Substitution Principle and Pre/Post Conditions

Table of contents

Predicate strength 1
Substitution principle 3
Writing pre/postconditions 5

Applied class for 2026-04-30-generics-in-java[1] (Week 9). More practice applying the Liskov Substitution Principle (see java-solid-principles[2]) and pre/postconditions (see java-specification[3]).

Everything here turns on one idea: **a condition is *stronger* when it is more restrictive** — when satisfying it implies satisfying the other, but not the reverse. Every task below is an application of that, first to bare predicates, then to a subclass's contract.

Predicate strength

	A	B
1	`a < 0 && b < 0`	`a != 0 && b < 0`
2	`a instanceof Animal`	`a instanceof String`
3	`a instanceof Animal || b instanceof Zebra`	`a instanceof Animal && b instanceof Zebra`
4	`a instanceof Animal && b instanceof Zebra`	`a instanceof Animal`
5	`a instanceof Animal && b instanceof Object`	`a instanceof Animal`
6	`a instanceof Zebra`	`a instanceof Tiger`

[1] courses/csse2002/2026-04-30-generics-in-java.pdf
[2] courses/csse2002/java-solid-principles.pdf
[3] courses/csse2002/java-specification.pdf

For each row, identify which column is a **stronger** (more restrictive) condition, or state that there's no relation.

> **Working**
>
> 1. **A is stronger than B** — `a < 0 && b < 0` implies `a != 0 && b < 0` (every `a < 0` is also `a != 0`), but not vice versa.
> 2. **No relation** — `Animal` and `String` are unrelated types; neither instance check implies the other.
> 3. **B is stronger than A** — A only needs *either* condition to hold (a looser OR), while B requires *both* (a stricter AND) — satisfying B always satisfies A, not the reverse.
> 4. **A is stronger than B** — A requires everything B requires, plus more (`b instanceof Zebra` on top of `a instanceof Animal`).
> 5. **A looks stronger than B, but they're equivalent** — since *everything* is an instance of `Object`, `b instanceof Object` is always true, so it adds no actual restriction.
> 6. **No relation** — `Zebra` and `Tiger` are siblings (both presumably subtypes of `Animal`, say), so neither instance check implies the other.

Now consider two possible class structures:

```
// Option 1: Preconditions
class ClassA {
    /** @requires A */
    void f(Object a, Object b) {}
}
class ClassB extends ClassA {
    /** @requires B */
    void f(Object a, Object b) {}
}
```

```
// Option 2: Postconditions
class ClassA {
    /** @ensures A */
    void f(Object a, Object b) {}
}
class ClassB extends ClassA {
    /** @ensures B */
    void f(Object a, Object b) {}
}
```

For each row above, if column A and column B are inserted as the specification text, which option (if any) satisfies the substitution principle?

Working

Recall: a subclass must not *strengthen* preconditions (they may only stay the same or weaken) and must not *weaken* postconditions (they may only stay the same or strengthen).

1. **Option 1 (precondition)** — `ClassB`'s precondition (B) is weaker than `ClassA`'s (A), which is allowed for preconditions.
2. **Neither** — unrelated conditions satisfy neither the "no stronger precondition" nor "no weaker postcondition" rule.
3. **Option 2 (postcondition)** — `ClassB`'s postcondition (B) is stronger than `ClassA`'s (A), which is allowed for postconditions.
4. **Option 1 (precondition)** — same reasoning as row 1: B is weaker than A.
5. **Both** — since A and B are equivalent here, substituting either as precondition or postcondition preserves the (unchanged) strength relationship.
6. **Neither** — unrelated conditions again satisfy neither rule.

Rows 1, 3 and 4 each land in exactly *one* column: a strength relation that is legal for a precondition is illegal for a postcondition and vice versa, which is why "B is weaker" is never on its own an answer — you have to know which end of the contract it sits at.

Substitution principle

```
class X {
    /**
     * @require fontSize >= 5
     * @ensure \result >= 0
     */
    int detexify(Object symbol, float fontSize) {
        // Things happen here
    }
}

class Y extends X {
    /**
     * @require fontSize > 0
     * @ensure \result > 0
     */
    int detexify(Object symbol, float fontSize) {
        // Other things happen here
    }
}

class Z extends X {
```

```
    /**
     * @require fontSize > 5
     * @ensure \result > 0
     */
    int detexify(Object symbol, float fontSize) {
        // Do other things happen here?
    }
}
```

The tags are spelled `@require`/`@ensure` here but `@requires`/`@ensures` in the sections either side. Both spellings appear across the course — see csse2002-diagram-notation[4], Inconsistencies 5 — so neither is safe to treat as *the* house form.

