2023년 봄학기 캡스톤디자인I 4주차 과제

1. Kotlin의 객체지향 특성인 code reuse를 위하여 inheritance, interface 뿐 아니라 delegation도 사용할 수 있습니다. Kotlin의 interface delegation, delegated property 를 조사하여 리포트 및 각각의 예제코드를 작성하여 제출하세요.

2. Kotlin에서 val과 const val의 차이와 top-level 및 object declaration(*Singleton*)에서만 const val 정의가 가능한 이유를 리포트로 작성하세요.

1. Delegation

Kotlin의 Interface Delegation

Kotlin은 코드의 재사용성을 높이기 위해 interface를 delegation 할 수 있는 기능을 제공한다. Interface delegation은 어떤 interface를 implementation 하는 class를 사용할 때, 현재 클래스에서 구현하려는 interface를 다른 클래스에게 위임하여 그 클래스에서 구현한 메소드를 사용하는 방법이다. 이를 일반화한 코드를 작성하면 다음과 같다.

```
interface A { ... }
class B : A { }
val b = B()

class C : A by b // A에서 정의하는 B의 모든 구현 메소드를 C에 delegate
```

즉, B에서 구현된 모든 A의 메소드를 C에서 참조할 수 있도록 하는 방식으로 구현되며, C의 모든 public member들을 B에 위임함으로써 C가 interface A를 구현할 수 있도록 한다.

Kotlin 언어 공식 문서(https://kotlinlang.org/docs/delegation.html)에 나온 예시 코드를 응용하면 다음과 같이 정리할 수 있다.

```kotlin
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class DelegatingBaseImpl(delegate : Base) : Base by delegate

fun main() {
    val b = BaseImpl(10)
    val delegator = DelegatingBaseImpl(b)
    delegator.print()
}
```

`BaseImpl` 클래스의 객체 `b` 는 `Base` interface를 구현하고 있다. 여기서 새로운 클래스인 `DelegatingBaseImpl` 의 constructor에 `b` 를 파라미터로 넘김으로써 `b` 의 모든 메소드를 `DelegatingBaseImpl` 클래스의 객체인 `delegator` 에 위임할 수 있다. 다시 말해, `DelegatingBaseImpl` 클래스에서는 `Base` interface를 구현하고 있지 않지만, 그 구현을 `BaseImpl` 클래스의 객체에 위임하는 것이다. 위의 코드 예시에서는 `DelegatingBaseImpl` 클래스가 `delegate` 객체에게 `print()` 함수 호출을 위임한다고 볼 수 있다.

`by` 키워드를 통해 constructor의 파라미터로 들어오는 `delegate` 객체가 `DelegatingBaseImpl` 클래스의 object에 내부적으로 저장되게 하며, 컴파일러가 `delegate` 객체가 구현한 `Base` interface의 메소드를 `DelegatingBaseImpl` 클래스의 object에 생성한다.

Kotlin의 Delegated Property

Kotlin의 Delegated Property

Delegated property는 현재 클래스의 property를 위임하려는 다른 객체에게 위임하여 그 property에 관한 처리를 매번 정의할 필요 없이 수행할 수 있어서 코드의 재사용성을 높이는 방법 중 하나이다. Kotlin 언어 공식 문서에 나온 예제를 응용하여 예시 코드를 작성하면 다음과 같다.

```
class Student(var name: String, idNumber: Integer) {
    var idNumber: Integer by IdNumberValidator(idNumber)
}

class IdNumberValidator(value: Integer) {
    var idNumber: Integer = value

    operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
        return idNumber
    }

    operator fun setValue(thisRef: Any?, property: KProperty<*>, newValue: String) {
        if (!isInt(newValue)) {
            throw IllegalArgumentException("Invalid student id number")
        }
        value = newValue
    }
}
```

위의 코드는 학생의 학번이 `Integer`가 아닌 잘못된 type의 값으로 들어오면 error로 처리하는 걸 구현한 예시이다. `Student` 클래스의 `idNumber` 속성을 `IdNumberValidator` 객체에게 위임하는데, `IdNumberValidator` 클래스에서 이 속성에 관한 get과 set 연산을 처리할 수 있도록 `getValue`와 `setValue` 함수를 구현하고 있다. 즉, `Student` 클래스의 `idNumber`에 관한 처리를 직접 그 클래스에서 구현하지 않고 미리 선언된 객체에 위임함으로써 유사한 속성을 다른 클래스에서도 사용할 때 코드를 직접 매번 구현하지 않고 이미 그 처리를 구현한 다른 코드를 가져다 쓸 수 있도록 한다.

2. `const val`

Kotlin의 `val`과 `const val`의 차이

기본적으로 `val`과 `const val`은 한 번 값을 초기화하면 그 값을 변경할 수 없는 type인 상수로 선언하는 키워드이지만, 그 값이 언제 boudning 되는지에 따라 차이가 있다.

`val` 로 선언한 상수는 그 값이 runtime에서 bounding 되는 반면에, `const val` 은 컴파일 시간에 bounding 되는 상수이다. 그래서 `const val` 로 선언한 상수는 runtime에서 생성된 속성에 관한 값으로 초기화할 수 없으며, 기본적인 primitive type 또는 문자열 등 기본적인 자료형이나 companion object만 할당할 수 있다. 여기서 companion object는 클래스 내부에서 `companion` 이라는 키워드와 함께 선언할 수 있는 object이며, 한 번 선언할 때 컴파일 시간에 이를 감싸는 클래스와 함께 하나의 instance로 생성된다. 그래서 그 클래스의 모든 instance에서 companion object의 멤버들을 공유할 수 있다.

Top-level 및 Object Declaration에서만 `const val` 정의가 가능한 이유

`const val` 로 선언한 상수는 기본적으로 컴파일 시간에 값이 결정된다고 했다. 이러한 구현이 가능하려면 stack 또는 heap에 메모리가 할당되는 함수 또는 객체의 scope 내에서 정의를 해야 하는 것이 아니라 그러한 클래스 또는 함수 안에서 선언되지 않고 가장 최상위 레벨에서 선언되어 프로그램의 메모리에서 code 또는 data 영역에 컴파일 시간에 적재될 수 있어야 한다. Object는 클래스와 다르며, 한 번 선언할 때 컴파일 시간에 하나의 default 객체만이 생성되어 singleton 객체로 볼 수 있다. 그래서 object에서는 컴파일 시간에 값이 bounding 되는 `const val` 을 정의할 수 있다.