[item 49] 매개변수가 유효한지 검사하라

요약

- 메서드나 생성자를 작성할 때 매개변수로 넘어오는 값들에 어떤 제약이 있을지 생각해야 한다
- 제약이 있다면, 그 제약들을 문서화 해야 하고 (@throws)
- 메서드 시작 부분에서 명시적으로 검사해야 한다

```
1. 조건 체크 후 예외를 던진다
ex. IllegalArgumentException, IndexOutOfBoundsException, NullPointerException

2. null 체크를 하는 경우라면
java.util.Objects.requireNonNull()

3. index 검사를 하는 경우라면
checkFromIndexSize(), checkFromToIndex(), checkIndex()

4. assert()
```

- 단 "매개변수에 제약을 두는게 좋다"고 해석하면 안된다. 오히려 그 반대다

  메서드는 최대한 범용적으로 설계해야 하므로 제대로 된 일을 할 수만 있다면 매개변수 제약은 적을수록 좋다

본문

메서드나 생성자로 넘어온 매개변수가 가질 수 있는 제약조건의 예

- 인덱스 값은 음수면 안된다
- 객체 참조는 null 이 아니어야 한다

매개변수의 제약조건이 있다면 (1) **문서화**나 (2) **메서드 시작 부분에서 검사**해야 한다

이는 `오류는 가능한 한 빨리 잡아야 한다` 는 원칙의 한 사례인데

오류를 발생 즉시 잡지 못하면 해당 오류는 감지하기 어려워지고, 오류 발생 지점을 찾기 어려워진다.

문서화 하는 법

메서드나 생성자에 넘어온 매개변수가 제약조건을 어길 시 예외를 던져 처리할 수 있다.

만약 `public`, `protected` 메서드가 예외를 던진다면 이를 문서화해야 한다

주석에서 @throws 를 사용하면 된다 - 자바독 태그

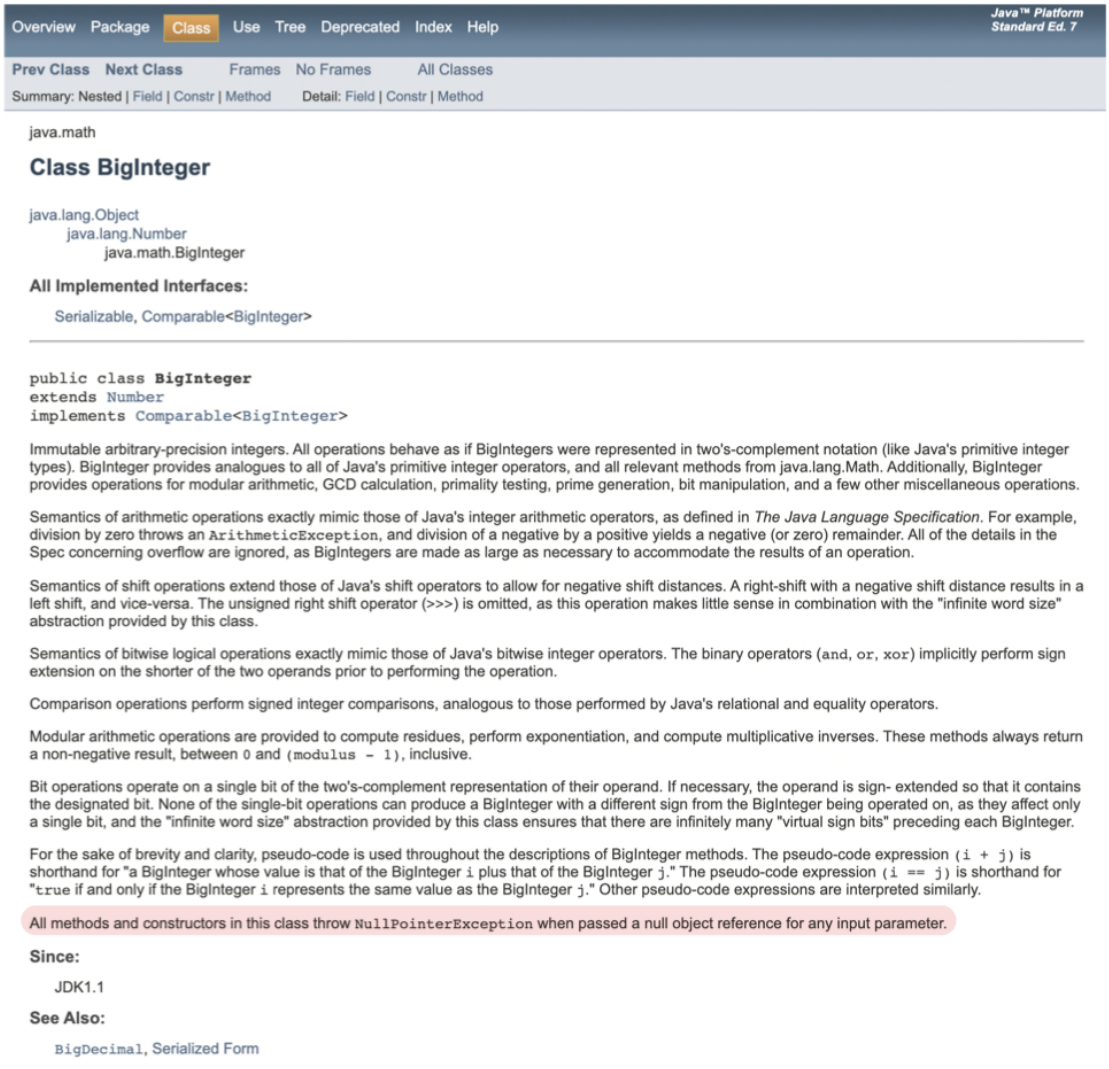
모든 메서드에 적용되는 사항이 있다면 이는 클래스 수준 주석에 적는것이 훨씬 깔끔하다

