# 2. 불변 객체

#1.인강/0.자바/3.자바-중급1편

- /기본형과 참조형의 공유
- /공유 참조와 사이드 이펙트
- /불변 객체 - 도입
- /불변 객체 - 예제
- /불변 객체 - 값 변경
- /문제와 풀이
- /정리

## 기본형과 참조형의 공유

자바의 데이터 타입을 가장 크게 보면 기본형(Primitive Type)과 참조형(Reference Type)으로 나눌 수 있다.

- **기본형**: 하나의 값을 여러 변수에서 절대로 공유하지 않는다.
- **참조형**: 하나의 객체를 참조값을 통해 여러 변수에서 공유할 수 있다.

하나의 값을 공유하거나 또는 공유하지 않는 다는 것이 무슨 뜻인지 예제를 통해 알아보자.

### 기본형 예제

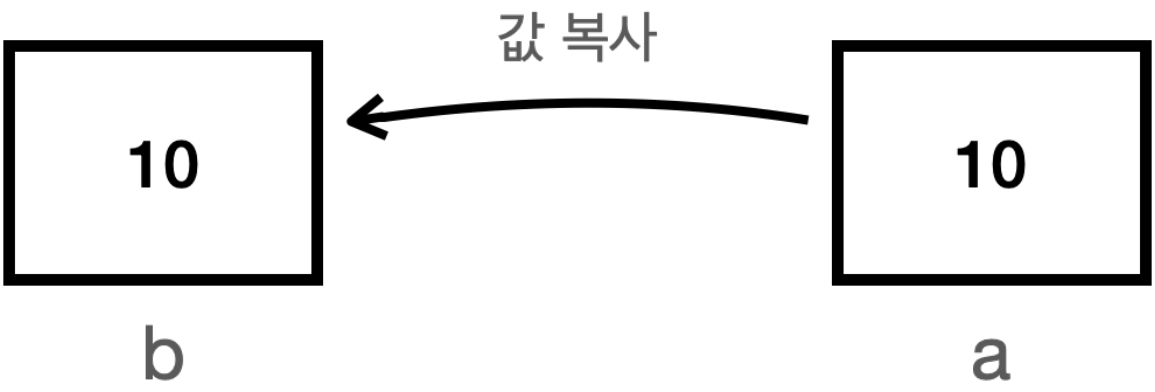
기본형은 하나의 값을 여러 변수에서 절대로 공유하지 않는다. 다음 예를 보자.

```
package lang.immutable.address;

public class PrimitiveMain {

    public static void main(String[] args) {
        //기본형은 절대로 같은 값을 공유하지 않는다.
        int a = 10;
        int b = a; // a -> b, 값 복사 후 대입
        System.out.println("a = " + a);
        System.out.println("b = " + b);

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

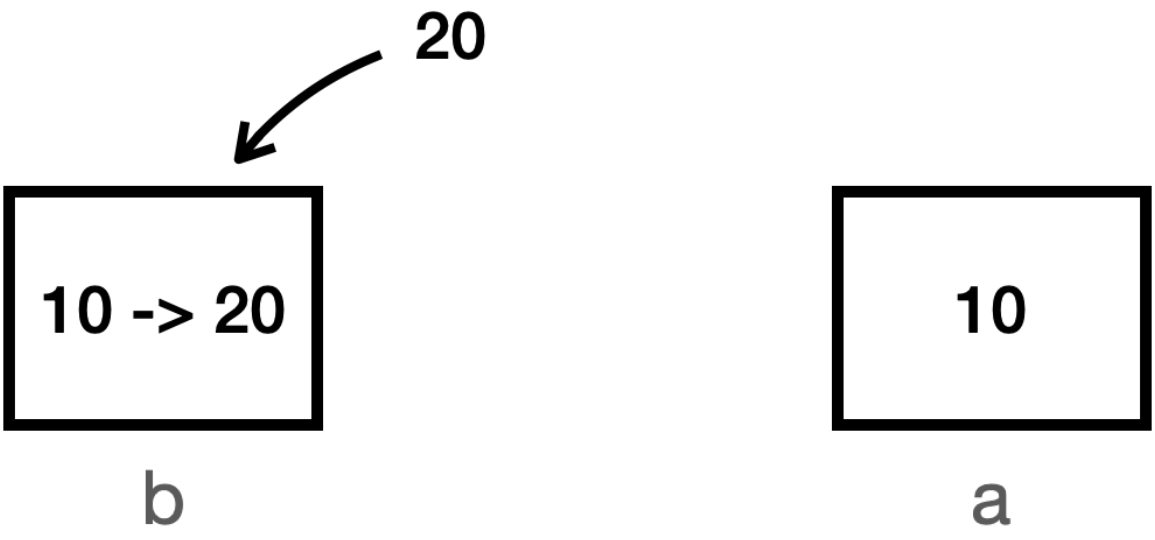
```
}
```

**실행 결과**

```
a = 10
b = 10
20 -> b
a = 10
b = 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.




- `b = 20`이라고 하면 `b`의 값만 `20`으로 변경된다.
- `a`의 값은 `10`으로 그대로 유지된다.
- 기본형 변수는 하나의 값을 절대로 공유하지 않는다. 따라서 값을 변경해도 변수 하나의 값만 변경된다. 여기서는 변수 `b`의 값만 `20`으로 변경되었다.

너무 당연한 이야기이다. 그렇다면 이번에는 참조형 예제를 보자.

## 참조형 예제

```
package lang.immutable.address;

public class Address {

    private String value;

    public Address(String value) {
        this.value = value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }

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

- 단순히 주소를 보관하는 객체이다.
- 객체의 값을 편하게 확인하기 위해 IDE의 도움을 받아서 `toString()`을 재정의하자

```
package lang.immutable.address;

public class RefMain1_1 {

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

```
        b.setValue("부산"); //b의 값을 부산으로 변경해야함
        System.out.println("부산 -> b");
        System.out.println("a = " + a); //사이드 이펙트 발생
        System.out.println("b = " + b);
    }
}
```

