



섹션 2. Java 서버를 Kotlin 서버로 리팩토링하자!

이번 섹션의 목표

11강. Kotlin 리팩토링 계획 세우기

12강. 도메인 계층을 Kotlin으로 변경하기 - Book.java

13강. 도메인 계층을 Kotlin으로 변경하기 - UserLoanHistory.java, User.java

14강. Kotlin과 JPA를 함께 사용할 때 이야기거리 3가지

15강. 리포지토리를 Kotlin으로 변경하기

16강. 서비스 계층을 Kotlin으로 변경하기 - UserService.java

17강. BookService.java를 Kotlin으로 변경하고 Optional 제거하기

18강. DTO를 Kotlin으로 변경하기

19강. Controller 계층을 Kotlin으로 변경하기

20강. 리팩토링 끝! 다음으로!

Section 3 미리보기

Section 4 미리보기

Section 5 미리보기

Section 6 미리보기

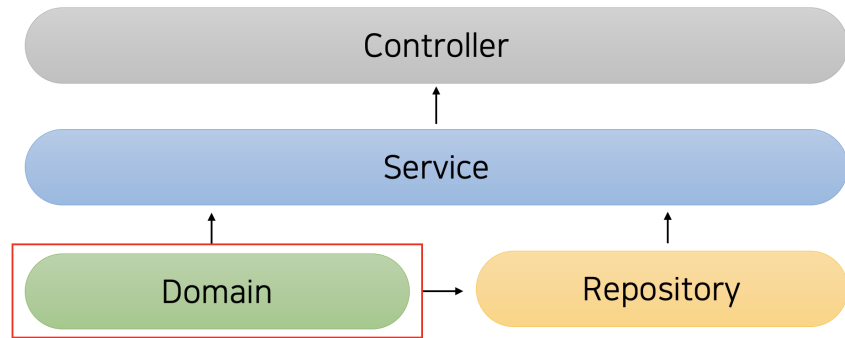
이번 섹션의 목표

1. Java로 작성된 도서관리 애플리케이션을 Kotlin으로 완전히 리팩토링 한다.
2. Kotlin + JPA 코드를 작성하며, 사용에 익숙해진다.
3. Kotlin + Spring 코드를 작성하며, 사용에 익숙해진다.
4. Java 프로젝트를 Kotlin으로 리팩토링 해야 하는 상황에 대한 경험을 쌓는다.

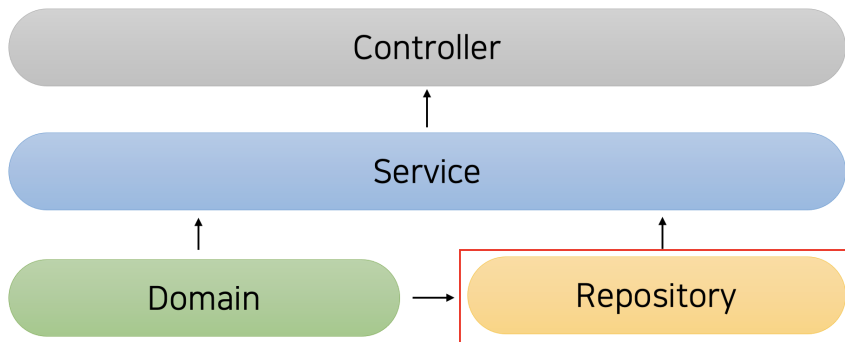
11강. Kotlin 리팩토링 계획 세우기



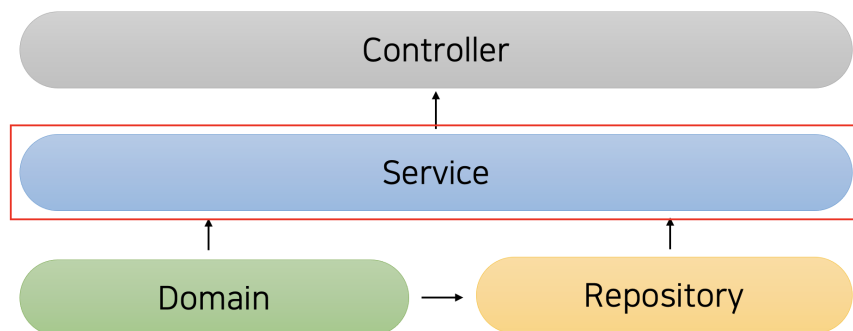
영상과 PPT를 함께 보시면 더욱 좋습니다 😊



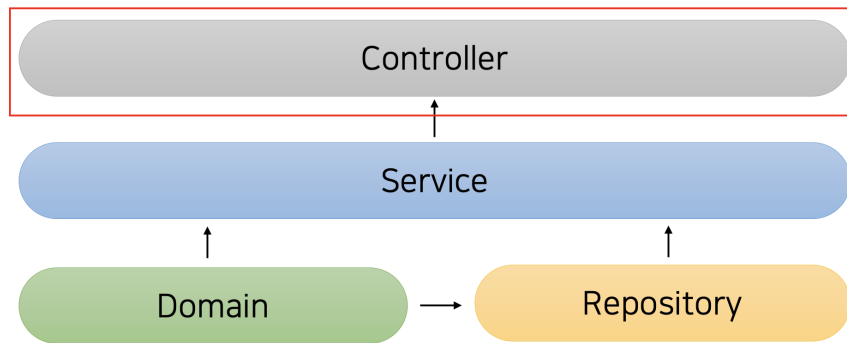
- 가장 먼저 Domain 계층을 Kotlin으로 변경한다.
- Domain 계층의 특징은 POJO이고 JPA Entity라는 점이다.



- 다음으로 Repository를 Kotlin으로 변경한다.
- Repository의 특징은 Spring Bean이고, 다른 Bean에 대한 의존성이 없다는 것이다.



- 세 번째는 대망의 Service 계층을 Kotlin으로 변경한다.
- Service 계층의 특징은 Spring Bean이고, 다른 Bean에 대한 의존성이 있다는 점이다. 또한 많은 비즈니스 로직이 들어간다.

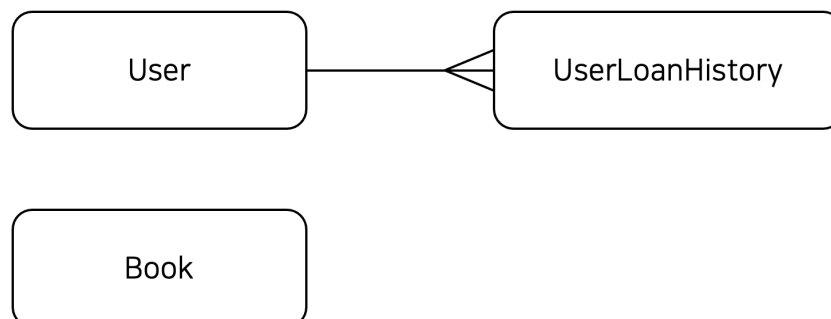


- 마지막으로 Controller와 DTO이다.
- Controller는 Spring Bean이고, 다른 Bean에 대한 의존성이 있다. DTO의 경우 간단하지만 클래스 개수가 많은 것이 특징이다.

다음시간부터 본격적으로 리팩토링을 진행해보자!!

12강. 도메인 계층을 Kotlin으로 변경하기 - Book.java

자 이번 시간에는 Java로 작성된 도메인 계층을 Kotlin으로 바꾸어보겠습니다! 현재 도메인 객체는 3개가 있는데요, 다음과 같은 구성을 가지고 있습니다.

