SOLID Principles

SOLID Principle

SOLID is a set of five design principles that aim to make software designs more understandable, flexible, and maintainable. The principles are:

- **Single Responsibility Principle**
- **Open-Closed Principle**
- **Liskov Substitution Principle**
- **Interface Segregation Principle**
- **Dependency Inversion Principle**

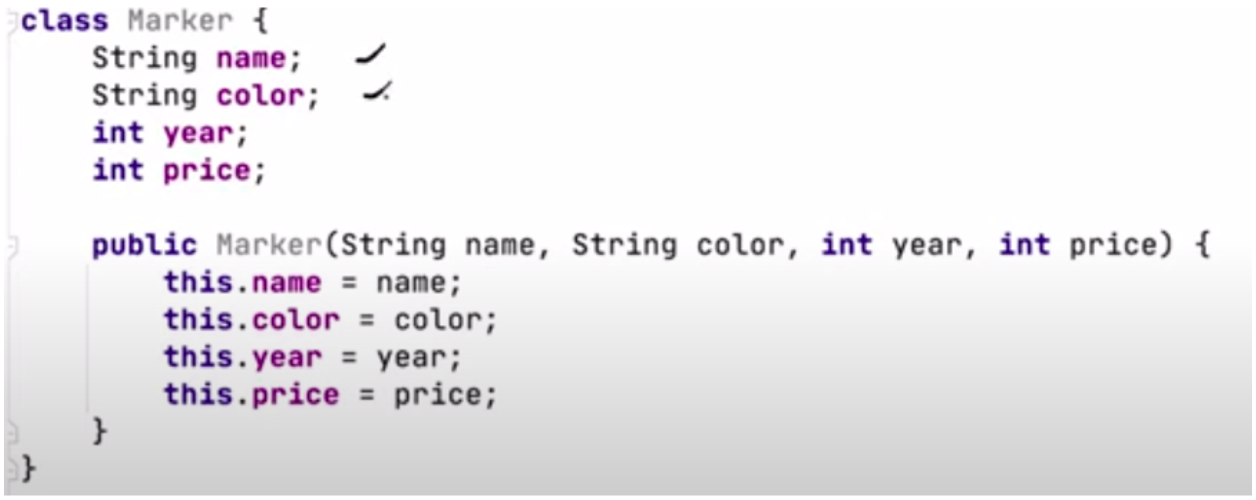
These principles can be applied to object-oriented programming to create more modular and flexible software architectures.

Single Responsibility Principle

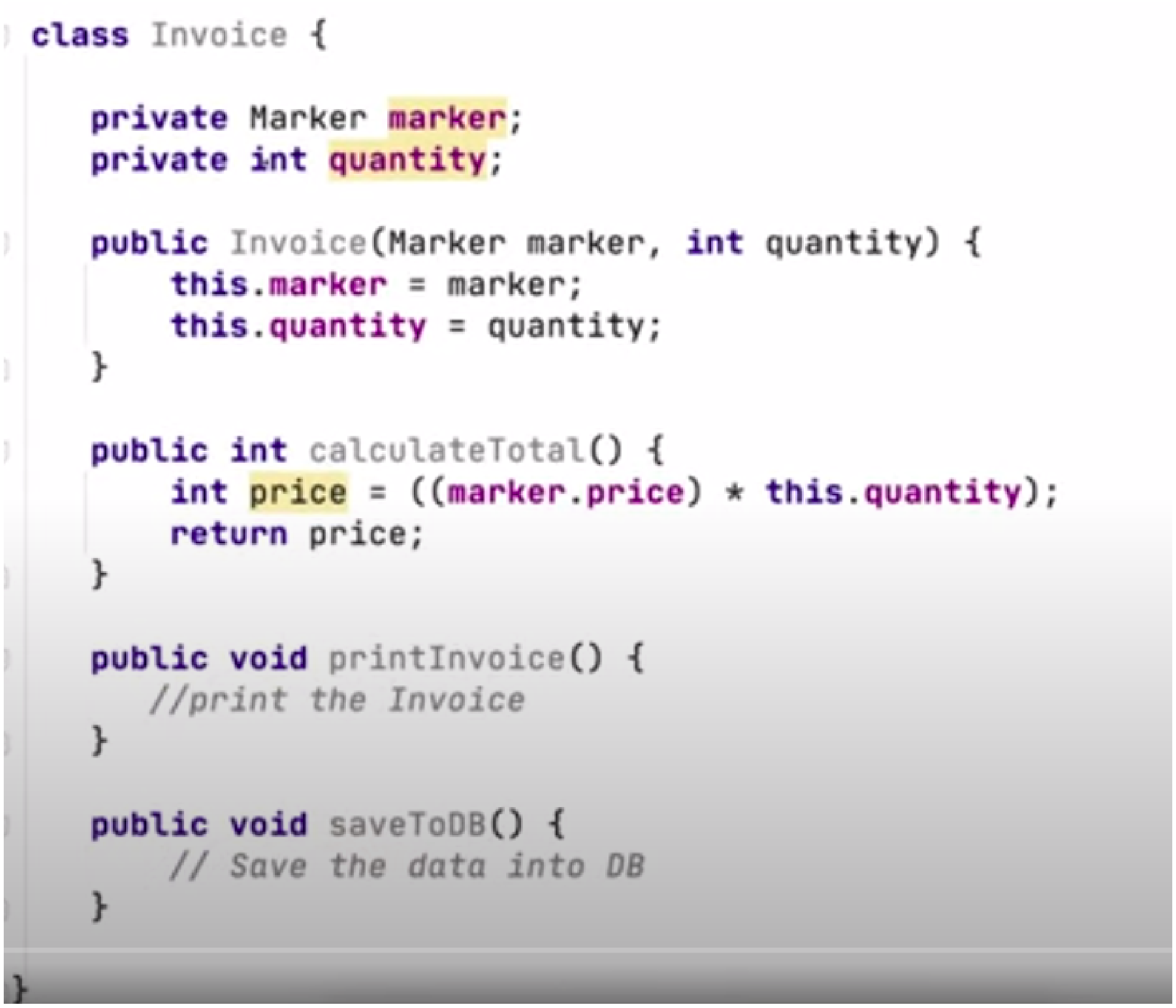
✏️ **The class should have only one reason to change**.

