The Hidden Crash and the Wrong Verdict

One program, two confident verdicts, both wrong: a false failure from ==, and a false denial from a case-sensitive equals nobody meant to be exact.

Automation Foundations · ashishautomationlabs.store

This program compiles, runs, and prints two clear verdicts. Both are wrong, and both come straight from this chapter. Read first, then run.

```
public class BugHunt6 {
    public static void main(String[] args) {
        String expectedStatus = "ACTIVE";
        String prefix = "ACT";
        String actualStatus = prefix + "IVE";
        if (actualStatus == expectedStatus) {
            System.out.println("Status check passed");
        } else {
            System.out.println("Status check FAILED");
        }
        String role = "Admin";
        if (role.equals("admin")) {
            System.out.println("Admin access granted");
        } else {
            System.out.println("Admin access denied");
        }
    }
}
```

```
Status check FAILED
Admin access denied
```

Two confident verdicts, both false:

1. `actualStatus == expectedStatus` **uses** `==` **on text.** `actualStatus` was joined together at run time, so it's a new object, not the pooled literal `expectedStatus`. The arrows differ, `==` is `false`, and the check reports FAILED -- even though the status text is exactly `"ACTIVE"`. This is the false failure: a correct system, marked broken by a test that asked the wrong question. The fix is `expectedStatus.equals(actualStatus)`.

2. `role.equals("admin")` **compares case-exactly, and the role is** `"Admin"` **with a capital A.** So the honest answer is `false`, and access is denied to a genuine admin. Whether this is a defect depends on intent: if role names are meant to be case-insensitive, this must be `equalsIgnoreCase`; if they're meant to be exact, then the data or the expectation is wrong. Either way, the mismatch is invisible until you know to look for it.

Two verdicts, both wrong, zero errors -- and both are pure equality bugs. This is why the chapter exists: the most dangerous thing text can do in a test is look right while being compared wrong.

Checkpoint

- What does `==` compare for a primitive, and what does it compare for an object?
- In one sentence, what does a `String` variable actually hold?
- Why can `String a = "hi"; String b = "hi";` make `a == b` true, and why is that misleading?
- Give the one rule for comparing text in Java, and the one exception where `==` on objects is meaningful.
- Rewrite `input.equals("YES")` so it cannot throw a `NullPointerException`, and explain why yours is safe.
- When would you choose `.equalsIgnoreCase` over `.equals`, and when must you not?

You can compare correctly now -- which, for a tester, is close to the whole job. You know that `==` compares arrows and `.equals` compares contents, so you use `.equals` for every text comparison and reserve `==` for numbers, booleans, and the rare, intentional identity check. You understand the String pool well enough to explain why `==` sometimes lies, and to never be fooled by a passing `==` on two literals again. You put the known value first so a `null` can never crash your comparison, and you reach for `.equalsIgnoreCase` when case shouldn't matter.

There's a natural next question sitting under all of this. A `String` is an object with its own `.equals`, its own `.length`, its own `.startsWith` -- a bundle of data and the operations that belong to it. Where do those operations come from, and how would you build a thing like that for your own test data: a `TestResult` that knows whether it passed, a `Defect` that knows its own severity? That's the move from using objects to creating them -- and it's Chapter 7.