Strings Never Change

A String is immutable, so every method that looks like it modifies one actually returns a new one -- which is why an ignored trim() does nothing, and why a loop wants a StringBuilder instead of +.

Automation Foundations · ashishautomationlabs.store

Here is the rule that explains a surprising amount of broken test code:

```
String id = "def-101";
String changed = id.toUpperCase();
System.out.println("original: " + id);
System.out.println("returned: " + changed);
id.toUpperCase();
System.out.println("ignored : " + id);
```

```
original: def-101
returned: DEF-101
ignored : def-101
```

`toUpperCase` did not change `id`. It could not. A String in Java is **immutable** -- once created, its contents can never be altered. Every method that appears to modify a String actually builds and returns a new one, leaving the original exactly as it was.

That third line is the mistake, and it is the same one from Chapter 8's Bug Hunt, now with its underlying reason. Calling `id.toUpperCase()` and ignoring the result does nothing whatsoever. There is no version of that call that edits `id`. You must assign the answer: `id = id.toUpperCase();`. Immutability is why Strings are safe to share between threads and safe as Map keys, which matters for Chapter 17. It also has a cost, and the cost shows up in loops.

StringBuilder: when you are assembling text

Every `+` on Strings creates a new object. One or two is irrelevant. Inside a loop that runs three hundred times, you are building and discarding three hundred objects to produce one line of text. `StringBuilder` is a mutable text buffer built for exactly this:

```
StringBuilder report = new StringBuilder();
for (int i = 0; i < tests.length; i++) {
    report.append(tests[i]).append(" -> ").append(results[i]);
    if (i < tests.length - 1) {
        report.append(" | ");
    }
}
System.out.println(report.toString());
```

```
LoginTest -> PASS | SearchTest -> FAIL | CartTest -> PASS
length: 57
```

`append` modifies the buffer in place and returns the builder itself, which is why the calls chain. `toString()` produces the finished String at the end.

Should you use `+`, `String.format`, or `StringBuilder`? They're three tools for three jobs, not three ways to do one job. Use `+` for joining two or three values in one place -- it's clear and fine, write the readable thing. Use `String.format` when there's a template with several values, or alignment matters. Use `StringBuilder` when you are building text across the passes of a loop. "Why is String immutable, and what is `StringBuilder` for" is one of the most reliable interview questions in this chapter, because the answer shows you understand what is happening in memory.

> *Given* `String s = " pass ";` *-- what does* `s.trim()` *return, and what is* `s` *afterwards? What does* `s.trim().toUpperCase()` *return? Why does* `s.trim();` *on its own line achieve nothing?*

>

> *Answer -- it returns* `"pass"`*, and* `s` *is unchanged, still* `" pass "`*.* `"PASS"`*, because each call operates on the result of the one before. And Strings are immutable, so the method returns a new value and the original is untouched -- an ignored return value is discarded.*

Cleaning and building text both assume you already have the whole String in hand. Real test data usually arrives as one row that needs cutting apart first -- and that's a different tool with its own trap.