The Filter That Never Ran

This program compiles and runs, and reports zero failures when there was one -- because a filter with no terminal operation isn't a bug that crashes, it's a pipeline that silently never executes.

Automation Foundations · ashishautomationlabs.store

This program compiles and runs. Read first, then run.

```
public class BugHunt16 {
    public static void main(String[] args) {
        List<String> results = Arrays.asList("PASS", "FAIL", "PASS");
        results.stream().filter(r -> r.equals("FAIL"));
        System.out.println("Failures reported: none");
        Stream<String> s = results.stream();
        System.out.println("count once: " + s.count());
        try {
            System.out.println("count twice: " + s.count());
        } catch (IllegalStateException e) {
            System.out.println("count twice: " + e.getMessage());
        }
        Optional<String> pay = results.stream().filter(r -> r.equals("PAY")).findFirst(
);
        try {
            System.out.println("value: " + pay.get());
        } catch (Exception e) {
            System.out.println("get() on empty: " + e.getClass().getSimpleName());
        }
    }
}
```

```
Failures reported: none
count once: 3
count twice: stream has already been operated upon or closed
get() on empty: NoSuchElementException
```

Three defects, one per rule in this chapter.

`results.stream().filter(...)` on its own line did nothing at all. `filter` is an intermediate operation, so with no terminal operation the pipeline never ran. There was one `FAIL` in the data and the program cheerfully reported none. This is the same shape as Chapter 14's ignored `trim()` -- a call whose result was discarded -- and it is silent in exactly the same way.

The stream was consumed by the first `count()`, and the second call threw `IllegalStateException` with a message that says so plainly. A stream is single-use. If you need two results from the same data, either start two streams from the collection or collect once into a list and work from that.

`pay.get()` on an empty `Optional` threw `NoSuchElementException`. `Optional` protected nothing here, because `get()` was called without checking. That is `Optional` used as decoration. Use `orElse`, `ifPresent`, or check `isPresent` first -- otherwise you have swapped a `NullPointerException` for a differently-named one.

Checkpoint

- What is a functional interface, and what does a lambda supply?
- Name the four common functional interfaces and the method each one

declares.

- What is the difference between an intermediate and a terminal

operation, and what happens with no terminal one?

- Why does the second `count()` on the same stream fail?
- Give the three safe ways to read an `Optional`, and the one to avoid.
- Name two situations where a plain loop is the better choice.

Increasingly common for automation roles: what a functional interface is (an interface with exactly one abstract method, which a lambda can implement); the common ones and their methods (`Predicate`/`test`, `Function`/`apply`, `Supplier`/`get`, `Consumer`/`accept`); the difference between intermediate and terminal operations (intermediate ones describe a step and return a stream; terminal ones produce a result and trigger execution); whether a stream can be reused (no -- a second terminal operation throws `IllegalStateException`); what problem `Optional` solves (it makes a possibly-absent value explicit rather than returning `null`); and whether `map` changes the source collection (no -- streams do not modify the source).

You can pass behavior to a method. You can write a lambda, recognize which functional interface it fits, and use method references where they read more clearly. You can build stream pipelines that filter, transform, count, and collect, and you know that nothing runs until a terminal operation asks for a result. You can handle a possibly-absent value with `Optional` instead of a null check, and you can say why the shortest version is not always the right one.

Most importantly, you built a `waitUntil` that takes a condition as a value and re-evaluates it. That is not an analogy for an explicit wait -- it is the mechanism, and you now understand the most-used feature in Selenium from the inside.

Your framework is nearly complete. It is organized, it reads its data from outside, it bends at the joints, and it can express its logic concisely. There is one thing left, and it is the one that breaks suites in production rather than in development. Right now everything you have written runs one test at a time. Then your team turns on parallel execution to get the nightly run under an hour,

and a suite that passed for months starts failing in ways nobody can reproduce -- different tests each run, on a machine nobody changed. The cause is a Core Java idea, not a Selenium one, and the fix is a single class. That is Chapter 17.