SOLID Principles

SOLID Principles is a coding standard that all developers should have a clear concept for developing software properly to avoid a bad design. It was promoted by Robert C Martin and is used across the object-oriented design spectrum. When applied properly it makes your code more extendable, logical, and easier to read.

There are total 5 principles which are totally based on the concept of OOP (Object Oriented Programming):

(1) Single Responsibility Principle
(2) Open/Closed Principle
(3) Liskov’s Substitution Principle
(4) Interface Segmenting Principle
(5) Dependency Inversion Principle

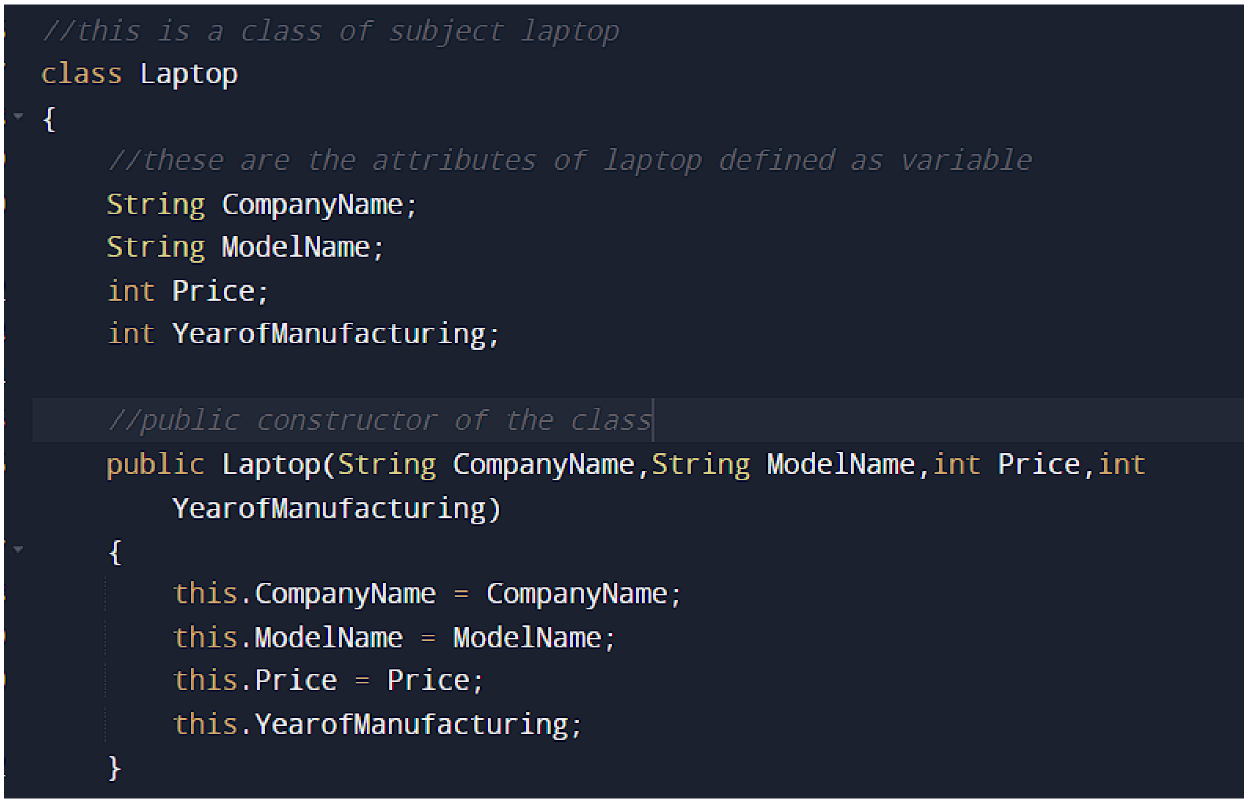
For the time being we will start with the first principle, we will try to understand the principles with simple examples.

Single Responsibility Principle

Single Responsibility principle is based on the concept of OOP. It suggests that

‘One class should have only one reason to change’

We will understand this principle using a simple example. Suppose there is a class of Laptops.

