EJ

이펙티브 자바 CP.7

작성 일시	@2023년 2월 22일 오후 9:28
최종 편집 일시	@2023년 3월 4일 오전 2:41
유형	이펙티브 자바
작성자	종현 박
참석자	
언어	

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 떄다.
    - 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 떄 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

- ex) 컬렉션 분류기 - 오류! 이 프로그램은 무엇을 출력할까?

```
public class CollectionClassFier {
    public static String classify(Set<?> s) {
        return "집합";
    }
    public static String classify(List<?> list) {
        return "리스트";
    }
    public static String classify(Collection<?> c) {
        return "그 외";
    }

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

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

  - 집합, 리스트, 그 외 를 차례로 출력할 것 같지만, 실제로 수행해보면 “그 외”만 세 번 연달아 출력한다.
  - **다중정의된 메서드는 어느 메서드를 호출할지가 컴파일타임에 정해지기 때문이다.**
    - 컴파일 타임에는 `for` 문 안의 `c` 는 항상 `Collection<?>` 타입이다.
    - 런타임에는 타입이 매번 달라지지만, 호출할 메서드를 선택하는 데는 영향을 주지 못한다.
  - 이 처럼 직관과 어긋나는 이유는 **재정의한 메서드는 동적으로 선택되고, 다중정의한 메서드는 정적으로 선택되기 때문이다.**

- ex) 재정의 메서드 호출 매커니즘

```
class Wine {
    String name() { return "포도주"; }
}
class SparklingWine extends Wine {
    @Override String name() { return "발포성 포도주"; }
}
class Champagne extends SparklingWine {
    @Override String name() { return "샴페인"; }
}
public class Overriding {
    public static void main(String [] args) {
```

```
        List<Wine> wineList = List.of(new Wine()
            new SparklingWine(), new Champagne());

        for (Wine wine: wineList)
            System.out.println(wine.name());
    }
}
```

  - 결과로 포도주, 발포성 포도주, 샴페인을 차례로 출력한다.
  - **`for` 문에서의 컴파일타임 타입이 모두 `Wine` 인 것과 무관하게 런타임에서 항상 가장 하위에서 정의한 재정의 메서드가 실행됨**
    - 다중정의된 메서드 사이에서는 객체의 런타임 타입은 전혀 중요치 않다. 선택은 컴파일 타임에, 오직 매개변수의 컴파일 타임 타입에 의해 이뤄진다.

- **다중정의가 혼동을 일으키는 상황을 피하자**
  - 프로그래머가 다중정의를 함으로써 런타임시 의도되로 동작하지 않아 보일 수 있어, 혼동을 일으킬 수 있다.
  - **안전하게 개발하려면 매개변수 수가 같은 다중정의는 만들지 말자.**
  - 가변인수를 사용하는 메서드라면 다중정의를 아예 하지 말아야 한다.(아이템 53)
  - **매개변수 가 같은 메서드는 다중정의 대신 메서드 이름을 다르게 지어주는 방법도 있다.**
  - ex) `ObjectOutputStream` 클래스

```
public class ObjectOutputStream
    extends OutputStream implements ObjectOutput, ObjectStreamConstants
{
    private final BlockDataOutputStream bout;

    ...
    public void writeInt(int val)  throws IOException {
        bout.writeInt(val);
    }
    public void writeLong(long val)  throws IOException {
        bout.writeLong(val);
    }
    public void writeDouble(double val) throws IOException {
        bout.writeDouble(val);
    }
    public void writeBytes(String str) throws IOException {
        bout.writeBytes(str);
    }
    ...
}
```

        - `write…` 메서드는 모두 같은 매개변수 갯수를 가진다. 다중 정의가 아닌 네이밍을 통해 메서드의 의도를 예상할 수 있다.
- 한편, 생성자의 경우는 이름을 다르게 지울 수 없으니 두 번째 생성자 부터는 무조건 다중 정의이다.
    - 정적 팩터리라는 대안을 활용할 경우가 많다. (아이템 1)
    - 또, 생성자는 재정의 할 수 없으니, 다중정의와 재정의가 혼용될 걱정은 없다.
    - 여러 생성자가 같은 수의 매개변수를 받아야 할 경우의 안전 대책
        - 매개변수 집합을 처리할지가 명확히 구분된다면, 헷갈릴 일은 업을 것이다. 즉, 매개변수 중 하나 이상이 근본적으로 다르다면 헷갈릴 일 없다.
          근본적으로 다르다. 서로 어느 쪽으로든 형변환이 불가하다.
          이 조건이 충족하면, 컴파일 타입에는 영향을 받지 않게 됨으로, 혼란을 주는 원인이 사라진다.
            - `ArrayList(int)` 생성자와 `ArrayList(Collection)` 생성자는 어느 것이 호출될지 헷갈릴 일은 없다.
- **AutoBoxing** 의 등장으로 주의점
    - Java 5 부터 등장한 오토박싱으로 문제가 발생할 수 있다.
    - ex) `SetList`

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

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

        for (int i = 0; i < 3; i++) {
            set.remove(i);
            list.remove(i);
        }
        System.out.println(set + " " + list);
    }
}
```