📎 java.math.BigInteger

```
/**
 * Immutable arbitrary-precision integers.  All operations behave as if
 * BigIntegers were represented in two's-complement notation (like Java's
 * primitive integer types).  BigInteger provides analogues to all of Java's
 * primitive integer operators, and all relevant methods from java.lang.Math.
 * Additionally, BigInteger provides operations for modular arithmetic, GCD
 * calculation, primality testing, prime generation, bit manipulation,
 * and a few other miscellaneous operations.
 *
 * <p>Semantics of arithmetic operations exactly mimic those of Java's integer
 * arithmetic operators, as defined in <i>The Java&trade; Language Specification</i>.
 * For example, division by zero throws an {@code ArithmeticException}, and
 * division of a negative by a positive yields a negative (or zero) remainder.
 *
 * <p>Semantics of shift operations extend those of Java's shift operators
 * to allow for negative shift distances.  A right-shift with a negative
 * shift distance results in a left shift, and vice-versa.  The unsigned
 * right shift operator ({@code >>>}) is omitted since this operation
 * only makes sense for a fixed sized word and not for a
 * representation conceptually having an infinite number of leading
 * virtual sign bits.
```

```
 *
 * <p>Semantics of bitwise logical operations exactly mimic those of Java's
 * bitwise integer operators.  The binary operators ({@code and},
 * {@code or}, {@code xor}) implicitly perform sign extension on the shorter
 * of the two operands prior to performing the operation.
 *
 * <p>Comparison operations perform signed integer comparisons, analogous to
 * those performed by Java's relational and equality operators.
 *
 * <p>Modular arithmetic operations are provided to compute residues, perform
 * exponentiation, and compute multiplicative inverses.  These methods always
 * return a non-negative result, between {@code 0} and {@code (modulus - 1)},
 * inclusive.
 *
 * <p>Bit operations operate on a single bit of the two's-complement
 * representation of their operand.  If necessary, the operand is sign-
 * extended so that it contains the designated bit.  None of the single-bit
 * operations can produce a BigInteger with a different sign from the
 * BigInteger being operated on, as they affect only a single bit, and the
 * arbitrarily large abstraction provided by this class ensures that conceptually
 * there are infinitely many "virtual sign bits" preceding each BigInteger.
 *
 * <p>For the sake of brevity and clarity, pseudo-code is used throughout the
 * descriptions of BigInteger methods.  The pseudo-code expression
 * {@code (i + j)} is shorthand for "a BigInteger whose value is
 * that of the BigInteger {@code i} plus that of the BigInteger {@code j}."
 * The pseudo-code expression {@code (i == j)} is shorthand for
 * "{@code true} if and only if the BigInteger {@code i} represents the same
 * value as the BigInteger {@code j}."  Other pseudo-code expressions are
 * interpreted similarly.
 *
 * <p>All methods and constructors in this class throw
 * {@code NullPointerException} when passed
 * a null object reference for any input parameter.
 *
 * BigInteger must support values in the range
 * -2<sup>{@code Integer.MAX_VALUE}</sup> (exclusive) to
 * +2<sup>{@code Integer.MAX_VALUE}</sup> (exclusive)
 * and may support values outside of that range.
 *
 * An {@code ArithmeticException} is thrown when a BigInteger
 * constructor or method would generate a value outside of the
 * supported range.
 *
 * The range of probable prime values is limited and may be less than
 * the full supported positive range of {@code BigInteger}.
 * The range must be at least 1 to 2<sup>500000000</sup>.
 *
 * @implNote
 * In the reference implementation, BigInteger constructors and
 * operations throw {@code ArithmeticException} when the result is out
 * of the supported range of
 * -2<sup>{@code Integer.MAX_VALUE}</sup> (exclusive) to
 * +2<sup>{@code Integer.MAX_VALUE}</sup> (exclusive).
 *
 * @see     BigDecimal
 * @jls     4.2.2 Integer Operations
 * @author  Josh Bloch
```

```
 * @author  Michael McCloskey
 * @author  Alan Eliasen
 * @author  Timothy Buktu
 * @since 1.1
 */

public class BigInteger extends Number implements Comparable<BigInteger> {

    /**
     * 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}
     * @throws ArithmeticException {@code m} &le; 0
     * @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));
    }

    // ...

}
```



