The Empty Catch That Still Printed PASS

This program runs cleanly and reports a healthy result. It's lying, and it's the most dangerous lie in the chapter: a run that skipped a third of its data and called it a pass.

Automation Foundations · ashishautomationlabs.store

This program runs cleanly and reports a healthy result. It is lying, and the lie is the most dangerous one in this book. Read first, then run.

```
public class BugHunt9 {
    public static void main(String[] args) {
        String[] rows = {"10", "oops", "30"};
        int total = 0;
        for (int i = 0; i < rows.length; i++) {
            try {
                total += Integer.parseInt(rows[i]);
            } catch (Exception e) {
            }
        }
        System.out.println("Total: " + total);
        System.out.println("Verdict: " + (total > 0 ? "PASS" : "FAIL"));
    }
}
```

```
Total: 40
Verdict: PASS
```

Three defects, and they compound.

The catch block is empty. The middle row failed to parse and the program said nothing at all -- no message, no count, no trace. This is called **swallowing an exception**, and it is the single worst habit in test automation, because the evidence is destroyed at the exact moment it was available.

It catches `Exception`, the broadest type. The code anticipated a parse failure, but this catch would equally swallow a null pointer, an index error, or any other bug in that block. A narrow `catch (NumberFormatException e)` would have handled the expected case and let genuine surprises surface.

The verdict is computed from a total built on incomplete data. The real total of the three rows is unknown, because one row never contributed. `40` looks like a plausible number, so nobody questions it, and the suite prints PASS. A run that skipped a third of its data reported success.

The fix is not to stop catching. It is to catch narrowly, report each skipped row with its value, count the skips, and fail the run if the count is above zero. Then the suite still survives bad data and still tells the truth. That is the whole discipline of this chapter in one sentence: handle what you can explain, and never hide what you cannot.

Checkpoint

- What are the three pieces of evidence in a stack trace, and which line do you look for first in your own code?
- Why does the placement of a `try` inside or outside a loop change the outcome?
- In what order must multiple catch blocks be written, and why?
- What does `finally` guarantee, and give the automation example from this chapter.
- State the difference between checked and unchecked exceptions in one sentence each.
- What is wrong with an empty catch block, in defect-report terms?

Reliable interview ground, all straight from this chapter: the difference between checked and unchecked exceptions (compiler forces handling of checked ones, which model conditions outside your control; unchecked ones model logic bugs); what `finally` does and whether it always runs (cleanup on both paths, whether or not an exception was thrown); why catching `Exception` broadly is a bad habit (it hides problems you did not anticipate and cannot explain); the difference between `throw` and `throws` (`throw` raises one now, `throws` declares that a method may pass one on); and how you read a stack trace (type and message on the first line, then the deepest line naming your own code).

Your code can survive contact with reality. You can catch an exception and keep a data-driven run going, place the `try` around the right unit of work, order multiple catches from specific to general, and guarantee cleanup with `finally`. You can read a stack trace and go straight to the line in your own code that caused it. You can throw your own exception with an expected-versus-actual message, and define a custom exception type that names your domain. And you know the discipline that matters more than any syntax here: catch what you can explain, report everything you skip, and never let a suite print PASS over hidden failures.

That completes your core Java toolkit -- data, decisions, loops, collections, your own classes, methods, and now failure handling. There is no longer any part of a simple Java program you cannot read and write. Which brings back the one Promise Note still open since Chapter 5. You can hold many values in a list, but you cannot yet look one up by name: give me the URL for "uat", give me the status of defect "DEF-101." And you cannot yet hold a collection that refuses duplicates. Those are the Map and the Set, they lean directly on the `equals` and `hashCode` work you did in Chapter 7, and they are Chapter 10.