In other words, a class should have only one responsibility.

For example,



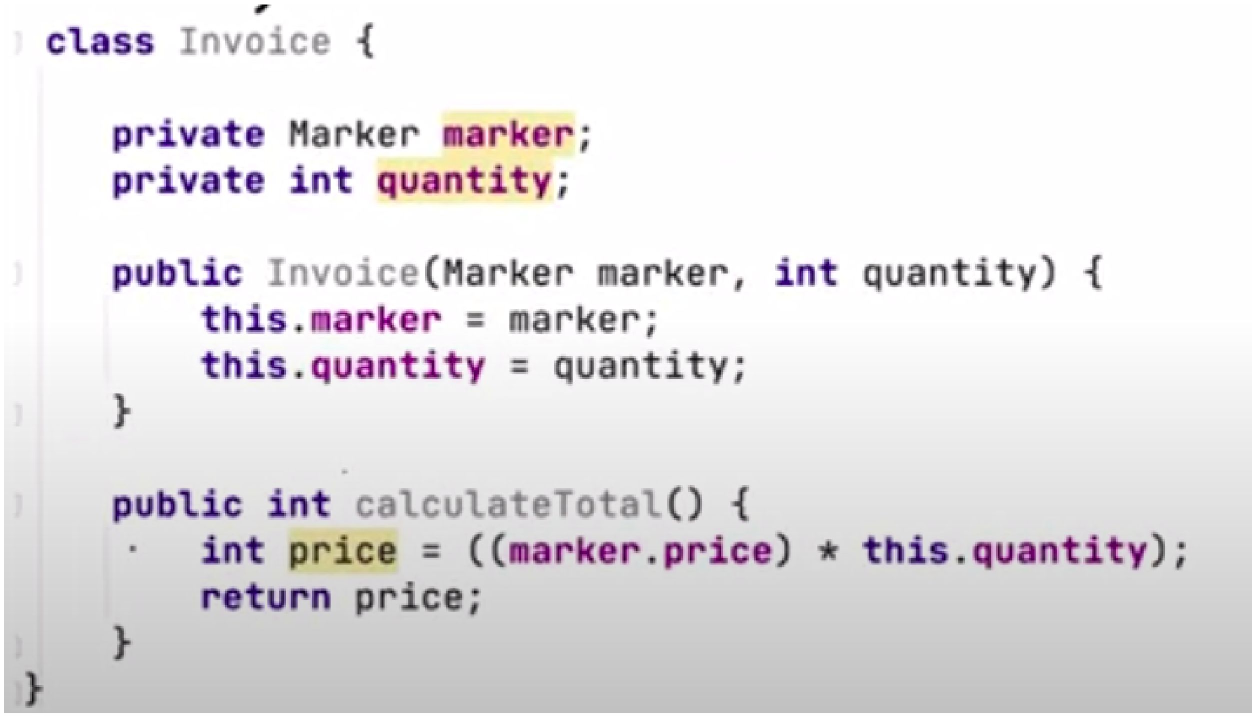
```java
class Marker {
    String name;   ✓
    String color;   ✓
    int year;
    int price;

    public Marker(String name, String color, int year, int price) {
        this.name = name;
        this.color = color;
        this.year = year;
        this.price = price;
    }
}
```