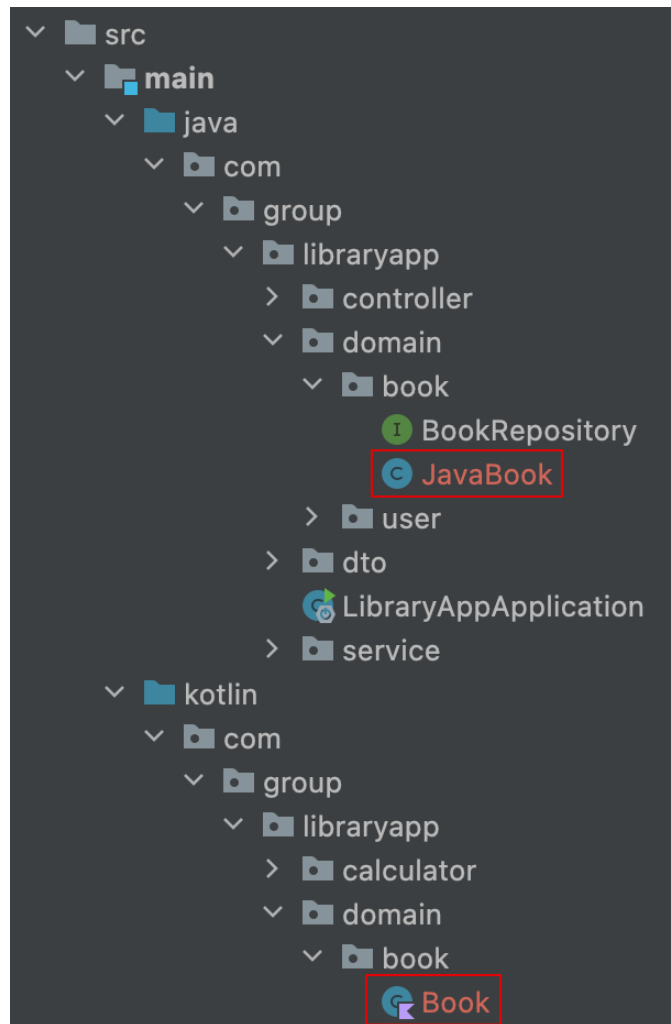


연관관계가 없는 Book부터 해보죠~

우선 Book라는 이름을 모두 JavaBook라고 바꾸겠습니다!

Mac/Windows/Linux 모두 **Shift + F6** 단축키를 사용하시면 쉽게 객체 이름을 변경할 수 있습니다.

그리고 이제 `/src/main/kotlin` 폴더 안, `com.group.libraryapp.domain.book` 패키지에 `Book.kt` 파일을 만들어주자!



잘 만들어졌다면 위와 같은 모습일 것이다!! 플러그인과 git 사용 유무에 따라 모양이나 색이 약간은 다를 수 있다. 핵심은 `.kt` 파일이 `/src/main/kotlin` 에 기존 패키지와 동일한 패키지 내 생성되었다는 점이다.

```
class Book {  
}
```

Book 도메인은 2가지 필드를 가지고 있다. id와 name이다. 두 필드를 추가해보자.

```
class Book(  
    val name: String,  
    val id: Long? = null,  
)
```

기존 Java 코드와 비교해 몇 가지를 살펴보자!

- id와 name 모두 불변 기능이니 `val` 이라는 지시어를 붙여 주었다.

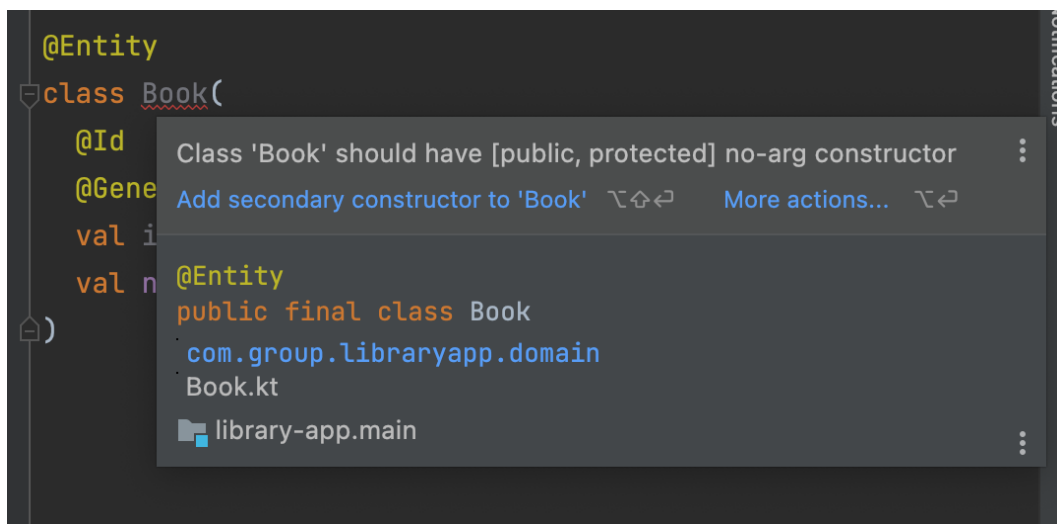
- name은 NULL이 들어갈 수 없으니 `String` 을 사용해주었다.
- id는 `java.lang.Long` 타입으로 null이 가능하기 때문에 코틀린의 `Long?` 을 사용해주었고 생성자에 대한 default parameter로 null을 넣어 주었다.

Book 클래스를 만들었다! 이제 이 클래스를 JPA의 Entity로 간주하도록 `@Entity` 어노테이션을 붙여주고, id 필드에 필요한 어노테이션도 붙여주었다.

```
@Entity
class Book(
    val name: String,

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long? = null,
)
```

여기까지 하고 나면, 다음과 같은 에러가 발생한다.



Class 'Book' should have [public, protected] no-arg constructor

이 에러가 발생하는 이유는 다음과 같다.

- JPA를 사용하기 위해서는 아무런 argument를 받지 않는 기본 생성자가 필요하다.

- 하지만 코틀린 코드에서는 '주 생성자'를 만들 때 프로퍼티를 함께 만들어주는 방식을 사용함으로써, 아무런 argument를 받지 않는 기본 생성자가 존재하지 않는다.
- 때문에 에러가 발생한다.

이 에러를 해결해주기 위해서는 다음과 같은 kotlin-jpa 플러그인이 필요하다.

```
// build.gradle
plugins {
    id "org.jetbrains.kotlin.plugin.jpa" version "1.6.21"
}
```

플러그인을 입력해 준 후, gradle을 refresh 하거나 IntelliJ를 잠시 종료했다, 다시 실행시켜 주자

그럼 이제 위의 에러는 사라지게 된다. 좋다~!! 👍 이제 이어서 빠진 로직을 구현해보자. 기존 JavaBook과 비교했을 때 빠진 부분은 Book을 생성할 때 name이 비어 있는지 확인하는 부분이다.

코틀린에서는 init block 을 활용해 객체 생성시점에, 생성자의 파라미터를 확인해줄 수 있다.

```
@Entity
class Book(
    val name: String,

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long? = null,
) {

    init {
        if (name.isBlank()) {
            throw IllegalArgumentException("이름은 비어 있을 수 없습니다")
        }
    }
}
```

기존에 `JavaBook.java` 를 사용했던 부분을 모두 제거하고 `Book.kt` 로 바꿔주자!

- 가장 먼저, BookRepository 부분을 바꿔주자

- 다음으로 BookService 부분에서 에러가 나는 부분을 따라가며 수정해주자.
 - id에는 우선 null을 넣어주자
- 그리고 BookServiceTest 도 변경해주자!

위의 과정을 수행하다, 한 두가지를 놓쳐도 상관없다! 우리는 테스트 코드가 있기 때문에, 테스트 코드를 실행했을 때 에러가 나는 부분으로 이동해 고쳐주면 된다.

좋다~ 에러가 나는 부분을 모두 제거하고 테스트를 돌려보면 ... 테스트가 실행된다!!! 하지만, 모든 테스트가 깨지게 되는데, 가장 근본적으로 발생한 에러는

```
nested exception is java.lang.NoClassDefFoundError: kotlin/reflect/full/KClasses
```

이 에러는 코틀린 클래스에 대한 리플렉션을 할 수 없어 발생하는데, 이를 해결하기 위해 Kotlin 리플렉션 라이브러리를 넣어주어야 한다. 리플렉션이란, 클래스나 메소드 등을 런타임으로 제어하기 위한 기술을 의미한다.

```
dependencies {
    implementation("org.jetbrains.kotlin:kotlin-reflect:1.6.21")
}
```

gradle refresh를 하고 전체 테스트를 돌려보자! 모든 테스트가 성공했다~!! 우리는 성공적으로 Book 도메인을 Kotlin으로 변경한 것이다~!! 😊🎉

다음 시간에는 이어서 User와 UserLoanHistory 도메인을 Kotlin으로 변경해보자.

