스프링 핵심 원리 - 기본편

▼ section1. 객체 지향 설계와 스프링

객체 지향 특징

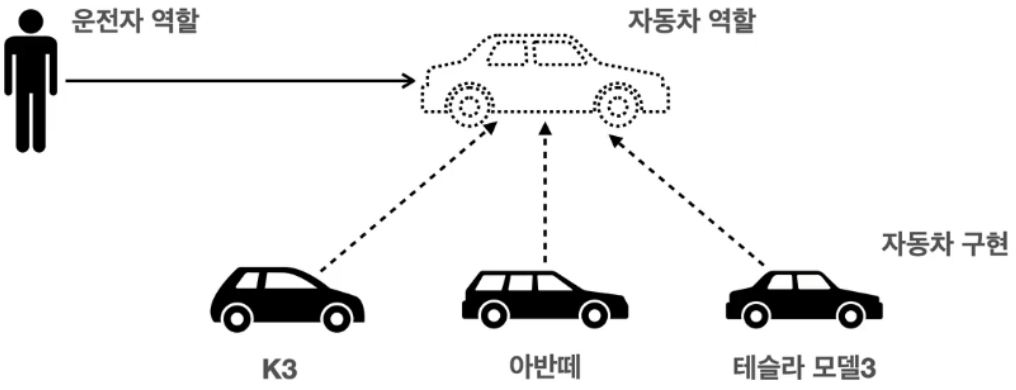
- 추상화
- 캡슐화
- 상속
- 다형성

객체 지향 프로그래밍의 개념

- 컴퓨터 프로그램을 여러 개의 독립된 단위, 즉 **'객체'** 들의 **모임**으로 파악하고자 하는 것
- 각각의 **객체**는 **메시지**를 주고받고, 데이터를 처리할 수 있다.**(협력)**
- 프로그램을 **유연**하고 **변경**이 용이하게 만들기 때문에 대규모 소프트웨어 개발에 사용된다.

다형성의 실세계 비유

- `역할`과 `구현`으로 세상을 구분



    - 운전자는 자동차의 역할(인터페이스)만 이해하고 있더라도 운전에 지장은 없다.
        - 자동차 세상을 무한히 확장 가능

          → 클라이언트에 영향을 주지 않고 새로운 기능을 제공할 수 있음

          → 따라서 새로운 자동차가 나온다고 하더라도 클라이언트는 익숙하게 사용 가능

역할과 구현을 분리

- 단순, 유연, 변경 ↑
- 장점
    - 클라이언트는 인터페이스만 알면 된다.
    - 내부 구조를 몰라도 되며, 구조가 변경되어도 영향을 받지 않는다.

다형성의 본질

- 클라이언트를 변경하지 않고, 서버의 구현 기능을 유연하게 변경할 수 있다.

한계

- 인터페이스 자체가 변하면 client, server 모두 큰 변경 발생

객체 지향 설계의 5가지 원칙

단일 책임 원칙 SRP(Single Responsibility Principle)

- 한 클래스는 하나의 책임만 가져야 한다
- 중요한 기준은 변경이다. 변경이 있을 때 파급 효과가 적으면 단일 책임의 원칙을 따른다.
    - 예) UI 변경, 객체의 생성과 사용을 분리

개방 폐쇄 원칙 OCP(Open/Closed Principle)

- 소프트웨어 요소는 확장에는 열려 있으나 변경에는 닫혀 있어야 한다.
- 다형성을 활용해보자

💡 문제점

- MemberService 클라이언트가 구현 클래스를 직접 선택
  - `MemberRepository m = new MemoryMemberRepository();` // 기존 코드
  - `MemberRepository m = new JdbcMemberRepository();` // 변경 코드
- 구현 객체를 변경하려면 클라이언트 코드를 변경해야 한다.
- 분명 다형성을 사용했지만 OCP 원칙을 지킬 수 없다.
- 이 문제를 해결하기 위해서는

  → 객체를 생성하고, 연관관계를 맺어주는 별도의 조립, 설정자가 필요하다.

리스코프 치환 원칙 LSP(Liskov Substitution Principle)

- 프로그램의 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다.

인터페이스 분리 원칙 ISP(Interface Segregation Principle)

- 특정 client 를 위한 인터페이스 여러 개가 범용 인터페이스 하나보다 낫다.

