Spring 생성자 주입에 대해.

추상화란?

객체지향에서의 추상화는 무엇일까?.

- 사전적 : 여러가지 사물이나 개념에서 공통되는 특성이나 속성 따위를 추출하여 파악하는 작용.
- OOP에서 추상화는 프로그램을 구성하는 객체들이 어떤 필드와 메소드를 갖는지를 추출함을 의미한다.

필요성으로 OOP에서 추상화를 개발하면서 고려해야 하는 사항들은?

- 어떤 종류의 객체들을 정의할 것인지?
- 각 객체들은 어떤 필드와 메소드를 갖을 것인지?

Spring 생성자 주입 공부중.

OCP 원칙?.

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }

    return area;
}
```

if - else가 계속추가됨.

추상화.

```
public abstract class Shape
{
    public abstract double Area();
}
```

Inheriting from ~~Shape the Rectangle~~ and Circle classes now looks like this:

```
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}
```

✓ 상속

✓ 상속

As we've moved the responsibility of actually calculating the area away from AreaCalculator's Area method it is now much simpler and robust as it can handle any type of Shape that we throw at it.

```
public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }

    return area;
}
```

param을 (Shape) 추상화한 타입을 사용한다

In other words we've closed it for modification by opening it up for extension.

2. 생성자 주입을 사용해야 하는 이유

최근에는 Spring을 포함한 DI 프레임워크의 대부분이 생성자 주입을 권장하고 있는데, 자세한 이유를 살펴보도록 하자.

[생성자 주입을 사용해야 하는 이유]

1. 객체의 불변성 확보
실제로 개발을 하다 보면 느끼겠지만, 의존 관계 주입의 변경이 필요한 상황은 거의 없다. 하지만 수정자 주입이나 일반 메소드 주입을 이용하면 불필요하게 수정의 가능성을 열어두게 되며, 이는 OOP의 5가지 개발 원칙 중 OCP(Open-Closed Principal, 개방-폐쇄의 법칙)를 위반하게 된다. 그러므로 생성자 주입을 통해 변경의 가능성을 배제하고 불변성을 보장하는 것이 좋다.

2. 테스트 코드의 작성
실제 코드가 필드 주입으로 작성된 경우에는 순수한 자바 코드로 단위 테스트를 작성하는 것이 불가능하다. 예를 들어 아래와 같은 실제 코드가 존재한다고 하자.

```
@Service
public class UserServiceImpl implements UserService {

  @Autowired
  private UserRepository userRepository;
  @Autowired
  private MemberService memberService;

  @Override
  public void register(String name) {
    userRepository.add(name);
  }

}
```

위의 코드에 대한 순수 자바 테스트 코드를 작성하면 다음과 같이 작성할 수 있다.

```
public class UserServiceTest {

  @Test
  public void addTest() {
    UserService userService = new UserServiceImpl();
    userService.register("MangKyu");
  }

}
```

위와 같이 작성한 테스트 코드는 어떻게 되겠는가? 테스트 코드가 Spring과 같은 DI 프레임워크 위에서 동작하지 않으므로 의존 관계 주입이 되지 않을 것이고, userRepository가 null이 되어 userRepository의 add 호출 시 NPE가 발생할 것이다. 이를 해결하기 위해 Setter를 사용하면 OCP(Open-Closed Principal, 개방-폐쇄의 법칙)를 위반하게 된다.
반면에 생성자 주입을 사용하면 컴파일 시점에 객체를 주입받아 테스트 코드를 작성할 수 있으며, 주입하는 객체가 누락된 경우 컴파일 시점에 오류를 발견할 수 있다. 심지어 우리가 테스트를 위해 만든 Test객체를 생성자로 넣어 편리함을 얻을 수도 있다.

3. final 키워드 작성 및 Lombok과의 결합
생성자 주입을 사용하면 필드 객체에 final 키워드를 사용할 수 있으며, 컴파일 시점에 누락된 의존성을 확인할 수 있다. 반면에 생성자 주입을 제외한 다른 주입 방법들은 객체의 생성(생성자 호출) 이후에 호출되므로 final 키워드를 사용할 수 없다.
또한 final 키워드를 붙임으로써 Lombok과 결합되어 코드를 간결하게 작성할 수 있다. Lombok에는 final 변수를 위한 생성자를 대신 생성해주는 @RequiredArgsConstructor를 여기 에서 살펴보았다.
Spring과 같은 DI 프레임워크는 Lombok과 환상적인 궁합을 보여주는데, 위에서 작성했던 생성자 주입 코드를 Lombok과 결합시키면 다음과 같이 간편하게 작성할 수 있다.

