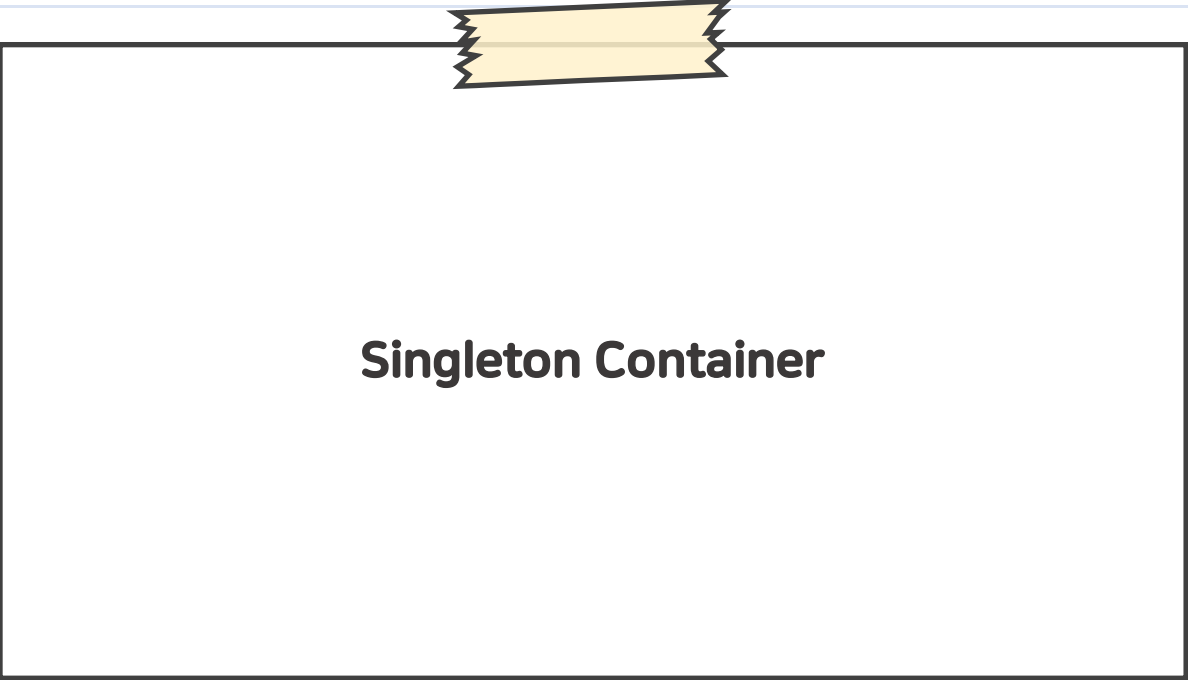




Singleton, Component Scan, Autowired

2020 Nov 5, SKKRYPTO Back-End Study Junho Bae



Singleton Container

웹 어플리케이션과 싱글톤

Why Singleton?

- ✓ 스프링은 기업용 온라인 서비스 기술을 지원하기 위해 탄생했으나, 웹 어플리케이션이 대다수
- ✓ 웹 어플리케이션은 여러 고객이 동시에 요청을 하는 경우가 매우 많다.
- ✓ 스프링이 없는 순수한 DI 컨테이너의 경우 이 요청에 대해 매번 서로 다른 객체를 만들어서 전달해야함.

```
"C:\Program Files\Java\jdk-11.0.8\bin\java.exe" ...
```

```
memberService1 = hello.core.member.MemberServiceImpl@38c6f217
```

```
memberService2 = hello.core.member.MemberServiceImpl@478190fc
```

```
Process finished with exit code 0
```

**=> 지나치게 비 효율적. 딱 하나만 생성해서
생성된 객체 인스턴스를 공유하자.**

Singleton Container

싱글톤 패턴(디자인 패턴)

Singleton?

