8장 메서드

분류	서적 이펙티브 자바
시작일자	@2022년 5월 26일 오후 3:10
종료일자	@2022년 5월 28일 오후 1:59
참고	참고 블로그 [자바봄 기술 블로그] 참고 github.ido
스터디 링크	GitHub Link [이펙티브 자바 스터디]
서적	
Java	

▼ 목차

아이템 49. 매개변수가 유효한지 검사하라.

1. 오류는 가능한 빨리 잡아야 한다.

```
private List<String> names;

...

public String getNameOf(int index) {
  return names.get(index);
}
```

> 위와 같은 이름을 관리하고 있는 클래스가 있고 관리하고 있는 이름의 인덱스값을 통해 값을 조회하는 getNameOf가 있다고 가정한다.
> 이때 index가 음수이거나 names의 크기보다 크게될 경우 에러가 발생하게 된다.
>
> 이러한 제약은 반드시 문서화해야하며, 메서드 바디가 실행되기 전에 검증을 해야한다.

2. public과 protected 메서드는 문서화하라.

> 매개변수의 제약을 문서화하고, 이 제약을 어겼을 때 발생할 수 있는 예외를 기술하면, 해당 API를 사용하는 개발자가 해당 제약을 지킬 확률을 높일 수 있다.

```
/**
 * Returns a BigInteger whose value is {@code (this mod m}).  This method
 * differs from {@code remainder} in that it always returns a
 * <i>non-negative</i> BigInteger.
 *
 * @param  m the modulus.
 * @return {@code this mod m} // 현재 값 mod m
 * @throws ArithmeticException {@code m} &le; 0 // 에러 발생
 * @see    #remainder
 */
public BigInteger mod(BigInteger m) {
    if (m.signum <= 0)
        throw new ArithmeticException("BigInteger: modulus not positive");

    BigInteger result = this.remainder(m);
    return (result.signum >= 0 ? result : result.add(m));
}
```

여기에서 NPE에 대한 내용은 기술되어 있지 않지만, BigInteger 클래스 수준에서 기술했기 때문에
이 클래스의 public 메서드 전체에 적용되기에 각 메서드에 기술 할 필요가 없다.

❓ **그럼 private랑 package는?**

> 개발자가 스스로 메서드가 호출되는 상황을 통제할 수 있다. 즉 유요한 값이 들어온다는 것을 보장할 수 있기에 문서화 필수 대상에는 속하지 않는다.

3. java.util.Objects.requireNonNull

```
Objects.requireNonNull(username);
Objects.requireNonNull(useranme, "username 값은 null 일 수 없습니다");
this.username = Objects.requireNonNull(username);
```

- java 7부터 추가된 java.uitl.Objects.requireNonNull() 유연하고 사용하기도 편리하다.
- null 체크를 수동으로 하지 않아도 된다.

> 자바 9에는 `checkFromIndexSize`, `checkFromToIndex`, `checkIndex` 등 다양한 검사 기능이 추가되었다.

4. 비공개 메서드는 단언문을 통해 검증할 수 있다.

> assert 단언문은 선언한 조건이 무조건 참이여야 다음 로직을 수행할 수 있다.
> 그렇지 않으면 AssertionError을 던진다.

```
class AssertTest {
    private static int age = 10;

    @Test
    @DisplayName("assert 단언문 테스트")
    void assert단언문테스트() {
        assertThrows(AssertionError.class, () ->{
            assert age > 20;
        });
    }
}
```

> 좀 억지 테스트이긴하지만 에러가 잡히는지 안잡히는지 체크하기 위한 테스트이다.

public이나 protected 와는 다르게 노출되지 않는 비공개 메서드는 개발자가 메서드의 호출을 통제할 수 있는데,
유효값에 대한 제약을 단언문(assert)를 사용해 유효성을 검증할 수 있다.

1. 유효성 검사 실패시 AssertionError을 던진다.
2. 런타임에서 버그를 잡을수가 없다.

