***Low Level Design***

# **Solid Principles Java:**

## **Single Responsibility**

As we know at some point of time the application will grow we can reduce their complexity and save ourselves a lot of headaches further that can create the use and we can form the reusability components

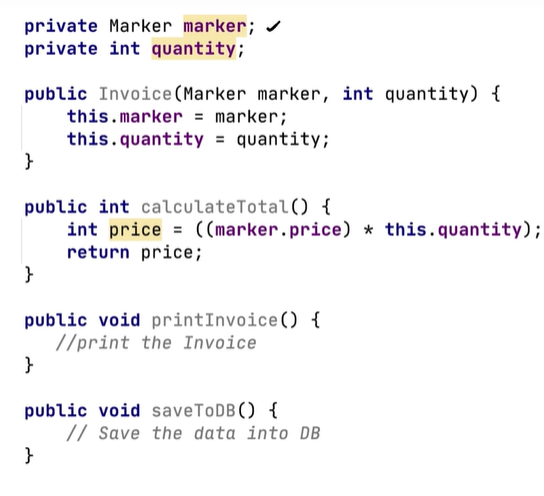
A class should only have one responsibility

Let say if we have a class BOOK then this class contains all the data fields method and logic related to book only

If we have a user class the all the data related to the user must be present there should not be a method for user Authentication & Authorization

For these Two we should have separate classes “UserAuthentication” & “UserAuthorization”.

These principles help us to avoid the duplicate code !!

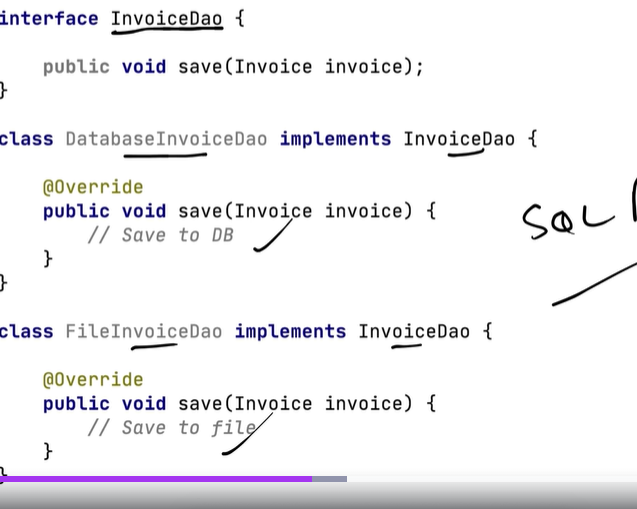
****

**Single Responsibility says one reason to change   
  
If in future if we introduce the GST then calculation logic will change**

**saveToDB() can also cause the issue as in future we need to save to the file as well   
  
  
So we will break our class into other classes   
  
1. InvoicePrinter -> Responsible to print the class**

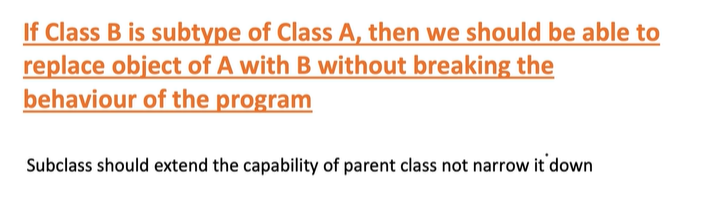
**2. InvoiceDao -> will save the db and we can introduce other methods related to the same**

