7️⃣

Section7. 의존관계 자동 주입

다양한 의존관계 주입 방법

의존관계 주입 방법은 크게 4가지가 있다.

1. 생성자 주입
2. 수정자 주입
3. 필드 주입
4. 일반 메서드 주입

생성자 주입

- 이름 그대로 생성자를 통해서 의존 관계 주입
  - 지금까지 진행했던 방법
- 특징
  - 생성자 호출시점에 딱 1번만 호출되는 것이 보장된다.
  - **"불변, 필수"** 의존관계에 사용
  - **아래 코드처럼 생성자가 딱 1개만 있는 경우에는 @Autowired 생략 가능**

```
@Component
public class OrderServiceImpl implements OrderService {
    private final MemberRepository memberRepository;
    private final DiscountPolicy discountPolicy;

    @Autowired
    public OrderServiceImpl(MemberRepository memberReposit
        this.memberRepository = memberRepository;
        this.discountPolicy = discountPolicy;
    }

    @Override
    public Order createOrder(Long memberId, String itemNam
```

```
        Member member = memberRepository.findById(memberId
        int discountPrice = discountPolicy.discount(member

        return new Order(memberId, itemName, itemPrice, di
    }

    public MemberRepository getMemberRepository(){
        return memberRepository;
    }
}
```

수정자 주입

- `setter`라 불리는 필드의 값을 변경하는 수정자 메서드를 통해서 의존관계를 주입하는 방법이다.
- 특징
  - **"선택, 변경"** 가능성이 있는 의존관계에 사용
  - 자바빈 프로퍼티 규약의 수정자 메서드 방식을 사용하는 방법이다

```
@Component
public class OrderServiceImpl implements OrderService {
 private MemberRepository memberRepository;
 private DiscountPolicy discountPolicy;

 @Autowired
 public void setMemberRepository(MemberRepository memberRe
 this.memberRepository = memberRepository;
 }

 @Autowired
 public void setDiscountPolicy(DiscountPolicy discountPoli
 this.discountPolicy = discountPolicy;
 }
}
```

💡 @Autowired 의 기본 동작은 주입 대상이 없으면 오류를 발생시킨다.

이를 피하기 위해서는 @Autowired(required = false)로 지정하면 된다.

💡 ※ 자바빈 프로퍼티 ※
자바에서는 과거부터 필드의 값을 직접 변경하기보다는 getter, setter를 이용하여 값을 읽거나 수정한다.

필드 주입

- 이름 그대로 필드에 바로 주입하는 방법이다.
- 특징
  - 외부에서 변경이 불가능해서 테스트 하기 힘들다는 치명적인 단점이 있다.
    - 순수 자바 테스트 시, 기존 생성자 주입 같은 경우 테스트 코드에서 `Service` 를 생성하고 생성자를 호출하면서 DI 가 이루어지지만, 필드 주입의 경우 외부에서 따로 DI를 해줄 수 없기때문에 `nullException` 발생
  - DI 프레임워크가 없으면 아무것도 할 수 없다.
  - **사용하지 말자!**

```
@Component
public class OrderServiceImpl implements OrderService {
 @Autowired
 private MemberRepository memberRepository;
 @Autowired
 private DiscountPolicy discountPolicy;
}
```

옵션 처리

주입할 스프링 빈이 없어도 동작해야 할 때가 있다.

`@Autowired` 의 `required` 기본값이 `true` 로 되어 있어 자동 주입 대상이 없으면 오류가 발생한다.

- 자동 주입 대상을 옵션으로 처리하는 방법 3가지

- `@Autowired(required = false)`
  - 자동 주입할 대상이 없으면 수정자 메서드 자체가 호출 안됨
- `org.springframework.lang.@Nullable`
  - 자동 주입할 대상이 없으면 null이 입력된다.
- `Optional<>`
  - 자동 주입할 대상이 없으면 `Optional.empty` 가 입력

```
//호출 안됨
@Autowired(required = false)
public void setNoBean1(Member member) {
 System.out.println("setNoBean1 = " + member);
}
//null 호출
@Autowired
public void setNoBean2(@Nullable Member member) {
 System.out.println("setNoBean2 = " + member);
}
//Optional.empty 호출
@Autowired(required = false)
public void setNoBean3(Optional<Member> member) {
 System.out.println("setNoBean3 = " + member);
}
```

생성자 주입 선택

DI 프레임워크 대부분이 생성자 주입을 권장한다. 그 이유는 다음과 같다.

