7. 의존관계 자동 주입

#1.인강/2.핵심원리 - 기본편/강의#

- /다양한 의존관계 주입 방법
- /옵션 처리
- /생성자 주입을 선택해라!
- /롬복과 최신 트랜드
- / 조회 빈이 2개 이상 - 문제
- /@Autowired 필드 명, @Qualifier, @Primary
- /애노테이션 직접 만들기
- /조회한 빈이 모두 필요할 때, List, Map
- /자동, 수동의 올바른 실무 운영 기준

다양한 의존관계 주입 방법

의존관계 주입은 크게 4가지 방법이 있다.

- 생성자 주입
- 수정자 주입(setter 주입)
- 필드 주입
- 일반 메서드 주입

생성자 주입

- 이름 그대로 생성자를 통해서 의존 관계를 주입 받는 방법이다.
- 지금까지 우리가 진행했던 방법이 바로 생성자 주입이다.
- 특징
  - 생성자 호출시점에 딱 1번만 호출되는 것이 보장된다.
  - **불변, 필수** 의존관계에 사용

```
@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;
    }
}
```

중요! 생성자가 딱 1개만 있으면 @Autowired를 생략해도 자동 주입 된다. 물론 스프링 빈에만 해당한다.

```
@Component
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;
    }
}
```

수정자 주입(setter 주입)

- setter라 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 memberRepository) {
        this.memberRepository = memberRepository;
    }
```

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

}
```

> 참고: `@Autowired`의 기본 동작은 주입할 대상이 없으면 오류가 발생한다. 주입할 대상이 없어도 동작하게 하려면 `@Autowired(required = false)`로 지정하면 된다.

> 참고: 자바빈 프로퍼티, 자바에서는 과거부터 필드의 값을 직접 변경하지 않고, setXxx, getXxx 라는 메서드를 통해서 값을 읽거나 수정하는 규칙을 만들었는데, 그것이 자바빈 프로퍼티 규약이다. 더 자세한 내용이 궁금하면 자바빈 프로퍼티로 검색해보자.

자바빈 프로퍼티 규약 예시

```
class Data {
  private int age;
  public void setAge(int age) {
    this.age = age;
  }

  public int getAge() {
    return age;
  }
}
```

필드 주입

- 이름 그대로 필드에 바로 주입하는 방법이다.
- 특징
  - 코드가 간결해서 많은 개발자들을 유혹하지만 외부에서 변경이 불가능해서 테스트 하기 힘들다는 치명적인 단점이 있다.
  - DI 프레임워크가 없으면 아무것도 할 수 없다.
  - 사용하지 말자!
    - 애플리케이션의 실제 코드와 관계 없는 테스트 코드
    - 스프링 설정을 목적으로 하는 @Configuration 같은 곳에서만 특별한 용도로 사용

```
@Component
public class OrderServiceImpl implements OrderService {

    @Autowired
    private MemberRepository memberRepository;
    @Autowired
    private DiscountPolicy discountPolicy;

}
```

> **참고:** 순수한 자바 테스트 코드에는 당연히 @Autowired가 동작하지 않는다. `@SpringBootTest` 처럼 스프링 컨테이너를 테스트에 통합한 경우에만 가능하다.

> **참고:** 다음 코드와 같이 `@Bean` 에서 파라미터에 의존관계는 자동 주입된다. 수동 등록시 자동 등록된 빈의 의존관계가 필요할 때 문제를 해결할 수 있다.

```
@Bean
OrderService orderService(MemberRepository memberRepoisitory, DiscountPolicy
discountPolicy) {
    return new OrderServiceImpl(memberRepository, discountPolicy);
}
```

일반 메서드 주입

- 일반 메서드를 통해서 주입 받을 수 있다.
- 특징
  - 한번에 여러 필드를 주입 받을 수 있다.
  - 일반적으로 잘 사용하지 않는다.

```
@Component
public class OrderServiceImpl implements OrderService {

    private MemberRepository memberRepository;
    private DiscountPolicy discountPolicy;

    @Autowired
    public void init(MemberRepository memberRepository, DiscountPolicy
```

```
    discountPolicy) {
            this.memberRepository = memberRepository;
            this.discountPolicy = discountPolicy;
        }

    }
```

> 참고: 어쩌면 당연한 이야기이지만 의존관계 자동 주입은 스프링 컨테이너가 관리하는 스프링 빈이어야 동작한다. 스프링 빈이 아닌 `Member` 같은 클래스에서 `@Autowired` 코드를 적용해도 아무 기능도 동작하지 않는다.