의존관계 역전 원칙 DIP(Dependency Inversion Principle)

- 추상화에 의존해야지, 구체화에 의존하면 안된다.
- 구현체에 의존하게 되면 변경이 아주 어려워진다.
  - `MemberRepository m = new MemoryMemberRepository();` ← **DIP 위반**

정리

- 다형성 만으로는 OCP, DIP를 지킬 수 없다.

Spring

- 스프링은 다음 기술로 다형성 + OCP, DIP를 가능하게 지원
  - DI(Dependency Injection): 의존관계, 의존성 주입
  - DI 컨테이너 제공
- 클라이언트 코드의 변경 없이 기능 확장

- 쉽게 부품을 교체하듯이 개발

▼ section2. 스프링 핵심 원리 이해1 - 예제 만들기

프로젝트 생성

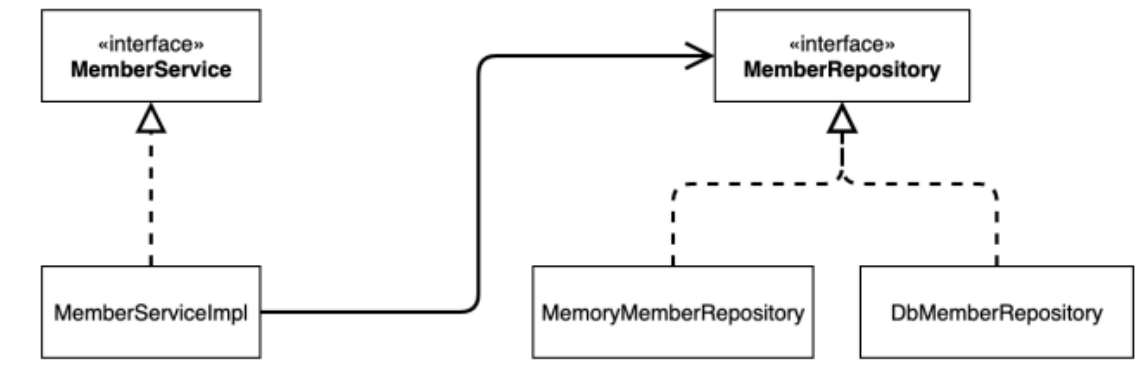
- Gradle
- groupId: hello
- artifactId: core
- Dependencies: 선택 X

비즈니스 요구사항과 설계

- 회원
  - 회원가입, 회원조회
  - 회원등급 - 일반, VIP
  - 회원 데이터는 자체 DB 구축, 외부 시스템 연동 (미확정)
- 주문, 할인 정책
  - 회원은 상품 주문 가능
  - 회원 등급에 따른 할인 정책 적용
  - 할인 정책은 VIP는 1,000원(고정 금액) 할인 ← 변경 가능성 큼

회원 도메인 설계

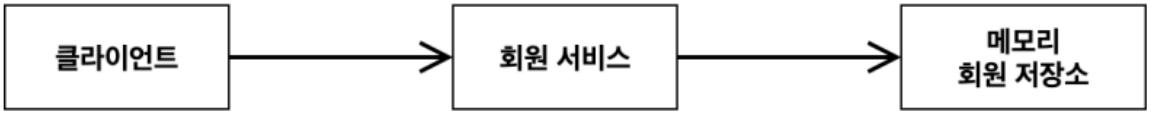
회원 클래스 다이어그램



회원 도메인 개발

💡 실습 코드 작성

- 사용자 저장 메서드 `save()`
- 사용자 조회(id) 메서드 `findMember()`



`MemberService` → `MemberServiceImpl` → `MemberRepository` → `MemoryMemberRepository`

회원 도메인 실행과 테스트

```
// 순수 자바 코드로 생성된 MemberApp
import hello.core.member.Grade;
import hello.core.member.Member;
import hello.core.member.MemberService;
import hello.core.member.MemberServiceImpl;

public class MemberApp {
    public static void main(String[] args) {
        MemberService memberService = new MemberServiceImpl
        Member member = new Member(1L, "memberA", Grade.VIP
        memberService.join(member);

        Member findMember = memberService.findMember(1L);
        System.out.println("new member = " + member.getName
        System.out.println("find Member = " + findMember.ge
    }
}
```

- 해당 코드의 설계 상 문제점은?
  - 다른 저장소로 변경할 때 OCP 원칙은 지켜지는가?
  - DIP 잘 지켜지고 있는가?