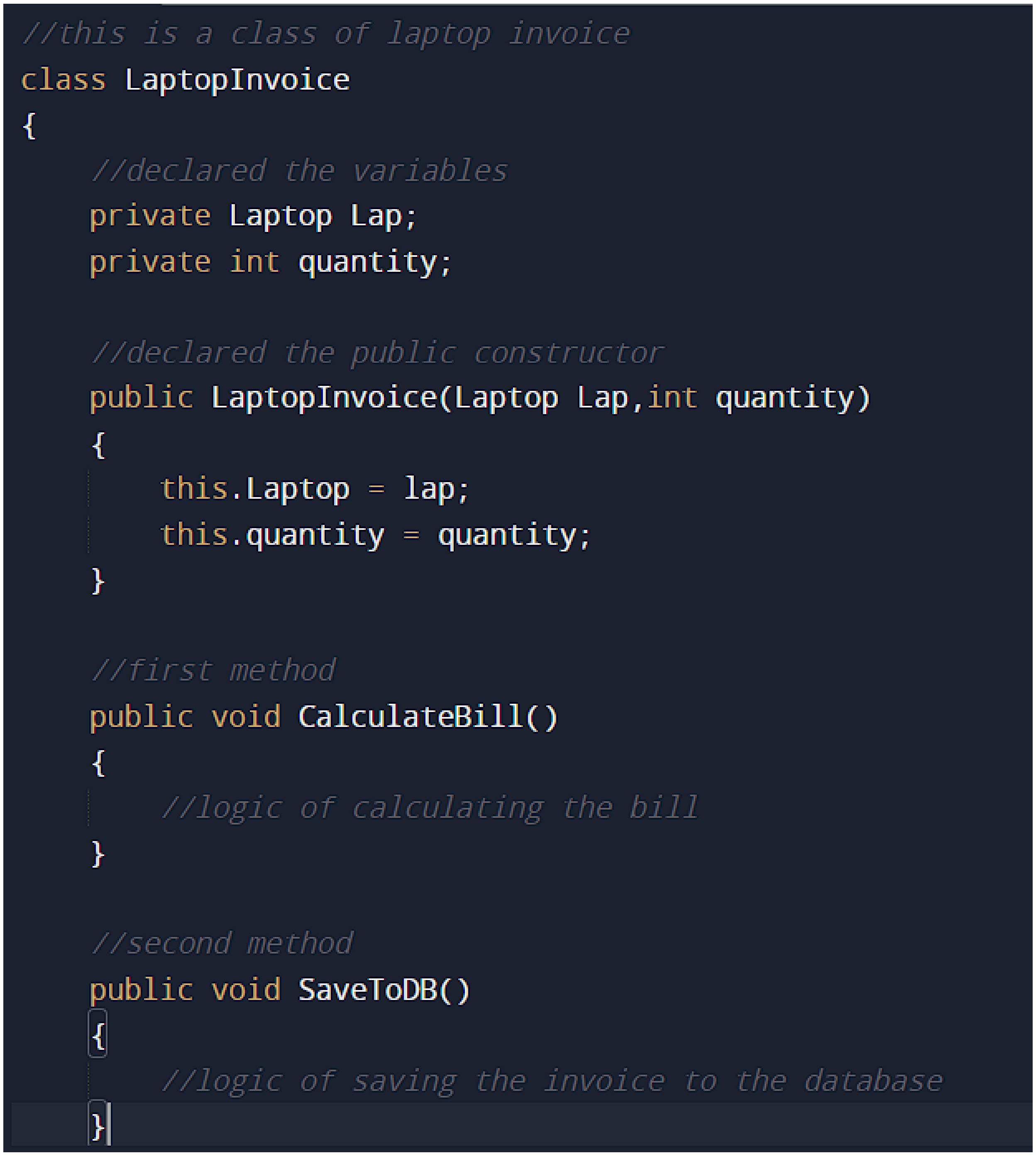


```
//this is a class of subject laptop
class Laptop
{
    //these are the attributes of laptop defined as variable
    String CompanyName;
    String ModelName;
    int Price;
    int YearofManufacturing;

    //public constructor of the class
    public Laptop(String CompanyName,String ModelName,int Price,int
        YearofManufacturing)
    {
        this.CompanyName = CompanyName;
        this.ModelName = ModelName;
        this.Price = Price;
        this.YearofManufacturing;
    }
```

Now here we can see some of the attributes of Laptop defined as the variables and a public constructor having some parameters passed as input and setting the value of the variables of the class.

For Laptop, we need to create a class named invoice where we will implement the details of invoicing the laptops from the business perspective.



```
//this is a class of laptop invoice
class LaptopInvoice
{
    //declared the variables
    private Laptop Lap;
    private int quantity;

    //declared the public constructor
    public LaptopInvoice(Laptop Lap,int quantity)
    {
        this.Laptop = lap;
        this.quantity = quantity;
    }

    //first method
    public void CalculateBill()
    {
        //logic of calculating the bill
    }

    //second method
    public void SaveToDB()
    {
        //logic of saving the invoice to the database
    }
```

Now we can see that this class of **LaptopInvoice** has the public constructor of its own name having some parameters to pass and value assignment.