옵션 처리

주입할 스프링 빈이 없어도 동작해야 할 때가 있다.
그런데 `@Autowired`만 사용하면 `required` 옵션의 기본값이 `true`로 되어 있어서 자동 주입 대상이 없으면 오류가 발생한다.

자동 주입 대상을 옵션으로 처리하는 방법은 다음과 같다.

- `@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 입력된다.

예제로 확인해보자.

```
//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);
    }
```

- **Member는 스프링 빈이 아니다.**
- `setNoBean1()`은 `@Autowired(required=false)`이므로 호출 자체가 안된다.

출력결과

```
setNoBean2 = null
setNoBean3 = Optional.empty
```

> 참고: @Nullable, Optional은 스프링 전반에 걸쳐서 지원된다. 예를 들어서 생성자 자동 주입에서 특정 필드에만 사용해도 된다.

생성자 주입을 선택해라!

과거에는 수정자 주입과 필드 주입을 많이 사용했지만, 최근에는 스프링을 포함한 DI 프레임워크 대부분이 생성자 주입을 권장한다. 그 이유는 다음과 같다.

불변

- 대부분의 의존관계 주입은 한번 일어나면 애플리케이션 종료시점까지 의존관계를 변경할 일이 없다. 오히려 대부분의 의존관계는 애플리케이션 종료 전까지 변하면 안된다.(불변해야 한다.)
- 수정자 주입을 사용하면, setXxx 메서드를 public으로 열어두어야 한다.
- 누군가 실수로 변경할 수 도 있고, 변경하면 안되는 메서드를 열어두는 것은 좋은 설계 방법이 아니다.
- 생성자 주입은 객체를 생성할 때 딱 1번만 호출되므로 이후에 호출되는 일이 없다. 따라서 불변하게 설계할 수 있다.

누락

프레임워크 없이 순수한 자바 코드를 단위 테스트 하는 경우에
다음과 같이 수정자 의존관계인 경우

```
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`가 프레임워크 안에서 동작할 때는 의존관계가 없으면 오류가 발생하지만, 지금은 프레임워크 없이 순수한 자바 코드로만 단위 테스트를 수행하고 있다.

이렇게 테스트를 수행하면 실행은 된다.

```
@Test
void createOrder() {
    OrderServiceImpl orderService = new OrderServiceImpl();
    orderService.createOrder(1L, "itemA", 10000);
}
```

그런데 막상 실행 결과는 NPE(Null Point Exception)이 발생하는데, memberRepository, discountPolicy 모두 의존관계 주입이 누락되었기 때문이다.

생성자 주입을 사용하면 다음처럼 주입 데이터를 누락 했을 때 **컴파일 오류**가 발생한다.
그리고 IDE에서 바로 어떤 값을 필수로 주입해야 하는지 알 수 있다.

```
@Test
void createOrder() {
  OrderServiceImpl orderService = new OrderServiceImpl();
  orderService.createOrder(1L, "itemA", 10000);
}
```

final 키워드

생성자 주입을 사용하면 필드에 `final` 키워드를 사용할 수 있다. 그래서 생성자에서 혹시라도 값이 설정되지 않는 오류를 컴파일 시점에 막아준다. 다음 코드를 보자.

```
@Component
public class OrderServiceImpl implements OrderService {

    private final MemberRepository memberRepository;
```

```
    private final DiscountPolicy discountPolicy;

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

- 잘 보면 필수 필드인 `discountPolicy`에 값을 설정해야 하는데, 이 부분이 누락되었다. 자바는 컴파일 시점에 다음 오류를 발생시킨다.
- `java: variable discountPolicy might not have been initialized`
- 기억하자! **컴파일 오류는 세상에서 가장 빠르고, 좋은 오류다!**

> **참고:** 수정자 주입을 포함한 나머지 주입 방식은 모두 생성자 이후에 호출되므로, 필드에 `final` 키워드를 사용할 수 없다. 오직 생성자 주입 방식만 `final` 키워드를 사용할 수 있다.

