이를 위해 항상 위임 호출 구문을 사용해야 한다.

클래스에 주 생성자가 있으면 부 생성자가 상위 클래스를 위임 호출할 수 없다는 점에 유의하라.

<코드>

open class Person(val name: String, val age: Int)  
  
// Error: call to Person constructor is expected  
class Student() : Person {  
 val university: String  
   
 constructor(name: String, age: Int, university: String) :  
 super(name, age) { // Error: can’t invoke Person constructor here  
 this.university = university  
 }  
}

</코드>

흥미로운 경우로 상위 클래스가 여러 생성자를 제공하고 하위 클래스에서 상위 클래스의 생성자 중 둘 이상을 지원하고 싶을 때를 생각해보자. 이 경우 부 생성자를 사용하는 것이 유일한 해법이다.

<코드>

open class Person {  
 val name: String  
 val age: Int  
   
 constructor(name: String, age: Int) {  
 this.name = name  
 this.age = age  
 }  
   
 constructor(firstName: String, familyName: String, age: Int) :  
 this("$firstName $familyName", age)  
 }  
}  
  
class Student : Person {  
 val university: String  
   
 constructor(name: String, age: Int, university: String) :  
 super(name, age) {  
 this.university = university  
 }  
   
 constructor(  
 firstName: String,  
 familyName: String,  
 age: Int,  
 university: String  
 ) :  
 super(firstName, familyName, age) {  
 this.university = university  
 }  
}  
  
fun main() {  
 Student("Euan", "Reynolds", 25, "MIT")  
 Student("Val Watts", 22, "ETHZ")  
}

</코드>

사실은 방금 본 경우가 코틀린 언어가 부생성자를 추가한 여러 가지 이유 중 하나다. 특히 주 생성자와 부 생성자를 구별하지 않는 자바 코드와의 상호 운용성을 고려한다면 이런 방식이 중요해진다.

이 절에서 강조하고 싶은 또 다른 문제로 this 누출(leaking this) 문제를 들 수 있다. 다음 코드를 보자.

<코드>

open class Person(val name: String, val age: Int) {  
 open fun showInfo() {  
 println("$name, $age")  
 }  
   
 init {  
 showInfo()  
 }  
}  
  
class Student(  
 name: String,  
 age: Int,  
 val university: String  
) : Person(name, age) {  
 override fun showInfo() {  
 println("$name, $age (student at $university)")  
 }  
}  
  
fun main() {  
 Student("Euan", "Reynolds", 25, "MIT")  
}

</코드>

이 프로그램을 실행하면 다음과 같은 출력을 볼 수 있다.

<코드>

Euan Reynolds, 25 (student at null)

</코드>

왜 university 변수가 null일까? showInfo()가 상위 클래스의 초기화 코드에서 호출되기 때문이다. 이 함수는 가상 함수(virtual function)이기 때문에, 프로그램은 런타임 객체의 실제 타입인 Student 클래스가 오버라이드한 shoInfo()를 호출한다. 하지만 Person 초기화 코드는 Student가 초기화되기 전에 실행되기 때문에 shoInfo()가 호출되는 시점에 university 변수는 아직 초기화 되지 않은 상태다. 이런 상황을 this 누출이라고 부르는 이유는 상위 클래스가 현재의 인스턴스를 코드에 누출하는데, 현재 인스턴스는 일반적으로 아직 초기화되지 않은 인스턴스의 상태에 의존할 수 있기 때문이다. 더 명확한 예제는 다음과 같다.

<코드>

open class Person(val name: String, val age: Int) {  
 override fun toString() = "$name, $age"  
   
 init {  
 println(this) // 잠재적인 위험 요소  
 }  
}  
  
class Student(  
 name: String,  
 age: Int,  
 val university: String  
) : Person(name, age) {  
 override fun toString() = super.toString() + "(student at $university)"  
}  
  
fun main() {  
 // Euan Reynolds, 25 (student at null)  
 Student("Euan Reynolds", 25, "MIT")  
}

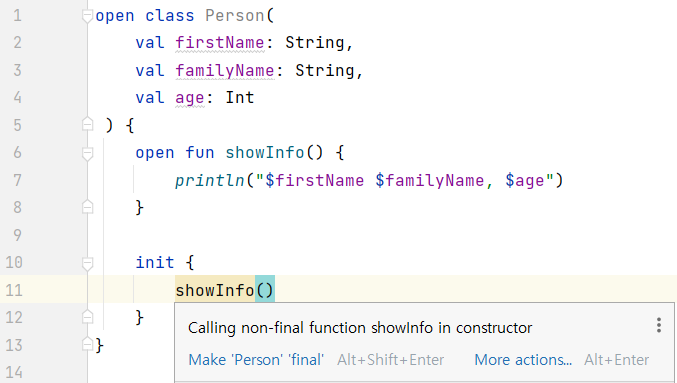
</코드>

this 누출 문제는 코틀린의 널이 될 수 없는 타입의 변수 값이 널이 될 수도 있는 아주 드문 경우다.

<참고> IDE 팁

인텔리J 코틀린 플러그인은 코드를 분석해 본문에서 설명한 것처럼 잠재적으로 안전하지 않은 호출을 알려주며 적절한 경고를 표시해준다(그림 8-2).

그림 8-2: 생성자 안에서 final이 아닌 함수를 호출한 경우에 대한 경고



</참고>

## 타입 검사와 캐스팅

어떤 클래스의 변수가 런타임에는 해당 클래스의 하위 타입 중 아무 타입의 객체나 가리킬 수 있기 때문에, 어떤 인스턴스가 더 구체적인 타입에 속하는지 검사하고, 필요할 때 타입을 변환할 수 있는 방법이 있으면 편리하다. 예를 들어 다음 코드를 살펴보자.

<코드>

val objects = arrayOf(“1”, 2, “3”, 4)

</코드>

컴파일러 관점에서 보면 Any가 문자열과 정수를 아우르는 최소한의 공통 상위 타입이기 때문에, 이 코드의 objects는 Any로 이뤄진 배열이다. 하지만 String이나 Int에만 사용할 수 있는 연산을 이 배열의 원소에 사용하고 싶다면 어떻게 해야 할까? 연산을 배열 원소에 직접 적용하면 원소 타입이 이런 연산을 지원하지 않는 Any 타입이기 때문에 컴파일이 되지 않는다.

<코드>

for (obj in objects) {  
 println(obj\*2) // 오류: Any는 \*를 지원하지 않음  
}

</코드>

코틀린은 타입 검사와 캐스팅 연산을 통해 이런 경우를 처리할 수 있는 해법을 제공한다. is 연산자는 왼쪽 피연산자가 오른쪽에 주어진 타입인 경우 true를 반환한다. 앞에서 본 예제를 약간 바꿔보자.

<코드>

for (obj in objects) {  
 println(obj is Int)  
}

</코드>

이 프로그램을 실행하면 다음과 같은 출력을 볼 수 있다.

<코드>

false  
true  
false  
true

</코드>

예상대로 null 값은 모든 널의 될 수 있는 타입의 인스턴스로 간주되지만, 모든 널이 될 수 없는 타입의 인스턴스는 아닌 것으로 간주된다.

<코드>

println(null is Int) // false  
println(null is String?) // true

</코드>

코틀린은 !is 연산자를 통해 is와 반대인 연산을 제공한다.

val o: Any = ""  
println(o !is Int) // true  
println(o !is String) // false

is나 !is 연산자의 왼쪽 피연산자의 정적 타입이 오른쪽에 오는 타입의 상위 타입인 경우에만 두 연산자를 사용할 수 있다. Int 타입(또는 그 하위 타입) 값의 타입을 String 타입과 비교하는 것은 의미가 없기 때문에, 컴파일러는 정적으로 String이 Int의 하위 타입이 아니라는 사실을 알고 컴파일 오류를 발생시킨다.

<코드>

println(12 is String) // Error

</코드>

is와 !is 연산은 in이나 !in과 똑같은 우선순위에 속한다.

**자바와 코틀린의 차이**: is 연산자는 자바의 instanceof 연산자와 매우 비슷하다. 하지만 이 두 연산이 널을 취급하는 방법은 아주 다르다는 점을 명심하라. instanceof 연산자는 null에 대해 항상 false를 반환하지만, is의 결과는 연산자 오른쪽에 있는 타입이 널이 될 수 있느냐 여부에 따라 결과가 달라진다.