13강. 도메인 계층을 Kotlin으로 변경하기 - UserLoanHistory.java, User.java

이번 시간에는 남은 도메인 계층을 모두 Kotlin으로 변경할 예정이다. 우선 UserLoanHistory 부터 처리해보자.

Book.java를 처리했던것처럼

- 기존의 UserLoanHistory를 JavaUserLoanHistory로 변경하고
- `UserLoanHistory.kt` 클래스를 `src/main/kotlin/` 내 동일한 패키지에 만들어 주었다.

- 다음으로, 불변 여부(var, val) / 널 가능성 여부를 확인해서 필요한 필드들을 생성자에 만들어주었다.
- 마지막으로, 추가적인 로직을 작성해주었다.

```
@Entity
class UserLoanHistory(

    @ManyToOne
    val user: User,

    val bookName: String,

    var isReturn: Boolean,

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long? = null
) {

    fun doReturn() {
        this.isReturn = true
    }

}
```

이제 JavaUserLoanHistory를 지우고 모두 UserLoanHistory로 대체하자.

JavaUserLoanHistory.java 를 제거하면 아래와 같은 코드에서 컴파일 에러가 날 것이다.

- UserLoanHistoryRepository.java
- User.java
- BookServiceTest.java

다 찾지 못해도 괜찮다! 테스트 코드 전체를 실행시켜 통과하면 된다.

다음으로 User.java를 Kotlin 코드로 리팩토링 하자. User를 JavaUser로 바꾸고 Kotlin을 이용해 새로운 User 클래스를 만들었다.

```
@Entity
class User(
    var name: String,

    val age: Int?,

    @OneToMany(mappedBy = "user", cascade = [CascadeType.ALL], orphanRemoval = true)
    val userLoanHistories: MutableList<UserLoanHistory> = mutableListOf(),
)
```



```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
val id: Long? = null,
) {

}

```

1 : N 관계를 정의한 `@OneToMany` 에서 `cascade = [CascadeType.ALL]` 부분을 보자.

Java에서는 `CascadeType[] cascade()` 라는 `@OneToMany` 필드에 `cascade = CascadeType.ALL` 을 바로 넣어주어도 됐지만, Kotlin에서는 배열 타입 어노테이션 필드에는 정확히 배열을 넣어 주어야 한다.

그리고 `userLoanHistories` 의 타입은 `MutableList` 가 되었다. `add()` 를 통해 `UserLoanHistory`를 추가해야 되기 때문이다.

이제 남은 기능은 `updateName()` `loanBook()` `returnBook()` 이다
하나씩 함수를 집어 넣어보자.

```

fun updateName(name: String) {
    this.name = name
}

```

`updateName()`은 간단하다!

```

fun loanBook(book: Book) {
    this.userLoanHistories.add(UserLoanHistory(this, book.name))
}

```

`loanBook()` 역시 마찬가지로 간단하다! 주의할점으로는 `Book.kt` 에 있던 `user: JavaUser` 를 `user: User` 로 변경해주어야 한다! id는 default parameter가 사용되어 null이 들어갈 것이다.

`isReturn` 필드 역시 default parameter를 사용할 수 있다!

```

@Entity
class UserLoanHistory(

    @ManyToOne
    val user: User,

    val bookName: String,

```

```

var isReturn: Boolean = false, // 여기 ' = false'를 넣어주자!

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
val id: Long? = null
) {

```

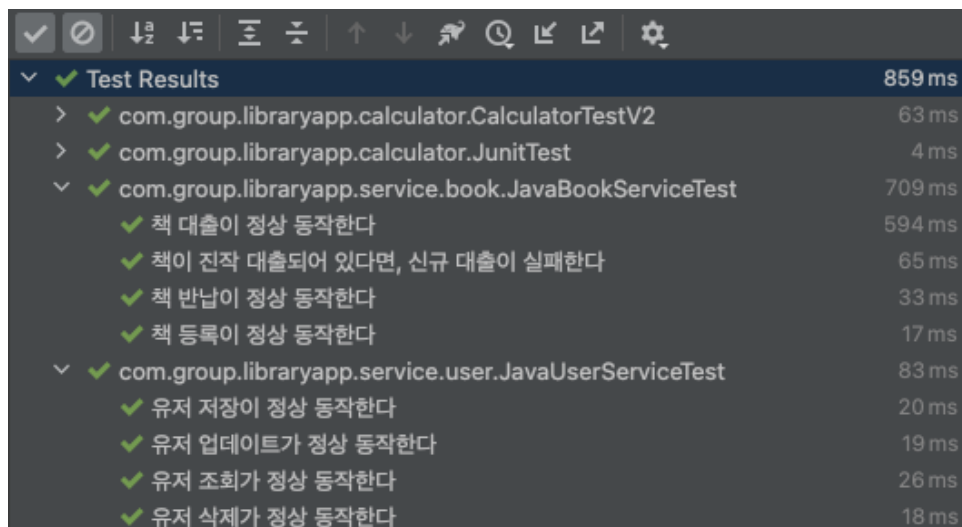
```

fun returnBook(bookName: String) {
    this.userLoanHistories.first { history -> history.bookName == bookName }.doReturn()
}

```

`returnBook` 도 코틀린의 FP를 사용하니 코드가 매우 간결해졌다.

다시 한 번 컴파일이 깨지는 부분을 모두 처리해주고, 전체 테스트를 돌려보자.



Test Name	Duration
Test Results	859 ms
com.group.libraryapp.calculator.CalculatorTestV2	63 ms
com.group.libraryapp.calculator.JunitTest	4 ms
com.group.libraryapp.service.book.JavaBookServiceTest	709 ms
책 대출이 정상 동작한다	594 ms
책이 진작 대출되어 있다면, 신규 대출이 실패한다	65 ms
책 반납이 정상 동작한다	33 ms
책 등록이 정상 동작한다	17 ms
com.group.libraryapp.service.user.JavaUserServiceTest	83 ms
유저 저장에 정상 동작한다	20 ms
유저 업데이트가 정상 동작한다	19 ms
유저 조회가 정상 동작한다	26 ms
유저 삭제가 정상 동작한다	18 ms

모든 테스트가 동작한다!!! 👍

매우 좋다~~ 우리는 도메인 계층 전부를 Java 코드에서 Kotlin 코드로 리팩토링했다!! 테스트 덕분에 모든 기능이 동작함을 보장할 수도 있다!! 다음 시간에는 Kotlin과 JPA를 함께 사용할 때 이야기거리 3가지를 살펴보자.

14강. Kotlin과 JPA를 함께 사용할 때 이야기거리 3가지

이번 시간에는 Kotlin과 JPA를 함께 사용할 때 이야기할만한 주제 3가지에 대해 살펴보자.
여기는 PPT를 먼저 만들어야 할 듯... 음 사실 PPT만 만들면 될듯 ㅋㅋㅋ

[1] setter에 관한 이야기

`User.kt` Entity를 보자. 사실 User 클래스에는 생성자 안의 var 프로퍼티가 있어 setter를 사용할 수 있지만, setter 대신 `updateName()` 이라는 추가적인 함수가 구현되어 있다.

```
@Entity
class User(
    var name: String,

    val age: Int?,

    @OneToMany(mappedBy = "user", cascade = [CascadeType.ALL], orphanRemoval = true)
    val userLoanHistories: MutableList<UserLoanHistory> = mutableListOf(),

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long? = null,
) {

    fun updateName(name: String) {
        this.name = name
    }

    fun loanBook(book: Book) {
        this.userLoanHistories.add(UserLoanHistory(this, book.name))
    }

    fun returnBook(bookName: String) {
        this.userLoanHistories.first {history -> history.bookName == bookName }.doReturn()
    }
}
```

이는 밖에서 setter를 바로 호출하는 것보다, 좋은 이름의 함수를 사용하는 것이 훨씬 좋은 코드이기 때문이다. 하지만 현재 name에 대한 setter는 public으로 열려있기 때문에 유저 이름 업데이트 기능에서 setter를 사용할 '수도' 있다. 즉, 코드상 setter가 완전히 사용 불가능한 것은 아니라는 의미이다.

