8장 메서드

아이템 49. 매개변수가 유효한지 검사하라.

메서드와 생성자 대부분은 입력 매개변수의 값이 특정 조건을 만족하기를 바란다.

이러한 경우에 오류는 **메서드 몸체가 시작되기 전에 매개변수를 검사하는 것이 좋다.**

- 매개변수 검사를 제대로 하지 못했을 때 발생할 수 있는 문제
  - 메서드가 수행되는 중간에 모호한 예외를 던지며 실패
  - 메서드가 수행되지만 잘못된 결과를 반환할 때
  - 메서드는 문제없이 수행되지만, 객체를 이상한 상태로 만들어 놓아서 미래에 메서드와는 관련 없는 오류 발생

Public & Protected 메서드

- public과 protected 메서드는 매개변수 값이 잘못 되었을 대 던지는 예외를 문서화 해야한다.
- 매개변수의 제약을 문서화한다면 그 제약을 어겼을 때 발생하는 예외도 함께 기술해야 한다.

```
/**
* (현재 값 mod m) 값을 반환한다. 이 메서드는
* 항상 음이 아닌 BigInteger를 반환한다는 점에서 remainder 메서드와 다르
*
* @param m 계수 (앙수여야 한다.)
* @return 현재 값 mod m
* @throws ArithmeticException m이 0보다 작거나 같으면 발생한다.
*/
public BigInteger mod(BigInteger m) {
    if (m.signum() < 0)
        throw new ArithmeticException("계수(m)는 양수여야 합니다.
    ...
}
```

자바 7에 추가된 java.util.Objects.requireNonNull 메서드는 null 검사를 수동으로 하지 않게 해준다.

```
this.startegy = Objects.requireNonNull(strategy, "전략");
```

private 메서드

공개되지 않은 메서드라면 메서드가 호출되는 상황을 통제할 수 있다. 오직 유효한 값 만이 메서드에 넘겨지는 것을 보증할 수 있어야한다.

```
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;
    ...//계산 수행
}
```

단언문은 유효성 검사와는 다르다.

- 실패하면 AssertionError을 던진다.
- 런타임에 아무런 효과도, 아무런 성능 저하도 없다.

메서드가 직접 사용하지는 않으나 나중에 쓰기 위해 저장하는 매개변수는 특히 더 신경써서 검사해야한다.

검사를 생략한다면 나중에 사용할 때 문제가 생겨도 추적하기 어렵다.

유효성 검사도 비용이 지나치게 높거나 실용적이지 않을 때, 혹은 계산 과정에서 암묵적으로 수행될 때는 하지 않아도 된다.

"매개변수에 제약을 두는게 좋다" 가 핵심이 아니다. 메서드는 범용적으로 설계해야한다. 하지만 구현하려는 개념 자체가 특정한 제약을 지닌다면 제약을 둬야한다.

아이템 50. 적시에 방어적 복사본을 만들라

자바는 안전한 언어이다. 네이티브 메서드를 사용하지 않기 때문에 메모리 충돌 오류로 부터 안전하다.

하지만 클라이언트가 불변식을 깨뜨리려 혈안이 되어있다고 가정하고 방어적으로 프로그래밍 해야한다.

어떤 객체든 그 객체의 허락 없이는 외부에서 내부를 수정하는 일은 불가능하다. 하지만 주의를 기울이지 않으면 자기도 모르게 내부를 수정하도록 허락하는 경우가 생긴다.

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

    /**
     * @param  start 시작 시각
     * @param  end 종료 시각. 시작 시각보다 뒤여야 한다.
     * @throws IllegalArgumentException 시작 시각이 종료 시각보다
     * @throws NullPointerException start나 end가 null이면 발생한
     */
    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;
    }

    public String toString() {
        return start + " - " + end;
    }
```

보기에는 불변처럼 보인다. 시작 시각이 종료 시각 시간보다 늦으면 안된다는 불변식이 지켜질 것 같지만 Date는 가변이다.