- ✓ 클래스의 인스턴스가 딱 1개만 생성되는 것을 보장하는 디자인 패턴
- ✓ 똑같은 객체 타입의 인스턴스를 2개 이상 생성하지 못하도록 막아야 한다!

```
public class SingletonService {  
  
    // 자기 자신을 내부에 static으로. class level에 올라가기 때문에 딱 하나만 만들어짐.  
    // static 영역에 요거 하나만 만들어져서 올라갑니다.  
    private static final SingletonService instance = new SingletonService();  
  
    public static SingletonService getInstance() {  
        return instance;  
    }  
  
    private SingletonService() {  
    }  
  
    public void logic() {  
        System.out.println("출력");  
    }  
}
```

1) static 영역에 객체 Instance를 하나 생성해서 가지고 있다.

2) 생성자를 private으로 만들어 다른 곳에서는 생성하지 못하게 한다.

3) 객체의 인스턴가 필요하다면 오직 getInstance()를 통해서만 접근할 수 있도록.

```
public static void main(String[] args) {  
    SingletonService singletonService = new SingletonService();  
}
```

'SingletonService()' has private access in 'hello.core.singleton.SingletonService'
Make 'SingletonService.SingletonService' package-private Alt+Shift+Enter More actions... Alt+Enter
hello.core.singleton.SingletonService
@Contract(pure = true)
private SingletonService()
Inferred annotations: @org.jetbrains.annotations.Contract(pure = true)

1 of 1 test - 338 ms
Files\Java\jdk-11.8.8\bin\java.exe ...

Singleton Container

싱글톤 패턴(디자인 패턴)

*생성자가 **Private**으로 되어 있으면 "싱글톤 패턴으로 디자인 했구나" 생각하자

```
@Test
@DisplayName("싱글톤 패턴을 적용한 객체 사용")
void singletonServiceTest() {
    SingletonService singletonService1 = SingletonService.getInstance();
    SingletonService singletonService2 = SingletonService.getInstance();
    System.out.println("singletonService1 = " + singletonService1);
    System.out.println("singletonService2 = " + singletonService2);
}
```

✓ 싱글톤 패턴이 적용되어 인스턴스를 호출하는
Test Code

✓ Tests passed: 1 of 1 test – 64 ms

"C:\Program Files\Java\jdk-11.0.8\bin\java.exe" ...

```
singletonService1 = hello.core.singleton.SingletonService@57a3af25
singletonService2 = hello.core.singleton.SingletonService@57a3af25
```

Process finished with exit code 0

✓ 동일한 Instance가 생성되고 있다.

Singleton Container

싱글톤 패턴(디자인 패턴)

Spring에서의 싱글톤

- ✓ 사실 스프링 컨테이너가 객체들을 **알아서 싱글톤으로** 관리를 해준다.

싱글톤 패턴의 문제점?

1. 코드가 많아진다.
2. 클라이언트가 구체 클래스에 의존한다.
3. 테스트가 너무 어렵다.
4. 내부 속성 변경이 어렵다.
5. 유연성이 떨어진다. 자식 클래스를 만들기도 어렵다.
6. 안티 패턴으로 불리기도 한다.



하지만 스프링 프레임워크는 싱글톤 패턴의 문제점을 **전부 해결**하면서 장점만 취하고 있음.

Singleton Container

싱글톤 컨테이너

* 스프링 컨테이너는 싱글톤 컨테이너의 역할을 한다 -> 빈들을 등록해서 관리 (싱글톤 레지스트리)

AppConfig.class

```
@Bean
public MemberService memberService() {
    return new MemberServiceImpl(memberRepository());
}

@Bean
public OrderService orderService() {
    return new OrderServiceImpl(
        memberRepository(),
        discountPolicy());
}

@Bean
public MemberRepository memberRepository() {
    return new MemoryMemberRepository();
}

@Bean
public DiscountPolicy discountPolicy() {
    return new RateDiscountPolicy();
}
```

스프링 컨테이너

스프링 빈 저장소

빈 이름	빈 객체
memberService	MemberServiceImpl@x01
orderService	OrderServiceImpl@x02
memberRepository	MemoryMemberRepository@x03
discountPolicy	RateDiscountPolicy@x04

✓ 싱글톤 패턴을 위한 지저분한 코드가 필요하지 않음

✓ DIP, OCP 테스트, **private** 생성자로부터 자유롭게 싱글톤을 사용할 수 있음.