불변

- 대부분의 의존관계 주입은 한번 일어나면 종료시점까지 의존관계 변경할 일이 없다.
- 수정자 주입 사용 시, public 으로 `setter` 를 열어두어야 한다.
- 누군가 실수로 변경할 수 있고, 변경하면 안되는 메서드를 열어두는 것은 좋은 설계가 아니다.
- 생성자 주입은 객체를 생성할 때 딱 1번만 호출되므로 이후 호출되는 일이 없다.

누락

- 수정자 의존관계인 경우 컴파일 단계에서는 문제가 발생하지 않아, 실행시킬 때 **런타임 에러**가 발생한다.

final 키워드

- `final` 키워드를 사용함으로써 DI 누락을 피할 수 있다.
  - **컴파일 에러**
- 오직 생성자 주입 방식만 `final`을 사용할 수 있다.

롬복과 최신 트렌드

막상 개발을 해보면, 대부분이 불변이다.

그래서 보통 생성자에 `final` 키워드를 사용한다.

필드 주입처럼 편리하게 사용하는 방법을 알아보자.

롬복 설정

- [Settings] - [Compiler] - [Annotation Processor]: enable 로 설정

```
// build.gradle 추가

//lombok 설정 추가 시작
configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}
//lombok 설정 추가 끝

dependencies {
    //lombok 라이브러리 추가 시작
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testCompileOnly 'org.projectlombok:lombok'
    testAnnotationProcessor 'org.projectlombok:lombok'
    //lombok 라이브러리 추가 끝
```

```
    implementation 'org.springframework.boot:spring-boot-star
    testImplementation 'org.springframework.boot:spring-boot-
}
```

롬복 사용 코드

- 아래 코드를 보면 기존 생성자 역할을 `RequiredArgsConstructor` 가 대체해준다.

```
@Component
@RequiredArgsConstructor // final 이 붙어있는 필드 멤버를 활용하여
public class OrderServiceImpl implements OrderService {
 private final MemberRepository memberRepository;
 private final DiscountPolicy discountPolicy;
}
```

정리

최근에는 생성자를 딱 1개 두고, `@Autowired` 를 생략하는 방법을 주로 사용한다.

Lombok 라이브러리의 `@RequiredArgsConstructor` 함께 사용하면 기능은 다 제공하면서, 코드는 깔끔하게 사용할 수 있다.

조회 빈이 2개 이상 - 문제

`@Autowired` 는 타입(Type)으로 조회한다.

```
@Autowired
private DiscountPolicy discountPolicy
```

타입으로 조회하기 때문에, 마치 다음 코드와 유사하게 동작한다.

```
ac.getBean(DiscountPolicy.class)
```

타입으로 조회하면 **선택된 빈이 2개 이상일 때 문제가 발생**한다.

- `DiscountPolicy` 의 하위 타입인 `FixDiscountPolicy` , `RateDiscountPolicy` 둘다 스프링 빈으로 선언해 보자
    - `NoUniqueBeanDefinitionException` 오류가 발생한다.

- 이때 하위 타입으로 지정할 수 도 있지만, 하위 타입으로 지정하는 것은 DIP를 위배하고 유연성이 떨어진다.
- 이름만 다르고, 완전히 똑같은 타입의 스프링 빈이 2개 있을 때 해결이 안된다.

스프링 빈을 수동 등록해서 문제를 해결해도 되지만, 의존 관계 자동 주입에서 해결하는 여러 방법이 있다.

@Autowired 필드 명, @Qualifier, @Primary

해결 방법을 하나씩 알아보자.