**4장 클래스와 객체 다루기**에서는 널이 될 수 있는 타입의 값을 null과 비교한 경우, 값의 타입을 자동으로 세분화해서 널이 될 수 없는 타입으로 바꿔주는 스마트 캐스트에 대해 설명했다. 이런 유용한 기능을 is/!is에도 사용할 수 있다. 다음 예를 보자.

<코드>

val objects = arrayOf("1", 2, "3", 4)  
  
var sum = 0  
  
for (obj in objects) {  
 if (obj is Int) {  
 sum += obj // 여기서는 obj의 타입을 `Int`로 세분화한다  
 }  
}  
println(sum) // 6

</코드>

is/!is 검사를 in이나 !in처럼 특별한 조건으로 사용할 수 있는 식 내부에서도 스마트 캐스트가 지원된다.

<코드>

val objects = arrayOf("1", 2, "3", 4)  
var sum = 0  
  
for (obj in objects) {  
 when (obj) {  
 is Int -> sum += obj // 여기서 obj는 Int 타입이다  
 is String -> sum += obj.toInt() // 여기서 obj는 String 타입이다  
 }  
}  
println(sum) // 10

</코드>

앞에서 이미 말했지만, 컴파일러는 검사 시점과 사용 시점 사이에 변수가 변경되지 않는다고 확신할 수 있을 때에만 스마트 캐스트를 허용한다. 이제 스마트 캐스트 규칙을 좀 더 정확하게 표현할 수 있다.

먼저 프로퍼티나 커스텀 접근자가 정의된 변수에 대해서는 스마트 캐스트를 쓸 수 없다. 컴파일러가 해당 변수를 검사한 다음에 값이 바뀌지 않는다고 보장할 수 없기 때문이다. 이런 유형의 변수에는 위임을 사용하는 프로퍼티나 지역 변수도 포함된다.

<코드>

class Holder {  
 val o: Any get() = ""  
}  
  
fun main() {  
 val o: Any by lazy { 123 }  
   
 if (o is Int) {  
 println(o\*2) // Error: 스마트 캐스트를 할 수 없음  
 }  
   
 val holder = Holder()  
   
 if (holder.o is String) {  
 println(holder.o.length) // Error: 스마트 캐스트를 할 수 없음  
 }  
}

</코드>

열린 멤버 프로퍼티의 경우, 하위 타입에서 이런 프로퍼티를 오버라이드하면서 커스텀 접근자를 추가할 수 있기 때문에 스마트 캐스트를 할 수 없는 범주에 속한다.

<코드>

open class Holder {  
 open val o: Any = ""  
}  
  
fun main() {  
 val holder = Holder()  
 if (holder.o is String) {  
 println(holder.o.length) // Error: 스마트 캐스트를 할 수 없음  
 }  
}

</코드>

가변 지역 변수의 경우 검사 시점과 변수를 읽는 시점 사이에 값을 명시적으로 변경하거나, 어떤 람다 안에서 변수를 변경하면 스마트 캐스트가 되지 않는다(람다 내부에서 변수를 변경하는 코드의 경우, 일반적으로 런타임의 어느 시점에 변수가 변경될지 예측할 수 없다).

<코드>

fun main() {  
 var o: Any = 123  
 if (o is Int) {  
 println(o + 1) // Ok: Int로 스마트 캐스트  
 o = ""  
 println(o.length) // Ok: String으로 스마트 캐스트  
 }  
 if (o is String) {  
 val f = { o = 123 }  
 println(o.length) // Error: 스마트 캐스트 불가능  
 }  
}

</코드>

반면 다른 코드에서 언제든 변경이 가능한 가변 프로퍼티는 스마트캐스트 대상이 아니다.

위임이 없는 불변 지역 변수는 항상 스마트캐스트가 가능하며, 이 사실이 불변 변수를 가변 변수보다 더 선호해야 하는 이유이기도 하다는 점을 명심하라.

하지만 스마트캐스트를 쓸 수 없는 경우에도 명시적인 연산자를 사용해 어떤 값의 타입을 강제로 변환(coerce)할 수 있다. 코틀린은 이런 연산자로 안전하지 않은 as와 더 안전한 버전인 as?를 제공한다. 이 둘의 차이는 객체의 실제 타입이 변환하려는 대상 타입과 일치하지 않을 때 이를 어떻게 처리하느냐에 있다. as는 예외를 던지지만 as?는 null을 돌려준다.

<코드>

val o: Any = 123  
println((o as Int) + 1) // 124  
println((o as? Int)!! + 1) // 124  
println((o as? String ?: "").length) // 0  
println((o as String).length) // 예외

</코드>

o as String?과 o as? String 식의 차이를 구분하라. o가 String? 타입의 값이라면 이 두 식이 똑같은 값을 가질 수 있지만(null을 포함함) o가 널이 될 수 없는 타입의 값이라면 그렇지 않다.

<코드>

val o: Any = 123  
println(o as? String) // null  
println(o as String?) // 예외

</코드>

널을 널이 될 수 없는 타입으로 변환하려고 하면 런타임에 오류가 발생한다.

<코드>

println(null as String) // 오류

</코드>

**자바와 코틀린의 차이**: as 연산자는 자바의 캐스팅 식과 같지만 null 처리가 다르다. 자바에서는 캐스팅을 해도 항상 널은 널로 남지만, 코틀린에서는 대상 타입의 널 가능성에 따라 예외가 발생할 수도 있고 널이 될 수도 있다.

## 공통 메서드

kotlin.Any 클래스는 코틀린 클래스 계층 구조의 루트다. 즉, 다른 모든 클래스는 Any를 직간접적으로 상속한다. 클래스를 정의하면서 상위 클래스를 명시하지 않으면 컴파일러가 자동으로 상위 클래스를 Any로 가정한다. 따라서 Any의 멤버는 모든 코틀린 값에 존재한다. Any의 정의를 살펴보자.

<코드>

open class Any {  
 public open operator fun equals(other: Any?): Boolean  
 public open fun hashCode(): Int  
 public open fun toString(): String  
}

</코드>

operator 키워드는 equals() 메서드가 연산자 형태(==나 !=)로 호출될 수 있다는 뜻이다. 이에 대해서는 **11장 도메인 특화 언어**에서 살펴본다.

이 메서드들은 널이 될 수 없는 타입의 값에 대해 적용할 수 있는 기본 연산을 정의한다.

* 구조적 동등성(==와 !=)
* 해시 코드 계산. HashSet, HashMap 등의 일부 컬렉션 타입이 해시코드를 사용한다.
* String으로 변환하는 기본적인 방법

**자바와 코틀린의 차이**: 자바를 잘 안다면 Any 정의가 java.lang.Object를 최소화한 버전이라는 점을 눈치챘을 것이다. 실제로 JVM에서 Any의 런타입 값은 Object 인스턴스로 표현된다.

**6장 특별한 클래스 사용하기**에서 데이터 클래스에 대해 설명하면서 참조 동등성(referential equality)과 구조적 동등성의 예제를 살펴봤다. 컴파일러는 클래스에 대해 참조 동등성과 데이터 클래스에 대한 구조적 동등성을 제공한다. 이제 임의의 코틀린 클래스에 대한 커스텀 동등성 연산을 구현해보자. 다음 코드를 보라.

<코드>

class Address(  
 val city: String,  
 val street: String,  
 val house: String  
)  
   
open class Entity(  
 val name: String,  
 val address: Address  
)  
   
class Person(  
 name: String,  
 address: Address,  
 val age: Int  
): Entity(name, address)  
  
class Organization(  
 name: String,  
 address: Address,  
 val manager: Person  
) : Entity(name, address)

</코드>

디폴트로 여기 정의된 모든 클래스에는 Any에서 상속받은 참조 동등성만 구현한다. 예를 들어 컬렉션 객체로 이런 클래스들의 인스턴스를 사용하면 프로퍼티가 똑같더라도 두 인스턴스가 같은 객체로 간주되지 않는다.

<코드>

fun main() {  
 val addresses = arrayOf(  
 Address("London", "Ivy Lane", "8A"),  
 Address("New York", "Kingsway West", "11/B"),  
 Address("Sydney", "North Road", "129")  
 )  
   
 // -1  
 println(addresses.indexOf(Address("Sydney", "North Road", "129")))  
}

</코드>

equals() 메서드를 오버라이드해서 내용을 바탕으로 동등성을 비교하게 하면 이 문제를 해결할 수 있다. 간단히 다음과 같이 구현할 수 있다.

