# Clean Code Tips

**Give your variables, constants and method names clear and descriptive names.**
They should clearly reveal the intentions of the code and not be mysterious or meaningless (not too short and not too long). You should not have to read the implementation of a variable or method to understand what it is doing. If a method has a meaningless name this is normally as a result of violating the single responsibility principle and being too long. You should refactor the code to ensure each method is clear and only carrying out one task. In Intellij you can use the refactoring tools to change variable names and extract methods without breaking your code. Use camelCase for variables and PascalCase for methods and classes.

**Duplicated Code -** Don't repeat yourself (DRY) simply extract th duplicated code to a method and replace it with method calls

**Long methods -** more than 15 lines of code are hard to understand, hard to change and hard to reuse. To refactor following the single responsibility principle and extract methods from lines of code that are logically related giving them meaningful names making the original method much easier to read. this will increase the cohesion of your code and reduce coupling. additionally if the method you are extracting is not related / cohesive with the rest of the classs then you should extract it to another class. To shorten methods further user the other refactoring tips mentioned in this page.

**Method Signatures** as mentioned ensure they are descriptive. Other notes include if there is a boolean value in the parameter list often time this method will be doing two things dependent on the value of that boolean and should be refactored to two methods. Ensure the return type, method name and parameters all present an accurate and descriptive picture of what the method is doing. If you have a long parameter list consider creating a class which can be used to replace several of the parameters e.g. if your method contains dateFrom and dateTo parameters then you could refactor this to take a date range class. This class can then be reused in other methods requiring a date range. In short if you have 3 or more parameters list try to encapsulate the logically related parameters with a class / object and then reuse in other method requiring similar parameters.

**Inline variables in methods** - within methods often it is more descriptive to utilise inline variables rather than declaring the variable at the top of the method and using it farther down. This stops the context of the code from switching making it easier to read. If the variables have to be initialised before they are used then they should be initialised close to where they are used rather than at the top of the method.

**Magic numbers**- should not be used e.g. in logic – if (int status == 1) { do something ... } else if (status == 2) { do something else ...}. As a reader we have no idea what the magic status number represents and instead should use a descriptive constant - const int draft = 1 and then use the draft constant in the logic. Of if you are going to use these statuses outside the method then you should use an Enum

public enm DocumentStatus { Draft = 1, Lodged = 2 }

if(DocumentStatus status == DocumentStatus.Draft) { do something ... } else if (status == DocumentStatus.Lodged) { do something else ...}.

**Comments** - often if you have to write a lot of comments it is a good sign that you should rewrite / refactor your code so that it is clear in its intention to the reader. When necessary your comments should attempt to explain "why" and "how" and not "what" the code is doing which should be obvious/ E.g. getCustomers does not need the comment //gets a list of customers - because this is obvious and can become outdated / a lie if the logic is updated and not the comment. There is no need to comment out code, if it is not needed then delete it and retrieve it from version control if necessary. There are examples of good comments such as TODO: comments which can be retrieved from the TODO list in most IDEs. If you see problems in the code it is a good idea to comment them this TODO comments. Also if there is a constraint that is not obvious to the reader such as database issues of legacy code then this code result in some code that is better explained as to why it is written that way and how it is achieving its purpose.

**Switch statements / large blocks of if else logic -** when used for business logic can often violate the Open Closed Principle, meaning that when you need to add a new business rule to a switch statement you have to modify the class rather than extend its functionality. This will cause cascading changes through any of the classes that ue the business rules / switch statement. To increase reuseability and maintainability of the code we can utilise polymorphism. To do this you can utilise the State pattern and introduce an interface or abstract class, for instance if your you switch statement operates on a customer type field of a customer and then performs some action such as generating a bill based on the customer type field. In this case toy shouuld consider creating an abstract customer class with an abstract generateBill method. You can then create differnt types of customers which exxtend this class such as payMonthlyCustomer or payAsYouGoCustomer. These child classes can then implement that logif that was in the switch statement / if else logic in the generateBill implementation. This state pattern is an example of polymorphism in action and you can read more about it here : State Pattern and see also the Strategy Pattern

**Nested Conditionals** - if else statements within if else statements can often be simplified to increase the readibility and maintainability of the code. The ternary opertaor is one option for this or simply rewriting your logic. Before refactoring the logic it is important to hacve unit tests written to ensure that you do not break the code when trying to make it more straight forward. Often the logic to enter an if block in the parentheses can be extracted to a method wth an expressive name for what the logic is trying to achieve. Beloe is an example of refactoring nested logic :

**Refactoring Nested Conditionals**

```
if(a) {
        if(b) { isValid = true; }
} if(c) {
        if(b) { isValid = true; }
}

Can be simplified to :

if(b && (a || c)) { isValid = true; }

Which can be further simplified to :

isValid = (b && (a || c));
```