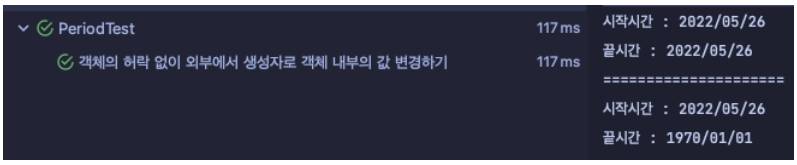
아이템 50. 적시에 방어적 복사본을 만들라.

> 어떤 객체든 그 객체의 허락 없이는 외부에서 내부를 수정하는 일은 불가능하다.
> 하지만 주의를 기울이지 않으면 자기도 모르게 내부를 수정하도록 하는 경우가 생기게 된다.

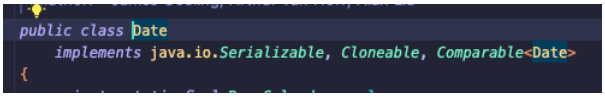
1. 수정 가능한 객체 (외부에서 생성자로 객체 내부의 값을 변경하는 경우)

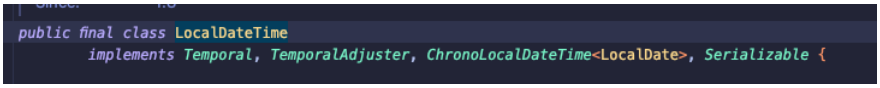
```
public class Period {
    private final Date start;
    private final Date end;

    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0) {
            throw new IllegalStateException(start + "가 " + end + " 보다 늦어서는 안됩니다. 다시 확인해주세요");
        }
        this.start = start;
        this.end = end;
    }

    public Date getStart() {
        return start;
    }

    public Date getEnd() {
        return end;
    }
}
```

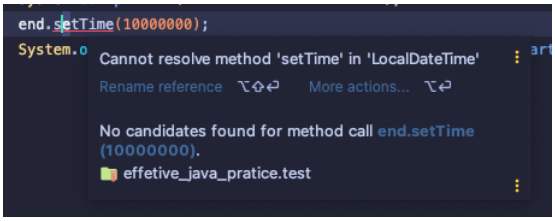
```
class PeriodTest {
    @Test
    @DisplayName("객체의 허락 없이 외부에서 생성자로 객체 내부의 값 변경하기")
    void 내부값변경() {
        SimpleDateFormat format = new SimpleDateFormat("yyyy/MM/dd");
        Date start = new Date();
        Date end = new Date();
        Period period = new Period(start, end);
        System.out.println("시작시간 : " + format.format(period.getStart()) + "\n끝시간 : " + format.format(period.getEnd()));
        System.out.println("======================");
        end.setTime(10000000);
        System.out.println("시작시간 : " + format.format(period.getStart()) + "\n끝시간 : " + format.format(period.getEnd()));
    }
}
```



- Date 클래스가 가변 클래스라는 이유로, 위의 불변식은 깨질 수 있다. 즉 불변식이 아니다.
- 날짜를 표현할 때 자바8 버전에 생긴 Instant 불변 클래스를 사용하면 해결할 수 있다.
- `LocalDateTime`, `ZoneDateTime`







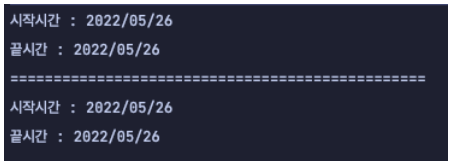
2. 객체의 허락없이 외부에서 setter로 객체 내부의 값을 변경하는 경우

```
class PeriodTest {

    @Test
    @DisplayName("객체의 허락 없이 외부에서 setter로 객체 내부의 값 변경하기")
    void 내부값변경() {
        SimpleDateFormat format = new SimpleDateFormat("yyyy/MM/dd");
        Date start = new Date();
        Date end = new Date();
        Period period = new Period(start, end);

        System.out.println("시작시간 : " + format.format(period.getStart()) + "\n끝시간 : " + format.format(period.getEnd()));
        System.out.println("================================================");
        period.getEnd().setYear(50);
        System.out.println("시작시간 : " + format.format(period.getStart()) + "\n끝시간 : " + format.format(period.getEnd()));
    }
}
```

