7/6 ~ 7/10 1주차 - 4

객체간의 의존관계인 Dependency와 이를 없애기 위한 DI에 대해서 알아보자!

의존관계(Dependency)

빈의 컨테이너로써의 스프링은 빈의 생성, 관계 설정, 객체 관리를 담당한다.

의존관계란 객체간의 참조로 인해 참조되는 객체 변경시 참조하는 객체가 영향을 받는 것을 의미한다.

일반적으로 PhoneUser에서 SPhone을 사용하려면 아래와 같이 SPhone을 멤버 변수로 선언하고 사용할 것이다.

```
public class PhoneUser {
    SPhone phone = new SPhone(); // Has a 관계 발생

    // SPhone 사용 코드
}
```

이 상황에서 만약 SPhone을 LPhone으로 변경되어야 한다면 PhoneUser에는 어떤 변화가 발생할까?

```
public class PhoneUser {
    Lphone phone = new LPhone();

    // SPhone 사용 코드 --> 모두 LPhone 사용 코드로 변경 필요
}
```

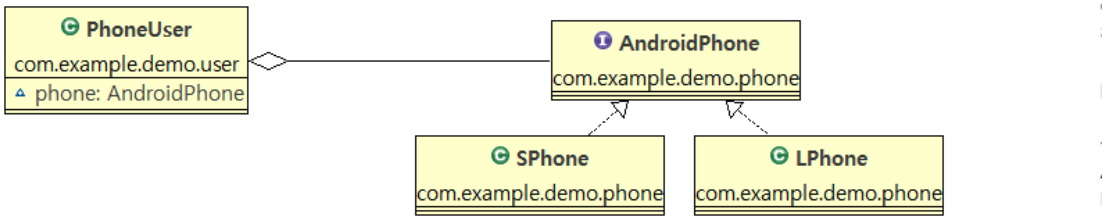
멤버인 phone이 변경되면서 원래 SPhone을 사용하던 코드를 모두 LPhone을 사용하는 코드로 변경되어야 한 것이다. 이때 SPhone이나 LPhone을 PhoneUser의 의존성이라고 하고 PhoneUser는 SPhone에 의존하고 있다.

이런 의존성이 커지면 프로젝트의 변화도 커진다.

어떻게 하면 줄일 수 있을까?

바로 Interface 사용하기

LPhone SPhone에 인터페이스를 적용해서 PhoneUser와 loose coupling 하도록 만들어 주면 좋다.



interface를 적용하면 PhoneUser는 아래와 같이 변경될 것이다.

```
public class PhoneUser {
    AndroidPhone phone = new SPhone();
    // AndroidPhone 사용 코드
}
```

구현체가 변경되어도 AndroidPhone 타입으로 사용하기 때문에 수정해야할 부분이 크게 준다.

하지만 각각 구현체를 생성하는 코드를 변경해야하는 일이 있다.

왜? → PhoneUser는 SPhone 또는 LPhone에 의존하고 있다

구현체 코드를 없애는 방법? - > Factory 패턴의 적용

말그래도 원하는 객체를 공장에서 찍어내는형태이다

maker 정보에 따라 SPhone, LPhone이 반환되고 리턴 타입은 AndroidPhone이다.

```java
public class PhoneFactory {
    public static AndroidPhone getPhone(String maker) {
        if (maker.equals("S")) {
            return new SPhone();
        } else {
            return new LPhone();
        }
    }
}
```

이를 사용하는 PhoneUser는 다음과 같다.

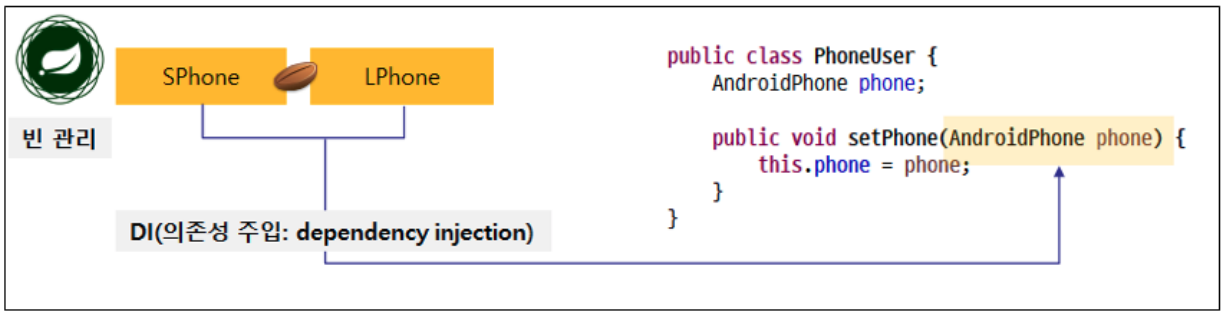
```java
public class PhoneUser {
    //AndroidPhone phone = PhoneFactory.getPhone("S");

    AndroidPhone phone = PhoneFactory.getPhone("L");
}
```