// Address에 정의한 equals 함수  
override fun equals(other: Any?): Boolean {  
 if (other !is Address) return false  
 return city == other.city &&  
 street == other.street &&  
 house == other.house  
}

이제 앞의 예제에서 index() 호출은 Address 객체를 찾을 수 있고 2를 반환한다.

==와 != 연산자가 공통으로 equals() 메서드를 사용한다는 점에 유의하라. 이 두 연산자를 널이 될 수 있는 값에도 적용할 수 있다. 연산자의 왼쪽 피연산자가 null인 경우에는 참조 동등성을 사용해 널과 비교하면 된다. 원래의 참조 동등성은 ===와 !==를 사용해 구현되어 있다. ==나 !=와 달리 이 참조 동등성 연산자의 동작을 사용자가 작성한 코드가 오버라이드할 수는 없다.

<코드>

val addr1 = Address("London", "Ivy Lane", "8A")  
val addr2 = addr1 // 같은 인스턴스  
val addr3 = Address("London", "Ivy Lane", "8A") // 다른 인스턴스지만, 동등함  
println(addr1 === addr2) // true  
println(addr1 == addr2) // true  
println(addr1 === addr3) // false  
println(addr1 == addr2) // true

</코드>

**자바와 코틀린의 차이**: 자바에서 ==와 != 연산자는 참조 동등성을 구현하며, 내용을 기반으로 하는 동등성은 equals() 호출을 통해 구현된다. 그리고 자바에서는 NPE를 방지하기 위해 equals()에서도 수신 객체가 널인 경우에 대한 보호를 추가해야 한다.

자바와 마찬가지로 equals() 메서드의 커스텀 구현은 대응하는 hashCode()와 서로 잘 조화돼야 한다. 두 구현은 서로 연관이 있어야 하고, equals()가 같다고 보고하는 두 객체는 항상 같은 hashCode()를 반환해야 한다. 이렇게 해야 하는 이유는 일부 컬렉션(HashSet 등)이 hashCode()를 사용해 해시 테이블에서 원소가 들어갈 슬롯을 먼저 찾고 그 후에 equals()를 통해 해시 코드가 같은 모든 후보를 검색하기 때문이다. (equals()입장에서 볼 때) 동등한 두 객체가 서로 다른 해시코드를 반환하면 이런 컬렉션은 두 객체에 대해 equals()를 호출하기도 전에 서로 다르다고 인식하고 검색에서 제외시켜 버리게 된다. 위에서 본 equals() 메서드와 호환이 되는 hashCode() 구현은 다음과 같을 수 있다.

<코드>

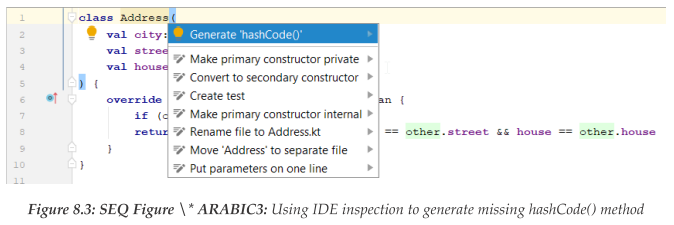
override fun hashCode(): Int {  
 var result = city.hashCode()  
 result = 31 \* result + street.hashCode()  
 result = 31 \* result + house.hashCode()  
 return result  
}

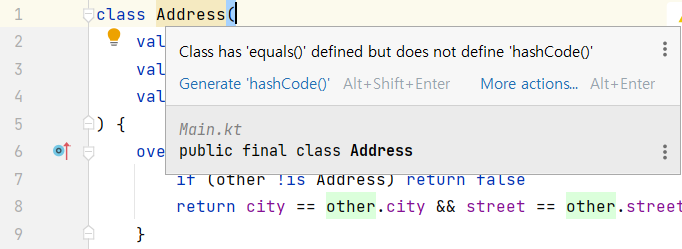
</코드>

<참고> IDE 팁

인텔리J 코틀린 플러그인은 equals() 구현을 제공하는데 hashCode()를 제공하지 않거나, equals() 구현을 제공하지 않는데 hashCode()구현을 제공하는 경우에 경고를 표시한다. 그리고 어느 정도 타당한 구현을 자동으로 생성해주기도 한다(그림 8-3).

그림 8-3: IDE 기능을 활용해 빠진 hashCode() 메서드 구현하기





</참고>

equals() 구현의 일반적인 요구사항은 기본적으로 자바와 같다.

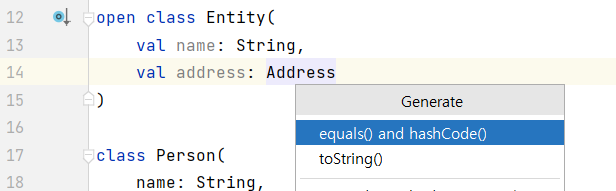
* 널이 아닌 객체가 널과 같을 수 없다.
* 동등성 연산은 반사적(reflexive)이어야 한다. 즉, 모든 객체는 자기 자신과 동등해야 한다.
* 동등성 연산은 대칭적(symmetric)이어야 한다. 즉, a == b이면 b == a여야 한다.
* 동등성 연산은 추이적(transitive)이어야 한다. 즉, a == b이고 b == c이면 a == c이어야 한다.

<참고> IDE 팁

인텔리J 코틀린 플러그인은 자동으로 클래스의 프로퍼티를 기반으로 equals()와 hashCode() 메서드 구현을 생성해준다. 이 메서드들은 데이터 클래스에 대해 컴파일러가 제공하는 메서드와 상당히 비슷하며, 대부분의 경우 타당한 동등성 동작을 보여준다. 이런 식으로 생성한 메서드로는 부족한 경우라 할지라도, 여러분 자신의 구현을 작성하기 위한 출발점으로 자동 생성된 메서드를 사용하는 것도 좋다.

메서드를 생성하려면 클래스 정의 안에서 [Alt]+[Insert]를 누르면 나오는 Generate 메뉴에서 equals()와 hashCode()를 선택하라(그림 8-4).

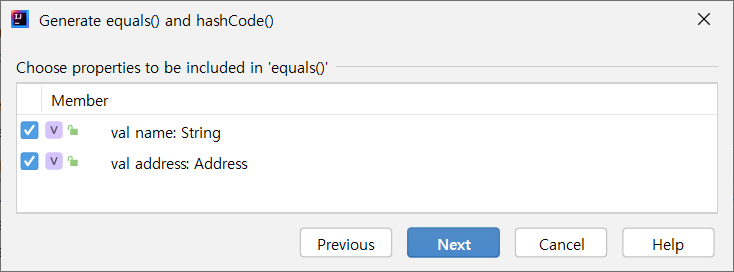
그림 8-4: SEQ Figure \\* ARABIC4: Generate menu



이때 클래스가 열린 클래스면 IDE가 하위 클래스의 인스턴스까지 서로 같은지 비교해주는 메서드를 생성하라고 제안한다. 이 제안에 동의하면 서로 다른 하위 클래스의 인스턴스들이 서로 같아질 수 있다. 하지만 이런 동작이 항상 바람직하지는 않다. 우리 예제에서는 Person과 Organization의 인스턴스를 서로 구분하고 싶기 때문에 이 옵션을 사용하지 않을 것이다.

그 후 (그림 8-5와 같이) 생성할 메서드에서 사용할 프로퍼티를 선택한다. equals()에서 선택한 프로퍼티들만 hashCode()에서 사용할 수 있다. 이렇게 해야 서로 동등한 두 객체의 해시코드가 항상 같아야 한다는 점에서 두 메서드가 서로 호환이 된다.

그림 8-5: equals() 메서드 구현에 사용할 프로퍼티 선택하기



</참고>

이 동작을 Entity 클래스에 대해 적용하면 다음과 같은 코드가 만들어진다[[2]](#footnote-2).

