The Empty List That Reported Green

This program prints three calm lines, and the last one is the most dangerous output in this book: no data, suite passes.

Automation Foundations · ashishautomationlabs.store

This program compiles and runs, and prints three calm lines. Read first, then run.

```
public class BugHunt15 {
    public static void main(String[] args) {
        List<String> rows = load("testdata/logins.csv");
        System.out.println("Rows loaded : " + rows.size());
        List<String> missing = load("testdata/does-not-exist.csv");
        System.out.println("Rows loaded : " + missing.size());
        System.out.println("Verdict     : " + (missing.isEmpty() ? "no data, suite
passes" : "data found"));
    }
    static List<String> load(String path) {
        try {
            return Files.readAllLines(Path.of(path));
        } catch (IOException e) {
            return List.of();
        }
    }
}
```

```
Rows loaded : 3
Rows loaded : 0
Verdict     : no data, suite passes
```

Read that last line again. It is the most dangerous output in this book.

The catch block returns an empty list and says nothing. The file was missing, and the program reported that fact as `0`. This is Chapter 9's habit of swallowing a failure, now applied to setup -- and it converts a broken configuration into what looks like a successful run with no test data.

The verdict logic then draws the natural conclusion: no data, nothing failed, so the suite passes. Every data-driven test built on this loader would run zero times and report green. A pipeline that checks nothing and shows a tick is worse than one that shows a cross, because a cross gets investigated.

The header row is counted as data. The first file loaded `3` lines for `2` test rows, because `readAllLines` does not know about headers. Any code treating `rows.size()` as a test count is already off by one, and the first row it processes is the column names.

The path is relative and the failure message does not say where the program looked. When this fails in a pipeline with a different working directory, the message `0` is all the evidence anyone

gets.

The fix for the first two is one decision: do not catch that exception here at all, or catch it and throw a clear setup failure naming the file and the absolute directory. Let the run stop. A suite that cannot find its data has not passed.

Checkpoint

- What does `try`-with-resources guarantee, and which classes can it be used with?
- What does `getProperty` return for a missing key, and what is the safer two-argument form?
- In the CSV loop, why is `readLine()` called once before the `while` begins?
- Why use `split(",", -1)` when parsing a data row?
- Give the rule for what a suite should do when its test data file is missing, and say why.
- Where should test data files live in a project, and what problem does that location solve?

This is the plumbing under every data-driven framework. TestNG's `@DataProvider` and JUnit's parameterized tests both need a method that returns rows -- and that method is exactly the CSV or Excel loader in this chapter. Configuration lives in `.properties` per environment, loaded once in `BaseTest` and read through a config class rather than scattered `getProperty` calls. API test payloads are JSON files mapped to request objects. Results are written out as CSV or JSON for reporting. And `src/test/resources` is where all of it lives, so the data ships with the build.

Practical interview questions with clear answers: how you do data-driven testing in your framework (external data file, a loader that turns rows into objects, and a data provider feeding one test method -- describe your loader); what `try`-with-resources is and why it's preferred to `finally` (automatic closing of anything `AutoCloseable`, on both the normal and exception paths); how you manage configuration across environments (properties files per environment, loaded once, with sensible fallbacks); what happens if your data file is missing (it must fail the run loudly -- the wrong answer is "return an empty list"); and why `src/test/resources` is the right place for test data (it is on the classpath, so it travels with the build and does not depend on the working directory).

Your test data can live outside your code. You can read and write files with automatic cleanup, load configuration from a properties file with sensible fallbacks, and turn CSV rows into typed objects that later readers can understand. You can skip headers and blank lines, keep empty trailing fields, and report a missing file with the directory you actually looked in. And you know the rule that matters more than any of the syntax: a missing data file fails the run, because a green suite that tested nothing is the worst result a framework can produce.

Look at the loops you have been writing to do this. Read every row, keep the ones that are not blank, turn each into an object, collect them into a list. Then later: walk the results, keep the failures, count them. That shape -- take a collection, filter it, transform it, collect the answer -- appears everywhere in test code. You have been writing it out longhand every time, with a loop, an `if`, and a variable to accumulate into. Modern Java can express that whole shape in a line, and the same feature is what Selenium's waits are built from. That is Chapter 16.