Java Refactoring

Table of contents

Introduced in 2026-04-02-refactoring[1] (Lecture, Week 6).

What is refactoring?

> A disciplined technique for continuously restructuring an existing body of code, altering its internal structure *without changing its external behaviour*. (refactoring.guru)

Why refactor?

Over time, as features are added: quick fixes ("hacks") accumulate, the design becomes messy, and code becomes harder to change.

Good practice

- **Don't** refactor and add functionality at the same time — keep the two activities separate.
- Have good tests *before* refactoring, so you'll know immediately if you've broken something (see java-testing[2]).

[1] courses/csse2002/2026-04-02-refactoring.pdf
[2] courses/csse2002/java-testing.pdf

- Take short, deliberate steps: move a member variable, split a method, rename a variable. Refactoring is usually many small, localised changes that add up to a larger-scale change — small steps + testing after each one avoids prolonged debugging.
- Refactor early, refactor often.

Example: making it easier to add a feature

A `Vehicle` class using a `type` string and a chain of `if/else` branches is fragile to extend — adding a `"scooter"` case means editing `travelTime` *and* remembering every other place that branches on `type`:

```
public class Vehicle {
    private String type;

    public Vehicle(String type) { this.type = type; }

    public int travelTime(int distance) {
        if (type.equals("car")) {
            return distance / 80;
        } else if (type.equals("bike")) {
            return distance / 20;
        }
        return distance;
    }
}
```

Refactored to use inheritance/polymorphism (see java-polymorphism[3]), adding a `Scooter` is just one new class — no existing code needs editing:

```
abstract class Vehicle {
    public abstract int travelTime(int distance);
}
class Car extends Vehicle {
    @Override
    public int travelTime(int distance) { return distance / 80; }
}
class Bike extends Vehicle {
    @Override
    public int travelTime(int distance) { return distance / 20; }
}
class Scooter extends Vehicle {
    @Override
    public int travelTime(int distance) { return distance / 5; }
}
```

[3] courses/csse2002/java-polymorphism.pdf

Example: making it easier to understand

Extracting named boolean-returning methods (`isHotAndSunny()`, `isPleasant()`) out of inline compound conditions makes the *intent* of a branch obvious at a glance, instead of making the reader re-derive it from raw comparisons every time:

```
// Before
if (temperature > 25 && !isRaining) { ... }
else if (temperature <= 25 && !isRaining) { ... }
else if (isRaining) { ... }

// After
if (isHotAndSunny()) { ... }
else if (isPleasant()) { ... }
else if (isRaining) { ... }
```

Code smells

A "code smell" is a surface indication that usually corresponds to a deeper problem in the design.

Bad naming — single-letter/cryptic names (`r`, `x`, `y`, `g(x)`) force the reader to trace logic to understand intent; descriptive names (`evenCount`, `number`, `isEven(number)`) make code self-documenting.

Duplication — near-identical blocks repeated for different data (e.g. computing an average for two separate arrays with two copy-pasted loops) should be extracted into a single shared method (`average(int[] numbers)`), so a bug fix or improvement only needs to happen once.

Feature envy — a method that spends most of its time reaching into *another* class's data (`cart.getItems()`, `cart.getVoucher()`) rather than its own is a sign the method's logic actually belongs on the class it's "envious" of. Moving `checkout()` onto `ShoppingCart` itself (next to `getItems`/`getVoucher`) removes the envy and reduces coupling between `User` and `ShoppingCart` (see java-cohesion-and-coupling[4]).

Many other code smells exist — see Martin Fowler's *Refactoring* (2nd ed., 2019), Robert Martin's *Clean Code* (2008), John Ousterhout's *A Philosophy of Software Design* (2018), and the refactoring.com catalog.

[4] courses/csse2002/java-cohesion-and-coupling.pdf