Singleton Container

싱글톤 컨테이너

* 스프링 컨테이너가 적용되어 싱글톤의 역할을 하는 코드

```
@Test
@DisplayName("스프링 컨테이너와 싱글톤")
void springContainer() {
    //AppConfig appconfig = new AppConfig();
    ApplicationContext ac = new AnnotationConfigApplicationContext(AppConfig.class);

    //1. 조회 : 호출할 때마다 객체를 생성?
    MemberService memberService1 = ac.getBean( name: "memberService", MemberService.class);
    //2. 조회 : 호출 할 때마다 객체를 생성?
    MemberService memberService2 = ac.getBean( name: "memberService", MemberService.class);

    System.out.println("memberService1 = " + memberService1);
    System.out.println("memberService2 = " + memberService2);

    //memberService1 != memberService2
    assertThat(memberService1).isSameAs(memberService2);
}
```

- ✓ 알아서 해준다. **ApplicationContext**에서 **Bean**들을 가져와 사용하는 모습.
- ✓ 기본적으로 스프링은 싱글톤이지만 꼭 그렇지 않은 않다. 아주 가끔 특별한 기능을 사용할 때(빈의 생명주기를 **httpResponse** or **session**에 맞춘다거나) 하는 경우에는 싱글톤을 사용하지 않을 수도 있다. 하지만 **99%정도** 싱글톤을 사용한다고 생각

Singleton Container

싱글톤 방식의 **주의점**

: 객체 인스턴스를 딱 하나만 생성해서 공유하는 싱글톤 방식은 여러 클라이언트가 하나의 객체 인스턴스를 공유하기 때문에, 싱글톤 객체는 **stateful**하게 설계하면 안됨

1. **Stateless**하게 설계해야함.
2. 특정 클라이언트에 의존적인 필드가 있으면 안됨.
3. 가급적 읽기만 가능해야 함.
4. 필드 대신 자바에서 공유되지 않는 지역변수, 파라미터, 스레드 로컬 등을 사용해야 함.
5. 스프링 빈의 필드에 공유값을 설정하면 **정말 큰 장애가 발생할 수 있음.**

Singleton Container

싱글톤 방식의 **주의점**

```
public class StatefulService {  
  
    private int price; // 상태를 유지하는 필드  
  
    public void order(String name, int price) {  
        System.out.println(" name = " + name + " price= " + price);  
        this.price = price;  
    }  
  
    public int getPrice() {  
        return price;  
    }  
}
```

```
@Test  
void statefulServiceSingleton() {  
    ApplicationContext ac = new AnnotationConfigApplicationContext(TestConfig.class);  
    StatefulService statefulService1 = ac.getBean(StatefulService.class);  
    StatefulService statefulService2 = ac.getBean(StatefulService.class);  
  
    //ThreadA : A사용자가 10000 원 주문  
    statefulService1.order( name: "userA", price: 10000);  
    //ThreadB : B 사용자가 20000 원 주문  
    statefulService2.order( name: "userB", price: 20000);  
  
    //ThreadA : 사용자A 주문 금액 조회  
    int price = statefulService1.getPrice();  
    System.out.println("price : "+price);  
  
    Assertions.assertThat(statefulService1.getPrice()).isEqualTo(20000);  
}
```

싱글톤 패턴에서 객체는 공유되기 때문에 **StatefulService의 인스턴스는 공유되는 객체임**

: statefulService에서 가격을 의미하는 price값이 상태를 유지하는 필드이기 때문에, 오른쪽 테스트 코드의 실행 결과 userA의 price는 10000으로 입력했음에도 불구하고 B에 간섭을 받아 20000으로 찍힌다.