```
@Service
@RequiredArgsConstructor
public class UserServiceImpl implements UserService {

  private final UserRepository userRepository;
  private final MemberService memberService;

  @Override
  public void register(String name) {
    userRepository.add(name);
  }

}
```

이러한 코드가 가능한 이유는 앞서 설명하였듯 Spring에서는 생성자가 1개인 경우 @Autowired를 생략할 수 있도록 도와주고 있으며, 해당 생성자를 Lombok으로 구현하였기 때문이다.

4. 순환 참조 에러 방지
애플리케이션 구동 시점(객체의 생성 시점)에 순환 참조 에러를 방지할 수 있다.
예를 들어 UserServiceImpl의 register 함수가 memberService의 add를 호출하고, memberServiceImpl의 add함수가 UserServiceImpl의 register 함수를 호출한다면 어떻게 되겠는가?

```
@Service
public class UserServiceImpl implements UserService {

  @Autowired
  private MemberServiceImpl memberService;

  @Override
  public void register(String name) {
    memberService.add(name);
  }

}
```

```
@Service
public class MemberServiceImpl extends MemberService {

  @Autowired
  private UserServiceImpl userService;

  public void add(String name){
    userService.register(name);
  }

}
```

위의 두 메소드는 서로를 계속 호출할 것이고, 메모리에 함수의 CallStack이 계속 쌓여 StackOverflow 에러가 발생하게 된다.

```
Caused by: java.lang.StackOverflowError: null
    at com.mang.example.user.MemberServiceImpl.add(MemberServiceImpl.java:20) ~[main/:na]
    at com.mang.example.user.UserServiceImpl.register(UserServiceImpl.java:14) ~[main/:na]
    at com.mang.example.user.MemberServiceImpl.add(MemberServiceImpl.java:20) ~[main/:na]
    at com.mang.example.user.UserServiceImpl.register(UserServiceImpl.java:14) ~[main/:na]
```

만약 이러한 문제를 발견하지 못하고 서버가 운영된다면 어떻게 되겠는가? 해당 메소드의 호출 시에 StackOverflow 에러에 의해 서버가 죽게 될 것이다.
하지만 생성자 주입을 이용하면 이러한 순환 참조 문제를 방지할 수 있다.

```
Description:

The dependencies of some of the beans in the application context form a cycle:

┌─────┐
|  memberServiceImpl defined in file [C:₩Users₩Mang₩IdeaProjects₩build₩classes₩java₩main₩com₩mang₩example₩user₩MemberServiceImpl.class]
↑     ↓
|  userServiceImpl defined in file [C:₩Users₩Mang₩IdeaProjects₩build₩classes₩java₩main₩com₩mang₩example₩user₩UserServiceImpl.class]
└─────┘
```

애플리케이션 구동 시점(객체의 생성 시점)에 에러가 발생하기 때문이다. 그러한 이유는 Bean에 등록하기 위해 객체를 생성하는 과정에서 다음과 같이 순환 참조가 발생하기 때문이다.

```
new UserServiceImpl(new MemberServiceImpl(new UserServiceImpl(new MemberServiceImpl(()...)))
```

[요약 정리]

- OCP 원칙을 지키며 객체의 불변성을 확보할 수 있다.
- 테스트 코드의 작성이 용이해진다.
- final 키워드를 사용할 수 있고, Lombok과의 결합을 통해 코드를 간결하게 작성할 수 있다.
- 순환 참조 문제를 를 애플리케이션 구동(객체의 생성) 시점에 파악하여 방지할 수 있다.

이러한 이유들로 우리는 DI 프레임워크를 사용하는 경우, 생성자 주입을 사용하는 것이 좋다.