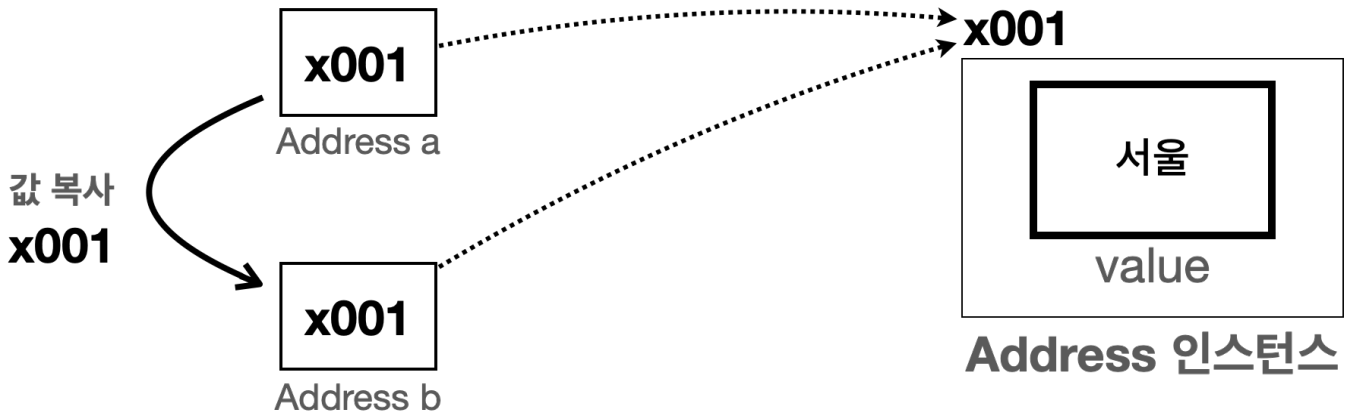
- 처음에는 `a`, `b` 둘다 서울이라는 주소를 가져야 한다고 가정하자.
  - 따라서 `Address b = a` 코드를 작성했고, 변수 `a`, `b` 둘다 서울이라는 주소를 가진다.
- 이후에 `b`의 주소를 부산으로 변경한다.
- 그런데 실행 결과를 보면 `b` 뿐만 아니라 `a`의 주소도 함께 부산으로 변경되어 버린다.

**실행 결과**

```
a = Address{value='서울'}
b = Address{value='서울'}
부산 -> b
a = Address{value='부산'}
b = Address{value='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.

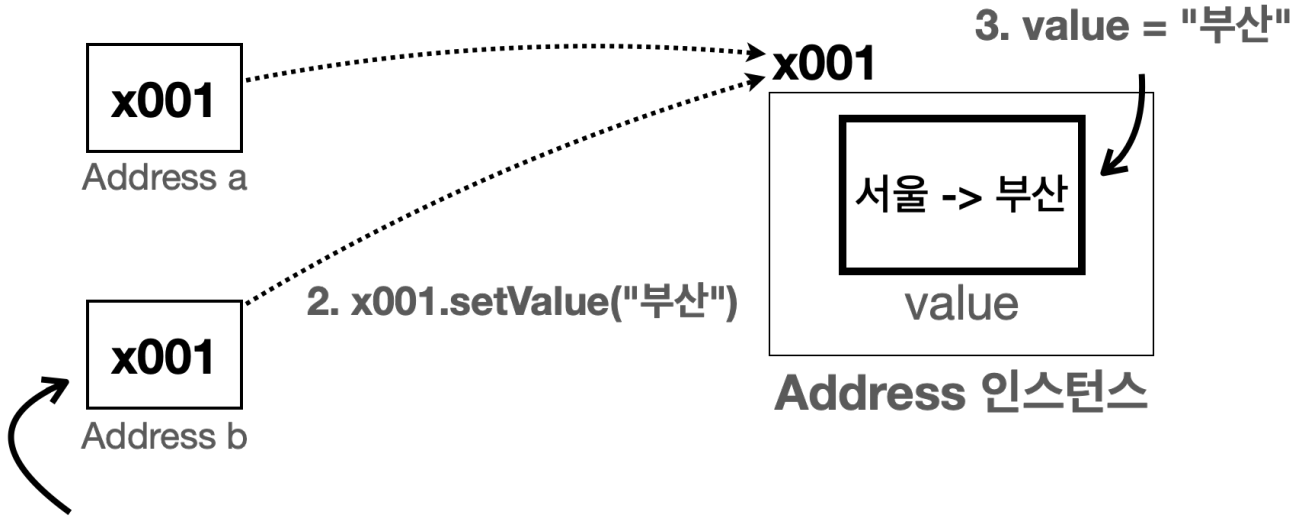
여기서 `b`의 주소만 부산으로 변경했는데, `a`의 주소도 함께 부산으로 변경되어 버린 이유는 무엇일까? 메모리 구조를 보면 바로 답이 나오겠지만, 개발을 하다 보면 누구나 이런 실수할 수 있을 것 같다는 생각도 함께 들 것이다.

## 공유 참조와 사이드 이펙트

사이드 이펙트(Side Effect)는 프로그래밍에서 어떤 계산이 주된 작업 외에 추가적인 부수 효과를 일으키는 것을 말한다.

앞서 `b`의 값을 부산으로 변경한 코드를 다시 분석해보자.

