4주차 과제

전공: 컴퓨터공학과 학년: 3학년 학번: 20211547 이름: 신지원

#1. Kotlin에서 code reuse를 위하여 inheritance, interface 뿐 아니라 delegation을 사용할 수 있습니다. Kotlin의 delegation 및 delegated property 를 조사. (각각의 예제코드 포함)

#1-1. Delegation

수업 시간에 공부했던 것처럼 Kotlin 에서는 inheritance, interface 등을 통해 객체를 상속받거나 클래스를 선언할 수 있다. 아래는 inheritance, interface 의 source code 다.

```kotlin
// inheritance
open class Shin {
    fun happy() {
        //
    }
}

class Jiwon: Shin() {
    //
}

fun main() {
    var shin = Jiwon()
    shin.happy()
}

// interface
interface Ena {
    fun smile()
    fun sad()
}

class Jiwon: Ena {
    override fun smile() {
        //
    }
}

fun main() {
    var ena = Jiwon()
    ena.smail()
}
```

하지만 이는 상속해야 할 프로퍼티의 수가 많아지면 많아질 수록 모두 처리해주기 어렵다는 한계를 갖는다. 따라서 Kotlin 에서는 이러한 상황을 위하여 delegation 를 제공한다. Delegation 은 상속을 하지 않아도 기존 기능을 그대로 사용하면서 새로운 기능을 추가할 수 있는 Delegate Design Pattern 이다.

Kotlin 에서는 delegation 을 by 와 함께 사용하도록 하는데 아래 source code 와 함께 보도록 하자.

```kotlin
interface Jiwon {
    val happy: Int,
    val sad: Int,
    val angry: Int,
    fun info()
}

class Today(private val jiwon: Jiwon) : Jiwon by jiwon {
    //paramater 로 들어오게 된 jiwon 을 by 라는 키워드를 통해 위임

        override val happy = 100
        override val sad = 0

}

fun main() {
    val today = Today(Jiwon())
    today.info()
}

```

위에서 제시한 source code 처럼 Jiwon 이라는 interface 를 생성한 뒤, Today 라는 class 를 통해서 jiwon 이라는 인자에 Jiwon 을 위임하고 있다. 이 때 main 에서 생성한 프로퍼티인 user 는 User class 안의 전체 내용을 알지는 못하지만 새로 추가된 내용에 대해 처리할 수 있게 된다.

Kotlin 에서는 무분별한 상속을 지양하여 final 접근 제어자를 사용하곤 한다. Delegation 을 사용하면 final 의 기능을 모두 사용할 수 있으면서 기능 확장까지 가능하기 때문에 용이하다.

#1-2. Delegated Property

Kotlin 에서는 delegation 의 이용에 있어 delegated property를 제공하고 있다.

1. lazy

lazy는 람다를 전달 받아 저장한 lazy<T> 인스턴스를 반환한다. 최초 getter 실행은 lazy()에 넘겨진 람다를 실행하고 결과를 기록하며, 이후 getter 실행은 기록된 값을 반환한다. lazy 는 속성의 초기화를 처음 접근하는 시점까지 지연시킨다. 이는 특히 값이 무거운 리소스를 로드하거나 계산하는 경우 유용하며, 속성에 접근하기 전까지는 해당 리소스가 로드되거나 계산되지 않는다. lazy 는 기본적으로 thread-safe 하기에 용이하며, 필요에 따라 스레드 안전 모드를 조정할 수 있다.

사용할 때는 아래 source code 와 같이 by 와 함께 사용한다.

```kotlin
class Jiwon {
    val myName : String by lazy { "ShinJiwon" }
}
```

2. Observable

Observable은 속성이 변경될 때 마다 알림을 받을 수 있게 해준다. 이를 통해 속성의 변경 사항을 관찰하고, 필요한 추가 로직을 실행할 수 있습니다. 즉, 프로퍼티의 데이터가 변화될 때마다 callback 을 받는 것이다.

```
var observableEx: Int by Delegates.observable(0) { property, old, new ->
    println("old: $old, new: $new")
}

fun main() {
    println(observableEx) // 0

    observableField = 1

    println(observableEx) // 1
}
```

위의 예시처럼 by Delegates.observable() 의 형태로 프로퍼티를 생성한 뒤, 프로퍼티의 값에 변화를 준다면 아래처럼 변화한 값을 return 한다.

3. map

map 위임은 특히 동적 속성(dynamic properties)을 사용하는 경우 유용하다. mutableMap을 이용하면 속성의 값을 수정 가능하도록 만들 수도 있다. 이를 이용하면 대표적으로 JSON 객체나 Map에서 키-값 쌍을 속성으로 직접 매핑할 수 있다. 이처럼 map 은 동적으로 변경되는 데이터 구조를 다룰 때 매우 효율적이다.

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int by map
}

fun main() {
    val user = User(mapOf(
        "name" to "Shin jiwon",
        "age" to 25
    ))

    println("Name: ${user.name}, Age: ${user.age}")
}
```

위 코드처럼 Map 을 사용하여 String, Any 로 묶어주었고 이를 통해 '키 - 값' 쌍을 속성으로 '이름-나이' 를 매핑한 것이다. 이와 같은 형태로 JSON 등에도 적용할 수 있다.