Boundary-Test Your Types

You do not trust a spec until you test its edges. int overflows in silence; int divided by int can zero a pass-rate report. Boundary analysis applies to the language too.

Automation Foundations · ashishautomationlabs.store

You do not trust a spec until you have tested its edges -- so boundary-test `int`. Its documented maximum is `2,147,483,647`. What lies one step past the boundary?

Overflow wraps in silence

```
public class Overflow {
    public static void main(String[] args) {
        int views = 2_147_483_647;
        views = views + 1;
        System.out.println("Views after one more visit: " + views);
    }
}
```

```
Views after one more visit: -2147483648
```

No error. No warning. The value silently wraps around to the most negative number.

This is not a Java quirk. It is how fixed-size number storage works everywhere. It has caused real production incidents: view counters and ID sequences that went negative when systems grew past `int`.

For you, the lesson is a design habit: when a quantity could realistically go past two billion (database row counts, millisecond timestamps, global counters), choose `long` **from day one**. And the next time you test a system with big counters -- you now know a boundary case the developers may not have planned for.

Integer division throws the fraction away

A less obvious edge, and a favorite silent bug. You are computing a pass rate -- 45 passed out of 60. Predict carefully:

```
public class IntDivision {
    public static void main(String[] args) {
        int passed = 45;
        int total = 60;
        double passRate = passed / total * 100;
        System.out.println("Pass rate: " + passRate + "%");
    }
}
```

The prediction says `75.0%`. The actual:

```
Pass rate: 0.0%
```

A pass rate of zero, from perfectly good numbers, stored in a perfectly good `double`.

Root cause: Java evaluates `passed / total` first. Both sides are `int`, so Java performs **integer division**, which keeps only the whole-number part. `45 ÷ 60 = 0.75`; whole part: `0`. Then `0 × 100 = 0`, and only after that does the `0` land in the `double` box as `0.0`. The `double` on the left could not save a calculation that had already thrown away the fraction.

The fix: a cast

A **cast** is a direct instruction to convert a value's type, written as the target type in parentheses:

```
double fixedRate = (double) passed / total * 100;
// Fixed: 75.0%
```

`(double) passed` converts `45` to `45.0` before the division. A division with even one decimal side is decimal division; the `0.75` survives.

Conversions that cannot lose information happen automatically (`int` to `double`). Conversions that can lose information need the written cast. The cast is Java making you accept the risk in writing.

File this pattern away permanently: integer division is a classic source of silently wrong numbers in report calculations, percentages, and average response times. You will find this bug in a real codebase someday. It might even be one you are paid to find.

You boundary-tested `int` like any other spec, and you know why a dashboard can print `0.0%` with honest inputs. The next lesson is the chapter's big idea: what is actually _inside_ the variable's box -- the value itself, or an arrow -- and why that picture is what makes `null` and the NPE make sense.