```
Date start = new Date();
Date end = new Date();
Period period = new Period(start, end);

// Period의 start()로 반환된 객체를 변경
period.start().setTime(end.getTime() + 10000);
System.out.println(period.start());  // 변경된 값 출력
```

자바 8 이후로는 불변인 Instant를 사용하면 된다.

외부 공격으로 부터 내부를 보호하려면 생성자에서 받은 기본 매개변수를 각각 방어적으로 복사해야한다.

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

    if (this.start.compareTo(this.end) > )
        throw new IllegalArgumentException(this.start + " aft
}
```

- start.getTime()를 기반으로 new Date()를 호출하면 start와 동일한 시간값을 갖는 새로운 Date객체가 생성된다.
- 새로 생성된 객체는 원래의 start와는 독립적이며, 원래 start 객체의 변경이 복사본에 영향을 미치지 않는다.
- this.start, this.end
    - 생성자는 외부로 부터 전달된 start와 end를 직접 저장하지 않고, 각각의 복사본을 생성하여 클래스 내부 필드인 this.start와 this.end에 저장한다.

매개변수의 유효성을 검사(아이템49)하기 전에 방어적 복사본을 만들고, 이 복사본으로 유효성을 검사한 점에 주목하자. 반드시 이렇게 작성해야 한다. 멀티스레딩 환경이라면 원본 객체의 유효성을 검사한 후 검사본을 만드는 찰나의 순간에 다른 스레드가 원본 객체를 수정할 위험이 있기 때문이다.

매개변수가 제 3자에 의해 확장될 수 있는 타입이라면 방어적 복사본을 만들 때 clone을 사용해서는 안된다.

```
// Period 인스턴스를 향한 두 번째 공격
Date start = new Date();
Date end = new Date();
Period p = new Period(start, end);
p.end().setYear(78);
```

두 번째 공격을 막으려면 접근자가 **가변 필드의 방어적 복사본을 반환하면 된다.**

```
// 수정한 접근자 - 필드의 방어적 복사본 반환
public Date start() {
    return new Date(start.getTime());
}

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

이제 더이상 자신 말고는 가변 필드에 접근할 방법이 없으니 모든 필드가 캡슐화되었다.

생성자와 달리 접근자 메서드에는 방어적 복사에 clone을 사용해도 된다.

길이가 1이상인 배열은 무조건 가변임을 잊지말자.

방어적 복사에는 성능 저하가 따르고 항상 쓸수 있는 것도 아니다. 방어적 복사를 생략할 때는 해당 매개변수가 반환값을 수정하지 말아야함을 명확히 문서화해야 한다.

아이템 51. 메서드 시그니쳐를 신중히 설계하라.

메서드 이름은 신중히 짓자

- 메서드 표준 명명규칙을 따르자
- 커뮤니티에서 널리 받아들여지는 이름을 사용하자.

편의 메서드를 너무 많이 만들진 말자

- 클래스에 메서드가 너무 많으면 다른 개발자가 알기도 힘들뿐더러 문서화 등 유지보수도 힘들다.

- 확신이 서지 않으면 만들지 말자.
- 편의 메서드란?
  - 말 그대로 편의를 위한 메서드 예를 들면, 헬퍼 클래스인 `Collections` 안에 있는 모든 메서드(`swap`, `min`, `max` 등등...)를 말한다.

매개변수 목록은 짧게 유지하자

- 일반적으로 4개 이하가 좋다.
- 같은 타입의 매개변수가 연속으로 나오면 특히 해롭다.
  - 실수로 순서를 바꿔 입력해도 그대로 컴파일되고 실행될 수 있다.

과하게 긴 매개변수 목록을 짧게 줄여주는 기술 세가지

- **1.메서드를 쪼개자**
  - 기능을 쪼개다보면 자연스럽게 중복이 줄고 결합성이 낮아집니다. 코드를 수정하고 테스트하기 쉬워진다. 하지만 무한정 작게 나누는 게 좋은 건 아니다.
  - 메서드의 직교성을 높여 메서드 수를 줄이는 효과가 있다.
  - 직교성 : 서로 영향을 주는 성분이 없다.
- **2.도우미 클래스를 만들자.**
  - 도우미 클래스의 다양한 필드들을 이용하고, 해당 클래스를 파라미터로 받자.
- **3.빌더 패턴을 이용하자.**
  - 먼저 모든 매개변수를 하나로 추상화한 객체를 정의합니다. 그리고 그 객체에 빌더 패턴을 적용합니다. 클라이언트에서 해당 객체의 setter를 호출해 필요한 값을 설정합니다. 마지막으로 `validate()`를 통해 필드 유효성 검사를 합니다. 객체를 넘겨 계산을 수행합니다. 매개 변수가 많으면서 그 중 일부는 생략해도 좋을 때 도움이 됩니다.
- **매개변수의 타입으로는 클래스보다 인터페이스가 더 낫다.**
  - 예를 들어, `HashMap이` 아니라 인터페이스인 Map을 사용하면 `TreeMap`, `ConcurrentHashMap`, `TreeMap` 등 어떤 `Map` 구현체라도 인수로 건넬 수 있다.
  - 인터페이스 대신 클래스를 사용하면 클라이언트에게 특정 구현체만 사용하도록 제한하는 꼴이 되며, 입력형태가 `HashMap`이 아니라 다른 형태로 존재한다면 다시 옮겨 담아야하니 그만큼의 복사 비용을 치뤄야한다.
- **`boolean`보다는 원소 2개짜리 열거 타입이 낫다.**

- 섭씨, 화씨를 예로 들면 true로 구분하지 않고 enum 타입으로 구분하자.
  - 코드의 가독성이 좋아질 뿐만 아니라, 다른 기준이 생겼을 때 확장하기도 좋다.

아이템 52. 다중정의는 신중히 사용하라.

다중정의(overloading)

- 이름은 같지만 매개변수의 타입과 개수가 다른 메서드를 여러개 가지는 것
- 컴파일 타입에 호출할 메서드가 정해진다.