3. 복사본을 생성하기 반환하기

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

    if (start.compareTo(end) > 0) { // 시작 시간이 끝 시간보다 최근인 경우는 IllegalStateException 오류 메세지 던지기
        throw new IllegalStateException(start + "가 " + end + " 보다 늦어서는 안됩니다. 다시 확인해주세요");
    }
}
```

> 매개변수로 받은 객체의 값을 꺼내어 새로운 객체를 생성(new Date)하여 복사본을 사용할 경우
> 외부에서 전달했던 기존 값의 레퍼런스를 토대로 위/변조를 수행하더라도 Period 객체의 내부 필드는 변하지 않는다.



1. **유효성 검사 로직이 아래로 내려온 이유**

> 멀티스레딩 환경에서는 원본 객체의 유효성을 검사 후 복사본을 만드는 찰나의 순간에 다른 스레드에서 원본 객체를 수헝할 위험이 있는데, 이를 검사시점/사용시점 공격 혹은 TOCTOU 공격이라고 한다.

2. **clone 메서드를 사용하지 않은 이유**

> Date 객체와 같이 final이 아닌 객체는 확장될 수 있기 때문에 실제 인수 Date가 아닌 악의적으로 재정의된 clone메서드를 가진 ExtendDate 객체가 전달될 수 있다.
> 이 경우 복제에 대한 책임을 공격자에게 위임하게 되어 문제가 생길 수 있다.
> → 즉 3자에 의해 확장될 수 있는 클래스는 방어적 복사본을 만들 때 clone을 사용해서는 안된다.

4. 접근자 메서드에서 방어적 복사본을 반환하라.

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

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

5. 많은 방어적 복사로 인한 성능 저하

- 객체를 생성할 때, 접근자를 통해 필드를 반환할 때마다 새롭게 객체를 생성하는 방어적 복사를 수행하면 성능 저하가 따라올 수 밖에 없다.
- 불변 객체를 조합해 객체를 구성함으로써 방어적 복사를 할 일 자체를 막는 것이 좋다.

> 🌟 **합의하에 방어적 복사를 생략할 수 있다.**
> 복사 비용이 너무 크거나, 클라이언트가 그 요소를 수정할 일이 없다는 것이 확실할 때 생략할 수 있다.

아이템 51. 메서드 시그니처를 신중히 설계하라.

❓ **메서드 시그니처란?**

> 메서드의 이름과 매개변수의 순서, 타입, 개수를 의미한다.
> 하지만 메서드의 리턴 타입과 예외처리하는 부분은 메서드 시그니처가 아니다!

1. 메서드 이름을 신중히 짓기

- 표준 명명 규칙를 따르자.
- 패키지에서 일관되게 이름을 짓자.
- 이해하기 쉬운 네이밍을 하자.
- 이름이 너무 길어지는 것은 피하자.

2. 편의 메서드는 너무 많이 만들지 말자.

❓ **편의 메서드란?**

> 말 그대로 편의 메서드이다. 예를 들어서 `Collections` 안에 있는 모든 메서드(swap, min, max 등)

- 메서드가 너무 많은 클래스는 사용, 문서화, 테스트, 유지보수에 어려움을 가진다. (인터페이스도 동일하다.)
- 클래스나 인터페이스는 자신의 각 기능을 완벽히 수행하는 메서드로 제공해야 한다.
- 자주 쓰일 경우에만 별도의 약칭 메서드를 두도록 하자. → **하지만 확신이 없으면 만들지 말자.**
- 매개변수 목록은 짧게 유지하자.
  - 4개 이하로 만드는게 좋다.
  - 같은 타입의 매개변수가 여러 개 연달아 나오는 경우를 주의하자.

3. 매개변수의 목록을 줄이는 방법.

1. **메서드를 분리한다.**

   > 하나의 메서드에서 책임이 과중하게 집중될 수록 매개 변수의 가짓수는 늘어날 수 밖에 없다.

   - 책의 예시인 List 인터페이스이다.
   - List의 지정 범위에서 주어진 원소의 인덱스를 찾아야 하는 경우 매개변수는 범위의 시작, 끝, 찾을 원소3개의 매개변수가 필요하다.
   - 하지만 List에서 제공하는 메서드를 별개로 이용하면 매개변수를 줄일 수 있다.