조회 대상 빈이 2개 이상일 때 해결 방법

- `@Autowired` 필드 명 매칭
- `@Qualifier` `@Qualifier`끼리 매칭 빈 이름 매칭
- `@Primary` 사용

@Autowired 필드 명 매칭

필드 명을 빈 이름으로 변경

```
@Autowired
private DiscountPolicy rateDiscountPolicy
```

@Autowried 매칭 정리

1. 타입 매칭
2. 타입 매칭의 결과가 2개 이상일 때 필드 명, 파라미터 명으로 빈 이름 매칭

@Qualifier 사용

`@Qualifier` 는 추가 구분자를 붙여주는 방법이다.

주입 시 추가적인 방법을 제공하는 것이지 빈 이름을 변경하는 것은 아니다.

빈 등록시 `@Qualifier`를 붙여 준다.

```
@Component
@Qualifier("mainDiscountPolicy")
public class RateDiscountPolicy implements DiscountPolicy {}
```

```
@Component
@Qualifier("fixDiscountPolicy")
public class FixDiscountPolicy implements DiscountPolicy {}
```

주입시에 `@Qualifier`를 붙여주고 등록한 이름을 적어준다.

```
// 생성자 자동 주입 예시
@Autowired
public OrderServiceImpl(MemberRepository memberRepository,
 @Qualifier("mainDiscountPolicy") DiscountPolicy
discountPolicy) {
 this.memberRepository = memberRepository;
 this.discountPolicy = discountPolicy;
}

// 수정자 자동 주입 예시
@Autowired
public DiscountPolicy setDiscountPolicy(@Qualifier("mainDisco
DiscountPolicy discountPolicy) {
 this.discountPolicy = discountPolicy;
}
```

다음과 같이 직접 빈 등록시에도 `@Qualifier`를 동일하게 사용할 수 있다.

```
@Bean
@Qualifier("mainDiscountPolicy")
public DiscountPolicy discountPolicy() {
 return new ...
}
```

`@Qualifier` 정리

1. `@Qualifier`끼리 매칭
2. 빈 이름 매칭
3. `NoSuchBeanDefinitionException` 예외 발생

@Primary 사용

`@Primary` 는 우선순위를 정하는 방법이다. `@Autowired` 시에 여러 빈이 매칭되면 `@Primary` 가 우선권을 가진다.

```
@Component
@Primary
public class RateDiscountPolicy implements DiscountPolicy {}

@Component
public class FixDiscountPolicy implements DiscountPolicy {}
```

`@Qualifier` 의 단점은 주입 받을 때 모든 코드에 `@Qualifier` 를 붙여주어야 한다는 점이다.

💡 **※ @Primary, @Qualifier 활용 ※**

코드에서 자주 사용하는 메인 데이터베이스의 커넥션을 획득하는 스프링 빈이 있고, 코드에서 특별한 기능으로 가끔 사용하는 서브 데이터베이스의 커넥션을 획득하는 스프링 빈이 있다고 생각해보자.

메인 데이터베이스의 커넥션을 획득하는 스프링 빈은 `@Primary` 를 적용해서 조회하는 곳에서 `@Qualifier` 지정 없이 편리하게 조회하고, 서브 데이터베이스 커넥션 빈을 획득할 때는 `@Qualifier` 를 지정해서 명시적으로 획득 하는 방식으로 사용하면 코드를 깔끔하게 유지할 수 있다.

물론 이때 메인 데이터베이스의 스프링 빈을 등록할 때 `@Qualifier` 를 지정해주는 것은 상관없다.

💡 **※ 우선순위 ※**

`@Primary` 는 기본값 처럼 동작하는 것이고, `@Qualifier` 는 매우 상세하게 동작한다. 이런 경우 어떤 것이 우선권을 가져갈까?

스프링은 자동보다는 수동이, 넓은 범위의 선택권 보다는 좁은 범위의 선택권이 우선 순위가 높다. 따라서 여기서도 `@Qualifier` 가 우선권이 높다.

애노테이션 직접 만들기

`@Qualifier("mainDiscountPolicy")` 이렇게 문자를 적으면 컴파일시 타입 체크가 안된다.

