if, else, and the = vs == Trap

An if only accepts a boolean. = orders; == asks. Put = inside an if on a boolean and your suite can run on a red build -- with no compiler error and no stack trace.

Automation Foundations · ashishautomationlabs.store

The `if` statement runs a block of code only when a condition is true:

```
if (condition) {
    // runs only when condition is true
} else {
    // runs only when condition is false
}
```

Read it exactly as English, because that is exactly what it means. Two mechanics the compiler cares about: the condition sits inside round brackets `( )`, and the code to run sits inside curly braces `{ }`.

And one thing the compiler does not allow, which will surprise you if you come from other languages: **the condition must be a boolean.** Not a number, not text, not "anything that looks true-ish." Java accepts `true` or `false` and nothing else. That strict rule is about to save you from a bug that has troubled programmers for fifty years -- and, once, to show you the bug it cannot catch.

Trap 1: = and == are different worlds

`==` asks a question: are these equal? `=` gives an order: make this equal to that. They are one keystroke apart and they do opposite things.

In an `if` you always **ask** -- use `==`. An `=` there is an order, and it changes your data.

Watch what happens when you type the wrong one -- first with a number:

```
public class C {
    public static void main(String[] args) {
        int statusCode = 500;
        if (statusCode = 200) {
            System.out.println("OK");
        }
    }
}

C.java:4: error: incompatible types: int cannot be converted to boolean
        if (statusCode = 200) {
            ^
1 error
```

The compiler catches it. `statusCode = 200` is an order, and its result is the number `200` -- and a number is not a boolean, so Java refuses. Your strict reviewer just prevented a bug. That is the value of the "must be a boolean" rule.

The version that gets through

Predict this one carefully:

```
public class B {
    public static void main(String[] args) {
        boolean buildIsGreen = false;
        if (buildIsGreen = true) {
            System.out.println("Running the regression suite...");
        } else {
            System.out.println("Build is red. Suite skipped.");
        }
        System.out.println("buildIsGreen is now: " + buildIsGreen);
    }
}
```

The variable is `false`. The suite should be skipped. Actual output:

```
Running the regression suite...
buildIsGreen is now: true
```

Read that twice. The build was red, and the code ran the suite anyway -- and changed the variable while doing it.

Why: `buildIsGreen = true` is an order, not a question. It sets the variable to `true`, and the result of that order is the value `true`. That value is a boolean, so the compiler is satisfied and says nothing. The `if` sees `true` and runs.

This is the perfect example of a trap that produces a **wrong verdict with no error at all**. No red text, no stack trace, no failure -- only a suite that ran when it should not have. The only protection is to know the difference so well that you never type the wrong one: `==` asks, `=` orders. When you compare, you always ask.

> *A test that crashes is annoying. A test that quietly passes -- or quietly runs -- when it should not is a disaster, because it destroys the one thing your suite is for: trust.*

You can branch on a boolean, and you know why `=` inside an `if` is a silent career- level bug. The next lesson combines conditions with `&&`, `||`, and `!` -- and shows why the **order** of a null check is what separates a suite that survives from one that crashes.