EJ

이펙티브 자바 CP.7

작성 일시	@2023년 2월 22일 오후 9:28
최종 편집 일시	@2023년 2월 25일 오후 9:31
유형	이펙티브 자바
작성자	종현 박
참석자	

7️⃣ 메소드

▼ 선행 내용

- 메서드 설계시 주의점
  - 매개변수와 반환 값 처리
  - 메서드 시그니처 설계화, 문서화
  - 상당 부분은 메서드 뿐만 아니라 생성자에 적용된다.
- 사용성, 견고성, 유연성에 집중

▼ 49. 매개변수가 유효한지 검사하라

- 오류는 가능한 빨리 잡아야 한다.
  - 메서드와 생성자 대부분은 입력 매개변수의 값이 특정 조건을 만족하기를 바란다.
    - 인덱스 값이 음수이면 안 되며, 객체 참조는 `null` 이 아니어야 하는 시깅다.

    - 이런 제약은 반드시 문서화를 해야 하며 메서드 몸체가 시작되기 전에 검사해야한다.
    - 오류를 발생한 즉시 잡지 못하면 해당 오류를 감지하기 어려워지고, 감지하더라도 오류 발생 지점을 찾기 어려워진다.
- 메서드가 실행되기 전에 매개변수를 체크한다면 잘못된 값이 넘어 올 때 즉각적이고 깔끔하게 예외를 처리 할 수 있다.
    - 매개변수 검사를 제대로 하지 못할 때 문제
        - 메서드가 수행되는 중간에 모호한 예외를 던지며 실패할 수 있다.
        - 메서드가 수행되었지만 잘못된 결과를 반환한다.
        - 메서드는 문제없이 수행되었지만, 어떤 객체를 이상한 상태로 만들어서 미래의 알수 없는 시점에 문제를 일으키는 경우
        - 즉, 매개변수 검사에 실패하면 **실패 원자성**(아이템 76)을 어기는 결과를 낳을 수 있다.
- `public`, `protected` 메서드는 매개변수 값이 잘못됐을 때, 던지는 예외를 문서화해야 한다.
    - `@throws` 자바독 태그를 사용하면 된다. (아이템 74)
        - 주로, `IllegalArgumentException`, `IndexOutOfBoundsException`, `NullPointException` 중 하나 가 될 것이다. (아이템 72)
    - 매개변수의 제약을 문서화 한다면 그 제약을 어겼을 때 발생하는 예외도 함께 기술해야 한다.
    - 이런 간단한 방법으로 API 사용자가 제약을 지킨 가능성을 높일 수 있다.
    - ex) `mod` 연산

```
/*
 * 현재 값 mod m 값을 반환한다. 이 메서드는
 * 항상 음이 아닌 BigInteger 를 반환한다는 점에서 remainer 메서드와 다름
 *
 * @param m 계수 (반드시 양수)
 * @return 현재 값 mod m
 * @throws ArithmeticException m 이 0보다 작거나 같으면 발생
 */
public BigInteger mod (BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("계수 (m)는 양수여야 합니다");
    ...
}
```

        - 만약 `m` 이 `null` 이면 `signum` 호출시 `NullPointerException` 을 던진다.

        - 해당 내용은 메서드 설명에 없다.
          왜냐면 (개별 메서드가 아닌) `BigInteger` 클래스 수준에서 기술했기 때문이다.
        - `@Nullable` 이나 이와 비슷한 애너테이션을 사용해 특정 매개변수는 `null` 의 가능성을 알려줄 수 있지만, 표준적인 방법이 아니다.
    - 클래스 수준 주석은 그 클래스의 모든 `public` 메서드에 적용되므로 각 메서드에 일일이 기술하는 것 보다 훨씬 깔끔한 방법이다.

