10-12 [Java]

소유자	종수 김
태그	

불변 객체.

기본형과 참조형의 공유

자바의 데이터 타입을 가장 크게보면 '기본형(Primitive Type)', '참조형(Reference Type)' 2가지.

- 기본형 : 하나의 값을 여러 변수에서 절대로 공유하지 않음. - 변수에 값을 복사해서 대입

```
package lang.immutable.address;

public class PrimitiveMain {
    public static void main(String[] args) {
        //기본형은 절대로 같은 값을 공유하지 않음.
        int a = 10;
        int b = a; // a -> b 값 복사 후 대입.

        System.out.println(a);
        System.out.println(b);

        b = 20;
        System.out.println("20 -> b");
        System.out.println(a);
        System.out.println(b);
    }
}

// 10
// 10
// 20 -> b
```

```
// 10
// 20
```

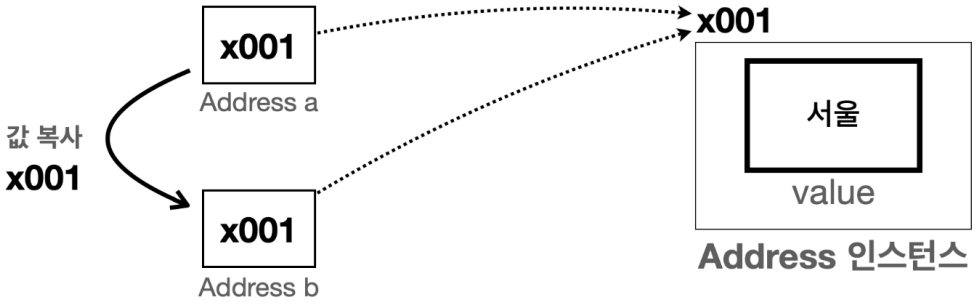
- 기본형 변수 `a`와 `b`는 절대로 하나의 값을 공유하지 않는다.
- `b = a`라고 하면 **자바는 항상 값을 복사해서 대입**한다. 이 경우 `a`에 있는 값 `10`을 복사해서 `b`에 전달한다.
- 결과적으로 `a`와 `b`는 둘다 `10`이라는 똑같은 숫자의 값을 가진다. 하지만 `a`가 가지는 `10`과 `b`가 가지는 `10`은 복사된 완전히 다른 `10`이다. 메모리 상에서도 `a`에 속하는 `10`과 `b`에 속하는 `10`이 각각 별도로 존재한다.

- 참조형 : 하나의 객체를 참조 값을 통해 여러 변수에서 공유할 수 있다. - 참조 값 자체를 대입

```
package lang.immutable.address;

public class RefMain1 {
    public static void main(String[] args) {
        //참조형 변수는 하나의 인스턴스를 공유할 수 있다.
        Address a = new Address("서울");
        Address b = a;

        System.out.println(a);
        System.out.println(b);

        b.setValue("부산"); // b의 값을 부산으로 변경
        System.out.println(a);
        System.out.println(b);
    }
}

//서울
//서울
//부산
//부산
```

순서대로 코드를 분석해보자.

```
Address a = new Address("서울");
Address b = a;
```



- 참조형 변수들은 같은 참조값을 통해 같은 인스턴스를 참조할 수 있다.
- `b = a`라고 하면 `a`에 있는 참조값 `x001`을 복사해서 `b`에 전달한다.
  - 자바에서 모든 값 대입은 변수가 가지고 있는 값을 복사해서 전달한다. 변수가 `int` 같은 숫자값을 가지고 있으면 숫자값을 복사해서 전달하고, 참조값을 가지고 있으면 참조값을 복사해서 전달한다.
- 참조값을 복사해서 전달하므로 결과적으로 `a`, `b`는 같은 `x001` 인스턴스를 참조한다.
- 기본형 변수는 절대로 같은 값을 공유하지 않는다.
  - 예) `a=10`, `b=10`과 같이 같은 모양의 숫자 `10`이라는 값을 가질 수는 있지만 같은 값을 공유하는 것은 아니다. 서로 다른 숫자 `10`이 두 개 있는 것이다.

- 참조형 변수는 참조값을 통해 같은 객체(인스턴스)를 공유할 수 있다.

여기서 `b`의 주소만 부산으로 변경했는데, `a`의 주소도 함께 부산으로 변경되어 버린 이유는 무엇일까? 메모리 구조를 보면 바로 답이 나오겠지만, 개발을 하다 보면 누구나 이런 실수할 수 있을 것 같다는 생각도 함께 들 것이다.

