OCTOBER 2025

CLEAN CODE IN JAVA

SIMPLE REFACTORING EXAMPLES FOR BETTER READABILITY AND MAINTAINABILITY

Author: Emir Totić — Java Backend Engineer

Why Clean Code Matters

Clean code is not a luxury - it’s what keeps projects alive as they grow.
Every messy method, unclear name, or duplicated block of logic adds hidden cost to future changes.

Why it matters:

- Easier to debug and extend - fewer surprises later
- Improves team collaboration - everyone understands the intent
- Reduces technical debt - less time wasted “decoding” legacy code
- Builds trust - clean code reflects professionalism and discipline

In fast-moving teams, **clarity beats cleverness.**
Clean code allows you to move fast without breaking things.

THE PROBLEM
Code that works, but doesn’t scale

Even working code can be dirty code. It runs fine - but it’s hard to read, test, and maintain.

Common symptoms:

- Long, nested methods doing too many things
- Duplicated logic across services
- Vague names like data or doStuff()
- Hidden business rules and “magic numbers”
- Mixed responsibilities (validation + persistence + logic)
- Inconsistent error handling and null checks

Result:
Harder debugging, more bugs, and slower development.

Goal:
Code that communicates clearly - not just runs.

What Is Clean Code?

Clean code isn’t just about making it work - it’s about making it clear.
It’s easy to read, test, and extend.

Key principles:

- **Readable**: Code should explain itself - no guessing required.
- **Focused**: One method = one responsibility.
- **Consistent**: Same logic, same structure across the project.
- **DRY**: Don’t Repeat Yourself - remove duplication.
- **Meaningful names**: Methods and variables that describe intent.
- **Simple over clever** — clarity wins every time.

In short:
Clean code is code that you and your teammates can understand six months later - without comments.

Refactoring Example #1: Long Method

Before (hard to read & maintain)

```
public void processUser(User user) {
    if (user != null) {
        if (user.isActive()) {
            System.out.println("Processing user: " + user.getName());
            if (user.getOrders() != null) {
                for (Order o : user.getOrders()) {
                    if (o.getStatus().equals("NEW")) {
                        o.process();
                    }
                }
            }
        }
    }
}
```

After (clean & modular)

```
public void processUser(User user) {
    if (isInactive(user)) return;
    processOrders(user.getOrders());
}

private boolean isInactive(User user) {
    return user == null || !user.isActive();
}

private void processOrders(List<Order> orders) {
    orders.stream()
          .filter(o -> "NEW".equals(o.getStatus()))
          .forEach(Order::process);
}
```

Achievements: Reduced nesting = easier flow, Separate responsibilities = reusable methods and Self-explanatory method names

Refactoring Example #2: Duplicated Code

Before (repetition everywhere)

```
if (status.equals("ACTIVE")) {
    sendEmail(user.getEmail());
}
if (status.equals("INACTIVE")) {
    sendEmail(user.getEmail());
}
```

After (apply DRY principle)

```
if (status.equals("ACTIVE") ||
    status.equals("INACTIVE")) {
    sendEmail(user.getEmail());
}
```

Achievements:

- Eliminates duplicate logic
- Easier to modify or extend later
- Keeps the intent clear and focused

Refactoring Example #3: Magic Numbers - Constants

Before (unclear rules)

```
if (age > 18 && balance > 1000) {
    grantAccess();
}
```

After (readable & maintainable)

```
private static final int MIN_AGE = 18;
private static final double MIN_BALANCE = 1000.0;

if (age > MIN_AGE && balance > MIN_BALANCE) {
    grantAccess();
}
```

Achievements:

- Business rules are explicit and reusable
- Easier to update values in one place
- Code communicates intent immediately

Clean Code Principles (Recap)

PRINCIPLE	DESCRIPTION
DRY – *Don’t Repeat Yourself*	Avoid duplication. Extract shared logic into reusable methods or utilities.
SRP – *Single Responsibility Principle*	Each class or method should have one clear purpose - no “god” classes.
KISS – *Keep It Simple, “Stupid”*	Prefer simplicity over clever solutions. Readability always wins.
YAGNI – *You Ain’t Gonna Need It*	Don’t build features “just in case.” Add them only when truly needed.
Meaningful Names	Use descriptive names for methods, variables, and classes - code should explain itself.

Conclusion

Writing clean code is not just a technical skill - it’s a mindset.
It’s about respecting the reader, reducing complexity, and building systems that evolve gracefully over time.

Refactoring doesn’t mean rewriting everything - it means improving clarity step by step.
Every renamed variable, extracted method, or removed duplication brings the code closer to what it was meant to express.

In real-world projects, deadlines will always exist - but clean code is what keeps your system stable when things move fast.

Remember:
Good code works today.
Clean code still works six months from now.

THANK YOU

Thank you for taking the time to read this short guide on Clean Code principles. I hope it helps you write code that’s not only functional - but elegant and easy to maintain.

Let’s connect and share ideas about Java, Spring Boot, and software craftsmanship!


www.linkedin.com/in/emirtotic


https://emirtotic.github.io/portfolio-site


emirtotic@gmail.com