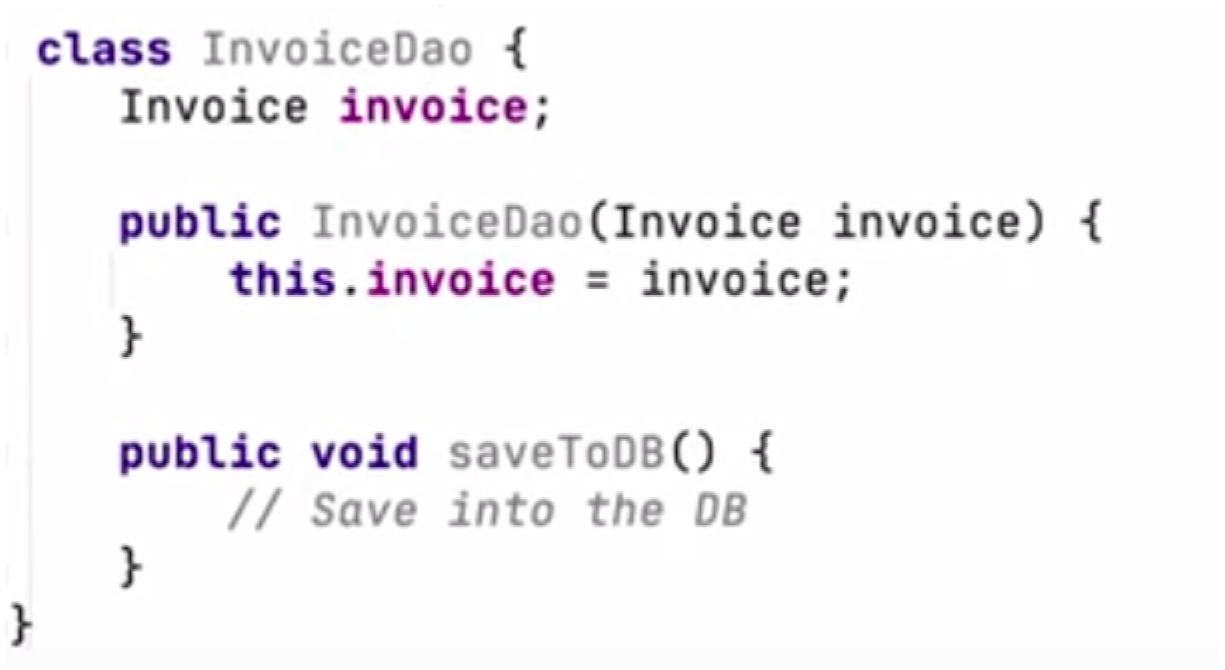
```java
class Invoice {

    private Marker marker;
    private int quantity;

    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }

    public int calculateTotal() {
        int price = ((marker.price) * this.quantity);
        return price;
    }

    public void printInvoice() {
       //print the Invoice
    }

    public void saveToDB() {
        // Save the data into DB
    }

}
```

In this example, If I were to change the way of calculating price (say adding GST to price), then this change will change the behaviour of the class. But at the same time, if I change the way to print the Invoice or what db to save the invoice? , then this will also change the behaviour of the class

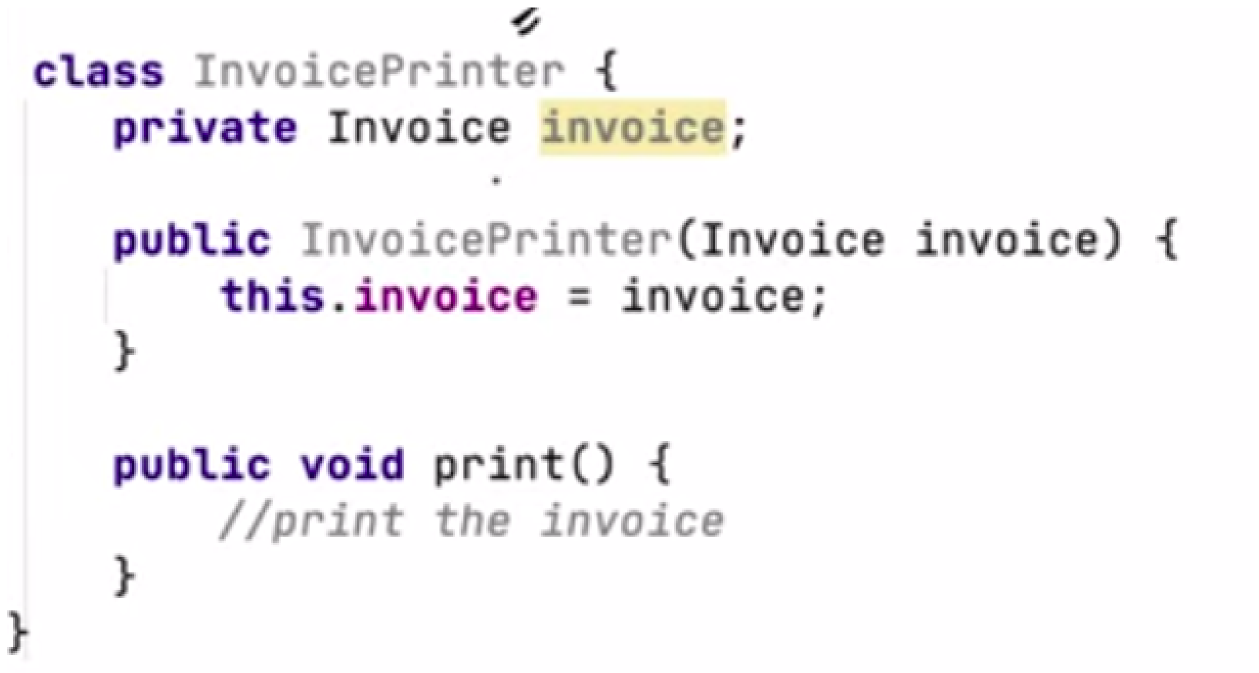
So this implementation of the **Invoice** class does not follow **Single Responsibility Principle**



```java
class Invoice {

    private Marker marker;
    private int quantity;

    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }

    public int calculateTotal() {
        int price = ((marker.price) * this.quantity);
        return price;
    }
}
```



```
class InvoiceDao {
    Invoice invoice;

    public InvoiceDao(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToDB() {
        // Save into the DB
    }
}
```