정리

- 생성자 주입 방식을 선택하는 이유는 여러가지가 있지만, 프레임워크에 의존하지 않고, 순수한 자바 언어의 특징을 잘 살리는 방법이기도 하다.
- 기본으로 생성자 주입을 사용하고, 필수 값이 아닌 경우에는 수정자 주입 방식을 옵션으로 부여하면 된다. 생성자 주입과 수정자 주입을 동시에 사용할 수 있다.
- 항상 생성자 주입을 선택해라! 그리고 가끔 옵션이 필요하면 수정자 주입을 선택해라. 필드 주입은 사용하지 않는 게 좋다.

롬복과 최신 트랜드

막상 개발을 해보면, 대부분이 다 불변이고, 그래서 다음과 같이 필드에 final 키워드를 사용하게 된다.
그런데 생성자도 만들어야 하고, 주입 받은 값을 대입하는 코드도 만들어야 하고...
필드 주입처럼 좀 편리하게 사용하는 방법은 없을까?
역시 개발자는 귀찮은 것은 못 참는다!
다음 기본 코드를 최적화해보자.

기본 코드

```
@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;
    }
}
```

생성자가 딱 1개만 있으면 `@Autowired`를 생략할 수 있다.

```
@Component
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;
    }
}
```

- 이제 롬복을 적용해보자. 롬복 라이브러리 적용 방법은 아래에 적어두었다.
- 롬복 라이브러리가 제공하는 `@RequiredArgsConstructor` 기능을 사용하면 final이 붙은 필드를 모아서 생성자를 자동으로 만들어준다. (다음 코드에는 보이지 않지만 실제 호출 가능하다.)
- 최종 결과는 다음과 같다! 정말 간결하지 않은가!

최종 결과 코드

```
@Component
@RequiredArgsConstructor
public class OrderServiceImpl implements OrderService {

    private final MemberRepository memberRepository;
    private final DiscountPolicy discountPolicy;
```

```
}
```

이 최종결과 코드와 이전의 코드는 완전히 동일하다. 롬복이 자바의 애노테이션 프로세서라는 기능을 이용해서 컴파일 시점에 생성자 코드를 자동으로 생성해준다. 실제 `class`를 열어보면 다음 코드가 추가되어 있는 것을 확인할 수 있다.

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

정리

최근에는 생성자를 딱 1개 두고, `@Autowired`를 생략하는 방법을 주로 사용한다. 여기에 Lombok 라이브러리의 `@RequiredArgsConstructor`함께 사용하면 기능은 다 제공하면서, 코드는 깔끔하게 사용할 수 있다.

롬복 라이브러리 적용 방법

`build.gradle`에 라이브러리 및 환경 추가

```
plugins {
    id 'org.springframework.boot' version '2.3.2.RELEASE'
    id 'io.spring.dependency-management' version '1.0.9.RELEASE'
    id 'java'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

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

```
repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter'

    //lombok 라이브러리 추가 시작
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'

    testCompileOnly 'org.projectlombok:lombok'
    testAnnotationProcessor 'org.projectlombok:lombok'
    //lombok 라이브러리 추가 끝

    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
    }
}

test {
    useJUnitPlatform()
}
```

1. Preferences(윈도우 File ➜ Settings) ➜ plugin ➜ lombok 검색 설치 실행 (재시작)
2. Preferences ➜ Annotation Processors 검색 ➜ Enable annotation processing 체크 (재시작)
3. 임의의 테스트 클래스를 만들고 @Getter, @Setter 확인

조회 빈이 2개 이상 - 문제

`@Autowired`는 타입(Type)으로 조회한다.

```
@Autowired
private DiscountPolicy discountPolicy
```

타입으로 조회하기 때문에, 마치 다음 코드와 유사하게 동작한다. (실제로는 더 많은 기능을 제공한다.)

`ac.getBean(DiscountPolicy.class)`

스프링 빈 조회에서 학습했듯이 타입으로 조회하면 선택된 빈이 2개 이상일 때 문제가 발생한다.
`DiscountPolicy`의 하위 타입인 `FixDiscountPolicy`, `RateDiscountPolicy`둘다 스프링 빈으로 선언해 보자.

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

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

그리고 이렇게 의존관계 자동 주입을 실행하면

```
@Autowired
private DiscountPolicy discountPolicy
```

`NoUniqueBeanDefinitionException` 오류가 발생한다.