Overview Package Class Use Tree Deprecated Index Help — Java™ Platform Standard Ed. 7

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.math

Class BigInteger

java.lang.Object
  java.lang.Number
    java.math.BigInteger

All Implemented Interfaces:

Serializable, Comparable<BigInteger>

```
public class BigInteger
extends Number
implements Comparable<BigInteger>
```

Immutable arbitrary-precision integers. All operations behave as if BigIntegers were represented in two's-complement notation (like Java's primitive integer types). BigInteger provides analogues to all of Java's primitive integer operators, and all relevant methods from java.lang.Math. Additionally, BigInteger provides operations for modular arithmetic, GCD calculation, primality testing, prime generation, bit manipulation, and a few other miscellaneous operations.

Semantics of arithmetic operations exactly mimic those of Java's integer arithmetic operators, as defined in *The Java Language Specification*. For example, division by zero throws an `ArithmeticException`, and division of a negative by a positive yields a negative (or zero) remainder. All of the details in the Spec concerning overflow are ignored, as BigIntegers are made as large as necessary to accommodate the results of an operation.

Semantics of shift operations extend those of Java's shift operators to allow for negative shift distances. A right-shift with a negative shift distance results in a left shift, and vice-versa. The unsigned right shift operator (>>>) is omitted, as this operation makes little sense in combination with the "infinite word size" abstraction provided by this class.

Semantics of bitwise logical operations exactly mimic those of Java's bitwise integer operators. The binary operators (`and`, `or`, `xor`) implicitly perform sign extension on the shorter of the two operands prior to performing the operation.

Comparison operations perform signed integer comparisons, analogous to those performed by Java's relational and equality operators.

Modular arithmetic operations are provided to compute residues, perform exponentiation, and compute multiplicative inverses. These methods always return a non-negative result, between `0` and `(modulus - 1)`, inclusive.

Bit operations operate on a single bit of the two's-complement representation of their operand. If necessary, the operand is sign- extended so that it contains the designated bit. None of the single-bit operations can produce a BigInteger with a different sign from the BigInteger being operated on, as they affect only a single bit, and the "infinite word size" abstraction provided by this class ensures that there are infinitely many "virtual sign bits" preceding each BigInteger.

