The Missing this, and the Silent Null

A constructor that compiles and runs, prints one line, and quietly loses two of its three fields to null -- because of a bug that has bitten every Java programmer once.

Automation Foundations · ashishautomationlabs.store

This program compiles and runs. It prints one line, and that line exposes a bug that has bitten every Java programmer at least once. Read the constructor closely first, then run it.

```
class TestResult {
    String name;
    String status;
    int durationMs;

    TestResult(String name, String status, int durationMs) {
        name = name;
        status = status;
        this.durationMs = durationMs;
    }

    public String toString() {
        return name + " [" + status + ", " + durationMs + "ms]";
    }
}

TestResult r = new TestResult("LoginTest", "PASS", 240);
System.out.println(r);
```

```
null [null, 240ms]
```

The duration is right, but the name and status are `null` -- even though you clearly passed `"LoginTest"` and `"PASS"`. Look at the constructor. `name = name` and `status = status` are missing `this`. So each line assigns the parameter to itself and never touches the field, which keeps its default value of `null`. The one line that's correct, `this.durationMs = durationMs`, is the one with `this` -- which is why 240 survived.

Two fields quietly lost, one keyword to blame. The fix is to write `this.name = name` and `this.status = status`, and the giveaway in real life is exactly this: an object whose fields are `null` or zero right after you built it, no matter what you passed in.

Checkpoint

- Define, in one sentence each: class, object, field, and method.
- What does a constructor do, and how do you recognize one?
- What does `this` mean, and give the specific situation in this chapter where

leaving it out causes a silent bug.

- What does a `toString` method give you, and what do you see without one?
- Why does the default `.equals` return `false` for two objects with identical

fields, and how do you change that?

- State the one rule linking `equals` and `hashCode`.

The interview shortlist for this chapter

All common, all straight from what you just learned: What's the difference between a class and an object? _Blueprint versus instance._ What does a constructor do? _Runs at `new` to build the object; same name as the class, no return type._ What is `this`? _A reference to the current object, used to tell a field apart from a parameter of the same name._ Why override `equals`, and why must `hashCode` come with it? _So objects compare by contents, not identity; equal objects must share a `hashCode` or they break in `Set` and `Map`._ What does `private` give you? _Control -- outside code goes through methods you own._

You can design your own types now. You can give a class fields for its data and methods for its behavior, build finished objects with a constructor, print them readably with `toString`, and protect their data with `private`. Most importantly, you can decide when two of your objects are equal and prove it to Java with `equals` and `hashCode`. That's what lets a list recognize a duplicate defect, and it redeems the promise open since Chapter 6. Your test data is no longer loose values; it's objects you shaped, each keeping its own pieces together.

There's one piece of Java you've used since page one and never decoded: `public static void main(String[] args)`. You now know almost every word in it. You know `String[]` is an array, you know a method is behavior with a name, and you know a class holds it. What you haven't met is what `static` means, how a method takes parameters and returns a value in full, and why `main` is written exactly that way. Methods have been quietly present in every chapter -- `isPassed`, `getUrl`, `equals`, `toString`. The next chapter makes them the main subject, and finally opens the very first promise note of the book. That's Chapter 8.