Class Invariants

Table of contents

XFiles 1
Cinema and Screening 3

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.

Two failure modes run through the sheet, and it is worth naming them up front, because every *fix* it asks for is a response to one or the other. (The grouping is mine, not the sheet's; Tasks 0 and 3, which only ask you to state an invariant, sit outside it.)

1. **The specification lets the invariant break.** No precondition rules out the argument that would violate it.
2. **The implementation lets the invariant break**, even though the specification is fine — because it hands out a reference to its own internal state (see Figure 1).

`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 {
    /**
```

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

```
     * @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) {...}

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

As specified, **`add` allows the invariant to be broken**: given a string that does *not* start with `'X'`, the specification still states that the string will be included in `getFiles()`. Fix — failure mode (1) — 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:

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

```
List<String> secret = xFiles.files;
secret.add("Doesn't start with X!"); // breaks the invariant
```

Fix: `private List<String> files;`.

2. **`getFiles()` returns the internal reference** — even with `files` made private, the *returned* list is still the real one:

```
List<String> secret = xFiles.getFiles();
secret.add("Doesn't start with X!"); // still breaks the invariant
```

Fix: return a defensive copy, `return new ArrayList<>(files);`.

Leak (2) is failure mode (2), and it is the one worth having a picture of — the same leak, in the same shape, is the "real bug" of the `Screening` task below (`return tickets;` on a `Set<Ticket>` in place of `return files;` on a `List<String>`), and is drawn there in Figure 1.

`Cinema` and `Screening`

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

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

```
/**
 * @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) {...}
}
```

(All times are unix timestamps — seconds since 1 January 1970. `946648800` is 1 January 2000, when the cinema opened.)

> **i** Working
>
> **Do the methods preserve the invariant?** No — the invariant may be violated if a **negative** `amount` is given to `extendRuntime(int)`, pushing `getEndTime()` below (or equal to) `getStartTime()`. Two sensible preconditions on `extendRuntime`:
>
> a. `amount >= 0`
> b. `getEndTime() + amount > getStartTime()`
>
> The name of the method (**extend**Runtime) implies the first is more likely.

Now extend `Screening` with ticket sales:

```
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() {...}
}
```

> **i** Working
>
> **Invariant to stop the screening overselling**:
>
> ```
> /**
>  * @invariant getSoldTickets().size() <= getCinema().getCapacity()
>  */
> ```
>
> **Precondition to preserve it** — added to `sellTicket()`, so we don't sell into a full cinema:
>
> ```
> getSoldTickets().size() < getCinema().getCapacity()
> ```
>
> Technically, since the invariant may itself be *assumed* as another precondition, specifying `getSoldTickets().size() != getCinema().getCapacity()` is enough — that inequality and the invariant 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;
    }
}
```

i Working

Does this protect the invariant? Not really, for two reasons.
1. `sellTicket()` never checks its precondition. In true programming by contract that is defensible — we make no guarantees about what happens when a precondition is violated. But if `sellTicket()` is likely to be exposed to others, we should aim to program **defensively** as well as by contract (see java-specification[5]), which improves robustness:

```
public Ticket sellTicket() {
    if (tickets.size() >= getCinema().getCapacity()) {
        return null;
    }
    ...
}
```

That protects the invariant but potentially introduces a difficult-to-track-down `null` bug in the calling code. A better solution:

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

2. The real bug is that `getSoldTickets()` exposes the reference to the internal `tickets` set, allowing the screening to become oversold without `sellTicket()` ever being called:

```
Screening screening = // screening with cinema of capacity = 100
for (int i = 0; i < 101; i++) {
    screening.getSoldTickets().add(new Ticket());
}
```

Fix by returning a copy:

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

(The solutions sheet writes this fix as `public Ticket getSoldTickets() { return new ArrayList<>(tickets); }` — a typo on two counts: the field is a `Set<Ticket>`, and the return type is `Set<Ticket>`, not `Ticket`. The intent is a defensive copy of the same collection type.)

The pattern to take away: **an invariant is only as strong as the widest way in to the state it constrains.** A precondition closes the front door; `private` closes the side door; a defensive copy closes the one left open by every getter that returns a mutable object.

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

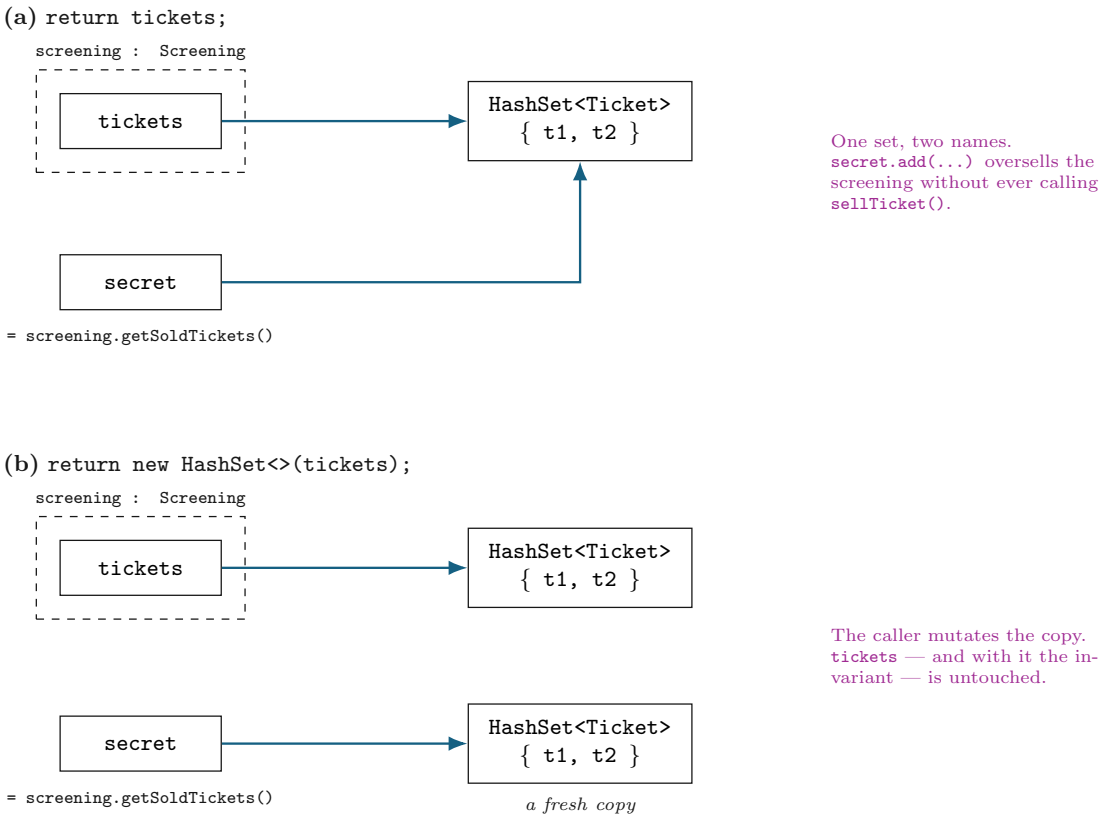

Figure 1: Why returning the internal collection breaks an invariant that the class otherwise enforces. (a) `return tickets;` — the caller's `secret` and the object's own field name **one** set, so `secret.add(...)` writes straight into `Screening`'s state, past every check `sellTicket()` makes. (b) `return new HashSet<>(tickets);` — the caller gets its own set to mutate. Dashed box = the object's encapsulation boundary; teal arrow = a reference. **CSSE2002 establishes no notation for object/reference (boxes-and-pointers) diagrams** — object diagrams, stack-and-heap diagrams and has-a relationships are all left unsettled by the course's own materials — so this rendering is mine. Only the teal reference colour and the magenta verdict text follow the course's own conventions.