```
List<Integer> list = List.of(1,2,3,4,5,....N);

// 매개변수 줄이기 전
findElementAtSubList(int fromIndexOfSubList, int toIndexOfSubList, Object element);

// 매개변수 줄이기
List<E> subList(int fromIndex, int toIndex);
int indexOf(Object o);
```

주어진 요구사항을 자세히생각해보면 지정된 범위 내에서 원소를 찾는 것이 요구사항이다.

그럼 기능을 두 가지로 분리할 수 있을 것 같다.

1. **지정된 범위의 부분 리스트를 구하는 기능**
2. **주어진 원소를 찾는 기능**

위 두 기능에 공통점이 있는지 생각해보면 없다. 각각의 다른 행동을 하기에 분리하면 다른 곳에서도 쉽게 조합해서 사용할 수 있게 된다.

→ 이런 경우 직교성이 높다고 한다.

2. **매개변수 여러 개를 묶는 도우미 클래스를 만들기**

> 매개변수가 많다면 매개변수를 묶어줄 수 있는 클래스를 만들어서 하나의 객체로 전달할 수 있다.

예를 들면 DTO를 예시로 볼 수 있으며, 특정 도메인에 소속된 필드 정보들을 묶어서 전달하기 위해 사용할 수 있다.

3. **빌더 패턴 응용하기(Item 2)**

> 도우미 클래스에 빌더 패턴을 적용한 것이라고 생각하면 된다.

매개변수를 하나로 추상화한 객체를 정의하고, 클라이언트에서는 이 객체의 setter 메서드를 호출해 값을 설정한 뒤 execute 메서드를 호출하여 매개변수의 유효성 검사 및 설정이 완료된 객체를 넘겨 계산을 수행하는 것

4. 매개변수 타입으로는 클래스보다 인터페이스가 낫다.

> 매개변수 타입은 되도록 인터페이스를 사용하면 좋다.

Map만 하더라도 HashMap, TreeMap 등이 있는데 매개변수 타입으로 특정 Map구현체를 선언하면 해당 구현체로 제한되지만 Map을 매개변수 타입으로 선언한다면 위 어떤 Map의 구현체라도 인수로 전달될 수 있다.

즉 구현체를 타입으로 지정하게 된다면 OCP, DIP원칙에도 어긋나게 된다.

5. Boolean 보다 원소 2개짜리 열거 타입이 낫다.

> 열거타입(enum)을 사용하면 코드를 읽고, 쓰기가 더 쉬워진다.

```
public enum TemperatureScale { FAHRENHEIT, CELSIUS }

// 위 열거타입을 받아 적합한 온도계 인스턴스를 만들어 줄 때

// 열거타입이 아닌 경우 즉 Boolean
TemperatureScale.newInstance(true)
// 열거타입의 경우
TemperatureScale.newInstance(TemperatureScale.CELSIUS)
```

위 처럼 열거 타입이 아닌경우 true면 섭씨라고 할 때 생성코드만 보고는 무엇인지 알수가 없다.

아이템 52. 다중정의는 신중히 사용하라.

```
public class Main {
    public static class A {}
    public static class B extends A {}
    public static class C extends A {}

    public static class Classifier {
        public static String str(A a){ return "A"; } //---(1)
        public static String str(B b){ return "B"; } //---(2)
        public static String str(C c){ return "C"; } //---(3)
    }


    public static void main(String[] args){
        A[] list = {new A(), new B(), new C()};
        Stream.of(list)
                .map(Classifier::str)
                .forEach(System.out::println);
    }
}
```

출력 결과를 한 번 예상해보자면 `A B C`가 나올 것이라고 기대할 수 있다 하지만 실제 출력은 아래와 같다.

