객체 지향 설계와 스프링

스프링의 핵심 개념, 컨셉?

- 자바 언어 기반의 프레임워크
- 자바 언어의 가장 큰 특징 - 객체 지향 언어
- 스프링은 객체 지향 언어가 가진 강력한 특징을 살려내는 프레임워크

  = 좋은 객체 지향 애플리케이션을 개발할 수 있게 도와주는 프레임워크

스프링

객체 지향을 잘 할 수 있게 도와주는 프레임워크

좋은 객체 지향이란?

- 유연하고 변경이 용이
  - 레고 블럭 조립하듯이, 컴퓨터 부품 갈아 끼우듯이...

  = 다형성

객체지향

- 객체들의 모임
- 각각의 객체는 메시지를 주고받고 데이터를 처리

다형성

- 운전자는 테슬라에서 아반떼로 차가 바뀌어도 운전할 수 있음.

  → 구현된 자동차는 다르지만, 자동차의 역할은 동일하기 때문!

새로운 자동차가 나와도, 클라이언트는 새로 안 배워도 됨!

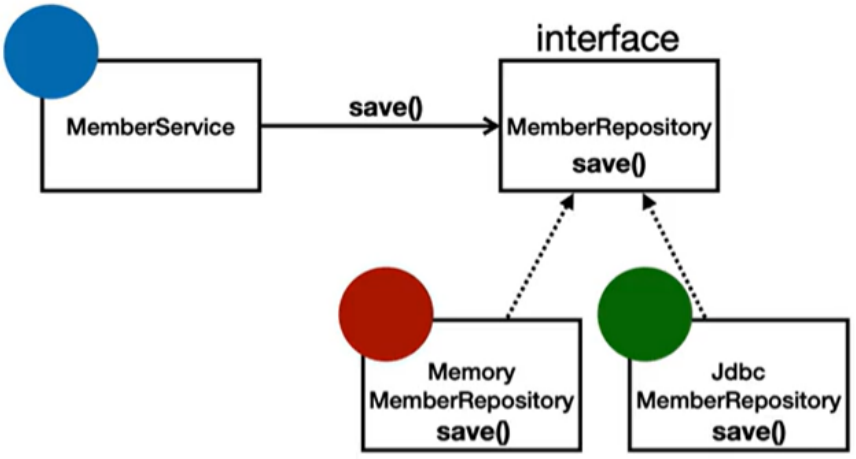
- 뮤지컬 더블 캐스팅 - 배우 대체 가능 = 유연하고 변경에 용이하다

역할과 구현을 분리

- 클라이언트는 대상의 역할 (인터페이스)만 알면 됨.
- 클라이언트는 구현 대상의 내부 구조를 몰라도, 변경되어도 영향 X
- 역할 = 인터페이스
- 구현 = 인터페이스를 구현한 클래스, 구현 객체
- 객체 설계 시 역할을 먼저 부여하고, 그 역할을 수행하는 구현 객체 만들기

자바 언어의 다형성

- 오버라이딩을 떠올리자.



- 다형성의 본질 : 인터페이스를 구현한 객체 인스턴스를 실행 시점에 유연하게 변경 가능

  → 클라이언트를 변경하지 않고, 서버의 구현 기능을 유연하게 변경할 수 있음

한계점

- 역할 자체가 변하면 클라이언트, 서버 모두에 큰 변경이 발생

  ex) 대본 자체가 변경된다면? 자동차를 비행기로 변경해야 한다면?

→ 인터페이스를 안정적으로 설계하는 것이 중요

스프링과 객체 지향

- 다형성이 가장 중요하다! 스프링은 다형성을 극대화해서 이용할 수 있게 도와준다.
- 스프링에서 이야기하는 제어의 역전(IoC), 의존관계 주입(DI)은 다형성을 활용해서 역할과 구현을 편리하게 다룰 수 있도록 지원.