다음과 같은 애노테이션을 만들어서 문제를 해결할 수 있다.

```
// 애노테이션 만들기
package hello.core.annotataion;
import org.springframework.beans.factory.annotation.Qualifier
import java.lang.annotation.*;
@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PA
ElementType.TYPE, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Qualifier("mainDiscountPolicy")
public @interface MainDiscountPolicy {}

// 애노테이션 선언
@Component
@MainDiscountPolicy
public class RateDiscountPolicy implements DiscountPolicy {}

//생성자 자동 주입
@Autowired
public OrderServiceImpl(MemberRepository memberRepository,
 @MainDiscountPolicy DiscountPolicy discountPolicy) {
 this.memberRepository = memberRepository;
 this.discountPolicy = discountPolicy;
}

//수정자 자동 주입
@Autowired
public DiscountPolicy setDiscountPolicy(@MainDiscountPolicy D
discountPolicy) {
 this.discountPolicy = discountPolicy;
}
```

애노테이션에는 상속이라는 개념이 없다. 이렇게 여러 애노테이션을 모아서 사용하는 기능은 스프링이 지원해주는 기능이다.

`@Qualifier` 뿐만 아니라 다른 애노테이션들도 함께 조합해서 사용할 수 있다.

단적으로 `@Autowired` 도 재정의 할 수 있다.

물론 스프링이 제공하는 기능을 뚜렷한 목적 없이 무분별하게 재정의 하는 것은 유지보수에 더 혼란만 가중할 수 있다.

조회한 빈이 모두 필요할 때, List, Map

의도적으로 정말 해당 타입의 스프링 빈이 다 필요한 경우도 있다.

예를 들어서 할인 서비스를 제공하는데, 클라이언트가 할인의 종류(rate, fix)를 선택할 수 있다고 가정해보자.

스프링을 사용하면 소위 말하는 전략 패턴을 매우 간단하게 구현할 수 있다.

```
public class AllBeanTest {

    @Test
    void findAllBean(){
        ApplicationContext ac = new AnnotationConfigApplicati

        DiscountService discountService = ac.getBean(Discount
        Member member = new Member(1L, "userA", Grade.VIP);
        int discountPrice = discountService.discount(member,

        Assertions.assertThat(discountService).isInstanceOf(D
        Assertions.assertThat(discountPrice).isEqualTo(1000);

        int rateDiscountPrice = discountService.discount(memb

        Assertions.assertThat(discountService).isInstanceOf(D
        Assertions.assertThat(rateDiscountPrice).isEqualTo(20

    }

    static class DiscountService{
        private final Map<String, DiscountPolicy> policyMap;
        private final List<DiscountPolicy> policyList;

        public DiscountService(Map<String, DiscountPolicy> po
            this.policyMap = policyMap;
```

```
            this.policyList = policyList;
            System.out.println("policyMap = " + policyMap);
            System.out.println("policyList = " + policyList);
        }

        public int discount(Member member, int price, String
            DiscountPolicy discountPolicy = policyMap.get(dis
            return discountPolicy.discount(member, price);
        }
    }
}
```

로직 분석

- `DiscountService`는 `Map`으로 모든 `DiscountPolicy`를 주입받는다. 이때 `fixDiscountPolicy`, `rateDiscountPolicy`가 주입된다.
- `discount()` 는 `discountCode`로
  - "fixDiscountPolicy"가 넘어오면 map에서 `fixDiscountPolicy` 스프링 빈을 찾아서 실행한다.
  - “rateDiscountPolicy”가 넘어오면 `rateDiscountPolicy` 스프링 빈을 찾아서 실행한다.

주입 분석

- `Map<String, DiscountPolicy>` : map의 키에 스프링 빈의 이름을 넣어주고, 그 값으로 `DiscountPolicy` 타입으로 조회한 모든 스프링 빈을 담아준다.
- `List<DiscountPolicy>` : DiscountPolicy 타입으로 조회한 모든 스프링 빈을 담아준다.
- 만약 해당하는 타입의 스프링 빈이 없으면, 빈 컬렉션이나 Map을 주입한다.

자동, 수동 빈 등록의 운영 기준

- 편리한 자동 기능을 기본으로 사용하자
- 직접 등록하는 기술 지원 객체는 수동 등록
- 다형성을 적극 활용하는 비즈니스 로직은 수동 등록 고민