💚

아이템 17) 변경 가능성을 최소화하라

불변 클래스 : 인스턴스 내부 값을 수정할 수 없는 클래스

- 자바 라이브러리의 다양한 불변 클래스들 예시) String, primitive type 박싱 클래스, BigInteger, BigDecimal ..

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
               Constable, ConstantDesc {

    private final byte coder;
    ...
}
```

클래스를 불변으로 만드는 규칙 5가지

1. 객체 상태를 변경하는 메소드 제공 X
2. 클래스를 확장할 수 없도록 final 로 클래스 선언
3. 모든 필드를 final 로 선언
4. 모든 필드를 private 로 선언
    - public final 로만 선언해도 불변 객체가 되지만, **다음 릴리즈에서 내부 표현을 바꾸지 못한다 ?**

      ```
      public class ImmutableObject {
          public final int value;

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

      - 이 코드에서 왜 내부 표현을 변경하는 것이 어려운지 살펴보자 !

      1. API 변경의 어려움
          - public 필드 노출 시, 외부에서 **value 라는 필드를 사용하는 다른 모든 코드를 업데이트 해야 함**
      2. 캡슐화가 되지 않음
          - public 필드를 노출함으로써 **내부 구현이 외부로 노출됨**

      - 불변 객체를 설계할 때는 필드를 `private`으로 선언하고, 필요한 경우 `public` 메서드를 사용하여 필드에 접근하도록 하자 ! → 내부 표현을 변경하거나 필드에 대한 접근을 제어할 수 있음
5. 자신 외에는 내부의 가변 컴포넌트에 접근할 수 없게 하라
    - 클래스에 가변 객체를 참조하는 필드가 하나라도 있다면, ***클라이언트에서 그 객체의 참조를 얻을 수 없도록 해야 한다.***
    - 이런 필드는 ***절대 클라이언트가 제공한 가변 객체 참조를 직접 가리키게 해서는 안되며***, 접근자 메소드가 그 필드를 그대로 반환해서도 안된다

```
public class MutableContainer{
    private List<String> mutableList; // 가변 리스트를 참조하는 필드

    public MutableContainer(List<String> list) {
        this.mutableList = new ArrayList<>(list);
    }

    // 접근자 메소드가 그 필드를 그대로 반환해서도 안됨
    public List<String> getMutableList(){
        return mutableList;
    }

    // 가변 리스트의 참조를 반환하지 않고 복사본 제공
    public List<String> getImmutableList() {
        return Collections.unmodifiableList(mutableList);
    }
}
```

```
public class Main {
  public static void main(String[] args){
    List<String> clientList = new ArrayList<>();
    clientList.add("one");
    clientList.add("two");

    MutableContainercontainer = new MutableContainer(clientList);

    // 절대 클라이언트가 제공한 가변 객체 참조를 직접 가리키게 해서는 안됨
    List<String> list = container.getMutableList();
    list.add("three"); // clientList 의 속성값이 변함
  }
}
```

- 생성자, 접근자, readObject 메소드 모두에서 `방어적 복사`를 수행하라
  - 방어적 복사 : 주로 가변 객체를 복사하여 **기존 객체의 변경에 대한 방어**를 구현하는 기법

```
public class DefensiveCopyExample {
    private List<String> mutableList;

    // 생성자
    public DefensiveCopyExample(List<String> list) {
        this.mutableList = new ArrayList<>(list);
    }

    // 접근자
    public List<String> getMutableList() {
        return new ArrayList<>(mutableList);
    }
```

함수형 프로그래밍 : 피연산자에 함수를 적용해 그 결과를 반환하지만, 피연산자 자체는 그대로인 프로그래밍 패턴

```
public class PureFunctionExample {
    public static int plusOne(int x) {
        return x + 1;
    }

    public static void main(String[] args) {
        int input = 5;

        // 순수 함수 addOne을 사용하여 입력값에 1을 더한 값을 반환
        int result = plusOne(input);

