Java Loop Invariants

Table of contents

What is an invariant? 1
Loop invariants 2
Deriving an invariant for `power` 2
Designing an algorithm *from* an invariant: fast exponentiation 4
Pragmatic considerations 5

Introduced in 2026-05-21-correct-programming[1] (Lecture, Week 12). Extends java-hoare-logic[2]'s backward-reasoning technique to loops, where the number of iterations isn't known up front.

> …proving the correctness of algorithms has another aspect that is even more important: it mirrors the way we understand an algorithm.
>
> — Donald Knuth, *The Art of Computer Programming* (Vol. 1), 1997

What is an invariant?

> An **invariant** is a property of a system that does not change.

Motivating puzzle

A jar contains 100 red and 100 blue beans. Repeatedly: pick two random beans; if they're the same colour, discard both; if they're different colours, discard the blue one. **What colour is the last bean?**

Let r, b be the current counts of red/blue beans. Each step does exactly one of:

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

Pick	Action	Effect
two reds	discard both	$r \mapsto r-2,\ b \mapsto b$
two blues	discard both	$r \mapsto r,\ b \mapsto b-2$
one of each	discard blue	$r \mapsto r,\ b \mapsto b-1$

What never changes? The *parity* of r — every action either leaves r unchanged or decreases it by exactly 2, so r stays even throughout (it starts at 100, itself even). When one bean remains, $r + b = 1$; since r must be even, $r \neq 1$, so $r = 0$ and $b = 1$ — **the last bean is blue**.

The property "r is even" is an invariant of the process: it (1) holds before any step, (2) is preserved by every step, and (3), combined with the termination condition ($r + b = 1$), implies what we wanted to prove. This is exactly the structure of a **loop invariant**.

Loop invariants

> A **loop invariant** is a condition that: (1) holds before the loop is entered, (2) is preserved by each iteration, and (3) combined with the negation of the guard, implies the postcondition.

```
int count = 0;
while (count < n) {
    // invariant: count >= 0
    count = count + 1;
}
```

```
// {0 >= 0}
int count = 0;
// {count >= 0}                    <-- invariant holds before the loop
while (count < n) {
    // invariant: count >= 0
    // {count + 1 >= 0}
    count = count + 1;
    // {count >= 0}                <-- preserved by the body
}
// {count >= 0 && count >= n}
```

Deriving an invariant for `power`

```
/**
 * @requires ????
 * @ensures \result == base^exp
 */
int power(int base, int exp) {
    int result = 1;
    int i = 0;
    while (i < exp) {
        result = result * base;
        i = i + 1;
    }
    return result;
}
```

Step 1 — guess an invariant. Replace `exp` in the postcondition with the loop variable `i`: I : result $=$ base^i.

Step 2 — verify. Three checks:

1. Does I hold *before* the loop first executes? `result = base^i` $\equiv 1 =$ base$^0 \equiv$ true.
2. Assuming I holds at the top of the loop body, does it still hold after? `result * base = base^i * base = base^{i+1}`, and after `i = i + 1`, that's exactly `result = base^i` again.
3. Does I combined with the negated guard ($i \geq$ exp) imply the postcondition? `result = base^i` $\wedge\ i \geq$ exp $\Rightarrow$ result $=$ base$^{\text{exp}}$? **Not quite** — we can only conclude $i \geq$ exp, but we need $i =$ exp.

Step 3 — strengthen. Add the bound $i \leq$ exp to the invariant: I : result $=$ base$^i \wedge i \leq$ exp. Re-checking: I now holds initially iff exp ≥ 0 (giving us our precondition), is still preserved by the loop body, and combined with $i \geq$ exp now forces $i =$ exp exactly, giving the postcondition:

```
/**
 * @requires exp >= 0
 * @ensures \result == base^exp
 */
int power(int base, int exp) {
    // {0 <= exp}
    // {1 == base^0 && 0 <= exp}
    int result = 1;
    int i = 0;
    // {result == base^i && i <= exp}
    while (i < exp) {
        ...
    }
```

```
    // {result == base^i && i <= exp && i >= exp}
    // {result == base^exp}
    return result;
}
```

Designing an algorithm *from* an invariant: fast exponentiation

The `power` above runs in $O(\text{exp})$ multiplications. We can do better using the identity:

$$b^e = \begin{cases} (b^{e/2})^2 & \text{if } e \text{ is even} \\ b \times (b^{(e-1)/2})^2 & \text{if } e \text{ is odd} \end{cases}$$

E.g. computing 3^{13}: $3^{13} = 3 \times 3^{12}$ [13 odd] $= 3 \times (3^2)^6$ [12 even] $= 3 \times 9^6 = 3 \times (9^2)^3$ [6 even] $= 3 \times 81^3 = 3 \times 81 \times 81^2$ [3 odd] $= 3 \times 81 \times 6561 = 1{,}594{,}323$ — only a handful of multiplications instead of 12 sequential ones.

Design from the invariant: rather than writing code and *then* finding its invariant, pick the invariant *first* and let the code follow. We want a loop maintaining:

$$I : \text{result} \times b^e = \text{base}^{\text{exp}}$$

starting from `result = 1`, `b = base`, `e = exp`, terminating when `e == 0`:

```
/**
 * @requires exp >= 0
 * @ensures \result == base^exp
 */
int power(int base, int exp) {
    int result = 1;
    int b = base;
    int e = exp;
    while (e > 0) {
        // invariant: result * b^e == base^exp
        if (e % 2 == 1) {
            result = result * b;
            e = e - 1;
        }
        b = b * b;
        e = e / 2;
    }
    return result;
}
```

Verifying I (using $b^e = (b^{e/2})^2$ when even, $b \times (b^{(e-1)/2})^2$ when odd, and $(a^b)^c = (a^c)^b$):

1. Before the loop: `result * b^e == base^exp` $\equiv 1 \times \text{base}^{\text{exp}} = \text{base}^{\text{exp}} \equiv$ true.
2. Through the body: splitting on `e % 2`, both branches reduce back to exactly `result * b^e == base^exp` after the update (the `if` branch peels off one factor of `b` into `result` and decrements `e` first; both branches then square `b` and halve `e`).
3. On exit (`e <= 0`): `result * b^e == base^exp` with `e <= 0` only concludes the post-condition when `e >= 0` too (i.e. `e == 0`) — which holds as long as `exp >= 0`, giving the same precondition as before.

This preserves the *exact same specification* as the slow version — both satisfy `@requires exp >= 0` / `@ensures \result == base^exp` — but the fast version reaches it in $O(\log \text{exp})$ multiplications instead of $O(\text{exp})$.

Pragmatic considerations

Proving correctness can be vital, but it's also time-consuming — you want code running on a plane or controlling trains to be correct; you may not care nearly as much whether a weekend web-scraping script is correct. Where formal reasoning tends to pay off:

- **Combinatorial algorithms** — proving a faster/trickier implementation still satisfies the original spec (as with `power` above).
- **Distributed and concurrent algorithms** — notoriously hard to reason about by testing alone (see CSSE3610/7610).

Reasoning about programs this way provides (1) a more rigorous way to think about your software, and (2) a better interface design for others to interact with it — it doesn't remove the need to think about your programs, but gives a framework to structure that thinking.

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> — Donald Knuth (memo to Peter van Emde Boas, 1977)