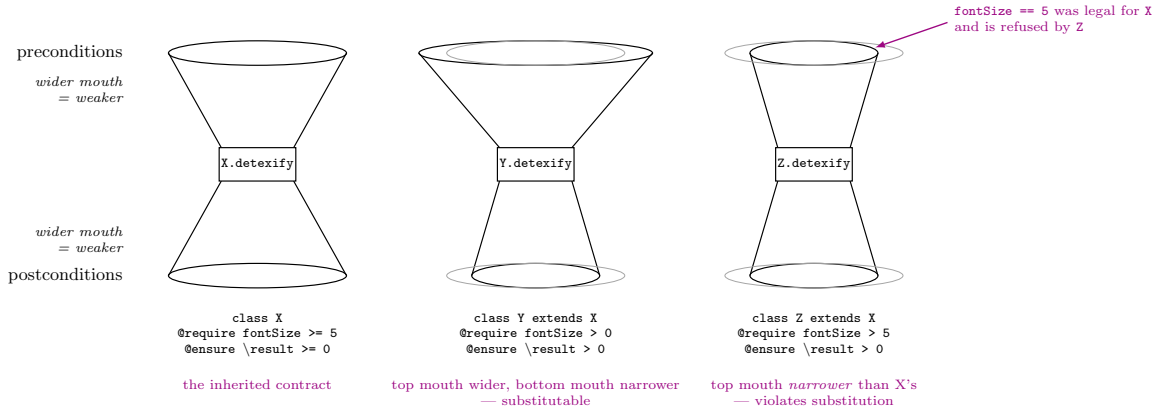

Figure 1: `X`, `Y` and `Z` drawn as the pre/postcondition funnels from 2026-04-16-solid-principles-ii[5]. The mouth above each box is the set of inputs the method accepts, the mouth below is the set of outcomes it may produce, and a *wider* mouth is a *weaker* condition — so a legal override needs a top mouth at least as wide as its parent's and a bottom mouth no wider. `X`'s two mouths are ghosted over `Y` and `Z` for comparison. Read off: `Y` widens the top and narrows the bottom, both legal; `Z` narrows the top, which is what breaks it. The verdicts are in magenta, with a pointer to the input `Z` refuses.

Why would a programmer choose to use a precondition at all?

> **Working**
>
> As a restriction on the allowable range of inputs, to avoid having to handle huge positive/negative numbers (or other awkward edge cases) inside the method body.

[4] courses/csse2002/csse2002-diagram-notation.pdf
[5] courses/csse2002/2026-04-16-solid-principles-ii.pdf

Does `Y` violate the substitution principle?

> **Working**
>
> No. `Y`'s precondition (`fontSize > 0`) is *weaker* than `X`'s (`fontSize >= 5` implies `fontSize > 0`, but not vice versa — the range of acceptable inputs expanded), and `Y`'s postcondition (`\result > 0`) is *stronger* than `X`'s (`\result >= 0` — the range of possible outputs contracted). Both changes are allowed by the principle.

Does `Z` violate the substitution principle?

> **Working**
>
> Yes. `Z`'s postcondition is correctly *more restrictive* of possible outputs (`\result > 0` vs. `\result >= 0`). However, the precondition is *also* more restrictive (`fontSize > 5` vs. `fontSize >= 5`) — it forbids `5` as a possible input, which `X` explicitly allowed. Strengthening a precondition violates the principle, regardless of what happens to the postcondition.

Writing pre/postconditions

```
public boolean q2(String[] strArray, int firstIndex, int secondIndex) {
    return strArray[firstIndex] == strArray[secondIndex];
}
```

Write a Javadoc comment for `q2`.

> **Working**
>
> ```
> /**
>  * Determines if two indicated elements of an array of Strings
>  * refer to the same String object.
>  *
>  * @param strArray an array of Strings
>  * @param firstIndex index of the first element to compare
>  * @param secondIndex index of the second element to compare
>  * @return true if the indicated elements in the array refer to
>  *         the same string, false otherwise
>  */
> public boolean q2(String[] strArray, int firstIndex, int secondIndex) {
>     return strArray[firstIndex] == strArray[secondIndex];
> }
> ```

The description says *refer to the same* ***`String`*** *object* rather than "are equal": the method uses `==` on references, not `.equals`.

Add `@requires`/`@ensures` tags for all appropriate pre- and postconditions.

Working

```
/**
 * ...
 * @requires strArray != null && 0 <= firstIndex < strArray.length
 *               && 0 <= secondIndex < strArray.length
 * @ensures \result == true
 *              <==> strArray[firstIndex] == strArray[secondIndex]
 */
```

The precondition is everything the body needs in order not to throw: a non-null array and two in-range indices. The postcondition is written as an `<==>` (if and only if), not just "returns true when …", so it also pins down when the result is `false`.

Specify that `q2` does not change any elements of `strArray`.

Working

```
/**
 * ...
 * @ensures \forall int i; 0 <= i && i < strArray.length
 *              ==> \old(strArray)[i] == strArray[i]
 */
```

`\old(...)` is the value on entry to the method, so this says every slot holds the same reference afterwards as it did before.