And, Or, Not -- and Why Order Saves You from NPE

Real verdicts combine conditions. && short-circuits: check null first, then use the thing. Reverse the order and you get the NullPointerException Chapter 2 taught you to read.

Automation Foundations · ashishautomationlabs.store

Real verdicts are rarely single-condition. "Ship it if the build is green **and** there are no blockers **and** the smoke suite passed." Java gives you three operators to build those:

- `&&` -- AND. True only when both sides are true.
- `||` -- OR. True when at least one side is true.
- `!` -- NOT. Flips `true` to `false` and `false` to `true`.

```
if (buildIsGreen && smokePassed) { /* both must hold */ }
if (isBlocker || isSecurityIssue) { /* either one is enough */ }
if (!isBlocker) { /* true when NOT a blocker */ }
```

That is the easy part. Now the part that matters -- and that almost no beginner book explains properly, even though it saves you from crashes every single day.

Short-circuit: Java is lazy, and that laziness protects you

When Java evaluates `A && B`, it checks `A` first. If `A` is false, the whole thing is false no matter what `B` says -- so Java does **not** evaluate `B` at all.

Same for `||`: when `A` is true, `B` is never evaluated.

This is **short-circuit evaluation**. It is easy to think of it as a small speed improvement. It is much more than that: it is a safety mechanism. Here is the single most useful pattern in this chapter:

```
public class D {
    public static void main(String[] args) {
        String defectId = null;
        if (defectId != null && defectId.startsWith("DEF-")) {
            System.out.println("Valid defect id.");
        } else {
            System.out.println("No defect id to check.");
        }
    }
}
```

```
No defect id to check.
```

No crash. Trace it: `defectId != null` is false. Java short-circuits -- it never evaluates `defectId.startsWith("DEF-")`, so it never follows the null arrow from Chapter 2. The `NullPointerException` that was waiting simply does not happen.

Now reverse the order, and the protection disappears:

```
if (defectId.startsWith("DEF-") && defectId != null) {
```

```
Exception in thread "main" java.lang.NullPointerException:
Cannot invoke "String.startsWith(String)" because "<local1>" is null
```

Same two conditions. Same `&&`. Different order -- and now a crash, because Java evaluated the left side first and followed a null arrow. The null check came after the thing that needed it, which is like checking whether the test environment exists after you have already tried to log into it.

> *Rule you will use for the rest of your career: check for null first, then use the thing. Write* `if (x != null && x.something())`*, never the reverse. This exact pattern appears in every automation framework you will ever read --* `if (element != null && element.isDisplayed())` *is what separates a suite that survives a slow page from one that crashes on it.*

The look-alikes: & and |

Java also has single `&` and single `|`. They produce the same true or false answer -- but they **always** evaluate both sides, even when the answer is already decided.

```
public class E {
    static boolean check(String name, boolean result) {
        System.out.println(" -> running check: " + name);
        return result;
    }

    public static void main(String[] args) {
        System.out.println("Using && :");
        if (check("A", false) && check("B", true)) { }
        System.out.println("Using & :");
        if (check("A", false) & check("B", true)) { }
    }
}
```

```
Using && :
 -> running check: A
Using & :
 -> running check: A
 -> running check: B
```

With `&&`, check B never ran. With `&`, check B ran anyway. Now imagine check B was `defectId.startsWith(...)` and `defectId` was null: with `&&` you are safe, with `&` you crash.

In conditions, always use `&&` and `||`. Single `&` and `|` have uses elsewhere; inside an `if`, typing one instead of two is a defect that will hurt you sooner or later.

You can combine judgments, and you know why null-check-first is not style -- it is crash prevention. The next lesson is two more silent traps: an `else if` chain in the wrong order, and comparing decimals with `==`.