Java Loop Invariants

Table of contents

What is an invariant? 1
Loop invariants 2
Deriving an invariant for `power` 3
Designing an algorithm *from* an invariant: fast exponentiation 4
Invariants in the Week 12 exercises 6
Pragmatic considerations 9

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.

Those three clauses are three separate proof obligations, and they attach to three different places around the loop:

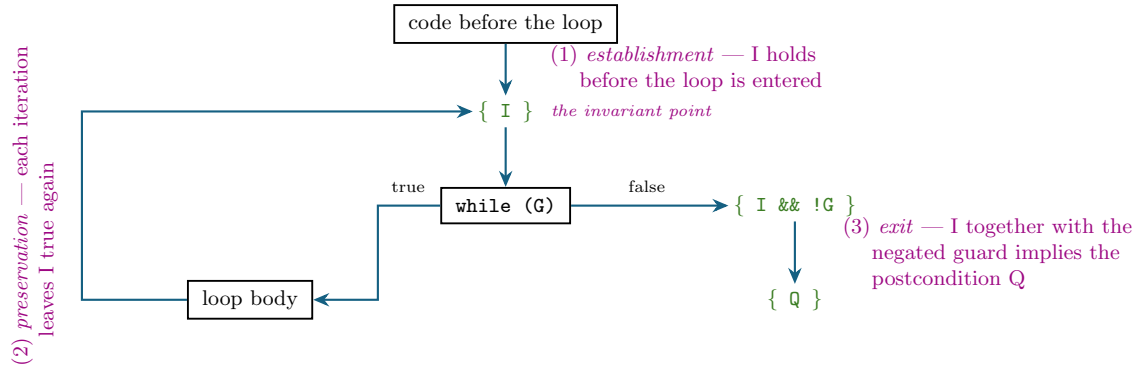

Figure 1: The three obligations in the definition above, each pinned to the point in the loop where it has to be discharged. The invariant point is the top of the loop, before the guard is tested: establishment is the arrow into it, preservation is the arrow back to it, and exit is what the guard's failure adds to it. Connectors are teal and assertions are green.

None of the three clauses in Figure 1 says anything about the loop *finishing*. An invariant that survives all three checks establishes only "*if* the loop ends, the postcondition holds"; termination is a separate argument — in `power` below, that `exp - i` strictly decreases and is bounded below.

```
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 ($i \geq \text{exp}$) imply the postcondition? `result = base^i` $\wedge\ i \geq \text{exp} \Rightarrow \text{result} = \text{base}^{\text{exp}}$? **Not quite** — we can only conclude $i \geq \text{exp}$, but we need $i = \text{exp}$.

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

The precondition comes from obligation (1) of Figure 1, failing, and being read backwards. `exp >= 0` was never designed in — it is whatever the establishment check needed in order to succeed.

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.

The invariant is easiest to believe by watching it *not move*. Running the loop on the same 3^{13} worked above:

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 $O(\text{exp})$.

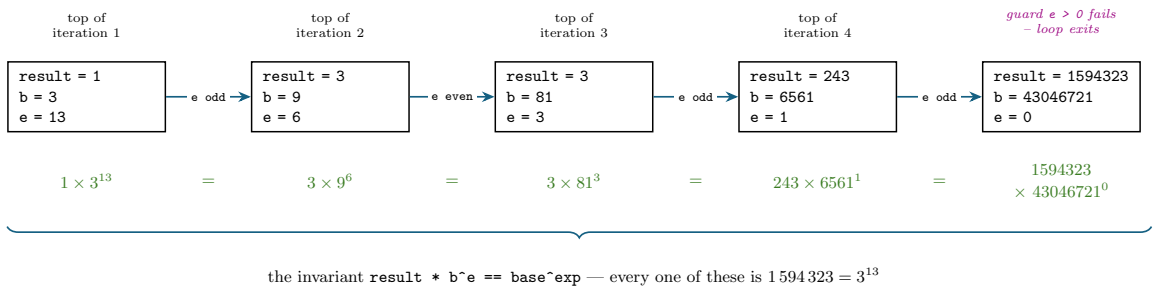

Figure 2: The fast-`power` loop traced on `power(3, 13)`, one box per arrival at the top of the loop. The three variables change on every step; the product `result * b^e` underneath does not, and the five products it takes are exactly the chain the section above works by hand (the section's `3 x 81 x 6561` and this trace's `243 x 6561` are the same number). Connectors are teal and the invariant is green.

Invariants in the Week 12 exercises

Neither Week 12 exercise set is *about* correctness — the applied class is lambdas and streams (week12-tutorial-lambdas-and-streams[3]) and the practical is generics (week12-lab-generics[4]) — but both contain loops whose behaviour is only explicable in these terms, and the practical writes contracts for two of them.

The loop bound that must not move

The practical's `transferStudents` (Task 4) empties one bus into another. Its loop carries a one-line comment:

```
// Ensure loop condition does not change while iterating
int numStudents = students.getCurrentCount();
for (int i = 0; i < numStudents; i++) {
    Concession student = students.passengerOff();
    bus.passengerOn(student);
}
```

The contract this loop has to establish is its `@ensure` (given in full in java-hoare-logic[5]), whose first clause is `students.getCurrentCount() == 0`. Taking the invariant at the top of the iteration with counter `i` to be

[3] courses/csse2002/week12-tutorial-lambdas-and-streams.pdf
[4] courses/csse2002/week12-lab-generics.pdf
[5] courses/csse2002/java-hoare-logic.pdf

`0 <= i && i <= numStudents && students.getCurrentCount() == numStudents - i`, and `bus` holds everything it started with plus the `i` students already moved,

the three obligations of Figure 1 go through: at `i == 0` nothing has moved and `getCurrentCount()` is still `numStudents`; each iteration takes exactly one passenger off and puts exactly one on before `i++`, restoring it; and exiting with `i == numStudents` gives `students.getCurrentCount() == 0` together with all `numStudents` students on `bus`.

Now write the guard the obvious way instead, reading the count live, and the invariant has nothing to hold onto: `i` climbs by one per iteration while `students.getCurrentCount()` falls by one, so the two meet halfway and the loop stops with half the bus still loaded.

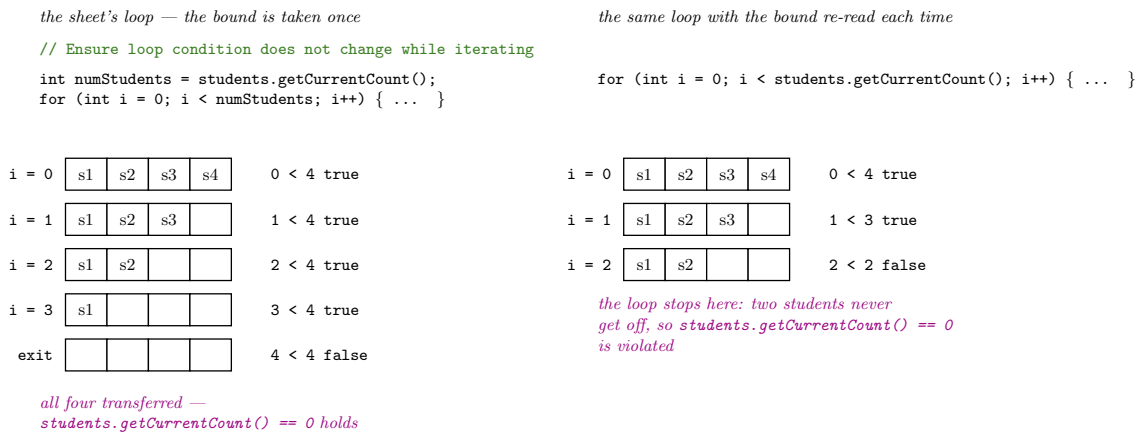

Figure 3: Why the practical snapshots its loop bound. Both columns run the same body on a source bus of four; the left column is the loop with the bound read once into `numStudents`, the right is the same loop with `students.getCurrentCount()` in the guard. Reading the count live makes the guard chase a quantity the body is decreasing, and the two meet after two iterations. Bus contents are drawn as array rows and the verdict on each column is in magenta.

In general the live-bound version runs $\lceil n/2 \rceil$ iterations and leaves $\lfloor n/2 \rfloor$ students behind, because `i` and the count meet at $i = n - i$. The fix is the standard one: **a loop guard should not read state the body is mutating.** Copying the bound into a local before the loop is what turns `numStudents` into something the invariant can be stated against. Task 5's `transferAllConcession` uses the same body and the same snapshot, with the type parameter `T` in place of `Concession`, so the same invariant carries over unchanged.

An accumulator loop inside a lambda

The applied class's Task 9 hides an ordinary counting loop inside a `Function`:

```
Function<Integer, Integer> factorial = n -> {
    int result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
};
System.out.println(factorial.apply(4));   // 24
```

The task asks only for the printed value. The invariant, following the `power` recipe of replacing the target quantity's bound with the loop variable, is I : result $= (i-1)!\ \wedge\ 2 \le i \le n+1$:

1. **Establishment.** At the top, `i == 2` and `result == 1 == 1!`. — but the bound clause needs $2 \le n+1$, so this invariant establishes only for `n >= 1`, which is the derived precondition. (`factorial.apply(0)` still returns `1`, correctly; that just isn't something *this* invariant proves.)
2. **Preservation.** `result * i == (i-1)! * i == i!`, and `i++` then makes that `(i-1)!` again.
3. **Exit.** The guard fails at `i > n`, which with `i <= n + 1` forces `i == n + 1`, so `result == n!`.

At `n == 4` that is `4! == 24`, derived rather than simulated.

The same idea, written into a method signature

The applied class's Task 18 computes the same shape of thing with no loop at all:

```
Stream<Integer> stream = Stream.iterate(1, n -> n * 2).limit(5);
Integer result = stream.reduce(0, (a, b) -> a + b);   // 31
```

`reduce`'s first argument is the identity and its second is the step. Read as a loop, those are precisely obligations (1) and (2): the identity `0` establishes the invariant "the accumulator holds the sum of the elements consumed so far", and `(a, b) -> a + b` is the step that preserves it. When the stream runs out — the negated guard — the accumulator holds the sum of all five elements, $1+2+4+8+16 = 31$. Nothing about the reasoning has changed; the fold's signature just makes you write the invariant's two halves down where a `for` loop would have let you leave them implicit. See java-lambdas-and-streams[6] for `reduce` itself.

[6] courses/csse2002/java-lambdas-and-streams.pdf

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.

A loop invariant is not the only kind of invariant in this course: a **class invariant** is the same idea applied to an object's fields across every public method rather than to a loop's variables across every iteration — established by the constructor, preserved by each method, relied on by all of them. See week8-tutorial-class-invariants[7].

> 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)

[7] courses/csse2002/week8-tutorial-class-invariants.pdf