어떻게 하면 setter를 완전히 막을 수 있을까?

첫 번째 방법은 4강에서 소개했던 backing property를 사용하는 것이다.

```
class User(
    private var _name: String
) {
```

```

val name: String
    get() = this._name
}

```

이렇게 `_name` 이라는 프로퍼티를 만들고, 읽기 전용으로 추가 프로퍼티 `name` 을 만든다. 두 번째 방법은 custom setter를 사용하는 방법이다.

```

class User(
    name: String // 프로퍼티가 아닌, 생성자 인자로만 name을 받는다
) {

    var name = name
        private set

}

```

User의 생성자에서 name을 프로퍼티가 아닌, 생성자 인자로만 받고 이 name을 변경가능한 name 프로퍼티로 넣어주되, name 프로퍼티에 private setter를 달아두는 것이다.

두 방법 모두 괜찮지만, 프로퍼티가 많아지면 번거롭다는 단점이 있다.

때문에 **개인적으로는** setter를 열어 두되 사용하지 않는 방법을 선호한다. 마음의 불편함을 더는 쪽을 선택한 것이다. 다행히 현재 팀에서도 setter를 사용하면 안된다는 Align이 완전히 맞은 상태라 개발을 하는데 큰 문제는 없는 상황이다.

결국 Trade-Off의 영역이라 생각하고, 팀간의 컨벤션을 잘 맞추는 것이 중요하지 않을까 싶다.

[2] 생성자 안의 프로퍼티, 클래스 body 안의 프로퍼티

다시 User 클래스를 보자. User 클래스 주생성자 안에 있는 `userLoanHistories` 와 `id` 는 꼭 주생성자 안에 있을 필요가 없다. 아래와 같이 코드가 바뀔 수 있는 것이다.

```

@Entity
class User(
    var name: String,

    val age: Int?,
) {

    @OneToMany(mappedBy = "user", cascade = [CascadeType.ALL], orphanRemoval = true)
    val userLoanHistories: MutableList<UserLoanHistory> = mutableListOf(),
}

```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
val id: Long? = null,

fun updateName(name: String) {
    this.name = name
}

fun loanBook(book: Book) {
    this.userLoanHistories.add(UserLoanHistory(this, book.name))
}

fun returnBook(bookName: String) {
    this.userLoanHistories.first {history -> history.bookName == bookName }.doReturn()
}
}

```

그렇다. 위 코드도 잘 동작한다. 그렇다면 어떻게 해야 더 좋을까?

개인적으로는 큰 상관이 없다고 생각한다. 테스트를 하기 위한 객체를 만들어 줄 때도 정적 팩토리 메소드를 사용하다 보니 프로퍼티가 안에 있건, 밖에 있건 두 경우 모두 적절히 대응할 수 있다.

하지만 명확한 가이드가 있는 것은 함께 개발을 할 때에 중요하므로

1. 모든 프로퍼티를 생성자에 넣거나
2. 프로퍼티를 생성자 혹은 클래스 body 안에 구분해서 넣을 때 명확한 기준이 있거나

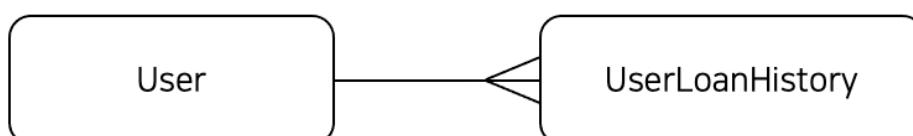
해야한다고 생각한다.

[3] JPA와 data class

다음 주제는 JPA와 data class이다. 결론부터 말하면, JPA Entity는 data class를 피하는 것이 좋다.

data class는 equals, hashCode, toString 등의 함수를 자동으로 만들어준다. 사실 원래 세 함수는 JPA Entity와 궁합이 그렇게 좋지 못했다. 연관관계 상황에서 문제가 될 수 있는 경우들이 존재했기 때문이다.

예를 들어, 현재 프로젝트에 있는 User와 UserLoanHistory를 생각해보자.

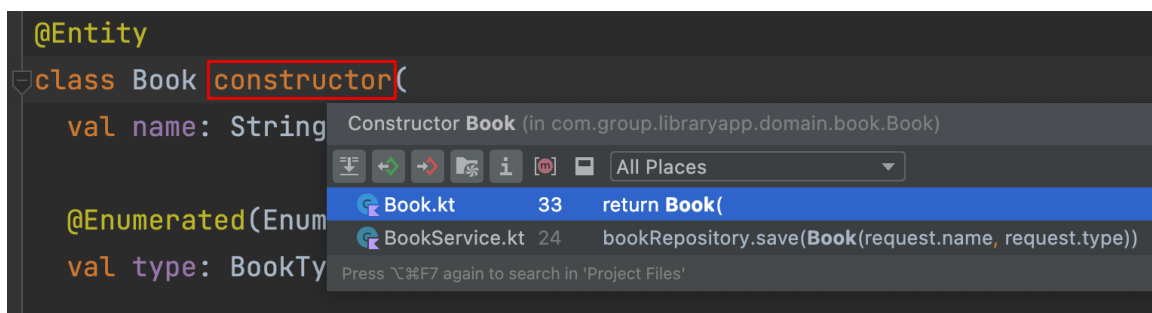


1 : N 연관관계를 맺고 있는 상황에서 User 쪽에 `equals()` 가 호출된다면, User는 본인과 관계를 맺고 있는 UserLoanHistory의 `equals()` 를 호출하게 되고, 다시 UserLoanHistory는 본인과 관계를 맺고 있는 User의 `equals()` 를 호출하게 된다.

때문에 JPA Entity는 data class를 피하는 것이 좋다.

마지막으로 작은 TIP 하나를 더 공유드린다. 이 TIP 꼭 JPA와 관련된 것은 아니지만, 현재 Kotlin Class가 Domain만 있으니 이 단계에서 말씀드리고자 한다.

- Entity (Class) 가 생성되는 로직을 찾고 싶은 경우, constructor 지시어를 명시적으로 작성하고 추적하면 훨씬 편하다.



이제 Kotlin으로 변경된 도메인 객체들을 남겨두고, Repository로 넘어가보자!

15강. 리포지토리를 Kotlin으로 변경하기

이번 시간에는 Java로 만들어진 3개의 Repository를 Kotlin으로 변경할 예정이다.

먼저 BookRepository이다. 이번엔 바로 `BookRepository.kt` 를 만들고 `findByName()` 메소드도 구현해보자.

```
interface BookRepository : JpaRepository<Book, Long> {  
  
    fun findByName(bookName: String): Optional<Book>  
  
}
```

Kotlin에서는 Optional을 사용하지 않아도 null 가능성을 표시할 수 있지만, 우선은 Service 계층의 구현 변경을 최소화 하기 위해 `Optional<Book>` 을 반환 타입으로 해주었다.

이렇게 처리해주고, 기존의 `BookRepository.java` 를 제거하면 패키지 자체가 동일해 다른 프로덕션 및 테스트 코드를 바꾸지 않아도 모든 테스트 코드가 통과한다.

이제 `UserLoanHistoryRepository.kt` 와 `UserRepository.kt` 를 만들어주고 똑같이 Java로 구성된 Repository를 제거하자.