Singleton Container

@Configuration

: @Configuration은 사실상 싱글톤을 위해 존재한다.

```
@Configuration
public class AppConfig {

    // @Bean memberService -> new MemoryMemberRepository()
    // @Bean orderService -> new MemoryMemberRepository()
    // => 싱글톤이 깨지는거 아니야?? 걱정되면 테스트코드로 돌립니다.
    @Bean
    public MemberService memberService() { return new MemberServiceImpl(memberRepository()); }

    @Bean
    public MemoryMemberRepository memberRepository() { return new MemoryMemberRepository(); }

    @Bean
    public OrderService orderService() { return new OrderServiceImpl(memberRepository(), discountPolicy()); }

    @Bean
    public DiscountPolicy discountPolicy() {
        //return new FixDiscountPolicy();
        return new RateDiscountPolicy();
    }
}
```

- ✓ AppConfig를 살펴보면 memberService를 호출하기 위해서는 memberRepository가 필요하고 orderService에서도 memberRepository가 필요하고...
- ✓ 자바의 관점에서 보면 결과적으로 두개의 MemoryMemberRepository가 생성되며 싱글톤의 원칙이 깨지는 것 처럼 보인다.

Singleton Container

@Configuration

: @Configuration은 사실상 싱글톤을 위해 존재한다.

```
@Configuration
public class AppConfig {

    // @Bean memberService -> new MemoryMemberRepository()
    // @Bean orderService -> new MemoryMemberRepository()
    // => 싱글톤이 깨지는거 아니야?? 걱정되면 테스트코드로 돌립니다.
    @Bean
    public MemberService memberService() { return new MemberServiceImpl(memberRepository()); }

    @Bean
    public MemoryMemberRepository memberRepository() { return new MemoryMemberRepository(); }

    @Bean
    public OrderService orderService() { return new OrderServiceImpl(memberRepository(), discountPolicy()); }

    @Bean
    public DiscountPolicy discountPolicy() {
        //return new FixDiscountPolicy();
        return new RateDiscountPolicy();
    }
}
```

- ✓ AppConfig를 살펴보면 memberService를 호출하기 위해서는 memberRepository가 필요하고 orderService에서도 memberRepository가 필요하고...
- ✓ 자바의 관점에서 보면 결과적으로 두개의 MemoryMemberRepository가 생성되며 싱글톤의 원칙이 깨지는 것 처럼 보인다.

Singleton Container

@Configuration

Test code : 스프링은 싱글톤을 보장한다.

```
ApplicationContext ac = new AnnotationConfigApplicationContext(AppConfig.class);

MemberServiceImpl memberService = ac.getBean( name: "memberService", MemberServiceImpl.class);
OrderServiceImpl orderService = ac.getBean( name: "orderService", OrderServiceImpl.class);

MemberRepository memberRepository = ac.getBean( name: "memberRepository", MemberRepository.class);

MemberRepository memberRepository1= memberService.getMemberRepository();
MemberRepository memberRepository2 = orderService.getMemberRepository();

// 똑같다.
System.out.println("memberService -> memberRepository1 = " + memberRepository1);
System.out.println("orderService -> memberRepository2 = " + memberRepository2);
System.out.println("memberRepository -> " + memberRepository);

Assertions.assertThat(memberService.getMemberRepository()).isSameAs(memberRepository);
Assertions.assertThat(orderService.getMemberRepository()).isSameAs(memberRepository);
```

```
✓ Tests passed: 1 of 1 test - 1 s 45 ms

memberService -> memberRepository1 = hello.core.member.MemoryMemberRepository@5023bb8b
orderService -> memberRepository2 = hello.core.member.MemoryMemberRepository@5023bb8b
memberRepository -> hello.core.member.MemoryMemberRepository@5023bb8b
```