```
NoUniqueBeanDefinitionException: No qualifying bean of type
'hello.core.discount.DiscountPolicy' available: expected single matching bean
but found 2: fixDiscountPolicy,rateDiscountPolicy
```

오류메시지가 친절하게도 하나의 빈을 기대했는데 `fixDiscountPolicy`, `rateDiscountPolicy` 2개가 발견되었다고 알려준다.

이때 하위 타입으로 지정할 수 도 있지만, 하위 타입으로 지정하는 것은 DIP를 위배하고 유연성이 떨어진다. 그리고 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 ➜ @Qualifier끼리 매칭 ➜ 빈 이름 매칭
- @Primary 사용

@Autowired 필드 명 매칭

`@Autowired`는 타입 매칭을 시도하고, 이때 여러 빈이 있으면 필드 이름, 파라미터 이름으로 빈 이름을 추가 매칭한다.

기존 코드

```
@Autowired
private DiscountPolicy discountPolicy
```

필드 명을 빈 이름으로 변경

```
@Autowired
private DiscountPolicy rateDiscountPolicy
```

필드 명이 `rateDiscountPolicy`이므로 정상 주입된다.
필드 명 매칭은 먼저 타입 매칭을 시도 하고 그 결과에 여러 빈이 있을 때 추가로 동작하는 기능이다.

@Autowired 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. 주입시 추가적인 방법을 제공하는 것이지 빈 이름을 변경하는 것은 아니다.

빈 등록시 @Qualifier를 붙여 준다.

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

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

주입시에 @Qualifier를 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("mainDiscountPolicy")
DiscountPolicy discountPolicy) {
    this.discountPolicy = discountPolicy;
}
```

`@Qualifier`로 주입할 때 `@Qualifier("mainDiscountPolicy")`를 못찾으면 어떻게 될까? 그러면 mainDiscountPolicy라는 이름의 스프링 빈을 추가로 찾는다. 하지만 경험상 `@Qualifier`는 `@Qualifier`를 찾는 용도로만 사용하는게 명확하고 좋다.

다음과 같이 직접 빈 등록시에도 @Qualifier를 동일하게 사용할 수 있다.

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

@Qualifier 정리

- 1. @Qualifier끼리 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.

`rateDiscountPolicy`가 우선권을 가지도록 하자.

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

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

사용코드

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

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

코드를 실행해보면 문제 없이 `@Primary`가 잘 동작하는 것을 확인할 수 있다.

여기까지 보면 `@Primary`와 `@Qualifier` 중에 어떤 것을 사용하면 좋을지 고민이 될 것이다. `@Qualifier`의 단점은 주입 받을 때 다음과 같이 모든 코드에 `@Qualifier`를 붙여주어야 한다는 점이다.

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

반면에 `@Primary`를 사용하면 이렇게 `@Qualifier`를 붙일 필요가 없다.

@Primary, @Qualifier 활용

코드에서 자주 사용하는 메인 데이터베이스의 커넥션을 획득하는 스프링 빈이 있고, 코드에서 특별한 기능으로 가끔 사

용하는 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.

```
package hello.core.annotataion;

import org.springframework.beans.factory.annotation.Qualifier;

import java.lang.annotation.*;

@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER,
ElementType.TYPE, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Qualifier("mainDiscountPolicy")
public @interface MainDiscountPolicy {
}
```

```
@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 DiscountPolicy
    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.

코드로 바로 설명하겠다.

