Why Loops Exist

Forty-seven bad passwords, one tired glance at attempt thirty-four -- that is why automation exists. A for loop is how you tell Java to do the check again without copying the steps.

Automation Foundations · ashishautomationlabs.store

Here is a task from your real working life. A defect was raised: the login page accepts a password it should reject. To prove it is fixed, you must try forty-seven known-bad passwords and confirm every one is refused.

Forty-seven times: type the password, click login, check the error, clear the field. If you have done this, you know exactly how it feels around attempt thirty -- the attention slips, and attempt thirty-four gets a tired glance instead of a real check. That is not a personal failing. It is what repetition does to human attention, and it is the single strongest reason automation exists.

A machine does not get tired at attempt thirty-four. It checks the forty-seventh password with exactly the same care as the first. But to use that, you need a way to tell Java "do this again, and again, until the work is done" -- without writing the instruction forty-seven times. That instruction is called a **loop**, and it is the moment your code stops being a longer test case and starts being automation.

The problem, seen plainly

Without a loop, checking three test data rows looks like this:

```
System.out.println("Processing row-A");
System.out.println("Processing row-B");
System.out.println("Processing row-C");
```

Three rows, three lines -- annoying but survivable. Now make it three hundred rows, or forty-seven passwords, or every product on a search results page. The copy-paste approach does not just get long; it gets dangerous. Change the message once and you must change it in every copy, and the day you miss one is the day your suite reports two different things for the same check. Repetition is not only tiring. It is a defect source.

A loop replaces all of that with one instruction and a count.

The for loop: when you know how many times

The `for` loop is the workhorse whenever you can answer "how many times?" -- five retries, three hundred rows, every item in a list:

```
public class L1 {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Running test case #" + i);
        }
    }
}
```

```
Running test case #1
Running test case #2
Running test case #3
Running test case #4
Running test case #5
```

Five lines from four lines of code. The magic is entirely inside the round brackets, which hold three parts separated by semicolons:

- `int i = 1` -- the **setup**. Runs once, before anything else. Creates a counter named `i` starting at 1. The name `i` is a decades-old convention for "index."
- `i <= 5` -- the **condition**. Checked before every pass. While it is true, the body runs. The moment it is false, the loop stops. Same boolean judgment from Chapter 3, new job.
- `i++` -- the **update**. Runs after every pass. Shorthand for "add one to `i`."

Trace it by hand: `i` starts at 1 (true, print #1), becomes 2 ... through 5 (true, print #5), then becomes 6 -- and `6 <= 5` is false, so the loop stops. That habit of tracing, one row per pass, is how you will debug a loop that misbehaves.

> *Without running anything: how many times does* `for (int i = 0; i < 4; i++)` *run? What is* `i` *the moment the loop stops?*

>

> *Answers: four times (*`i` *= 0, 1, 2, 3); then* `i` *is 4, because* `4 < 4` *is false.*

You can replace copy-paste with a counted `for`. The next lesson puts real test data in the loop -- passwords and browsers -- and shows the cleaner "for each" form when you do not need the index.