Process finished with exit code 0

✓ AppConfig에서는 5번 호출되었어야 했는데 3번 호출되고 있다.

```
✓ Tests passed: 1 of 1 test - 1 s 45 ms

call AppConfig.memberService
11:25:23.218 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean '
call AppConfig.memberRepository
11:25:23.223 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean '
call AppConfig.orderService
11:25:23.226 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of singleton bean '
```

✓ 같은 객체가 생성된다.

Singleton Container

@Configuration과 바이트 코드 조작

: 스프링 컨테이너는 싱글톤 레지스트리라 빈이 싱글톤이 되도록 해야함. 하지만 자바를 바꿀수는 없기 때문에 @Configuration을 통해서 약간의 조작이 필요함

```
@Test
void configurationDeep() {
    ApplicationContext ac = new AnnotationConfigApplicationContext(AppConfig.class);
    AppConfig bean = ac.getBean(AppConfig.class);

    System.out.println("bean= " + bean.getClass());
}
```

```
bean= class hello.core.AppConfig$$EnhancerBySpringCGLIB$$9031f2e0
```

- ✓ AppConfig도 빈으로 저장
- ✓ 직접 확인해보면 AppConfig@CGLB를 확인.
- ✓ AppConfig를 그대로 저장하는 것이 아니라 바이트 코드를 조작하는 라이브러리를 활용해 내가 만든 AppConfig를 상속받아 다른 임의의 클래스를 만드로 그게 빈으로 등록.

=> AppConfig@CGLB는 @Bean이 붙은 메서드마다 스프링 컨테이너에 존재하는지 확인

Matrix Gradients for Neural Nets

Derivative WRT Weight Matrix

3) Deriving gradients for backprop (weight matrix)

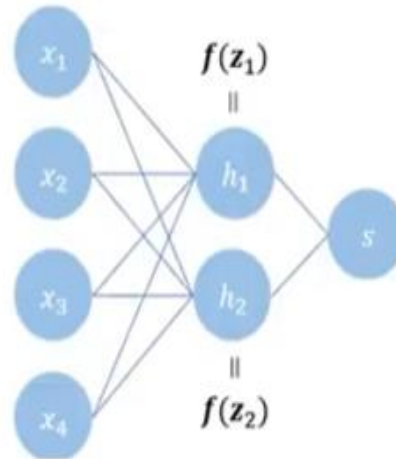
- Deriving gradients for backprop

$$\frac{\partial s}{\partial \mathbf{W}} = \frac{\partial s}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{W}}$$

$$\frac{\partial s}{\partial W_{ij}} = \delta \frac{\partial z}{\partial W_{ij}} = \sum_{k=1}^m \delta_k \frac{\partial z_k}{\partial W_{ij}} = \delta_i x_j$$

$$\frac{\partial s}{\partial W_{ij}} = \underbrace{\delta_i}_{\text{Error signal from above}} \underbrace{x_j}_{\text{Local gradient signal}}$$

$$\frac{\partial s}{\partial \mathbf{W}} = \boldsymbol{\delta} \mathbf{x}^T$$
$$[n \times m] \quad [n \times 1][1 \times m]$$



*결과인 s 에 대해 \mathbf{w} 의 편미분을 구하는 경우

*chain rule을 그대로 적용

*에러 값이 전파되는 부분은 델타로 파악하여 계산.



Component Scan

Component Scan

컴포넌트 스캔과 의존관계 자동 주입

Bean 등록하기

- ✓ @Bean을 사용하여 AppConfig를 만들거나, xml 설정 <bean>을 통해서 설정 정보에 bean들을 직접 등록
- ✓ 만약 Bean이 수백개라면? 너무 귀찮고 누락하기도 쉬움.
- ✓ 스프링은 설정정보 없이도 자동으로 빈을 등록하는 **Component Scan**기능을 제공한다.
- ✓ 의존관계 역시 자동으로 주입하는 **@Autowired** annotation도 사용 가능.

