The Discarded Return and the Reset That Never Happens

One program, two clean-looking lines, both silently wrong: a cleaned string that was never saved, and a reset that can't reset a primitive its caller passed in.

Automation Foundations · ashishautomationlabs.store

This program compiles and runs, and prints two lines that are both wrong. Both mistakes come from this chapter, and both are silent. Read first, then run.

```
public class BugHunt8 {
    static String clean(String id) {
        return id.trim().toUpperCase();
    }
    static void resetTotal(int total) {
        total = 0;
    }
    public static void main(String[] args) {
        String rawId = "  def-101  ";
        clean(rawId);
        System.out.println("Cleaned id: '" + rawId + "'");
        int total = 99;
        resetTotal(total);
        System.out.println("Total after reset: " + total);
    }
}
```

```
Cleaned id: '  def-101  '
Total after reset: 99
```

Neither line did what its name promised.

`clean(rawId)` was called and its answer was thrown away. The method did the work correctly and returned the cleaned text, but nothing caught it. A returned value that is not assigned or used is simply discarded. Worse, `rawId` could never have changed anyway -- strings cannot be modified in place, so every `String` method hands back a new value rather than editing the old one. The fix is `rawId = clean(rawId);`. The giveaway in real code is a line that calls a method and ignores its result while expecting a change.

`resetTotal(total)` cannot work, for the reason the previous article covered: the method received a copy of the number `99` and set its copy to zero. The variable in `main` was never touched. A `void` method taking a primitive cannot change the caller's variable -- it should have been `static int resetTotal() { return 0; }`, called as `total = resetTotal();`.

Two methods that look like they do something, both quietly doing nothing. This is the method-shaped version of the lesson every Bug Hunt has taught: code that runs is not code that works.

Checkpoint

- Name the four parts of a method signature, in order.
- What is the difference between a parameter and an argument?
- What two things does `return` do?
- A method takes an `int` and changes it. A method takes a `List` and adds to it.

  Which change is visible to the caller, and why?

- What does `static` mean, and why must `main` be `static`?
- Decode `public static void main(String[] args)`, one word at a time.

You can break a job into named pieces. You can write methods that take parameters and return values, use early returns to handle a special case, choose between static utilities and instance methods, and overload a name where it genuinely helps. You know what a method can and cannot change in the caller -- the difference between a helper that works and one that quietly does nothing. And you can decode `public static void main(String[] args)` word by word; the line you copied on faith in Chapter 1 is now fully yours.

Look at what your toolkit holds now: variables and objects to store data, conditions to judge it, loops to repeat, lists to collect it, classes to model it, and methods to organize the work. That is a complete set of core Java tools. But every program you have written has assumed nothing goes wrong -- the file is always there, the number always parses, the element is always on the page. Real test runs are the opposite: they meet missing files, timeouts, and unexpected nulls constantly, and how your code responds decides whether your suite reports a real defect or just falls over. Handling those moments on purpose is the next skill. That's Chapter 9.