```
b.setValue("부산"); //b의 값을 부산으로 변경해야함
System.out.println("부산 -> b");
System.out.println("a = " + a); //사이드 이펙트 발생
System.out.println("b = " + b);
```




### 1. b.setValue("부산")

- 개발자는 `b`의 주소값을 서울에서 부산으로 변경할 의도로 값 변경을 시도했다.
- 하지만 `a`, `b`는 같은 인스턴스를 참조한다. 따라서 `a`의 값도 함께 부산으로 변경되어 버린다.

이렇게 주된 작업 외에 추가적인 부수 효과를 일으키는 것을 사이드 이펙트라 한다. 프로그래밍에서 사이드 이펙트는 보통 부정적인 의미로 사용되는데, 사이드 이펙트는 프로그램의 특정 부분에서 발생한 변경이 의도치 않게 다른 부분에 영향을 미치는 경우에 발생한다. 이로 인해 디버깅이 어려워지고 코드의 안정성이 저하될 수 있다.

## 사이드 이펙트 해결 방안

생각해보면 문제의 해결방안은 아주 단순하다. 다음과 같이 `a`와 `b`가 처음부터 서로 다른 인스턴스를 참조하면 된다.

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

코드를 작성해보자.

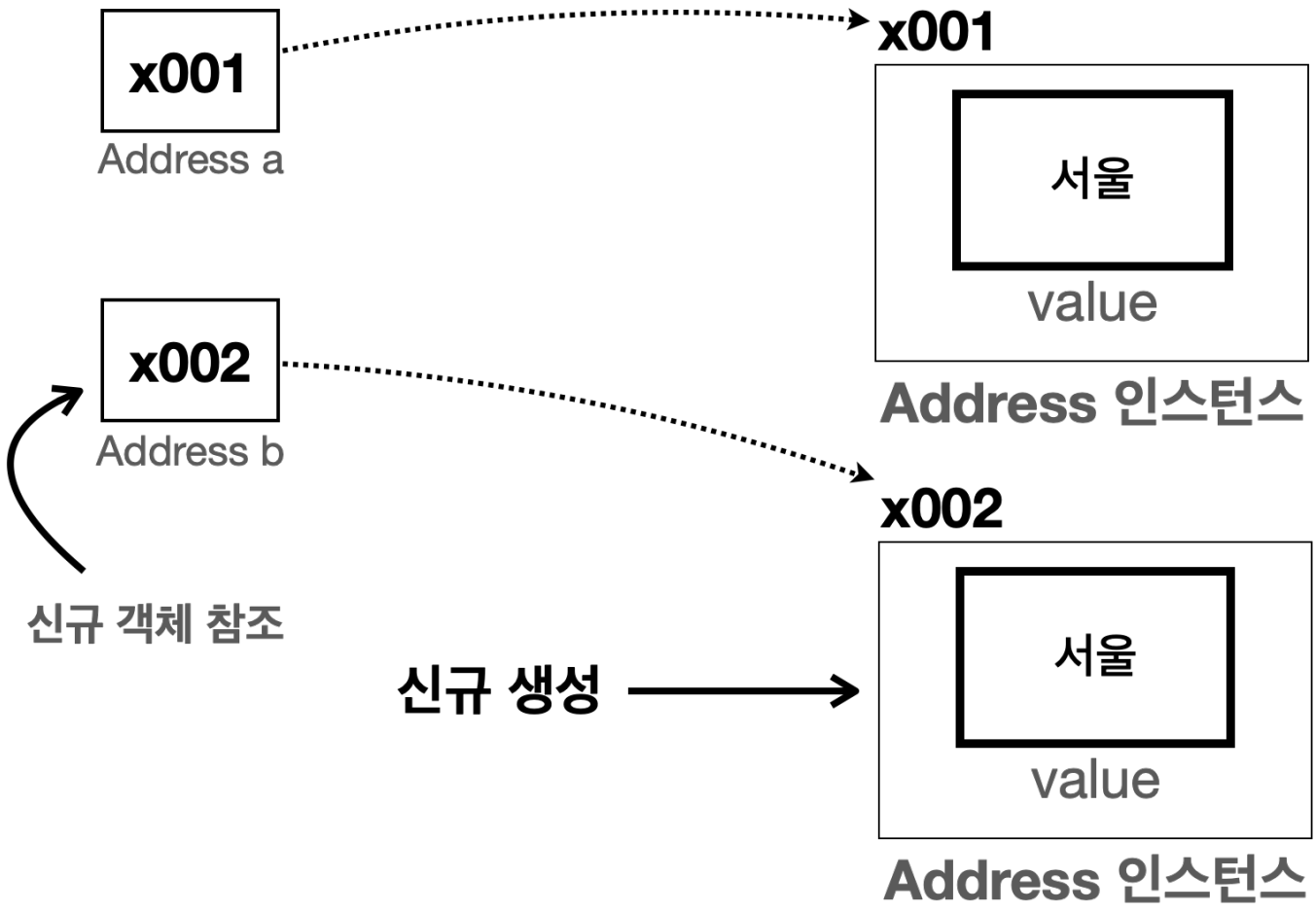
```
package lang.immutable.address;

public class RefMain1_2 {

    public static void main(String[] args) {
        Address a = new Address("서울");
        Address b = new Address("서울");
        System.out.println("a = " + a);
        System.out.println("b = " + b);

        b.setValue("부산");
        System.out.println("부산 -> b");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
}
```

**실행 결과**

```
a = Address{value='서울'}
b = Address{value='서울'}
부산 -> b
a = Address{value='서울'}
b = Address{value='부산'}
```

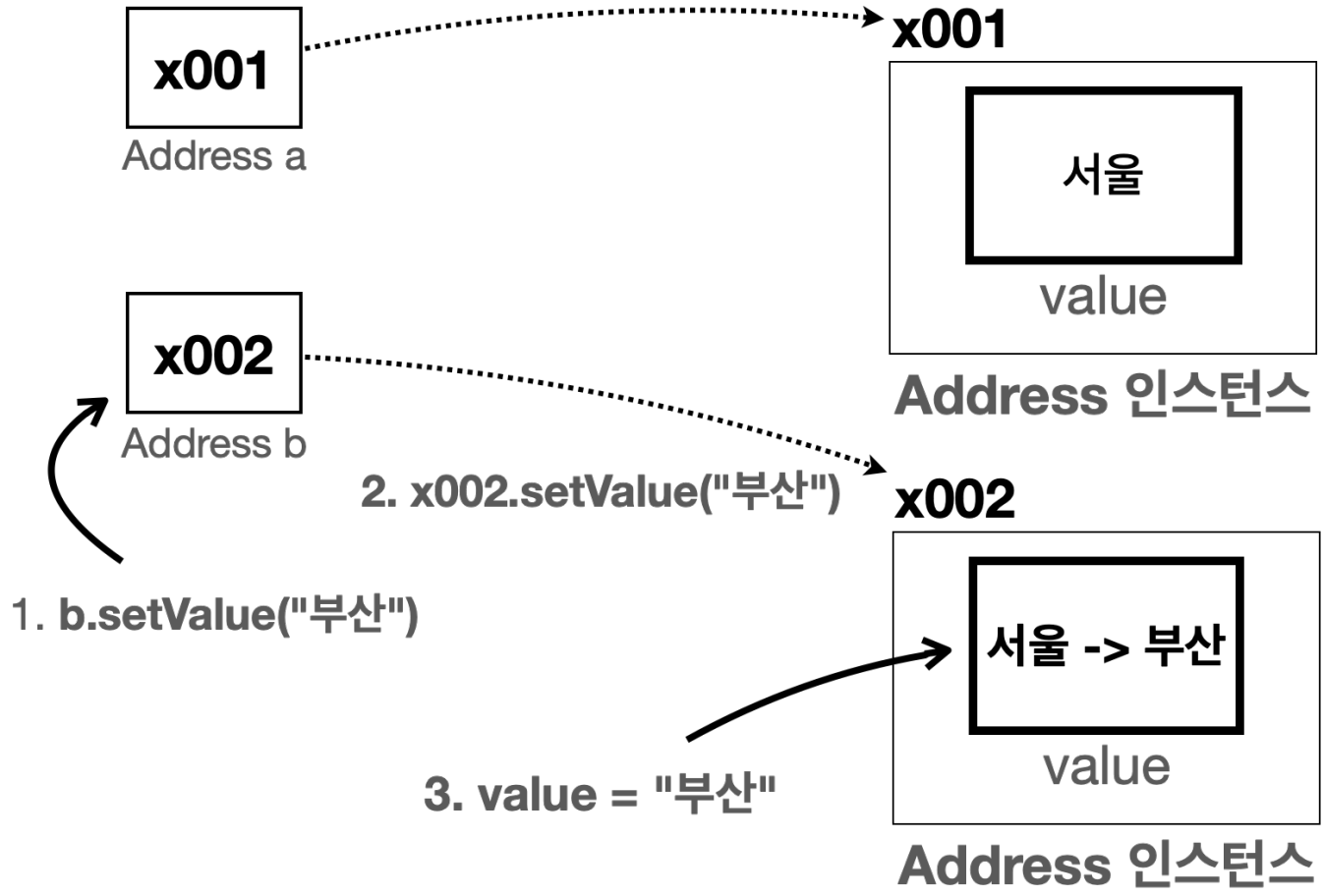
실행 결과를 보면 `b`의 주소값만 부산으로 변경된 것을 확인할 수 있다.

**그림 - 생성 코드**




- `a`와 `b`는 서로 다른 `Address` 인스턴스를 참조한다.

**그림 - 변경 코드**




- `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 방법은 없다

지금까지 발생한 모든 문제는 같은 객체(인스턴스)를 변수 `a`, `b`가 함께 공유하기 때문에 발생했다. 따라서 객체를 공유하지 않으면 문제가 해결된다. 여기서 변수 `a`, `b`가 서로 각각 다른 주소지로 변경할 수 있어야 한다. 이렇게 하려면 서로 다른 객체를 참조하면 된다.

#### 객체를 공유

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

- 이 경우 `a`, `b` 둘다 같은 `Address` 인스턴스를 바라보기 때문에 한쪽의 주소만 부산으로 변경하는 것이 불가능하다.

#### 객체를 공유 하지 않음

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

- 이 경우 `a`, `b`는 서로 다른 `Address` 인스턴스를 바라보기 때문에 한쪽의 주소만 부산으로 변경하는 것이 가능

하다.

이처럼 단순하게 서로 다른 객체를 참조해서, 같은 객체를 공유하지 않으면 문제가 해결된다.

쉽게 이야기해서 여러 변수가 하나의 객체를 공유하지 않으면 지금까지 설명한 문제들이 발생하지 않는다.
그런데 여기서 문제가 있다. 하나의 객체를 여러 변수가 공유하지 않도록 강제로 막을 수 있는 방법이 없다는 것이다.
다음 예를 보자.

**참조값의 공유를 막을 수 있는 방법이 없다.**

