A Lambda Is Behavior You Can Pass Around

Until now, everything you passed to a method was data. A lambda lets you pass behavior -- a small block of code -- as an argument, and its type is always a functional interface.

Automation Foundations · ashishautomationlabs.store

You have written this shape a dozen times since Chapter 5:

```
List<String> failures = new ArrayList<>();
for (String r : results) {
    if (r.equals("FAIL")) {
        failures.add(r);
    }
}
```

Take a collection. Keep the items that match a condition. Collect them into a new list. Four lines, an empty list to prepare, a loop, an `if`, and a variable to accumulate into -- every time. Modern Java expresses that same shape in one line:

```
List<String> failures = results.stream()
                               .filter(r -> r.equals("FAIL"))
                               .collect(Collectors.toList());

loop   : [FAIL, FAIL]
stream : [FAIL, FAIL]
```

Identical results. This chapter is about the features behind that second version, and there is a reason it matters beyond tidier code: every explicit wait you will ever write in Selenium is built from this same feature. A wait takes a condition -- a piece of behavior handed to a method as a value -- and that is exactly what the arrow above is.

The syntax

Until now, everything you passed to a method was data: a number, a String, an object. A **lambda** lets you pass behavior -- a small block of code -- as an argument. The syntax is short: parameters on the left, an arrow, then the body:

```
r -> r.equals("FAIL")
```

Read that as: given `r`, produce whether `r` equals `"FAIL"`. No method name, no return type, no `public` -- Java works those out from the context. When the body is a single expression, its value is returned automatically. When you need several statements, use braces and an explicit `return`.

Functional interfaces: the type a lambda fits into

A lambda has to have a type, and its type is an interface with exactly one abstract method. Such an interface is called a **functional interface**, and the lambda supplies the body of that single method. That's also the full answer to "can I use a lambda anywhere?" -- only where a functional interface is expected. That's why `Runnable`, `Comparator`, `Predicate`, and Selenium's condition types all accept lambdas: each has a single method for the lambda to supply.

Java provides the common ones, and four cover most of what a tester needs:

```
Predicate<String> isFail = r -> r.equals("FAIL");
Function<String, Integer> length = s -> s.length();
Supplier<String> runId = () -> "RUN-8891";
Consumer<String> log = s -> System.out.println("Consumer  : [LOG] " + s);

Predicate : true / false
Function  : 9
Supplier  : RUN-8891
Consumer  : [LOG] suite finished
```

Learn them by what they do. A `Predicate` takes a value and answers true or false -- a condition, which is what `filter` needs. A `Function` takes a value and returns a different one, usually a different type -- a transformation, which is what `map` needs. A `Supplier` takes nothing and produces a value, useful for something computed only when needed. A `Consumer` takes a value and returns nothing, for actions like logging. Note the method names, which are easy to mix up: `test` for `Predicate`, `apply` for `Function`, `get` for `Supplier`, `accept` for `Consumer`.

These four interfaces are useful in the abstract. The next example is where this chapter stops being abstract altogether -- a functional interface you write yourself, doing a job you already recognize.