Clean Coding in Java

1. Overview

In this tutorial, we'll go through clean coding principles. We'll also understand why clean code is important and how to achieve that in Java. Further, we'll see if there are any tools available to help us out.

2. What Is Clean Code?

So, before we jump into the details of clean code, let's understand what do we mean by clean code. Honestly, there can not be one good answer to this. In programming, some concerns reach across and hence result in general principles. But then, every programming language and paradigm present their own set of nuances, which mandates us to adopt befitting practices.

Broadly, **clean code can be summarized as a code that any developer can read and change easily**. While this may sound like an oversimplification of the concept, we'll see later in the tutorial how this builds up. Anywhere we hear about clean code, we perhaps come across some reference to Martin Fowler. Here is how he describes clean code in one of the places:

> *Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*

3. Why Should We Care About Clean Code?

Writing clean code is a matter of personal habit as much as it's a matter of skill. As a developer, we grow through experience and knowledge over time. But, we must ask why we should invest in developing clean code after all? We get that others will probably find it easier to read our code, but is that incentive enough? Let's find out!

Clean coding principles help us achieve a lot of desirable goals related to the software we intend to produce. Let's go through them to understand it better:

- *Maintainable Codebase*: Any software that we develop has a productive life and during this period will require changes and general maintenance. Clean code **can help develop software that is easy to change and maintain** over time.
- *Easier Troubleshooting*: Software can exhibit unintended behavior due to a variety of internal or external factors. It may often require a quick turnaround in terms of fixes and availability. Software developed with clean coding principles **is easier to troubleshoot for problems**.
- *Faster Onboarding*: Software during its lifetime will see many developers create, update, and maintain it, with developers joining at different points in time. This requires **a quicker onboarding to keep productivity high**, and clean code helps achieve this goal.

4. Characteristics of Clean Code

Codebases written with clean coding principles exhibit several characteristics that set them apart. Let's go through some of these characteristics:

- *Focused*: A piece of **code should be written to solve a specific problem**. It should not do anything strictly not related to solving the given problem. This applies to all levels of abstraction in the codebase like method, class, package, or module.
- *Simple*: This is by far the most important and often ignored characteristic of clean code. The software **design and implementation must be as simple as possible**, which can help us achieve the desired outcomes. Increasing complexity in a codebase makes them error-prone and difficult to read and maintain.

- *Testable*: Clean code, while being simple, must solve the problem at hand. It must be **intuitive and easy to test the codebase, preferably in an automated manner**. This helps establish the baseline behavior of the codebase and makes it easier to change it without breaking anything.

These are what help us achieve the goals discussed in the previous section. It's beneficial to start developing with these characteristics in mind compared to refactor later. This leads to a lower total cost of ownership for the software lifecycle.

5. Clean Coding in Java

Now that we've gone through enough background, let's see how we can incorporate clean coding principles in Java. Java offers a lot of best practices that can help us write clean code. We'll categorize them in different buckets and understand how to write clean code with code samples.

5.1. Project Structure

While Java doesn't enforce any project structure, **it's always useful to follow a consistent pattern to organize our source files, tests, configurations, data, and other code artifacts**. Maven, a popular build tool for Java, prescribes a particular project structure (https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html). While we may not use Maven, it's always nice to stick to a convention.

Let's see some of the folders that Maven suggests we create:

- *src/main/java*: For source files
- *src/main/resources*: For resource files, like properties
- *src/test/java*: For test source files
- *src/test/resources*: For test resource files, like properties

Similar to this, there are other popular project structures like Bazel (/bazel-build-tool) suggested for Java, and we should choose one depending on our needs and audience.