```
Address a = new Address("서울");
Address b = a; //참조값 대입을 막을 수 있는 방법이 없다.
```

`b = a`와 같은 코드를 작성하지 않도록 해서, 여러 변수가 하나의 참조값을 공유하지 않으면 문제가 해결될 것 같다.
하지만 `Address`를 사용하는 개발자 입장에서 실수로 `b = a`라고 해도 아무런 오류가 발생하지 않는다.
왜냐하면 자바 문법상 `Address b = a`와 같은 참조형 변수의 대입은 아무런 문제가 없기 때문이다.

다음과 같이 새로운 객체를 참조형 변수에 대입하든, 또는 기존 객체를 참조형 변수에 대입하든, 다음 두 코드 모두 자바 문법상 정상인 코드이다.

```
Address b = new Address("서울") //새로운 객체 참조
Address b = a //기존 객체 공유 참조
```

참조값을 다른 변수에 대입하는 순간 여러 변수가 하나의 객체를 공유하게 된다. 쉽게 이야기해서 **객체의 공유를 막을 수 있는 방법이 없다!**

기본형은 항상 값을 복사해서 대입하기 때문에 값이 절대로 공유되지 않는다. 하지만 참조형의 경우 참조값을 복사해서 대입하기 때문에 여러 변수에서 얼마든지 같은 객체를 공유할 수 있다.
객체의 공유가 꼭 필요할 때도 있지만, 때로는 공유하는 것이 지금과 같은 사이드 이펙트를 만드는 경우도 있다.

물론 개발자가 눈을 크게 잘 뜨고! 집중에서 코드를 잘 작성하면 사이드 이펙트 문제를 일으키지 않을 수 있다. 하지만 실제로는 훨씬 더 복잡한 상황에서 이런 문제가 발생한다. 다음 코드를 보자.

```
package lang.immutable.address;

public class RefMain1_3 {

    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 changeAddress) {
        System.out.println("주소 값을 변경합니다 -> " + changeAddress);
        address.setValue(changeAddress);
    }
}
```

- 앞서 작성한 코드와 같은 코드이다. 단순히 `change()` 메서드만 하나 추가되었다. 그리고 `change()` 메서드에서 `Address` 인스턴스에 있는 `value` 값을 변경한다.
- `main()` 메서드만 보면 `a`의 값이 함께 부산으로 변경된 이유를 찾기가 더 어렵다.

**실행 결과**

```
a = Address{value='서울'}
b = Address{value='서울'}
주소 값을 변경합니다 -> 부산
a = Address{value='부산'}
b = Address{value='부산'}
```

여러 변수가 하나의 객체를 참조하는 공유 참조를 막을 수 있는 방법은 없다. 그럼 공유 참조로 인해 발생하는 문제를 어떻게 해결할 수 있을까? 단순히 개발자가 공유 참조 문제가 발생하지 않도록 조심해서 코드를 작성해야 할까?

## 불변 객체 - 도입

지금까지 발생한 문제를 잘 생각해보면 공유하면 안되는 객체를 여러 변수에서 공유했기 때문에 발생한 문제이다.
하지만 앞서 살펴보았듯이 객체의 공유를 막을 수 있는 방법은 없다.
그런데 사이드 이펙트의 더 근본적인 원인을 고려해보면, 객체를 공유하는 것 자체는 문제가 아니다. 객체를 공유한다고 바로 사이드 이펙트가 발생하지는 않는다. **문제의 직접적인 원인은 공유된 객체의 값을 변경한 것에 있다.**

앞의 예를 떠올려보면 `a`, `b`는 처음 시점에는 둘다 `"서울"`이라는 주소를 사용해야 한다. 그리고 이후에 `b`의 주소를 `"부산"`으로 변경해야 한다.

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

```
Address b = a;
```

따라서 처음에는 `b = a`와 같이 `"서울"`이라는 `Address` 인스턴스를 `a`, `b`가 함께 사용하는 것이, 다음 코드와 같이 서로 다른 인스턴스를 사용하는 것 보다 메모리와 성능상 더 효율적이다. 인스턴스가 하나이니 메모리가 절약되고, 인스턴스를 하나 생성하지 않아도 되니 생성 시간이 줄어서 성능상 효율적이다.

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

여기까지는 `Address b = a`와 같이 공유 참조를 사용해도 아무런 문제가 없다. 오히려 더 효율적이다.

**진짜 문제는 이후에 b가 공유 참조하는 인스턴스의 값을 변경하기 때문에 발생한다.**

```
b.setValue("부산"); //b의 값을 부산으로 변경해야함
System.out.println("부산 -> b");
System.out.println("a = " + a); //사이드 이펙트 발생
System.out.println("b = " + b);
```

자바에서 여러 참조형 변수가 하나의 객체(인스턴스)를 참조하는 공유 참조 문제는 피할 수 없다.
기본형과 다르게 참조형인 객체는 처음부터 여러 참조형 변수에서 공유될 수 있도록 설계되었다. 따라서 이것은 문제가 아니다.
**문제의 직접적인 원인은 공유될 수 있는 `Address` 객체의 값을 어디선가 변경했기 때문이다.**

만약 `Address` 객체의 값을 변경하지 못하게 설계했다면 이런 사이드 이펙트 자체가 발생하지 않을 것이다.

## 불변 객체 도입

객체의 상태(객체 내부의 값, 필드, 멤버 변수)가 변하지 않는 객체를 불변 객체(Immutable Object)라 한다.
앞서 만들었던 `Address` 클래스를 상태가 변하지 않는 불변 클래스로 다시 만들어보자.

