**SOLID**

클린 코드로 유명한 로버트 마틴이 좋은 객체 지향 설계의 5가지 원칙을 정리

SRP : 단일 책임 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 책임 원칙**

- 한 클래스는 하나의 책임만 가져야 한다.

- 하나의 책임이라는 것은 모호하다.

> 클 수 있고, 작을 수 있다.

> 문맥과 상황에 따라 다르다.

- 중요한 기준은 변경이다. 변경이 있을 때 파급 효과가 적으면 단일 책임 원칙을 잘 따른 것이다.

> UI 변경, 객체의 생성과 사용을 분리.

Ex ) UI 하나를 변경하는 데 sql코드부터 다 바꿔야하면 잘못 설계한 것.

범위를 너무 작게 하면 기능이 너무 작게 쪼개지고, 또 너무 크게 하면 책임이 너무 커질 수 있기 때문에 상황에 따라 범위를 잘 선정해야한다.

**OCP 개방 폐쇄 원칙**

- 소프트웨어 요소는 확장에는 열려 있으나 변경에는 닫혀 있어야한다.

- 이런 거짓말 같은 말이? 확장을 하려면, 당연히 기존 코드를 변경?

- 다형성을 활용해보자

- 인터페이스를 구현한 새로운 클래스를 하나 만들어서 새로운 기능을 구현

- 지금까지 배운 역할과 구현의 분리를 생각해보자.

확장을 하려면, 코드를 변경하면 안된다. -> 이게 가능한가?

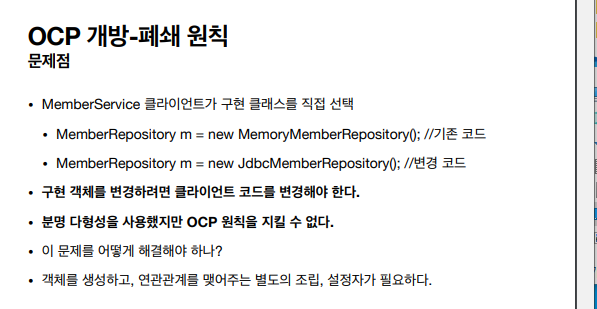
자바 언어의 다형성을 잘 지키면 기존 코드를 유지한 채 기능을 확장할 수 있다.

- > 인터페이스를 구현한 새로운 클래스를 하나 만드는 것은 기존 코드를 변경하는 것이 아니다.

구현체를 변경하는 것은 기존의 코드를 바꾸는 데 영향을 끼치지 않는다.



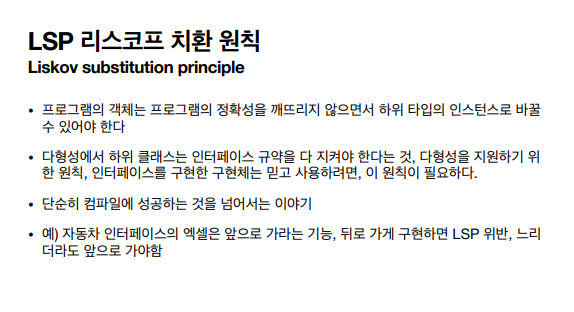
인터페이스를 구현한 객체를 변경하기 위해서는 위처럼 코드를 변경해야하는데, 이는 변경에 닫혀있지않다. 이 문제는 어떻게 해결할까?



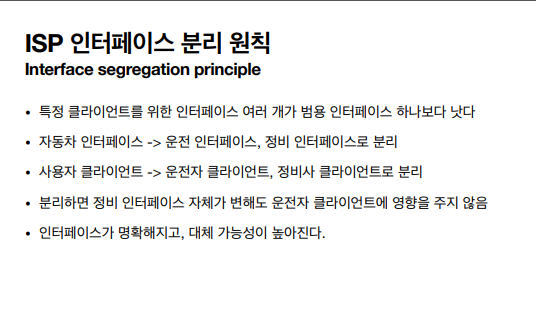
다형성을 지키고 있긴 하지만, 클라이언트가 변경을 해야한다. OCP 원칙을 지킬 수 없음.

객체를 생성하고, 연관관계를 맺어주는 별도의 조립, 설정자가 필요하다.