5.2. Naming Convention

Following **naming conventions can go a long way in making our code readable and hence, maintainable**. Rod Johnson, the creator of Spring, emphasizes the importance of naming conventions (https://blog.atomist.com/eighteen-years-of-spring/) in Spring:

> *"... if you know what something does, you got a pretty good chance guessing the name of the Spring class or interface for it ..."*

Java prescribes a set of rules (https://www.oracle.com/technetwork/java/codeconventions-135099.html) to adhere to when it comes to naming anything in Java. A well-formed name does not only help in reading the code, but it also conveys a lot about the intention of the code. Let's take some examples:

- *Classes*: Class in terms of object-oriented concepts is a blueprint for objects which often represent real-world objects. Hence it's meaningful to use nouns to name classes describing them sufficiently:

```
public class Customer {
}
```

- *Variables*: Variables in Java capture the state of the object created from a class. The name of the variable should describe the intent of the variable clearly:

```
public class Customer {
    private String customerName;
}
```

- *Methods*: Methods in Java are always part of classes and hence generally represent an action on the state of the object created from the class. It's hence **useful to name methods using verbs (/java-pojo-class#javabeans)**:

```
public class Customer {
    private String customerName;
    public String getCustomerName() {
        return this.customerName;
    }
}
```

While we've only discussed how to name an identifier in Java, please note that there are additional best practices like camel casing, which we should observe for readability. There can be more conventions related to naming interfaces, enums, constants as well.

5.3. Source File Structure

A source file can contain different elements. While Java **compiler enforces some structure, a large part is fluid**. But adhering to a specific order in which to places elements in a source file can significantly improve code readability. There are multiple popular style-guides to take inspiration from, like one by Google (https://google.github.io/styleguide/javaguide.html) and another by Spring (https://github.com/spring-projects/spring-framework/wiki/Code-Style).

Let's see how should a typical ordering of elements in a source file look:

- Package statement
- Import statements
    - All static imports
    - All non-static imports
- Exactly one top-level class
    - Class variables
    - Instance variables
    - Constructors
    - Methods

Apart from the above, **methods can be grouped based on their functionality or scope**. There is no one good convention, and the idea should be **decided once and then followed consistently.**

Let's see a well-formed source file:

```
# /src/main/java/com/baeldung/application/entity/Customer.java
package com.baeldung.application.entity;

import java.util.Date;

public class Customer {
    private String customerName;
    private Date joiningDate;
    public Customer(String customerName) {
        this.customerName = customerName;
        this.joiningDate = new Date();
    }

    public String getCustomerName() {
        return this.customerName;
    }

    public Date getJoiningDate() {
        return this.joiningDate;
    }
}
```

5.4. Whitespaces

We all know that it is easier to read and understand short paragraphs compared to a large block of text. It is not very different when it comes to reading code as well. Well-placed and consistent whitespaces and blank lines can enhance the readability of the code.

The idea here is to introduce logical groupings in the code which can help organize thought processes while trying to read it through. There is no one single rule to adopt here but a general set of guidelines and an inherent intention to keep readability at the center of it:

- Two blank lines before starting static blocks, fields, constructors and inner classes
- One blank line after a method signature that is multiline
- A single space separating reserved keywords like if, for, catch from an open parentheses
- A single space separating reserved keywords like else, catch from a closing parentheses

The list here is not exhaustive but should give us a bearing to head towards.

5.5. Indentation

Although quite trivial, almost any developer would vouch for the fact that **a well-indented code is much easier to read and understand**. There is no single convention for code indentation in Java. The key here is to either adopt a popular convention or define a private one and then follow it consistently across the organization.

Let's see some of the important indentation criteria:

- A typical best practice is to use four spaces, a unit of indentation. Please note that some guidelines suggest a tab instead of spaces. While there is no absolute best practice here, the key remains consistency!
- Normally, there should be a cap over the line length, but this can be set higher than traditional 80 owing to larger screens developers use today.
- Lastly, since many expressions will not fit into a single line, we must break them consistently:
  - Break method calls after a comma
  - Break expressions before an operator
  - Indent wrapped lines for better readability (we here at Baeldung prefer two spaces)

Let's see an example:

```
List<String> customerIds = customer.stream()
  .map(customer -> customer.getCustomerId())
  .collect(Collectors.toCollection(ArrayList::new));
```

5.6. Method Parameters

Parameters are essential for methods to work as per specification. But, **a long list of parameters can make it difficult for someone to read and understand the code**. So, where should we draw the line? Let's understand the best practices which may help us:

- Try to restrict the number of parameters a method accepts, three parameters can be one good choice