<코드>

open class Entity(  
 val name: String,  
 val address: Address  
) {  
 override fun equals(other: Any?): Boolean {  
 if (this === other) return true  
 if (javaClass != other?.javaClass) return false  
   
 other as Entity  
 if (name != other.name) return false  
 if (address != other.address) return false  
   
 return true  
 }  
   
 override fun hashCode(): Int {  
 var result = name.hashCode()  
   
 result = 31 \* result + address.hashCode()  
   
 return result  
 }  
}

</코드>

프로퍼티는 각자의 equals()와 hashCode() 구현에 위임하여 해시와 동등성을 계산한다. 배열 타입은 예외다. 배열에는 자체적인 내용 기반 동등성 구현이 없기 때문에 생성된 코드가 contentEquals()와 contentHashCode()를 사용한다(또는 다차원 배열인 경우 contentDeepEquals()와 contentDeepHashCode()를 사용한다).

상위 클래스가 커스텀 equals()/hashCode() 구현을 제공한다면, IDE에 의해 자동으로 생성되는 equals()/hashCode()는 해당 구현을 equals()/hashCode() 구현에서 호출해준다. 예를 들어 Person 클래스에 대해 Generate equals()/hashCode()를 적용하면 다음과 같은 코드를 얻는다.

<코드>

class Person(  
 name: String,  
 address: Address,  
 val age: Int  
): Entity(name, address) {  
 override fun equals(other: Any?): Boolean {  
 if (this === other) return true  
 if (javaClass != other?.javaClass) return false  
 if (!super.equals(other)) return false  
   
 other as Person  
 if (age != other.age) return false  
 return true  
 }  
   
 override fun hashCode(): Int {  
 var result = super.hashCode()  
 result = 31 \* result + age  
 return result  
 }  
}

</코드>

자바와 마찬가지로 모든 코틀린 클래스에는 toString() 메서드가 들어있다. 이 메서드는 주어진 인스턴스의 기본 문자열 표현을 제공한다. 디폴트 표현은 클래스 이름 뒤에 객체 해시 코드를 조합하는 방식이다. 따라서 대부분의 경우 이를 좀 더 읽기 좋은 표현으로 오버라이드하는 편이 좋다.

<코드>

class Address(  
 val city: String,  
 val street: String,  
 val house: String  
) {  
 override fun toString() = "$city, $street, $house"  
}  
  
open class Entity(  
 val name: String,  
 val address: Address  
)  
  
class Person(  
 name: String,  
 address: Address,  
 val age: Int  
): Entity(name, address) {  
 override fun toString() = "$name, $age at $address"  
}  
  
class Organization(  
 name: String,  
 address: Address,  
 val manager: Person?  
) : Entity(name, address) {  
 override fun toString() = "$name at $address"  
}  
  
fun main() {  
 // Euan Reynolds, 25 at London, Ivy Lane, 8A  
 println(Person("Euan Reynolds", Address("London", "Ivy Lane", "8A"), 25))  
   
 // Thriftocracy, Inc. at Perth, North Road, 129  
 println(  
 Organization(  
 "Thriftocracy, Inc.",  
 Address("Perth", "North Road", "129"),  
 null  
 )  
 )  
}

</코드>

<참고> IDE 팁

인텔리J는 equals()/hashCode() 메서드와 마찬가지로 간단한 toString() 구현을 생성해준다. 생성하려면 Generate 메뉴에서 toString()을 선택하고(그림 8-4 참고), toString()에서 사용할 프로퍼티를 선택하면 된다. 결과 문자열을 단일 문자열 템플릿으로 만들 수도 있고, 문자열을 이어 붙일 수도 있다. 상위 클래스에 커스텀 toString()이 있으면 super.toString() 호출을 추가할지 결정할 수 있다.

다음은 Generate toString() 액션을 Person 클래스에 적용한 결과다.

</참고>

<코드>

class Person(  
 val name: String,  
 val age: Int,  
 address: Address  
): Entity(address) {  
 override fun toString(): String {  
 return "Person(name='$name', age=$age) ${super.toString()}"  
 }  
}

</코드>

코틀린 표준 라이브러리에는 Any? 타입에 대한 toString() 확장 정의가 들어있다. 이 함수는 수신 객체가 널이 아니면 단순히 수신 객체의 toString()에게 문자열 생성을 위임하고, 널인 경우에는 "null"이라는 문자열을 반환한다. 이를 통해 널이 될 수 있는 값과 널이 될 수 없는 값 양쪽에 (안전한 호출을 사용하지 않더라도 NPE 발생을 걱정할 필요 없이) toString()을 사용할 수 있다.

## 추상 클래스와 인터페이스

지금까지 살펴본 모든 상위 클래스는 자기 자신에 대한 인스턴스를 만들 수 있는 타입들이었다. 하지만 별도의 인스턴스가 있을 수 없고, 구체적인 경우를 구현한 다른 클래스의 인스턴스만 만들 수 있는 추상적인 개념을 표현하는 클래스라면 인스턴스 생성은 바람직하지 않다. 예를 들어 우리가 살펴본 예제 중에는 Person과 Organization이 상속한 Entity 클래스가 있다. 하지만 엔티티 자체는 추상적인 개념이다. 따라서 구체적인 하위 클래스의 인스턴스가 아닌 Entity 인스턴스를 생성하는 것은 의미가 없다. 이제부터 코틀린에서 이런 추상적인 타입을 정의하고 사용하는 방법을 살펴보자.

### 추상 클래스와 추상 멤버

자바와 마찬가지로 코틀린도 추상(abstract) 클래스를 지원한다. 추상 클래스는 직접 인스턴스화할 수 없고 다른 클래스의 상위 클래스 역할만 할 수 있는 클래스를 뜻한다. 클래스를 추상 클래스로 만들려면 abstract라는 변경자 키워드를 붙여야 한다.

<코드>

abstract class Entity(val name: String)  
  
// Ok: 하위 클래스에서 위임 호출  
class Person(name: String, val age: Int) : Entity(name)  
  
val entity = Entity(“Unknown”) // Error: Entity를 인스턴스화할 수 없음

</코드>

앞의 예제를 보면 알 수 있듯이 추상 클래스에도 생성자가 있을 수 있다. 추상 클래스와 비-추상클래스의 차이는 추상 클래스의 생성자가 오직 하위 클래스의 생성자에서 위임 호출로만 호출될 수 있다는 점이다. 다음 코드에서 부 생성자는 추상 클래스의 생성자를 위임 호출한다.

<코드>

abstract class Entity(val name: String)  
  
class Person : Entity {  
 constructor(name: String) : super(name)  
 constructor(  
 firstName: String,  
 familyName: String  
 ) : super("$firstName $familyName")  
}

</코드>

추상 클래스의 또 다른 특징은 추상 멤버를 정의할 수 있다는 점이다. 추상 멤버는 타입, 파라미터, 반환 타입 등 함수나 프로퍼티의 기본적인 모습을 정의하지만 세부 구현을 생략한 멤버다. 비-추상 클래스가 추상적인 부모 클래스로부터 이런 멤버를 상속할 때는 반드시 멤버를 오버라이드해서 구현을 제공해야 한다.

<코드>

import kotlin.math.PI  
  
abstract class Shape {  
 abstract val width: Double  
 abstract val height: Double  
 abstract fun area(): Double  
}  
  
class Circle(val radius: Double) : Shape() {  
 val diameter get() = 2\*radius  
 override val width get() = diameter  
 override val height get() = diameter  
 override fun area() = PI\*radius\*radius  
}  
  
class Rectangle(  
 override val width: Double,  
 override val height: Double  
) : Shape() {  
 override fun area() = width\*height  
}  
  
fun Shape.print() {  
 println("Bounds: $width\*$height, area: ${area()}")  
}  
  
fun main() {  
 // Bounds: 20.0\*20.0, area: 314.1592653589793  
 Circle(10.0).print()  
   
 // Bounds: 3.0\*5.0, area: 15.0  
 Rectangle(3.0, 5.0).print()  
}

</코드>

추상 멤버 자체는 구현을 가질 수 없기 때문에 (추상 클래스에) 추상 멤버를 정의할 때는 몇 가지 제약이 있다.