```
interface UserLoanHistoryRepository : JpaRepository<UserLoanHistory, Long> {  
  
    fun findByBookNameAndIsReturn(bookName: String, isReturn: Boolean): Optional<UserLoanHistory>  
  
}
```

```
interface UserRepository : JpaRepository<User, Long> {  
  
    fun findByName(userName: String): Optional<User>  
  
}
```

간단하다!! 이제 우리는 Domain, 그리고 Repository 까지 Kotlin 코드로 바꾸었다. 다음 시간에는 Service 계층에 대해서 처리해보자.

16강. 서비스 계층을 Kotlin으로 변경하기 - UserService.java

이번 시간에는 `UserService.java` 를 코틀린으로 변경해볼 것이다.

자 우선 `@Service` 어노테이션을 붙인 `UserService.kt` 클래스를 `src/main/kotlin/` 안의 `com.group.libraryapp.service.user` 패키지에 만들어주자!

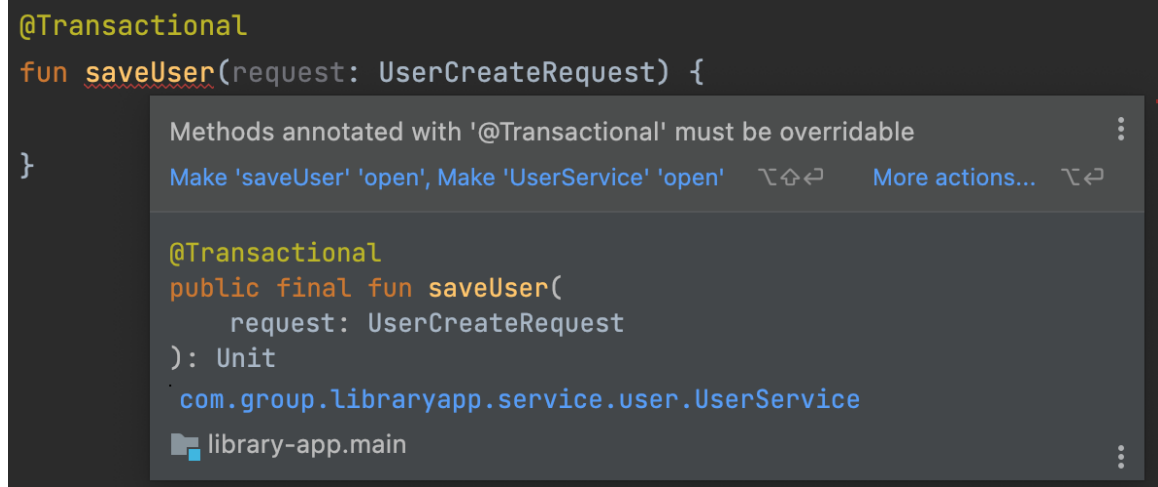
```
@Service  
class UserService {  
  
}
```

이제 UserRepository를 주입 받아야 하는데, Kotlin에서도 Java에서 사용했던 것처럼 생성자를 사용해 주입 받을 수 있다.

```
@Service
class UserService(
    private val userRepository: UserRepository,
) {

}
```

UserService 클래스를 만들 때 주생성자에 UserRepository를 명시해줌으로써 Bean이 주입되었다!! 이제 `saveUser()` 메소드를 만들고 `@Transactional` 어노테이션을 붙여보자. 그랬더니 아래와 같은 에러가 발생한다.



```
@Transactional
fun saveUser(request: UserCreateRequest) {

}
```

Methods annotated with '@Transactional' must be overridable

Make 'saveUser' 'open', Make 'UserService' 'open' More actions...

```
@Transactional
public final fun saveUser(
    request: UserCreateRequest
): Unit
    com.group.libraryapp.service.user.UserService
    library-app.main
```

Methods annotated with '@Transactional' must be overridable

이 에러는 다음 이유로 발생한다.

- `@Transactional` 어노테이션이 정상 동작하려면, `UserService.saveUser()` 메소드를 상속 받을 수 있어야 한다.
- 하지만 Kotlin에서는 기본적으로 final class, final method이다. 즉, 클래스에 대한 상속이나 메소드에 대한 오버라이드가 불가능하다.
- 때문에 에러가 발생한다.

이 에러에 대한 해결책은 다음 플러그인을 사용해주는 것이다! kotlin-spring 플러그인을 사용하면 필요한 클래스에 대해 상속과 오버라이드를 자동 허용해준다.

```
plugins {
    id "org.jetbrains.kotlin.plugin.spring" version "1.6.21"
```



```
}
```

kotlin-jpa 플러그인을 적용했을 때와 마찬가지로, 1) gradle을 refresh 해주고 2) 필요에 따라서 IntelliJ를 종료했다가 다시 열어주자. 그러면 에러가 사라진다!! 😊

이제 `saveUser()` 의 로직을 구현해보자.

```
@Transactional
fun saveUser(request: UserCreateRequest) {
    val newUser = User(request.name, request.age)
    userRepository.save(newUser)
}
```

Kotlin 코드의 default parameter를 활용할 수 있으므로 다음과 같이 코드가 작성되었다.
User 객체를 인스턴스화 하는 부분이 충분히 짧아 다음과 같이 축약해도 괜찮을 것 같다!

```
@Transactional
fun saveUser(request: UserCreateRequest) {
    userRepository.save(User(request.name, request.age))
}
```

다음은 `getUsers()` 이다.

```
@Transactional(readOnly = true)
fun getUsers(): List<UserResponse> {
    return userRepository.findAll()
        .map { user -> UserResponse(user) }
}
```

람다를 사용하는 map을 쓰게 되면 위와 같이 바꿔줄 수 있으며, 상황에 따라서는 `.map { UserResponse(it) }` 을 사용할 수도 있다.

생성자 레퍼런스를 사용하고 싶다면, `UserResponse::new` 였던 Java와 다르게 `::` 이 클래스 이름 앞으로 와야 한다.

```
@Transactional(readOnly = true)
fun getUsers(): List<UserResponse> {
    return userRepository.findAll()
        .map(::UserResponse)
}
```

매우 좋다~ Java Stream에 비해 조금 더 간결한 함수형 프로그래밍이 가능하다.

다음은 `updateUserName()` 과 `deleteUser()` 이다. 크게 어려운 부분은 없으니 한 번에 코틀린 코드로 변경해보자!

```
@Transactional
fun updateUserName(request: UserUpdateRequest) {
    val user = userRepository.findById(request.id).orElseThrow(::IllegalArgumentException)
    user.updateName(request.name)
}

@Transactional
fun deleteUser(name: String) {
    val user = userRepository.findByName(name).orElseThrow(::IllegalArgumentException)
    userRepository.delete(user)
}
```

간단하다~! 다음 시간에는 바로 이어서 `UserService.java` 를 코틀린으로 변경하고, Java에서 사용했던 Optional을 제거해보도록 하자.

17강. BookService.java를 Kotlin으로 변경하고 Optional 제거하기

이번 시간에는 `BookService.java` 를 Kotlin으로 변경하고 Repository에서 Optional을 제거할 예정이다.

`BookService.kt` 를 만들고 Java 코드를 Kotlin으로 옮겨주자!

```
@Service
class BookService(
    private val bookRepository: BookRepository,
    private val userRepository: UserRepository,
    private val userLoanHistoryRepository: UserLoanHistoryRepository,
) {
```

```

@Transactional
fun saveBook(request: BookRequest) {
    bookRepository.save(Book(request.name))
}

@Transactional
fun loanBook(request: BookLoanRequest) {
    if (userLoanHistoryRepository.findByBookNameAndIsReturn(request.bookName, false) !
= null) {
        throw IllegalArgumentException("진작 대출되어 있는 책입니다")
    }

    val book = bookRepository.findByName(request.bookName).orElseThrow(::IllegalArgume
ntException)
    val user = userRepository.findByName(request.userName).orElseThrow(::IllegalArgume
ntException)
    user.loanBook(book)
}

