**CLEAN CODING TECHNIQUES**

**Refer this link for more detailed explanation about each category of Code smells:**

1. [**https://sourcemaking.com/refactoring/smells**](https://sourcemaking.com/refactoring/smells)
2. [**https://refactoring.guru/refactoring/smells**](https://refactoring.guru/refactoring/smells)
3. [**https://refactoring.guru/refactoring/techniques**](https://refactoring.guru/refactoring/techniques)

**Definition**: Clean Code makes the code more readable and hence easily maintainable. [ easily readable, testable, Modifiable, Scalable and less prone to errors]

**Characteristics of Clean Code**

1. **Simple**: the code should be made as simple as possible.
2. **Maintainable**: It should be maintainable in the long run as many different developers can work on that code.
3. **Testable**: It should be easily testable and less prone to errors.
4. **Readable**: it should be easily readable.

## Symptoms of bad code (Unclean code)

Any code that leads to problems in the long run can be referred to as Bad Code. The problems can be:

1. Hard to read code
2. Unnecessarily complex
3. Not easily testable
4. Hard to modify

## To write Clean Code in Java:

### **Structure**

There is no such rule to follow a particular structure for your project but it does not mean that you should not.

src

├── main

│ ├── java

│ └── resources

└── test

├── java

└── resources

* **1. *src/main/java*:** It contains the source files of your project. You can identify it as the sub-folder is *main* in this case.
* **2. *src/main/resources*:** It contains the resource files of your project. You can identify it as the sub-folder is *main* in this case also.
* **3. *src/test/java*:** It contains the test source files of your project. You can identify it as the sub-folder is *test* in this case.
* **4. *src/test/resources*:** It contains the test resource files of your project. You can identify it as the sub-folder is *test* in this case also.

### **2. Proper Naming**

In Java, we have to name variables, classes, methods, etc., hence we should always give them a name that they can relate to.

**For e.g.** If you want to create a variable for counting, name it 'counter' instead of naming it 'c' or something like that. **You should always avoid naming variables with a single character like 'a', 'b', 'c', etc.**

### **3. Source File Structure**

Normally the elements placed in the source file are in the following order:

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

### **4. Whitespaces and Indentation**

Whitespaces can help you make your code very readable that is why it is advised to add whitespaces whenever you can add them.

In this function, whitespaces and indentation are not taken care of.

**public int sum(int num1,int num2,int num3)**

**{**

**int sum=num1+num2+num3;**

**return sum;**

**}**

While in this function, whitespaces and indentation are properly taken cared.

**public int sum (int num1, int num2, int num3)**

**{**

**int sum = num1 + num2 + num3;**

**return sum;**

**}**

### **5. Method Parameters**

Avoid adding more than 3 parameters if possible as it can create unnecessary confusion for other developers or programmers who are going to deal with that code in future.

You can also go for refactoring, if possible, like we have done in the example given below:

In Programming, **Refactoring** is a technique used for restructuring an existing code without modifying the external behaviour of code.

***// before***

**private void employeeDetails(String name, String age, String awards, String ctc, String experience).**

***// after***

**private void employeeDetails(String name, Details employeeDetails).**

### 6. Avoid Hardcoding

Hardcoding is the practice of adding the data directly into the source code of a program instead of getting the data from external sources. Remember that heavy hardcoding can make your program difficult to maintain and it also leads to bugs in your program.

 if you create a program that calculates the days and time remaining for the new year to arrive, you need the current date and time for that. Instead of hard coding the current date and time you can get the current date from various classes in java that provide you the current date and time.

### 7. Code Comments

Commenting is also considered as one of the best practises to make the code easily understandable.

In Java, two kinds of comments are allowed:

* Documentation comments - The target audience is the users of codebase
* Implementation comments - The target audience is the developers that work on the codebase

### **8. Logging**

Logging is one of the most important factors in clean coding. A well logged code is easy to debug and can save hours of work for a developer.

### **9. SOLID**

 SOLID is a mnemonic acronym for the five principles given below:

1. Single Responsibility Principle
2. Open/Closed Principle
3. Liskov Substitution Principle
4. Interface Segregation Principle
5. Dependency Inversion Principle
   1. **Single Responsibility Principle:** This principle states that, In Java, each and every interface, class, or method that is defined by us should be very specific which means that it should have a single responsibility. In simple words, it should do only one task and do it really well. By following this principle consistently, the methods and classes become smaller hence easily testable.
   2. **Open-Closed Principle:** This principle states that our code should ideally be open to any kind of extension and closed for any type of modification. In simple words, our code should be easy to extend but good enough that no modification is required in our code.
   3. **Liskov Substitution Principle:** This principle states that the objects of a superclass should be replaceable with the objects of its subclasses but without ca using the application to break.
   4. **Interface Segregation Principle:** Defined by Robert C., the goal of this principle is to reduce the side effects from the code and frequency of required changes by splitting or dividing the software into many independent parts or interfaces.
   5. **Dependency Inversion Principle:** According to this principle, classes should only depend on the abstractions and not on the implementations. In simple words, High-level(complex) modules, which provide complex logic to your software should be easily reusable and they should not get affected by the changes in the low-level modules which generally provide utility features.

### **10. DRY & KISS**

**DRY** is an acronym that stands for "Don't Repeat Yourself". DRY emphasises on not repeating the same code again and again. In simple words, the main motive behind this principle is to make the code reusable.

**KISS** is an acronym that stands for "Keep It Simple, Stupid". As the words of this acronym suggests, this principle states that the code should be made as simple as possible which is of course to avoid the understanding confusions. If this principle is followed properly, the whole code eventually becomes easy to understand i.e., simple.