- Consider **refactoring (/cs/refactoring)** the method if it needs more than recommended parameters, typically a long parameter list also indicate that the method may be doing multiple things
- We may consider bundling parameters into custom-types but must be careful not to dump unrelated parameters into a single type
- Finally, while we should use this suggestion to judge the readability of the code, we must not be pedantic about it

Let's see an example of this:

```
public boolean setCustomerAddress(String firstName, String lastName, String streetAddress,
  String city, String zipCode, String state, String country, String phoneNumber) {
}

// This can be refactored as below to increase readability

public boolean setCustomerAddress(Address address) {
}
```

5.7. Hardcoding

Hardcoding values in code can often lead to multiple side effects. For instance, **it can lead to duplication, which makes change more difficult**. It can often lead to undesirable behavior if the values need to be dynamic. In most of the cases, hardcoded values can be refactored in one of the following ways:

- Consider replacing with constants or enums defined within Java
- Or else, replace with constants defined at the class level or in a separate class file
- If possible, replace with values which can be picked from configuration or environment

Let's see an example:

```
private int storeClosureDay = 7;

// This can be refactored to use a constant from Java

private int storeClosureDay = DayOfWeek.SUNDAY.getValue()
```

Again, there is no strict guideline around this to adhere to. But we must be cognizant about the fact the some will need to read and maintain this code later on. We should pick a convention that suits us and be consistent about it.

5.8. Code Comments

Code comments (/cs/clean-code-comments) can be **beneficial while reading code to understand the non-trivial aspects**. At the same time, care should be taken to **not include obvious things in the comments**. This can bloat comments making it difficult to read the relevant parts.

Java allows two types of comments: Implementation comments and documentation comments. They have different purposes and different formats, as well. Let's understand them better:

- Documentation/JavaDoc Comments
  - The audience here is the users of the codebase
  - The details here are typically implementation free, focusing more on the specification
  - Typically useful independent of the codebase
- Implementation/Block Comments
  - The audience here is the developers working on the codebase
  - The details here are implementation-specific
  - Typically useful together with the codebase

So, how should we optimally use them so that they are useful and contextual?

- Comments should only complement a code, if we are not able to understand the code without comments, perhaps we need to refactor it
- We should use block comments rarely, possibly to describe non-trivial design decisions
- We should use JavaDoc comments for most of our classes, interfaces, public and protected methods
- All comments should be well-formed with a proper indentation for readability

Let's see an example of meaningful documentation comment:

```
/**
* This method is intended to add a new address for the customer.
* However do note that it only allows a single address per zip
* code. Hence, this will override any previous address with the
* same postal code.
*
* @param address an address to be added for an existing customer
*/
/*
* This method makes use of the custom implementation of equals
* method to avoid duplication of an address with same zip code.
*/
public addCustomerAddress(Address address) {
}
```

5.9. Logging

Anyone who has ever laid their hands onto production code for debugging has yearned for more logs at some point in time. The **importance of logs can not be over-emphasized in development in general and maintenance in particular**.

There are lots of libraries and frameworks in Java for logging, including SLF4J, Logback. While they make logging pretty trivial in a codebase, care must be given to logging best practices. An otherwise done logging can prove to be a maintenance nightmare instead of any help. Let's go through some of these best practices:

- Avoid excessive logging, think about what information might be of help in troubleshooting
- Choose log levels wisely, we may want to enable log levels selectively on production
- Be very clear and descriptive with contextual data in the log message
- Use external tools for tracing, aggregation, filtering of log messages for faster analytics

Let's see an example of descriptive logging with right level:

```
logger.info(String.format("A new customer has been created with customer Id: %s", id));
```

6. Is That All of It?

While the previous section highlights several code formatting conventions, these are not the only ones we should know and care about. A readable and maintainable code can benefit from a large number of additional best practices that have been accumulated over time.

We may have encountered them as funny acronyms over time. They **essentially capture the learnings as a single or a set of principles that can help us write better code**. However, note that we should not follow all of them just because they exist. Most of the time, the benefit they provide is proportional to the size and complexity of the codebase. We must access our codebase before adopting any principle. More importantly, we must remain consistent with them.

