Naming a Job So You Can Reuse It

You've typed public static void main(String[] args) since page one on faith. A method gives a name to one job, written once -- the same idea as a reusable test step -- and this is where that line starts to make sense.

Automation Foundations · ashishautomationlabs.store

You have written methods already -- `isPassed()`, `getUrl()`, `equals`, `toString` -- without stopping to study what a method really is. You have also used the idea outside Java, every time you wrote a test case: you did not spell out "open browser, navigate to URL, enter username, enter password, click submit" in all forty test cases. You wrote one reusable step called "Login as a valid user" and referred to it. A method is exactly that: a named job, written once, used wherever you need it.

Here is the smallest useful method -- it gives a name to one job, printing a step in a readable format:

```
public class M1 {
    static void logStep(String step) {
        System.out.println("[STEP] " + step);
    }
    public static void main(String[] args) {
        logStep("Open the login page");
        logStep("Enter valid credentials");
        logStep("Click submit");
    }
}
```

```
[STEP] Open the login page
[STEP] Enter valid credentials
[STEP] Click submit
```

One method, called three times. Change the format once -- say, to add a timestamp -- and all three lines change together. That is the whole argument for methods, and it is the same argument you would make for a reusable test step.

The four parts of a method signature

Every method you meet has these four parts, in this order: `static void logStep(String step)`.

- `static` -- a modifier: who can call this, and how.
- `void` -- the return type: what it hands back. `void` means nothing at all.
- `logStep` -- the name. Use a verb, because a method does something.
- `(String step)` -- the parameter list: the information the method needs to do its

job.

The word `step` inside the parentheses is a **parameter** -- a variable that exists only inside this method, filled in fresh each time it is called. What you pass at the call, `"Open the login page"`, is an **argument**. Parameter is the empty slot; argument is what you drop into it.

When a block earns its own name

A short method is easy to reuse; a long one is not. A good size is one job at one level of detail -- describable in one sentence without an "and." If the sentence needs an "and," it is probably two methods. That is also why a method and a function are, in Java, the same idea under one name: other languages call a standalone block of work a function, but every block in Java lives inside a class, so Java always calls it a method. And methods calling methods is not a special trick -- it is how test frameworks are built at all: `main` calls `logStep`, a verdict method gets called with the result of a check method, and each stays small enough to trust on sight.

That is also the whole case against the 90-line test method that opens a browser, logs in, searches, adds to cart, checks out, and asserts a total in one unbroken block. It reads top to bottom like the test case document, which feels like an advantage -- until it fails at line 60 and the report says only that one test failed, with no hint which step. Split it into `login`, `search`, `addToCart`, `checkout`, and a failure names the step for you. The next test that also needs to log in gets it for free, instead of forty copied lines; the day the login page changes, you edit one method instead of hunting through a dozen tests. A block earns its own method when you can say its name out loud, or when a second caller wants it -- long methods aren't wrong for being long, but for being unreusable and unable to say which part failed.

Every method so far in this chapter has printed something and handed back nothing. The next question is what a method looks like when its job is to answer one -- and that is where `return` comes in.