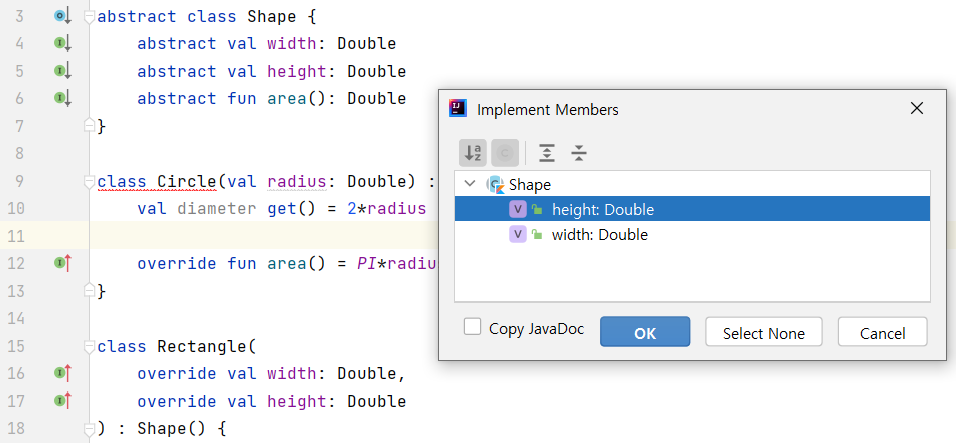
* 추상 프로퍼티를 초기화할 수 없고 명시적인 접근자나 by 절을 추가할 수 없다.
* 추상 함수에는 본문이 없어야 한다.
* 추상 프로퍼티와 함수 모두 명시적으로 반환 타입을 적어야 한다. 본문이나 초기화 코드가 없으므로 타입을 추론할 수 없기 때문이다.

추상 멤버는 암시적으로 열려있다는 점에 유의하라. 따라서 명시적으로 open을 지정할 필요가 없다.

<참고> IDE 팁

인텔리J는 Overriding Class Members 부분에 보이는 Override Members 액션과 비슷한, Implement Members라는 액션을 제공한다. [Ctrl]+[I] 단축키를 누르면 Override Members와 비슷한 대화창이 열리면서 이 기능을 사용할 수 있다. 이 대화창에는 구현하지 않은 추상 멤버만 표시된다(그림 8-6의 Circle 클래스 예제를 보라).

그림 8-6: Implement Members 대화창



</참고>

다른 방법으로는 클래스 이름이나 키워드에서 [Alt]+[Enter]를 누르면 나오는 퀵픽스를 사용하는 방법이 있다(그림 8-7에서 붉은색으로 강조된 부분). 퀵픽스를 사용하면 추상 프로퍼티를 구현하는 메뉴가 생성자 파라미터를 추가하거나, 현재 클래스를 추상 클래스로 표시하는 등의 여러 가지(문맥에 따라 적절한) 해결 방법과 함께 표시된다(예제의 Rectangle 클래스 참고).

### 인터페이스

코틀린 인터페이스 개념은 자바의 인터페이스와 상당히 비슷하다. 특히 자바 8에 디폴트 메서드가 도입된 이후로는 코틀린 인터페이스와 자바 인터페이스가 더 많이 비슷해졌다. 근본적으로 인터페이스는 (추상 또는 비-추상) 메서드나 프로퍼티를 포함하지만 자체적인 인스턴스 상태나 생성자를 만들 수는 없는 타입이다.

클래스와 달리 인터페이스 정의는 interface라는 키워드로 시작한다.

<코드>

interface Vehicle {  
 val currentSpeed: Int  
 fun move()  
 fun stop()  
}

</코드>

인터페이스 멤버는 디폴트가 추상 멤버다. 따라서 (앞의 코드처럼) 구현을 제공하지 않으면 abstract 변경자가 자동으로 붙은 것처럼 간주된다. 명시적으로 abstract를 붙일 수도 있지만 불필요하다.

인터페이스는 클래스나 다른 인터페이스의 상위 타입이 될 수 있다. 비 추상 클래스가 인터페이스를 상속할 때는 모든 추상 멤버에 대한 구현을 제공해야 한다(그리고 비-추상 멤버를 오버라이드할 수 있다). 마찬가지로 클래스가 인터페이스를 상속한 클래스에 있는 인터페이스 멤버를 상속해 구현할 때도 override 키워드를 추가해야 한다.

<코드>

interface FlyingVehicle : Vehicle {  
 val currentHeight: Int  
 fun takeOff()  
 fun land()  
}  
  
class Car : Vehicle {  
 override var currentSpeed = 0  
 private set  
   
 override fun move() {  
 println("Riding...")  
 currentSpeed = 50  
 }  
   
 override fun stop() {  
 println("Stopped")  
 currentSpeed = 0  
 }  
}  
  
class Aircraft :FlyingVehicle {  
 override var currentSpeed = 0  
 private set  
   
 override var currentHeight = 0  
 private set  
   
 override fun move() {  
 println("Taxiing...")  
 currentSpeed = 50  
 }  
   
 override fun stop() {  
 println("Stopped")  
 currentSpeed = 0  
 }  
   
 override fun takeOff() {  
 println("Taking off...")  
 currentSpeed = 500  
 currentHeight = 5000  
 }  
   
 override fun land() {  
 println("Landed")  
 currentSpeed = 50  
 currentHeight = 0  
 }  
}

</코드>

세 가지 타입 정의에서 모두 상위 타입 이름 뒤에 괄호를 붙이지 않았다는 점에 유의하라. 클래스와 달리 인터페이스에는 생성자가 없어서 하위 클래스를 초기화할 때 호출해야 할 코드가 없기 때문이라고 이유를 설명할 수 있다.

**코틀린과 자바의 차이**: 인터페이스를 상속할 때는 implements 키워드를 쓰고 클래스를 상속할 때는 extends를 써야만 하는 자바와 달리, 코틀린에서는 모든 상속(클래스가 클래스를 상속, 인터페이스가 인터페이스를 상속, 클래스가 인터페이스를 상속)을 똑같은 기호(:)를 사용해 표시한다.

자바와 마찬가지로 코틀린 인터페이스가 다른 클래스를 상속할 수는 없다. 단, 모든 코틀린 클래스와 인터페이스가 암시적으로 상속하는 것으로 간주되는 Any 클래스는 예외다.

인터페이스 안의 함수와 프로퍼티에 구현을 추가할 수도 있다.

<코드>

interface Vehicle {  
 val currentSpeed: Int  
   
 val isMovingget() = currentSpeed != 0  
   
 fun move()  
   
 fun stop()  
   
 fun report() {  
 println(if (isMoving) “Moving at $currentSpeed” else “Still”)  
 }  
}

</코드>

이런 구현은 암시적으로 열려있는 것으로 간주된다. 따라서 상속하는 클래스에서 이런 멤버를 오버라이드할 수 있다. 인터페이스 멤버를 final로 정의하면 컴파일 오류가 발생한다.

<코드>

interface Vehicle {  
 final fun move() {} // Error  
}

</코드>

하지만 확장 함수나 프로퍼티를 사용하면 final 멤버를 대신할 수 있다.

<코드>

fun Vehicle.relativeSpeed(vehicle: Vehicle) =  
currentSpeed - vehicle.currentSpeed

</코드>

인터페이스를 인터페이스로 상속할 때도 메서드를 오버라이드할 수 있다.

<코드>

interface Vehicle {  
 fun move() {  
 println("I'm moving")  
 }  
}  
  
interface Car : Vehicle {  
 override fun move() {  
 println("I'm riding")  
 }  
}

</코드>

<참고> IDE 팁

앞 절에서 설명한 Override Members와 Implement Members 액션을 인터페이스 본문에서도 사용할 수 있다.

</참고>

인터페이스 내부에 상태를 정의할 수 없으므로, 인터페이스 안에는 뒷받침하는 필드가 들어있는 프로퍼티를 정의할 수 없다. 초기화 코드나 위임이 붙은 프로퍼티는 금지된다.

<코드>

interface Vehicle {  
val currentSpeed = 0 // Error  
val maxSpeed by lazy { 100 } // Error  
}

</코드>

인터페이스는 암묵적으로 추상 타입이다. 하지만 추상 클래스와 달리 인터페이스에 대한 생성자는 금지되어 있다.

<코드>

interface Person(val name: String) // Error  
  
interface Vehicle {  
 constructor(name: String) // Error  
}

</코드>

자바와 마찬가지로 코틀린 인터페이스도 다중 상속을 지원한다. 예제를 하나 살펴보자.

