Java Hoare Logic

Table of contents

What does it mean for a program to be correct? 1
Hoare triples 2
Deriving preconditions by propagating backward 2
Writing the triple down: `\result` and `\old(...)` 7

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:

```
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.

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

- 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`.
- 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.

The direction is the part that trips people up: the listing reads top-to-bottom but is *derived* bottom-to-top, and each rung of the derivation is a single textual substitution.

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}
```

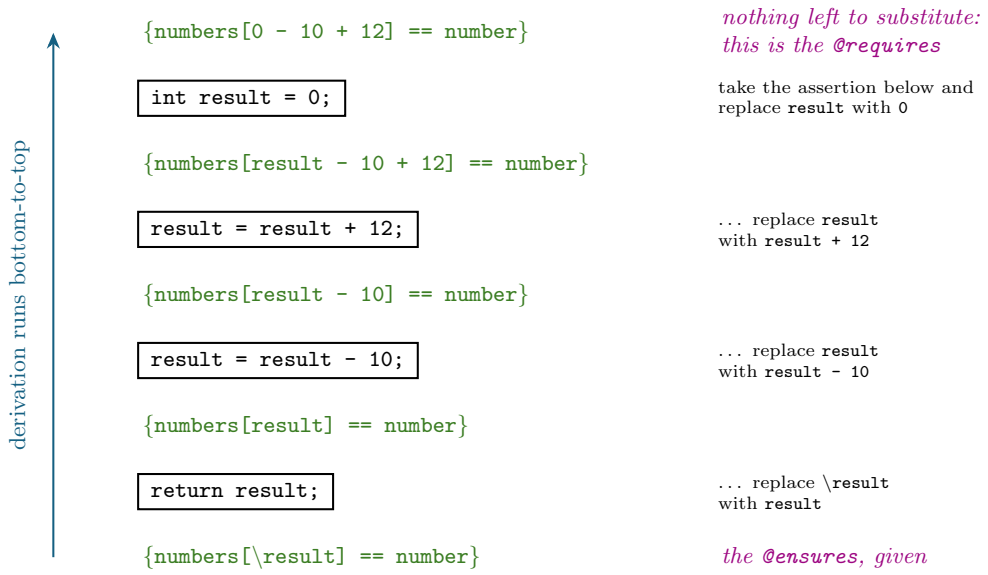

Figure 1: The same derivation as the listing above, laid out so the direction of the reasoning is visible: the `@ensures` at the bottom is given, each statement rewrites the assertion *below* it by replacing the assigned variable with the expression assigned to it, and whatever is left at the top is the `@requires`. The assertions are in green; they are `//` comments in the listing.

```
    return result;
}
```

Both branches must reach the *same* postcondition `{numbers[result] == number}`, so the derivation forks there, runs up each branch independently, and only rejoins once each branch's precondition has been guarded by the condition that selects it:

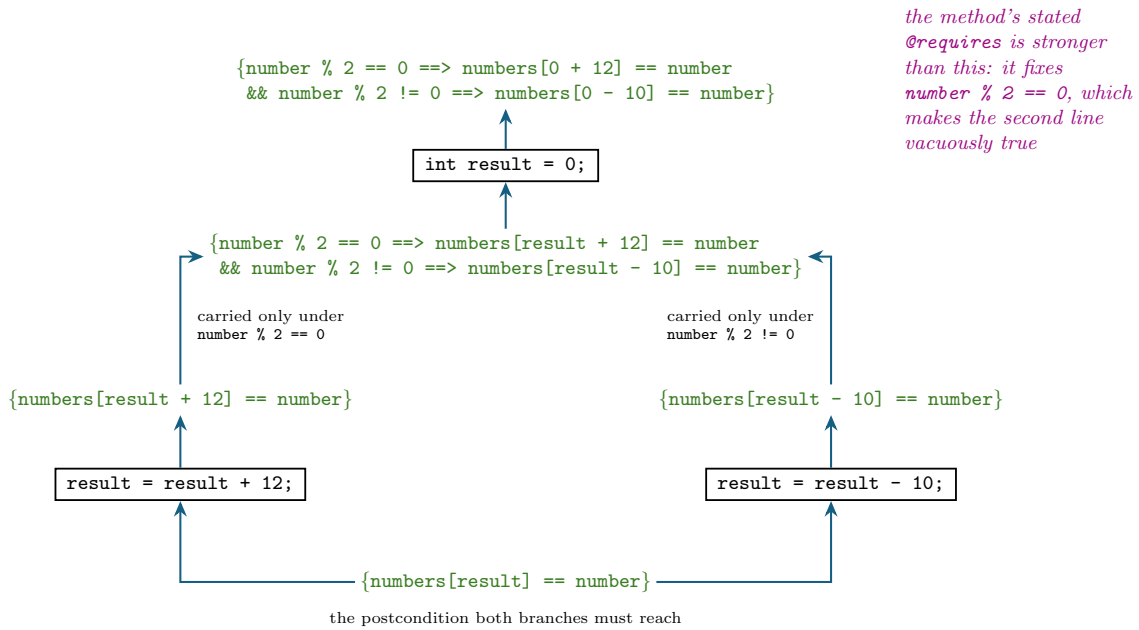

Figure 2: The `if/else` derivation above drawn as a fork and join. Reading upward: one shared postcondition, two independent branch derivations, then a single combined condition in which each branch's requirement is guarded by the test that reaches it, and finally one more substitution through `int result = 0;`. The combined assertion in the middle is the top condition one substitution earlier, a step the listing skips over.

Compare the top of Figure 2 with the `@requires` the method actually declares. The derived condition covers *both* branches; the declared one, `numbers != null && 12 < numbers.length && numbers[12] == number && number % 2 == 0`, pins `number` to the even case. That is legal — a **stronger** precondition than the derived one is always safe, because it still implies it (with `number % 2 == 0` fixed, the second disjunct is vacuously true) — it just means the method promises less than the code could deliver. Deriving gives you the *weakest* precondition; what you write in the Javadoc may be anything that implies it.

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 $a \oplus b = b \oplus a$, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$, $a \oplus 0 = a$, $a \oplus a = 0$.)

Writing the triple down: `\result` and `\old(...)`

A Hoare triple written in Java lives in a Javadoc comment (see java-specification[4]): P goes in the precondition tag, Q in the postcondition tag, and S is the method body. Two pieces of vocabulary let a postcondition say things a plain Java boolean expression cannot.

- **`\result`** names the value the method returns. It is the established way to refer to a return value inside a postcondition.
- **`\old(e)`** names the value the expression `e` had *before* the call. It is defined in java-specification[5] and is used again in the Week 12 practical. Without it the swap derivations above could not be stated at all: `x == y && y == x` says nothing, whereas `x == \old(y) && y == \old(x)` is exactly the swap.

> **i** The contract tags are spelled two ways in this course
>
> The tags are `@requires` / `@ensures` here and in the tag table in java-specification[6]. The **Week 12 practical sheet** (week12-lab-generics[7]) writes `@require` / `@ensure`, with no trailing `s`, and so do the SOLID contracts. Neither form is canonical. Use whichever form the sheet in front of you uses.

A worked contract from the Week 12 practical

The practical's `transferStudents` (Task 4) is the fullest contract in the Week 12 material, and it uses both pieces of vocabulary. Its `@ensure` writes `bus.getCurrentCount` without the parentheses:

```
/**
 * Transfer students to the new bus.
 *
 * @param students Bus only containing Concession fare riders.
 * @param bus      Bus to which students will transfer.
 * @require students != null
 *          && bus.getCapacity() >= students.getCurrentCount()
 * @ensure  students.getCurrentCount() == 0
 *          && bus.getCurrentCount >= \old(students).getCurrentCount()
 *          && bus contains all passengers from \old(students)
 */
```

[4] courses/csse2002/java-specification.pdf
[5] courses/csse2002/java-specification.pdf
[7] courses/csse2002/java-specification.pdf
[7] courses/csse2002/week12-lab-generics.pdf

```
public static void transferStudents(Bus<Concession> students,
        Bus<? super Concession> bus) { ... }
```

The method returns `void`, so there is no `\result` to talk about; everything the postcondition has to say is about how two objects' states *changed*, which is what makes `\old(...)` indispensable here. `students.getCurrentCount() == 0` is a fact about the state after; `\old(students).getCurrentCount()` reaches back to the state before, so the middle clause can hold the target bus's final count up against the number of passengers the source bus had before the call.