        System.out.println("Input: " + input);   // 5
        System.out.println("Result: " + result); // 6
    }
}
```

- 절차 / 명령형 프로그래밍에서는 메소드에서, 피연산자인 자신을 수정해 자신의 상태가 변함
- `순수함수` : 입력값을 받아 출력값을 반환하지만, 외부 상태를 변경하지 않고 동일한 입력에 대해 항상 동일한 출력을 반환하는 함수
- `함수형 프로그래밍` : 순수함수를 사용하여 프로그램을 구성하는 방법을 강조
- 이러한 방식으로 프로그래밍하면 코드에서 **불변이 되는 영역 비율이 높아진다는 장점이 있음**

- 불변 객체는 근본적으로 `thread-safe` 하여 따로 동기화할 필요가 없음
  - 여러 스레드가 동시에 사용해도 절대 훼손되지 않으므로, **불변 클래스라면 한번 만든 인스턴스를 최대한 재활용하기를 원한다**
  - 가장 쉬운 재활용 방법 : 자주 쓰이는 값을 상수 (public static final) 로 제공하는 것
  - 불변 클래스는, **자주 사용되는 인스턴스를 캐싱하여, 같은 인스턴스를 중복 생성하지 않게 해주는 정적 팩토리 제공 가능**
    ex) 박싱된 primitive class, BigInteger

```
public static final BigInteger ZERO = new BigInteger(new int[0], 0);

public static BigInteger valueOf(long val) {
        // If -MAX_CONSTANT < val < MAX_CONSTANT, return stashed constant
        if (val == 0)
```

```
            return ZERO;
        if (val > 0 && val <= MAX_CONSTANT)
            return posConst[(int) val];
        else if (val < 0 && val >= -MAX_CONSTANT)
            return negConst[(int) -val];

        return new BigInteger(val);
    }
```

- 불변 객체는 자유롭게 공유할 수 있고, **불변 객체끼리는 내부 데이터를 공유할 수 있다.**
  - BigInteger 클래스의 negate 메소드 : 크기만 같고, 부호만 반대인 새로운 BigInteger 를 생성함
  - 배열 (mag) 는 가변이지만, 복사하지 않고 원본 인스턴스랑 공유해도 되므로, 새로만든 인스턴스도 원본 인스턴스가 가리키는 내부 배열을 그대로 가리킴

```
final int[] mag;
final int signum;

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

- 불변 객체는 그 자체로 실패 원자성 제공
  - 상태가 변하지 않으니, 불일치 상태에 빠질 가능성 X
- 불변 클래스는 값이 다른 경우 **반드시 독립된 객체로 만들어야 함**

  예시) 백만 비트짜리 BigInteger 에서 비트 하나를 바꾸는 경우
  - 비트가 하나만 다르지만, flipBit 메소드는 새로운 BigInteger 인스턴스를 생성하게 됨

```
BigInteger moby = ...;
moby = moby.flipBit(0);
```

- 중간 단계에서 만들어진 객체를 모두 버리게 되면 성능 문제가 불거지는데, 이 문제에 대처하는 방법 2가지

1. **다단계 연산 (multistep operation) 들을 예측하여 기본 기능으로 제공하기**
   - 모듈러 지수 같은 다단계 연산 속도를 높여주는 `가변 동반 클래스` 를 package-private 으로 두고 있음
   - BigInteger 의 경우, BitSieve, SignedMutableBigInteger, MutableBigInteger 등을 가변 동반 클래스로 가짐.

```
// BigInteger Class
public BigInteger gcd(BigInteger val) {
   if (val.signum == 0)
       return this.abs();
   else if (this.signum == 0)
       return val.abs();

   MutableBigInteger a = new MutableBigInteger(this);
   MutableBigInteger b = new MutableBigInteger(val);

   MutableBigInteger result = a.hybridGCD(b);

   return result.toBigInteger(1);
}

// MutableBigInteger Class
BigInteger toBigInteger(int sign) { // package-private 으로 두어 클라이언트의 직접 조작을 막고, BigInteger 에서만 사용하도록 함
    if (intLen == 0 || sign == 0)
        return BigInteger.ZERO;
    return new BigInteger(getMagnitudeArray(), sign);
}
```

2. **정확하게 복잡한 연산을 예측할 수 없는 경우, 이 클래스를 public 으로 제공하기**
   - String 클래스의 가변 동반 클래스인 StringBuilder 가 그 예시
     - StringBuilder 인스턴스를 생성한 후, append 메소드를 통해 String 을 추가하고, toString 으로 String 결과값을 얻을 수 있다.