        - 0, 1, 2 를 제거 한 후 [-3, -2, -1] [-3, -2, -1] 을 출력할 것이라 예상할 것이다.
        - 실제 값으로는 [-3, -2, -1] [-2, 0, 2] 출력한다.
            - `set.remove(i)` 는 `remove(Object)` 이다.
            - `list.remove(i)` 는 `remove(int index)` 이다.

        - 해결법으로는 `list.remove((Integer) i);` 로 하면된다.
            - `List<E>` 인터페이스가 `remove(Object)`, `remove(int)`를 다중 정의했기 때문이다. Java 4 까지 `List` 에서는 `Object` 와 `int` 가 근본적으로 달라서 문제 없었다.
            - 자바 언어에 제네릭과 오토박싱을 더한 결과 `List` 인터페이스가 취약해졌다.
- **람다와 메서드 참조** 의 등작으로 주의점
    - Java 8 부터 등장한 람다와 메서드 참조로 문제를 발생할 수 있다.

```
// 1번 Thread 생성자 호출
new Thread(System.out::println).start();

// 2번 ExecutorService 의 submit 메서드 호출
ExecutorSerivce exec = Executors.newCachedThreadPool();
exec.submit(System.out::println);
```

    - 양쪽 모두 `Runnable` 을 받는 형제 메서드를 다중정의하고 있다. 하지만 2번에서만 컴파일 오류가 난다. 원인은 바로 `submit` 다중정의 메서드 중에는 `Callable<T>` 를 받는 메서도 있다는데 있다.
    - `println` 이 `void` 를 반환하니, 반환값이 있는 `Callable`과 헷갈릴 리는 없다고 생각할지도 모르겠다. 합리적인 추론이지만, 다중정의 해소(적절한 다중정의 메서드를 찾는 알고리즘)는 이렇게 동작하지 않는다.
    - 원인은 `println` 메서드와 호출한 `submit` 메서드 양쪽 다 다중정의되어, 다중정의 해소 알고리즘이 우리의 기대처럼 동작하지 않는 상황이다.
    - `System.out::println` 은 부정확한 메서드 참조이다.
      암시적 타입 람다식, 부정확한 메서드 참조 같은 인수 표현식은 목표 타입이 선택되기 전에는 그 의미가 정해지지 않기 때문에 적용성 테스트가 무시된다. 이게 문제의 원인이다. - 책에서는 컴파일러 제작자를 위한 설명이라고 한다.
    - 핵심은 다중정의된 메서드(생성자)들이 함수형 인터페이스를 인수로 받을 때, 비록 서로 다른 함수형 인터페이스라도 인수 위치가 같으면 혼란이 생긴다.
    - **메서드를 다중정의할 때, 서로 다른 함수형 인터페이스라도 같은 위치의 인수로 받아서는 안 된다.**
- 다중정의를 해야 할때
    - 이미 만들어진 클래스의 경우에 다중 정의를 어기고 싶을 수 있다.
    - 예를 들어 Java 4 부터 `String` 은 `contentEquals(StringBuffer)` 메서드를 가지고 있었다. Java 5 때 `StringBuffer`, `StringBuilder`, `String`, `CharBuffer` 등 의 등장으로

비슷한 부류의 타입을 위한 공통 인터페이스로 `CharSequence`가 등장하였고, 자연스럽게 `String` 에도 `CharSequence`를 받은 `contentEquals` 가 다중 정의되었다.

