Break It on Purpose

A compiler error is a defect report -- where, what, and position. Finding it is good news, not a verdict on your ability. That is the skill that keeps people in automation.

Automation Foundations · ashishautomationlabs.store

You would never trust a login feature tested only with valid credentials. Apply that instinct to your own learning: feed the compiler invalid input on purpose, and see what its defect reports look like.

This is the most important lesson in the Chapter 1 path.

Experiment 1 -- Delete a semicolon

```
public class Broken1 {
    public static void main(String[] args) {
        System.out.println("Build 42 deployed")
    }
}
```

The semicolon after `println(...)` is gone. Predict what happens; then run it. The compiler responds:

```
Broken1.java:3: error: ';' expected
        System.out.println("Build 42 deployed")
                                                ^
1 error
```

Read it as the defect report it is, field by field:

- **Where:** file `Broken1.java`, line 3 -- the exact line, no "steps to reproduce"

needed.

- **What:** `';' expected` -- the summary.
- **Position:** the `^` marker points at the exact spot the compiler noticed something

missing.

Fix: add the semicolon. Run again. Green. You resolved your first defect as the developer, and the investigation took seconds.

Experiment 2 -- Break the capitalization

```
public class Broken2 {
    public static void main(String[] args) {
        system.out.println("Smoke test starting");
    }
```

```
}
```

Lowercase `s` in `system`. The report:

```
Broken2.java:3: error: package system does not exist
        system.out.println("Smoke test starting");
        ^
1 error
```

Two lessons. First: Java is **case-sensitive**, always -- `System`, `system`, and `SYSTEM` are three unrelated names, and only the first one exists.

Second, deeper: the compiler describes the problem from _its_ point of view. It looked for something called `system`, found nothing, and said so. It does not say "you probably meant `System`" -- though the IDE often will. Compiler messages tell you what failed; your job is to work out why -- exactly the relationship between symptom and root cause in defect analysis. You have done this professionally for years. You are only reading a new report format.

> *Accept the IDE's auto-fix when you understand why it is right. Read the message first, every time. The day the IDE guesses wrong, blind acceptance turns working code into broken code.*

The permanent lesson

When you see an error, you have not failed -- you have received information.

The beginners who give up on automation are not the ones who make errors. Everyone makes them; senior engineers make them daily. They are the ones who treat every error as a judgment of their ability, and quietly stop.

You have a professional advantage no computer-science graduate has: you have spent years believing -- correctly -- that **finding a defect is good news**. Nothing about that changes when the defect is yours. Sixty seconds of reading beats sixty minutes of staring.

Bug Hunt #1

A colleague sends you this file, `SmokeCheck.java`, complaining that "Java is broken." There are three defects. Find all three by reading -- do not run it yet. Your boundary-analysis instinct applies: check the edges of every line.

```
public class smokeCheck {
    public static void main(String[] args) {
        String build = "R2.4.1";
        System.out.println("Testing build: " + build)
```

```
        system.out.println("Smoke suite: PASS");
    }
}
```

Try first. Then check:

- The class is named `smokeCheck` but the file is `SmokeCheck.java` -- the names must match exactly, including capital letters.
- Line 4 is missing its semicolon.
- Line 5 has lowercase `system`.

Type it, run it, and confirm the compiler reports them. Note: it often reports the semicolon first, then stops -- errors arrive in waves. Fix, run again, repeat. Just like retesting after a fix and finding the defect behind the defect.

Checkpoint

Answer from memory:

- In one sentence: what is programming? Which part of your testing experience is it closest to?
- Your `.java` file becomes what kind of file, produced by which tool, executed by what?
- An interviewer asks the JDK / JRE / JVM question. Go.
- Your program shows `error: ';' expected` at line 7. Describe your next thirty seconds.
- True or false: `String` and `string` mean the same thing in Java. What is the general rule?

You installed a JDK, wrote a real check, broke it on purpose, and read both reports calmly. You can answer the opening interview question of the trade.

Your check still has a weakness you may have already spotted: the "actual" result was fixed in the code. Real checks work with data that changes -- usernames, amounts, counts, environment names. The moment data changes, you need to understand what Java data _is_: its types, its containers, and the surprisingly deep question of where it lives in memory.

That question sounds academic -- until you watch a program insist that `0.1 + 0.2` is not `0.3`. That is exactly where the next chapter begins: with a bug report.