Java Hoare Logic

Table of contents

Introduced in 2026-05-21-correct-programming[1] (Lecture, Week 12). Formalises reasoning about java-specification[2]'s preconditions/postconditions.

What does it mean for a program to be correct?

> A correct program **satisfies its specification**.

Correctness is only meaningful *relative to a specification* — asking "is this program correct?" without one is unanswerable:

```java
public int indexOf(int[] numbers, int number) {
    for (int i = 0; i < numbers.length; i++) {
        if (numbers[i] == number) {
            return i;
        }
    }
    return -1;
}
```

- With **no spec at all**: maybe correct, maybe not — we can't say without knowing what it's *meant* to do.
- Spec'd as `/** Returns how many times number occurs in numbers. */`: **incorrect** — counter-example `indexOf({1, 1, 1, 1}, 1) == 0`, but a correct implementation of *that* spec should return `4`.

[1] courses/csse2002/2026-05-21-correct-programming.pdf
[2] courses/csse2002/java-specification.pdf

- Spec'd as `/** Returns the index of number within numbers, if number is in numbers. */`: **correct**, but the spec is *underspecified* — it says nothing about what happens when `number` isn't in `numbers`.

A specification restricts the space of acceptable implementations: think of all possible input/output pairs as arrows crossing an Implementation/Specification boundary — an implementation is correct exactly when every arrow it draws stays within the region the specification permits. A `@requires`/`@ensures` Javadoc comment (see java-specification[3]) states the precondition and postcondition that bound this region.

Hoare triples

Preconditions and postconditions aren't just for whole methods — **every block of code has them**. We write this as a **Hoare triple**:

$$\{P\}\ S\ \{Q\}$$

meaning: if precondition P holds before statement(s) S run, postcondition Q holds afterwards. E.g.:

```
// {true}
if (x > y) {
    max = x;
} else {
    max = y;
}
// {max >= x && max >= y}
```

Deriving preconditions by propagating backward

Given a method's postcondition, we can work out its precondition (or verify a block of code) by propagating the postcondition **backward** through each statement, substituting as we go.

Straight-line code

[3] courses/csse2002/java-specification.pdf

```
/**
 * @requires numbers != null && 2 < numbers.length && numbers[2] == number
 * @ensures numbers[\result] == number
 */
public int indexOf(int[] numbers, int number) {
    // {numbers[0 - 10 + 12] == number}
    int result = 0;
    // {numbers[result - 10 + 12] == number}
    result = result + 12;
    // {numbers[result - 10] == number}
    result = result - 10;
    // {numbers[result] == number}
    return result;
    // {numbers[\result] == number}
}
```

Each line's precondition is obtained by substituting the assigned expression into the *next* line's already-derived precondition, working from the `@ensures` postcondition backward to the top of the method — the resulting `@requires` is whatever's left once you reach the very first line.

Through `if/else`

Each branch is handled separately, and the two derived preconditions are combined into a single condition depending on which branch executes:

```
/**
 * @requires numbers != null && 12 < numbers.length
 *               && numbers[12] == number && number % 2 == 0
 * @ensures numbers[\result] == number
 */
public int indexOf(int[] numbers, int number) {
    // {number % 2 == 0 ==> numbers[0 + 12] == number
    //  && number % 2 != 0 ==> numbers[0 - 10] == number}
    int result = 0;
    if (number % 2 == 0) {
        // {numbers[result + 12] == number}
        result = result + 12;
    } else {
        // {numbers[result - 10] == number}
        result = result - 10;
    }
    // {numbers[result] == number}
    return result;
}
```

Proving a block correct

The same backward-propagation technique proves an *arbitrary* block of code satisfies a given Hoare triple — substitute back through each branch/statement and confirm the starting precondition (`true`, in these examples) is actually implied:

```
// {true}
if (x > y) {
    // {x > y} ==> {x >= x && x >= y}
    max = x;
    // {max >= x && max >= y}
} else {
    // {y >= x} ==> {y >= x && y >= y}
    max = y;
    // {max >= x && max >= y}
}
// {max >= x && max >= y}
```

A three-statement swap works the same way, substituting each assignment's right-hand side backward through `\old(...)` references:

```
// {true}
// {y == \old(y) && x == \old(x)}
int tmp = x;
// {y == \old(y) && tmp == \old(x)}
x = y;
// {x == \old(y) && tmp == \old(x)}
y = tmp;
// {x == \old(y) && y == \old(x)}
```

A trickier variant swaps `x`/`y` using only arithmetic (no temporary variable), which still propagates the same way:

```
// {true}
// {y == \old(y) && x == \old(x)}
// {y - (y - x) + (y - x) == \old(y) && y - (y - x) == \old(x)}
x = y - x;
// {y - x + x == \old(y) && y - x == \old(x)}
y = y - x;
// {y + x == \old(y) && y == \old(x)}
x = y + x;
// {x == \old(y) && y == \old(x)}
```

(A similar swap is possible using bitwise XOR instead of subtraction, exploiting $a \oplus b = b \oplus a$, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$, $a \oplus 0 = a$, $a \oplus a = 0$.)