Nested Loops and the Silent Average Bug

A loop inside a loop is a test matrix: browsers across environments. Then a hunt with two believable numbers, three silent defects, and zero crashes.

Automation Foundations · ashishautomationlabs.store

Put one loop inside another and you get every combination of two sets -- which is precisely a **test matrix**. Two browsers across two environments is four runs:

```
public class L9 {
    public static void main(String[] args) {
        String[] browsers = {"chrome", "firefox"};
        String[] envs = {"uat", "prod"};
        int combo = 0;
        for (String browser : browsers) {
            for (String env : envs) {
                combo++;
                System.out.println(combo + ". " + browser + " on " + env);
            }
        }
        System.out.println("Total combinations: " + combo);
    }
}
```

```
1. chrome on uat
2. chrome on prod
3. firefox on uat
4. firefox on prod
Total combinations: 4
```

The inner loop runs completely for each single pass of the outer loop: chrome pairs with uat then prod, and only then does the outer loop move to firefox. That is the shape of every cross-browser, multi-environment matrix you will ever build.

One warning: the run count **multiplies**. Two × two is four; ten × ten is a hundred; add a third loop and the numbers grow fast. Nested loops are powerful -- and that same power is why an accidental extra layer can turn a fast suite into one that never seems to finish.

Compact counters: i++ and +=

`i++` adds one to `i`. `total += responseTimes[i]` adds a value to `total` and stores the result back -- shorthand for `total = total + responseTimes[i]`. Both are ordinary assignments, written short because you write them so often.

> *Predict:* `for (int i = 5; i > 0; i--) { System.out.print(i + " "); }`

>

> *Answer:* `5 4 3 2 1` *-- a loop can climb or descend; the update decides the direction.*

Bug Hunt #4

This program compiles, runs, and prints two clean-looking results. Both are wrong. Three silent defects. Read first, then run.

```
public class BugHunt4 {
    public static void main(String[] args) {
        int[] responseTimes = {120, 95, 210, 88};
        int total = 0;
        for (int i = 1; i < responseTimes.length; i++) {
            total += responseTimes[i];
        }
        int average = total / responseTimes.length;
        System.out.println("Average response time: " + average + " ms");

        int slowCount = 0;
        for (int i = 0; i < responseTimes.length; i++) {
            if (responseTimes[i] > 200);
            slowCount++;
        }
        System.out.println("Slow responses (>200ms): " + slowCount);
    }
}
```

```
Average response time: 98 ms
Slow responses (>200ms): 4
```

Both numbers look believable, which is what makes them dangerous:

- The first loop starts at `i = 1`, so it silently skips `responseTimes[0]` (120). Quiet off-by-one -- no crash, just a wrong total.
- `total / responseTimes.length` divides by 4, but the buggy loop only added 3 values -- and both are `int`, so this is integer division from Chapter 2. The true average of all four is 128 ms; the program confidently prints 98.
- Look at `if (responseTimes[i] > 200);` -- the semicolon is an empty `if` body. The `slowCount++` below is not inside the `if` at all; it runs every pass. So `slowCount` counts all four responses instead of the one that is actually slow. The indentation says one thing; the semicolon

does another, and the compiler believes the semicolon.

Three defects, two confident wrong numbers, zero errors. A loop that runs is not a loop that is correct. The defense is to **trace it**, one pass and one row.

Checkpoint

- Name the three parts inside a `for (...)` header, and say when each one runs.
- When do you choose `while` over `for`? Give a test-automation example of each.
- What single thing must every `while` loop contain, and what happens if it is missing?
- What is an off-by-one error? Give the two most common causes in a `for` condition.
- Explain `break` vs `continue` in one sentence each, with a testing use for each.
- A nested loop runs an inner loop of 3 inside an outer loop of 4. How many times does

the innermost line run, and why does the run count deserve caution?

You can automate repetition -- which is the reason you picked up this path. You can count with `for`, wait with `while`, poll with `do-while`, fail-fast with `break`, skip with `continue`, and build a matrix with nested loops. Most of all, you can trace a loop by hand before you run it.

But notice what you have been leaning on: the **array**. You held passwords, browsers, and response times in `String[]` and `int[]`, looked them up with `[i]`, and counted them with `.length` -- on trust. Real test data rarely stays a fixed row of boxes. It arrives, it grows, it must be searched and filtered. That full story is Chapter 5: arrays and lists.