Component Scan

컴포넌트 스캔과 의존관계 자동 주입

@ComponentScan

- ✓ 아래의 예시는 다른 설정 파일인 **AppConfig**를 제외하기 위해 **excludeFilters** 사용
- ✓ 기존의 **AppConfig**와 다르게 **@Bean**으로 등록한 클래스가 없음.
- ✓ **@ComponentScan**을 사용하면 **@Component**가 붙은 클래스들을 전부 빈으로 등록함.
- ✓ **@Configuration** 역시 **@Component**를 포함하고 있음.

```
@Configuration
@ComponentScan(
    excludeFilters = @ComponentScan.Filter(type= FilterType.ANNOTATION,classes = Configuration.class )
)

public class AutoAppConfig {

}
```

+) 의존관계 설정 자체가 없어졌기
때문에 의존성 주입을 클래스 안에서
해결해야함
=> **@Autowired**

Component Scan

탐색 위치와 스캔 대상

@ComponentScan의 탐색 위치

설정된 패키지부터 아래 패키지까지만 쪽 탐색.

- ✓ @ComponentScan(basePackages = "hello.core.member" 과 같은 형식으로 설정 가능(클래스도 지정가능)
- ✓ 디폴트가 가장 중요한데, 지정하지 않으면 본인의 위치부터 시작하여 하위 패키지들을 쪽 탐색함.
- ✓ 따라서, 가장 권장되는 방법은 그냥 패키지 위치를 따로 지정하지 말고 설정 정보 클래스의 위치를 나의 프로젝트 최상단에 두자.
- ✓ SpringBoot의 경우는 아예 @SpringBootApplication이라는 어노테이션이 붙은 메인 함수가 아예 프로젝트 최상단에 자체 제작

그래서 아예 모든 빈이 다 등록되고
@ComponentScan도 필요 없다.



The screenshot shows an IDE with a project structure on the left and a Java class on the right. The project structure includes a 'main' directory with 'generated' and 'java' subdirectories. The 'java' directory contains a 'hello.core' package with sub-packages 'discount', 'member', and 'order'. The 'hello.core' package also contains classes 'AppConfig', 'AutoAppConfig', 'CoreApplication', 'MemberApp', and 'OrderApp'. The 'CoreApplication' class is selected, and its code is displayed on the right. The code is as follows:

```
1 package hello.core;
2
3 import ...
4
5
6 @SpringBootApplication
7 public class CoreApplication {
8
9     public static void main(String[] args) { SpringApplication.run(CoreApplication.class, args); }
10
11 }
12
13
14
```

Component Scan

탐색 위치와 스캔 대상

@ComponentScan의 탐색 대상

1. @Component
2. @Controller : 스프링 MVC 컨트롤러로 인식하게 함. @GetMapping, @PostMapping..
3. @Service : 스프링 비즈니스 로직에서 주로 사용. 별 뜻은 없음. 알아보기 좋으라고.
4. @Repository : 스프링 데이터 접근 계층으로 인식을 하면서 데이터 계층의 예외를 스프링 계층의 추상적 예외로 바꿔줌.
5. @Configuration : 스프링 설정 정보에서 사용.

Component Scan

중복 등록과 충돌

@ComponentScan에서 같은 빈 이름을 등록하면 어떻게 될까?