- 자바에서 모든 값 대입은 변수가 가지고 있는 값을 복사해서 전달.
- 즉, 변수가 int 같은 숫자값을 가지고 있으면 숫자값을 복사해서 전달, 참조값을 가지고 있으면 참조값을 복사해서 전달.

공유 참조와 사이드 이펙트

- 사이드 이펙트(side effect) : 프로그래밍에서 어떤 계산이 주된 작업 외에 추가적인 부수 효과를 일으키는 것.

```
b.setValue("부산"); // b의 값을 부산으로 변경
```

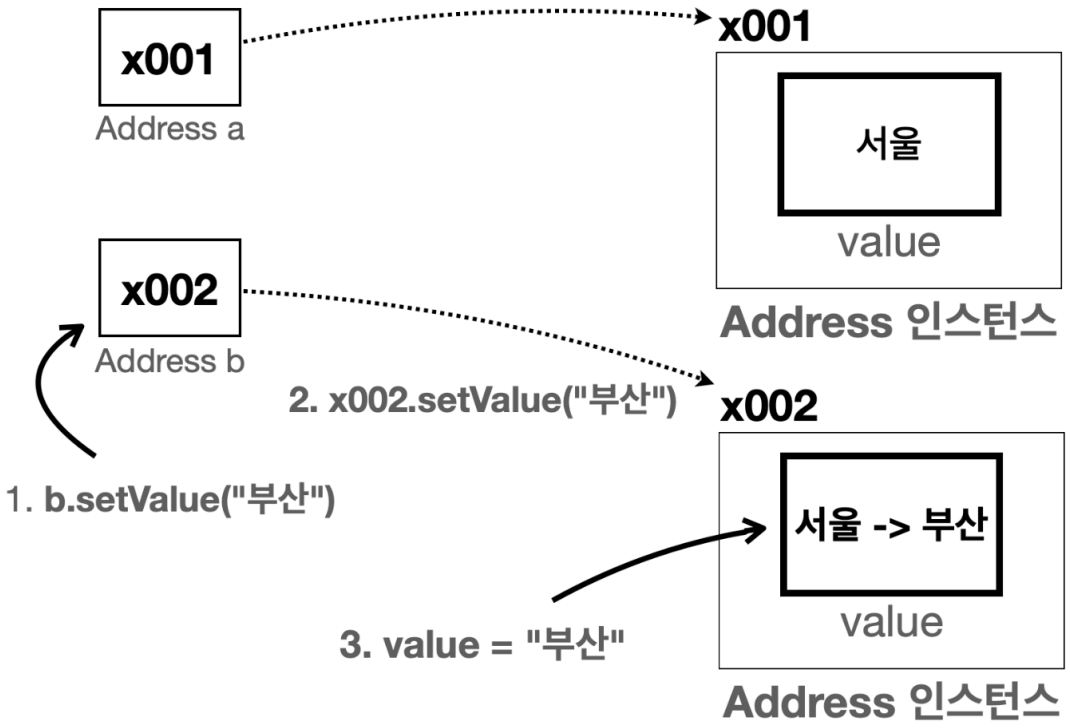
b의 값 만을 부산으로 변경하고 싶었으나, a와 b 모두 같은 인스턴스를 바라보고 있었기에 a 또한 변경됨.

위와 같이 개발자가 의도하지 않은 특정 부분에서 변경이 의도치 않게 다른 부분에 영향을 미치는 것을 사이드 이펙트라고 함.

사이드 이펙트 해결 방안

a와 b가 다른 인스턴스를 참조하게 하면 됨.

```
package lang.immutable.address;

public class RefMain2 {
    public static void main(String[] args) {
        //참조형 변수는 하나의 인스턴스를 공유할 수 있다.
        Address a = new Address("서울"); // x001
        Address b = new Address("서울"); // x002

        System.out.println(a);
        System.out.println(b);

        b.setValue("부산"); // b의 값을 부산으로 변경
        System.out.println(a);
        System.out.println(b);

        a.setValue("대구");
        System.out.println(a);
        System.out.println(b);
    }
}

// 서울
// 서울
// 서울
// 부산
// 대구
// 부산
```



- `a`와 `b`는 서로 다른 인스턴스를 참조한다. 따라서 `b`가 참조하는 인스턴스의 값을 변경해도 `a`에는 영향을 주지 않는다.