<코드>

interface Car {  
 fun ride()  
}  
  
interface Aircraft {  
 fun fly()  
}  
  
interface Ship {  
 fun sail()  
}  
  
interface FlyingCar : Car, Aircraft  
  
class Transformer :FlyingCar, Ship {  
 override fun ride() {  
 println("I'm riding")  
 }  
 override fun fly() {  
 println("I'm flying")  
 }  
 override fun sail() {  
 println("I'm sailing")  
 }  
}

</코드>

FlyingCar 인터페이스와 Transformer 클래스는 모두 둘 이상의 인터페이스를 상속하며, 따라서 이들의 멤버를 모두 상속받는다. 비-추상 Transformer 클래스는 상속받은 멤버를 모두 구현해야 한다.

한 타입이 동일한 시그니처를 가지는 멤버가 들어있는 다른 인터페이스를 둘 이상 상속할 때 흥미로운 문제가 생긴다. 이 경우 이런 멤버들이 한 멤버로 합쳐지고 하위타입은 이를 상속하는 것 같은 효과가 일어난다. 다음 예에서 Car와 Ship이 Any를 제외한 공통 상위 타입이 없다고 하자.

<코드>

interface Car {  
 fun move()  
}  
  
interface Ship {  
 fun move()  
}  
  
class Amphibia : Car, Ship {  
 override fun move() {  
 println("I'm moving")  
 }  
}

</코드>

위 코드에서 move() 메서드는 추상메서드다. 따라서 비-추상 클래스인 Amphibia에서는 이 메서드를 구현해야 한다. 하지만 상위 타입의 메서드 중에 구현이 존재하는 경우라 할지라도, 컴파일러는 모호성을 해결하기 위해 반드시 메서드를 명시적으로 구현하도록 강제한다.

<코드>

interface Car {  
 fun move(){  
 println("I'm riding")  
 }  
}  
  
interface Ship {  
 fun move()  
}  
  
class Amphibia : Car, Ship {  
 override fun move() {  
 super.move() // Car에서 상속받은 메서드를 호출  
 }  
}  
  
fun main() {  
 Amphibia().move() // I'm riding  
}

</코드>

이런 식으로 합쳐지는 멤버에 대한 구현이 둘 이상의 상위 타입에 존재하는 경우, super 호출 자체가 모호해진다. 이럴 때는 super를 상위 타입으로 한정시킨 키워드를 사용해야 한다.

<코드>

interface Car {  
 fun move(){  
 println("I'm riding")  
 }  
}  
  
interface Ship {  
 fun move() {  
 println("I'm sailing")  
 }  
}  
  
class Amphibia : Car, Ship {  
 override fun move() {  
 super<Car>.move() // Car에서 상속받은 메서드를 호출  
 super<Ship>.move() // Ship에서 상속받은 메서드를 호출  
 }  
}  
  
fun main() {  
 /\*  
 I'm riding  
 I'm sailing  
 \*/  
 Amphibia().move()   
}

</코드>

**자바와 코틀린의 차이**: 자바8도 같은 목적으로 Ship.super.move()처럼 한정된 super를 사용한다.

코틀린 버전 1.1부터는 컴파일러가 자바 8의 디폴트 메서드 형태로 비-추상 인터페이스 멤버 코드를 생성할 수 있다. **12장 자바 상호 운용성**에서 이런 상호 운용 관련 문제를 살펴본다.

인터페이스에 생성자나 상태를 사용할 수 없다는 제약은 다중 상속에 대한 지원을 위해서다. 주 목적은 악명 높은 다이아몬드 상속(diamon inheritance) 문제를 방지하는 것이다. 다음 코드를 보자.

<코드>

interface Vehicle {  
 val currentSpeed: Int  
}  
  
interface Car : Vehicle  
  
interface Ship : Vehicle  
  
class Amphibia : Car, Ship {  
 override var currentSpeed = 0  
 private set  
}

</코드>

인스턴스의 상태를 허용하면 Vehicle 인터페이스에서 상태 변수로 currentSpeed를 정의할 수 있다. 그 결과 Amphibia 클래스에는 currentSpeed의 복사본이 두 가지 존재하게 된다. 하나는 Ship에서, 다른 하나는 Car에서 상속한 것인데 두 가지 모두 Vehicle에서 물려받은 것이다. 코틀린 설계는 인터페이스에서 상태를 허용하지 않음으로써 이런 문제를 회피한다. 프로그램 상태를 초기화하는 순서를 예측할 수 있으려면 생성자 사용 금지가 중요하다. 인터페이스에 생성자를 허용하면 초기화 순서 규칙(**하위 클래스 초기화** 절에서 살펴봤다)이 다중 상속도 제대로 처리할 수 있게 확장해야 한다[[3]](#footnote-3). 이런 규칙을 따라서 초기화 순서를 (프로그래머가) 알아내기는 상당히 힘들다. 특히 상위 타입 관계 그래프에서 같은 인터페이스가 여러 번 나타난 경우에는 더욱 더 힘들어진다(앞에서 본 예제의 Vehicle을 보라).

## 봉인된 클래스

때로는 우리가 프로그램에서 표현하고 싶은 개념이 몇 가지 정해진 변종의 집합으로 구성될 때가 있다. **6장 특별한 클래스 사용하기**에서는 똑같은 타입을 공유하는 미리 정의된 상수 집합을 표현하는 enum 클래스를 소개했다. 예를 들어 이넘 클래스를 사용해 어떤 계산이 성공인지 실패인지를 표현할 수 있다.

<코드>

enum class Result {  
 SUCCESS, ERROR  
}  
  
fun runComputation(): Result {  
 try {  
 val a = readLine()?.toInt() ?: return Result.ERROR  
 val b = readLine()?.toInt() ?: return Result.ERROR  
   
 println("Sum: ${a + b}")  
   
 return Result.SUCCESS  
 } catch (e: NumberFormatException) {  
 return Result.ERROR  
 }  
}  
  
fun main() {  
 val message = when (runComputation()) {  
 Result.SUCCESS -> "Completed successfully"  
 Result.ERROR -> "Error!"  
 }  
   
 println(message)  
}

</코드>

하지만 어떤 경우에는 각 종류별로 애트리뷰트가 다를 수가 있다. 예를 들어 성공인 경우에는 생성된 결과가 들어있고, 실패인 경우에는 실패 이유에 대한 정보가 들어있을 수 있다. 이는 이 장에서 이미 설명했던 예제와 비슷하다. 이와 같은 개념을 클래스 계층을 활용해 모델링할 수 있다. 클래스 계층의 루트에 있는 추상 클래스는 전체 개념을 표현하며 하위 클래스들은 특정 변종을 표현한다. 앞에서 본 예제를 다듬으면서 Success와 Error 경우를 표현하는 클래스를 추가해보자.

<코드>

abstract class Result {  
 class Success(val value: Any) : Result() {  
 fun showResult() {  
 println(value)  
 }  
 }  
   
 class Error(val message: String) : Result() {  
 fun throwException() {  
 throw Exception(message)  
 }  
 }  
}  
  
fun runComputation(): Result {  
 try {  
 val a = readLine()?.toInt()  
 ?: return Result.Error("Missing first argument")  
 val b = readLine()?.toInt()  
 ?: return Result.Error("Missing second argument")  
   
 return Result.Success(a + b)  
 } catch (e: NumberFormatException) {  
 return Result.Error(e.message ?: "Invalid input")  
 }  
}  
  
fun main() {  
 val message = when (val result = runComputation()) {  
 is Result.Success -> "Completed successfully: ${result.value}"  
 is Result.Error -> "Error: ${result.message}"  
 else -> return  
 }  
 println(message)  
}

</코드>

이 구현에는 흠이 있다. 이 구현은 Result의 변종을 Success나 Error로 제한시켜주지 못한다. 다른 클라이언트 코드가 다음과 같이 새로운 하위 클래스를 추가해도 막을 방법이 없다.

<코드>

class MyStatus: Result()

</코드>

그리고 이런 서브 클래싱이 가능하다는 점이 when 식에서 else가 필요한 이유이기도 하다. 컴파일러는 result 변수의 값이 Success나 Error 인스턴스만 담는다고 알 수 없기 때문에 else 절을 추가하라고 강제한다.

코틀린에서는 봉인된 클래스(sealed class)를 통해 이런 문제를 극복할 수 있다. 클래스 정의에 sealed 변경자를 추가하자.