```
A
A
A

Process finished with exit code 0
```

❓ **나는 분명 B객체와 C객체를 매개변수로 호출 했는데 왜 A A A가 호출된걸까?**

> 어느 메서드를 호출할지 선택되는 시점이 컴파일타임이기 때문이다.

```
public class Main {
    public static class A {
        public String str(){
            return "A";
        }
    }
    public static class B extends A {
        @Override
        public String str() {
            return "B";
        }
    }
    public static class C extends A {
        @Override
        public String str() {
            return "C";
        }
    }

    public static void main(String[] args){
        A[] list = {new A(), new B(), new C()};
        Stream.of(list)
                .map(A::str)
                .forEach(System.out::println);
    }
}
```

```
A
B
C

Process finished with exit code 0
```

🌟 **위 처럼 다중정의와 재정의의 결정적인 차이점은 호출되는 메서드가 선택되는 시점이다.**
다중 정의 메서드의 선택시점은 정적으로 컴파일타임에 결정되고, 재정의 메서드의 선택시점은 런타임시점의 가장 하위 재정의 메서드로 선택된다.

1. 매개변수가 개수가 같은 오버로딩을 피할 수 없다면?

> 매개변수 중 하나 이상이 근본적으로 다르면 그나마 괜찮다.

`ArrayList` 의 인자가 1개 이상인 생성자는 `ArrayList(int A)` 와 `ArrayList(Collection<? extends E> c` 가 있지만 `int` 와 `collection`은 근본적으로 다르므로 괜찮다.

❓ **근본적으로 다르면 다 해결이 되는건가?**

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
list.remove(3); // 3을 지우라는 것일까? 3번째 원소를 지우라는 것일까?
```

- remove(int index)
- remove(Object o)

위 처럼 `Lise<E>` 인터페이스는 위 두 개를 오버로딩 했기 때문이다.

제네릭과 오토박싱이 등장하면서 int를 Integer로 자동 현봔 해주니까 근본적으로 달라지지 않게 됐다.

2. 오버로딩할때 서로 다른 함수형 인터페이스라도 같은 위치의 인수로 받아서는 안된다.

```
new Thread(System.out::println).start(); // 컴파일 성공

ExecutorService exec = Executors.newCachedThreadPool();
exec.submit(System.out::println); // 컴파일 에러
```

- 넘겨지는 인수는 모두 똑같고, 둘 다 Runnable을 받는 형제 메서드를 오버로딩 하고 있다.
- 하지만 아래만 컴파일 에러가 뜨게 된다.

→ println 참조가 모호합니다. println() 및 println(boolean)이 모두 일치합니다.

> 이 원인은 println()메서드와 submit() 메서드가 모두 다중정의가 되어있기 때문이다.

아이템 53. 가변인수는 신중히 사용하라.

❓ 가변인수란?

> 매개변수를 동적으로 받을 수 있는 방법이다.

```
// ...을 사용하여 가변인수를 설정할 수 있고 0개 이상의 인수를 받을 수 있다.
static int sum(int... numbers) {
  int sum = 0;
  for (int number : numers) {
    sum += number;
  }

  return sum;
}

sum(1, 2);
sum(1);
sum();
```

- 가변인수 메서드를 호출하면 먼저 인수의 갯수와 같은 배열을 만들고 인수들을 배열에 저장해 가변인수 메서드에 전달한다.
- 최소한 인수가 한개라도 필요하게 하려면 아래과 같이 `validation` 로직을 작성해주면 된다.

```
static int sum(int... numbers) {
  if (numbers.lenght == 0) {
    throw new IllegalArgumentException("인수가 1개 이상 필요합니다.");
  }
  int sum = 0;
  for (int number : numers) {
    sum += number;
  }

  return sum;
}

sum(1, 2);
sum(1);
sum(); // 에러 발생
```

문제점

> 위와 같이 코딩을하게 되면 컴파일 단계가 아니라 런타임시 에러가 발생한다.