There are two methods which we have implemented in the above **LaptopInvoice** class

(1) **CalculateBill** - which will calculate the bill from the quantity and laptop which is being derived from the above class.
(2) **SaveToDB -** which will save the invoice into the database.

Now let us see how this example fits the **Single Responsibility Principle.**

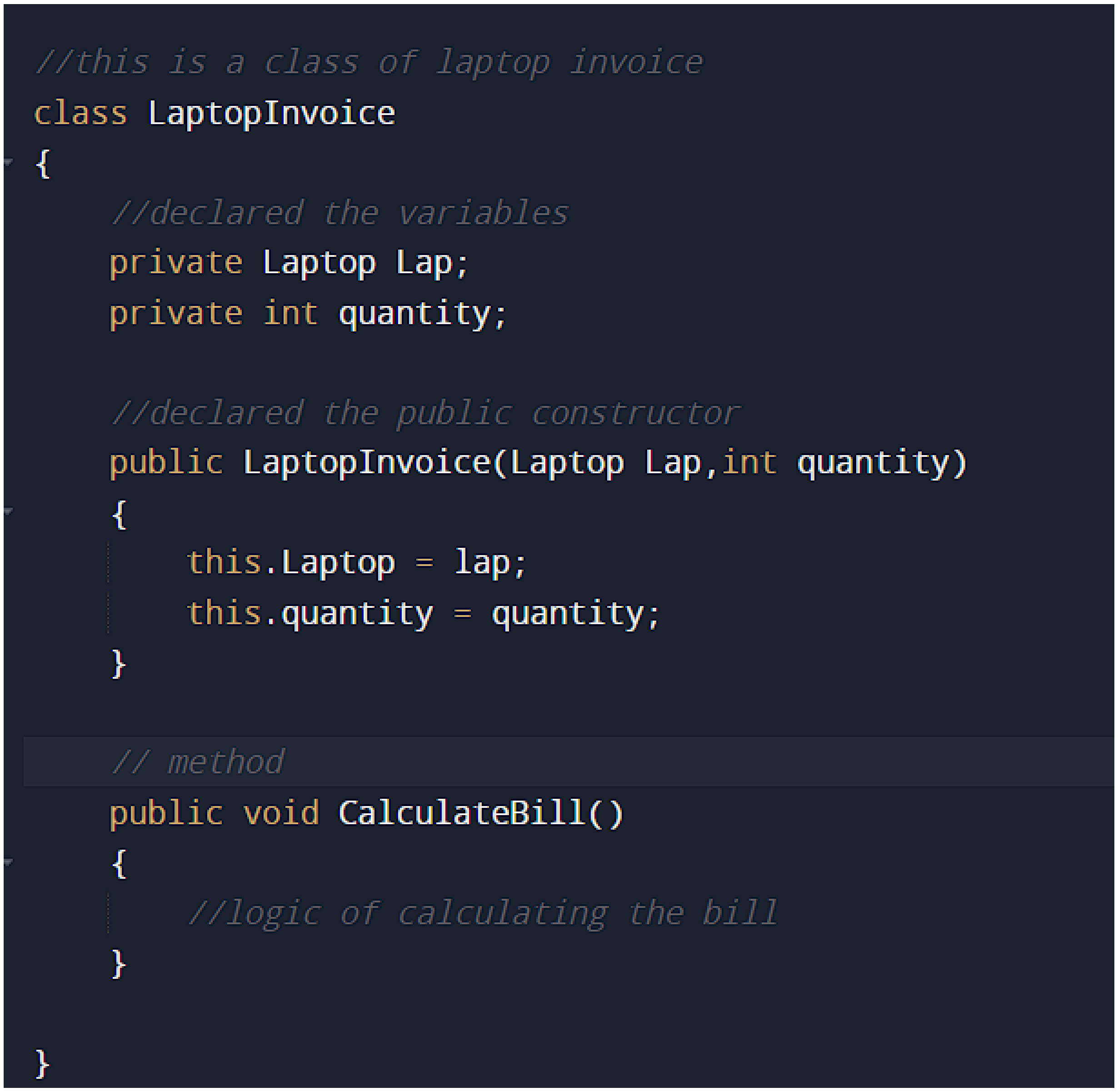
As we can see that, this class implements 2 methods in which I can change the logic as I can do some changes in first method for calculating the bill or I can do the changes in writing the logic for saving the invoice in the database as per example, I can change the code for storing the invoice in the NoSQL database.

So We can say that there are 2 reasons that this class can change and which is definitely a violation of the Single Responsibility Principle.

Now there is a question of how we can transform this condition to implementation of the SR Principle.

There is a way which is demonstrated as per below:

We can convert this single class into multiple classes implementing the single purpose of implementation.