- 이런 경우 어떤 다중 정의가 선택되더라도 기능이 동일하다면 문제가 발생하지 않는다. 일반적인 방법은 상대적으로 더 특수한 다중 정의 메서드에서 덜 특수한(더 일반적인) 다중정의 메서드로 일을 넘겨 버리는 것이다.

```
public boolean contentEquals(StringBuffer sb) {
    return contentEqulas((CharSequnce) sb);
}
```

- 정리
  - 프로그래밍 언어가 다중정의를 허용한다고 해서 다중정의를 꼭 활용하라는게 아니다.
  - 재정의한 메서드는 동적으로 런타임에 선택되고,
    다중정의한 메서드는 정적으로 컴파일 타입에 정적으로 선택된다.
  - 일반적으로 매개변수 수가 같을 때는 다중정의를 피하는 게 좋다.
    - 가변 인수를 사용하는 메서드는 다중 정의 하지말자
    - 매개변수 수가 같다면 네이밍으로 다중 정의를 피하자.
    - 다중 정의를 사용해야 할 때
    - 그럴 때는 헷갈릴 만한 매개변수는 형변환하여 정확한 다중정의 메서드가 선택되도록 해야 한다.
    - 이것이 불가능하다면, 예컨대 기존 클래스를 수정해 새로운 인터페이스를 구현해야 할 때는 같은 객체를 입력받는 다중 정의 메서드들이 모두 동일하게 동작하게 만들어야 한다.
  - 매개변수로 서로 다른 함수형 인터페이스라도 같은 위치의 인수로 받아서는 안된다.
  - 매개변수 중 하나 이상이 근본적으로 다르다면 헷갈릴 일 없다.

▼ 53. 가변인수는 신중히 사용하라

- 가변 인수 메서드
  - 매개변수를 동적으로 받을 수 있는 방법으로 0개 이상 받을 수 있음.
  - ex) `int` 인수들의 합을 계산해주는 가변인수 메서드

```
static int sum(int... args) {
    int sum = 0;
```

```
    for (int arg : args)
        sum += arg;
    return sum;
}
```

- 문제가 되는 경우
  - 인수를 0개만 받을 수도 있도록 하는 설계는 좋지 않다.
  - 인수 개수는 런타임에 (자동 생성된) 배열의 길이로 알 수 있다.
  - ex) 인수가 1개 이상이어야 하는 가변인수 메서드 - 잘못 구현된 예

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

    - **인수를 0개만 넣어 호출하면 (컴파일 타임이 아닌) 런타임에 실패한다.**
    - `args` 유효성 검사를 명시적으로 해야 하고, `min` 의 초기값을 `Integer.MAX_VALUE` 로 설정하지 않고는 (더 명료한) `for-each` 문도 사용할 수 없다.
- 해결 방법
  - 매개변수를 2개 받도록 하는 메서드
  - ex) 인수가 1개 이상이어야 할 때, 가변인수를 제대로 사용하는 방법

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

    - 첫 번째 인수로 평범한 매개변수, 두 번째 인수로 가변인수로 받는다.
- 가변인수 특징
  - 인수 개수가 정해지지 않았을 때 아주 유용하다.
    - `printf` 는 가변 인수와 한 묶음으로 자바에 도입되었고, 이때 핵심 리플렉션 기능(아이템 65) 도 재정비 되었다.

- 성능 문제
  - 가변인수 메서드는 호출될 때마다 배열을 새로 하나 할당하고 초기화한다.
  - 다행히, 이 비용을 감당할 수는 없지만 가변인수의 유연성이 필요할 때 선택할 수 있는 멋진 **패턴**이 있다.
  - ex) 95% 가 인수를 3개 이하로 사용하고 5%가 4개 이상의 호출인 경우

```
public void foo(){ ... }
public void foo(int a1){ ... }
public void foo(int a1, int a2){ ... }
public void foo(int a1, int a2, int a3){ ... }
public void foo(int a1, int a2, int a3, int... rest){ ... }
```

    - 단 5% 만이 배열을 생성한다. 대다수의 성능 최적화와 마찬가지로 이 기법도 보통 때는 별 이득이 없지만, 꼭 필요한 특수 상황에서 좋은 방법이 될 것이다.
    - `EnumSet` 의 정적 팩터리도 이 기법을 사용해 열거 타입 집합 생성 비용을 최소화한다.