- `java.util.Objects.requireNonNull`
    - Java 7 에 추가된 메서드이며, 유연하고 사용하기 편하니 ,더 이상 수동으로 `null` 검사를 하지 않아도 된다. 원하는 예외 메시지도 지정할 수 있다.
    - 입력을 그대로 반환함으로 값을 사용하는 동시에 `null` 검사를 수행 할 수 있다.

```
public static <T> T requireNonNull(T obj) {
    if (obj == null)
        throw new NullPointerException();
    return obj;
}

public static <T> T requireNonNull(T obj, String message) {
    if (obj == null)
        throw new NullPointerException(message);
    return obj;
}
```

- `Objects` 의 `checkFromIndexSize`, `checkFromToIndex`, `checkIndex` 메서드
    - Java 9 에 범위 검사 기능으로 추가된 메서드
    - `null` 검사 메소드 만큼 유연하지는 않고, 예외 메시지를 지정할 수 없고, 리스트와 배열 전용으로 설계되었다.
    - 닫힌 범위 (closed range: 양 끝단 값을 포함)는 다루지 못한다.
- `public` 이 아닌 메서드라면 단언문(`assert`)을 사용해 매개변수 유호성을 검증하자
    - `public` 이 아닌 메서드라면 메서드가 호출되는 상황을 통제할 수 있음으로, 오직 유효한 값만이 메서드에 넘겨지리라는 것을 보증할 수 있다.
    - ex) 재귀 정렬용 `private` 도우미 함수

```
private static void sort(long a[], int offset, int length) {
    assert a != null;
    assert offset >= 0 && offset <= a.length;
    assert length >= 0 && length <= a.length - offset;
    ...
}
```

- 단언문들은 자신이 단언한 조건이 무조건 참이라고 선언한다는 것이다.
- 단언문은 일반적인 유효성 검사와 다르다.
  - 실패하면 `AssertionError` 를 던진다.
  - 런타임에서 아무런 효과도, 아무런 성능 저하도 없다.
    (java 실행시 `-ea`, `-enableassertions` 플래그를 설정하면 런타임에 영향 줌)

- 매개변수 유효성을 검사 규칙 예외
  - 유효성 검사 비용이 지나치게 높거나 실용적이지 않을 때, 혹은 계산과정에서 암묵적으로 검사가 수행될 때다.
  - 하지만, 암묵적 검사에 의존했다가는 실패 원자성을 해칠 수 있다.
- 생성자
  - 생성자는 “나중에 쓰려고 저장하는 매개변수의 유효성을 검사하라”는 원칙의 특수한 사례이다.
  - 생성자 매개변수의 유효성 검사는 클래스 불변식을 어기는 객체가 생성되지 않도록 해준다.
- 정리
  - 메서드나 생성자를 작성할 때면 그 매개변수들에 어떤 제약이 있을지 생각해야한다.
  - 그 제약들을 문서화하고 일반적으로 메서드 시작 부분에서 명시적으로 검사해야 한다.
  - 유효성 검사시 실제 오류를 걸러낼 때 빛을 본다. (Spring 의 `@Vaild`)
  - 이번 아이템은 “매개변수에 제약을 두는게 좋다” 로 해석하면 안된다.
    사실 그 반대로 메서드는 최대한 범용적으로 설계해야 한다.

▼ 50. 적시에 방어적 복사본을 만들라

- 자바는 안전한 언어인가?
  - 네이티브 메서드를 사용하지 않아, 버퍼 오버런, 배열 오버런 등 메모리 충돌 오류에서 안전하지만, 사용자가 어떻게든 불변식을 깨뜨린다고 가정하고 방어적 프로그래밍을 해야한다.
- ex) 불변식을 지키지 못하는 클래스 - 기간 표현 클래스

```
public final class Period {
    private final Date start;
    private final Date end;
```

```
    /** (item 49)
     * @param start 시작 시각
     * @param end 종료 시각; 시작 시각보다 뒤어야 한다.
     * @throws IllegalArgumentException 시작 시각이 종료 시각보다 늦을 때 발생한다.
     * @throws NullPointerException start 나 end 가 null 이면 발생한다
    */
    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0)
            throw new IllegalArgumentException(start + "가" + end + "보다 늦다");
        this.start = start;
        this.end = end;
    }
    public Date start() {
        return start;
    }
    public Date end() {
        return end;
    }
    ...
}

class Item50 {
    public static void main(String[] args) {
        Date start = new Date();
        Date end = new Date();
        Period period = new Period(start, end);
        end.setYear(78); // period의 내부를 수정했다.
    }
}
```