의존 관계를 완전히 없앴다 비즈니스 로직을 구현해야할 개발자에게 이런 코드까지 작성하라고 하기에는 시스템의 복잡도가 너무나 증가하게된다 .

DI

스프링은 위와 같은 상황을 DI(Dependency Injection) 즉 의존성 주입이라는 것으로 처리한다.!



스프링은 객체들을 빈으로 관리한다.

PhoneUser에서는 AndroidPhone 타입의 멤버 변수 phone이 선언만 되어있는 상ㄴ태이다.

phone이 할당되는 시점은 누군가가 setPhone메서드를 호출하면서 AndroidPhone타입의 객체를 전달한 후이다.

이처럼 의존성인 AndroidPhone을 PhoneUser에서 만들지 않고 외부에서 넣어주는 것을 의존성 주입이라 한다. 코드로는 큰 차이가 없지만 메타설정에 의해 스프링 프레임워크가 해준다는 것이 중요하다.!

결국 DI란 객체의 의존관계 즉 멤버 변수를 외부에서 설정해주는 것으로 일반적으로 생성사 기반으로 처리하거나 setter 메서드를 이용해서 처리하게 된다.

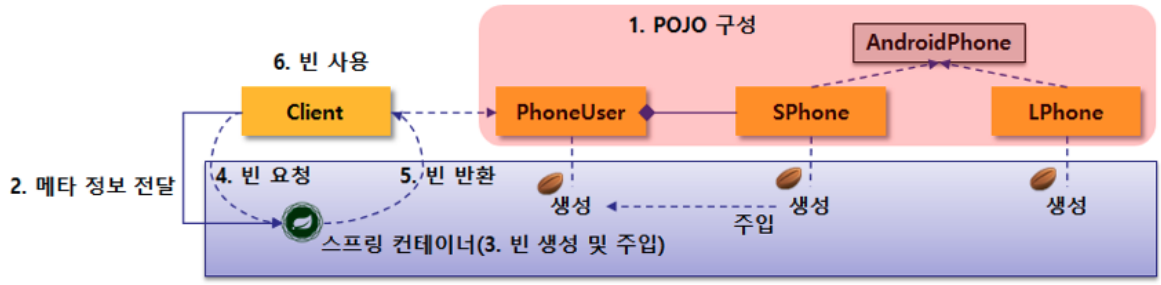
```
// 생성자를 통한 DI
public class PhoneUser {
    AndroidPhone phone;

    public PhoneUser(AndroidPhone phone) {
        this.phone = phone;
    }
}

// setter를 통한 DI
public class PhoneUser {
    AndroidPhone phone;

    public void setPhone(AndroidPhone phone) {
        this.phone = phone;
    }
}
```

스프링에성의 DI동작은 어떻게 될까?



1. 개발자는 비즈니스 로직을 담은 POJO들을 작성한다. 물론 이 POJO들은 빈으로 관리될 녀석들이다.
2. 개발자는 각 POJO들을 어떻게 객체화 하고 관계를 맺어줄 지 설명서인 메타정보를 작성해서 스프링 컨테이너에게 전달한다.
3. 스프링 컨테이너는 메타정보에 근거해 POJO 객체를 만들고 setter/생성자를 호출해서 빈 설정을 마무리한다.
4. 클라이언트 코드에서 빈에 대한 요청을 컨테이너에게 하면
5. 컨테이너는 관리하고 있던 빈 객체를 반환해준다.

묵시적 DI와 명시적 DI는 추후에 정리할 것이다.