Class Invariants

Table of contents

Applied class for 2026-04-23-java-io[1] (Week 8). Applies java-encapsulation[2]'s class invariants (and the preconditions/postconditions from java-specification[3]) to worked examples.

`XFiles`

`XFiles` maintains the invariant that every stored string must be prefixed with `'X'`:

$$\forall\, 0 \le i < \texttt{getFiles().size()} \implies \texttt{getFiles().get(i).startsWith("X")}$$

```
/**
 * @invariant
 *     \forall i; 0 <= i < getFiles().size(); getFiles().get(i).startsWith("X")
 */
public class XFiles {
    /**
     * @ensures getFiles().contains(newFile)
     * @ensures getFiles().size() == \old(getFiles()).size() + 1
     */
    public void add(String newFile) {...}

    /**
     * @ensures !getFiles().contains(file)
     */
    public void remove(String file) {...}
```

[1] courses/csse2002/2026-04-23-java-io.pdf
[2] courses/csse2002/java-encapsulation.pdf
[3] courses/csse2002/java-specification.pdf

```
    public List<String> getFiles() {...}
}
```

As specified, `add` allows the invariant to be broken (nothing stops a non-`'X'` string being added). Fix: add a precondition

```
/**
 * @requires newFile.startsWith("X")
 */
```

Even with that precondition documented, a naive implementation still has two further leaks (see java-encapsulation[4] — Protecting invariants):

```
public class XFiles {
    public List<String> files = new ArrayList<>(); // (1) public field

    public void add(String newFile) {
        if (newFile.startsWith("X")) {
            files.add(newFile);
        }
    }
    public void remove(String file) { files.remove(file); }

    public List<String> getFiles() {
        return files; // (2) leaks the internal reference
    }
}
```

1. **`files` is public** — callers can grab the list directly and mutate it, bypassing `add`'s check entirely: `xFiles.files.add("Doesn't start with X!")`. Fix: make it `private`.
2. **`getFiles()` returns the internal reference** — even with `files` made private, the *returned* list is still the real one: `xFiles.getFiles().add("Doesn't start with X!")` still breaks the invariant. Fix: return a defensive copy, `return new ArrayList<>(files);`.

`Cinema` and `Screening`

[4] courses/csse2002/java-encapsulation.pdf

```java
public class Cinema {
    /**
     * @ensures getCapacity() == capacity
     */
    public Cinema(int capacity) {...}
    public int getCapacity() {...}
}
```

> **i** Working
>
> **A useful invariant**: `getCapacity() >= 0`.
> **Precondition to preserve it**: the constructor needs `capacity >= 0`.

```java
/**
 * @invariant getEndTime() > getStartTime()
 */
public class Screening {
    /** @ensures \result > 946648800 */
    public int getStartTime() {...}
    /** @ensures \result > 946648800 */
    public int getEndTime() {...}
    /** @ensures getEndTime() == \old(getEndTime()) + amount */
    public int extendRuntime(int amount) {...}
}
```

(Timestamps are unix time; `946648800` = 1st January 2000, when the cinema opened.)

> **i** Working
>
> **Do the methods preserve the invariant?** No — a negative `amount` passed to `extendRuntime` could push `getEndTime()` below (or equal to) `getStartTime()`. Two sensible fixes:
>
> a. `amount >= 0`
> b. `getEndTime() + amount > getStartTime()`
>
> The method's name (**extend**Runtime) implies (a) is the more likely intended precondition.

Now extend `Screening` with ticket sales:

```java
public class Screening {
    // getStartTime(), getEndTime(), extendRuntime(int) as before.
    public Cinema getCinema() {...}
```

```
    /**
     * @ensures getSoldTickets().contains(\result)
     * @ensures getSoldTickets().size() == \old(getSoldTickets()).size() + 1
     */
    public Ticket sellTicket() {...}

    public Set<Ticket> getSoldTickets() {...}
}
```

Working

Invariant to prevent overselling:

```
/**
 * @invariant getSoldTickets().size() <= getCinema().getCapacity()
 */
```

Precondition to preserve it — add to `sellTicket()`: `getSoldTickets().size() < getCinema().getCapacity()`. (Technically, since the invariant can be *assumed* as a precondition too, `getSoldTickets().size() != getCinema().getCapacity()` is enough — the invariant plus that inequality together imply the strict `<`.)

Now consider a concrete (simplified) implementation:

```
public class Screening {
    private Set<Ticket> tickets = new HashSet<>();

    public Ticket sellTicket() {
        Ticket ticket = new Ticket(...);
        tickets.add(ticket);
        return ticket;
    }

    public Set<Ticket> getSoldTickets() {
        return tickets;
    }
}
```

Working

Does this protect the invariant? Not really, for two reasons:

1. In pure programming-by-contract, `sellTicket()` doesn't check the precondition

at all — under a pure contract it's *allowed* to do anything if the caller violates the precondition, but that's fragile if `sellTicket()` is exposed to code you don't control. Defensive programming (see java-specification[5]) is safer here:

```
public Ticket sellTicket() {
    if (tickets.size() >= getCinema().getCapacity()) {
        throw new IllegalArgumentException("Screening full!");
    }
    ...
}
```

(Returning `null` instead of throwing protects the invariant too, but risks a hard-to-trace `NullPointerException` later in the caller — throwing immediately at the point of misuse is clearer.)

2. The *real* bug is that `getSoldTickets()` leaks the internal `tickets` reference — external code can oversell the screening directly: `screening.getSoldTickets().add(new Ticket())`, bypassing `sellTicket()` entirely. Fix with a defensive copy:

```
public Set<Ticket> getSoldTickets() {
    return new HashSet<>(tickets);
}
```

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