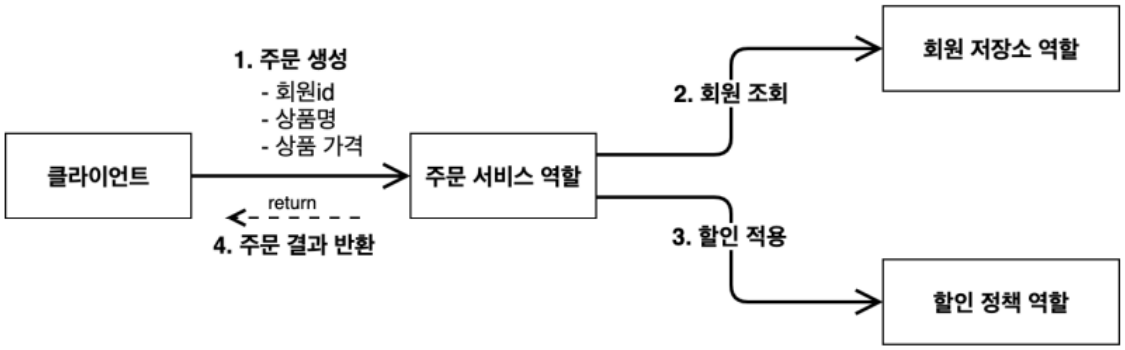
→ '의존관계가 인터페이스 뿐만 아니라 구현까지 모두 의존하는 문제점이 있음'

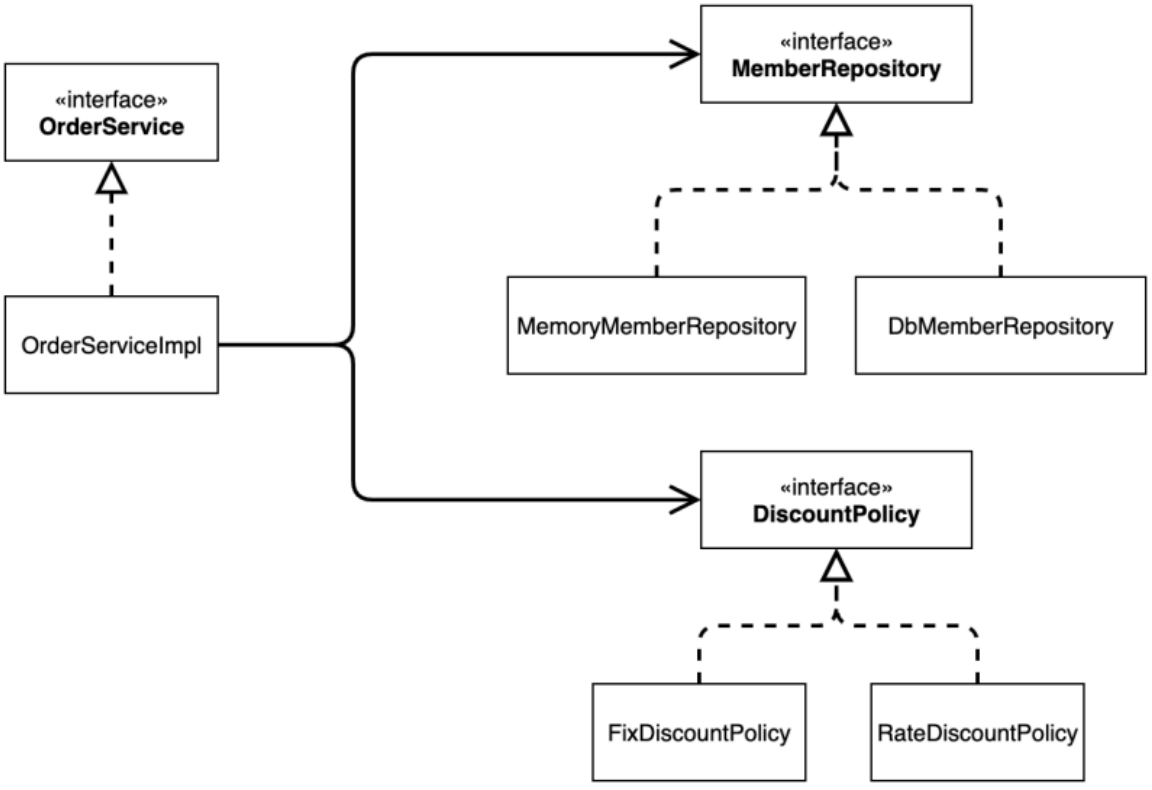
실습코드를 보면 `MemberRepository memberRepository = new MemoryMemberRepository;`

