Kotlin Delegation and Const

20191629 이주헌

코드 재사용성(code reusability)은 객체지향 프로그래밍 언어의 가장 큰 특징 중 하나이다. Kotlin에서는 다른 객체지향 프로그래밍 언어가 지원하는 상속(inheritance)과 인터페이스(interface) 이외에도 위임(delegate)이라는 코드 재사용 패턴을 지원한다. 또한, Kotlin에서는 불변(immutable) 변수와 상수(constant)도 언어의 문법 단에서 구분하여 지원한다. 이 리포트에서는 위임자 패턴(delegator pattern)과 불변 변수와 상수의 차이점에 대해 분석한다.

위임자 패턴(delegator pattern)

Kotlin에서는 어떤 메서드나 프로퍼티의 구현을 다른 인터페이스나 메서드에 위임할 수 있다. 여기서 “위임한다”는 말은 클래스의 멤버의 처리 방식을 다른 클래스에 위임하여 처리한다는 의미이다. 아래는 어떤 클래스에서 메서드를 위임받는 클래스의 예시이다.

```
interface LicenseProvider {
    abstract fun getLicense(): String;
}

class WTFPLLicenseProvider: LicenseProvider {
    override fun getLicense(): String {
        return "DO WHAT THE F**K WHAT YOU WANT TO PUBLIC LICENSE"
    }
}

class Program(license: LicenseProvider) : LicenseProvider by license
```

위 예시에서 Program 클래스는 getLicense() 메서드를 가지게 되는데, 이 메서드는 생성자의 인자로 받은 license 변수에서 제공하는 메서드이다. 즉, 여기서 Program 클래스는 아래와 같이 정의된 것과 같다.

```
class Program(private val license: LicenseProvider) : LicenseProvider {
    override fun getLicense(): String = license.getLicense()
}
```

또, Kotlin에서는 클래스 뿐만 아니라 프로퍼티도 위임받을 수 있다. 프로퍼티를 위임하기 위해서는 클래스가 getValue 메서드와 setValue 메서드를 구현하고 있어야 한다. 예를 들어, 아래와 같은 경우를 생각할 수 있다.

```
interface NotificationProvider {
    abstract operator fun getValue(thisRef: Any?, property:
KProperty<*>): String
    abstract operator fun setValue(thisRef: Any?, property:
KProperty<*>, value: String)
}

class StdioNotificationProvider : NotificationProvider {
    var stuff: String = ""
    override operator fun getValue(thisRef: Any?, property:
KProperty<*>): String  {
        println("${property.name} accessed!")
        return this.stuff
    }
    override operator fun setValue(thisRef: Any?, property:
KProperty<*>, value: String) {
        println("${property.name} changed to $value!")
        this.stuff = value
    }
}

class Notifier(provider: NotificationProvider) : NotificationProvider by
provider {
    var quote: String by provider
}
```

이 패턴은 Kotlin에서 게으른 계산(lazy evaluation)을 구현할 때 자주 사용된다. Kotlin 표준 라이브러리에는 lazy 함수가 제공되는데, 이 함수는 getValue가 처음 호출되었을 때 비로소 초깃값을 계산하는 인터페이스인 Lazy<T>의 인스턴스를 반환한다. 따라서, 아래와 같이 사용할 수 있다.

```
class LazyClass {
    val lazyValue: String by lazy {
        println("Now I'm waking up!")
        "Still lazy"
    }
}
```

다른 언어에서 Kotlin의 위임자 패턴과 비슷한 개념을 찾는다면, Ruby의 module inclusion이나 SCSS예 mixin을 예로 들 수 있다. 두 개념 모두 클래스나 모듈에 선언된 메서드를 다른 클래스에 삽입하는 것이 가능하지만, Kotlin은 여기서 더 나아가 클래스의 생성자에 따라 동작 방식을 바꿀 수 있도록 설계하였다.

상수 선언(declaration of `const val`)

Kotlin에서는 const val(이하 "상수")을 i) 최고 수준에서 정의하거나 ii) 싱글턴 객체에서밖에 정의할 수 없다. 이 차이는 불변 변수인 val과 컴파일 타임 상수인 const val의 차이에서 기인한다.

불변 변수 val은 JVM IR 수준에서 getter만 정의되는 변수이다. 즉, 이 변수는 JVM 수준에서 "변수"로 취급되며, 이 변수는 JVM 레퍼런스 타입을 저장할 수 있다.

그러나 const val으로 선언된 상수는 Kotlin 컴파일 시에 값이 결정되어야 한다. 즉, const val으로 선언된 변수는 컴파일 된 이후 JVM 바이트코드의 “immediate” 값으로 취급된다. 따라서, 상수는 항상 JVM 바이트코드에서 사용 가능한 "원시 값(primitive value)”여야 한다.

Kotlin 컴파일러가 컴파일 타임에 어떤 값이 정해져 있는지 알기 위해서는 컴파일 타임에 인스턴스가 만들어져야 한다. 일반적인 클래스는 클래스 인스턴스가 만들어지는 시점에서 멤버의 값이 결정되기 때문에 Kotlin 컴파일러가 원시 값의 내용을 결정지을 수 없다. 반면 singleton 객체나 최고 수준에서 선언된 상수는 컴파일 타임에 값을 알 수 있다. 전자는 해당 파일명과 같은 이름을 가지는 클래스에 public final static으로 선언되는 변수이므로 컴파일 타임에 값을 알 수 있고, 후자는 컴파일 타임에 인스턴스를 만드는 코드가 삽입되므로 해당 상수의 값이 실제로 상수라는 것을 타입 수준에서 검증 가능하기 때문이다.

이 관계를 가장 잘 설명하는 것이 있다면 C++의 const와 constexpr가 있을 것이다. C++에서 const는 값이 변할 수 없다는 타입에 불과하지만, constexpr는 컴파일 타임에 값이 결정되니 컴파일러가 마음대로 최적화를 해도 괜찮다는 표시이다. 마찬가지로 val은 setter가 정의되지 않는 타입 수준에서의 상수이지만, const val은 JVM Bytecode Immediate로 삽입될 수 있는 값이다.