```
package lang.immutable.address;

public class ImmutableAddress {

    private final String value;

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

```
    public String getValue() {
        return value;
    }

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

- 내부 값이 변경되면 안된다. 따라서 `value`의 필드를 `final`로 선언했다.
- 값을 변경할 수 있는 `setValue()`를 제거했다.
- 이 클래스는 생성자를 통해서만 값을 설정할 수 있고, 이후에는 값을 변경하는 것이 불가능하다.

불변 클래스를 만드는 방법은 아주 단순하다. 어떻게든 필드 값을 변경할 수 없게 클래스를 설계하면 된다.

```
package lang.immutable.address;

public class RefMain2 {

    public static void main(String[] args) {
        ImmutableAddress a = new ImmutableAddress("서울");
        ImmutableAddress b = a; //참조값 대입을 막을 수 있는 방법이 없다.
        System.out.println("a = " + a);
        System.out.println("b = " + b);

        //b.setValue("부산"); //컴파일 오류 발생
        b = new ImmutableAddress("부산");
        System.out.println("부산 -> b");
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }
}
```

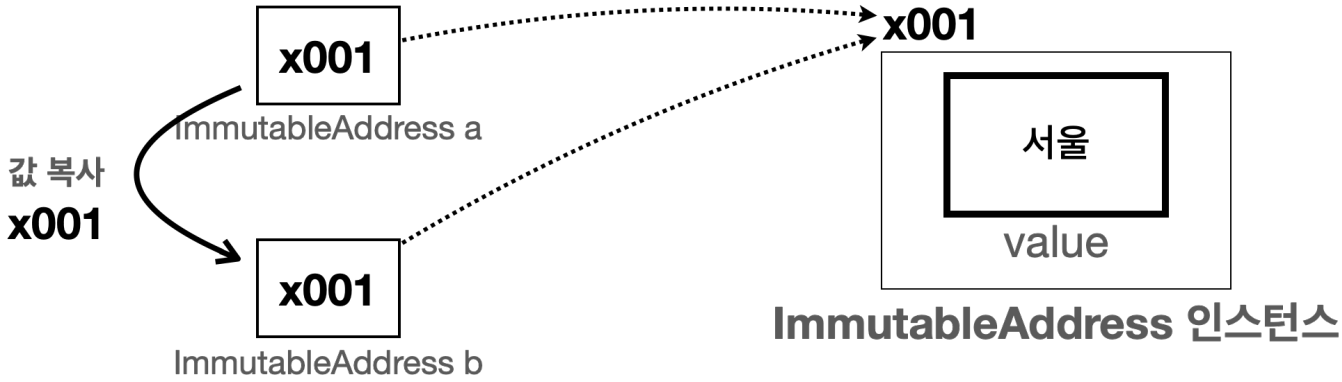
- `ImmutableAddress`의 경우 값을 변경할 수 있는 `b.setValue()` 메서드 자체가 제거되었다.
- 이제 `ImmutableAddress` 인스턴스의 값을 변경할 수 있는 방법은 없다.
- `ImmutableAddress`를 사용하는 개발자는 값을 변경하려고 시도하다가, 값을 변경하는 것이 불가능하다는 사실을 알고, 이 객체가 불변 객체인 사실을 깨닫게 된다.
    - 예를 들어 `b.setValue("부산")`을 호출하려고 했는데, 해당 메서드가 없다는 사실을 컴파일 오류를 통해 인지한다.

- 따라서 어쩔 수 없이 새로운 `ImmutableAddress("부산")` 인스턴스를 생성해서 `b`에 대입한다.
- 결과적으로 `a`, `b`는 서로 다른 인스턴스를 참조하고, `a`가 참조하던 `ImmutableAddress`는 그대로 유지된다.

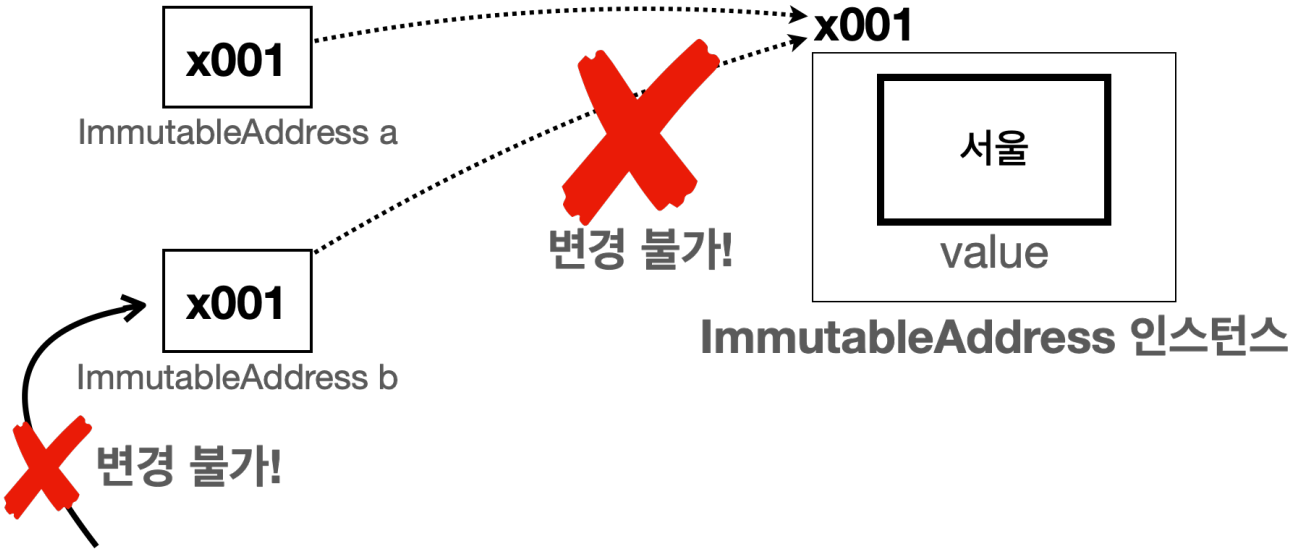
**실행 결과**