```
package hello.core.autowired;

import hello.core.AutoAppConfig;
import hello.core.discount.DiscountPolicy;
import hello.core.member.Grade;
import hello.core.member.Member;
import org.junit.jupiter.api.Test;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

import java.util.List;
import java.util.Map;
```

```java
import static org.assertj.core.api.Assertions.assertThat;

public class AllBeanTest {

    @Test
    void findAllBean() {
        ApplicationContext ac = new
AnnotationConfigApplicationContext(AutoAppConfig.class, DiscountService.class);
        DiscountService discountService = ac.getBean(DiscountService.class);
        Member member = new Member(1L, "userA", Grade.VIP);
        int discountPrice = discountService.discount(member, 10000,
"fixDiscountPolicy");

        assertThat(discountService).isInstanceOf(DiscountService.class);
        assertThat(discountPrice).isEqualTo(1000);
    }

    static class DiscountService {

        private final Map<String, DiscountPolicy> policyMap;
        private final List<DiscountPolicy> policies;

        public DiscountService(Map<String, DiscountPolicy> policyMap,
List<DiscountPolicy> policies) {
            this.policyMap = policyMap;
            this.policies = policies;
            System.out.println("policyMap = " + policyMap);
            System.out.println("policies = " + policies);
        }

        public int discount(Member member, int price, String discountCode) {

            DiscountPolicy discountPolicy = policyMap.get(discountCode);

            System.out.println("discountCode = " + discountCode);
            System.out.println("discountPolicy = " + discountPolicy);

            return discountPolicy.discount(member, price);
        }
    }
}
```

로직 분석

- DiscountService는 Map으로 모든 `DiscountPolicy`를 주입받는다. 이때 `fixDiscountPolicy`, `rateDiscountPolicy`가 주입된다.
- `discount ()` 메서드는 discountCode로 "fixDiscountPolicy"가 넘어오면 map에서 `fixDiscountPolicy` 스프링 빈을 찾아서 실행한다. 물론 “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.

참고 - 스프링 컨테이너를 생성하면서 스프링 빈 등록하기

스프링 컨테이너는 생성자에 클래스 정보를 받는다. 여기에 클래스 정보를 넘기면 해당 클래스가 스프링 빈으로 자동 등록된다.