## **Open / Closed Principle**

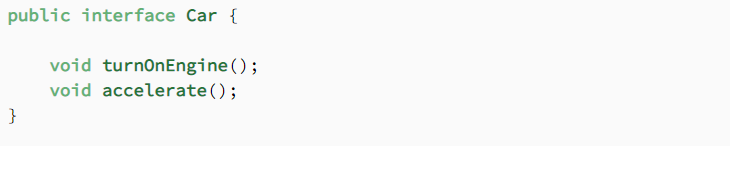


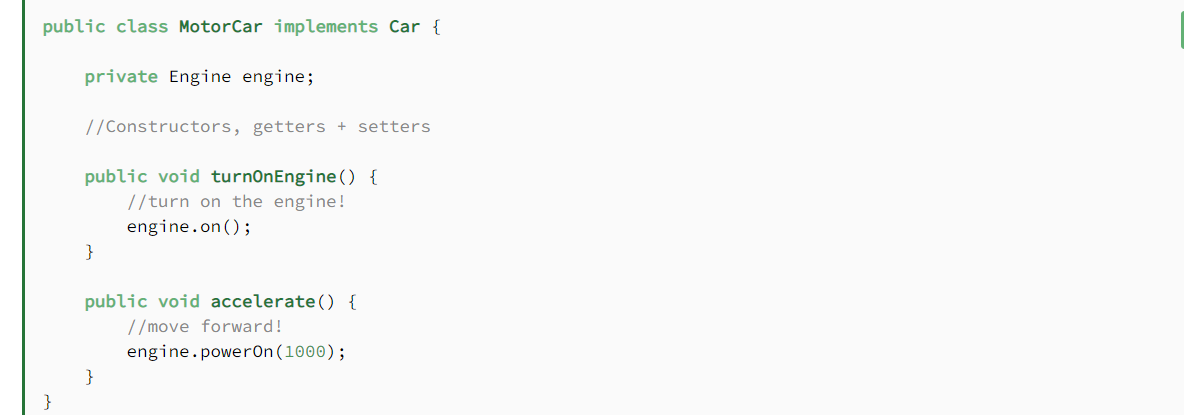
Before that we have the invoiceDao class which will handle the functionality but in future we need to add more cases so we need to support the backward computability of the code and It must be extendable with new features as well.

## **Liskov Substitution principle**

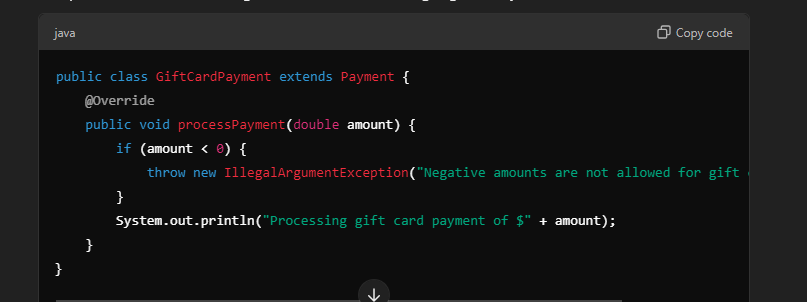


If class A is a subtype of Class B we should be able to replace B with A without disrupting the behaviour of our program





Here I have define the payment Abstract class along with the implementation of that payment class with other classes

1. Let Say If One of the child classes changes the behaviour of the payment by throwing the exception and it might break the code as we are not explicitly handling the exception
2. 

Payment pay = new GiftCardPayment();

pay.processPayment(-50);

this will break the logic of the code as it will through the exceptions.

If we have 2 classes ClassA & ClassB

## **Interface Segmented Principle:**

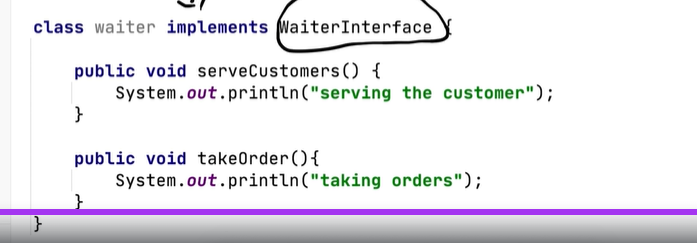
Interfaces should be such that client should not implement the unnecessary methods

I will not club the methods into one interface, as per functionality will segregated the methods and by doing this I am not forcing my client to implement everything