1. 자동 빈 등록 vs 자동 빈 등록 : **Conflicting Bean Definition Error**가 나기 때문에 쉽게 확인 가능함.
2. 수동 빈 등록 vs 자동 빈 등록 : **수동 등록된 빈이 우선권을 가짐**. 수동 빈이 자동 빈을 오버라이딩 해버림
 - ✓ 의도한 결과이면 괜찮으나 현실적으로 사실상 설정이 꼬여서 망하는 경우가 매우 많다.
 - ✓ 그러한 경우의 버그는 매우매우 잡기 어렵다. 애매한 상황을 만들지 말자.
 - ✓ 스프링 부트로 실행을 하면 아예 에러를 띄워준다.
 - ✓ **Application.properties**에서 **spring.main.allow-bean-definition-overriding = True** (False가 디폴트)로 바꿔 에러를 안 나게 할수도 있다.
 - ✓ 근데 그냥 하지 말자.



@Autowired : 의존관계 자동 주입

다양한 의존관계 주입 방법

1. 생성자 주입

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

    @Autowired
    public OrderServiceImpl(MemberRepository memberRepository, DiscountPolicy discountPolicy) {
        this.memberRepository = memberRepository;
        this.discountPolicy = discountPolicy;
    }
}
```

- 1) 생성자를 호출 시점에 딱 1번만 호출.
- 2) 불변, 필수 의존관계에 주로 사용 (사실 주로 사용)
- 3) 컴포넌트들을 찾아서 빈에 등록할 때 @Autowired 있는 친구들 의존성 주입.
- 4) 생성자를 통해서 의존관계가 주입되기 때문에 외부 수정 불가능
- 5) 이 방식에서는 setter 설정X

6) 필수 의존 관계 : **private final** 7) 생성자가 딱 한 개만 있으면 @Autowired 생략 가능

@Autowired

다양한 의존관계 주입 방법

2. 수정자 주입

- 1) 보통 필드를 setXXX로 주로 수정하는데, 이 셋터에다가 @Autowired해서 사용 가능.
- 2) 스프링 빈 등록 사이클 : 빈을 싹 등록하고 @Autowired를 주입함.
- 3) 생성자 주입은 빈을 등록하면서 의존관계 주입도 바로 일어남. 뭘 만들려면 넣어줘야 하니까. 자바언어 자체를 변경할 수는 없음
- 4) 수정자 주입은 두번째 단계에서 일어남

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

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

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


@Autowired

다양한 의존관계 주입 방법

3. 필드 주입

- 1) 그냥 필드에다가 @Autowired 때려박기
- 2) Private 변수에도 사용 가능
- 3) 코드가 매우 간결
- 4) 하지만, "Field injection is not recommended": 외부에서 변경 불가능하고 테스트하기 매우 불편
- 5) 필드 주입을 하려면 어차피 setter를 만들어줘야 하는데 그럴거면 수정자 주입하지.
- 6) **쓰지 마세요.** 단, Test code 작성 시에는 사용 권장

```
@Autowired
private MemberRepository memberRepository;

@Autowired
private DiscountPolicy discountPolicy;
```

옵션처리

: 주입할 빈이 없어도 동작해야 할 때가 있을 수도 있음. 스프링 빈을 옵셔널하게 해놓고 등록을 안해도 기본 로직으로 동작하거나 혹은 동작을 안하거나 해야 할 수도 있음

- 1) `@Autowired(required=false)`: 자동 주입할 대상이 없으면 수정자 메서드 자체가 호출이 안됨
- 2) [`Org.springframework.lang.Nullable`](#): 자동 주입할 대상이 없으면 `null`이 입력됨
- 3) `Optional<>`: 값이 있을 수도 있고, 없을 수도 있는 것을 감싼 것. Java8. 자동 주입할 대상이 없으면 `Optional.empty` 입력

```
static class TestBean {  
  
    @Autowired(required=false)  
    public void setNoBean1(Member noBean1) {  
        System.out.println("noBean1 = " + noBean1);  
    }  
  
    @Autowired  
    public void setNoBean2(@Nullable Member noBean2) {  
        System.out.println("noBean2 = " + noBean2);  
    }  
  