- 정리
  - 인수 개수가 일정하지 않은 메서드를 정의해야 한다면 가변인수가 반드시 필요하다.
  - 메서드를 정의할 때 필수 매개변수는 가변인수 앞에 두고, 가변인수를 사용할 때는 성능 문제까지 고려하자.

▼ 54. null 이 아닌, 빈 컬렉션이나 배열을 반환하라

- 컬렉션이 비었으면 `null` 을 반환한다. - 따라하지 말 것

```
private final List<Cheese> cheeseInStock = ...;

/**
 * @return 매장 안의 모든 치즈 목록을 반환한다.
 *   단, 재고가 하나도 없다면 null 을 반환한다.
 */
public List<Cheese> getCheese() {
    return cheeseInStock.isEmpty() ? null : new ArrayList<>(cheeseInStock);
}
```

  - 이런 코드는 클라이언트에서 `null` 상황을 처리하는 코드를 추가로 작성해야한다.

```
List<Cheese> cheeses = shop.getCheeses();
if (cheese != null && cheeses.contains(Cheese.STILTON))
    System.out.println("GOOD");
```

- 컬렉션, 배열 같은 컨테이너가 비었을 때 `null` 을 반환하는 메서드를 사용할 때면, 이와 같은 방어코드를 추가해야한다.
    - `null` 을 반환하려면, `null` 을 받는 쪽에도 방어 처리 해줌으로 코드가 더 복잡해진다.
- 빈 컨테이너 생성에도 비용이드니 `null` 을 반환하는 게 낫다. 의 반론 2가지
    1. 성능 분석 결과 이 할당이 성능 저하의 주범이라고 확인되지 않는 한 (아이템 67), 이 정도의 성능 차이는 신경 쓸 수준이 못 된다.
    2. 빈 컬렉션과 배열은 굳이 새로 할당하지 않고 반환할 수 있다.

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

        - 가능성은 작지만, 사용 패턴에 따라 빈 컬렉션 할당이 성능을 눈에 띄게 떨어 트릴 수 있다. 해법으로는 **불변** 컬렉션을 반환하는 것이다. 불변 객체는 자유롭게 공유해도 안전하다 (아이템 17) `Collections.emptyList` , `Collections.emptySet` , `Collections.emptyMap` 메서드 등, 최적화에 해당하니 꼭 필요할 때만 사용하자.

```
public List<Cheese> getCheese() {
    return cheeseInStock.isEmpty() ? Collections.emptyList() :
      new ArrayList<>(cheeseInStock);
}
```

- 배열도 마찬가지다. 절대 null 을 반환하지 말고 길이가 0 인 배열을 반환하자

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

    - 이 방식 역시 성능 저하가 우려된다면, 길이 0짜리 배열을 미리 선언하고 그 배열을 반환하면 된다. 길이가 0인 배열은 모두 불변이기 때문이다.

```
private static final Cheese[] EMPTY_CHEESE_ARRAY = new Cheese[0];

public Cheese[] getCheese() {
    return cheesesInStock.toArray(EMPTY_CHEESE_ARRAY);
}
```

        - 단순히 성능 개선할 목적이라면 `toArray` 에 넘기는 배열을 미리 할당하는 건 추천하지 않는다. 오히려 성능이 떨어진다는 결과도 있다.

- 정리
  - `null` 이 아닌, 빈 배열, 빈 컬렉션을 반환하라
  - `null` 을 반환하는 API 는 사용하기 어렵고 오류 처리 코드도 늘어난다. 그렇다고 성능이 좋은 것도 아니다.

▼ 55. 옵셔널 반환은 신중히 하라

- 자바 8 이전 메서드에서 반환할 수 없을 때 취할 수 있는 선택지
  두 방법 모두 허점이 있다.

  1. 예외를 던진다.
     진짜 예외적인 상황에서만 사용해야 하며,
     예외 생성 시 스택 추적 전체를 캡쳐하므로 비용이 크다.
  2. 반환 타입이 객체일 경우 `null` 을 반환한다.
     예외 생성 비용은 들지 않는다.
     `null`을 반환할 수 있는 메서드를 호출하는 경우에 별도의 `null` 처리 코드를 추가해야한다. 그렇게 하지 않는 경우 `NullPointerException` 이 발생할 수 있다.