그러므로 아래와 같이 해결할 수 있다.

```
static int sum(int firstNumber, int... numbers) {
  int sum = firstNumber;
  for (int number : numers) {
    sum += number;
  }

  return sum;
}

sum(1, 2);
sum(1);
sum(); // 컴파일 시 에러
```

`firstNumber` 매개변수를 추가해서 무조건 1개 이상은 값이 넘어오게 하는 것이다.

- 성능에 민감한 사항에서는 가변인수가 걸림돌이 될 수도 있다.
- 호출될때마다 배열을 새로 하나 할당하고 초기화 하기 때문
- 최소한으로 하기 위해서는 여러개의 메서드를 오버로딩 하는 전략으로 사용하면 된다.

```
static int sum(int number1)
static int sum(int number1, int number2)
static int sum(int number1, int nubmer2, int number3)
static int sum(int number1, int nubmer2, int number3, int... numbers)
```

> 하지만 코드가 너무 더러워져서 개인적으로는 불호이다.

```
public static <E extends Enum<E>> EnumSet<E> of(E e) {
    EnumSet<E> result = noneOf(e.getDeclaringClass());
    result.add(e);
    return result;
}


public static <E extends Enum<E>> EnumSet<E> of(E e1, E e2) {
    EnumSet<E> result = noneOf(e1.getDeclaringClass());
    result.add(e1);
    result.add(e2);
    return result;
}


public static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3) {
    EnumSet<E> result = noneOf(e1.getDeclaringClass());
    result.add(e1);
    result.add(e2);
    result.add(e3);
    return result;
}

public static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4) {
    EnumSet<E> result = noneOf(e1.getDeclaringClass());
    result.add(e1);
    result.add(e2);
    result.add(e3);
    result.add(e4);
    return result;
}

public static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4,
                                                E e5)
{
    EnumSet<E> result = noneOf(e1.getDeclaringClass());
    result.add(e1);
    result.add(e2);
    result.add(e3);
    result.add(e4);
    result.add(e5);
    return result;
}
```

아이템 54. null이 아닌 빈 컬렉션이나 배열을 반환하라.

null 반환하는 경우

> 컬렉션이 비었을 때 null을 반환하는 메서드

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

빈 컬렌셕을 반환

> null 대신에 아래와 같이 빈 컬렉션을 반환할 수 있다.

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

❓ **왜 null보다 빈 컬렉션을 반환하는게 좋을까?**

- null대신 빈 컬렉션을 반환하더라도 성능 차이가 크지 않다.
- 빈 컬렉션은 굳이 새로 할당하지 않고도 반환할 수 있다.
- 빈 컬렉션을 새로 할당하지 않고 반환하는 방법은 빈 불변 컬렉션 반환을 통해 가능하다

빈 불변 컬렉션 반환

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

> Collections를 이용하면 빈 불변 컬렉션을 반환할 수 있다.

불변이기 때문에 공유로부터 안전하고 매번 새로 할당하지 않아도 된다.

배열의 경우

> null이 아닌 빈 배열을 반환하자.

```
//default case
public List<Cheese> getCheeses() {
  return values.toArray(new Cheese[0]);
}
```

아이템 55. 옵셔널 반환은 신중히 하라.

Java 8 이전 값을 반환할 수 없을 때 두 가지 선택지

> Java 8 이전 메서드에서 반환할 수 있는 값이 없는 경우 할 수 있는 선택지는 두 가지가 있었다.

- **예외를 던진다.**
  - 진짜 예외적인 상황에서만 사용해야 하며, 예외 생성시점에서 stracktrace 전체를 캡쳐하기에 비용이 크다.
- **반환 타입이 객체일 경우 null을 반환한다.**
  - 예외 생성에서 비용은 들지 않는다.
  - null을 반환할 수 있는 메서드를 호출하는 클라이언트는 null check 코드를 항상 추가해야 하며, 그렇게 하지 않을 경우 예기치 못한 상황에서 NPE가 발생할 수 있다.