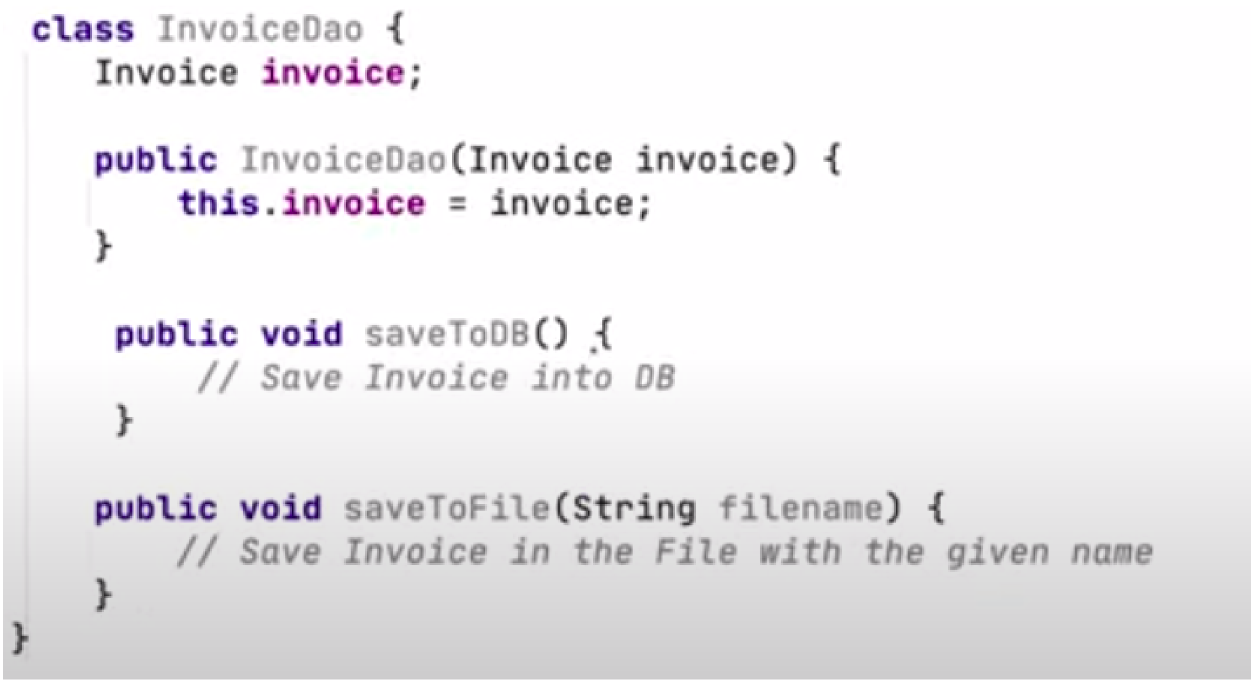
```
class InvoicePrinter {
    private Invoice invoice;

    public InvoicePrinter(Invoice invoice) {
        this.invoice = invoice;
    }

    public void print() {
        //print the invoice
    }
}
```

So now, we have divided the whole class into 3 parts doing specific works. Now each class has only one reason to change**!**