    @Autowired  
    public void setNoBean3(Optional<Member> noBean3) {  
        System.out.println("noBean3 = " + noBean3);  
    }  
}
```

```
14:10:32.194 [main] DEBUG org.springframework  
noBean2 = null  
noBean3 = Optional.empty
```

생성자 주입 선택

: 과거에는 필드, 수정자 주입을 많이 썼지만 최근 스프링 뿐만 아닌 DI 프레임 워크 대부분이 **생성자 주입**을 권장.

Why 생성자 주입? - 불변성

- 1) 대부분의 의존관계 주입은 한번 일어나면 종료 시점까지 **의존관계를 변경할 일이 없고 오히려 변하면 안됨.**
- 2) 근데 수정자 주입을 하기 위해 setter를 public으로 둔다면 누군가는 변경할 가능성이 있음(실수)

Why 생성자 주입? - 누락의 가능성

- 1) 테스트코드 작성에 있어서 순수 자바를 사용하는데 만약 수정자에 의존한다면 **컴파일 에러가 뜨지 않기 때문에 혼란의 가능성**
- 2) 생성자를 쓰면 final 키워드를 써야 함. 만약 테스트 코드에서 생성자에 전달할 값을 누락하면 컴파일 에러가 뜨기 때문에 쉽게 판별

```
@Test
void createOrder() {
    MemoryMemberRepository memoryMemberRepository = new MemoryMemberRepository();
    memoryMemberRepository.save(new Member(id: 1L, name: "NAME", Grade.VIP));

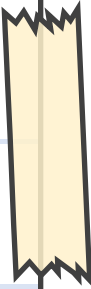
    OrderService orderService = new OrderServiceImpl();
    /*
    ..
    */
}
```

순수한 자바의 특징을 가장 잘 살리는 방법
기본으로 생성자 주입을 사용하고, 옅서널 하게 수정자 주입, 필드 주입은 지양

@Autowired

롬복과 최신 트렌드

:그래서 생성자 주입을 할려면 막상 불편함. 코드가 많아짐.



```
@Component
@RequiredArgsConstructor //final이 붙은 애들을 가지고 생성자를 만들어준다.
public class OrderServiceImpl implements OrderService {
    private final MemberRepository memberRepository;
    private final DiscountPolicy discountPolicy;

    /*
    public OrderServiceImpl(MemberRepository memberRepository, DiscountPolicy discountPolicy) {
        this.memberRepository = memberRepository;
        this.discountPolicy = discountPolicy;
    }
    */
}
```

- ✓ @Getter, @Setter, @RequiredConstructor
등등 편리한 annotation제공
- ✓ 생성자가 한 개라면 @Autowired 생략 +
Lombok => 매우 간결한 코드

조회할 빈이 **두개 이상**일 때의 문제

:@Autowired는 사실상 '타입'으로 탐색 like `ac.getBean(DiscountPolicy.class)` 만약 같은 타입의 여러 개의 빈이 있다면? => **NoUniqueBeanDefinition Error**

1. @Autowired

- ✓ 처음에는 타입 매칭을 시도하고 만약 복수의 타입의 빈을 발견하면 필드 이름이나 파라미터 이름으로 빈 이름을 추가 매핑
- ✓ 필드 이름, 파라미터 이름을 살펴보고 이름이 똑같으면 그걸로 가져간다.

2. @Qualifier 사용

- ✓ 주입시 "추가 구분자" 를 붙여주는 방법. 이름을 바꾸는건 아님.
- ✓ @Autowired와 비슷하게 만약 **Qualifier**를 찾지 못한다면 빈에서 추가로 탐색. 주입 받을 코드에도 @Qualifier를 붙여줘야

3. @Primary 사용

- ✓ 가장 쉬운 사용법
- ✓ @Autowired가 여러 빈을 찾으면 거기서 @Primary가 붙어 있는 빈이 우선권을 가짐. Like db 선택

Thank You