- 자바 8 이후 Optional API 등장
  - `Optional<T>` 는 `null` 이 아닌 `T` 타입 참조를 하나 담거나, 혹은 아무것도 담지 않을 수 있다.
  - `Optional` 은 원소를 최대 1개 가질 수 있는 **불변 컬렉션**이다.
  - 메서드에서 `T` 라는 타입을 반환해야 하지만, 특정 조건에서는 아무것도 반환하지 않아야 할 때 `T` 대신 `Optional<T>` 를 반환하도록 선언하면 된다.
  - `null` 을 반환하는게 아닌 `Optional<T>` 를 반환함으로, 예외를 던지는 비용을 줄일 수 있다.

- ex) 컬렉션에서 최대값을 뽑아주는 메서드

```
public static <E extends Comparable<E>> E max(Collection<E> c) {
    if (c.isEmpty())
        throw new IllegalArgumentException("빈 컬렉션");

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

    return result;
}
```

    - `IllegalArgumentException` 을 던진다.
- ex) 컬렉션에서 최대값을 구해 `Optional<E>` 를 반환한다.

```
public static <E extends Comparable<E>> Optional<E> max(Collection<E> 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);
}
```

    - 적절한 정적 팩터리를 사용해 옵셔널을 생성해주기만 하면 된다.
    - **옵셔널을 반환하는 메서드에서는 절대 `null` 을 반환하지 말자**. 옵셔널을 도입한 취지가 필요가 없어진다.
- ex) 스트림으로 변경한 버전

```
public static <E extends Comparable<E>> Optional<E> max(Collection<E> c) {
    return c.stream().max(Comparator.naturalOrder());
}
```

    - 스트림의 종단 연산 중 상당수가 옵셔널을 반환한다.
- 옵셔널 반환을 선택해야 하는 기준 (`null` 반환이나 예외를 던지는게 아닌)
    - **옵셔널은 검사 예외와 취지가 비슷하다.** (아이템 71)
      즉, 반환 값이 없을 수도 있음을 API 사용에게 명확히 알려준다.
    - 비검사 예외를 던지거나 `null` 을 반환한다면 API 사용자가 그 사실을 인지하지 못해 끔찍한 결과로 이어질 수 있다.
    - 검사 예외를 던지면 클라이언트에서는 반드시 이에 대처하는 코드를 작성해야한다.
- 메서드가 옵셔널을 반환한다면 값을 받지 못했을 때 취할 행동
    1. 기본값을 설정

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

    2. 원하는 예외를 던질 수 있다.

    ```
    Toy myToy = max(toys).orElseThrow(TemperTantrumException::new);
    ```

```

- 실제로 예외가 발생하지 않는 한 예외 생성 비용은 들지 않는다.

3. 항상 값이 채워져 있다고 가정한다.

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

- 다만, 값이 없다면 `NoSuchElementException` 을 던진다.

▼ 옵셔널 API 메서드

- Optional.of(T)
  - `Optional` 객체를 만듭니다. `null` 값이 들어가면 `NullPointException` 이 발생한다.
- Optional.ofNullable(T)
  - `null` 을 허용할 수 있는 옵셔널 메서드
- Optional.empty
  - 내부값이 비어있는 `Optional` 을 반환
- Optional.isPresent
  - `Optional` 내부에 값이 있으면 `true`, 없으면 `false`
- Optional.isEmpty
  - 값이 비어 있으면 `true` 를 반환, 아니면 `false`
- Optional.get
  - 안에 값을 가져온다
- Optional.ifPresent
  - `Consumer Functional Interface` 전달 후 로직 실행
- Optional.orElse
  - 값이 있는 경우 가져오고 없는 경우 선언한 내용을 반환.
- Optional.orElseGet
  - 값이 있으면 가져오고 없는 경우 `Supplier Functional Interface` 로직을 수행한다.
- Optional.orElseThrow
  - 값이 있으면 가져오고 없으면 에러를 던진다.
```

- Optional.filter
  - 내부 값을 걸러서 가져온다.
  - `Predicate Functional Interface` 를 수행 하여 조건에 부합하는 값을 가져온다.
- Optional.map
  - `Function Functional Interface`를 수행하여 내부 값을 순회하며 변경 후 반환한다.
- Optional.flatMap
  - `Optional` 안에 들어있는 인스턴스가 `Optional`인 경우 내부 원소값을 꺼낼 때 사용한다.
- 자바 9 Optional.stream
  - 옵셔널에 값이 있으면 그 값을 원소로 담은 스트림으로
    값이 없다면 빈 스트림으로 변환한다.

