Kotlin 알못 벗어나기

1년동안 코틀린을 쓰면서 알게된 개인적인 팁들

코틀린 프로퍼티(Kotlin Property)

코틀린은 `var`, `val` 키워드를 제공한다.

보통 변하지 않는 프로퍼티는 `val`로, 변하는 프로퍼티는 `var`로 작성한다.

클래스 외부에 공개할 값은 대부분 `val`을 사용하겠지만, 어쩔 수 없이 가변 상태의 `var` 변수를 외부에 공개해야 할 때가 있다.

이런 경우 `프로퍼티 getter/setter`를 지정하여 클래스 외부에 해당 프로퍼티를 Read-Only로 공개할 수 있다.

```
class Car(
    distance: Int = 0
) {
    var distance = distance
        private set //클래스 내부에서만 프로퍼티 변경 가능

    fun move() {
        distance++
    }
}

fun main() {
    val car = Car()
    car.move()
    println(car.distance)
    car.distance = 10 // 컴파일 에러 발생
}
```

프로퍼티 getter/setter는 자바로 디컴파일해보면 메서드로 나타난다.

이를 활용하면 직렬화 대상이 아닌 프로퍼티를 `@Transient` 없이도 표현할 수 있다.

```
@Entity
class Entity(
    val price: Int,
    val quantity: Int,
) {
    // 자바 메서드로 컴파일되기 때문에 직렬화 대상에서 벗어난다.
    val amount: Int
        get() = price * quantity
}
```

그럼 언제 프로퍼티를 쓰고 언제 함수를 써야 할까?

코틀린 공식 문서에서 말하는 함수 대신 프로퍼티를 사용해야 하는 경우는 아래와 같다.

💡 몇몇 경우 인자가 없는 함수는 읽기 전용 프로퍼티로 변경될 수 있다.
아래 사항에 해당한다면 함수 대신 프로퍼티를 사용하자.

- 예외를 던지지 않는다.
- 연산 비용이 싸다(또는 데이터가 캐시되어 있다)
- 객체의 상태가 변경되지 않는다면 항상 같은 값을 반환한다.

범위 지정 함수(Scoping Function) Anti 활용법

코틀린을 사용하면서 가장 당황했던 기능이 범위 지정 함수다.

객체에서 범위 지정 함수를 호출하면 임시 범위가 형성되고 이 범위 안에선 객체 내부 프로퍼티에 이름 없이 접근할 수 있다.

범위 지정 함수는 코드를 깔끔하게 작성할 수 있게 도와주지만, 잘못 사용하면 코드의 가독성을 바닥까지 처박아버리기도 한다.

언제 어떤 범위 지정 함수를 써야하는지는 인터넷에 잘 나와있기 때문에 여기서는 범위 지정 함수를 쓸 때 `제발 이렇게는 쓰지 말자` 정도의 코드만 간단히 소개한다.

범위를 최소화 시키자

유우명한 이펙티브 자바 책에선 지역 변수의 범위를 최소화하라는 내용이 있다.

이 내용은 코틀린의 범위 지정 함수에도 `100%` 적용된다.

범위 지정 함수 `run`을 사용한 코드를 보자.

```
data class SimpleRequest(val name: String, val age: Int)

fun simpleWork(simpleRequest: SimpleRequest): String  = simpleRequest.run {
    name + " " + age
}
```

객체의 이름 없이도 프로퍼티에 접근할 수 있다니 너무 편리하다.

하지만 이런 편리함에 중독되면 온갖 곳에 범위 지정 함수를 남발하는 참사가 발생할 수 있다.

대표적인 예는 범위 지정 함수의 범위가 너무 큰 경우다.