  - 해당 클래스는 불변처럼 보이고, 불변식이 지켜질 것 처럼 보이지만, `Date` 클래스가 가변임으로 쉽게 불변식을 깨뜨릴 수 있습니다.
    (객체를 재 생성하는 것이 아닌 참조, 참조 타입이 아닌 값 타입이라면 저렇게 해도 됨)
  - java 8 이후로는 `Date` 대신 불변(아이템 17) 인 `Instant` (`LocalDateTime`, `ZonedDateTime`)을 사용하면 된다.
  - **외부 공격으로 부터 `Period` 인스턴스의 내부를 보호하려면 생성자에서 받은 가변 매개변수 각각을 방어적으로 복사(defensive copy)해야 한다.**

- ex) 생성자 - 매개변수의 방어적 복사본을 만든다.

```
public Period(Date start, Date end) {
    this.start = new Date(start.getTime());
    this.end = new Date(end.getTime());

    if (this.start.compareTo(this.end) > 0)
        throw new IllegalArgumentException(this.start + "가" + this.end + "보다 늦다");
}
```

    - **매개변수 유효성 검사(아이템 49)하기 전에 방어적 복사본을 만들고, 이 복사본으로 유효성을 검사해야한다.**
        - 멀티스레드 환경이라 가정했을 때, 원본 객체의 유효성을 검사한 후 본사본을 만드는 그 찰나의 취약한 순간에 다른 스레드가 원본 객체를 수정할 위험이 있기 때문이다.
        - 방어적 복사를 매개변수 유효성 검사 전에 수행하면 이런 위험에서 해방될 수 있다.
            - 이러한 공격을 검사시점/사용시점(time-of-check/time-of-use) 줄여서 TOCTOU 공격이라 한다.
    - 방어적 복사에 `Date`의 `clone` 메서드를 사용하지 않는 점
        - `Date`는 `final` 클래스가 아니므로 상속이 가능한 타입이라면 `clone` 을 사용하면 안됩니다.
        - 즉, `clone` 이 악의를 가진 하위 클래스의 인스턴스를 반환할 수도 있다.
        - **매개변수가 제 3자에 의해 확장될 수 있는 타입이라면 방어적 복사본을 만들 때, `clone` 을 사용해서는 안된다.**
- ex) 접근자 메서드 변경 - 방어적 복사본 반환

```
public Date start() {
    return new Date(start.getTime());
}
public Date end() {
    return new Date(end.getTime());
}
```

    - 생성자와 달리 접근 메서드에서는 방어적 복사에 `clone` 을 사용해도 된다.
        - `Period` 가 가지고 있는 `Date` 객체는 `java.util.Date` 임이 확실하기 때문이다.
        - 그렇더라도, 인스턴스를 복사하는 데는 일반적으로 생성자나 정적 팩터리를 사용하는게 좋다. (아이템 13)
- 매개변수를 방어적으로 복사하는 다른 이유 (불변 객체 생성이 아닌 다른 이유)
    - 메서드든 생성자든 클라이언트가 제공한 객체의 참조를 내부의 자료구조에 보관해야 할 때면 항시 그 객체가 잠재적으로 변경될 수 있는지를 생각해야 한다.
        - 변경될 수 있는 객체라면, 그 객체가 클래스에 넘겨진 이후 임의로 변경되어도 그 클래스가 문제없이 동작할지를 따져보라. 확신할 수 없다면 복사본을 만들어 저장한다.

