Choosing a Type Is Writing a Spec

DEF-8801 looks like a wallet bug: 0.1 + 0.2 is not 0.3. Reproduce it in five lines of Java -- the root cause is the type you chose, not broken application code.

Automation Foundations · ashishautomationlabs.store

Imagine this defect lands in your queue for triage:

DEF-8801 -- Wallet balance incorrect after cashback Steps: Wallet balance is Rs 0.10. A cashback of Rs 0.20 is credited. Expected: balance shows 0.3 Actual: balance shows 0.30000000000000004 Priority: the payments team is in a meeting room with the door closed.

Your triage instinct fires immediately: display bug? Rounding bug? Calculation bug? Can it be reproduced?

Here is the twist -- you can reproduce it right now, in five lines of Java, and the root cause is not in anyone's application code. It is in how computers store decimal numbers. That means it exists in every language, every application, and one day in your test data.

Reproduce it

Type this and run it. Predict the output first -- you already know the habit.

```
public class FloatSurprise {
    public static void main(String[] args) {
        double walletBefore = 0.1;
        double cashback = 0.2;
        double walletAfter = walletBefore + cashback;
        System.out.println("Expected: 0.3");
        System.out.println("Actual : " + walletAfter);
    }
}
```

Predicted output: obviously `0.3`. Actual output:

```
Expected: 0.3
Actual : 0.30000000000000004
```

The machine did exactly what we wrote -- Chapter 1's law -- and still the answer is wrong by a tiny amount. Nothing is broken.

What actually happened

`double`, the type we chose for our data, stores decimal numbers as **close approximations** in binary. `0.1` has no exact binary form -- just as `1/3` has no exact decimal form (`0.3333...` forever).

Each stored value was a tiny bit off, and adding them made the tiny error visible.

Stop and think about what just happened, because it changes how you see the rest of this chapter:

> *Choosing a data type is like writing a requirements spec for your data.*

And like any spec, it has capabilities, limits, and edge cases. You have spent years testing systems against their specs. Now you are on the writing side.

Triage verdict for DEF-8801

Two professional fixes, named early so you are not left hanging:

- **Never compare doubles for exact equality.** A later lesson covers tolerance-based comparison -- the tester's instinct, written in code.
- **For money, use exact-decimal machinery** (Java's `BigDecimal`). We will only mention it here; the point today is the diagnosis, not the full API.

Works as designed at the language level. The defect, when it appears, is in code that trusted doubles to be exact.

`double` is still the right choice for rates, measurements, and response times, where a tiny approximation does not matter. Fear nothing; just choose on purpose.

You just watched a type choice create a believable production incident. The next lesson is what a variable _physically_ is -- a labeled box -- and how the compiler protects you from putting the wrong kind of data in it.