6.1. SOLID

SOLID is a mnemonic acronym that draws from the five principles it sets forth (/solid-principles) for writing understandable and maintainable software:

- *Single Responsibility Principle*: Every **interface, class, or method we define should have a clearly defined goal**. In essence, it should ideally do one thing and do that well. This effectively leads to smaller methods and classes which are also testable.
- *Open-Closed Principle*: The code that we write should ideally be **open for extension but closed for modification**. What this effectively means is that a class should be written in a manner that there should not be any need to modify it. However, it should allow for changes through inheritance or composition.
- ***Liskov Substitution Principle* (/cs/liskov-substitution-principle)**: What this principle states is that **every subclass or derived class should be substitutable for their parent or base class**. This helps in reducing coupling in the codebase and hence improve reusability across.
- *Interface Segregation Principle*: Implementing an interface is a way to provide a specific behavior to our class. However, **a class must not need to implement methods that it does not require**. What this requires us to do is to define smaller, more focussed interfaces.
- *Dependency Inversion Principle*: According to this principle, **classes should only depend on abstractions and not on their concrete implementations**. This effectively means that a class should not be responsible for creating instances for their dependencies. Rather, such dependencies should be injected into the class.

6.2. DRY & KISS

DRY stands for “Don's Repeat Yourself”. This principle states that **a piece of code should not be repeated across the software**. The rationale behind this principle is to reduce duplication and increase reusability. However, please note that we should be careful in adopting this rather too literally. Some duplication can actually improve code readability and maintainability.

KISS stands for “Keep It Simple, Stupid”. This principle states that **we should try to keep the code as simple as possible**. This makes it easy to understand and maintain over time. Following some of the principles mentioned earlier, if we keep our classes and methods focussed and small, this will lead to simpler code.

6.3. TDD

TDD stands for “Test Driven Development”. This is a programming practice that asks us to write any code only if an automated test is failing (/java-test-driven-list). Hence, we've to **start with the design development of automated tests**. In Java, there are several frameworks to write automated unit tests like JUnit and TestNG.

The benefits of such practice are tremendous. This leads to software that always works as expected. As we always start with tests, we incrementally add working code in small chunks. Also, we only add code if the new or any of the old tests fail. Which means that it leads to reusability as well.

7. Tools for Help

Writing clean code is not just a matter of principles and practices, but it's a personal habit. We tend to grow as better developers as we learn and adapt. However, to maintain consistency across a large team, we've also to practice some enforcement. Code **reviews have always been a great tool to maintain consistency** and help the developers grow through constructive feedback.

However, we do not necessarily have to validate all these principles and best practices manually during code reviews. Freddy Guime from Java OffHeap (https://www.javaoffheap.com/2019/10/episode-47-microsoft-flexing-its-java-muscle-javafx-is-alive-and-well-and-would-you-approve-my-low-quality-pr.html) talks about

the value of automating some of the quality checks to end-up with a certain threshold with the code quality all the time.

There are **several tools available in the Java ecosystem**, which take at least some of these responsibilities away from code reviewers. Let's see what some of these tools are:

- Code Formatters: Most of the popular Java code editors, including Eclipse and IntelliJ, allows for automatic code formatting. We can use the default formatting rules, customize them, or replace them with custom formatting rules. This takes care of a lot of structural code conventions.
- Static Analysis Tools: There are several static code analysis tools for Java, including **SonarQube (/sonar-qube)**, **Checkstyle (/checkstyle-java)**, **PMD (/pmd)** and **SpotBugs (https://spotbugs.github.io/)**. They have a rich set of rules which can be used as-is or customized for a specific project. They are great in detecting a lot of **code smells (/cs/code-smells)** like violations of naming conventions and resource leakage.

8. Conclusion

In this tutorial, we've gone through the importance of clean coding principles and characteristics that clean code exhibits. We saw how to adopt some of these principles in practice, which developing in Java. We also discussed other best practices that help to keep the code readable and maintainable over time. Finally, we discussed some of the tools available to help us in this endeavor.

To sum up, it's important to note that all of these principles and practices are there to make our code cleaner. This is a more subjective term and hence, must be evaluated contextually.

While there are numerous sets of rules available to adopt, we must be conscious of our maturity, culture, and requirements. We may have to customize or for that matter, devise a new set of rules altogether. But, whatever may be the case, it's important to remain consistent across the organization to reap the benefits.