

# 아이템 17

## 변경 가능성을 최소화하라



오빠!  
변했어!!

오빠도 불변 객체로 만들자

# 불변 객체

- 인스턴스 내부의 값을 수정할 수 없는 객체
- 간직한 정보는 고정되어 객체가 파괴될 때 까지 절대 달라지지 않아야 한다.
- 자바 플랫폼 라이브러리에는 다음과 같은 불변 클래스가 있다.
  - String, Integer, Double ...

# 불변 객체를 만드는 규칙

- 객체의 상태를 변경할 수 있는 메서드(변경자)를 제공하지 않는다.


```
public class Money {  
    private int value;  
  
    public Money(int value) {  
        this.value = value;  
    }  
  
    public int getValue() {  
        return value;  
    }  
  
    public void setValue(int value) {  
        this.value = value;  
    }  
}
```

Setter가 있으면 불변 객체가 아니다

# 불변 객체를 만드는 규칙

- 클래스를 확장할 수 없도록 한다. (상속 금지)

final 클래스는 확장 불가

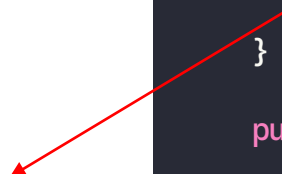


```
public final class Money {  
    private final int value;  
  
    public Money(int value) {  
        this.value = value;  
    }  
  
    public int getValue() {  
        return value;  
    }  
}
```

## 더 좋은 방법 - 정적 팩토리 메서드

- 생성자를 private로 잠가서 상속을 막는다

생성자가 private이면 상속받는 클래스에서  
super()를 호출할 수 없다



```
public class Money {  
    private final int value;  
  
    private Money(int value) {  
        this.value = value;  
    }  
  
    public static Money valueOf(int value) {  
        return new Money(value);  
    }  
  
    public int getValue() {  
        return value;  
    }  
}
```

# 불변 객체를 만드는 규칙

- 모든 필드를 `private final`로 만든다.
  - 일반적으로는 `public final`로만 만들어도 충분
- 그러나 `private final`로 만들자.
  - 아이템 15 클래스와 멤버의 접근 권한을 최소화하라
  - 아이템 16 `public` 클래스에서는 `public` 필드가 아닌 접근자 메서드를 사용하라

# 필드가 `private final`이라고 불변이 아니다

- 원시 타입의 필드는 `private final`이면 불변
- 참조 타입의 필드는 `private final`이어도 가변일 수 있다.

# Quiz. 이 객체는 왜 불변이 아닐까?

```
public class Nation {
    private final City capital;
    private final List<City> cities;
    private final long population;

    public Nation(City capital, List<City> cities, long population) {
        this.capital = capital;
        this.cities = cities;
        this.population = population;
    }

    public City getCapital() {
        return capital;
    }

    public List<City> getCities() {
        return cities;
    }

    public long getPopulation() {
        return population;
    }
}
```



# 불변 객체를 만드는 규칙

- 자신 외에는 내부의 가변 컴포넌트에 접근할 수 없도록 한다.

```
public class Lotto {  
    private final List<Integer> numbers;  
  
    public Lotto(List<Integer> numbers) {  
        this.numbers = List.copyOf(numbers);  
    }  
  
    public List<Integer> getNumbers() {  
        return Collection.unmodifiableList(numbers);  
    }  
}
```

→ 생성자, 접근자에 방어적 복사

# 불변 객체는 왜 써야 하는가?

- 근본적으로 Thread-safe 하며 동기화 할 필요가 없다.
  - 안심하고 공유할 수 있다.
- 방어적 복사가 필요 없다.
  - clone이나 복사 생성자 구현 X

이렇게 불변 객체가 있다면 재사용이 가능하다.  
(예제에서는 1~45의 인스턴스를 캐싱해두고 사용)

```
public class LottoNumber {
    static final Map<Integer, LottoNumber> LOTTO_NUMBER_POOL;

    static {
        LOTTO_NUMBER_POOL = IntStream.rangeClosed(1, 45)
            .boxed()
            .collect(Collectors.toUnmodifiableMap(Function.identity(), LottoNumber::new));
    }

    private final int number;

    private LottoNumber(int number) {
        validateRange(number);
        this.number = number;
    }

    public static LottoNumber valueOf(int number) {
        validateRange(number);
        return LOTTO_NUMBER_POOL.get(number);
    }

    private static void validateRange(int number) {
        if (number < 1 || number > 45) {
            throw new IllegalArgumentException();
        }
    }

    ...
}
```

# 불변 객체는 왜 써야 하는가?

- 자유롭게 공유할 수 있으며, 불변 객체끼리는 내부 데이터를 공유할 수 있다.

```
public BigInteger negate() {  
    return new BigInteger(this.mag, -this.signum);  
}
```


```
public class BigInteger extends Number implements Comparable<BigInteger> {  
    final int signum;  
  
    final int[] mag;  
}
```

negate()로 부호가 반대인 BigInteger 생성 시 복사 과정 없이 공유한다  
mag는 가변이지만 불변 객체끼리여서 상관 X

# 불변 객체는 왜 써야 하는가?

- 객체를 만들 때 구성 요소로 다른 불변 객체를 사용하면 이점이 많다.
  - ex) Set의 구성 요소나 Map의 key
- Set의 구성요소나 Map의 key가 가변이면 해당 자료구조 자체의 불변식이 허물어진다.

```
Map<Month, Integer> attendanceBook = new EnumMap<Month, Integer>(Month.class);
```



Month가 가변이라면 Month를 수정하여  
attendanceBook의 불변식을 깨뜨릴 수 있다.

# 불변 객체 사용의 단점?

- 값이 다르면 반드시 독립된 새 인스턴스를 만들어주어야 한다.
  - 값의 가짓수가 많으면 비용이 크다.
    - ex) BigInteger의 비트 하나를 바꾸는 연산

- 문제 해결: 다단계 연산 or 가변 동반 클래스

ex) 모듈러 지수 연산

ex) StringBuilder, MutableBigInteger

MutableBigInteger, BitStieve 등은 package-private이어서 클라이언트가 사용하지 못하고 BigInteger의 메서드들이 내부적으로 사용

```
public BigInteger sqrt() {  
    if (this.signum < 0) {  
        throw new ArithmeticException("Negative BigInteger");  
    }  
  
    return new MutableBigInteger(this.mag).sqrt().toBigInteger();  
}
```

가변 동반 클래스 사용

```
BigInteger toBigInteger(int sign) {  
    if (intLen == 0 || sign == 0)  
        return BigInteger.ZERO;  
    return new BigInteger(getMagnitudeArray(), sign);  
}
```

default 접근 제어자로  
package-private

**Q. 특수한 경우를 제외하면 불변 객체로 인한  
자원 소모를 신경 쓸 필요가 있을까?**

# 정리

- 클래스는 꼭 필요한 경우가 아니면 불변이어야 한다. (특히나 값 객체(VO) 라면)
  - 무작정 getter setter부터 만들고 보지 말자
- 불변으로 만들 수 없는 클래스라도 가변 부분을 최대한 줄이자.
- 합당한 이유가 없다면 모든 필드는 private final이어야 한다.
- 생성자는 초기화가 완벽히 끝난 상태의 객체를 생성해야 한다.