객체 지향 캡슐화

Tags	
스터디 일자	@2023/11/30

이번주차 주제 객체 지향

객체 지향의 특징 중 *캡슐화 에 대해 알아보자 ~!*

🤔 먼저, 캡슐화가 뭔가요?

👉 *나무위키 피셜*

캡슐화(encapsulation)는 객체 지향 프로그래밍에서 다음 2가지 측면이 있다:

객체의 **속성**(data fields)과 **행위**(메서드, methods)를 **하나로 묶고**,

실제 구현 내용 일부를 **외부에 감추어 은닉한다.**

> *우리는 **캡슐** 약 내부에 무엇이 들어있는지 알 수 없다.*
>
> 그런데 분명 캡슐 약 안에는 **여러가지 성분들**이 들어있는 것을 알고,
> 캡슐의 **용도** 또한 알고 있다. 약 먹으면 몸이 나아지는 것!!
>
> 그런데 알약 껍질에 의해 **물리적으로 접근할 수 없기 때문**에
> 캡슐에 무엇이 들어있는지 우리는 모른다.

자바의 캡슐화

자바의 캡슐화는 알약 껍질처럼 외부 접근에 대한 차단이고, 차단의 방법은 **접근 제어자**를 사용한다.

public	접근 제한 x
protected	동일 패키지 + 상속관계에 있는 클래스만 접근 가능
defualt	동일 패키지 내에서만 접근 가능
private	동일 클래스 내에서만 접근 가능

간단한 소스코드로 알아보자.

```
class Capsule {
    int number;

    public Capsule(int number) {
        this.number = number;
    }

    public double getHalf() {
        return number / 2;
    }
}
```

```
class Main {
    public static void main(String[] args) {
        Capsule capsule = new Capsule(10);
        System.out.println(capsule.getHalf());
    }
}
```

👉 `int`값을 초기값으로 갖는 객체가 있고, 그 값의 절반을 반환하는 `getHalf()` 라는 메소드가 존재한다.

❓ 여기서 **캡슐화**는 바로 **Capsule 클래스** 자체를 의미한다.

위 코드가 캡슐화의 정의를 만족하는지 보자.

객체 지향 캡슐화 :
데이터와, 데이터를 처리하는 행위를 묶고, 외부에는 그 행위를 보여주지 않는 것.

Capsule 클래스는

- `int` 값의 데이터를 가지고 있다.
- 그리고 `getHalf` 라는 데이터를 처리하는 행위 또한 가지고 있다.
- 마지막으로 Main메소드의 입장에서 Capsule 클래스의 `getHalf()` 를 사용할 수는 있지만, 구현이 어떻게 되어 있는지는 알 수 없다.

즉, 캡슐화 되어 있다고 말할 수 있다!

여기서 드는 의문은?

데이터와, 행위를 하나로 묶고, 그걸 외부에 노출 시키지 않는 캡슐화를 왜 하는가?

잘 된 캡슐화를 통해 얻을 수 있는 이점 중 가장 큰 것은

- **코드의 중복을 피할 수 있다는 점**
- **데이터를 처리하는 동작 방식을 외부에서 알 필요가 없다는 점**

> 이 이점들이 왜 필요한지 코드로 알아보자!

- 아래와 같은 제품 클래스가 있다.

```
class Product {
    int price = 10000;

    public Product(int price) {
        this.price = price;
    }
}
```

- 만약 어떤 제품의 10% 할인된 금액을 구해서 다른 로직으로 넘겨야 한다면 다음과 같은 코드를 추가하게 될 것이다.

```
class Product {
    int price = 10000;

    public Product(int price) {
        this.price = price;
    }

    public int getPrice() { // getter 추가
        return price;
    }
}
```

```
public void first(Product product) {
    double discountedPrice = product.getPrice() * 0.9;
    show(discountedPrice);
}
```

상품의 가격을 가지고 와서 10프로 할인된 가격을 구하고, 다른 로직으로 넘겼다.

- 그렇다면, 만약 10프로 할인된 금액을 다른 로직에서도 사용하게 된다면 어떻게 될까?

```
public void first(Product product) {
    double discountedPrice = product.getPrice() * 0.9; // 중복
    show(discountedPrice);
}

public void second(Product product) {
    double discountedPrice = product.getPrice() * 0.9;  // 중복
    secondShow(discountedPrice); // 할인된 금액을 show와 다르게 사용하는 메서드
}
```

코드에서 중복이 일어났다.

한 줄짜리 코드 중복 좀 되면 어때??? 할 수 있지만, 이러한 로직이 수십개,, 수백개 더 필요하면 일일이 타이핑하는 것은 정말 힘들 것이다.

🤔 코드의 중복은 좀 별로군?

> 다음으로는 **데이터를 처리하는 방식이 외부에 드러나지 않는 것**은 어떤 이점이 있는지 알아보자.

- 요구사항이 변경되어 10프로 할인된 금액이 아니라 20프로 할인된 금액으로 로직을 바꿔야 하는 상황이다.
- 위와 같이 코드가 작성되었다면, 엄청난 양의 코드를 아래와 같이 고쳐야 한다.

```
public void first(Product product) {
    double discountedPrice = product.getPrice() * 0.8; // 20프로 할인
    show(discountedPrice);
}
```

- 그러다가 10프로 할인 로직을 20프로 할인로직으로 변경하지 못한 코드가 하나라도 존재한다면, 서비스에 큰 타격을 입게 될 것이다.

🤔 **캡슐화**를 지키기 위한 규칙중에는 *Tell, Don't Ask* 라는 원칙이 있다.

객체 내부의 데이터를 꺼내와서 처리하는게 아닌, 객체에게 처리할 **행위를 요청**하라는 행위이다.
이러한 행위를 우리는 **"객체에 메세지를 보낸다"** 라고 말한다.

- 데이터를 객체로부터 받아오는 것이 아닌, 객체에게 처리를 요청하는 방식의 코드를 작성해보자.

```
class Product {
    int price = 10000;

    public Product(int price) {
        this.price = price;
    }

    public int getDiscountedPrice() { // 할인된 금액을 반환하도록
```

```
        return price * 0.9;
    }
}
```

```
public void first(Product product) {
    double discountedPrice = getDiscountedPrice(); // 할인된 금액을 알려줘!
    show(discountedPrice);
}
```

이렇게 하면 위에서 생겼던 문제들이 다 해결된다.

- 할인된 금액을 사용하는 로직이 수백개가 있고, 요구사항이 20프로를 할인하도록 변경됐다면?

```
public int getDiscountedPrice() {
    return price * 0.8; // 20프로 할인
}
```

Product 클래스의 `getDiscountedPrice()` 로직만 수정하면 끝이다.

이게 캡슐화다 ^-^

캡슐화를 잘 해봅시다!