```
a = Address{value='서울'}
b = Address{value='서울'}
부산 -> b
a = Address{value='서울'}
b = Address{value='부산'}
```

실행 결과를 보면 `a`의 값은 그대로 유지되는 것을 확인할 수 있다.

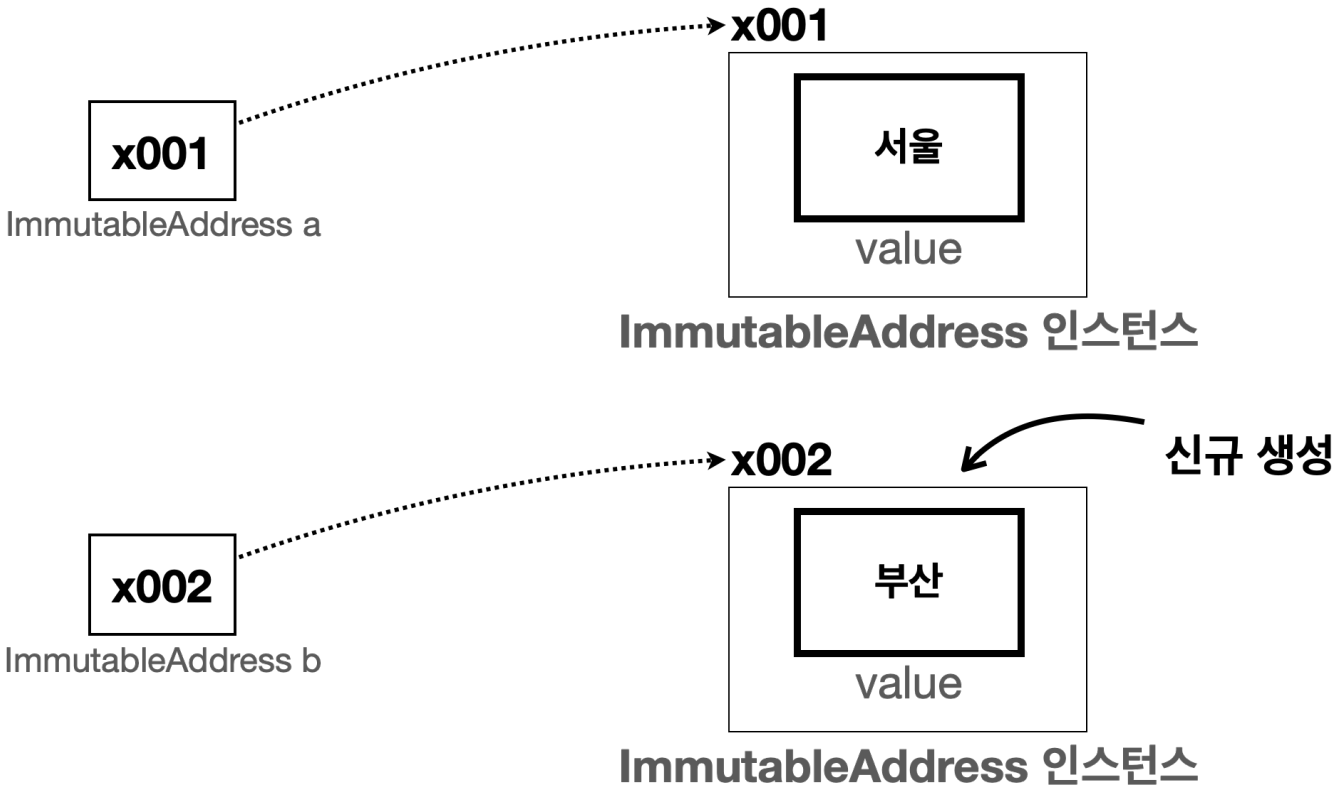



- 자바에서 객체의 공유 참조는 막을 수 없다.




### 1. b.setValue("부산")

- `ImmutableAddress`는 불변 객체이다. 따라서 값을 변경할 수 없다.




- `ImmutableAddress`은 불변 객체이므로 `b`가 참조하는 인스턴스의 값을 서울에서 부산으로 변경하려면 새로운 인스턴스를 생성해서 할당해야 한다.

**정리**

**불변이라는 단순한 제약을 사용해서 사이드 이펙트라는 큰 문제를 막을 수 있다.**

- 객체의 공유 참조는 막을 수 없다. 그래서 객체의 값을 변경하면 다른 곳에서 참조하는 변수의 값도 함께 변경되는 사이드 이펙트가 발생한다. 사이드 이펙트가 발생하면 안되는 상황이라면 불변 객체를 만들어서 사용하면 된다. 불변 객체는 값을 변경할 수 없기 때문에 사이드 이펙트가 원천 차단된다.
- 불변 객체는 값을 변경할 수 없다. 따라서 불변 객체의 값을 변경하고 싶다면 변경하고 싶은 값으로 새로운 불변 객체를 생성해야 한다. 이렇게 하면 기존 변수들이 참조하는 값에는 영향을 주지 않는다.

**참고 - 가변(Mutable) 객체 vs 불변(Immutable) 객체**

가변은 이름 그대로 처음 만든 이후 상태가 변할 수 있다는 뜻이다. (사전적으로 사물의 모양이나 성질이 달라질 수 있다는 뜻이다.)

불변은 이름 그대로 처음 만든 이후 상태가 변하지 않는다는 뜻이다. (사전적으로 사물의 모양이나 성질이 달라질 수 없다는 뜻이다.)

`Address`는 가변 클래스이다. 이 클래스로 객체를 생성하면 가변 객체가 된다.

`ImmutableAddress`는 불변 클래스이다. 이 클래스로 객체를 생성하면 불변 객체가 된다.

# 불변 객체 - 예제

조금 더 복잡하고 의미있는 예제를 통해서 불변 객체의 사용 예를 확인해보자.

앞의 `Address`, `ImmutableAddress`를 그대로 활용한다.

## 변경 클래스 사용

```
package lang.immutable.address;

public class MemberV1 {

    private String name;

    private Address address;

    public MemberV1(String name, Address address) {
        this.name = name;
        this.address = address;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    @Override
    public String toString() {
        return "Member{" +
                "name='" + name + '\'' +
                ", address=" + address +
                '}';
    }
}
```

- `MemberV1`은 변경 가능한 `Address` 클래스를 사용한다.

```java
package lang.immutable.address;

public class MemberMainV1 {

    public static void main(String[] args) {
        Address address = new Address("서울");

        MemberV1 memberA = new MemberV1("회원A", address);
        MemberV1 memberB = new MemberV1("회원B", address);

        //회원A, 회원B의 처음 주소는 모두 서울
        System.out.println("memberA = " + memberA);
        System.out.println("memberB = " + memberB);

        //회원B의 주소를 부산으로 변경해야함
        memberB.getAddress().setValue("부산");
        System.out.println("부산 -> memberB.address");
        System.out.println("memberA = " + memberA);
        System.out.println("memberB = " + memberB);
    }
}
```

- `회원A`와 `회원B`는 둘다 서울에 살고 있다.
- 중간에 `회원B`의 주소를 부산으로 변경해야 한다.
- 그런데 `회원A`와 `회원B`는 같은 `Address` 인스턴스를 참조하고 있다.
- `회원B`의 주소를 부산으로 변경하는 순간 `회원A`의 주소도 부산으로 변경된다.

**실행 결과**