여러 변수가 하나의 객체를 공유하는 것을 막을 방법이 없음.

같은 객체를 공유하지 않으면 위와 같은 문제는 발생하지 않음. 하지만, 하나의 객체를 여러 변수가 공유하지 않도록 강제로 막을 수 있는 방법이 없음.

객체를 공유하는 문법이 자바 문법상 아무 오류가 없고, 실제로 객체를 공유해야 하는 상황이 있을 수도 있기 때문.

```
package lang.immutable.address;

public class RefMain3 {
    public static void main(String[] args) {
        Address a = new Address("서울");
        Address b = a;

        System.out.println("a = " + a);
        System.out.println("b = " + b);
```

```
        change(b, "부산");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }

    private static void change(Address address, String changeA
        System.out.println("주소 값을 변경합니다 -> " + changeAdd
        address.setValue(changeAddress);
    }
}
```

위와 같이 메서드가 분리되어있을 때 해당 문제를 추적하기가 더 어려움.

그럼 공유 참조를 예방하기 위한 방법이 아예 없을까 ?

불변 객체

공유하면 안되는 객체를 여러 변수에서 공유했기 때문에 발생한 문제이고, 객체의 공유를 막을 수 있는 방법이 없음.

하지만, 해당 문제의 더 근본적인 원인은 객체를 공유한 것이 아닌, 공유된 객체의 값을 변경했기 때문에 발생한 문제.

즉, b가 공유 참조하는 인스턴스의 값을 '변경'하기 때문에 생기는 문제.
만약, Address 객체의 값을 변경하지 못하게 설계했다면 이런 문제가 발생하지 않았을 것.

불변 객체 도입

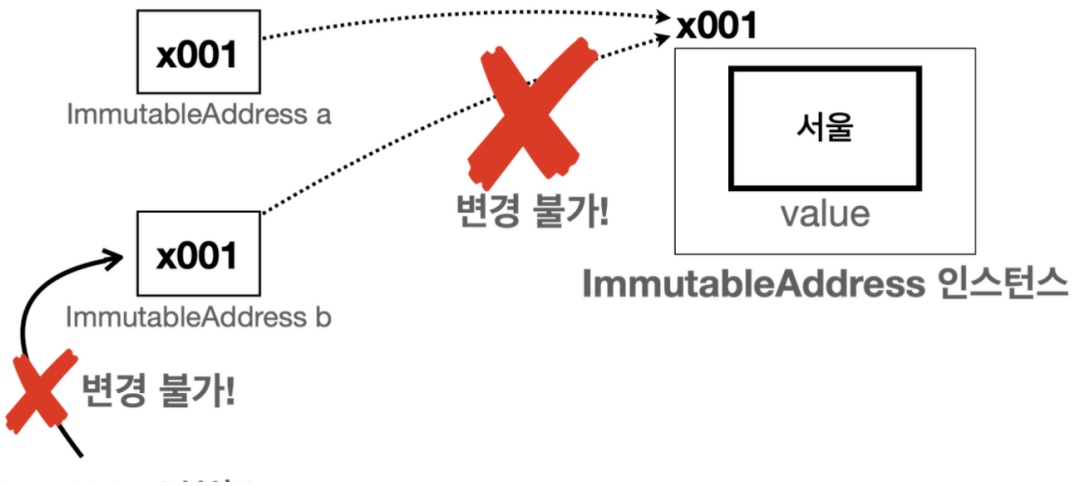
객체의 상태(객체 내부의 값, 필드, 멤버 변수)가 변하지 않는 객체를 불변 객체(Immutable Object)라고 함.

```
package lang.immutable.address;

public class ImmutableAddress {
    private final String value;

    public ImmutableAddress(String address) {
        this.value = address;
    }

    public String getValue() {
```

```
        return value;
    }

    @Override
    public String toString() {
        return "ImmutableAddress{" +
                "address='" + value + '\'' +
                '}';
    }
}

package lang.immutable.address;

public class RefMain2 {
    public static void main(String[] args) {
        //참조형 변수는 하나의 인스턴스를 공유할 수 있다.
        ImmutableAddress a = new ImmutableAddress("서울");
        ImmutableAddress b = a;

        System.out.println(a);
        System.out.println(b);

//        b.setValue("부산"); // Setter가 존재하지않을 뿐 더러, fi
              b = new ImmutableAddress("부산"); // 불변 객체이
        System.out.println(a);
        System.out.println(b);

    }
}

// 서울
// 서울
// 부산
// 서울
```

final 키워드를 사용하여, 변경되지 않는 것을 컴파일 단계에서 보장.
setter를 두지 않는 것으로도 같은 효과를 줄 수 있지만, 후에 생기지 않는다는 것을 보장해 줄 수는 없음.

final키워드를 통해서 의도를 명확히 하며, 후에 setter가 생기지 않는다는 것을 컴파일 단계에서 보장.