```
public class Main {
    public static String classify(Set<?> set) {
        return "Set";
    }

    public static String classify(List<?> list) {
        return "List";
    }

    public static String classify(Collection<?> list) {
        return "Collection";
    }

    public static void main(String[] args) {
        Collection<?>[] collections = {
                new HashSet<>(),
                new ArrayList<>(),
                new HashMap<String, String>().values()
        };

        for (Collection<?> c : collections) {
            System.out.print(classify(c) + " ");
        }
```

```
    }
}
```

제시된 코드에서 출력 결과는 Collection Collection Collection이다.

왜 저렇게 출력될까?

- 메서드 오버로딩은 컴파일 시점에 결정된다.
- 배열의 타입이 Collection<?>으로 선언 되었다. 메서드 호출 시점에서 컴파일러는 Collection<?>타입으로 적합한 메서드를 찾는다.
- 컴파일러는 참조 변수의 타입을 기준으로 메서드를 선택하는데, 3가지 모두 참조 변수의 타입이 Collection<?>으로 선언되어 컬렉션이 출력된다.

```
class Wine {
    String name() { return "Wine"; }
}

class SparklingWine extends Wine {
    @Override String name() { return "SparklingWine"; }
}

class Champagne extends SparklingWine {
    @Override String name() { return "Champagne"; }
}

public class Main {
    public static void main(String[] args) {
        List<Wine> wines = Arrays.asList(new Wine(), new Spar

        for (Wine wine : wines) {
            System.out.print(wine.name() + " ");
        }
    }
}
```

하지만 오버라이딩은 런타임에 동적으로 선택된다. 여기서는 `wine, SparklingWine, Champagne`으로 출력된다. 앞의 예시와 같이 `for` 문에서 상위 타입으로 순회 했음에도 앞선

메서드와 달리 오버라이드 된 메서드가 호출된다.

오버로딩된 메서드는 정적으로 선택되고, 오버라이딩된 메서드는 동적으로 선택된다.

- 오버로딩은 컴파일 시점에 메서드가 선택되고, 오버라이딩은 런타임 시점에 객체의 실제 타입에 따라 메서드가 선택된다.

그럼 어떻게 해야할까?

- 오버로딩이 혼동이 일으키는 상황을 피하자.
  - 안전하게 오버로딩하기 위해 매개변수 수가 같은 오버로딩 메서드는 최대한 피하자.
  - 매개변수의 수가 같다면, 오버로딩이 아니라 이름을 다르게 부여하는 방법을 생각해보자. 생성자는 이름을 변경할 수 없으니 정적 팩터리 메서드를 고려해보자.

주의!!

- 원시타입과 객체는 근본적으로 다르지만 자바5에서 오토박싱이 도입되면서 상황이 복잡해졌다.

```
boolean remove(Object o);
E remove(int index);
```

- `List` 인터페이스에 정의된 `remove` 메서드이다. `Object`와 `int`는 근본적으로 다르기 때문에 오버로딩해서 구현해도 문제가 없다. 하지만 `Object`가 `Integer`라면 오토박싱을 지원하면서 다르다고 할 수가 없다.