Open/Closed Principle

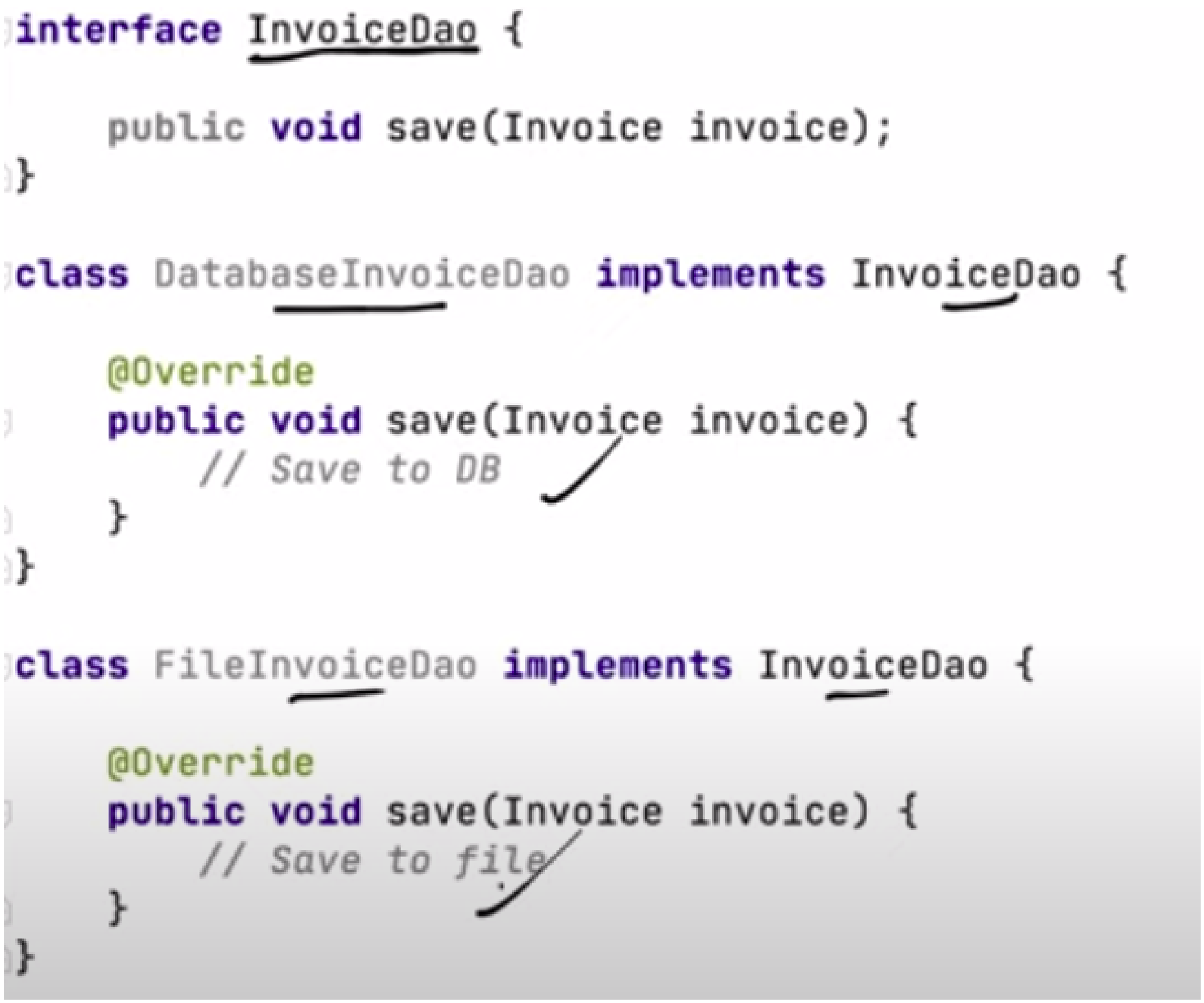
✏️ Open for extension but Closed for modification

For example,



```
class InvoiceDao {
   Invoice invoice;

   public InvoiceDao(Invoice invoice) {
       this.invoice = invoice;
   }

   public void saveToDB() {
       // Save Invoice into DB
   }

   public void saveToFile(String filename) {
       // Save Invoice in the File with the given name
   }
}
```

This class does not follow Open/Closed Principle because we modified the already (tested+live) class which was working file with **saveToFile** method. We should not do this, instead we should extend the original class if we require to add new methods to it.



```
interface InvoiceDao {

    public void save(Invoice invoice);
}

class DatabaseInvoiceDao implements InvoiceDao {

    @Override
    public void save(Invoice invoice) {
        // Save to DB
    }
}

class FileInvoiceDao implements InvoiceDao {

    @Override
    public void save(Invoice invoice) {
        // Save to file
    }
}
```