- ex) 옵셔널 API 메서드 사용 예제

```
// ProcessHandle 자바 9에서 소개된 클래스

Optional<ProcessHandle> parentProcess = ph.parent();
System.out.println("부모 PID: " + (parentProcess.isPresent() ?
    String.valueOf(parentProcess.get().pid()) : "N/A"));

// Optional map 사용
System.out.println("부모 PID: " +
    ph.parent().map(h -> String.valueOf(h.pid())).orElse("N/A");

// Stream, Optional 사용 예제

streamOfOptionals
    .filter(Optional::isPresent)  // 옵셔널에 값이 있다면
    .map(Optional::get)           // 그 값을 꺼내 스트림에 매핑한다.
```

- 반환 값으로 옵셔널을 사용하는 것이 답은 아니다.
  - 컬렉션, 스트림, 배열, 옵셔널 같은 컨테이너 타입은 옵셔널로 감싸면 안 된다.
  - 빈 `Optional<List<T>>` 를 반환하기보다는 빈 `List<T>` 를 반환하는게 좋다. (아이템 54)
  - 굳이 `Optional` 처리코드를 굳이 작성하지 않아도 된다.
- 메서드 반환 타입을 `T` 대신 `Optional<T>`로 선언해야 하는 경우
  - 결과가 없을 수 있으며, 클라이언트가 이 상황을 특별하게 처리해야 한다면 `Optional<T>`를 반환한다.

        - 이렇게 하더라도 `Optional<T>`를 반환하는 데는 대가가 따른다.
          `Optional`도 엄연히 새로 할당하고 초기화해야 하는 객체이고, 그 안에서 값을 꺼내려면 메서드를 호출해야 하니 한 단계를 더 거치는 셈이다.
        - 그래서 성능이 중요한 상황에서는 옵셔널이 맞지 않을 수 있다.
          어떤 메서드가 이 상황에 처하는지 알아내려면 세심히 측정해야 한다. (아이템 67)
    - 박싱된 기본 타입을 담는 옵셔널은 기본 타입 자체보다 무거울 수 밖에 없다.
        - 그래서 `OptionalInt`, `OptionalLong`, `OptionalDouble` 을 제공한다.
        - 이렇게 대체제가 있으니 **박싱된 기본 타입을 담은 옵셔널을 반환하는 일은 없도록 하자.**
    - **옵셔널을 컬렉션의 키, 값, 원소나 배열의 원소로 사용하는게 적절한 상황은 거의 없다.**
    - 정리
        - 값을 반환하지 못할 가능성이 있고, 호출할 때마다 반환값이 없을 가능성을 염두에 둬야 하는 메서드라면 옵셔널을 반환해야 할 상황일 수 있다.
        - 하지만 옵셔널 반환에는 성능 저하가 뒤따르니, 성능에 민감한 메서드라면 `null`을 반환하거나 예외를 던지는 편이 나을 수도 있다.
        - 옵셔널은 반환값 이외의 용도로 쓰는 경우는 매우 드물다.

▼ 56. 공개된 API 요소에는 항상 문서화 주석을 작성하라.

- **javadoc (이번 item 의 key)**
    - 소스코드 파일에서 문서화 주석 (자바독 주석) 이라는 특수한 형태로 기술된 설명을 추려 API 문서로 변환해준다.
    - 자바 버전이 올라가면서 `@interal`(5), `@code`(5), `@implSpec`(6), `@index`(9) 계속 발전하고 있다.
- **API 를 올바로 문서화하려면 공개된 모든 클래스, 인터페이스, 메서드, 필드 선언에 문서화 주석을 달아야 한다.**
    - 직렬화 할 수 있는 클래스라면 직렬화 형태(아이템 87)에 관해서도 작성해야 한다.
    - 문서화 주석이 없으면 javadoc도 그저 공개 API 요소를 선언만 나열해주는게 끝이다.
    - 기본 생성자에는 문서화 주석을 달 방법이 없으니, 공개 클래스는 절대 기본 생성자를 사용하면 안된다.
    - 유지보수까지 고려한다면, 대다수의 공개되지 않은 클래스, 인터페이스, 생성자, 메서드, 필드에도 문서화 주석을 달아야 할 것 이다.