<코드>

sealed class Result {  
 class Success(val value: Any) : Result() {...}  
 class Error(val message: String) : Result() {...}  
}

</코드>

어떤 클래스를 sealed로 지정하면 이 클래스를 상속하는 클래스는 내포된 클래스나 객체로 정의되거나, 같은 파일 안에서 최상위 클래스로 정의돼야만 한다(코틀린 1.1부터 같은 파일 안의 최상위 클래스를 지원하기 시작했다). 이런 영역 밖에서 봉인된 클래스는 final 클래스와 동일한 효과를 지니며 아무도 봉인된 클래스를 상속할 수 없다.

봉인된 클래스는 직접 인스턴스를 만들 수 없기 때문에 추상 클래스이기도 하다는 점에 유의하라. 봉인된 클래스의 인스턴스를 만들 때는 봉인된 클래스의 하위 클래스 중 하나를 선택해 만들어야 한다.

<코드>

val result = Result() // Error: can’t instantiate an abstract class

</코드>

실제로 봉인된 클래스의 생성자는 디폴트로 비공개(private)이고, 가시성을 다른 값으로 변경하면 컴파일 시점에 오류로 간주된다.

이넘과 마찬가지로 봉인된 클래스도 불필요한 else를 쓰지 않아도 되는, 빠진 부분이 없는 when 형태를 지원한다.

<코드>

val message = when (val result = runComputation()) {  
 is Result.Success -> "Completed successfully: ${result.value}"  
 is Result.Error -> "Error: ${result.message}"  
}

</코드>

상속 제한은 봉인된 클래스를 직접 상속한 클래스에 대해서만 성립한다. 하위 클래스가 final이 아니라면 이를 상속한 하위 클래스가 있을 수 있다.

<코드>

// Result.kt  
sealed class Result {  
 class Success(val value: Any) : Result()  
 open class Error(val message: String) : Result()  
}  
  
// util.kt  
class FatalError(message: String): Result.Error(message)

</코드>

코틀린 1.1부터 봉인된 클래스가 다른 클래스를 상속할 수도 있다. 이로 인해 봉인된 클래스의 하위 클래스가 다시 봉인된 클래스가 될 수도 있다.

<코드>

sealed class Result  
  
class Success(val value: Any) : Result()  
  
sealed class Error : Result() {  
 abstract val message: String  
}  
  
class ErrorWithException(val exception: Exception): Error() {  
 override val message: String get() = exception.message ?: ""  
}  
  
class ErrorWithMessage(override val message: String): Error()

</코드>

코틀린 1.1부터 허용된 데이터 클래스의 상속 기능으로 인해, 데이터 클래스가 봉인된 클래스 계층에 속할 수도 있다. 이를 통해 데이터 클래스와 봉인된 클래스의 장점을 함께 취할 수 있다. 예를 들어 간단한 연산식의 구문 트리(syntactic tree)를 표현하는 클래스를 생각해보자.

<코드>

sealed class Expr  
  
data class Const(val num: Int): Expr()  
data class Neg(val operand: Expr): Expr()  
data class Plus(val op1: Expr, val op2: Expr): Expr()  
data class Mul(val op1: Expr, val op2: Expr): Expr()  
  
fun Expr.eval(): Int = when (this) {  
 is Const -> num  
 is Neg -> -operand.eval()  
 is Plus -> op1.eval() + op2.eval()  
 is Mul -> op1.eval() \* op2.eval()  
}  
  
fun main() {  
 // (1 + 2) \* 3  
 val expr = Mul(Plus(Const(1), Const(2)), Const(3))  
   
 // Mul(op1=Plus(op1=Const(num=1), op2=Const(num=2)), op2=Const(num=3))  
 println(expr)  
 println(expr.eval()) // 9  
   
 // 2 \* 3  
 val expr2 = expr.copy(op1 = Const(2))  
   
 // Mul(op1=Const(num=2), op2=Const(num=3))  
 println(expr2)  
 println(expr2.eval()) // 6  
}

</코드>

sealed 변경자를 인터페이스에 적용할 수 없다는 점에 유의하라. 코틀린에서 클래스를 다중 상속할 수는 없기 때문에, 이 말은 봉인된 클래스 계층을 이루는 하위 클래스가 봉인된 클래스 외의 다른 클래스를 상속할 수는 없다는 뜻이다.

봉인된 클래스를 객체로 구현할 수도 있다. Result의 결과를 세분화해서 실제 만들어진 값이 없이 성공한 경우를 표현하고 싶다고 하자.

<코드>

sealed class Result {  
 object Completed : Result()  
 class ValueProduced(val value: Any) : Result()  
 class Error(val message: String) : Result()  
}

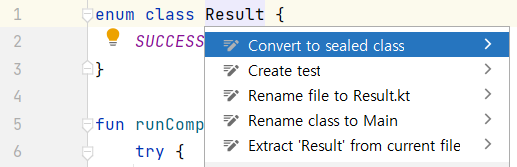
</코드>

모든 직접적인 상속자가 객체인 경우 봉인된 클래스는 결과적으로 이넘 클래스처럼 동작한다.

<참고> IDE 팁

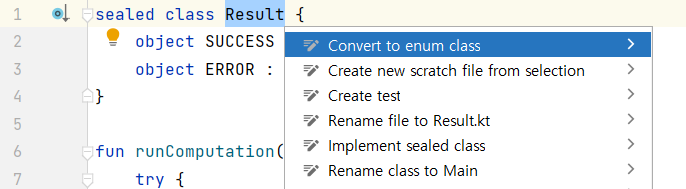
이넘 클래스를 봉인된 클래스로 리팩터링하고 싶다면 [Alt]+[Enter] 메뉴에 있는 봉인된 클래스로 변환하는 액션이 좋은 출발점이 될 수 있다(그림 8-7). 이렇게 변환한 결과에서 이넘의 각 상수는 추상 봉인 클래스를 구현하는 싱글턴 객체들로 변환된다.

그림 8-7: 이넘 클래스를 봉인된 클래스 계층구조로 바꾸기



인텔리J는 역방향 변환도 제공한다. 봉인된 클래스를 직접 상속한 모든 상속자가 객체라면 이들의 구현을 이넘 상수로 변경함으로써 봉인된 클래스를 이넘 클래스로 만들 수 있다(그림 8-8).

그림 8-8: 봉인된 클래스를 이넘으로 바꾸기



</참고>

## 위임

앞 절에서 코틀린 클래스가 기본적으로 final이라는 사실을 배웠다. 이렇게 정한 목적은 상속 가능한 클래스를 좀 더 심사숙고한 후 설계해서 하위 클래스를 만들 수 없는 클래스를 실수로 상속하는 일을 없애기 위함이다. 이런 성질은 앞에서 살펴본 깨지기 쉬운 기반 클래스 문제를 완화시켜준다.

하지만 기존 클래스를 확장하거나 변경해야 하는데, 이 클래스를 상속할 수는 없다면 어떻게 해야 할까? 이런 경우 기존 클래스를 재사용하는 잘 알려진 패턴인 위임 패턴을 사용할 수 있다. 어떤 인터페이스의 구현을 만들고 싶다면 이미 있는 구현의 인스턴스를 가져와서 우리가 만드는 클래스로 감싸고, 필요할 때 인터페이스 메서드 구현을 이 기존 구현 인스턴스에게 위임할 수 있다.

예를 하나 살펴보자. 다음과 같은 타입들이 있다고 하자.

<코드>

interface PersonData {  
 val name: String  
 val age: Int  
}  
  
open class Person(  
 override val name: String,  
 override val age: Int  
): PersonData  
  
data class Book(val title: String, val author: PersonData) {  
 override fun toString() = "'$title' by ${author.name}"  
}  
  
fun main() {  
 val valWatts = Person("Val Watts", 30)  
 val introKotlin = Book("Introduction to Kotlin", valWatts)  
   
 println(introKotlin) // ‘Introduction to Kotlin’ by Val Watts  
}

</코드>

이제 작가들이 필명을 사용할 수 있게 허용하자. 이를 통해 작가는 다른 사람인 것처럼 가장할 수 있다.

<코드>

class Alias(  
 private valrealIdentity: PersonData,  
 private valnewIdentity: PersonData  
) :PersonData {  
 override val name: String  
 get() = newIdentity.name  
   
 override val age: Int  
 get() = newIdentity.age  
}

</코드>

이제 이 클래스를 사용해 어떤 사람의 가명을 만들 수 있다.