- MutableInteger 는 클라이언트가 사용할 수 없었던 것과는 다르게, 클라이언트가 객체를 생성하여 조작이 가능함

```
// StringBuilder Class

@Override
public StringBuilder append(Object obj) {
    return append(String.valueOf(obj));
}

@Override
@HotSpotIntrinsicCandidate
public StringBuilder append(String str) {
    super.append(str);
    return this;
}

@Override
@HotSpotIntrinsicCandidate
public String toString() {
    // Create a copy, don't share the array
    return isLatin1() ? StringLatin1.newString(value, 0, count)
                      : StringUTF16.newString(value, 0, count);
}
```

- **클래스가 불변임을 보장하기 위한 “Final 클래스” 외의 다른 방법**

  → 모든 생성자를 private 또는 package-private 으로 만들고, public 정적 팩토리 제공

  - 외부의 클라이언트 입장에선, 이 클래스는 사실상 final 클래스
  - public / protected 생성자가 없으니까 다른 패키지에서는 이 클래스 확장하는 게 불가능

- 계산 비용이 큰 값을 처음 쓰일 때 계산하여 final 이 아닌 필드에 캐시해두기도 하고, 똑같은 값을 다시 요청하면 캐시 값을 반환하여 계산 비용 절감
- 직렬화 할 때는 Serializable 을 구현하는 불변 클래스 내에 가변 객체를 참조하는 필드가 있다면 **readObject, readResolve 메소드를 제공해야 함**

```
public class Person implements Serializable {
    private final String name;
    private final Date birthDate;

    public Person(String name, Date birthDate) {
        this.name = name;
        this.birthDate = new Date(birthDate.getTime()); // Date를 복제하여 불변성 유지
    }

    public String getName() {
        return name;
    }

    public Date getBirthDate() {
        return new Date(birthDate.getTime()); // Date를 복제하여 불변성 유지
    }

    // readObject 메서드를 사용하여 가변 필드를 복원
    private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
        ois.defaultReadObject();

        /*
          직렬화 및 역직렬화 중에 객체의 상태가 변경되는 것을 방지하려면 객체의 복사본을 사용해야 함
          Date 필드를 복원할 때 Date 객체를 복제해서 가변성 제거
        */
        this.birthDate = new Date(birthDate.getTime());
    }

    // readResolve 메서드를 사용하여, 객체가 역직렬화 된 후 반환할 불변 객체를 반환
    private Object readResolve() {
        return new Person(name, birthDate);
    }
}
```

  - `readObject` 메소드는 객체를 역직렬화할 때 호출됨
    - `ObjectInputStream`을 통해 역직렬화된 데이터를 처리하고, 역직렬화된 객체의 초기화 작업을 수행

        - 주로 가변 필드를 복원하고, 보안 검사, 데이터 유효성 검사 등을 수행하는 데 사용

    - `readResolve` 메소드는 객체 역직렬화 후에 호출됨
        - 이 메소드는 역직렬화된 객체를 대체하고, 대체된 객체를 반환
        - 주로 불변 객체를 반환하거나, 역직렬화된 객체를 캐시된 객체로 대체하는 데 사용됨

- 클래스는 꼭 필요한 경우가 아니라면 불변이어야 한다 !! **특히 단순한 값 객체일 경우에는 항상 불변으로 만들자**
    - 무거운 값 객체라도 불변으로 만들 수 있을지 생각하고, 성능 때문에 불변으로 만들 수 없다면 불변 클래스와 쌍을 이루는 가변 동반 클래스를 public 클래스로 제공하도록 하자!
- 불변으로 만들 수 없는 클래스라도, 변경할 수 있는 부분을 최소한으로 줄여서 변경해야 할 필드를 뺀 나머지를 final 로 선언하자
- 생성자는 불변식 설정이 모두 완료된, **초기화가 완벽히 끝난 상태의 객체를 생성해야 하며 확실한 이유가 없는 경우, 생성자/정적 팩토리 외의 그 어떤 초기화 메소드도 public 로 제공해서는 안된다** (객체를 재활용할 목적으로 상태를 다시 초기화하는 메소드도 안된다)