- **메서드용 문서화 주석에는 해당 메서드와 클라이언트 사이의 규약을 명료하게 기술한다.**
    - 상속용으로 설계된 클래스(아이템 19)의 메서드가 아니라면 (그 메서드가 어떻게 동작하는지가 아니라) 무엇을 하는지 기술해야 한다.
      즉, how 가 아닌 what 을 기술해야 한다.
    - 문서화 주석에는 클라이언트가 해당 메서드를 호출하기 위한 **전제조건**을 모두 나열해야 한다. 또한 메서드가 성공적으로 수행된 후에 만족해야 하는 **사후조건**도 모두 나열해야 한다.
        - 전제조건은 `@throws` 태그로 비검사 예외를 선언하여 암시적으로 기술한다.
            - 비검사 예외 하나가 전제조건 하나와 연결되는 것이다.
        - 또한 `@param` 태그를 이용해 그 조건에 영향받는 매개변수에 기술할 수도 있다.
- 부작용도 문서화를 해야한다.
    - 부작용이란 사후조건으로 명확히 나타나지는 않지만, 시스템의 상태에 어떠한 변화를 가져오는 것을 뜻한다.
      예를 들어 백그라운드 스레드를 동작하는 메서드라면, 그 사실을 문서에 밝혀야 한다.
- **메서드의 계약**
    - 모든 매개변수에 `@param` 태그
    - 반환 타입이 `void` 아니라면 `@return` 태그
        - 태그 설명이 메서드 설명과 같을 때는 `@return` 태그를 생략해도 된다.
    - 발생할 가능성이 있는 (검사든 비검사든) 모든 예외에 `@throws` 태그 (아이템74)
        - 태그 설명이 if 로 시작해 해당 예외를 던지는 조건을 설명한다.
    - 관례상 `@param`, `@return` 태그는 명사구를 쓴다.
    - ex) `List` `get` 메서드

```
/**
 * Returns the element at the specified position in this list.
 *
 * @param index index of the element to return
 * @return the element at the specified position in this list
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= size()})
 */
E get(int index);
```

        - javadoc 은 문서화 주석을 HTML 로 변환하므로 문서화 주석 안의 HTML 요소들이 최종 HTML 문서에 반영된다.
        - `{@code}`
            - 태그로 감싼 내용을 코드용 폰트로 렌더링 한다.
            - 태그로 감싼 내용에 포함된 HTML 요소나 다른 javadoc 태그를 무시한다.
- 클래스를 상속용으로 설계 할 때
    - 자기사용 패턴에 대해서 문서화하여 해당 메서드를 올바로 재정의 하는 방법을 알려줘야 한다. (아이템 19 - 책 내용 (15)이 잘못 된 듯하다.)
    - 자기사용 패턴은 자바 8에 추가된 `@implSpec` 태그로 문서화한다.
    - `@implSpec` 주석은 해당 메서드와 하위 클래스 사이의 계약을 설명하여, 하위 클래스들이 그 메서드를 상속하거나 `super` 키워드를 이용해 호출할 때, 그 메서드가 어떻게 동작하는지를 명확히 인지하고 사용하도록 해줘야 한다.
    - ex) `List` `isEmpty` 메서드

```
/**
 * Returns {@code true} if this list contains no elements.
 *
 * @implSpec
 * This implementation return {@code this.size() == 0}.
 *
 * @return {@code true} if this list contains no elements
 */
public boolean isEmpty() {
    return size == 0;
}
```

        - Java 11까지 javadoc 명령줄에서 아래 옵션을 키지 않으면 해당 태그를 무시한다.
            - `-tag “implSpec:a:Implementation Requirements:”`
- API 설명에 HTML 메타 문자 (<, >, & 등) 포함하려면 `{@literal}` 태그로 감싼다.
    - 이 태그는 HTML 마크업이나 자바독 태그를 무시하게 해준다.
    - `{@code}` 태그와 비슷하지만 코드 폰트로 렌더링하지는 않는다.
    - ex) `* A geometric series converges if {@literal |r| < 1}.`
- **문서화 주석의 첫 번째 문장은 해당 요소의 요약 설명으로 간주된다.**
    - 요약 설명은 반드시 대상의 기능을 고유하게 기술해야 한다.

