Java Specification

Table of contents

Javadoc 1
What makes a good specification 2
Formality 3
Contracts 4
Defensive programming 4

Introduced in 2026-03-12-object-oriented-programming-ii[1] (Lecture, Week 3, Part 2).

Javadoc

- Ordinary comments: `//` and `/* ... */`.
- **Javadoc comments**: begin with `/**` (note the second `*`), end with `*/`, must sit immediately above the thing being documented, and use tags beginning with `@` (some take parameters, some just text).

```
/**
 * Calculates a sum by combining the hash code of the provided string
 * and the long value of the provided float.
 *
 * @param inputString the input string whose hash code will be used
 * @param inputFloat the float value to be converted to long and combined with the
 ↪  string hash code
 * @return a long value representing the sum of the string's hash code and the long
 ↪  value of the float
 */
public long doCalculation(String inputString, float inputFloat) { ... }
```

[1] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf

Common tags:

Tag	Meaning
`@param varname ...`	Describe a parameter
`@return ...`	Describe the return value
`@throws ExceptionType ...`	Describe when a particular exception is thrown
`@requires Precondition`	Assumptions for the method to execute properly
`@ensures Postcondition`	Effects of executing the method
`@author authorname`	Author of the class

What makes a good specification

Ideally, a specification should:

- Allow a method to be *used* by only reading its specification, not its implementation.
- Allow a method to be *re-implemented* without requiring changes to its callers.
- Be **restrictive** enough to rule out unacceptable implementations.
- Be **general** enough to not preclude acceptable (alternative) implementations.
- Be **clear** enough for programmers to understand.

Restrictiveness — keep out incorrect implementations

```
/**
 * Returns an index (i) of ar such that ar[i] == x, if any.
 */
public int search(int[] ar, int x) { ... }
```

What happens if `x` isn't in `ar`? The spec doesn't say — by its silence it allows *any* return value, so a caller can't distinguish "found at index 0" from "not found". Adding `else, return -1` fixes that, but if `x` appears multiple times, nothing requires the lowest index or a consistent answer across calls — the spec is still non-deterministic unless it says `smallest index`.

Generality — allow acceptable alternative versions

```java
/**
 * Examine ar[0], ar[1], ... in turn and return the index of the
 * first one that is equal to x, if any, else return -1.
 */
public int search(int[] ar, int x) { ... }
```

This is a **bad** spec even though it's restrictive: it describes *how* (forward iteration order), not *what*. A backward-iterating implementation that returns the same first-match index would violate this over-specific wording despite being equally acceptable — specs should describe outcomes, not implementation strategy. Prefer wording like "return the **smallest** index such that…", which both rules out bad implementations and permits any implementation strategy that achieves the same outcome. Both a backward-iterating and an early-`return`-on-first-match forward implementation satisfy this better wording equally well:

```java
public int search(int[] ar, int x) {
    for (int i = 0; i < ar.length; i++) {
        if (ar[i] == x) {
            return i; // early return, still finds the smallest index
        }
    }
    return -1;
}
```

Clarity

A specification should facilitate communication — it can fail either because the reader doesn't understand, or because the reader only *thinks* they understand. Clarity improves by being concise (long specs are more likely to contain contradictions, be skipped, or be misread) and by marking any deliberate redundancy explicitly (e.g. with "e.g." or "i.e.").

Formality

Specifications range from informal to formal:

- **Informal** — plain comments, e.g. `// Withdraws an amount and returns how much is left.` Leaves open questions (what if `amount` is negative? bigger than the balance? is the balance changed on failure?).
- **Semi-formal** — structured English via Javadoc tags (`@param`, `@return`, `@throws`).
- **Formal** — mathematical/boolean constraints, e.g. `@require`/`@ensure` using Java boolean-expression syntax:

```
/**
 * Withdraws an amount from this account.
 *
 * @require amount >= 0
 * @require amount <= getBalance()
 * @ensure getBalance() == \old(getBalance()) - amount
 */
public int withdraw(int amount) { ... }
```

Contracts

A specification can be written as a **contract**:

- If the caller satisfies the precondition, the method guarantees to satisfy the postcondition.
- If the caller does *not* satisfy the precondition, the method guarantees nothing — any behaviour is allowed, and the method body doesn't need to check for it.

This contrasts with a defensive, "no contract" style that instead documents and handles every failure case explicitly (e.g. returning a sentinel value like `-1` on invalid input) — contracts push that responsibility onto the caller instead.

Defensive programming

> Explicitly checking for invalid inputs and bad situations, ensuring the software does not behave dangerously regardless of input.

Even with a documented precondition, some caller eventually won't check it. When dealing with external input or guarding critical resources, it's often safer to validate defensively at the system boundary (throwing on bad input, e.g. `IllegalArgumentException` for a null/empty array) while relying on well-defined contracts *between* internal methods:

```
/**
 * Finds the maximum value in the given array of integers.
 *
 * @throws IllegalArgumentException if the array is null or empty.
 * @requires numbers != null && numbers.length > 0
 * @ensures the method returns the maximum value found in the array.
 */
public int findMax(int[] numbers) {
    if (numbers == null || numbers.length == 0) {
        throw new IllegalArgumentException("Array must not be null or empty.");
    }
    int max = Integer.MIN_VALUE;
    for (int i = 0; i < numbers.length; i++) {
```

```
        if (numbers[i] > max) {
            max = numbers[i];
        }
    }
    return max;
}
```

Further reading: Preconditions, Postconditions, and Class Invariants, Assertions, and Effective Java (3rd ed.), Item 56: Write doc comments for all exposed API elements.