        - 클라이언트가 건네준 객체를 내부의 `Set` 인스턴스에 저장하거나 `Map` 의 인스턴스 key로 사용한다면, 추후 그 객체가 변경될 경우 객체를 담고 있는 `Set`, `Map` 불변식이 깨질 것이다.
- 방어적 복사 생략
    - 메서드나 생성자의 매개변수로 넘기는 행위가 그 객체의 통제권을 명백히 이전함을 뜻하기도 한다. 통제권을 이전하는 메서드를 호출 하는 클라이언트는 해당 객체를 더 이상 직접 수정하는 일이 없다고 해야한다.
    - 클라이언트가 건네주는 가변 객체의 통제권을 넘겨받는 메서드나 생성자에게도 그 사실(해당 객체가 직접 수정하는 일)을 확실히 문서에 기재해야한다.
    - 통제권을 넘겨받은 메서드나 생성자를 가진 캘래스들은 악의적인 클라이언트의 공격에 취약하다.
    - 따라서 방어적 복사를 생략해도 되는 상황에는 해당 클래스와 클라이언트가 상호 신뢰되거나, 혹은 불변식이 깨지더라도 그 영향이 오직 호출한 클라이언트로 국한되어야 한다.
    - ex) Wapper Class Patten (아이템 18)
        - 래퍼 클래스의 특성상 클라이언트는 래퍼에 넘긴 객체에 여전히 직접 접근할 수 있다.
        - 래퍼의 불변식을 쉽게 파괴할 수 있지만, 그 영향은 오직 클라이언트 자신만 받는다.
- 정리
    - 클라이언트에 의해 객체 상태가 변경되지 않도록 주의해야 한다.
    - 클래스의 구성요소가 `final` 이 아니라면 그 요소는 반드시 방어적 복사, 정적 팩토리를 해야한다. (접근자 메서드에는 `clone` 를 사용해도 되지만, 웬만하면 사용X)
    - (Collection 객체를 담고 있는 클래스는) 복사 비용이 너무 클 수 있기 때문에, 불변 객체들로 객체를 구성해 방어적 복사를 하지 않는 게 좋다.
    - 방어적 메서드를 생략하는 경우에는 통제권 이전을 문서화를 해야하며, 불변식이 깨지는 영향이 있을 경우, 그영향이 호출한 클라이언트로만 국한되어야 한다.

▼ 51. 메서드 시그니처를 신중하게 설계하라

- API 설계 요령
    - 배우기 쉽고, 쓰기 쉬우며, 오류 가능성이 적은 API 목표

1. **메서드 이름을 신중히 짓자**
    - 항상 표준 명명 규칙(아이템 68)을 따라야 한다.

1. 패키지에 속한 다른 이름들과 일관되게 짓자.
2. 개발자 커뮤니티에서 널리 사용되는 이름을 사용하자

- 긴 이름은 피하자
- 이해하기 쉽게 네이밍을 하자

2. **편의 메서드를 너무 많이 만들지 말자**
   - 메서드가 너무 많은 클래스, 인터페이스는 익히고, 사용하고, 문서화하고, 테스트하고, 유지보수하기 힘들다.
   - 클래스나 인터페이스는 자신의 각 기능을 완벽히 수행하는 메서드로 제공해야 한다.
   - 자주 쓰이는 경우에만 별도의 약칭 메서드를 두자 (확신이 되지 않으면 만들지 말자)
3. **매개변수 목록을 짧게 유지하자.**
   - 4개 이하가 좋다.
     - (이노 회사)의 경우 3개 이상 사용되면, (Java - DTO, VO, JS - Map Object 로 바꿨다.)
   - 같은 타입의 매개변수가 여러 개가 연달아 나오는 경우가 특히 해롭다.
     - 실수로 순서를 바꿔 입력한 경우, 의도와 다르게 동작 가능성이 있다.
   - 매개변수 목록이 많은 경우 줄여주는 기술 (책 내용)
     1. 여러 메서드로 쪼갠다.
        - 쪼개진 메서드 각각은 원래 매개변수 목록의 부분집합을 받는다.
        - 잘못하면 메서드가 너무 많아질 수 있지만, 직교성을 높여 오히려 메서드 수를 줄여주는 효과도 있다. (`List` 인터페이스의 좋은 예이다)
        - ex) `List` 지정범위 원소의 인덱스를 찾아야하는 경우 시작점, 끝 점, 찾을 원소 총 3개의 매개변수가 필요하다. 하지만, `subList`, `indexOf` 를 사용하면 원하는 목적을 이룰 수 있다.