- 참고
  - 가변(Mutable) 객체 / 불변(Immutable) 객체
    가변 이름 그대로 처음 만든 이후 상태가 변할 수 있음.
    불변 이름 그대로 처음 만든 이후 상태가 변할 수 없음.
  - Address는 가변, ImmutableAddress 불변

불변 객체의 값 변경

불변 객체를 사용하지만 그래도 값을 변경해야 하는 메서드가 필요하다면 ?

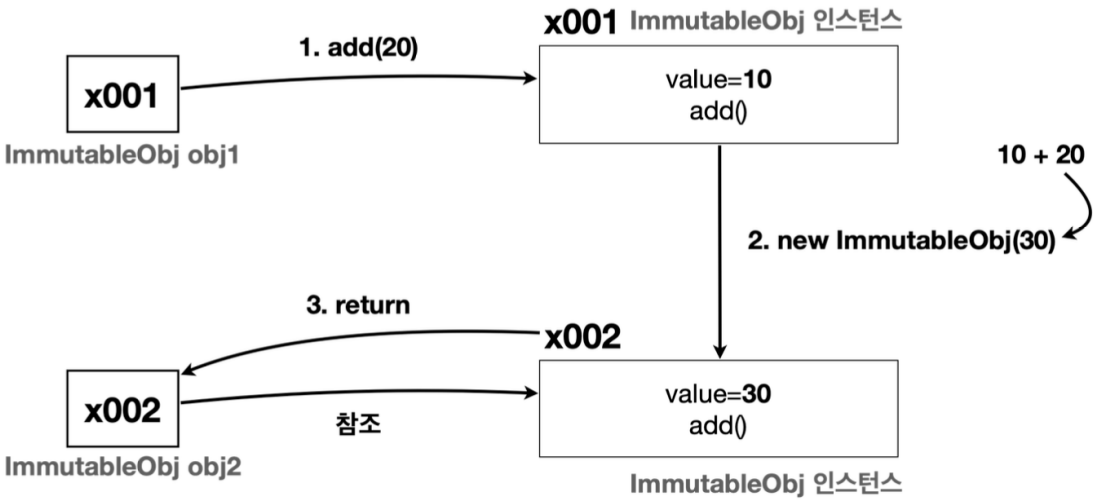
Ex) 기존 값에 새로운 값을 더하는 add() 메서드

```
package lang.immutable.change;


import lang.immutable.address.ImmutableAddress;

public class ImmutableObj {
    private final int value;

    public ImmutableObj(int value) {
        this.value = value;
```

```
    }

    public int getValue() {
        return value;
    }

    public ImmutableObj add(int addValue) {
        int result = value + addValue;
        return new ImmutableObj(result);
    }
}

package lang.immutable.change;

public class ImmutableMain1 {
    public static void main(String[] args) {
        ImmutableObj obj1 = new ImmutableObj(10);
        ImmutableObj obj2 = obj1.add(20);

        //계산 이후 기존값과 신규 값 모두 확인 가능
        System.out.println("obj1.getValue() = " + obj1.getVal
        System.out.println("obj2.getValue() = " + obj2.getVal

    }
}

// 10
// 30
```

- 여기서 핵심은 `add()` 메서드이다.
- 불변 객체는 값을 변경하면 안된다! 그러면 이미 불변 객체가 아니다!
- 하지만 여기서는 기존 값에 새로운 값을 더해야 한다.
- 불변 객체는 기존 값은 변경하지 않고 대신에 계산 결과를 바탕으로 새로운 객체를 만들어서 반환한다.
- 이렇게 하면 불변도 유지하면서 새로운 결과도 만들 수 있다.

실행 순서를 메모리 구조로 확인해보자.



1. `add(20)`을 호출한다.
2. 기존 객체에 있는 `10`과 인수로 입력한 `20`을 더한다. 이때 기존 객체의 값을 변경하면 안되므로 계산 결과를 기반으로 새로운 객체를 만들어서 반환한다.
3. 새로운 객체는 `x002` 참조를 가진다. 새로운 객체의 참조값을 `obj2`에 대입한다.