3️⃣

Section3. 스프링 핵심 원리 이해2 - 객체 지향 원리 적용

새로운 할인 정책 개발

고정 금액 할인이 아닌 정률 할인으로 변경(금액의 10%)

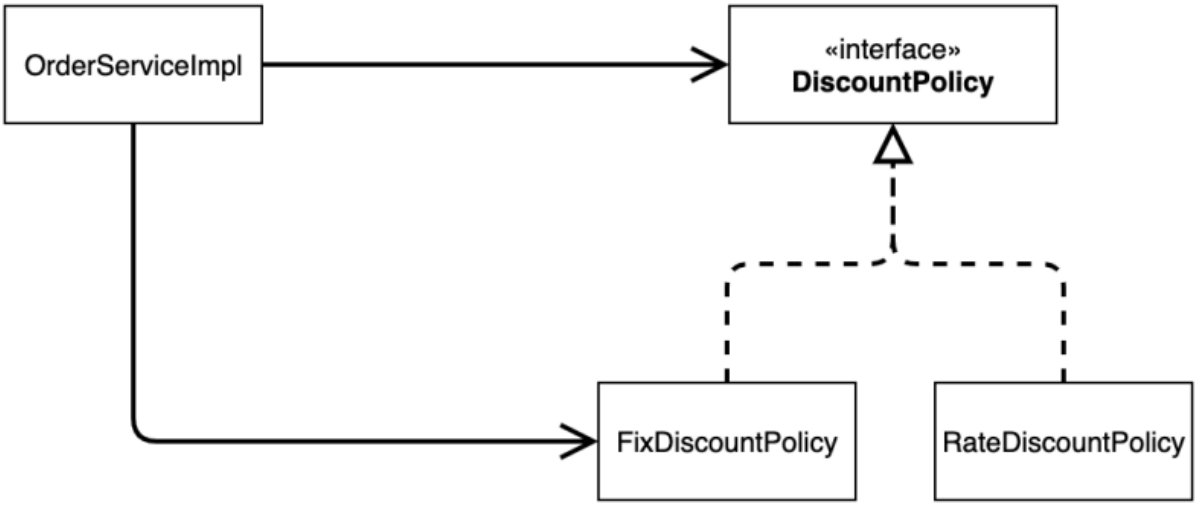
> 💡 실습 코드 작성

- `RateDiscountPolicy()` 구현

새로운 할인 정책 적용과 문제점

- 다음 코드와 같이 `OrderServiceImpl` 은 인터페이스인 `DiscountPolicy` 에 의존하고, 구현체 `RateDiscountPolicy` 도 의존하고 있다. ← **DIP 위반**
- 정률 할인 정책으로 변경하는 경우 `OrderServiceImpl` 코드의 변경 ← **OCP 위반**

```
private final DiscountPolicy discountPolicy = new RateDiscoun
//private final DiscountPolicy discountPolicy = new FixDiscou
```



- 다음과 같이 코드를 변경하면 두 문제를 해결 가능하지만 `NPE` 발생

```
private final DiscountPolicy discountPolicy
```

해결방안: 누군가 `OrderServiceImpl` 에 `DiscountPolicy` 의 구현 객체를 대신 생성하고 주입해주어야 한다. → **AppConfig 의 등장**

관심사의 분리

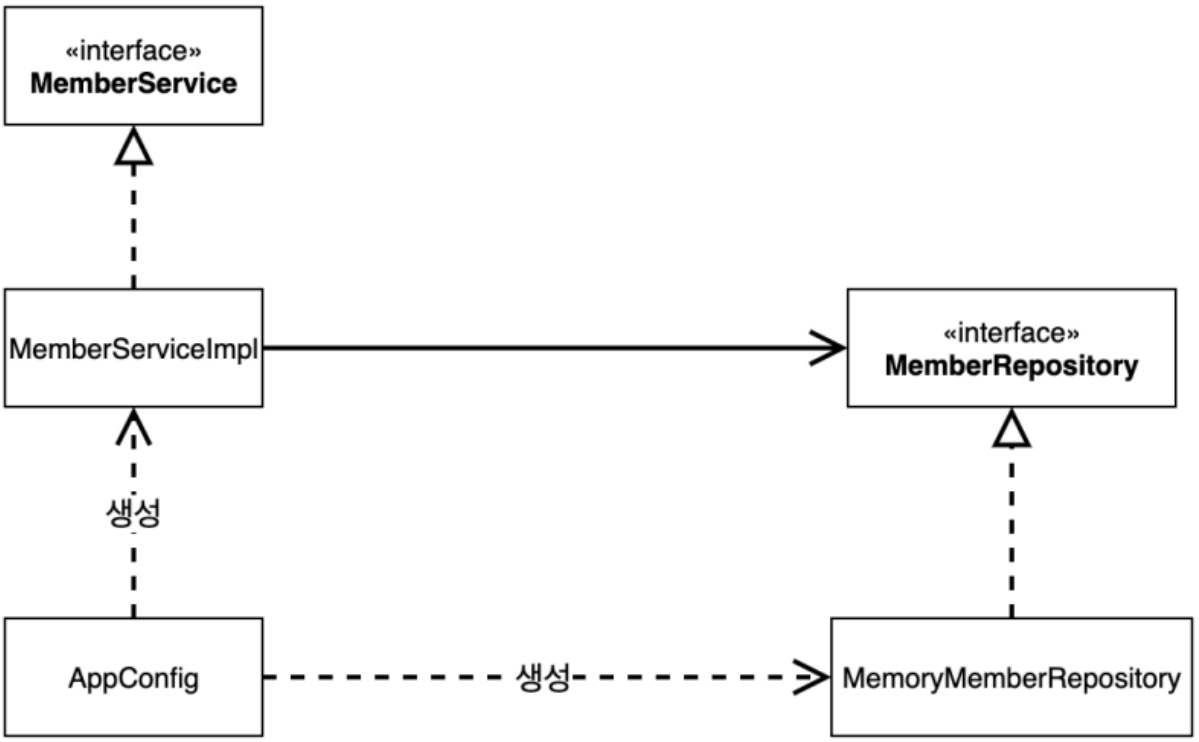
- AppConfig
  - 애플리케이션의 전체 동작 방식을 구성하고, **구현 객체를 생성**, **연결**하는 책임을 가지는 별도의 설정 클래스

