What == Really Compares

Chapter 3's == trap and Chapter 5's case-sensitive miss were both warnings. Here's the full picture: == asks whether two variables point at the same object.

Automation Foundations · ashishautomationlabs.store

Two moments in this book were left unexplained on purpose, and both were warnings. In Chapter 3, a test compared a status with `status == "ACTIVE"`, the suite passed, and you were told it passed by luck. In Chapter 5, a list search for `"logintest"` returned `false` when the list clearly held `"LoginTest"`. Both times the promise was the same: the full picture is coming, and it's one of the most asked questions in any Java interview.

This is that chapter. The idea is what a `String` really is, where it lives, and why two pieces of text that look identical can be treated as different -- or, worse, why two values can appear equal by accident and pass a test they should fail. That last case is the most dangerous defect a test suite can carry: a false pass.

The one picture everything rests on

Return to the memory model from Chapter 2, because it explains all of this. A primitive variable holds its value directly in its box. A reference variable does not hold the object; it holds an arrow to an object that lives elsewhere, in the large memory area called the **heap**.

A `String` is a reference type. So when you write `String status = "ACTIVE";`, the box named `status` does not contain the letters A-C-T-I-V-E. It contains an arrow pointing to a `String` object that holds those letters. Keep that picture in mind and every rule below becomes obvious.

For a primitive, `==` compares the values in the two boxes: `5 == 5` is `true` because the boxes hold the same number, and you'll keep using `==` for numbers and booleans without a second thought. For an object, `==` compares the **arrows**. It asks: do these two variables point at the very same object? Not "do they hold the same text" -- "are they literally the same object in the heap." That difference is the whole chapter.

Why == sometimes works: the String pool

Run the simplest possible case and watch it pass:

```
public class Eq1 {
    public static void main(String[] args) {
        String a = "ACTIVE";
        String b = "ACTIVE";
        System.out.println("a == b      : " + (a == b));
        System.out.println("a.equals(b) : " + a.equals(b));
```

```
    }
}
```

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

Both `true` -- so `==` looks fine, and this is the trap laid. Java keeps a memory-saving store called the **String pool**. When you write the same text literal twice, Java doesn't build two objects; it reuses one and points both arrows at it. So `a` and `b` hold the same arrow, and `==` is true -- not because the text matches, but because there is genuinely only one object.

The pool is a good optimization. It is also a liar, because it makes `==` appear to work on text. Every beginner who tests `==` with two literals sees `true`, concludes `==` compares text, and ships that belief into code where it's false.

That belief breaks on the very next thing this book covers: any string built while the program actually runs.