Clean Code

Meaningful Names

Explicit

Name of an element should tell the purpose of its existence, how will it be used, where will it be used etc.

```
List<User> userList;
```

What is the purpose of creating this list ? Where exactly will it be used?

```
for(int i=0 ; i <=19 ; i ++){
     chooseARandomPlayerAndAddToTeam();
}
```

Do we know why exactly are we iterating till 19? why not more or less?

```
List<User> houseOwners;

for(int i=0 ; i <= MAX_NUMBER_OF_TEAM_MEMBERS ; i ++){
     chooseARandomPlayerAndAddToTeam();
}
```

Distinct and Clear

Name of elements should not be generalized and should be distinct such that it distinguishes itself from similar elements

```
Class UserService{
     .....
}
```

We know that it does something related to User objects. But what exactly ?

```
function searchGyms(double lat, double long, long distance)
function searchGyms(Sting zip, long distance)
```

When do we need to call which method is not explicit from the name of the method. One has to go through the parameters to find the exact match.

```
Class UserAuthenticationService{
     ......
}

function searchGymsForLatLongWithDistance(double lat, double long, long
distance)
function searchGymsWithinGivenDistanceFromZip(Sting zip, long distance)
```

Class Names

Class names should be nouns e.g. Account, Vehicle, Employee

Function Names

Function names should be verbs e.g. transferMoney, validate, authenticate etc.

Factory Methods vs Constructors

Use Factory Methods instead of Constructors e.g. Employee.

createWithDefaultValues()

One Word Per Context

Pick one word per context and the whole team should agree to it e.g. don’t use findByName and fetchByEmail together in the same project.

Don’t Add Unnecessary Prefixes

Don’t prefix your class names with name of the modules eg BBAccountCreationService, BBUserAuthenticationService.

Classes

Small

Try to keep your classes small. Size of the class is not measured in terms of lines, but it is measured in terms of responsibilities.

Single Responsibility Principle

A class should have only one reason to change. So, if there are two reasons for a class to change, it needs to be then split into two different classes.

Cohesion

The variables and functions should be both interdependent on one. Each method should be using at least one instance variable of the class.

Functions

Small

Try to make your functions as small as possible. Extracting smaller functions from a bigger function adds “Clarity” and “Documentation”

One Thing

Your function should only be doing one thing at time. This is one of the most confusing statements. e.g.

- There Should always be a call to function inside an if-else-if statement
- Switch statement should be the first one in a function
- Try - Catch - Finally should be the only thing that should be in one method

One Level of Abstraction Per Function

We want all the statements in a function to be of same level of abstraction

e.g.

When I place an order on a website following are the things that are done by the web site

Functions

1. Validate if Credit Card can be used for purchase
    a. validate that the card exists
    b. verify that user has enough balance
2. Charge the credit card for the purchased product
    a. Get Total payable amount.
        i. Get product price
        ii. Get delivery charges if applicable
        iii. Calculate taxes ;
        iv. Get Total amount by adding price + delivery charges + taxes;
    b. Ask the bank to charge the total amount
3. Save the order information in database
    a. Save the Order Number and Transaction Id
    b. Save the Billing and Shipping Address for this order
4. Process for home delivery

```
processOrder(Product product, CreditCard creditCard){
   validateCreditCardIsCapableOfPurchasing(product,creditCard);
    Order order = chargeForPurchaesdProduct(product,creditCard);
    saveOrderInformationInSystem(product, order);
    processForHomeDelivery(creditCard.getUser());
}

validateCreditCardIsCapableOfPurchasing(Product product,
CreditCard creditCard) {
    checkIfCardIsValid(creditCard);
    checkIfCardHasEnoughMoney(creditCard, product.getPrice());
}

chargeForPurchaesdProduct(Product product, CreditCard creditCard)
{
    double totalPrice = getTotalChargablePriceProduct(product);
    Bank bank = creditCard.getAssociatedBank();
    bank.makeNewPurchaseTransaction(creditCard, totalPrice);
}
```

Separation of Command and Query Functions

Functions should either be performing some operation should either be answering, but not both

```
Order order = chargeForPurchaesdProduct(product,
creditCard);
```

```
chargeForPurchaesdProduct(product,creditCard);
Order order = getNewlyPlacedOrder(product,creditCard)
```

Function Arguments

Lesser Arguments

Our target should be to reduce the number of arguments to as less as possible. Following are the reasons for this

- Function with no argument is easier to understand than function with one argument. Lesser the arguments, easier is the function to understand.
- Testing becomes difficult, especially when you have multiple number of same type of arguments.
- Flag arguments shows that the function is doing more than one thing.

Output Arguments

It is very difficult to understand output arguments as we generally expect the output value to be returned by a return statement

```
includePageResponse(outputStream)
```

```
byte[] finalHtmlPage = getPageResponse()
outputStream.set(finalHtmlPage);
```

Pass & Returning “null”

You should not return a null from you function. If you don’t have anything to return, return a dummy/default/special Object.

Data Structures and Objects

Procedural Programming

Functions are the core of Procedural Programming. These functions typically take some input, do something, then produce some output. Ideally your functions would behave as "black boxes" where input data goes in and output data comes out.

Object Oriented Programming

In Object Oriented Programming data and related functions are bundled together into an "object". Ideally, the data inside an object can only be manipulated by calling the object's functions. This means that your data is locked away inside your objects and your functions provide the only means of doing something with that data.

OOPS vs Procedural Programming
<==>
Objects vs Data Structures

Code Smells

Inappropriate Comments

We should just be adding technical details to the comments which cannot be be inferred straight away and nothing fetched from version control or some other tracking tool.

Obsolete Comment

Sometimes comments become obsolete as they are not changed when we change the method signature or functionality.

Redundant Comment

One example of this is just adding of comments for Java docs which don’t tell anything extra about the methods or its arguments.

Commented-Out Code

There is no use of commenting out code and keeping it as backup. The previous version i.e. the commented out code can always be retrieved from Version Control.

Environment Setup

If you have to do more than two three steps to have the complete setup done for a project.

Code Smells Related to Functions

Long Functions

Too many arguments

Output Arguments

Flag Arguments

Dead Function

Code at Wrong Level of Abstraction

Duplication

Multiple if/else switch statements