특히 범위 지정 함수에 다른 지역 변수가 선언되는 경우 이 지역변수와 수신 객체의 프로퍼티를 구분하기 힘든 경우가 많이 발생한다. 또는 범위 지정 함수 속에 또 다른 범위 지정 함수가 사용되거나.

```
data class PersonRequest(
    val id: Long,
    val name: String,
```

```
    val age: Int,
    val gender: Gender
    // 그 외 수많은 필드들
)


fun longestFunction(personRequest: PersonRequest): PersonDetail = personRequest.run {

    //이제 이 범위에선 id, name, age, gender 같은 이름은 못쓴다고 보면 된다


}
```

따라서 범위 지정 함수는 최대한 적은 범위에만 적용하자.

let으로 null check 하지 말자

인터넷에 떠도는 많은 글이 let으로 null check를 한다.

과연 이 방식이 좋을까? 백문이 불여일견이다. 다음 코드를 보자.

```
fun main() {
    work("aaa")
}

fun work(input: String?) {
    input?.let {
        println("null 1")
        null
    } ?: println("null 2")
}

// null 1
// null 2
```

`?.`과 `let`을 자칫 잘못 사용하면 위 처럼 `?.let`과 `?:` 모두 실행되는 경우가 발생할 수도 있다.

코드 가독성도 if로 null check하는 것 보다 더 떨어진다.

그러니 null check할 때 let 사용은 지양하자.

null을 다루는 다양한 방법들

코틀린은 `nullable 타입`, `non-null` 타입을 구분한다.

개인적으로 `nullable` 타입을 다루는 가장 좋은 방법은 가능한 빨리 `non-null 타입`으로 변환하는 방법이라고 생각한다.

왜냐하면 `nullable 타입`을 사용할 때마다 `null check`를 해야 하는데 이게 여간 귀찮은 작업이 아니다.

```
fun split(input: String?): Set<Int> {
   // 체이닝되는 null safe call 만큼 보기 싫은게 없다
   input?.split(",")?.map { it.toIntOrNull() }?.toSet()
}
```

그렇다고 null을 강제로 없애주는 `!!` 연산자를 사용하는 것도 별로 좋은 방식은 아니라고 생각한다.

왜냐하면 nullable 타입을 사용하면 코틀린 컴파일러가 null 체크를 요구하고, null에 대한 안전한 처리를 유도한다.

하지만 `!!` 연산자를 사용하면 이 모든게 무용지물이 되어 nullable 타입 안전성을 해칠 수 있다.(코드 가독성 저하는 덤이다)

로직에 nullable 타입이 필수가 아니라면 최대한 빨리 non-null로 만들어 사용하자.

> 💡 단 테스트 코드에선 사용해도 될 것 같다. 왜냐하면 테스트 코드는 완벽하게 개발자가 제어하는 환경이기 때문이다.

예외 처리

자바의 예외 처리는 try catch로만 가능했다.

하지만 코틀린은 다양한 방법으로 예외를 처리할 수 있도록 지원한다.

예외 대신 null 반환

예외 대신 null을 반환한다.

장점은 간단하게 예외를 처리할 수 있다. 특히 어떤 예외가 발생했는지 몰라도 되는 경우엔 가장 간단한 예외 처리 방법이다.

단점은 어떤 예외가 발생했는지 알아야 하는 경우 사용하기 힘들다.

```
val a = "1234".toIntOrNull() ?: throw 커스텀 예외()
```

sealed class 사용

예외를 추상화한 `sealed class`를 반환하는 방식이다.

대표적으로 `runCatching`에서 반환하는 `Result` 클래스가 있다.

`Result` 클래스를 반환 받은 곳에서 예외를 명시적으로 제어 할 수 있는 장점이 있다.

```
fun work(input: String): Result<Int> = runCatching {
    input.toInt()
}

fun main() {
    val a = work("1")
    a.onSuccess {
        println("success")
    }

    val b = work("one")
    b.onFailure {
        println("error")
    }
}
```

try-catch

자바 try-catch와 비슷하지만 식이라는 차이가 있다.

```
val happy: Int = try {
    "1".toInt()
} catch (e : Exception) {
    throw e
}
```

use

자바의 try-catch-resource 문과 동일하다.
`Closable` 타입에 사용할 수 있다.

```
Resource resource = new Resource();
try {
    resource.work();
} catch(Exception ex) {
    throw ex
} finally {
    resource.close();
}

// java try-catch-with-resource

try (Resource resource = new Resource();) {
    resource.work();
} catch(Exception ex) {
    throw ex
}
```

```
val resource = Resource()
resource.use {
   work()
}
```

함수형 코틀린

자바의 람다는 사실 인터페이스다.