```
memberA = Member{name='회원A', address=Address{value='서울'}}
memberB = Member{name='회원B', address=Address{value='서울'}}
부산 -> memberB.address
memberA = Member{name='회원A', address=Address{value='부산'}}
memberB = Member{name='회원B', address=Address{value='부산'}}
```

사이드 이펙트가 발생해서 회원B 뿐만 아니라 회원A의 주소도 부산으로 변경된다.

## 불변 클래스 사용

```java
package lang.immutable.address;

public class MemberV2 {
```

```
    private String name;

    private ImmutableAddress address;

    public MemberV2(String name, ImmutableAddress address) {
        this.name = name;
        this.address = address;
    }

    public String getName() {
        return name;
    }

    public ImmutableAddress getAddress() {
        return address;
    }

    public void setAddress(ImmutableAddress address) {
        this.address = address;
    }

    @Override
    public String toString() {
        return "Member{" +
                "name='" + name + '\'' +
                ", address=" + address +
                '}';
    }
}
```

- `MemberV2`는 주소를 변경할 수 없는, 불변인 `ImmutableAddress`를 사용한다.

```
package lang.immutable.address;

public class MemberMainV2 {

    public static void main(String[] args) {
        ImmutableAddress  address = new ImmutableAddress("서울");
        MemberV2 memberA = new MemberV2("회원A", address);
        MemberV2 memberB = new MemberV2("회원B", address);
```

```
        //회원A, 회원B의 처음 주소는 모두 서울
        System.out.println("memberA = " + memberA);
        System.out.println("memberB = " + memberB);

        //회원B의 주소를 부산으로 변경해야함
        //memberB.getAddress().setValue("부산"); //컴파일 오류
        memberB.setAddress(new ImmutableAddress("부산"));
        System.out.println("부산 -> memberB.address");
        System.out.println("memberA = " + memberA);
        System.out.println("memberB = " + memberB);
    }
}
```

- `회원B`의 주소를 중간에 부산으로 변경하려고 시도한다. 하지만 `ImmutableAddress`에는 값을 변경할 수 있는 메서드가 없다. 따라서 컴파일 오류가 발생한다.
- 결국 `memberB.setAddress(new ImmutableAddress("부산"))`와 같이 새로운 주소 객체를 만들어서 전달한다.

**실행 결과**

```
memberA = Member{name='회원A', address=Address{value='서울'}}
memberB = Member{name='회원B', address=Address{value='서울'}}
부산 -> memberB.address
memberA = Member{name='회원A', address=Address{value='서울'}}
memberB = Member{name='회원B', address=Address{value='부산'}}
```

사이드 이펙트가 발생하지 않는다. `회원A`는 기존 주소를 그대로 유지한다.

## 불변 객체 - 값 변경

불변 객체를 사용하지만 그래도 값을 변경해야 하는 메서드가 필요하면 어떻게 해야할까?

예를 들어서 기존 값에 새로운 값을 더하는 `add()`와 같은 메서드가 있다.

먼저 변경 가능한 객체에서 값을 변경하는 간단한 예를 만들어보자.

```
package lang.immutable.change;

public class MutableObj {

    private int value;
```

```
    public MutableObj(int value) {
        this.value = value;
    }

    public void add(int addValue) {
        value = value + addValue;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

```
package lang.immutable.change;

public class MutableMain {

    public static void main(String[] args) {
        MutableObj obj = new MutableObj(10);
        obj.add(20);
        //계산 이후 기존 값은 사라짐
        System.out.println("obj = " + obj.getValue());
    }
}
```

**실행 결과**

```
obj = 30
```

- `MutableObj`을 `10`이라는 값으로 생성한다.
- 이후에 `obj.add(20)`을 통해서 `10 + 20`을 수행한다.
  - 계산 이후에 기존에 있던 `10`이라는 값은 사라진다.
  - `MutableObj`의 상태(값)가 `10` ➜ `30`으로 변경되었다.
- `obj.getValue()`를 호출하면 `30`이 출력된다.

이번에는 불변 객체에서 `add()` 메서드를 어떻게 구현하는지 알아보자.
참고로 불변 객체는 변하지 않아야 한다.

```
package lang.immutable.change;

public class ImmutableObj {

    private final int value;

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

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

    public int getValue() {
        return value;
    }
}
```

- 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.

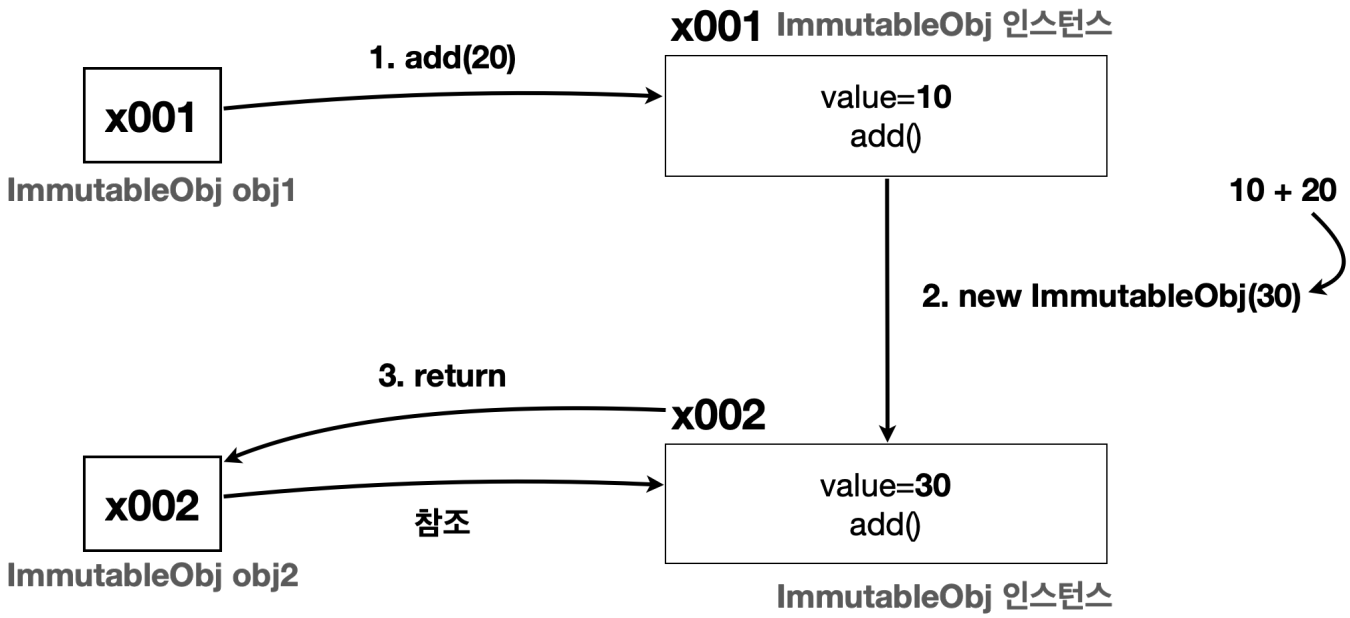
```
package lang.immutable.change;

public class ImmutableMain1 {

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

        //계산 이후에도 기존값과 신규값 모두 확인 가능
        System.out.println("obj1 = " + obj1.getValue());
        System.out.println("obj2 = " + obj2.getValue());
    }
}
```

**실행 결과**

```
obj1 = 10
obj2 = 30
```

불변 객체를 설계할 때 기존 값을 변경해야 하는 메서드가 필요할 수 있다. 이때는 기존 객체의 값을 그대로 두고 대신에 변경된 결과를 새로운 객체에 담아서 반환하면 된다. 결과를 보면 기존 값이 그대로 유지되는 것을 확인할 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.