          ```
          List<Integer> list = List.of(1,2,3,4,5,6,7,8,9);
          List<Integer> subList = list.subList(1, 5);
          List<Integer> subList = list.indexOf(3);
          ```

     2. 매개변수 여러 개를 묶어주는 도우미 클래스를 만든다.
        - 일반적으로 이런 도우미 클래스는 정적 멤버 클래스(아이템 24)로 만든다.
        - 매개변수 몇 개를 독립된 하나의 개념으로 볼 수 있을 때 추천하는 기법

        - DTO, VO, …
    - 도우미 클래스를 만들어 하나의 매개변수로 주고 받으면 API 물론 클래스 내부 구현도 깔끔해진다.
3. 1, 2 번 방식을 혼합한 방식인 객체 생성에 사용한 빌더 패턴 (아이템 2)
    - 이 기법은 매개변수가 많을 때, 특히 그 중 일부를 생략해도 괜찮을 때 도움된다.
    - 모든 매개변수를 하나로 추상화한 객체를 정의하고, 클라이언트에서 이 객체의 세터(`setter`) 메서드를 호출해 필요한 값을 설정하고 `execute` 메서드를 호출해 메서드 매개변수의 유효성을 검사 후 생성된 객체를 넘긴다.

4. **매개변수의 타입으로는 클래스보다는 인터페이스가 낫다.**(아이템 64)
    - 매개변수로 적합한 인터페이스가 있다면 (이를 구현한 클래스가 아닌) 그 인터페이스를 직접 사용하자.
    - `Map` 을 예로 들면 여러 구현체 클래스인 `HashMap`, `TreeMap` 등이 있다. 만약 매개변수 타입으로 `HashMap` 으로 설정하면 `HashMap` 만 사용하는 메서드가 만들어지지만, `Map` 으로 둔다면, 어떤 구현체가 와도 동작 가능하게 할 수 있다.

5. `boolean` **보다는 원소 2개짜리 열거 타입이 낫다.**
    - `is___` 로 시작하는 메서드가 아니면 열거 타입이 좋다.
      (메서드 이름상 boolean 을 받아야 의미가 더 명확할 때는 예외)
    - 열거 타입을 사용하면 코드를 읽고 쓰기가 더 쉬워지고, 나중에 선택지를 추가하기도 쉽다.

> **직교**
> 소프트웨어 설계 관점으로 볼 때, “직교성이 높다”는 공통점이 없는 기능들이 잘 분리되어 있다. 또는 기능을 원자적을 쪼개 제공한다.
> 기능을 원자적을 쪼개다 보면, 자연스럽게 중복이 줄고 결합성이 낮아져 코드를 수정하기 수월해진다. **일반적으로 직교성이 높은 설계는 가볍고 구현하기 쉽고 유연하며 강력하다.**
> 그렇다고 무작정 나누는게 완벽한게 아니다. **API가 다루는 개념의 추상화 수준에 맞게 조절해야한다.** 특정 조합 패턴이 상당히 자주 사용되거나 최적화하여 성능을 크게 개선할 수 있다면, 직교성이 낮아지더라도 편의 기능으로 제공하는 편이 나을 수도 있다.

▼ **52. 다중 정의는 신중히 사용하라**

▼ **53. 가변인수는 신중히 사용하라**

▼ **54. null 이 아닌, 빈 컬렉션이나 배열을 반환하라**

▼ **55. 옵셔널 반환은 신중히 하라**

▼ **56. 공개된 API 요소에는 항상 문서화 주석을 작성하라.**