switch, Fall-Through, and the Arrow Form

A classic switch without break launches three browsers from one name. The arrow form cannot fall through -- and Bug Hunt #3 shows three silent defects that still print PASS.

Automation Foundations · ashishautomationlabs.store

Some decisions are not ranges -- they are a single value matched against a list of options. Browser name. Environment name. Severity code. An `else if` chain works, but `switch` says it better.

Here is the classic form, and hidden inside it is Trap 4:

```
public class I {
    public static void main(String[] args) {
        String browser = "firefox";
        switch (browser) {
            case "chrome":
                System.out.println("Launching Chrome");
            case "firefox":
                System.out.println("Launching Firefox");
            case "edge":
                System.out.println("Launching Edge");
            default:
                System.out.println("Unknown browser");
        }
    }
}
```

Predict the output. The obvious answer: "Launching Firefox." Actual:

```
Launching Firefox
Launching Edge
Unknown browser
```

Three browsers launched from one name. This is **fall-through**: once `switch` finds a matching case, it runs that line and keeps running every line below it until something stops it. What stops it is the keyword `break`:

```
switch (browser) {
    case "chrome":
        System.out.println("Launching Chrome");
        break;
    case "firefox":
        System.out.println("Launching Firefox");
        break;
```

```
    case "edge":
        System.out.println("Launching Edge");
        break;
    default:
        System.out.println("Unknown browser: " + browser);
}
```

```
Launching Firefox
```

Fixed. And `default` is the branch that runs when nothing else matched -- the automation equivalent of handling unexpected input instead of pretending it cannot happen.

A forgotten `break` is one of the oldest bugs in programming. Which is exactly why modern Java gives you a form where the trap cannot exist at all.

The modern switch (prefer it)

Since Java 14, `switch` can be written with an arrow, and it can produce a value:

```
public class K {
    public static void main(String[] args) {
        String env = "uat";
        String url = switch (env) {
            case "dev" -> "https://dev.shopcart.internal";
            case "uat" -> "https://uat.shopcart.internal";
            case "prod" -> "https://www.shopcart.com";
            default -> throw new IllegalArgumentException("Unknown env: " + env);
        };
        System.out.println("Target URL: " + url);

        String sev = "P2";
        int slaHours = switch (sev) {
            case "P1" -> 4;
            case "P2" -> 24;
            case "P3", "P4" -> 72;
            default -> 168;
        };
        System.out.println("SLA hours : " + slaHours);
    }
}
```

```
Target URL: https://uat.shopcart.internal
SLA hours : 24
```

No `break` anywhere -- arrow cases never fall through, so Trap 4 cannot happen in this form. Multiple values share one branch (`case "P3", "P4" ->`). And the whole `switch` produces a value assigned straight into a variable.

That first example is the environment-URL switch that lives in essentially every automation framework ever built. You just wrote framework code.

When to use what: `switch` when one value is matched against a list of fixed options; `if` / `else if` when the conditions are ranges, comparisons, or combinations. When you use `switch`, prefer the arrow form.

The ternary: a decision on one line

For a small either-or choice:

```
int failed = 0;
String verdict = (failed == 0) ? "PASS" : "FAIL";
System.out.println("Suite verdict: " + verdict);

Suite verdict: PASS
```

Read it as: `condition ? value-if-true : value-if-false`. You will see it in framework code, so you must be able to read it. Use it only for short, simple choices. As soon as it grows long, or you nest one inside another, switch back to a plain `if` / `else` -- the next person to maintain your suite will thank you. That person may well be you, six months from now, at 11 p.m., before a release.

Bug Hunt #3

This code compiles, runs, and prints three happy success messages. Every one of them is hiding a defect. All silent -- none a crash. Read first, then run.

```
public class BugHunt {
    public static void main(String[] args) {
        String status = "ACTIVE";
        int retries = 3;
        double responseTime = 1.5;
        double expected = 1.5;
        if (status == "ACTIVE") {
            System.out.println("User is active");
        }
        if (retries > 0 & retries < 5) {
            System.out.println("Retry count is sane");
        }
        if (responseTime == expected) {
            System.out.println("Response time matches");
        }
    }
}

User is active
Retry count is sane
Response time matches
```

All three printed. So where are the defects?

- `status == "ACTIVE"` compares **references**, not text. It happened to work because

both sides are the same literal. The moment that value arrives from an API, `==` quietly returns false. Always use `.equals()` for text (Chapter 6 shows why).

- `retries > 0 & retries < 5` uses single `&`. Right answer today; no short-circuit -- and the day the right-hand side can crash on null, this line will crash. Use `&&`.
- `responseTime == expected` compares doubles exactly. It passes only because both were literals. Use a tolerance.

Three defects, three passing messages, zero errors. **Passing is not the same as correct.**

Checkpoint

- What is the only kind of value Java accepts inside an `if (...)`?
- `buildIsGreen = true` inside an `if` compiles fine but is almost always a bug. Why does it compile, and what does it do?
- Explain short-circuit evaluation, and write the safe way to check a `String` that might be null.
- Predict: `int x = 7; if (x > 5) { print("A"); } else if (x > 3) { print("B"); }` -- what prints, and why does the other branch not?
- Two `double` values should be equal but the test fails. What is happening, and what is the fix?
- What does `break` do in a `switch`, and why does the arrow form not need it?

Your programs can judge. That is the difference between a program that reports and a program that tests. You have met the silent traps that still print PASS -- which makes you a more careful reviewer than most developers are of their own code.

Next, the boring part of testing becomes the powerful part: **loops** -- running the same check across many rows of data without copying the steps fifty times.