Where You Have Already Met This

A hand-written waitUntil that takes a condition as a lambda and re-evaluates it until true isn't an analogy for a Selenium explicit wait -- it's the actual mechanism, built from scratch.

Automation Foundations · ashishautomationlabs.store

Here is the payoff that connects this chapter to your day job. Write your own functional interface -- one abstract method -- and you can drive it with a lambda:

```
interface Condition {
    boolean isMet();
}
static boolean waitUntil(Condition condition, int maxTries) {
    for (int i = 1; i <= maxTries; i++) {
        if (condition.isMet()) {
            System.out.println("  met on try " + i);
            return true;
        }
        System.out.println("  not yet, try " + i);
    }
    return false;
}

boolean ok = waitUntil(() -> pageTitle().equals("ShopCart"), 5);
```

```
waiting for the page title...
  not yet, try 1
  not yet, try 2
  met on try 3
condition met? true
attempts made: 3
```

Look carefully at what was passed to `waitUntil`. Not a value -- a question, which the method asked repeatedly until it became true. The page title was `"Loading..."` twice and `"ShopCart"` on the third check.

That is an explicit wait. Selenium's version has more machinery around timeouts and polling intervals, and its conditions come from a class of ready-made ones, but the mechanism is precisely this: a method that takes a condition as a value and re-evaluates it until it holds. When you write a custom wait condition as a lambda in a real framework, you are doing what you just did here.

> *Using the four functional interfaces from the previous article: which one would you pass to* `filter`*, and what does it return? Which one takes a value and gives back nothing? What makes an interface a functional interface?*

>

> *Answer -- a* `Predicate`*, which returns true or false. A* `Consumer`*. And it has exactly one abstract method, which is what the lambda supplies.*

A single lambda answering one question is useful. The next step is chaining several of them together over a whole collection at once -- which is what a stream actually is.