try, catch, and Where You Wrap It

Wrapping the smallest unit of work in try/catch lets a data-driven run survive one bad row and keep checking the rest -- but only if the try sits inside the loop, not around it.

Automation Foundations · ashishautomationlabs.store

You handle an exception by wrapping the risky work in a `try` block and saying what to do in a `catch` block:

```
public class X2 {
    public static void main(String[] args) {
        String[] rows = {"12", "abc", "45"};
        for (int i = 0; i < rows.length; i++) {
            try {
                int value = Integer.parseInt(rows[i]);
                System.out.println("Parsed: " + value);
            } catch (NumberFormatException e) {
                System.out.println("Bad data in row " + (i + 1) + ": '" + rows[i] +
"'");
            }
        }
        System.out.println("Done");
    }
}
```

```
Parsed: 12
Bad data in row 2: 'abc'
Parsed: 45
Done
```

Same data, completely different outcome. The bad row is reported clearly, with its row number, and the run continues to check row 3 and finish. This is the shape of a data-driven test that survives dirty data, and it is one of the most useful patterns in this book.

Note carefully where the `try` sits: inside the loop, wrapping one row. That placement is a decision, not an accident. Put the `try` outside the loop instead and the first bad row would still end the loop -- you would catch the exception, but you would have abandoned every remaining row. Wrap the smallest unit of work you want to survive independently.

When the code in `try` throws, Java abandons the rest of the `try` block immediately and jumps to a matching `catch`. If nothing throws, the `catch` is skipped entirely. The `e` in `catch (NumberFormatException e)` is the exception object itself, and it carries useful evidence:

```
try {
    String value = "n/a";
    int n = Integer.parseInt(value);
    System.out.println(n);
} catch (Exception e) {
```

```
    System.out.println("type    : " + e.getClass().getSimpleName());
    System.out.println("message : " + e.getMessage());
}
```

```
type    : NumberFormatException
message : For input string: "n/a"
```

`getMessage()` gives the explanation, and the type tells you what went wrong. Both belong in whatever your suite reports, because a failure message that says only "test failed" is a defect report with no steps to reproduce.

Should you wrap your test methods' own assertions in try/catch like this? Usually not for the check itself -- the framework catches the assertion failure and reports it as a failed test, which is what you want. Reserve `try`/`catch` for setup and cleanup work where you can add useful context, such as which data file or which environment was involved.

> *Given the loop above with the `try` inside it: after a bad row is caught, does the loop continue? If the `try` were placed outside the loop instead, what would happen to the rows after the bad one? And when nothing throws, does the `catch` block run?*

>

> *Answer -- Yes, the next pass starts normally. Placed outside, the rows after the bad one would be skipped, because the exception would end the loop before jumping to the catch. And no -- when nothing throws, the catch is skipped entirely.*

One `catch` caught everything so far. Real code usually needs to tell one kind of failure from another, and the order you write multiple catches in is not a detail -- it decides which ones can ever run at all.