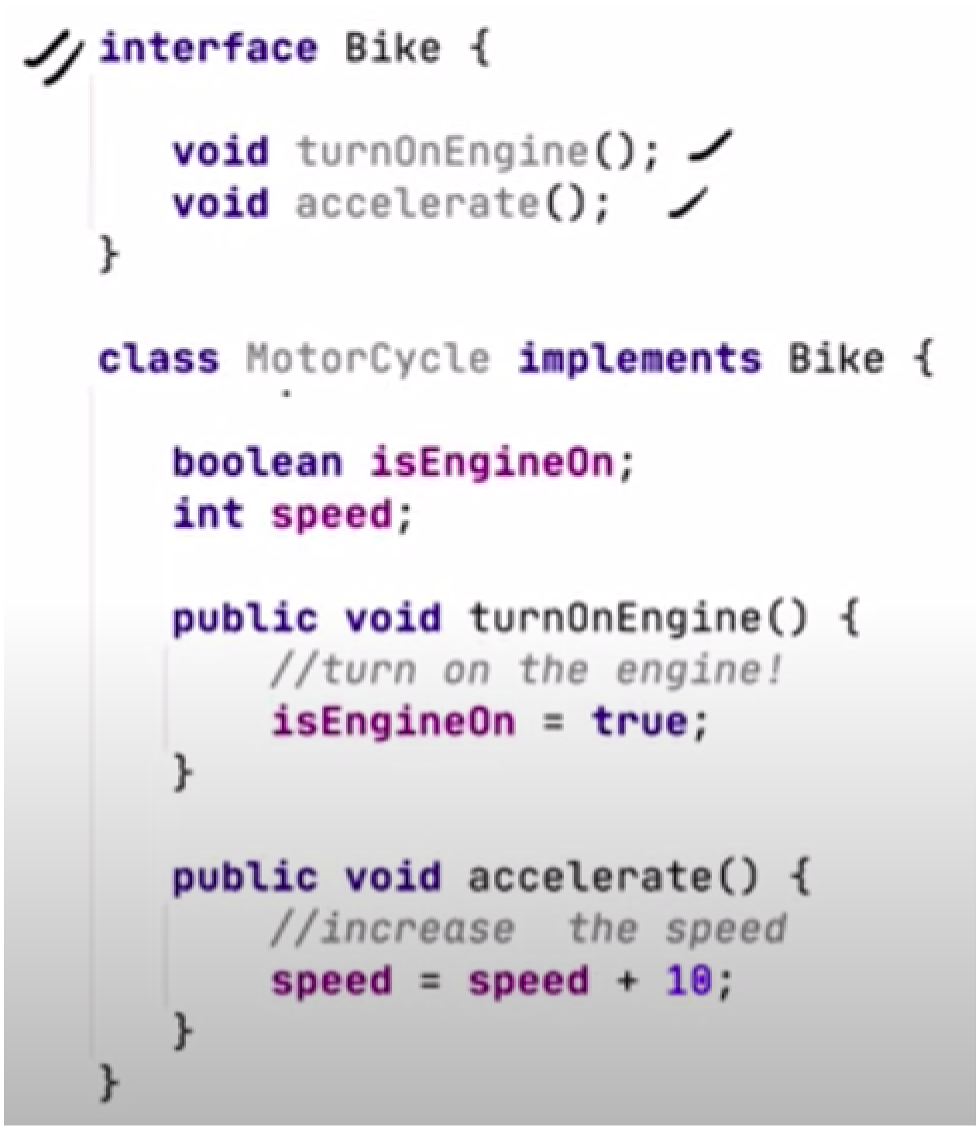
Now, the above example does follow the O/C principle as we are extending classes from our original class to implement new methods.

Liskov Substitution Principle

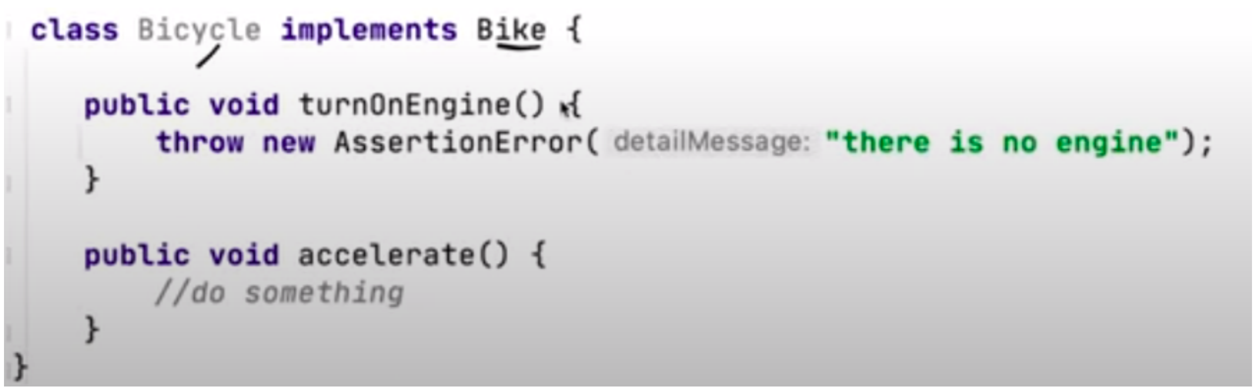
✏️ Subclass should extend the capability of the parent class not narrow it down.

If Class B is a subclass of Class A, then if a program takes an object of class A as input, it should not break the behavior of the program even if we pass a class B (or any other subclass of class A) object to it.

For example,



```java
interface Bike {

    void turnOnEngine();
    void accelerate();
}

class MotorCycle implements Bike {

    boolean isEngineOn;
    int speed;

    public void turnOnEngine() {
        //turn on the engine!
        isEngineOn = true;
    }

    public void accelerate() {
        //increase  the speed
        speed = speed + 10;
    }
}
```



```java
class Bicycle implements Bike {

    public void turnOnEngine() {
        throw new AssertionError( detailMessage: "there is no engine");
    }

    public void accelerate() {
        //do something
    }
}
```

In the above example, if I create a program taking the class **Bike** as a parameter and I give **Motorcycle** as an argument, then it works fine. However, if I give **Bicycle** (also subclass of Bike) as input and call the **turnOnEngine** method, it will give an **error** that breaks the **behaviour** of the program.