> 💡 실습 코드 작성

```
package hello.core;

import hello.core.discount.FixDiscountPolicy;
import hello.core.member.MemberService;
import hello.core.member.MemberServiceImpl;
import hello.core.member.MemoryMemberRepository;
import hello.core.order.OrderService;
import hello.core.order.OrderServiceImpl;

public class AppConfig {
    private MemberService memberService(){
        return new MemberServiceImpl(new MemoryMemberReposito
    }
    public OrderService orderService(){
        return new OrderServiceImpl(new MemoryMemberRepositor
    }
}
```



- 실제 동작에 필요한 **'구현 객체'**생성
  - `MemberServiceImpl`
  - `MemoryMemberRepository`
  - `OrderServiceImpl`
  - `FixDiscountPolicy`
- 생성한 객체 인스턴스의 참조(레퍼런스)를 **생성자를 통해서 주입(연결)** 해준다.
  - `MemberServiceImpl` → `MemoryMemberRepository`
  - `OrderServiceImpl` → `MemoryMemberRepository`, `FixDiscountPolicy`
- appConfig 객체는 `memoryMemberRepository` 객체를 생성하고 그 참조값을 `memberServiceImpl` 을 생성하면서 생성자로 전달한다.
- 클라이언트인 `memberServiceImpl` 입장에서 보면 의존관계를 마치 외부에서 주입해주는 것 같다고 해서 **DI(Dependency Injection)** 우리말로 의존관계 주입 또는 의존성 주입이라 한다.

```
MemberService memberService = appConfig.memberService();
OrderService orderService = appConfig.orderService();
```

AppConfig 리팩터링

💡 실습 코드 작성

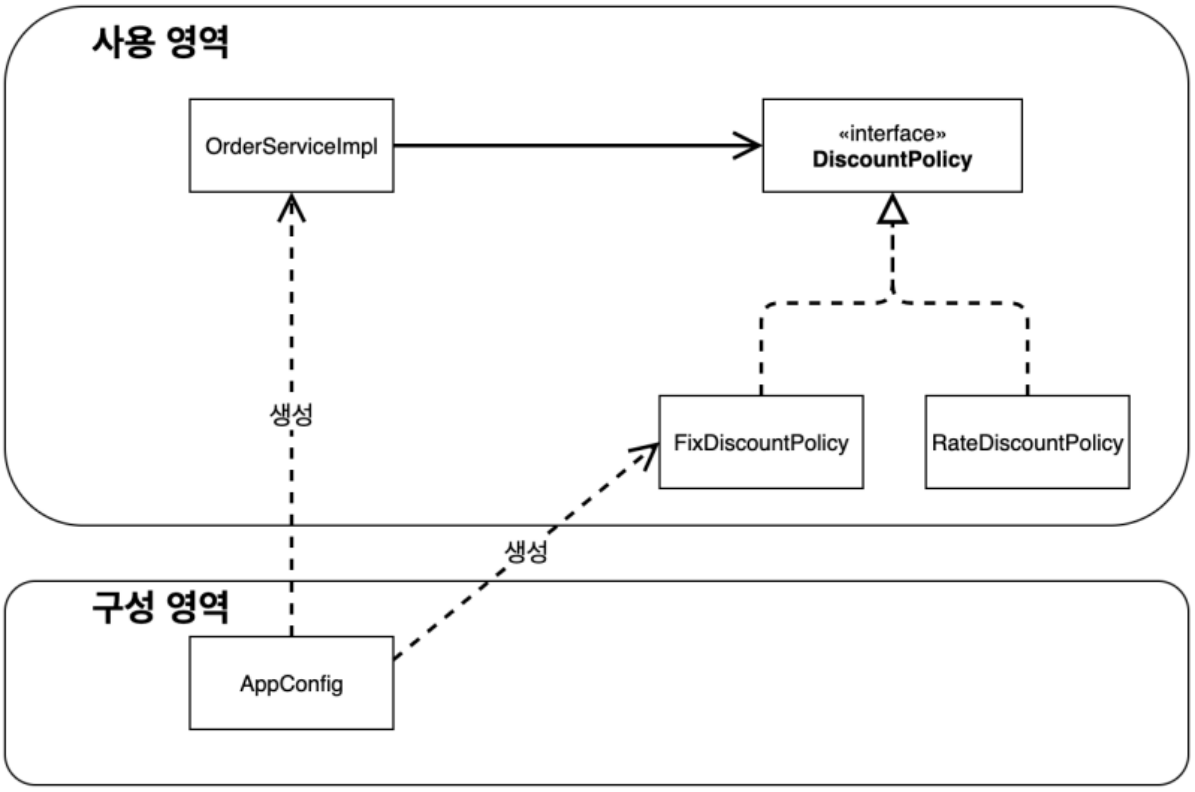
```
public class AppConfig {
    public MemberService memberService(){
        return new MemberServiceImpl(memberRepository());
    }

    private static MemoryMemberRepository memberRepository()
        return new MemoryMemberRepository();
        return new
    }

    public OrderService orderService(){
        return new OrderServiceImpl(memberRepository(), disco
    }

    private static DiscountPolicy discountPolicy() {
        return new RateDiscountPolicy();
    }
}
```

