Reading and Writing a File

try-with-resources closes anything AutoCloseable automatically, on both the normal and exception paths -- so there's no finally block left to forget.

Automation Foundations · ashishautomationlabs.store

Your 47 invalid passwords are still typed into a Java file. So is the base URL, the timeout, and the list of usernames. Every one of them requires a developer, an edit, and a rebuild to change. The business analyst who knows the real test cases cannot touch them. The pipeline that needs to run against three environments cannot switch without recompiling. Test data belongs outside the code -- that single move is what turns a suite of scripts into a data-driven framework: the same test method, run once per row, against data that other people can maintain.

Modern Java handles files through `Path` and `Files`, and the basics are short:

```
Path file = Path.of("testdata/logins.csv");
Files.createDirectories(file.getParent());
Files.writeString(file, "username,password,expected\n"
                      + "priya,Valid#123,PASS\n"
                      + "raj,wrong,FAIL\n");
List<String> lines = Files.readAllLines(file);
```

```
Exists : true
Size   : 63 bytes
Lines  : 3
  username,password,expected
  priya,Valid#123,PASS
  raj,wrong,FAIL
```

`Path.of(...)` names a file without touching the disk. `Files` does the actual work: `exists`, `size`, `writeString`, `readAllLines`. Note `createDirectories` -- writing to a folder that does not exist fails, and creating it first is a one-line habit worth keeping. `readAllLines` loads the whole file into memory, which is right for test data and wrong for a two-gigabyte log; for anything large, read line by line instead.

Every one of these methods can throw `IOException`, the checked exception from Chapter 9. The compiler will not let you ignore it: either catch it or declare `throws`. That is the language insisting you decide what happens when the file is not there.

try-with-resources

Chapter 9 used `finally` to guarantee cleanup. For anything you open -- a file, a stream, a connection -- Java has something better:

```
try (BufferedReader reader = Files.newBufferedReader(file)) {
```

```
    String line = reader.readLine();
    // ...
}
```

The resource is declared inside the brackets of `try`, and Java closes it automatically when the block ends, whether it ended normally or by exception. No `finally`, nothing to forget.

This matters more than it looks. An unclosed file handle is not a crash; it is a slow leak. A suite that opens a data file per test and never closes it will run fine for forty tests and then start failing with errors that have nothing to do with the application. `try`-with-resources makes that impossible, so use it for everything you open. It works with any class that implements `AutoCloseable`, which includes readers, writers, streams, and -- relevant later -- a `WebDriver` in some setups.

Files and streams are one kind of external data. The other kind -- the settings that change per environment -- has its own dedicated format, and it leans on ideas from two chapters ago.