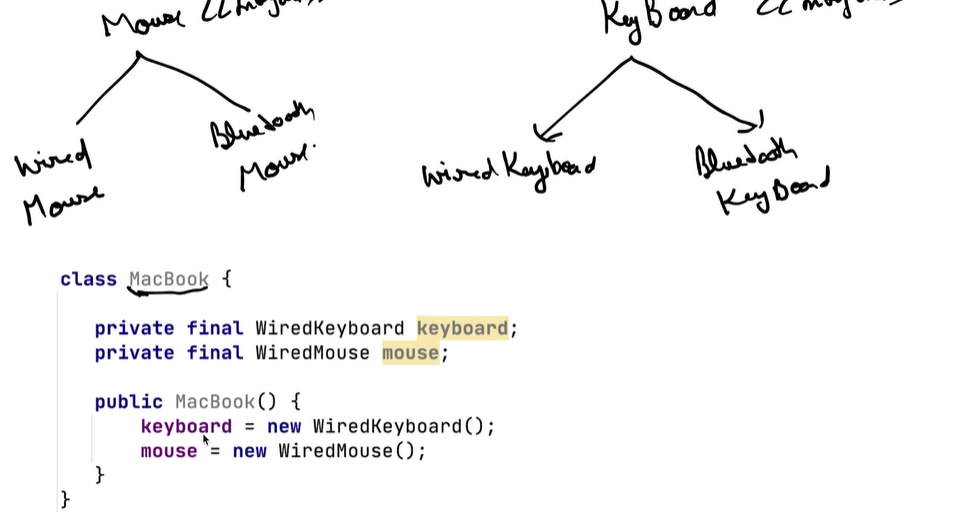
Here waiter class is unnecessary implementing the cookFood() & washDish() methods so we need to granularize these methods into more interfaces

Segmented interface are below   
  

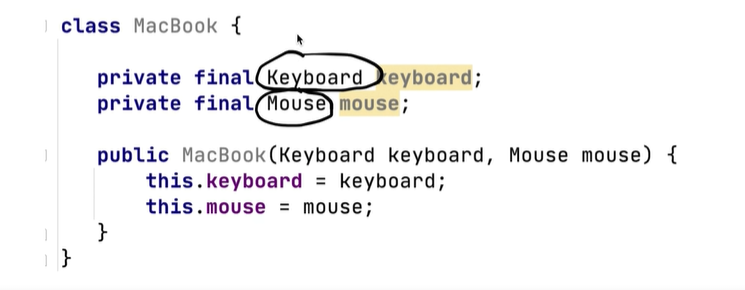



## **Dependency Inversion principle:**

**Class should depends on interfaces rather than on concrete classes**.



MacBook is responsible for object creation and maintenance

And here we are providing the object of concrete classes in future if we want to enhance the functionality then we are not able to do the same   
  


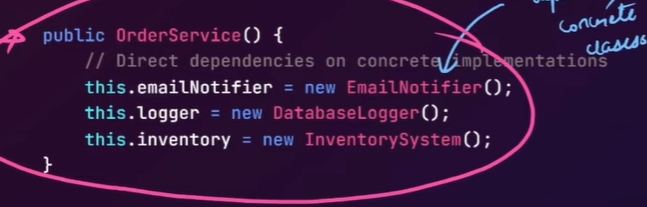
**Example 2: Ecommerce notification system:**

**Class OrderServices{**

**Private EmailNotifier emailNotifier;**

**Private DatabaseLogger dbLogger;**

**Private InventorySystem inventory;**

****

**It** is tightly coupled with EmailNotifier and other classes and in any changes I Need to change the entire class



I will pass the Notification type at the time of calling the order services and maintainability and scalability is there along with the proper test cases,

And here we are not creating the object of the class it’s injecting through the client side.

Here we are assigning the object using constructor injection so we are not dependent on the macbook to create the object calling class will pass the value which type of keyboard and mouse the want

## LSP Solutions:

How to avoid this problem and provide the solution for the same.

# DRY: (Don’t repeat yourself)

Get logic in one place and call the function from any place as per requirement.

SubmitButton: classs :onClick

CancelButton: Class :onClick()

Both classe have there onClick so we will create abstractClass click and extends the same and remove the delicacy

# Strategy Design Pattern:

Is a relation -|> is used for inheritance

* This represent has a relation

# State Design pattern(Vending machine):

Vending machine can have the products

1. Product code
2. Product price
3. Product quantity

Cash flow:

1. Insert cash/coin
2. (which product we want)Product menu button
3. then insert case and refund the change
4. Refund flow through cancel button