```
public static void main(String[] args) {
    Set<Integer> set = new TreeSet<>();
    List<Integer> list1 = new ArrayList<>();
    List<Integer> list2 = new ArrayList<>();

    for(int i = -3; i < 3; i++) {
        set.add(i);
        list1.add(i);
        list2.add(i);
    }
```

```
    for(int i = 0; i < 3; i++) {
        set.remove(i);
        list1.remove(i);
        list2.remove((Integer) i);
    }

    System.out.println(set);
    // [-3, -2, -1]
    System.out.println(list1);
    // [-2, 0, 2]
    System.out.println(list2);
    // [-3, -2, -1]
}
```

왜 List은 저렇게 반환될까?

- 루프를 돌면서 첫번째 루프에서 0번 인덱스 제거, 두번째 루프에서 1번 인덱스, 세번째 루프에서 2번 인덱스를 제거 함으로써 저렇게 값이 나온다.

아이템53. 가변인수는 신중히 사용하라.

가변인수

- 가변인수는 명시한 타입의 인수를 0개 이상 받을 수 있다.

```
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

- 가변인수 메서드를 호출하면, 가장 먼저 인수의 개수와 길이가 같은 배열을 만들고 인수들을 배열에 저장하여 가변인수 메서드에 건네준다.

```
static int min(int... args) {
    if (args.length == 0)
        throw new IllegalArgumentException("인수가 1개 이상 필요함
    int min = args[0];
    for (int i = 1; i < args.length; i++)
        if (args[i] < min)
            min = args[i];
    return min;
}
```

- 최솟값을 찾는 메서드이다. 하지만 인수를 0개만 받을 수 있도록 설계하는 것은 좋지 않다. 인수 개수는 런타임에 자동생성된 배열의 길이로 알 수 있다.
- 이 코드에서는 인수를 0개만 넣어 호출하면 런타임에 실패한다는 문제점이 있다.
- 코드도 깔끔하지 않다.
    - args 유효성 검사를 명시적으로 해야하고, min의 초깃값을 Integer.MAX_VALUE로 설정하지 않고는 for-each문도 사용할 수 없다.
        - args = 가변 인자를 받는 배열, 비어 있는 경우 메서드의 로직이 동작하지 않으므로 예외를 발생시켜야한다.
        - 비어있는 배열로는 최소값을 계산할 수 없다.
        - 기존 코드에서는 최소값을 찾기위해 배열의 첫번째 값을 초기값으로 사용한 후 루프를 돌면서 최소값을 업데이트한다. for-each문은 첫번째 값을 미리 초기화 하지 않고 모든 요소를 순회하기 때문에 별도로 초기값 설정이 필요하다. for-each는 초기값 설정이 어렵기 때문에 Integer.MAX_VALUE를 이용해 비교가 필요하다.

```
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}
```

- 다음 코드 처럼 매개변수를 2개 받도록 하면 된다. 매개변수 하나, 가변인수를 두번째로 받으면 앞에서의 문제가 해결된다.

- 하지만 가변인수 메서드는 호출될 때마다 배열을 새로 하나 할당하고 초기화하기 때문에 성능 비용이 발생할 수는 있지만 가변인수의 유연성이 필요하다면 고려해볼만 하다.

아이템 54.null이 아닌, 빈 컬렉션이나 배열을 반환하라.

- 왜 null을 반환하면 안될까?

```
    public List<Cheese> getCheeses() {
        return cheesesInStock.isEmpty() ? null : new ArrayLis
    }
```

해당 코드는 창고에 저장되어 있는 치즈를 확인한 후 비어있다면 null을 리턴하고 그렇지 않다면 저장된 치즈가 담긴 새로운 리스트를 반환한다.

null을 반환하는 메서드를 사용할 때에는 아래와 같은 방어코드를 넣어줘야한다.

```
List<Cheese> cheeses = shop.getCheeses();
            if (cheeses != null && cheeses.contains(Cheese.ST
                    System.out.println("좋았어, 바로 그거야.");
```

클라이언트에 방어 코드를 빼먹으면 오류가 발생할 수도 있다. 실제로 객체가 0개일 가능성이 거의 없는 상황에서는 수년 뒤에야 오류가 발생하기도 한다.