- 한 클래스(혹은 인터페이스) 안에서 요약 설명이 똑같은 멤버(혹은 생성자) 가 둘 이상 있으면 안 된다.
  - 다중 정의된 메서드가 있다면 특히 조심하자.
- 요약 설명에서는 마침표(.)에 주의해야한다.
  - 마침표가 요약 설명의 끝을 짓는 구분짓는 구분자이다.
  - 요약 설명이 끝나는 판단 기준은
    - {<마침표><공백><다음 문장 시작>} 이다.
      - 마침표: 구분자 .
      - 공백: 스페이스, 탭, 줄바꿈
      - 다음 문장 시작: 소문자가 아닌 문자
  - ex) Suspect 책의 예시

```
/**
 * A suspect, such as Colonel Mustard or {@literal Mrs. Peacock}.
 */
public class Suspect { ... }
```

- 메서드와 생성자의 요약 설명은 해당 메서드와 생성자의 동작을 설명하는 동사구
  - `ArrayList(int initialCapacity): Constructs an empty list with the speci-fied initial capacity.`
  - `Collection.size(): Returns the number of elements in this collection.`
- 클래스, 인터페이스, 필드의 요약설명은 대상을 설명하는 명사절
  - `Instant: An instantaneous poin on the time-line.`
  - `Math.PI: The double value that is closer than any other to pi, the ratio of the circumference of a circle its diameter`
- 색인 기능으로 `{@index}` 태그를 사용할 수 있다.

- 문서화 주석 작성시 주의 점
  - 제네릭 타입이나 제네릭 메서드를 문서화할 때는 모든 타입 매개변수에 주석을 달아야 한다.

```
/**
 * An object that maps keys to values. A map cannot contain
 * duplicate keys; each key can map to at most one value.
 *
 * (Remainder omitted)
```

```
 *
 * @param <K> the type of keys maintained by thiss map
 * @param <V> the type of mapped values
 */
public interface Map<K, V> { ... }
```

- 열거 타입을 문서화할 때는 상수들에게 주석을 달아야 한다.
  열거 타입 자체와 그 열거 타입의 public 메서드도 물론이다.

```
/**
 * An instrument section of a symphony orchestra.
 */
public enum OrchestraSection {
  /** Woodwinds, such as flute, clarinet, and oboe. */
  WOODWIND,

  /** Brass instruments, such as french horn and trumpet. */
  BRASS,

  /** Percussion instruments, such as timpani and cymbals. */
  PERCUSSION,

  /** Stringed instruments, such as violin and cello . */
  STRING;
}
```

- 애너테이션 타입을 문서화할 때는 멤버들에도 모두 주석을 달아야 한다.
  애너테이션 설명은 동사구로 한다.

```
/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exceptionto pass.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTest {
    /**
     * The exception that the annotated test method must throw
     * in order to pass. (The test is permitted to throw any
     * subtype of the type described by this class object.)
     */
    Class<? extends Throwable> value();
}
```

- 패키지를 설명하는 문서화 주석은 package-info.java 파일에 저장한다.
  (Java 9 부터 지원하는 모듈 시스템은 moule-info.java 파일에 작성하면 된다.)
  이 파일은 패키지 선언에 반드시 포함해야 하며 패키지 선언 관련 애너테이션을 추가로 포함할 수도 있다.

- API 문서화 시 자주 누락되는 설명

1. 스레드 안전성

   클래스 혹은 정적 메서드가 스레드 안전하든 그렇지 않든, 스레드 안전 수준을 받드시 API 설명에 포함해야 한다.(아이템 82)

2. 직렬화 가능성

   직렬화 가능성이 있는 클래스라면 직렬화 형태도 API 설명에 기술해야 한다 (아이템 87)

- javadoc 은 메서드 주석을 상속 시킬 수 있다.
  - 주석이 없는 API 요소를 발견하면 자바독이 가장 가까운 문서화 주석을 찾아준다. 이때 상위 클래스 보다 그 클래스가 구현한 인터페이스를 먼저 찾는다.
  - 또한 `{@inheritDoc}` 태그를 사용해 상위 타입의 문서화 주석 일부를 상속할 수 있다.
    클래스는 자신이 구현한 인터페이스의 문서화 주석을 재사용할 수 있다는 뜻이다.
    대신, 사용하기 까다롭고 제약도 조금 있다