Java 8 이후의 선택지 - Optional

> 해당 객체는 제네릭 타입 매개변수로 선언한 객체를 참조하거나 혹은 비어있는 상태로 존재할 수 있다.
> 즉 Optional은 하나의 값을 가지는 불변 컬렉션이다.

- 보통은 T를 반환하지만, 특정 조건에서는 아무것도 반환하지 않아야 할 때 T 대신 `Optional<T>` 를 반환하도록 선언하면 된다.
- 옵셔널을 반환하는 메서드는 예외를 던지는 메서드보다 유연하고 사용하기 쉬우며, null을 반환하는 메서드보다 오류 가능성이 적다.

```
// java 8 이전
public static <E extends Comparable<E>> E max(Collection<E> c) {
  if(c.isEmpty()){
    throw new IllegalArgumentException();
  }
  E result = null;
  for (E e  C) {
    if(result == null  || e.compareTo(result) > 0){
      result = Objects.requireNonNull(e);
    }
  }

  return result;
}

// java 8 이후
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);
}
```

Optional API를 사용해야하는 경우

- **반환값이 없을 수도 있음을 클라이언트(API 사용자) 에게 명확히 알려줘야 하는 경우**

```
//case 1. 기본값 설정
String lastWordInLexicon = max(words).orElse("단어 없음...");

//case 2. 예외 던지기
Toy myToy = max(toys).orElseThrow(TemperTantrumException::new);
```

- **결과가 없으며 클라이언트에게 이 상황을 특별하게 처리해야 하는 경우**

  DB에서 데이터를 조회했는데 없는 경우 반드시 특정 행동을 해야하거나, 특정 예외를 던지는등의 의미있는 비즈니스 로직이 동작되야 하는 경우 Optional을 반환하여 orElse, orElseThrow, orElseGet 등의 메서드 체이닝 동작을 할 수 있다.

주의사항

- **반환 값으로 Optional을 사용하는 것이 항상 정답은 아니다.**

  `Collecttion` `Stream` `Array` `Optional` 같은 컨테이너 타입을 Optional로 감싸서는 안된다.

  즉 List<T>로 반환하라는 말이다.

- **박싱된 기본 타입을 담은 Optional을 반환하지 않도록 하자.**

  기본적으로 박싱된 타입을 담는 Optional API는 Primitive Type 값에 대해서도 박싱을 해서 담기 때문에 더 무거울 수 밖에 없다.

아이템 56. 공개된 API요소에는 항상 문서화 주석을 작성하라.

> 코드가 변경되면 문서도 매번 함께 수정해줘야 한다.
> javadoc이라는 유틸리티는 이 귀찮은 작업을 도와준다.

- how가 아닌 what 즉 어떻게 동작이 아니라 무엇을 하는지가 중요하다.
- 클라이언트가 해당 메서드를 호출하기 위한 전제조건과 사후조건을 모두 나열해야 한다.
- 전제조건은 `@throw` 로 비검사 예외를 선언한다. 비검사 예외 하나당 전제조건 하나와 연결된다.
- `@param` 으로 그 전제조건에 영향받는 매개변수에 기술한다.

```
/**
 * Returns the element at the specified position in this list.
 *
 * <p>This method is <i>not</i> guaranteed to run in constant
 * time. In some implementations it may run in time proportional
 * to the element position.
 *
 * @param  index index of element to return; must be
 *         non-negative and less than the size of this list
 * @return the element at the specified position in this list
 * @throws IndexOutOfBoundsException if the index is out of range
 *         ({@code index < 0 || index >= this.size()})
 */
E get(int index) {
    return null;
}
```

- HTML 태그를 사용할 수 있다
  - `<p>` , `<i>` 등등.. `<table>` 도 가능
- `@param`
  - 전제조건에 영향받는 매개변수에 붙인다
  - 모든 매개변수에 붙이는게 좋다
  - 관례상 명사구를 쓰고, 마침표를 붙이지 않는다
- `@return`
  - 반환타입이 void가 아니라면 붙인다
  - void가 아니라도 메서드의 설명과 일치할 경우엔 생략해도 된다
  - 관례상 명사구를 쓰고, 마침표를 붙이지 않는다
- `@throws`
  - 비검사 예외를 선언.