```
new AnnotationConfigApplicationContext(AutoAppConfig.class,DiscountService.class);
```

이 코드는 2가지로 나누어 이해할 수 있다.

- `new AnnotationConfigApplicationContext()`를 통해 스프링 컨테이너를 생성한다.
- `AutoAppConfig.class`, `DiscountService.class`를 파라미터로 넘기면서 해당 클래스를 자동으로 스프링 빈으로 등록한다.

정리하면 스프링 컨테이너를 생성하면서, 해당 컨테이너에 동시에 `AutoAppConfig`, `DiscountService`를 스프링 빈으로 자동 등록한다.

자동, 수동의 올바른 실무 운영 기준

편리한 자동 기능을 기본으로 사용하자

그러면 어떤 경우에 컴포넌트 스캔과 자동 주입을 사용하고, 어떤 경우에 설정 정보를 통해서 수동으로 빈을 등록하고, 의존관계도 수동으로 주입해야 할까?

결론부터 이야기하면, 스프링이 나오고 시간이 갈 수록 점점 자동을 선호하는 추세다. 스프링은 `@Component` 뿐만 아니라 `@Controller`, `@Service`, `@Repository` 처럼 계층에 맞추어 일반적인 애플리케이션 로직을 자동으로 스캔할 수 있도록 지원한다. 거기에 더해서 최근 스프링 부트는 컴포넌트 스캔을 기본으로 사용하고, 스프링 부트의 다양

한 스프링 빈들도 조건이 맞으면 자동으로 등록하도록 설계했다.

설정 정보를 기반으로 애플리케이션을 구성하는 부분과 실제 동작하는 부분을 명확하게 나누는 것이 이상적이지만, 개발자 입장에서 스프링 빈을 하나 등록할 때 `@Component`만 넣어주면 끝나는 일을 `@Configuration` 설정 정보에 가서 `@Bean`을 적고, 객체를 생성하고, 주입할 대상을 일일이 적어주는 과정은 상당히 번거롭다.
또 관리할 빈이 많아서 설정 정보가 커지면 설정 정보를 관리하는 것 자체가 부담이 된다.
그리고 결정적으로 자동 빈 등록을 사용해도 OCP, DIP를 지킬 수 있다.

그러면 수동 빈 등록은 언제 사용하면 좋을까?

애플리케이션은 크게 업무 로직과 기술 지원 로직으로 나눌 수 있다.

- **업무 로직 빈:** 웹을 지원하는 컨트롤러, 핵심 비즈니스 로직이 있는 서비스, 데이터 계층의 로직을 처리하는 리포지토리등이 모두 업무 로직이다. 보통 비즈니스 요구사항을 개발할 때 추가되거나 변경된다.
- **기술 지원 빈:** 기술적인 문제나 공통 관심사(AOP)를 처리할 때 주로 사용된다. 데이터베이스 연결이나, 공통 로그 처리 처럼 업무 로직을 지원하기 위한 하부 기술이나 공통 기술들이다.

- 업무 로직은 숫자도 매우 많고, 한번 개발해야 하면 컨트롤러, 서비스, 리포지토리 처럼 어느정도 유사한 패턴이 있다. 이런 경우 자동 기능을 적극 사용하는 것이 좋다. 보통 문제가 발생해도 어떤 곳에서 문제가 발생했는지 명확하게 파악하기 쉽다.
- 기술 지원 로직은 업무 로직과 비교해서 그 수가 매우 적고, 보통 애플리케이션 전반에 걸쳐서 광범위하게 영향을 미친다. 그리고 업무 로직은 문제가 발생했을 때 어디가 문제인지 명확하게 잘 드러나지만, 기술 지원 로직은 적용이 잘 되고 있는지 아닌지 조차 파악하기 어려운 경우가 많다. 그래서 이런 기술 지원 로직들은 가급적 수동 빈 등록을 사용해서 명확하게 드러내는 것이 좋다.

애플리케이션에 광범위하게 영향을 미치는 기술 지원 객체는 수동 빈으로 등록해서 딱! 설정 정보에 바로 나타나게 하는 것이 유지보수 하기 좋다.

비즈니스 로직 중에서 다형성을 적극 활용할 때
의존관계 자동 주입 - 조회한 빈이 모두 필요할 때, List, Map을 다시 보자.
`DiscountService`가 의존관계 자동 주입으로 `Map<String, DiscountPolicy>`에 주입을 받는 상황을 생각해보자. 여기에 어떤 빈들이 주입될 지, 각 빈들의 이름은 무엇일지 코드만 보고 한번에 쉽게 파악할 수 있을까? 내가 개발했으니 크게 관계가 없지만, 만약 이 코드를 다른 개발자가 개발해서 나에게 준 것이라면 어떨까?
자동 등록을 사용하고 있기 때문에 파악하려면 여러 코드를 찾아봐야 한다.

이런 경우 수동 빈으로 등록하거나 또는 자동으로하면 **특정 패키지에 같이 묶어**두는게 좋다! 핵심은 딱 보고 이해가 되어야 한다!

이 부분을 별도의 설정 정보로 만들고 수동으로 등록하면 다음과 같다.

```
@Configuration
public class DiscountPolicyConfig {

    @Bean
    public DiscountPolicy rateDiscountPolicy() {
        return new RateDiscountPolicy();
    }
    @Bean
    public DiscountPolicy fixDiscountPolicy() {
        return new FixDiscountPolicy();
    }
}
```

이 설정 정보만 봐도 한눈에 빈의 이름은 물론이고, 어떤 빈들이 주입될지 파악할 수 있다. 그래도 빈 자동 등록을 사용하고 싶으면 파악하기 좋게 `DiscountPolicy`의 구현 빈들만 따로 모아서 특정 패키지에 모아두자.

참고로 **스프링과 스프링 부트가 자동으로 등록하는 수 많은 빈들은 예외**다. 이런 부분들은 스프링 자체를 잘 이해하고 스프링의 의도대로 잘 사용하는게 중요하다. 스프링 부트의 경우 `DataSource`같은 데이터베이스 연결에 사용하는 기술 지원 로직까지 내부에서 자동으로 등록하는데, 이런 부분은 메뉴얼을 잘 참고해서 스프링 부트가 의도한 대로 편리하게 사용하면 된다. 반면에 **스프링 부트가 아니라 내가 직접 기술 지원 객체를 스프링 빈으로 등록한다면 수동으로 등록해서 명확하게 드러내는 것이 좋다.**

정리

편리한 자동 기능을 기본으로 사용하자

직접 등록하는 기술 지원 객체는 수동 등록

다형성을 적극 활용하는 비즈니스 로직은 수동 등록을 고민해보자