DTO와 VO

LichKing 2017. 7. 1. 15:27

MVC 구조로 개발을 하게되면 공통적으로 생성되는 자바파일들이 있다.

Controller, Service, DAO, DTO가 그것인데 여기서 DTO는 VO라는 명칭으로도 많이 사용된다.

같은 뜻이라면 애초에 2가지 용어가 존재하지 않을텐데, 이 두 용어의 차이를 포스팅하고자한다.

1. DTO(Data Transfer Object)

Data Transfer Object 의 약자로 레이어를 이동할때 데이터를 들고있는 객체를 말한다. 객체지향적인 요소는 없이 단순히 데이터만 들고 이동시키기때문에 필드를 public 접근제어자를 사용해 직접 접근하도록 하는 경우도 있다. 자바에서는 뭔가를 하려면 일단 객체를 만들어야되니 객체화해서 사용할뿐 실질적으로 객체지향이라는 거리가 있는 객체.

DTO와 VO의 혼용은 사실상 DTO를 칭하는 경우가 많다.

2. VO(Value Object)

DTO는 원래 DTO처럼 사용하고있던거니 크게 설명할 부분이 없었다. VO가 뭔지 잘 모르기때문에 DTO와 혼용이 있었던것 같다. VO가 뭔지 살펴보자.

2-1. 참조(Reference)와 가변(Mutable)

```
class Money {
   private long amount;
   private Currency currency;

   public long getAmount() {
      return amount;
   }

   public void setAmount(long amount) {
      this.amount = amount;
   }

   public Currency getCurrency() {
      return currency;
   }

   public void setCurrency(Currency currency) {
      this.currency = currency;
   }

   @Override
   public String toString() {
      return this.amount + this.currency.getDescription();
   }
}

enum Currency {
   KRW("원"), DOLLAR("달러");

   private final String description;

   Currency(String description) {
      this.description = description;
   }

   public String getDescription() {
      return this.description;
   }
}
```

아주 일반적인 코드다. 크게 어려울건없을것이다. 100원짜리 Money 객체를 만들어보자.

```
Money money = new Money();
money.setAmount(100);
money.setCurrency(Currency.KRW);

System.out.println(money.toString());
```

이 역시 크게 어려울건없다. 객체를 생성한 후 적절한 값을 setter를 이용해 넣어줬다.

setter는 여러번 사용할 수 있으며, 이미 만들어진 Money 객체는 얼마든지 변경할 수 있다. amount 값으로 100을 넣었다가 500으로 다시 변경하는것도 가능하다.

** 이런식으로 내부 값을 변경할 수 있는 객체를 가변객체라고 표현한다(Mutable Object).*

```
Money money = new Money();
money.setAmount(100);
money.setCurrency(Currency.KRW);

System.out.println(money.toString());

money.setAmount(500);

System.out.println(money.toString());
```

가변객체는 참조이동으로 인한 예상치못한 상황이 발생할 수도 있다.

```
Money money1 = new Money();
money1.setAmount(100);
money1.setCurrency(Currency.KRW);
```

```
Money money2 = money1;
money2.setCurrency(Currency.DOLLAR);

System.out.println(money1);
System.out.println(money2);
```

자바를 처음 시작하는분들이 가장 많이 헷갈려하고, 심심찮게 각종 커뮤니티에 질문글로 올라오는 내용이기도 하다.

money1은 원화로 값을 넣어놨고 money2는 달러로 값을 넣었다. money1의 출력이 100원을 리턴하길 바라지만 실제 출력은 그렇지 않다. 지금이야 어디가 문제고 뭐가 문제인지 바로 알 수 있지만 실제 프로젝트에서는 이로인한 디버깅이 만만치않다. 얼마전 내가 재직중인 회사에서도 이와 유사한 문제가 발생한 적이 있다. 이런 문제를 방지하기 위해선 어떻게 해야할까

2-2. Copy

객체를 복사하자. 복사하는 방법중 하나는 Cloneable 인터페이스를 구현하는 것이다.

```
class Money implements Cloneable {

   @Override
   public Money clone() {
      try {
         return (Money) super.clone();
      } catch (CloneNotSupportedException e) {
         throw new IllegalStateException("copy fail");
      }
   }
}
```

(불필요한 코드는 감췄다. 삭제한게 아니다.)

```
Money money1 = new Money();
money1.setAmount(100);
money1.setCurrency(Currency.KRW);

Money money2 = money1.clone();
money2.setCurrency(Currency.DOLLAR);

System.out.println(money1);
System.out.println(money2);
```

Cloneable을 구현한 뒤 clone() 메서드 호출로 객체를 복사했다. 이제 원하는 결과가 나온다.

Cloneable 인터페이스가 신기할수도 있겠지만 복사는 크게 어려운게 아니다. 단순하게 아래처럼 구현해줘도 된다.

```
class Money {

   public Money copy(){
      Money copy = new Money();
      copy.currency = this.currency;
      copy.amount = this.amount;

      return copy;
   }
}
```

```
Money money1 = new Money();
money1.setAmount(100);
money1.setCurrency(Currency.KRW);

Money money2 = money1.copy();
money2.setCurrency(Currency.DOLLAR);

System.out.println(money1);
System.out.println(money2);
```

copy() 메서드를 구현한 뒤 copy() 메서드를 이용해 결과를 확인해도 clone()과 같음을 알 수 있다. Cloneable 인터페이스를 사용하는게 더 있어보이는 방법이긴하지만 개인적으론 후자를 더 좋아하고, 사용한다.