    - 비검사 예외 하나당 전제조건 하나를 연결한다.
    - 관례상 마침표를 붙이지 않는다
- `{@code ...코드... }`
    - 태그로 감싼 내용을 코드용 폰트로 렌더링한다
    - 태그로 감싼 내용에 포함된 HTML요소나 다른 자바독 태그를 무시한다
    - `<pre>{@code ...코드... }</pre>` 와 같이 사용하면 여러줄로 된 코드도 작성 가능하다. 단 @을 쓸땐 탈출문자 붙여야 함
- `@implSpec`
    - 해당 메서드와 하위 클래스 사이의 계약을 설명한다
    - 하위 클래스들이 그 메서드를 상속하거나 super키워드를 이용해 호출할 때 그 메서드가 어떻게 동작하는지를 명확히 인지하고 사용하도록 해야 한다
- `{@literal ... }`
    - <, >, &등의 HTML메타문자를 포함시킨다.
    - `{@code ...코드... }` 와 비슷하지만 코드 폰트로 렌더링하진 않는다
- 첫 문장은 주로 요약 설명이다
    - 한 클래스 혹은 한 인터페이스 안에 요약설명이 중복되면 안된다. (특히 오버로딩된 메서드들에서 특히 조심하자)
    - 마침표에 주의해야 한다.

```
/**
 * A suspect, such as Colonel Mustard or Mrs. Peacock.
*/
```

`A suspect, such as Colonel Mustard or Mrs` 까지 요약설명으로 간주된다.

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

`@literal` 로 감싸줌으로써 해결한다. 자바10부터는 요약 설명 전용 태그 `@summary`가 추가되었다.
이를 활용하면 더 깔끔하게 나타낼 수 있다.

```
/**
 * {@summary A suspect, such as Colonel Mustard or Mrs. Peacock.}
*/
```

- 메서드와 생성자의 요약 설명
    - 주어가 없는 동사구여야 한다. 2인칭 문장(Return)이 아닌 3인칭 문장(Returns)을 사용해야 한다

```
/**
 * Constructs an empty list with the specified initial capacity.
 * ...
 */
public ArrayList(int initialCapacity) { ... }
```

```
/**
 * Returns the number of elements in this collection.
 * ...
 */
int size();
```

- 클래스, 인터페이스, 필드의 요약 설명
    - 명사절이어야 한다.

```
/**
 * The {@code double} value that is closer than any other to
 * <i>pi</i>, the ratio of the circumference of a circle to its
 * diameter.
 */
public static final double PI = 3.14159265358979323846;
```

- `{@index ... }`

- 중요한 용어를 추가로 색인화할 수 있다.

```
/**
 * This method complies with the {@index IEEE 754} standard.
 */
public void function() {
}
```

- 제네릭 문서화
  - 모든 타입 매개변수에 주석을 달아야 한다.

```
/**
 * An object that maps keys to values.  A map cannot contain duplicate keys;
 * ...
 * @param <K> the type of keys maintained by this map
 * @param <V> the type of mapped values
 */
public interface Map<K,V> { ... }
```

- 열거 타입 문서화
  - 상수들, 열거 타입 자체, public메서드에도 주석을 달아야 한다

```
/**
 * 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 타입 문서화
  - 타입 자체, 멤버들에도 모두 주석을 달아야 한다.
  - 필드 설명은 명사구로 한다
  - 요약 설명은 동사구로 한다

```
/**
 * Indicates that the annotated method is a test method that
 * must throw the designated exception to 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();
}
```

📕 7장 람다와 스트림