새로운 구조와 할인 정책 적용



- 이제 할인 정책을 변경해도, 애플리케이션의 구성 역할을 담당하는 `AppConfig` 만 변경하면 된다.

  → 클라이언트 코드인 `OrderServiceImpl` 를 포함해서 사용 영역의 어떤 코드도 변경할 필요가 없다.

IoC, DI, 그리고 컨테이너

- IoC(Inversion of Control) 제어의 역전
  - 프로그램의 제어 흐름을 직접 제어하는 것이 아니라 외부에서 관리하는 것을 제어의 역전(IoC) 이라 한다.
- DI(Dependency Injection) 의존성 주입
  - `OrderServiceImpl` 은 `DiscountPolicy` 인터페이스만을 의존
    - 실제 어떤 구현 객체가 사용될 지 모른다.
  - 정적인 클래스 의존 관계
  - 실행 시점에 결정되는 동적인 객체(인스턴스) 의존 관계

    → 둘을 분리해서 생각해야 한다.
- IoC, DI 컨테이너

- `AppConfig` 처럼 객체를 생성하고 관리하면서 의존관계를 연결해주는 것
- 의존관계 주입에 초점을 맞추어 최근에는 주로 **DI 컨테이너**라 한다.

스프링으로 전환하기

- 기존 `AppConfig` 의 class와 method 들에 `@Configuration` , `@Bean` Annotation을 선언해주자.
- 그 후 App에서는 ApplicationContext 객체(스프링 컨테이너)를 생성

```
public class OrderApp {
    public static void main(String[] args) {

        ApplicationContext applicationContext = new Annotatio
        MemberService memberService = applicationContext.getB
        OrderService orderService = applicationContext.getBea

        Long memberId = 1L;
        Member member = new Member(memberId, "memberA", Grade
        memberService.join(member);

        Order order = orderService.createOrder(memberId, "ite

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

스프링 컨테이너

- `ApplicationContext` → 스프링 컨테이너라 한다.
- 기존에는 개발자가 `AppConfig` 를 사용해서 직접 객체를 생성하고 DI를 했다.
- 스프링 컨테이너는 `@Configuration` 이 붙은 `AppConfig` 를 설정(구성) 정보로 사용한다. 여기서 `@Bean` 이라 적힌 메서드를 모두 호출해서 반환된 객체를 컨테이너에 등록한다.
- 스프링 컨테이너에 등록된 객체 → 스프링 빈(`Bean`)
- 스프링 빈은 `@Bean` 이 붙은 메서드의 명을 `Bean` 의 이름으로 사용한다.
  - `( memberService , orderService )`

- 이전에는 개발자가 필요한 객체를 `AppConfig` 를 사용해서 직접 조회했지만, 이제부터는 컨테이너를 통해서 필요한 `Bean` 을 찾아야 한다.
- 스프링 빈은 `applicationContext.getBean()` 를 사용해서 찾을 수 있다.
- 기존에는 개발자가 직접 자바코드로 모든 것을 했다면 이제부터는 컨테이너에 객체를 `Bean`으로 등록 하고, 스프링 컨테이너에서 `Bean`을 찾아서 사용하도록 변경되었다