- 빈 컨테이너를 할당하는 데도 비용이 드니 null을 반환하는 것이 더 나을까?
  - 성능 분석 결과 이 할당이 성능 저하의 주범이라고 확인되지 않는 한, 이 정도의 성능 차이는 신경 쓸 수준이 못 된다.
  - 빈 컬렉션과 배열은 굳이 새로 할당하지 않고도 반환할 수 있다.

```
  public List<Cheese> getCheeses() {
        return new ArrayList<>(cheesesInStock);
    }
```

  빈 컬렉션을 반환하는 코드. 대부분의 상황에서는 이렇게 하면 된다

```
    public Cheese[] getCheeses() {
        return cheesesInStock.toArray(new Cheese[0]);
    }
```

배열을 사용할 때도 null을 반환하지 말고 길이가 0인 배열을 반환하자.

길이가 0인 배열이 주어졌을 경우, 만약 컬렉션에 하나라도 원소가 들어있다면 새로운 배열이 만들어져서 리턴되고, 원소가 없다면 인자로 전달된 크기가 0인 배열이 리턴되는 것이다.

아이템 55. 옵셔널 반환은 신중히 하라.

자바 8 이전에는 값을 반환할 수 없는 상황에서 null을 반환하거나 예외를 던지는 선택지가 있었다. null을 반환하는 경우에는 null 처리 코드가 필요하다. 예외는 진짜 예외적인 상황에서마 사용해야하며, 예외를 생성할 때 스택 추적 전체를 캡쳐하므로 비용도 만만치 않다.

자바 8에 도입된 Optional<T>는 null이 아닌 T 타입 참조를 하나 담거나, 혹은 아무것도 담지 않을 수 있다. 옵셔널은 원소를 최대 1개 가질 수 있는 불변 컬렉션이다.

보통은 T를 반환해야 하지만 특정 조건에서 아무것도 반환하지 않아야 할 때, T 대신 Optional<T>를 반한하도록 하면 된다. 옵셔널은 예외를 던지는 메서드보다 유연하고, null을 반환하는 메서드보다 안전하다.

```
public static <E extends Comparable<E>> E max(Collection<E
        if (c.isEmpty())
        throw new IllegalArgumentException("Empty Collecti

        E result = null;
        for (E e : c)
        if (result == null || e.compareTo(result) > 0)
        result = Objects.requireNonNull(e);

        return result;
    }
```

위의 코드는 컬렉션에서 최댓값을 뽑아주는 코드이다. 컬렉션이 비어있으면 예외를 던지고 있다.

```
 public static <E extends Comparable<E>> Optional<E> max(C
        if (c.isEmpty())
            return Optional.empty();
```

```
        E result = null;
        for (E e : c)
            if (result == null || e.compareTo(result) > 0)
                result = Objects.requireNonNull(e);

        return Optional.of(result);
    }
```

옵셔널을 적용한 코드이다. 반환 타입을 옵셔널로 감싸고, 리턴문에서 옵셔널의 정적 팩터리 메서드를 통해 옵셔널을 반환하고 있다.

`ofNullable` 메서드를 통해 null값도 옵셔널로 집어넣을 수 있다. 그러나 옵셔널을 반환하는 메서드에서는 절대 null을 반환하지 말자. 옵셔널의 도입 취지를 완전히 무시하는 것이다.

- 옵셔널은 검사예외와 취지가 비슷하다. 반환값이 없을 수도 있음을 명확히 알려준다.

메서드가 옵셔널을 반환한다면 클라이언트는 값을 받지 못했을 때 취할 행동을 선택해야 한다.

- 기본값으로 정하기

```
String lastWordInLexicon = max(words).orElse("Empty");
```

- 원하는 예외 던지기

```
Toy myToy = max(toys).orElseThrow(TemperTrantrumException:
```

- 값이 채워져 있다고 확신하면 곧바로 꺼내 사용

```
Element lastNobleGas = max(Elements.NOBLE_GASES).get();
```

- **주의사항**

  옵셔널로 감싼다고 무조건 득이 되는 건 아니다. 컬렉션, 스트림, 배열, 옵셔널 같은 컨테이너 타입은 옵셔널로 감싸면 안 된다. 아이템 54에서 다룬 것처럼,
  빈 `Optional<List<T>>`를 반환하기보다, 그냥 빈 리스트를 반환하는 것이 좋다.

결과가 없을 수 있고, 클라이언트가 이 상황을 특별히 처리해야 하는 경우에 `T` 대신 `Optional<T>`를 반환하게 만드는 것이 좋다.

박싱된 옵셔널을 반환하는 일은 없도록 하자. `OptionalInt`, `OptionalLong`, `OptionalDouble`을 사용하자.

마지막으로 옵셔널을 컬렉션의 키, 값 원소나 배열의 원소로 사용하는게 적절한 상황은 거의 없다는 점에 유의하자.