For the sake of brevity and clarity, pseudo-code is used throughout the descriptions of BigInteger methods. The pseudo-code expression `(i + j)` is shorthand for "a BigInteger whose value is that of the BigInteger `i` plus that of the BigInteger `j`." The pseudo-code expression `(i == j)` is shorthand for "`true` if and only if the BigInteger `i` represents the same value as the BigInteger `j`." Other pseudo-code expressions are interpreted similarly.

All methods and constructors in this class throw `NullPointerException` when passed a null object reference for any input parameter.

Since:

JDK1.1

See Also:

`BigDecimal`, Serialized Form

[클래스 수준 문서화] 모든 메서드와 생성자에서 input parameter 가 null 이면 NullPointerException 이 발생한다

▼ BigInteger ?

https://coding-factory.tistory.com/604

💡 int는 메모리 크기는 4byte로 표현할 수 있는 범위는 -2,147,483,648 ~ 2,147,483,647이고 long은 메모리 크기는 8byte로 표현할 수 있는 범위는 -9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807입니다. 그 범위를 넘어서게 되면 모두 0으로 출력이 됩니다. 숫자의 범위가 저 범위를 넘을 경우는 잘 없겠지만 프로그램 개발 특히 돈과 관련된 개발이나 알고리즘 문제를 풀 때 항상 최악의 상황을 고려해야 하므로 무한의 정수가 들어갈 수 있는 가능성이 있다면 BigInteger이라는 클래스를 활용하는 것이 좋습니다. BigInteger은 문자열 형태로 이루어져 있어 숫자의 범위가 무한하기에 어떠한 숫자이든지 담을 수 있습니다.

```
BigInteger bigNumber = new BigInteger("10000");
```

사칙연산은 BigIntger 의 메서드를 이용해야 한다고 합니다

Null check - Objects.requireNonNull()

```
public final class Objects {

    /**
     * Checks that the specified object reference is not {@code null}. This
     * method is designed primarily for doing parameter validation in methods
     * and constructors, as demonstrated below:
     * <blockquote><pre>
     * public Foo(Bar bar) {
     *     this.bar = Objects.requireNonNull(bar);
     * }
     * </pre></blockquote>
     *
     * @param obj the object reference to check for nullity
     * @param <T> the type of the reference
     * @return {@code obj} if not {@code null}
     * @throws NullPointerException if {@code obj} is {@code null}
     */
    public static <T> T requireNonNull(T obj) {
        if (obj == null)
            throw new NullPointerException();
        return obj;
    }

    /**
     * Checks that the specified object reference is not {@code null} and
     * throws a customized {@link NullPointerException} if it is. This method
     * is designed primarily for doing parameter validation in methods and
     * constructors with multiple parameters, as demonstrated below:
```

```
     * <blockquote><pre>
     * public Foo(Bar bar, Baz baz) {
     *     this.bar = Objects.requireNonNull(bar, "bar must not be null");
     *     this.baz = Objects.requireNonNull(baz, "baz must not be null");
     * }
     * </pre></blockquote>
     *
     * @param obj     the object reference to check for nullity
     * @param message detail message to be used in the event that a {@code
     *                NullPointerException} is thrown
     * @param <T> the type of the reference
     * @return {@code obj} if not {@code null}
     * @throws NullPointerException if {@code obj} is {@code null}
     */
    public static <T> T requireNonNull(T obj, String message) {
        if (obj == null)
            throw new NullPointerException(message);
        return obj;
    }

}
```

```
// 1
Objects.requireNonNull(username);

// 2
Objects.requireNonNull(username, "username 값은 null 일 수 없습니다");

// 3
this.username = Objects.requireNonNull(username);
```

- java 7 부터 추가된 java.util.Objects.requireNonNull() 은 유연하고 사용하기도 편하다
- null 체크를 수동으로 하지 않아도 된다.

1. null check 하려는 object 를 매개변수로 사용
2. NullPointerException 의 메세지를 넣어줄 수도 있고
3. 입력을 그대로 반환하므로, 반환값을 사용할 수도 있다.