<코드>

fun main() {  
 val valWatts = Person("Val Watts", 30)  
 val johnDoe = Alias(valWatts, Person("John Doe", 25))  
 val introJava = Book("Introduction to Java",johnDoe)  
   
 println(introJava) // ‘Introduction to Java’ by John Doe  
}

</코드>

이 접근 방법의 문제는 필요한 메서드나 프로퍼티를 다른 객체에게 위임하기 위해 작성해야 하는 준비 코드가 너무 많다는 것이다. 다행히도 코틀린은 이런 위임을 처리하는 기능을 내장하고 있다. 여러분이 해야 할 일은 상위 인터페이스 이름 바로 뒤에 by 키워드를 붙이고 그 다음에 위임할 인스턴스를 쓰는 것뿐이다.

<코드>

class Alias(  
 private val realIdentity: PersonData,  
 private val newIdentity: PersonData  
) :PersonData by newIdentity

</코드>

이제 Alias가 PersonData 인터페이스에서 상속한 모든 멤버는 newIdentity 인스턴스에 있는 이름과 시그니처가 같은 메서드를 통해 구현된다. 구현을 바꾸고 싶으면 직접 멤버를 오버라이드할 수도 있다.

<코드>

class Alias(  
 private val realIdentity: PersonData,  
 private val newIdentity: PersonData  
) :PersonData by newIdentity {  
 override val age: Int get() = realIdentity.age  
}  
  
fun main() {  
 val valWatts = Person("Val Watts", 30)  
 val johnDoe = Alias(valWatts, Person("John Doe", 25))  
 println(johnDoe.age) // 30  
}

</코드>

일반적으로 클래스 초기화 시 사용할 수 있는 대부분의 일을 위임으로 처리할 수 있다. 컴파일러는 위임된 값을 저장하는 필드를 자동으로 만들어준다. 예를 들어 앞의 정의에서 newIdentity 파라미터에 붙은 val을 없애고 간단한 생성자 파라미터로 바꿔도 된다.

<코드>

class Alias(  
 private val realIdentity: PersonData,  
 newIdentity: PersonData  
) :PersonData by newIdentity

</코드>

하지만 클래스 본문에 정의된 프로퍼티를 클래스 위임에 사용할 수는 없다.

<코드>

class Alias(  
 private valrealIdentity: PersonData  
) :PersonData by newIdentity { // Error: newIdentity를 찾을 수 없음  
 val newIdentity = Person("John Doe", 30)  
}

</코드>

위임과 객체 식을 조합하면 원래 객체와 약간 다른 구현을 만들 때 유용할 수 있다.

<코드>

fun PersonData.aliased(newIdentity: PersonData) =  
 object :PersonData by newIdentity {  
 override val age: Int get() = this@aliased.age  
 }  
  
fun main() {  
 val valWatts = Person("Val Watts", 30)  
 val johnDoe = valWatts.aliased(Person("John Doe", 25))  
   
 println("${johnDoe.name}, ${johnDoe.age}") // John Doe, 30  
}

</코드>

클래스는 인터페이스 멤버를 구현할 때만 위임을 쓸 수 있다. 예를 들어 앞의 코드에서 Person이 클래스라면 오류가 발생한다.

<코드>

class Alias(  
 private val realIdentity: PersonData,  
 private val newIdentity: PersonData  
) : Person by newIdentity // Error: 인터페이스만 위임할 수 있음

</코드>

여기서 핵심은 다음과 같다.

클래스 위임을 사용하면 번거로운 준비 코드를 사용하지 않고도 객체 합성(composition)과 상속의 이점을 살릴 수 있다. 따라서 코틀린의 위임은 "상속보다는 합성(composition over inheritance)"이라는 잘 알려진 객체 지향 설계 원칙을 사용하도록 장려한다.

## 결론

이 장에서는 코틀린 타입 시스템이 제공하는 강력한 상속 메커니즘을 살펴봤다. 하위 클래스를 어떻게 설계하는지 살펴보고, 클래스 초기화가 클래스 계층 구조 전체와 어떻게 들어맞는지 봤고, 기반 클래스의 동작을 하위 클래스에서 변경하기 위해 멤버를 오버라이드하는 방법도 배웠다. 그리고 추상적인 개념을 표현하기 위한 도구인 추상 클래스나 인터페이스를 다루는 방법도 살펴봤다. 마지막으로 상속과 관련한 두 가지 유용한 패턴을 구현해주는 봉인된 클래스와 위임에 대해 살펴봤다.

다음 장에서는 제네릭스에 초점을 맞춘다. 제네릭스는 코틀린 타입 시스템의 특별한 기능으로 여러분의 클래스 선언을 미지의 타입으로 파라미터화해서 나중에 사용할 때 구체적인 타입을 지정한 클래스로 활용할 수 있게 해주는 기능이다.

## 질문

1. 코틀린에서 하위 클래스를 어떻게 정의하는가? 클래스를 상속할 수 있게 만들려면 어떤 조건을 만족시켜야 하는가?
2. 자바와 코틀린 클래스 상속의 차이점에 대해 설명하라.
3. 어떤 클래스가 다른 클래스를 상속할 때, 이 클래스의 인스턴스가 초기화되는 과정을 설명하라. 상위 클래스 초기화를 자바에서는 어떻게 강제하는가? 코틀린의 상위 클래스 초기화 방식과 자바의 방식을 비교하라.
4. is/as/as? 연산자의 목적에 대해 설명하라. 자바의 타입 체크와 타입 캐스트와 이들을 비교하라.
5. Any 클래스에 정의된 공통 메서드 이름을 나열하라. 각 메서드를 구현할 때 지켜야 할 기본 지침을 설명하라.
6. 추상 클래스와 추상 클래스 멤버란 무엇인가? 추상 클래스나 추상 클래스 멤버를 처리하는 규칙은 무엇인가?
7. 추상 클래스와 인터페이스는 어떤 차이가 있는가? 코틀린과 자바 인터페이스를 비교하라.
8. 인터페이스 상속을 어떤 식으로 하는지 설명하라. 클래스 멤버를 오버라이딩할 때와 인터페이스 멤버를 오버라이딩할 때 어떤 차이가 있는지 설명하라.
9. 봉인된 클래스 계층은 무엇인가? 자바를 사용한다면 봉인된 클래스 계층을 어떤 식으로 상속할 수 있겠는가?
10. 코틀린에서 클래스 위임이 어떻게 작동하는지 설명하라.

1. 옮긴이 - 제공하는 연산의 관점에서 본다면 하위 클래스의 구현이 상위 클래스의 정의를 포함하는 더 많은 기능을 제공하므로 이상하지 않다고 생각할 수도 있지만, 불변성과 가변성을 놓고 생각해보면 상위 클래스에서는 불변 값으로 가정하고 있는 프로퍼티를 하위 클래스에서 마음대로 바꿔 쓰게 하므로 문제가 발생할 수 있다. 꼭 필요한 경우가 아니라면 이런 식의 오버라이딩을 피해야 한다. [↑](#footnote-ref-1)
2. 옮긴이 - 중간의 other as Entity는 other를 Entity 타입으로 캐스팅하고 문제가 생기면 오류를 발생시킨다. 따라서 컴파일러는 이 구문 뒤부터는 안전하게 other를 Entity 타입으로 가정할 수 있다. 한편, 혹시라도 여기서 캐스팅 예외가 발생하는 것은 아닌가 걱정하는 독자도 있을 것이다. 하지만 이 as 구문 바로 위에서 이미 this와 other의 자바 클래스가 같음을 확인했기 때문에 이 부분에서 ClassCastException이 발생할 가능성은 없다. 따라서 이런 식의 구문이 if문이나 안전한 as?문보다 더 짧고 이해하기도 쉽다. [↑](#footnote-ref-2)
3. 옮긴이 - 이런 식으로 인터페이스까지 포함한 상속 그래프의 생성자 초기화 순서를 정의해 사용하는 언어도 있다. 대표적으로 스칼라에는 인터페이스에 해당하는 트레이트(trait)가 있는데, 클래스 선언 시 확장한 클래스와 트레이트들을 미리 정해진 규칙을 따라 줄을 세워주는 선형화(linearization)를 통해 생성자 호출 순서를 유일하게 결정해준다. [↑](#footnote-ref-3)