만약 여기서 다음과 같이 새로 생성된 반환 값을 사용하지 않으면 어떻게 될까?

```
package lang.immutable.change;

public class ImmutableMain2 {

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

        System.out.println("obj1 = " + obj1.getValue());
    }
```

```
}
```

**실행 결과**

```
obj1 = 10
```

실행 결과처럼 아무것도 처리되지 않은 것 처럼 보일 것이다.

불변 객체에서 변경과 관련된 메서드들은 보통 객체를 새로 만들어서 반환하기 때문에 **꼭! 반환 값을 받아야 한다.**

# 문제와 풀이

**문제 설명**

- `MyDate` 클래스는 불변이 아니어서 공유 참조시 사이드 이펙트가 발생한다. 이를 불변 클래스로 만들어라.
- 새로운 불변 클래스는 `ImmuableMyDate`로 이름 지으면 된다.
- 새로운 실행 클래스는 `ImmuableMyDateMain`으로 이름 지으면 된다.

```
package lang.immutable.test;

public class MyDate {

    private int year;
    private int month;
    private int day;

    public MyDate(int year, int month, int day) {
        this.year = year;
        this.month = month;
        this.day = day;
    }

    public void setYear(int year) {
        this.year = year;
    }

    public void setMonth(int month) {
        this.month = month;
    }

    public void setDay(int day) {
```

```
        this.day = day;
    }

    @Override
    public String toString() {
        return year + "-" + month + "-" + day;
    }
}
```

```
package lang.immutable.test;

public class MyDateMain {

    public static void main(String[] args) {
        MyDate date1 = new MyDate(2024,1,1);
        MyDate date2 = date1;
        System.out.println("date1 = " + date1);
        System.out.println("date2 = " + date2);

        System.out.println("2025 -> date1");
        date1.setYear(2025);
        System.out.println("date1 = " + date1);
        System.out.println("date2 = " + date2);
    }
}
```

**실행 결과**

```
date1 = 2024-1-1
date2 = 2024-1-1
2025 -> date1
date1 = 2025-1-1
date2 = 2025-1-1
```

**정답**

```
package lang.immutable.test;

public class ImmutableMyDate {

    private final int year;
    private final int month;
```

```
    private final int day;

    public ImmutableMyDate(int year, int month, int day) {
        this.year = year;
        this.month = month;
        this.day = day;
    }

    public ImmutableMyDate withYear(int newYear) {
        return new ImmutableMyDate(newYear, month, day);
    }

    public ImmutableMyDate withMonth(int newMonth) {
        return new ImmutableMyDate(year, newMonth, day);
    }

    public ImmutableMyDate withDay(int newDay) {
        return new ImmutableMyDate(year, month, newDay);
    }

    @Override
    public String toString() {
        return year + "-" + month + "-" + day;
    }
}
```

```
package lang.immutable.test;

public class ImmutableMyDateMain {

    public static void main(String[] args) {
        ImmutableMyDate date1 = new ImmutableMyDate(2024, 1, 1);
        ImmutableMyDate date2 = date1;
        System.out.println("date1 = " + date1);
        System.out.println("date2 = " + date2);

        System.out.println("2025 -> date1");
        date1 = date1.withYear(2025);
        System.out.println("date1 = " + date1);
        System.out.println("date2 = " + date2);
    }
}
```

**실행 결과**

```
date1 = 2024-1-1
date2 = 2024-1-1
2025 -> date1
date1 = 2025-1-1
date2 = 2024-1-1
```

**참고 - withXxx()**

불변 객체에서 값을 변경하는 경우 `withYear()` 처럼 "with"로 시작하는 경우가 많다.

예를 들어 "coffee with sugar"라고 하면, 커피에 설탕이 추가되어 원래의 상태를 변경하여 새로운 변형을 만든다는 것을 의미한다.

이 개념을 프로그래밍에 적용하면, 불변 객체의 메서드가 "with"로 이름 지어진 경우, 그 메서드가 지정된 수정사항을 포함하는 객체의 새 인스턴스를 반환한다는 사실을 뜻한다.

정리하면 "with"는 관례처럼 사용되는데, 원본 객체의 상태가 그대로 유지됨을 강조하면서 변경사항을 새 복사본에 포함하는 과정을 간결하게 표현한다.

# 정리

지금까지 왜 이렇게 불변 객체 이야기를 많이 했을까?

자바에서 가장 많이 사용되는 `String` 클래스가 바로 불변 객체이기 때문이다. 뿐만 아니라 자바가 기본으로 제공하는 `Integer`, `LocalDate` 등 수 많은 클래스가 불변으로 설계되어 있다. (이후에 학습한다.)

따라서 불변 객체가 필요한 이유와 원리를 제대로 이해해야, 이런 기본 클래스들도 제대로 이해할 수 있다.

**모든 클래스를 불변으로 만드는 것은 아니다.**

우리가 만드는 대부분의 클래스는 값을 변경할 수 있게 만들어진다. 예를 들어서 회원 클래스의 경우 회원의 여러 속성을 변경할 수 있어야 한다. 가변 클래스가 더 일반적이고, 불변 클래스는 값을 변경하면 안되는 특별한 경우에 만들어서 사용한다고 생각하면 된다. 때로는 같은 기능을 하는 클래스를 하나는 불변으로 하나는 가변으로 각각 만드는 경우도 있다.

클래스를 불변으로 설계하는 이유는 더 많다.

- 캐시 안정성
- 멀티 쓰레드 안정성
- 엔티티의 값 타입

지금은 이런 부분을 다 이해할 수 없다. 관련 내용을 학습하다 보면 자연스럽게 이번에 배운 불변 객체가 떠오르면서 관련된 내용을 본질적으로 더 잘 이해할 수 있을 것이다. 프로그래밍을 더 깊이있게 학습할 수 록 다양한 불변 클래스 이용 사례를 만나고 이해하게 된다. 따라서 **지금은 불변 클래스가 어디에 사용되고, 어떻게 활용되는지 보다는 불변 클래스의 원리를 이해하는 정도면 충분하다.**