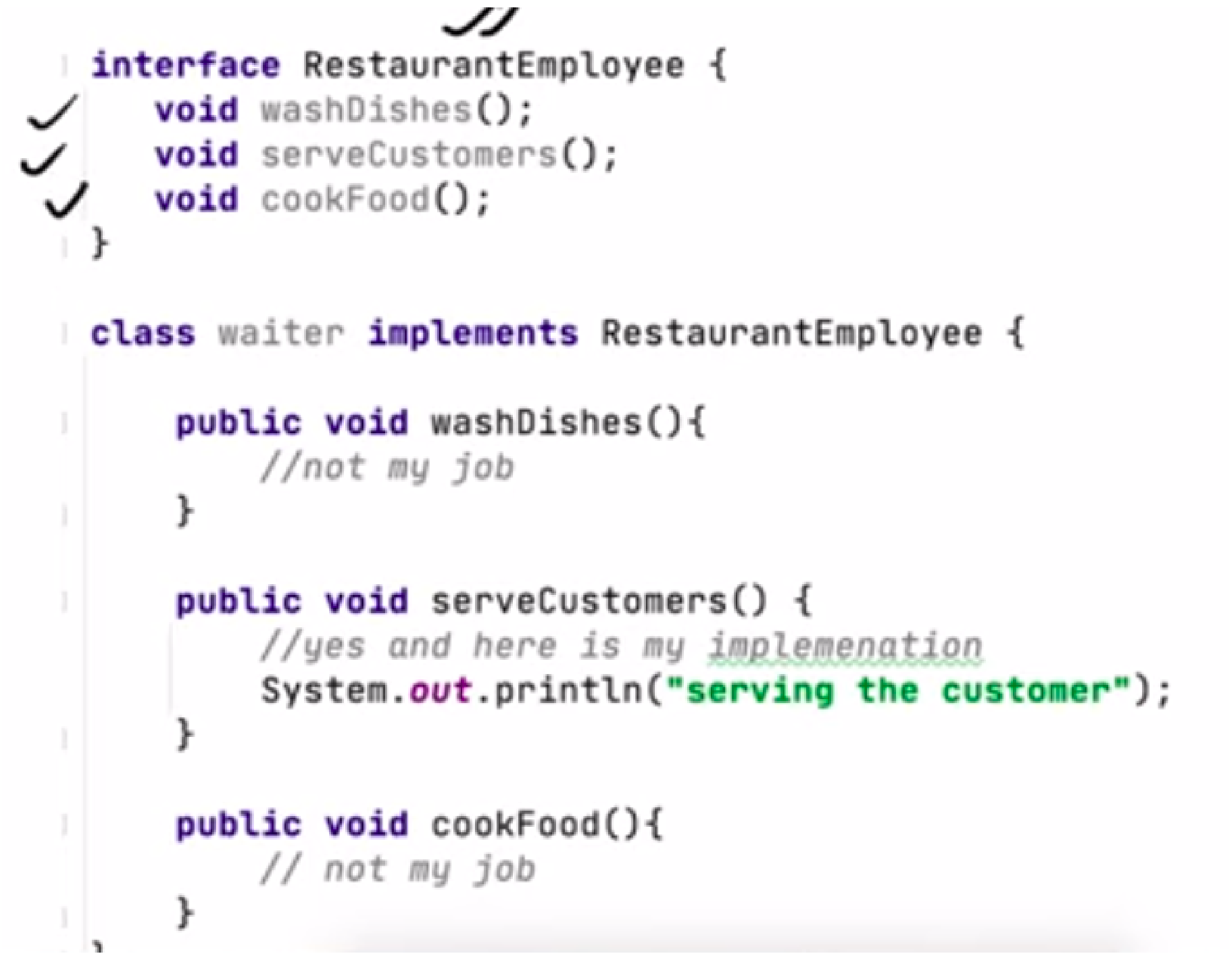
Therefore, it does not follow the above principle.

Interface Segregation Principle

> ✏️ Clients should not implement unnecessary methods which they do not need.

It is applied to interfaces instead of classes.