좋은 객체 지향 설계의 5가지 원칙 (SOLID)

- SRP : 단일 책임의 원칙 (single responsibility principle)
- OCP : 개방-폐쇄 원칙 (Open/closed principle)
- LSP : 리스코프 치환 원칙 (Liskov substitution principle)
- ISP : 인터페이스 분리 원칙 (Interface segregation principle)
- DIP : 의존관계 역전 원칙 (Dependency inversion principle)

- **SRP 단일 책임의 원칙**
  - 한 클래스는 **하나의 책임**만 가져야 한다.
  - 책임?? = 변경을 할 때 파급 효과가 적도록

    ex) UI 변경, 객체의 생성과 사용을 분리

- **OCP 개방-폐쇄 원칙**
  - 소프트웨어 요소는 확장에는 열려있으나 변경에는 닫혀야 한다.
  - 무슨 말? ⇒ 다형성을 생각해 보자!
  - 인터페이스를 구현한 새로운 클래스를 하나 만들어서 새로운 기능을 구현
  - 문제점

```
MemberRepository m = new MemoryMemberRepository(); // 기
MemberRepository m = new JdbcMemberRepository(); // 변경
```

    - 구현 객체를 변경하려면 클라이언트 코드를 변경해야 한다.
    - 다형성의 원칙을 사용했지만, OCP를 원칙을 지킬 수 없음.

        - 이걸 어떻게 해결? : 객체를 생성하고 연관관계를 맺어주는 별도의 조립, 설정자가 필요

- **LSP 리스코프 치환 원칙**
    - 프로그램 객체는 프로그램의 정확성을 깨뜨리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 함
    - ex) 자동차 인터페이스는 구현체가 구현해야 됨.
        - 이때 악셀을 밟으면 무조건 앞으로 가야 한다! 뒤로 가면 안됨.
        - 뒤로 가도록 만들 거면 이건 규약을 어긴 거니 LSP 원칙을 벗어남.

- **ISP 인터페이스 분리 원칙**
    - 특정 클라이언트를 위한 인터페이스 여러 개가 범용 인터페이스 하나보다 낫다.
    - 자동차 인터페이스 → 운전 인터페이스, 정비 인터페이스로 분리
    - 사용자 클라이언트 → 운전자 클라이언트, 정비사 클라이언트로 분리
    - 정비 인터페이스 자체가 변해도 운전자 클라이언트에 영향을 주지 않음.
    - 인터페이스가 명확 & 대체 가능성 ↑

- **DIP 의존관계 역전 원칙**
    - “추상화에 의존해야지, 구체화에 의존하면 안 된다!”
    - 클라이언트 코드가 구현 클래스에 의존하지 말고 인터페이스에 의존하라.
    - 역할에 의존해야 한다는 것과 같은 말.

      ex) 연극에서 배우 한 명에 의존하면 안 됨.

      ```
      MemberRepository m = new MemoryMemberRepository(); // 기
      ```

        - 이 코드는 인터페이스에 의존하지만, 구현 클래스도 동시에 의존.
        - MemberService 클라이언트가 구현 클래스를 직접 선택.

    - DIP 위반

정리

- 객체 지향의 핵심은 **다형성**
- 다형성만으로는 쉽게 부품을 갈아끼우듯이 개발 불가
- 다형성만으로는 구현 객체를 변경할 때 클라이언트 코드도 함께 변경됨
- 다형성만으로는 **OCP, DIP를 지킬 수 없다**.
- 뭔가가 더 필요하다!

객체 지향 설계와 스프링

스프링

- DI (의존관계, 의존성 주입), DI 컨테이너를 통해 다형성 + OCP, DIP를 가능하게 지원
- 클라이언트 코드를 변경 없이 기능 확장
- 쉽게 부품을 교체하듯 개발

실무 고민

- 인터페이스를 도입하면 추상화라는 비용이 발생
- 기능을 확장할 가능성이 없다면, 구체 클래스를 직접 사용하고 향후 꼭 필요할 때 리팩토링 해서 인터페이스를 도입