@Transactional
fun returnBook(request: BookReturnRequest) {
    val user = userRepository.findByName(request.userName).orElseThrow(::IllegalArgume
ntException)
    user.returnBook(request.bookName)
}
}

```

- BookService는 3가지 Bean을 주입 받고 있으므로 생성자에 3가지 private 프로퍼티를 넣어주었다.
- 이전 UserService와 마찬가지로 `::IllegalArgumentException` 와 같은 생성자 레퍼런스를 사용했다.

`BookService.kt` 의 구현이 모두 끝나면 `BookService.java` 를 제거하고 전체 테스트를 돌려보자.

Test Results	859 ms
> com.group.libraryapp.calculator.CalculatorTestV2	63 ms
> com.group.libraryapp.calculator.JunitTest	4 ms
com.group.libraryapp.service.book.JavaBookServiceTest	709 ms
✓ 책 대출이 정상 동작한다	594 ms
✓ 책이 진작 대출되어 있다면, 신규 대출이 실패한다	65 ms
✓ 책 반납이 정상 동작한다	33 ms
✓ 책 등록이 정상 동작한다	17 ms
com.group.libraryapp.service.user.JavaUserServiceTest	83 ms
✓ 유저 저장에 정상 동작한다	20 ms
✓ 유저 업데이트가 정상 동작한다	19 ms
✓ 유저 조회가 정상 동작한다	26 ms
✓ 유저 삭제가 정상 동작한다	18 ms

테스트가 모두 성공한다면, 서비스 계층 역시 Java에서 Kotlin으로 성공적인 리팩토링을 한 것이다~! 👍

좋다~ 이제 Java에서 Kotlin으로 넘어오면서 사용을 하지 않아도 되는 Optional을 제거해보자.

JDK8에서 등장한 Optional은 null이 될 가능성을 가진 값을 wrapping 하기 위해 생긴 타입으로, Kotlin에서는 타입 시스템에서 `?`를 활용해 null 가능성을 판단할 수 있기 때문에 더 이상 사용할 필요가 없다.

우선 `UserRepository` 와 `BookRepository` 의 메소드 시그니처를 다음과 같이 변경하자.

```
// UserRepository.kt
fun findByName(userName: String): User?

// BookRepository.kt
fun findByName(bookName: String): Book?
```

Optional을 제거하고 `User?` `Book?` 으로 바꿔주었다. 만약 주어진 이름을 가진 유저나 책이 없다면 null이 반환 될 것이다.

Optional을 제거하였기 때문에 서비스 계층에서 에러가 나오게 될 것이다. 아래는 예시 사진이다.

```

@Transactional
fun loanBook(request: BookLoanRequest) {
    userLoanHistoryRepository.findByBookNameAndIsReturn(request.bookName, isReturn: false)
        ?: throw IllegalArgumentException("진작 대출되어 있는 책입니다")

    val book = bookRepository.findByName(request.bookName).orElseThrow(::IllegalArgumentException)
    val user = userRepository.findByName(request.userName).orElseThrow(::IllegalArgumentException)
    user.loanBook(book)
}

```

이런 부분 역시 Elvis 연산자로 해결해줄 수 있다.

```

@Transactional
fun loanBook(request: BookLoanRequest) {
    if (userLoanHistoryRepository.findByBookNameAndIsReturn(request.bookName, false) !=
        null) {
        throw IllegalArgumentException("진작 대출되어 있는 책입니다")
    }
    val book = bookRepository.findByName(request.bookName) ?: throw IllegalArgumentException()
    val user = userRepository.findByName(request.userName) ?: throw IllegalArgumentException()
    user.loanBook(book)
}

```

조금 더 나아가보자.

이렇게 `?: throw IllegalArgumentException()` 이 반복되게 되면 나중에 예외 종류를 바꾸어야 한다거나, 적절한 메시지를 넣어주어야 할 때 많은 부분에 변경이 일어나게 된다.

때문에 이 부분을 함수로 만들어 처리해주자.

`com.group.libraryapp.util` 패키지에 `ExceptionUtils.kt` 파일을 만들고, 그 안에 `fail()` 이란 함수를 만들어주었다.

```

fun fail(): Nothing {
    throw IllegalArgumentException()
}

```

`Nothing` 타입은 이 함수는 항상 정상적으로 종료되지 않는다는 의미이다.

매우 좋다~ 😊 `BookRepository.findByIdName()` 을 사용하는 로직과,
`UserRepository.findByIdName()` 을 사용하는 로직을 모두 `?: fail()` 로 처리해주었다!
하지만 아직 `Optional`을 사용하고 있는 코드가 있다. 바로 `updateUserName()` 이다.

```
@Transactional
fun updateUserName(request: UserUpdateRequest) {
    val user = userRepository.findById(request.id).orElseThrow(::IllegalArgumentException)
    user.updateName(request.name)
}
```

이 코드에서 사용하는 `findById()` 는 `CrudRepository.java` 의 메소드이기 때문에 우리가 컨트롤 할 수 없다. 이 코드에서도 `Optional`을 제거할 수 있을까?

코틀린의 **확장함수**를 사용하면 가능하다! 스프링 프레임워크는 Kotlin과 `CrudRepository`를 함께 사용할 때 이런 상황에 대비하여 `CrudRepositoryExtension.kt` 이라는 확장함수가 담긴 파일을 만들어 두었다.

```
fun <T, ID> CrudRepository<T, ID>.findByIdOrNull(id: ID): T? =
    findById(id).orElse(null)
```

우리는 이 함수를 사용하면 다음과 같이 `updateUserName()` 을 바꿀 수 있다!

```
@Transactional
fun updateUserName(request: UserUpdateRequest) {
    val user = userRepository.findByIdOrNull(request.id) ?: fail()
    user.updateName(request.name)
}
```

아주 좋다~!! 여기서 한 단계 더 나아갈 수 있다! 우리도 확장함수를 사용해

- id를 통해 Entity를 조회했을 때 존재하지 않으면 지금처럼 `IllegalArgumentException`을 던지는 함수

를 만들어 보자!

아까 만들어두었던 `com.group.libraryapp.util` 패키지에 `JpaRepositoryExtension.kt` 파일을 만들고 `findByIdOrThrow()` 라는 `CrudRepository`의 확장 함수를 다음과 같이 만들어주었다.

