finally: The Cleanup That Always Runs

A finally block runs on both paths, success or failure, which is why every automation framework closes its browser there. Skip it, and one failed test leaves a process running that dooms the next forty.

Automation Foundations · ashishautomationlabs.store

Some work must happen whether or not things went wrong. Close the browser. Close the file. Release the connection. That is `finally`:

```
try {
    System.out.println("Opening the browser");
    String id = null;
    System.out.println(id.length());
} catch (NullPointerException e) {
    System.out.println("Caught: " + e.getMessage());
} finally {
    System.out.println("Closing the browser");
}
System.out.println("Suite continues");
```

```
Opening the browser
Caught: Cannot invoke "String.length()" because "<local1>" is null
Closing the browser
Suite continues
```

The `finally` block runs after the `try` and any `catch`, on both paths -- when everything worked and when it did not. This is why every automation framework closes its browser in a `finally` or its equivalent teardown. Without it, a test that fails halfway leaves a browser process running. After forty failures the machine is out of memory, and every later test fails for reasons that have nothing to do with the application.

That is the whole shape of `finally`: it does not care which path the code took to get there, only that it runs afterward, every time.

Catching your own mistakes and cleaning up after them handles half the picture. The other half is reporting a failure your code detected on purpose -- and for that, you don't wait to catch someone else's exception. You throw your own.