하지만 코틀린의 람다는 진짜 함수 타입이기 때문에 자바보다 훨씬 유연한 구조의 코드를 작성할 수 있다.

특히 DSL을 정의할 때 그 진가가 들어난다.

```
mockMvc.perform(
  post("/authentications/login")
    .contentType(MediaType.APPLICATION_JSON)
    .content(content)
  ).andDocument(
      //DSL로 작성한 RestDoc 문서
      "관리자가 로그인을 성공하다",
      requestBody {
```

```
            "email" type STRING means "관리자 아이디"
            "password" type STRING means "관리자 비밀번호"
        },
        responseBody {
            "token" type STRING means "관리자 인증 토큰"
        }
    )
```

뿐만 아니라, 고차 함수로 사용하거나 Single Abstract Method(SAM)을 함수형 인터페이스로 선언 후 람다로 작성하는 등 다양한 곳에 활용할 수 있다.

값 클래스(Value Class)

너무 좋은 기능이라고 생각한다.

특히 원시 값(String, Int 등)을 성능 문제 없이 Wrapping 하는 최고의 방법이다.

```
@JvmInline
value class Capacity(val value: Int) : Comparable<Capacity> {

    init {
        require(value >= 0) {
            "Required positive number. Invalid value: $value"
        }
    }

    companion object {
        val ZERO = Capacity(0)
    }
}
```

하지만 슬프게도 실제로 사용하기엔 치명적인 문제가 하나 있다.

바로 Jackson에서 코틀린 값 클래스의 Deserialize를 지원하지 않는다.

(이유는 Jackson은 자바 리플렉션을 사용하기 때문이다, Gson 도 똑같다…)

제네릭

제네릭(Generic)은 개발자가 컴파일 시점에 타입을 확인할 수 있게 해주는 기능이다.

자바 컴파일러는 제네릭을 구현하기 위해 타입 제거(Type Erasure)라는 기능을 사용한다.

타입 제거는 제네릭으로 표현된 모든 타입을 Object 혹은 제한된 타입(Bounded Type)으로 변경한다.

예를 들어 다음 메서드를 컴파일하면

```
private static <E> void print(E[] array) {
    for (E element : array) {
        System.out.printf("%s ", element);
    }
}
```

다음과 같이 컴파일된다.

```
private static void print(Object[] array) {
    for (Object element : array) {
        System.out.printf("%s ", element);
    }
}
```

타입을 지움으로써 제네릭으로 넘어온 타입을 새로 생성할 필요가 없어 실행 시간에 오버헤드가 발생하지 않는다.

더 자세한 내용은...

- https://docs.oracle.com/javase/tutorial/java/generics/erasure.html
- https://www.baeldung.com/java-type-erasure

```
```

Type Eraser로 인해 발생하는 문제

제네릭은 타입 안정성을 보장해주지만 타입 제거로 인해 실행 시간에는 타입이 지워진다.

따라서 실행 시간에 제네릭으로 넘어온 타입 정보를 알 수 없는 문제가 발생한다.

```
private <T> void method(T type) {
    Class<T> classType = T.class; // 컴파일 에러가 발생한다
}
```

따라서 자바에서는 실행 시간에 타입 정보가 필요한 경우(주로 리플렉션을 사용할 때) 클래스의 메타 정보를 함께 넘겨주는 방식을 사용한다.

```
private <T> void method(T type, Class<T> clazz) {
    ...
}
```

위와 같은 방식을 `Super Type Token 패턴`이라고 부른다.

그런데 이 `Super Type Token 패턴`을 코틀린에서 사용하면 코드가 상당히 지저분해질 수 있다.