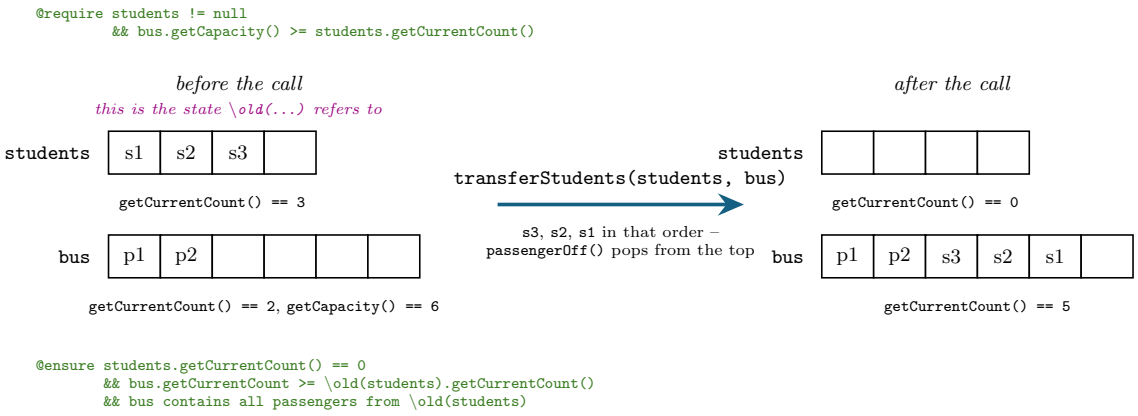

Figure 3: The Week 12 practical's `transferStudents` contract as a before/after picture, on a source bus of capacity 4 carrying three passengers and a target bus of capacity 6 already carrying two. Each bus is an array with its name to the left. The contract text is in green and the annotation in magenta.

Two things about this contract as written.

The postcondition mixes Java and English in the same tag. `students.getCurrentCount() == 0` is a boolean expression; `bus contains all passengers from \old(students)` is a sentence. The course convention is that conditions are written in Java boolean syntax, not mathematical notation, but the only queries `Bus` exposes are `getCurrentCount()` and `getCapacity()` (its other two methods, `passengerOn` and `passengerOff`, both mutate), so there is no expression in the API that *can* say "contains". The contract falls back to prose rather than inventing one, and the same happens in Task 3, whose whole postcondition is `@ensure \result contains trainee and all trainers`.

The precondition is weaker than the postcondition needs. `bus.getCapacity() >= students.getCurrentCount()` compares the target's *total* capacity against the number of students, not its *free* space. `Bus.passengerOn` (Task 0) silently does nothing when the bus is full:

```
public void passengerOn(T passenger) {
    if (currentCount < capacity) {
        passengers[currentCount++] = passenger;
    }
}
```

so a `bus` of capacity 6 already carrying 4, given a `students` bus of 3, satisfies the precondition (`6 >= 3`) yet drops the third student on the floor — `students.getCurrentCount() == 0` still holds, because `passengerOff` removes them regardless, but `bus contains all passengers from \old(students)` does not. The precondition that actually discharges the postcondition is `bus.getCapacity() - bus.getCurrentCount() >= students.getCurrentCount()`. The marked solution is the contract above, with the weaker precondition.

The loop inside that method — and why its bound is copied into a local variable before the loop starts — is taken up in java-loop-invariants[8].

[8] courses/csse2002/java-loop-invariants.pdf