즉, `MemberServiceImpl` 은 추상화, 구현 모두에게 의존하고 있음

주문과 할인 도메인 설계

> 💡 실습 코드 작성





주문과 할인 도메인 개발

💡 실습 코드 작성

`Order createOrder(Long memberId, String itemName, int itemPrice)`

- 위 실습 코드의 설계는 Order 자체는 할인 정책과, 사용자의 정보를 알고있지 않다.
  - 잘 설계된 코드 ← 단일 책임의 원칙

주문과 할인 도메인 실행과 테스트

💡 실습 코드 작성

▼ section3. 스프링 핵심 원리 이해2 - 객체 지향 원리 적용

새로운 할인 정책 개발

고정 금액 할인이 아닌 정률 할인으로 변경(금액의 10%)

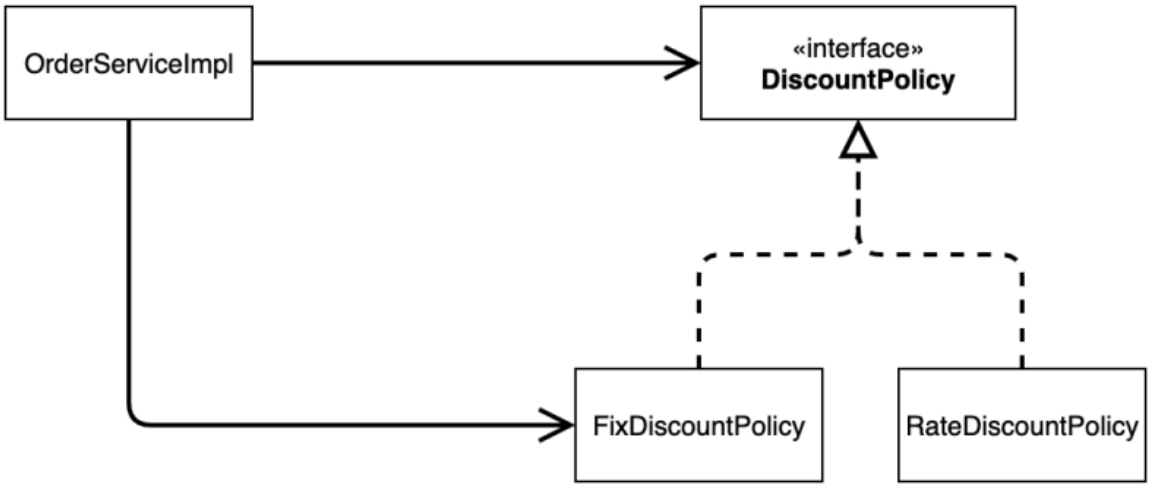
💡 실습 코드 작성

- `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 RateDisco
//private final DiscountPolicy discountPolicy = new FixDisc
```



- 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

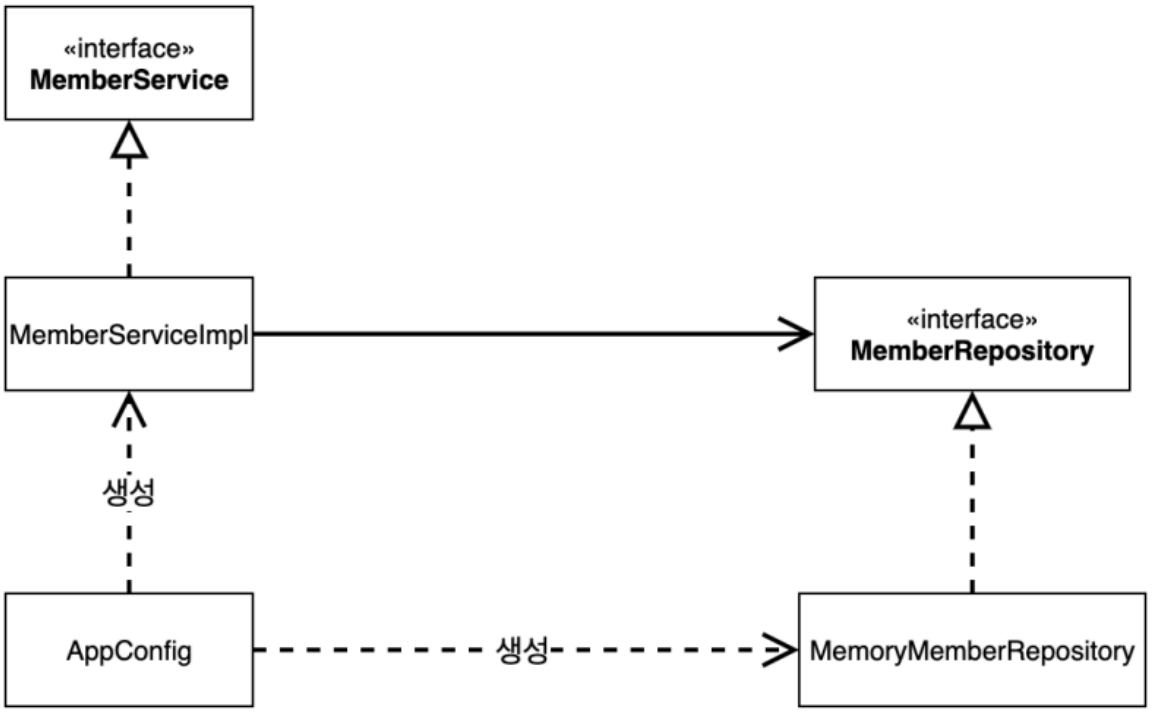
💡 실습 코드 작성

