
1Code Quality Exercise: Refactor Bad Code

Task:

The following code violates **naming conventions, formatting, and documentation best practices**. **Refactor it** to follow best practices.

Bad Code:

```
class c {
    int a;

    public c(int b) {
        a = b;
    }

    void p() {
        System.out.println("Value: " + a);
    }

    public static void main(String[] args) {
        c obj = new c(10);  // creating an object with value 10
        obj.p();          // calling the p() method to print the value
    }
}
```

xpected Output After Refactoring

```
class Counter {
    private int value;

    public Counter(int initialValue) {
        this.value = initialValue;
    }

    public void printValue() {
        System.out.println("Value: " + value);
    }

    public static void main(String[] args) {
        Counter counter = new Counter(5);  // creating a
Counter with initial value 5
```

```
        counter.printValue();              // printing the value
    }
}
```

What we Fixed:

Class name follows **PascalCase (Counter).**
Variable names are meaningful (**value instead of a**).
Method names use **camelCase (printValue() instead of p()).**

Checkstyle Rules Summary

Rule Category	Rule Description
Naming Conventions	- **Class names** must be in PascalCase (e.g., CheckstyleExample)
	- **Method names** must be in camelCase (e.g., printValue)
	- **Variable names** should be meaningful (avoid single-letter names like a)
Visibility Modifiers	- Class fields should be **private**
	- Top-level classes should be explicitly **public**
Documentation	- **Javadoc comments** are required for:
	- Public classes
	- Public methods and constructors
Code Formatting	- No **tab characters** (use spaces instead)
	- **Consistent indentation** (typically 2 or 4 spaces depending on config)
	- **Braces {}** should always be used for control structures (if, for, etc.)
	- **Spaces** between keywords and parentheses (e.g., if (condition))
Comment Style	- **Single-line comments** should begin with // followed by a space
	- Comments should be **meaningful and properly capitalized**
Code Structure	- Avoid too many statements on one line
	- Keep lines within **max line length** (usually 100–

	120 characters)

---

2 Static Code Analysis Exercise: Run Checkstyle

Task:

Install and run **Checkstyle** to detect code violations.

Steps:

1 Add Checkstyle to your Maven pom.xml

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-
plugin</artifactId>
    <version>3.1.2</version>
    <executions>
        <execution>
            <phase>validate</phase>
            <goals>
                <goal>check</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

2 Run Checkstyle from the terminal

mvn checkstyle:check

Checkstyle will detect any code formatting issues.

What is SpotBugs?

SpotBugs is a static code analysis tool that detects **potential bugs** in Java programs. It's the successor to **FindBugs** and helps identify:

- **Null pointer dereferences**
- **Dead code**
- **Bad practices**
- **Multithreading issues**
- **Performance issues**
- **Security vulnerabilities**

Installing SpotBugs in STS (Spring Tool Suite)Step-by-Step:

1. **Open STS (Spring Tool Suite)**
   (Works the same way as Eclipse Marketplace — **STS is built on Eclipse.)**
2. **Go to:**
   Help → Eclipse Marketplace
3. **Search for:**
   SpotBugs
   (**Or sometimes: SpotBugs Eclipse Plugin)**
4. **Click Install**
   - Follow the installation steps
   - Accept licenses
   - Restart STS when prompted
5. **Verify Installation**
   - **Go to: Window → Preferences → SpotBugs**
   - **Or**
   - **Right Click on project/class and click on spot Bug**
   - You can configure settings like bug categories, filters, effort level, etc.

---

3 Code Review Exercise: GitHub Pull Request

Task:

Create a **GitHub** repository, push a branch, and open a **Pull Request (PR)** for review.

Steps:

1 **Initialize Git in your project**

```
git init
git add .
git commit -m "Initial commit"
```

2 **Create and push a feature branch**

```
git checkout -b feature-login
git commit -am "Added login feature"
git push origin feature-login
```

3 **Go to GitHub → Open a PR**
Assign **reviewers** and request feedback.

---

4 Metrics & SonarQube Exercise

Task:

Install and run **SonarQube** to measure code quality.

Steps:

1 **Run SonarQube via Docker**

```
docker run -d --name sonar -p 9000:9000 sonarqube
```

2 **Analyze a Maven project**

```
mvn sonar:sonar
```

SonarQube will show Cyclomatic Complexity, Code Duplication, and Maintainability Index.

---

5 Secure Coding Exercise: Prevent SQL Injection

Task:

Fix the SQL Injection vulnerability in the following code.

Insecure Code:

```
String query = "SELECT * FROM users WHERE username = '" + userInput + "'";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

Secure Fix (Using PreparedStatement)

```
String query = "SELECT * FROM users WHERE username = ?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString(1, userInput);
ResultSet rs = stmt.executeQuery();
```

This prevents attackers from injecting malicious SQL commands.
