Converting Text Safely

A safe conversion checks for blank, catches the parse failure, and reports every value it couldn't use. The same lesson from a different angle: parse text into the type you actually mean, and compare that.

Automation Foundations · ashishautomationlabs.store

Real test data has blank cells and values like `"n/a"`. A parse that trusts the input will end your run, as Chapter 9 showed. Combine what you know:

```
static int toMs(String cell) {
    if (cell == null || cell.isBlank()) {
        System.out.println("  blank cell, using 0");
        return 0;
    }
    try {
        return Integer.parseInt(cell.trim());
    } catch (NumberFormatException e) {
        System.out.println("  not a number: '" + cell + "', using 0");
        return 0;
    }
}
```

```
  blank cell, using 0
  not a number: 'n/a', using 0
Total ms: 1752
```

Four ideas from four chapters in one method: the null-first check from Chapter 3, `isBlank` from this one, the `try`/`catch` from Chapter 9, and a *reported* fallback rather than a silent one. Note that it tells you about every value it could not use. A version that returned `0` in silence would repeat Chapter 9's worst habit: swallowing a failure.

Reports that line up

`String.format` builds text from a template, and it is what turns a wall of output into a readable report:

```
String line = String.format("%-12s | %5d ms | %6.2f%%", name, ms, rate);
```

```
LoginTest    |   240 ms |  97.50%
SearchTest   |  1512 ms |  88.13%
```

The placeholders are worth knowing. `%s` inserts text and `%-12s` pads it to twelve characters, left-aligned, which is what makes the columns line up. `%d` is a whole number and `%5d` right-aligns it in five characters. `%6.2f` is a decimal with two places. And `%%` prints a literal percent sign. Notice `88.125` printed as `88.13` -- formatting rounds for display without changing the value. The everyday

use is assertion messages: `String.format("Expected %s but got %s", expected, actual)` is more readable than joining with `+`, especially once there are three or four values.

Assert on the value, not on its presentation

A test asserts on a price by comparing the scraped text directly against `"Rs 1,299.00"`. It matches what the page shows, and if the price changes, the test should fail -- that's the point. But it will also fail when someone adds a space, or the site switches to `"1,299.00 Rs"`, or the currency label gets localized. None of those are price defects. The fix is not to clean the string and compare text anyway -- it's to clean it, convert it to a number, and compare numbers. Then formatting changes can't break the test, and a genuine price change still does. Keep the raw text in the failure message so a human can see what arrived. Assert on the value, not on its presentation. Parse the text into the type you actually mean -- a number, a date, an enum -- and compare that. Text comparison is right only when the text itself is the thing under test.

Cleaning, splitting, matching, converting, formatting -- five tools, one job: turning whatever a system hands you into something worth asserting on. The closing bug hunt shows what happens when any one of them gets used carelessly.