**Example for Clean Code:**

**public class Employee {**

**private String employeeName;**

**public String getEmployeeName() {**

**return this.employeeName;**

**}**

**}**

Why the above is called as Clean Code?

1. The variable, class and method are named according to the naming convention like class has the first letter capital, variable and method are named as per camelCase naming convention and they also reflect a specific meaning according to their name.
2. We made sure that our method is doing only one thing i.e., getting employee name.
3. Whitespaces and indentation are also properly utilized.

**Code Smells**

Code Smells are not the bugs of the program. With code smells too, your program might work just fine. They do not prevent the program from functioning or are incorrect. They just signify the weakness in design and might increase the risk of bugs and program failure in the future.

Code Smells motivates for Code Refactoring.

### **Types of Code Smells**

* Within Classes
* Between Classes

### Code Smells Within Classes

1. **Comments:**Yes, Comments are code smells too. It is like deodorant to the code. If comments explain a few lines of complex code or a complex expression, it should be made as a complete another function/ sub-expression respectively.
2. **Long Method:**A long method contains too many lines of code. Any code with more than 25 lines of code should make you question. It could be solved using refactoring techniques like changing expression into sub-expression, complex codes into a new function in order to make a function or method a small and easy to read without changing its functionality.
3. **Long Parameter List:**Any function with more parameters is obviously more complex. One of the thumb rules is to use a maximum of 3 to 4 parameters in a function.
4. **Large Classes:** A class contains many methods/lines of code/fields is considered a code smell.  Refactoring techniques could be done like, extract class, extract subclass, extract interface, duplicate observed data.
5. **Duplicate Code:**When a code is repeated for more than two times, merge it into a function or a method of the class. If the same code is found in two or more classes, using the extract method we can refactor the code smell.
6. **Dead Code:**The code that is not being used. A function with no calls or the condition that never occurs. Delete unused code and unnecessary files.

### Code Smells Between Classes

1. **Data Classes:**A class that just store data and no methods. These classes don’t contain any functionality. They can’t independently operate on the data that they own. A class should contain both data ***and***methods. Use global or local variables to refactor this code smell.  If the data class contains public data, we can use the Encapsulation Method to hide it.

Other refactoring techniques to solve this are: Move, Extract, and Remove Methods.

1. **Data Clumps:** Data that looks similar maybe belongs to the same class. Consider using a superior class. For example, parameters for connecting to a database. These clumps should be merged into a new class. If data that is repeating data comprises the fields of a class, use extract class to move the fields to their own class or create their own class.

Other Refactoring Techniques to deal with this code smells are:  Introducing parameter objects, Preserve the whole objects, etc.

1. **Alternative Classes with Different Interfaces:** If two classes are similar on the inside, perhaps they can be modified to share a common interface. For example, the programmer who created one of the classes is might not aware that a similar class already existed or created by other programmers of the team. Refactoring techniques to deal with them are: Rename Method, Move, Add Parameter, Parameterize Method.

1. **Refused Bequest:** A class inherits from other classes but not using the inheritance methods of its parent class. For example, assume a class, body, human class, and animal class both are inherited from the class body. A car has a body too, it could inherit from the body class too, but that will not make any sense. Refactoring Techniques to deal with this are: Replace inheritance with the delegation and Extract superclass.

1. **Lazy Class:** A class doesn’t do enough to earn your attention; it should be deleted because it can cost your time and money both.  For Example, a class that was designed to be fully functional but after some refactoring and change in code, it has become of no use or a little use maybe.   
   Inline Class and Collapsing Hierarchy can be used to make the code size smaller, easy to understand and maintain.

1. **Shotgun Surgery:** A single fire causing multiple shots. A single change in classes may lead to cascading changes in several related classes. This can happen after the overzealous application of Divergent Change. Use of Move Method and Move Field could be done to make the code easier to maintain and less duplicated.

There are many known **code smells that have been categorized** as follows:

1. **Bloaters**

Bloaters are code, methods and classes that have increased to such enormous proportions that they are hard to work with. Usually, these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

1.Long Methods

2.Data clumps

3.Large Class

4.Long Parameter list

5.Primitive Obsession

* Use of primitives instead of small objects for simple tasks (such as currency, ranges, special strings for phone numbers, etc.)
* Use of constants for coding information (such as a constant USER\_ADMIN\_ROLE = 1 for referring to users with administrator rights.)
* Use of string constants as field names for use in data arrays.

1. **Object Oriented Abusers**

All these smells are incomplete or incorrect application of object-oriented programming principles.

1. Alternative classes with different interfaces.

2. Refused Bequest

3. Temporary field

4. Switch statements

1. **Change Preventers**

These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result.

1. Divergent Change

2. Parallel Inheritance Hierarchies

3. Shot gun Surgeries

1. **Dispensable**

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

1. Comments

2. Data Class

3. Lazy Class

4. Duplicate Code

5. Dead Code

6. Speculative Generality

1. **Couplers**

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

* 1. Feature Envy
  2. Incomplete Library Class
  3. Middle Man
  4. Inappropriate Intimacy
  5. Message Chains

**Refer this link for more detailed explanation about each category of Code smells:**

* 1. [**https://sourcemaking.com/refactoring/smells**](https://sourcemaking.com/refactoring/smells)
  2. [**https://refactoring.guru/refactoring/smells**](https://refactoring.guru/refactoring/smells)
  3. [**https://refactoring.guru/refactoring/techniques**](https://refactoring.guru/refactoring/techniques)