JSON 문자열을 SampleRequest 클래스로 역직렬화하는 ObjectMapper 클래스를 예로 들어보자.

```
data class SampleRequest(
    val data: String
)
```

역직렬화 시 SampleRequest 클래스를 생성하기 위해 **자바** 클래스의 타입 정보를 함께 넘겨줘야 한다.

```
fun main() {
    val mapper = ObjectMapper()
    val json = """{"data" : "test"}"""
    val sampleRequest = mapper.readValue(json, SampleRequest::class.java)
}
```

이처럼 `Super Type Token 패턴`은 함수를 호출할 때 타입 정보를 함께 넘겨줘야 하기 때문에 코드의 가독성을 해칠 수 있다.

Kotlin Refied Function

코틀린은 놀랍게도 실행 시간에 제네릭 타입을 알 수 있다.

사용법은 굉장히 간단하다. 타입 파라미터에 `refied`를 함께 붙여주면 된다.

```
inline fun <reified T> ObjectMapper.myReadValue(json: String): T
= readValue(json, T::class.java)
```

함수를 호출하는 방법을 보면 차이점이 더 확 와 닿는다.

```
fun main() {
    val mapper = ObjectMapper()
    val json = """{"data" : "test"}"""
    // val sampleRequest = mapper.readValue(json, SampleRequest::class.java)
    val sampleRequest = mapper.myReadValue<SampleRequest>(json)
}
```

누더기 같던 자바 코드보다 훨씬 보기 좋다.

Kotlin Inline Function

사실 `refied`는 inline 함수에서만 사용할수 있다.

왜냐하면 inline 함수는 컴파일 시 함수를 호출하는 것이 아니라 직접 코드가 삽입된다.

따라서 실행 시점에도 클래스 정보가 유지된다.

```
inline fun <reified T> ObjectMapper.myReadValue(json: String): T = readValue(json, T::class.java)


//위 코드가 런타임엔 이렇게 실행된다
fun main() {
    val mapper = ObjectMapper()
    val json = """{"data" : "test"}"""
    val sampleRequest = mapper.readValue(json, SampleRequest::class.java)
}
```

Named Parameter를 활용한 가짜 생성자

named parameter를 활용하면 빌더를 따로 구현할 필요가 없다.

이를 활용해 가짜 생성자를 최상위 함수로 구현하면 테스트 용 fixture를 만들 때 편하다.

```
fun fanClub(
    fanClubId: String = UuidGenerator.generate(),
    artistId: Long,
    artistName: String = "artistName",
    // 그 외 필드
): FanClub = FanClub(
    fanClubId = fanClubId,
    artistId = artistId,
    artistName = artistName,
    // 그 외 필드
)

//실제 사용
val fanClub = fanClub(fanClubId = "id", artistId = 12345)
```

코딩 컨벤션

코틀린은 공식 컨벤션이 있다.

코드가 조금 이상하다 싶을 땐 컨벤션을 참고하자.

https://kotlinlang.org/docs/coding-conventions.html

JPA와 Kotlin

JPA 영속성 컨텍스트는 엔티티의 ID가 null 인 것을 보고 영속 상태를 판단한다.

- id가 null → 영속화 되지 않은 엔티티
- id가 null이 아님 → 한 번이라도 영속화된 엔티티

따라서 엔티티 클래스의 id는 null 타입으로 선언하곤 한다.

```
@Entity
class Domain(
    @Id
    val id: Long?
)
```

하지만 id는 엔티티를 다룰 때 항상 사용한다.

엔티티를 사용할 때마다 매번 null-check를 하는 건 너무 불편하다.

이런 경우 `Persistable<T>` 인터페이스를 구현하는 방법을 고려할 수 있다.

```
@MappedSuperclass
abstract class BaseEntity(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private val id: Long = -1,
) : Persistable<Long> {

    @Transient
```

```
    private var _isNew = true

    override fun getId(): Long = id

    override fun isNew(): Boolean = _isNew

    @PostPersist
    @PostLoad
    protected fun load() {
        _isNew = false
    }
}
```

위 인터페이스를 구현한 엔티티는 영속성 컨텍스트에서 id의 null 유무가 아닌 `isNew()`를 보고 확인하기 때문에 id를 non-null 타입으로 사용할 수 있다.