Chains, Order, and Decimal Tolerance

An else if chain stops at the first true branch -- wrong order creates dead code and wrong ratings. And never compare doubles with ==; compare the difference against a tolerance.

Automation Foundations · ashishautomationlabs.store

When there are more than two outcomes, you chain:

```
public class F {
    public static void main(String[] args) {
        int openDefects = 12;
        boolean blockerOpen = false;
        if (blockerOpen) {
            System.out.println("Verdict: NO-GO (blocker open)");
        } else if (openDefects > 20) {
            System.out.println("Verdict: NO-GO (too many defects)");
        } else if (openDefects > 5) {
            System.out.println("Verdict: GO WITH RISK");
        } else {
            System.out.println("Verdict: GO");
        }
    }
}
```

```
Verdict: GO WITH RISK
```

That is a release sign-off meeting, expressed in code. And it works because of one rule: Java checks the conditions from top to bottom, and **stops at the first one that is true**. Everything below it is skipped -- even if it is also true.

That rule is helpful. It is also Trap 2, and it is a silent one.

Trap 2: a chain in the wrong order lies to you

You are rating API response times. Under 50ms is EXCELLENT, under 200ms is FAST, under 1000ms is ACCEPTABLE. Predict what a very fast 45ms response gets rated:

```
public class G {
    public static void main(String[] args) {
        int responseMs = 45;
        if (responseMs < 1000) {
            System.out.println("Rating: ACCEPTABLE");
        } else if (responseMs < 200) {
            System.out.println("Rating: FAST");
        } else if (responseMs < 50) {
            System.out.println("Rating: EXCELLENT");
        }
```

```
    }
}

Rating: ACCEPTABLE
```

45 milliseconds -- excellent performance -- rated only as "acceptable." No error, no warning, no crash. The program ran perfectly and produced a wrong answer.

Root cause: `45 < 1000` is true, so Java stopped there. The EXCELLENT and FAST branches could never run, for any value -- 5ms would also be rated "acceptable." Two whole branches were **dead code**: code that can never be reached. And nothing told you.

The fix is to order the chain from **most specific to most general** -- narrowest condition first:

```
if (responseMs < 50) {
    System.out.println("Rating: EXCELLENT");
} else if (responseMs < 200) {
    System.out.println("Rating: FAST");
} else if (responseMs < 1000) {
    System.out.println("Rating: ACCEPTABLE");
}
```

You already know this instinct from test design: when you write boundary conditions, you check the tightest boundary first. Same discipline, new syntax.

When you review someone's `else if` chain, read it from the bottom upward and ask one question for each branch: "could a condition above this one already be true, so this branch never runs?" That single review habit will make you look senior very quickly.

Trap 3: comparing decimals with ==

Chapter 2 promised this fix. You know that `0.1 + 0.2` does not produce exactly `0.3`. So what happens when a test compares them?

```
public class H {
    public static void main(String[] args) {
        double expectedTotal = 0.3;
        double actualTotal = 0.1 + 0.2;
        if (actualTotal == expectedTotal) {
            System.out.println("Exact compare : PASS");
        } else {
            System.out.println("Exact compare : FAIL");
        }
        double tolerance = 0.0001;
        if (Math.abs(actualTotal - expectedTotal) < tolerance) {
            System.out.println("Tolerance : PASS");
        } else {
            System.out.println("Tolerance : FAIL");
        }
    }
```

```
}
```

```
Exact compare : FAIL
Tolerance : PASS
```

A correct cart total, reported as a defect. That is a **false failure** -- and a suite that produces false failures gets ignored, which is how automation initiatives die.

The professional fix is the second check: compare with a **tolerance**. `Math.abs(...)` gives the size of the difference; if that difference is smaller than a tolerance you choose, the values are equal for testing purposes.

> *Never compare two decimal values with `==`. Always compare the difference against a tolerance. Every serious assertion library offers a "delta" or "tolerance" parameter for exactly this reason -- now you know what it is for.*

You can write a release sign-off as a chain, order it so no branch is dead, and compare decimals the way a professional suite does. The next lesson is `switch` -- including the forgotten-`break` trap, and the modern arrow form that makes fall-through impossible.