```
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 MemoryMemberReposi
    }
    public OrderService orderService(){
        return new OrderServiceImpl(new MemoryMemberReposit
    }
}
```



- 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 주입해 주는 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

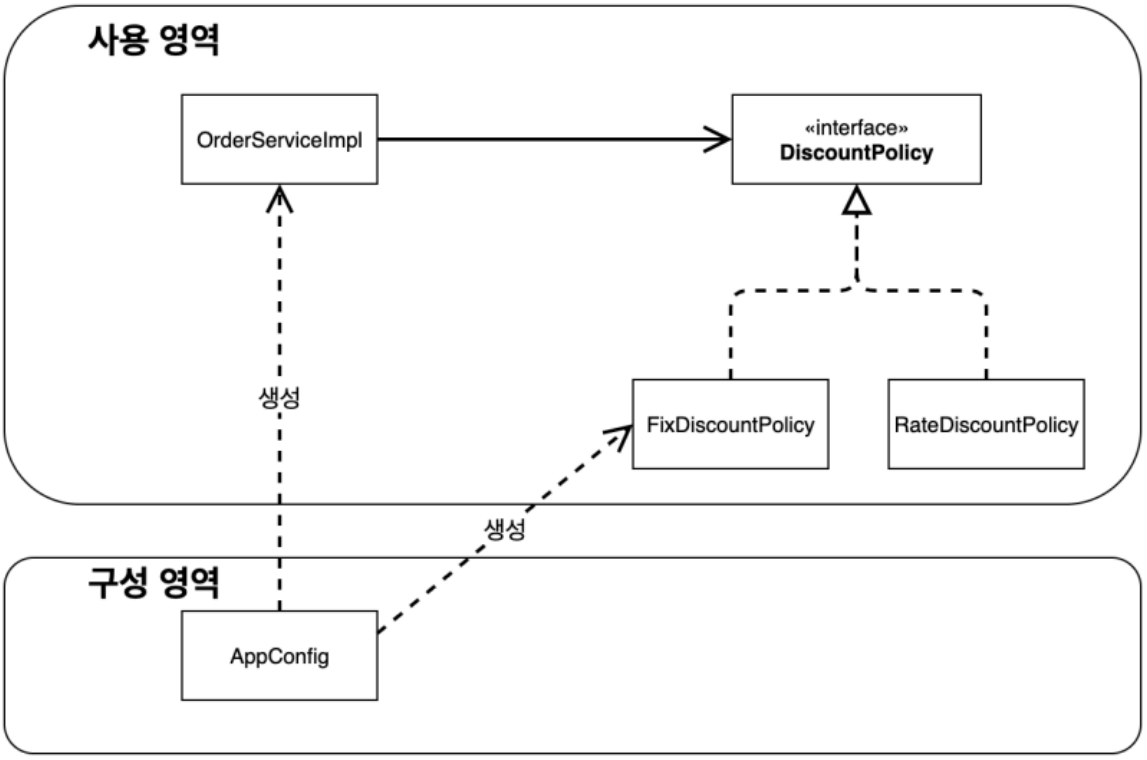
```
public class AppConfig {
    public MemberService memberService(){
        return new MemberServiceImpl(memberRepository());
    }

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

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

    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 Annotat
        MemberService memberService = applicationContext.ge
        OrderService orderService = applicationContext.getB

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

        Order order = orderService.createOrder(memberId, "i

        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