Index check - Objects.checkIndex(), Objects.checkFromToIndex(), Objects.checkFromIndexSize()

📎 Objects.checkIndex()

```
/**
 * Checks if the {@code index} is within the bounds of the range from
 * {@code 0} (inclusive) to {@code length} (exclusive).
 *
 * <p>The {@code index} is defined to be out of bounds if any of the
 * following inequalities is true:
 * <ul>
 *  <li>{@code index < 0}</li>
 *  <li>{@code index >= length}</li>
 *  <li>{@code length < 0}, which is implied from the former inequalities</li>
 * </ul>
 *
 * @param index the index
 * @param length the upper-bound (exclusive) of the range
 * @return {@code index} if it is within bounds of the range
 * @throws IndexOutOfBoundsException if the {@code index} is out of bounds
 * @since 9
 */
@ForceInline
public static
int checkIndex(int index, int length) {
    return Preconditions.checkIndex(index, length, null);
}
```

📎 Objects.checkFromToIndex()

```
/**
 * Checks if the sub-range from {@code fromIndex} (inclusive) to
 * {@code toIndex} (exclusive) is within the bounds of range from {@code 0}
 * (inclusive) to {@code length} (exclusive).
 *
 * <p>The sub-range is defined to be out of bounds if any of the following
 * inequalities is true:
 * <ul>
 *  <li>{@code fromIndex < 0}</li>
 *  <li>{@code fromIndex > toIndex}</li>
 *  <li>{@code toIndex > length}</li>
 *  <li>{@code length < 0}, which is implied from the former inequalities</li>
 * </ul>
 *
 * @param fromIndex the lower-bound (inclusive) of the sub-range
 * @param toIndex the upper-bound (exclusive) of the sub-range
 * @param length the upper-bound (exclusive) the range
 * @return {@code fromIndex} if the sub-range within bounds of the range
 * @throws IndexOutOfBoundsException if the sub-range is out of bounds
 * @since 9
 */
public static
int checkFromToIndex(int fromIndex, int toIndex, int length) {
```

```
    return Preconditions.checkFromToIndex(fromIndex, toIndex, length, null);
}
```

📎 Objects.checkFromIndexSize()

```
/**
 * Checks if the sub-range from {@code fromIndex} (inclusive) to
 * {@code fromIndex + size} (exclusive) is within the bounds of range from
 * {@code 0} (inclusive) to {@code length} (exclusive).
 *
 * <p>The sub-range is defined to be out of bounds if any of the following
 * inequalities is true:
 * <ul>
 *  <li>{@code fromIndex < 0}</li>
 *  <li>{@code size < 0}</li>
 *  <li>{@code fromIndex + size > length}, taking into account integer overflow</li>
 *  <li>{@code length < 0}, which is implied from the former inequalities</li>
 * </ul>
 *
 * @param fromIndex the lower-bound (inclusive) of the sub-interval
 * @param size the size of the sub-range
 * @param length the upper-bound (exclusive) of the range
 * @return {@code fromIndex} if the sub-range within bounds of the range
 * @throws IndexOutOfBoundsException if the sub-range is out of bounds
 * @since 9
 */
public static
int checkFromIndexSize(int fromIndex, int size, int length) {
    return Preconditions.checkFromIndexSize(fromIndex, size, length, null);
}
```

assert

메서드의 제작자, 메서드의 소비자 입장으로 나눠서 생각해 볼 때

공개되지 않은 메서드 (private, protected) 의 경우라면

매개변수를 지정하는 쪽은 메서드의 제작자인 우리일 것이고

따라서 오직 유효한 값만을 우리가 메서드에 넘길 것임을 보증해야 한다.

이럴 때 assert 를 사용한다.

cf) BigInteger.mod()

즉 public 이 아닌 메서드라면 assert 단언문을 사용해 매개변수 유효성을 검증할 수 있다.

```
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;
}
```

단언한 조건이 무조건 참이라고 선언한다

- 실패하면 AssertionError 를 던진다
- 런타임에 아무런 효과도, 아무런 성능 저하도 없다