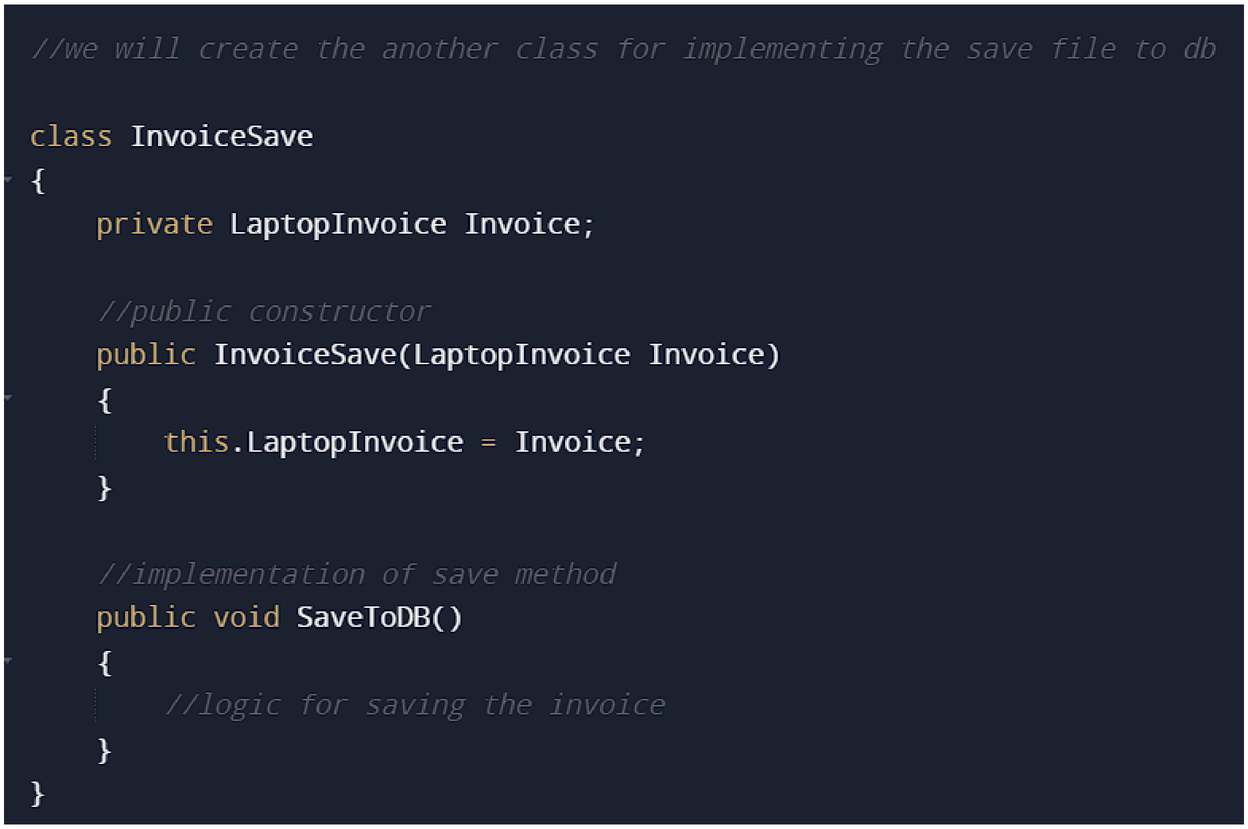
```java
//this is a class of laptop invoice
class LaptopInvoice
{
    //declared the variables
    private Laptop Lap;
    private int quantity;

    //declared the public constructor
    public LaptopInvoice(Laptop Lap,int quantity)
    {
        this.Laptop = lap;
        this.quantity = quantity;
    }

    // method
    public void CalculateBill()
    {
        //logic of calculating the bill
    }

}
```

This is the first class which will implement the functionality of calculating the bill



```
//we will create the another class for implementing the save file to db

class InvoiceSave
{
    private LaptopInvoice Invoice;

    //public constructor
    public InvoiceSave(LaptopInvoice Invoice)
    {
        this.LaptopInvoice = Invoice;
    }

    //implementation of save method
    public void SaveToDB()
    {
        //logic for saving the invoice
    }
}
```

This is the second class in which we will implement the saving of the invoice to database methods.

The benefit is that whenever I need to change the code or any functionality, it won't affect the other class as every class has only one reason and one responsibility to implement.

This is how we can understand the **Single Responsibility Principle** with simple examples.

Advantages of using SOLID Principles:

(1) Duplicate Code avoidance
(2) Easy to maintain
(3) Easy to understand
(4) Flexible Software Development