Off-by-One, Infinite Loops, break and continue

i <= length reaches a box that is not there. A while with no update never stops. break leaves the loop; continue skips one pass -- fail-fast versus skip-row.

Automation Foundations · ashishautomationlabs.store

Trap: the off-by-one

This is the most common loop bug in the world. Miscounting the boundary by one usually comes from a `<` where you needed `<=`, or the reverse:

```
public class L6 {
    public static void main(String[] args) {
        String[] rows = {"row-A", "row-B", "row-C"};
        for (int i = 0; i <= rows.length; i++) {
            System.out.println("Processing " + rows[i]);
        }
    }
}
```

There are three rows, at positions 0, 1, and 2. But `.length` is 3, and `i <= rows.length` lets `i` reach 3 -- a position that does not exist:

```
Processing row-A
Processing row-B
Processing row-C
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of
bounds for length 3
```

The three real rows processed correctly, then the loop reached for `rows[3]`. Read the exception like Chapter 2 taught you: index 3, out of bounds, length 3. The fix is a single character: `i < rows.length`, not `i <= rows.length`.

This off-by-one is loud -- it crashes, so you find it. Its quieter cousin is worse: a loop that starts at 1 instead of 0, silently skipping the first item and reporting a result that looks reasonable. Tracing by hand is the defense.

Trap: the infinite loop

A loop stops only when its condition becomes false. If nothing inside moves the condition toward false, the loop runs forever and your test run hangs:

```
int attempts = 0;
while (attempts < 3) {
```

```
        System.out.println("Retrying...");
        // the line that should say attempts++ is missing
    }
```

`attempts` starts at 0 and never changes, so `attempts < 3` is true forever. There is no output block to show you, because there is no end to the output -- which is exactly the problem.

Rule: every loop must contain something that moves its condition toward stopping. For a counted `for`, the `i++` does it automatically. For a `while`, you are responsible. A safety limit like `attempts < maxAttempts` is how professionals make sure a wait cannot become a hang.

break and continue

`break` stops the loop immediately and moves on to whatever comes after it. Natural use: fail-fast -- the moment a check fails, stop the run.

```
public class L7 {
    public static void main(String[] args) {
        int[] responseCodes = {200, 200, 500, 200};
        for (int i = 0; i < responseCodes.length; i++) {
            System.out.println("Check " + (i + 1) + ": " + responseCodes[i]);
            if (responseCodes[i] != 200) {
                System.out.println("Failure found. Stopping the run.");
                break;
            }
        }
    }
}
```

```
Check 1: 200
Check 2: 200
Check 3: 500
Failure found. Stopping the run.
```

The fourth code was never checked.

`continue` skips the rest of the current pass and jumps to the next one. Natural use: skipping data you do not want to process:

```
public class L8 {
    public static void main(String[] args) {
        String[] rows = {"valid", "", "valid", "skip", "valid"};
        int processed = 0;
        for (int i = 0; i < rows.length; i++) {
            if (rows[i].isEmpty() || rows[i].equals("skip")) {
                System.out.println("Skipping row " + (i + 1));
                continue;
            }
            processed++;
            System.out.println("Processing row " + (i + 1));
        }
```

```
        System.out.println("Processed " + processed + " of " + rows.length + " rows");
    }
}
```

```
Processing row 1
Skipping row 2
Processing row 3
Skipping row 4
Processing row 5
Processed 3 of 5 rows
```

> `break` *leaves the loop;* `continue` *leaves only the current pass. Reach for* `break` *when the whole job is done or clearly failed (or when one failure makes the rest meaningless). Reach for* `continue` *when this one item is not worth processing but the rest still are. Independent rows often want the full defect list, not a trickle -- that is why assertion libraries offer both hard and soft assertions.*

You can spot the two classic loop traps and control a run with fail-fast or skip-row. The next lesson nests loops into a browser × environment matrix, then closes with a hunt: three silent defects that still print confident numbers.