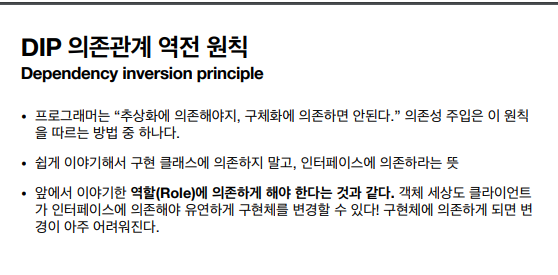
이 역할을 스프링 컨테이너가 해주고 있고, 이 OCP 원칙을 지키기 위해서 DI도 필요하고 IOC 컨테이너라는 것도 필요하다. -> 이후에 코드를 가지고 설명할 예정



인터페이스가 있고, 그거에 대한 구현체가 있음. 인터페이스에서 정의되어있는 규약을 지키는 구현체를 만들어야한다. 예를 들어, 자동차의 인터페이스를 구현한 구현체에서 엑셀이라는 기능을 구현할 때, 엑셀을 밟으면 속도가 증가하게 만드는 것이 자동차의 인터페이스 구약으로 볼 수 있다. 하지만, 만약 엑셀을 밟으면 속도가 감소하는 자동차를 만든다고 했을 때, 이는 컴파일 오류가 나지 않지만 인터페이스 규약을 지키지 않는 클래스를 만든 것이므로 리스코프 치환원칙을 위배한다고 볼 수 있다. 인터페이스의 목적 구현을 지켜야한다.



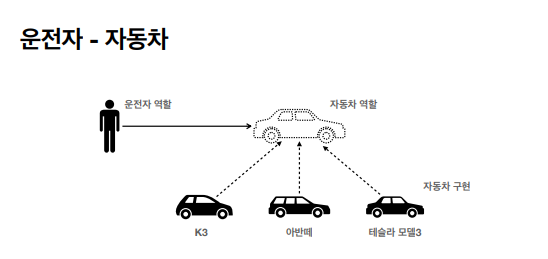
기능을 딱 거기에 맞게 잘라서 만들어야한다.



OCP 와 DIP 가 중요한 원칙이다.

클라이언트 코드가 구현 클래스를 바라보지 말고 인터페이스만 바라보라는 뜻.

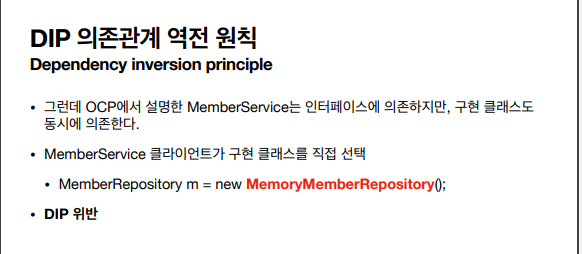
멤버 서비스가 멤버 리포지토리 인터페이스만 바라보고 메모리 멤버 리포지토리나 JDBC 멤버 리퍼지토리는 몰라야한다. 구현체를 직접 바라보는 것이 아닌 인터페이스를 바라봐야지 (다형성) 변경을 쉽게 할 수 있다.



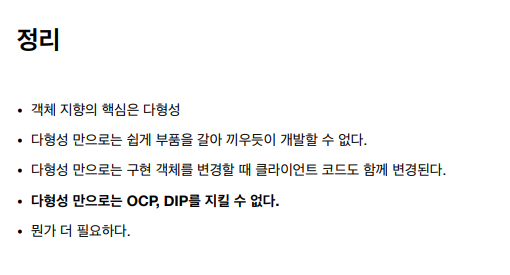
운전자는 자동차의 역할만 알고 차를 몰면 되지 k3의 구체적인 기능을 다 알아야만 운전을 할 수 있으면 안된다.

역할과 구현을 철저하게 분리해서 구현하여 시스템도 언제든지 갈아 끼울 수 있게 설계를 해야한다.

Ex) 배우도 대본(인터페이스)에 의존해서 연습을 해야지, 다른 실제 담당 배우랑 따로 대본을 짜서 연습을 하게 되면 그 담당배우 없이는 무대에 설 수 없다.



인터페이스에 의존하지만, 구현 클래스도 동시에 의존하고 있음. 의존한다는 것 : 내가 저 코드를 알고 있다. 코드의 존재를 알고 있다.



멤버 서비스는 멤버 리포지토리 인터페이스에만 의존하도록 설계를 해야한다.

다형성이 핵심이지만, 다형성만으로는 쉽게 부품을 갈아 끼우듯이 개발할 수 없다.

-> 이것을 해결하기 위한 목적으로 스프링이 개발됨.