아이템 56. 공개된 API 요소에는 항상 문서화 주석을 작성하라.

Javadoc

- Javadoc은 프로그래머가 자바독 문서를 올바르게 작성했는 지 확인하는 기능을 제공한다.
- 자바 7에서는 명령줄에서 -Xdoclint 스위치를 켜주면 이 기능이 활성화되고, 자바 8부터는 기본으로 작동한다.
- 소스코드 파일에서 문서화 주석이라는 특수한 형태로 기술된 설명을 추려 API 문서로 변환해준다.
- API를 올바르게 문서화하려면 공개된 모든 클래스, 인터페이스, 메서드, 필드 선언에 문서화 주석을 달아야 한다.
- 직렬화할 수 있는 클래스라면 직렬화 형태에 관해서도 적어야 한다.
- 메서드용 문서화 주석에는 해당 메서드와 클라이언트 사이의 규약을 명료하게 기술해야 한다.
    - 상속용으로 설계된 클래스의 메서드가 아니라면 무엇을 하는지를 기술해야 한다. (즉, how가 아닌 what을 기술해야 한다.)
    - 문서화 주석에는 클라이언트가 해당 메서드를 호출하기 위한 전제조건을 모두 나열해야한다.
    - 또한 메서드가 성공적으로 수행된 후에 만족해야 하는 사후조건도 모두 나열해야 한다.
    - 일반적으로 전제조건은 @throws 태그로 비검사 예외를 선언하여 암시적으로 기술한다. 비검사 예외 하나가 전제조건 하나와 연결되는 것이다.
    - @param 태그를 이용해 그 조건에 영향받는 매개변수에 기술할수도 있다.
- 전제조건과 사후조건 뿐만 아니라 부작용도 문서화해야 한다.

- 부작용이란, 사후조건으로 명확히 나타나지는 않지만 시스템의 상태에 어떠한 변화를 가져오는 것을 뜻한다.
- 문서화 주석에 HTML 태그를 쓰기도 한다. Javadoc 유틸리티는 문서화 주석을 HTML로 변환하기 때문이다.
- 관례상, 인스턴스 메서드의 문서화 주석에 쓰인 “this”는 호출된 메서드가 자리하는 객체를 가리킨다.

Javadoc 주석 태그

- @param
  - 매개 변수를 설명모든 매개변수에 설명을 달아야 함설명에 마침표를 붙이지 않는다.
- @return
  - 반환 타입이 void가 아니라면 무엇을 return 하는 지 기술설명에 마침표를 붙이지 않는다.
- @throws
  - 발생할 가능성이 있는 검사든 비검사든 모든 예외를 기술설명에 마침표를 붙이지 않는다.
- {@code}
  - 주석 내에 HTML 요소나 다른 Javadoc 태그를 무시함주석에 여러 줄로 된 코드 예시를 넣으려면 {@code + 코드}
- {@literal}
  - HTML 마크업이나 Javadoc 태그를 무시하게 해준다.위의 {@code} 태그와 비슷하지만 코드 폰트로 렌더링하지 않는다.
- @implSpec
  - 해당 메서드와 하위 클래스 사이의 계약을 설명하여, 하위 클래스들이 그 메서드를 상속하거나 super 키워드를 이용해 호출할 때 그 메서드가 어떻게 동작하는지를 알려준다.
- {@index}
  - 클래스, 메서드, 필드 같은 API 요소의 색인은 자동으로 만들어지며, 원한다면 {@index} 태그를 사용해 API에서 중요한 용어를 추가로 색인화할 수 있다.
- {@inheritDox}

    - 상위 타입의 문서화 주석 일부를 상속할 수 있다. 즉, 클래스는 자신이 구현한 인터페이스의 문서화 주석을 재사용할 수 있다는 것이다.

주석은 어떻게 작성해야할까?

- 제네릭 타입이나 제네릭 메서드를 문서화할 때는 모든 타입 매개변수에 주석을 달아야 한다.
- 열거 타입을 문서화할 때에 상수들에도 주석을 달아야 한다.
- 애너테이션 타입을 문서화할 때는 멤버들에도 모두 주석을 달아야 한다.
- 패키지를 설명하는 문서화 주석은 package-info.java 파일에 작성한다.
- 클래스 혹은 정적 메서드가 스레드 안전하든 그렇지 않든 스레드 안전 수준을 API 설명에 포함해야 한다.