Streams: A Pipeline Over a Collection

A stream pipeline with no terminal operation does nothing at all -- no error, no output. filter and map only describe a step; nothing runs until collect, count, or another terminal operation asks for a result.

Automation Foundations · ashishautomationlabs.store

A stream is not a collection. It is a pipeline you run over one. You start it with `.stream()`, chain operations, and finish with an operation that produces a result.

```
List<String> tests = Arrays.asList("LoginTest", "CartTest", "SearchTest", "PayTest");
long count = tests.stream().filter(t -> t.length() > 8).count();
List<String> upper = tests.stream()
                          .map(t -> t.toUpperCase())
                          .collect(Collectors.toList());
boolean anyPay = tests.stream().anyMatch(t -> t.startsWith("Pay"));
String joined = tests.stream()
                     .filter(t -> !t.equals("PayTest"))
                     .collect(Collectors.joining(", "));

longer than 8 : 2
mapped        : [LOGINTEST, CARTTEST, SEARCHTEST, PAYTEST]
any Pay test? : true
joined        : LoginTest, CartTest, SearchTest
```

The vital distinction is between two kinds of operation. **Intermediate** operations such as `filter` and `map` return another stream and do nothing on their own -- they only describe a step. **Terminal** operations such as `collect`, `count`, `anyMatch`, and `forEach` produce a result and cause the whole pipeline to run.

That means a pipeline with no terminal operation does nothing at all. No error, no output -- the code simply never executes. It catches people who expect `filter` to act like a loop.

Two more rules worth fixing now. A stream can be used once: after a terminal operation it is finished, and using it again throws. And a stream does not modify the source collection -- `map` produces new values and leaves the original list exactly as it was, which should feel familiar after Chapter 14's immutable Strings.

Method references

When a lambda does nothing but call an existing method, you can name that method directly:

```
List<String> cleaned = raw.stream()
                          .map(String::trim)
                          .map(String::toUpperCase)
                          .collect(Collectors.toList());
```

```
raw.stream().map(String::trim).forEach(System.out::println);

method refs: [LOGIN, CART, PAY]
login
cart
pay
```

`String::trim` means exactly the same as `s -> s.trim()`, and `System.out::println` means `s -> System.out.println(s)`. Use a method reference when the lambda would add nothing but noise, and a full lambda when there is real work in the body. This is a readability choice, not a performance one.

What's the difference between `map` and `filter`, precisely? `filter` keeps or discards items, so the count can shrink and the type stays the same. `map` transforms each item, so the count stays the same and the type can change. Remember it as: filter narrows, map converts.

A stream can express, correctly, the exact same job as a loop you already know. That doesn't mean it's always the better choice -- and one particular kind of "missing result" needs a tool of its own before this chapter is done.