** Effective Java Item 11. clone을 재정의 할 때는 신중하라.*

그럼 이제 clone()이나 copy() 메서드를 사용하면 끝날까? 하지만 새로 들어온 개발자가 저런게 구현되어있음을 모르고, 혹은 신입이라 참조이동으로 발생하는 끔찍한 상황을 모르고 여전히 그냥 참조 이동만 한다면? 문제방지책은 마련된 셈이지만 그럼에도 여전히 문제는 발생할 수 있다.

2-3. 불변(Immutable)

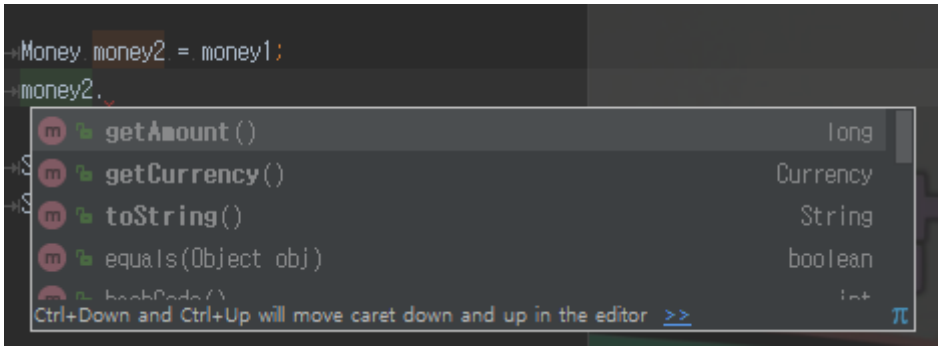
우리는 참조가 이동하는게 문제라고 생각하여, 참조 이동시 그냥 참조 이동만 하지말고 객체를 복사하게했다. 하지만 그럼에도 여전히 문제는 발생한다. 문제의 핵심을 다른쪽으로 생각해보자. *참조이동을 막을 수 없다면 값의 변경을 막으면 어떨까?* 그럼 이건 어떻게 막을 수 있을까?

```
class Money {
   private long amount;
   private Currency currency;

   public Money(long amount, Currency currency) {
      this.amount = amount;
      this.currency = currency;
   }

   public long getAmount() {
      return amount;
   }

   public Currency getCurrency() {
      return currency;
```

```
    }

    @Override
    public String toString() {
        return this.amount + this.currency.getDescription();
    }
}
```

```
Money money1 = new Money(100, Currency.KRW);

Money money2 = money1;

System.out.println(money1);
System.out.println(money2);
```

이 코드는 같은 결과를 출력한다. 그도 그럴것이 money2를 변경하지않기때문이다. money2를 변경하고도 값이 서로 달라지는지 확인해보자.



헉, 변경하고싶은데 변경할 메서드가 없다. 위에 올린 Money 코드를 확인해보자. 값은 생성자를 통해서만 받고 setter가 없다. 값을 변경할 수가 없어진것이다. 값을 변경하고싶으면 새로운 객체를 만드는 수밖에 없다.

```
Money money1 = new Money(100, Currency.KRW);

Money money2 = new Money(500, Currency.DOLLAR);

System.out.println(money1);
System.out.println(money2);
```

이제 어디서 넘어온건지, 복사가 된건지, 새로생성된건지도 모를 객체의 값을 바꿀일은 없어졌다. 무조건 새로 객체를 만들어야하므로 새 값을 넣을일이 생기면 객체를 만들어줘야한다.

이렇게 *내부 값을 변경하지못하는 객체를 불변객체(Immutable Object)*라고 한다.

2-4. VO(Value Object)

하지만 객체지향언어에서 객체는 상태와 행위를 가지고 있는것을 뜻한다. 그 중 상태를 못바꾸게 모든 객체의 setter를 막아버리면 객체지향언어를 쓰는 이유가 없어지는것 아닌가? 그럼 어떤 객체를 상태를 갖게하고 어떤 객체를 불변으로 해야할까?

잘 생각해보자. '사람'이라는 클래스가 있다고 가정하자. '사람'이라는 클래스를 이용해 '나'라는 인스턴스를 생성했다. '나'는 상태가 변하게된다. 나라는 사람은 1년뒤에 1살 더 먹게된다. 그렇다고 내가 1살 더 먹을때마다 날 새로 만들진 않는다. 이런경우에는 가변객체를 사용한다.

실제 500원짜리 동전을 생각해보자. 500원짜리 동전은 상태라는게 없다. 시간이 지난다고 500원이 100원되지는 않는다. 녹이 슬거나 기스는 날 지언정 500원짜리 동전이 갖는 핵심 가치는 변하지않는다. 100원짜리를 갖고싶으면 새로 만드는수밖에 없다.

이제 다시 VO로 돌아오자. VO는 Value Object의 준말이다. 상태가 있는 객체가 아니라 그저 값으로 취급하는 객체를 값 객체라고 표현한다. 위 예제코드에서 Money 객체가 바로 값 객체인것이다. 이런 값 객체를 Value Object라고 표현하고(실제 그냥 뜻도 값객체다.), 불변으로 만드는게 VO인 것이다.

3. 정리

정리해보면 DTO는 그저 값을 전달하는 객체다. get(), set()만 구현되어있고, 필요에 따라 이걸 굳이 구현할 필요도 없다. 객체지향적 관점에서보면 사실 굉장히 맘에 안드는 형태의 객체다.

VO는 값으로 취급할 객체다. 값으로 취급하는 객체는 상태가 필요없기 때문에 불필요한 상태변경 행위를 구현하지않는다. 대표적인 VO로는 String이 있다. 이외 기본형 래핑 클래스들도 모두 불변의 VO이다(Integer, Long...).

포스팅 제목은 DTO와 VO인데 내용은 VO에 대한 설명이 대부분을 차지한다. 아마도 VO라는 이름으로 DTO형태의 클래스를 만드는일이 많기때문일것이다. 이제 잘 구분해서 사용하자.