new String and the Real Trap

Almost every string your tests actually judge is built at runtime -- read from a file, returned by an API -- and none of those come from the pool that made == look safe.

Automation Foundations · ashishautomationlabs.store

Change one thing from the last article -- force a genuinely separate object with `new` -- and the illusion breaks:

```
public class Eq2 {
    public static void main(String[] args) {
        String a = "ACTIVE";
        String b = new String("ACTIVE");
        System.out.println("a == b      : " + (a == b));
        System.out.println("a.equals(b) : " + a.equals(b));
    }
}
```

```
a == b      : false
a.equals(b) : true
```

Identical text. `==` says `false`. The word `new` orders Java to build a brand-new object, separate from the pooled one. So the two arrows point at different objects, and `==`, which compares arrows, correctly reports that they differ. Meanwhile `.equals` still says `true`, because it compares the text.

It's the opposite of rare

You might think `new String` is rare in real code, so the trap is academic. It's the opposite of rare. Almost every string your tests actually judge is built at run time: read from a file, returned by an API, pulled from a database, or joined together from parts. None of those come from the pool. Watch a run-time join do the same thing:

```
public class Eq3 {
    public static void main(String[] args) {
        String expected = "UAT-2";
        String prefix = "UAT-";
        String actual = prefix + "2";
        System.out.println("Same text?  : " + actual.equals(expected));
        System.out.println("Same object?: " + (actual == expected));
    }
}
```

```
Same text?  : true
Same object?: false
```

`actual` was joined together while the program ran, so it's a new object, not the pooled literal `expected`. Same text, different object, and `==` reports `false`. This is the real shape of the bug: your expected value is a literal from the pool, your actual value came from the running system, and `==` between them is false no matter how perfectly the text matches. The test fails on a correct result -- or, swap the logic, passes on a wrong one.

The rule, and it has no exceptions for text

Always compare text with `.equals`, never with `==`. `.equals` looks inside both objects and compares the actual characters, which is the question you almost always mean. `==` on text asks about object identity, which you almost never mean, and which produces answers that depend on invisible details like the pool. There is a narrow, expert use of `==` for checking object identity on purpose, but for comparing values -- which is the entire job of a test -- the answer is `.equals`, every time.

> *Predict, then run: `String x = "PASS"; String y = new String("PASS");*

print(x == y); print(x.equals(y));` >

> *Answer -- `false` then `true`. `new` makes a separate object, so the arrows differ and `==` is `false`. The text is identical, so `.equals` is `true`. That's the trap in three lines.*

This is not theory -- it's the single most common cause of tests that lie. Every assertion comparing text, `assertEquals(expected, actual)` in TestNG or JUnit, uses `.equals` inside, exactly because `==` would be wrong. When you compare a value from the page against an expected value, they are two different objects with, you hope, the same text -- so `.equals` is the only correct tool. The pool is why a junior engineer's `==` check passes on their laptop and fails in the pipeline against real data.