The Ignored trim() and the Silent Zero

This program prints four lines. Three are wrong, and the fourth is right for the wrong reason -- none of the four are new mistakes, they're every text habit in this chapter, used carelessly once each.

Automation Foundations · ashishautomationlabs.store

This program compiles and runs, printing four lines. Read first, then run.

```
public class BugHunt14 {
    public static void main(String[] args) {
        String raw = "  DEF-101  ";
        raw.trim();
        System.out.println("Trimmed id : '" + raw + "'");
        String version = "4.2.1";
        String[] parts = version.split(".");
        System.out.println("Version parts: " + parts.length);
        String report = "";
        for (int i = 1; i <= 3; i++) {
            report = report + "row" + i + ";";
        }
        System.out.println("Report: " + report);
        String price = "1,299";
        int value = safeParse(price);
        System.out.println("Price as int: " + value);
    }
    static int safeParse(String s) {
        try {
            return Integer.parseInt(s);
        } catch (NumberFormatException e) {
            return 0;
        }
    }
}
```

```
Trimmed id : '  DEF-101  '
Version parts: 0
Report: row1;row2;row3;
Price as int: 0
```

Four issues. Three produce a wrong value; the fourth produces the right value the wrong way, and it's worth being precise about which is which.

`raw.trim();` on its own line does nothing, because Strings are immutable and the returned value was discarded. The id still carries its spaces, and the quote marks in the output are the only reason you can see it. The fix is `raw = raw.trim();`.

`version.split(".")` gave zero fields. `split` takes a regular expression, and `.` matches any character, so everything was consumed. Any code reading `parts[0]` next would fail with the array

index error from Chapter 4 -- a crash whose real cause is three lines earlier. The fix is `split("\\.")`.

Building `report` with `+` inside a loop produces exactly the right text, and this line is not wrong. It is wasteful: each pass discards the previous String and builds a new one. At three rows nobody would notice; at three thousand it is real time spent for no benefit. Use a `StringBuilder` when the loop is the thing generating the text.

`safeParse("1,299")` returned `0`. The comma made it invalid for `parseInt`, the exception was caught, and the fallback was returned in silence. A price of `1299` became `0`, and any total built from it is wrong with no clue why. This is Chapter 9's habit of swallowing an exception again: the fallback is reasonable, but it must be reported. Strip the separators before parsing, and log every value you could not use.

Checkpoint

- What does immutability mean for a String, and what must you do with the

result of `trim()`?

- When should you use a `StringBuilder` instead of `+`?
- Why does `split(".")` fail, and how do you split on a literal dot?
- Give two things `isBlank` catches that `isEmpty` does not.
- What does `%-12s` do in a format template, and why does it matter in a

report?

- Name the two checks that make text-to-number conversion safe on real

test data.

Text handling is most of the glue in a framework. `getText()` from a page returns a String that almost always needs trimming and cleaning before it means anything. Building a dynamic locator is `String.format("//div[@id='%s']", id)`, which is safer and clearer than joining with `+`. CSV test data is `split` plus a constructor from Chapter

- API responses are parsed, and status text is normalized with

`toLowerCase` before comparing. And every failure message your suite produces is formatted text -- the difference between `"assertion failed"` and `"Expected 200 but got 503"` is the difference between a defect somebody can triage and one they must reproduce first.

Common, and easy marks if you have run the code: why String is immutable (its contents can never change; every method returns a new String -- which makes Strings safe to share and to use as Map keys); the difference between `String`, `StringBuilder`, and `StringBuffer` (immutable;

mutable; mutable and synchronized for threads); why `"a.b.c".split(".")` returns nothing (`split` takes a regular expression and a dot matches any character -- escape it as `"\\."`); how you compare two Strings ignoring case (`equalsIgnoreCase`, from Chapter 6); and what `trim` returns, and whether it changes the original (a new String; the original is unchanged).

You can take any text a system hands you and turn it into something worth asserting on. You can trim, cut, and clean a scraped value into a number, split a data row into fields without falling into the regular-expression trap, and validate an id format with a short pattern. You can build report lines whose columns line up and failure messages that name the expected and actual values. And you know why an ignored `trim()` does nothing, which is a small piece of knowledge that quietly prevents a whole family of bugs.

Notice, though, where all this text has been coming from: a literal typed into the program. Your 47 passwords are still hard-coded, your environment URLs live in an enum you compiled, and the CSV row you parsed was a String in the source file. Real test data lives outside the code -- in a properties file the team edits, a spreadsheet the business owns, a JSON payload from an API contract. Getting that data in and out is what turns a script into a data-driven suite, and it is Chapter 15.