```
fun <T, ID> CrudRepository<T, ID>.findByIdOrThrow(id: ID): T {
    return this.findByIdOrNull(id) ?: fail()
}
```

간단하다~! 핵심은 이 함수의 반환타입이 `T?` 가 아닌 `T` 라는 것이다!

이 코드를 사용하면 어떤 도메인의 Repository이건 `findByIdOrThrow` 를 사용해 변경의 여파가 적은 예외처리를 할 수 있다!! 이제 UserService를 최종적으로 수정해주자!

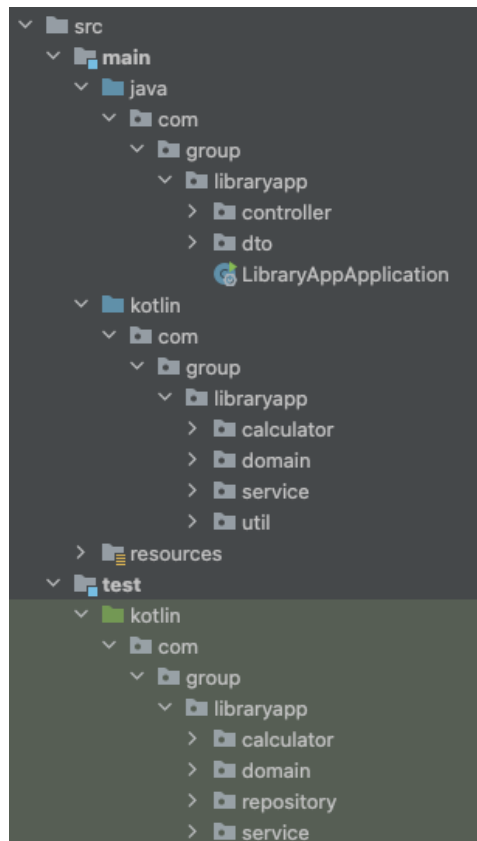
```
@Transactional
fun updateUserName(request: UserUpdateRequest) {
    val user = userRepository.findByIdOrThrow(request.id)
    user.updateName(request.name)
}
```

원래 코드와 비교했을 때 조금 더 간결해진 모습을 보인다~! 이것이 바로 Kotlin 확장함수의 힘이다!

이제 다음 시간에는 마지막으로 Controller 계층과 DTO를 Kotlin으로 변경해보자!

18강. DTO를 Kotlin으로 변경하기

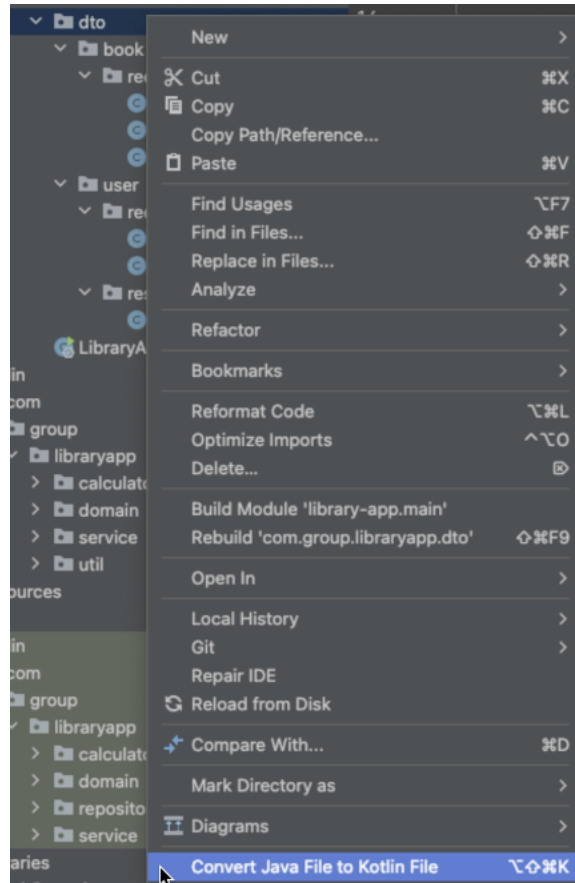
지금까지 우리는 Kotlin으로 테스트를 작성하고 Domain, Repository, Service를 코틀린으로 옮겨왔다. 현재의 패키지 구조는 다음과 같은 모습일 것이다.



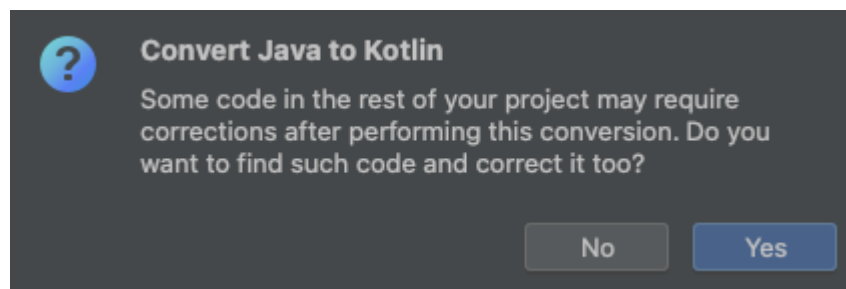
이번 시간에는 DTO를 Kotlin으로 변경해 볼 예정이다. DTO는 클래스 하나하나의 양이 많지 않지만 클래스가 6개나 된다. 때문에 지금까지와는 또 다른 방법을 사용해보려고 한다! 바로 IntelliJ에서 지원하는 **Convert Java File to Kotlin File** 기능이다.

이 기능은 파일 단위, 혹은 더 넓은 단위로 사용할 수도 있다.

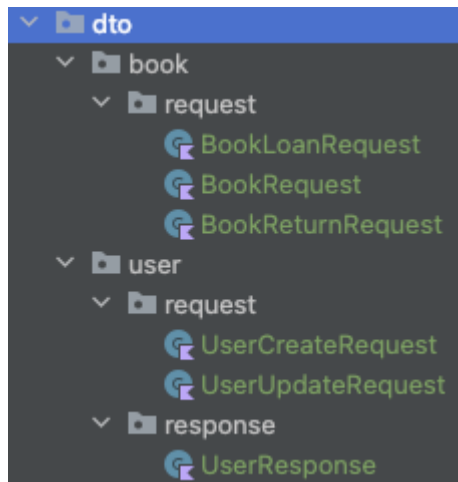
dto 패키지에서 우클릭을 하고 **Convert Java File to Kotlin File** 을 클릭해보자.



그러면, IntelliJ가 자동으로 Java 파일들을 Kotlin 파일로 변경해준 뒤, 아래와 같은 확인 창을 띄워준다. Java 파일을 Kotlin 파일로 모두 변경했는데, 확인을 해보겠느냐는 의미이다.



여기서 YES를 누르면, 아래와 같이 Kotlin 파일들로 변해 있는 DTO들을 확인할 수 있다.



정말 잘 변했는지 몇 가지 확인을 해보자.

예를 들어 책 대출 요청에서 사용된 BookLoanRequest는 Java로 구현되어 있을 때 다음과 같이 생겼었다.

```
public class BookLoanRequest {  
  
    private String userName;  
    private String bookName;  
  
    public BookLoanRequest(String userName, String bookName) {  
        this.userName = userName;  
        this.bookName = bookName;  
    }  
  
    public String getUsername() {  
        return userName;  
    }  
  
    public String getBookName() {  
        return bookName;  
    }  
  
}
```

이 클래스는 코틀린의 class로 바뀌어 다음과 같이 잘 변한 것을 확인할 수 있다.

```
class BookLoanRequest(val userName: String, val bookName: String)
```

UserCreateRequest는 원래 `Integer` 타입의 age를 가지고 있었는데, Kotlin에서는 다음과 같이 변경되었다.

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

원래 의미에 부합될 수 있도록, `val age: Int?` 로 `?` 을 붙여주자.

또다른 예로 `UserResponse`는 원리 아래와 같은 java 파일이었다.

```
public class UserResponse {

    private final long id;
    private final String name;
    private final Integer age;

    public UserResponse(User user) {
        this.id = user.getId();
        this.name = user.getName();
        this.age = user.getAge();
    }

    public long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public Integer getAge() {
        return age;
    }

}
```

위와 같았던 `UserResponse.java` 클래스는 다음과 같은 `UserResponse.kt` 로 바뀌게 된다.

```
class UserResponse(user: User) {
    val id: Long
    val name: String
    val age: Int?

    init {
        id = user.id!!
        name = user.name
        age = user.age
    }
}
```

하지만 이 Kotlin Class 보다는 다음 Kotlin Class가 좋다.

```
class UserResponse(  
    val id: Long,  
    val name: String,  
    val age: Int?  
) {  
  
    constructor(user: User): this(  
        id = user.id!!,  
        name = user.name,  
        age = user.age  
    )  
  
}
```

그리고 이렇게 부생성자를 사용하는 것보다는 정적 팩토리 메소드를 사용하면 더 좋다.

```
class UserResponse(  
    val id: Long,  
    val name: String,  
    val age: Int?  
) {  
  
    companion object {  
        fun of(user: User): UserResponse {  
            return UserResponse(  
                id = user.id!!,  
                name = user.name,  
                age = user.age  
            )  
        }  
    }  
  
}
```

생성자 방식에서 정적 팩토리 메소드 방식으로 변경되었으니 UserService의 로직도 잘 챙겨주자.

```
@Transactional(readOnly = true)  
fun getUsers(): List<UserResponse> {  
    return userRepository.findAll()  
        .map(UserResponse::of)  
}
```

좋다 전체적으로 잘 바뀐 것 같다. 추가로 DTO의 경우는 class 대신 data class가 적합하다.

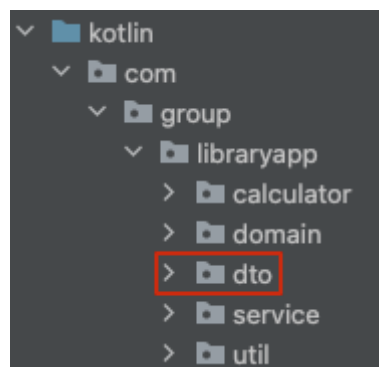
현재는 equals, hashCode, toString을 사용하고 있지 않지만 DTO의 의미를 생각해 보았을 때 언제라도 사용될 수 있기 때문이다.

그래서 class를 data class로 바꿔주고 (개인의 선호에 따라) 클래스 포매팅을 정리해주면 된다.

예를 들어 BookLoanRequest 클래스는 다음과 같이 바뀌게 된다.

```
data class BookLoanRequest(  
    val userName: String,  
    val bookName: String  
)
```

다른 클래스들도 적절하게 처리해주고, dto 패키지 전체를 `src/main/kotlin` 폴더의 `com.group.libraryapp` 패키지 아래로 옮겨주자!



전체 테스트도 한 번 수행해보자~! 😊

다음 시간에 이어, 마지막 남은 Controller 계층까지 Kotlin으로 변경할 예정이다.

19강. Controller 계층을 Kotlin으로 변경하기

이제 Controller를 옮길 차례이다. BookController부터 `BookController.kt` 를 만들어 코딩하고 `BookController.java` 를 제거하는 방식으로 진행할 것이다. 그리고 두 Controller와 `@SpringBootApplication` 이 붙어 있는 `LibraryAppApplication` 까지 바꾼 후에는 UI를 열어, 각 기능이 정상적으로 동작하는 최종적인 테스트를 해볼 것이다.

```
@RestController  
class BookController(  
    private val bookService: BookService,
```

```

) {

    @PostMapping("/book")
    fun saveBook(@RequestBody request: BookRequest) {
        bookService.saveBook(request)
    }

    @PostMapping("/book/loan")
    fun loanBook(@RequestBody request: BookLoanRequest) {
        bookService.loanBook(request)
    }

    @PutMapping("/book/return")
    fun returnBook(@RequestBody request: BookReturnRequest) {
        bookService.returnBook(request)
    }

}

```

UserController도 이어서 변경해보자.

```

@RestController
class UserController(
    private val userService: UserService,
) {

    @PostMapping("/user")
    fun saveUser(@RequestBody request: UserCreateRequest) {
        userService.saveUser(request)
    }

    @GetMapping("/user")
    fun getUsers(): List<UserResponse> {
        return userService.getUsers()
    }

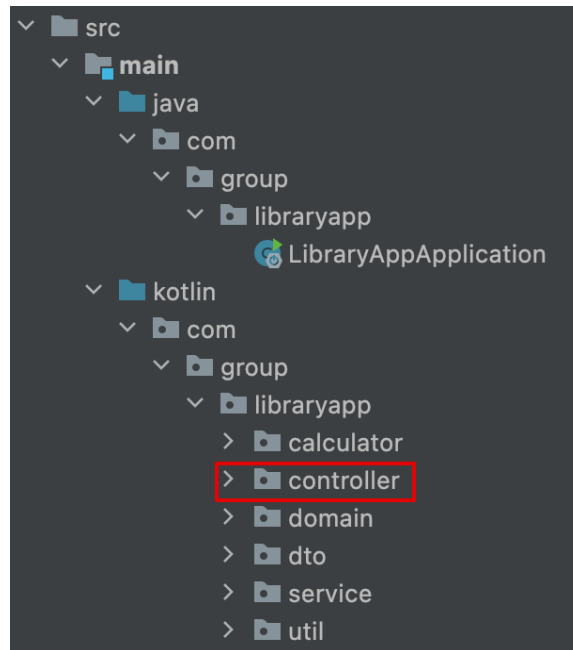
    @PutMapping("/user")
    fun updateUsername(@RequestBody request: UserUpdateRequest) {
        userService.updateUserName(request)
    }

    @DeleteMapping("/user")
    fun deleteUser(@RequestParam name: String) {
        userService.deleteUser(name)
    }

}

```

마지막에 사용한 `@RequestParam name: String` 의 경우, String에 null을 허용하지 않았다. 바꿔 말하면 name이라는 쿼리는 API를 요청할 때 필수라는 의미이다. 만약 `@RequestParam name: String?` 과 같이 null이 허용된다면, `@RequestParam(required = false)` 를 해주지 않더라도 API 요청시 name이라는 쿼리는 선택이 된다.



Controller까지 모두 옮겼다! 이제 현존하는 마지막 Java Class인 `LibraryAppApplication` 도 Kotlin으로 옮겨보자.

```
@SpringBootApplication
class LibraryAppApplication

fun main(args: Array<String>) {
    runApplication<LibraryAppApplication>(*args)
}
```

Spring에서 CrudRepository와 관련해 Kotlin을 지원하기 위해 `findByIdOrNull` 을 만들어 두었던 것처럼 SpringBootApplication 역시 `runApplication()` 이라는 inline 함수를 만들어 두었다.

<자바 개발자를 위한 코틀린 입문> 강의에서 다루었던 것처럼, 함수를 top line에 사용하게 되면 Java의 static 함수를 쓴 것과 동일하게 처리되어, Java로 만들어진 기존

`LibraryAppApplication` 과 동일한 효과를 갖게 된다.

이제 UI를 열어 기능들을 간단히 확인해보자.

등록하기 목록			
사용자 이름	나이		
최태현	100세	수정	삭제
김태현	99세	수정	삭제

```
Resolved [org.springframework.http.converter.HttpMessageNotReadableException: JSON parse error: Cannot construct instance of `com.group.libraryapp.dto.book.request.BookRequest` (although at least one Creator exists): cannot deserialize from Object value (no delegate- or property-based Creator); nested exception is com.fasterxml.jackson.databind.exc.MismatchedInputException: Cannot construct instance of `com.group.libraryapp.dto.book.request.BookRequest` (although at least one Creator exists): cannot deserialize from Object value (no delegate- or property-based Creator)<EOL> at [Source: (org.springframework.util.StreamUtils$NonClosingInputStream); line: 1, column: 2]]
```

```
implementation 'com.fasterxml.jackson.module:jackson-module-kotlin:2.13.3'
```


gradle refresh를 하고, 서버를 재시작한 다음 다시 한 번 UI로 테스트해보면, 기능이 정상 동작하는 것을 확인할 수 있다.

20강. 리팩토링 끝! 다음으로!

우리는 기존에 존재하던 Java 기반의 도서관리 애플리케이션에 대해 Kotlin으로 테스트를 작성하고, 프로덕션 코드 역시 Kotlin 코드로 모두 교체하였다!! 🎉🎉🎉

이 과정에서 우리는 3가지 리팩토링 방법을 적절히 섞어서 사용했다.

- 기존 코드를 남겨두고 에러 나지 않는 새로운 클래스를 만들어 하나씩 교체하기
- 새로운 클래스를 만들어 한 번에 교체하기
- IntelliJ의 기능을 활용해 Java 파일을 Kotlin 파일로 만들고 수정하여 교체하기

또한 우리는 **Section 2. Java 서버를 Kotlin 서버로 리팩토링하자!** 를 통해 다음과 같은 내용을 배울 수 있었다.

1. Kotlin과 JPA를 함께 사용하는 방법과 주의할 점
2. Kotlin과 Spring을 함께 사용하는 방법
3. Spring Application에서 Kotlin의 언어적 특성을 활용하는 방법

다음 Section부터는 새로운 요구사항을 Kotlin으로 구현할 것이다. 요구사항을 미리 확인해보면 다음과 같다.

Section 3 미리보기

책 등록 요구사항 추가

- 책을 등록할 때에 '분야'를 선택해야 한다.
 - 분야에는 5가지 분야가 있다 - 컴퓨터 / 경제 / 사회 / 언어 / 과학

Section 4 미리보기

유저 대출 현황 화면

- 유저 대출 현황을 보여준다.

등록하기 목록 히스토리 통계

사용자 이름	책 이름	대여 상태
번개맨	엘리스를 찾아서	반납 완료
아이언맨	-	-

- 과거에 대출했던 기록과 현재 대출 중인 기록을 보여준다.
- 아무런 기록이 없는 유저도 화면에 보여져야 한다.

Section 5 미리보기

책 통계 화면

- 현재 대여 중인 책이 몇 권이 보여준다.
- 분야별로 도서관에 등록되어 있는 책이 각각 몇 권인지 보여준다.

등록하기 목록 히스토리 통계

등록된 책 분류	총 권 수
과학	3권
사회	5권
경제	12권
컴퓨터	4권
언어	1권

대출 중인 권 수 : 10권

Section 6 미리보기

기술적인 요구사항

- 현재 사용하는 JPQL은 몇 가지 단점이 있다.
- Querydsl을 적용해서 단점을 극복하자.



Query DSL