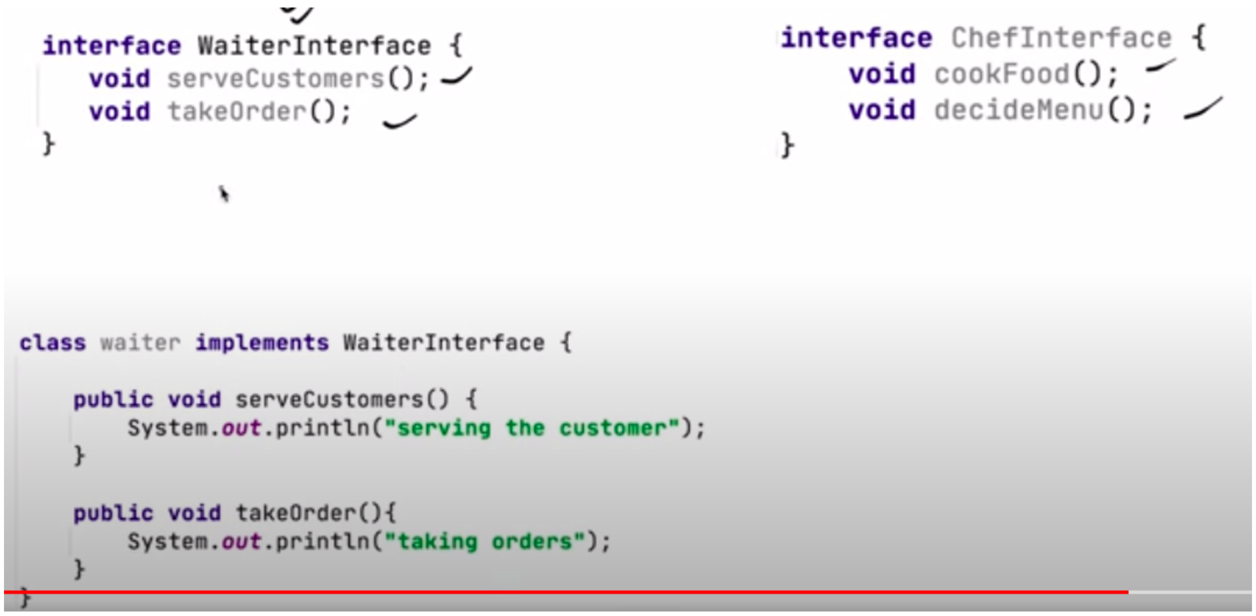
For example,



```java
interface RestaurantEmployee {
    void washDishes();
    void serveCustomers();
    void cookFood();
}

class waiter implements RestaurantEmployee {

    public void washDishes(){
        //not my job
    }

    public void serveCustomers() {
        //yes and here is my implemenation
        System.out.println("serving the customer");
    }

    public void cookFood(){
        // not my job
    }
```

The implementation does not follow the Interface Segregation Principle as the Waiter class does not need to implement the **washDishes** and **cookFood** methods (which are unnecessary).

Corrected Implementation:

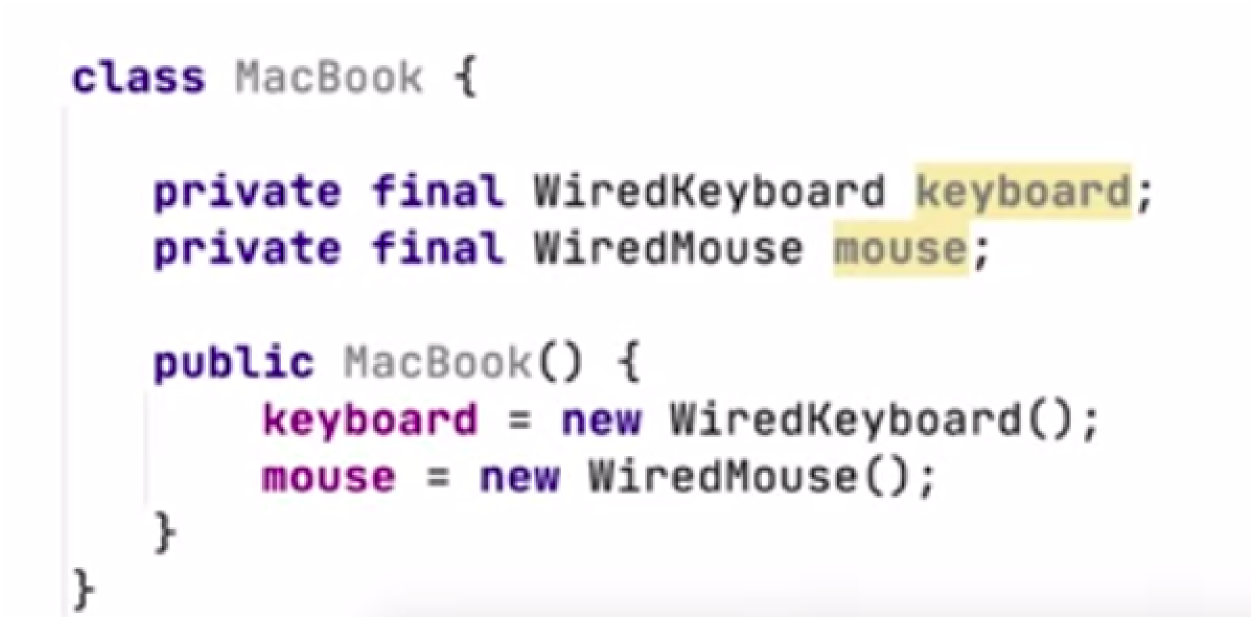


```
interface WaiterInterface {
    void serveCustomers();
    void takeOrder();
}

interface ChefInterface {
    void cookFood();
    void decideMenu();
}

class waiter implements WaiterInterface {

    public void serveCustomers() {
        System.out.println("serving the customer");
    }

    public void takeOrder(){
        System.out.println("taking orders");
    }
}
```

We divided the interfaces to ensure that each interface has specific work attached to it, without any unnecessary methods.

Dependency Inversion Principle

✏️ Classes should depend on interfaces rather than on concrete classes.

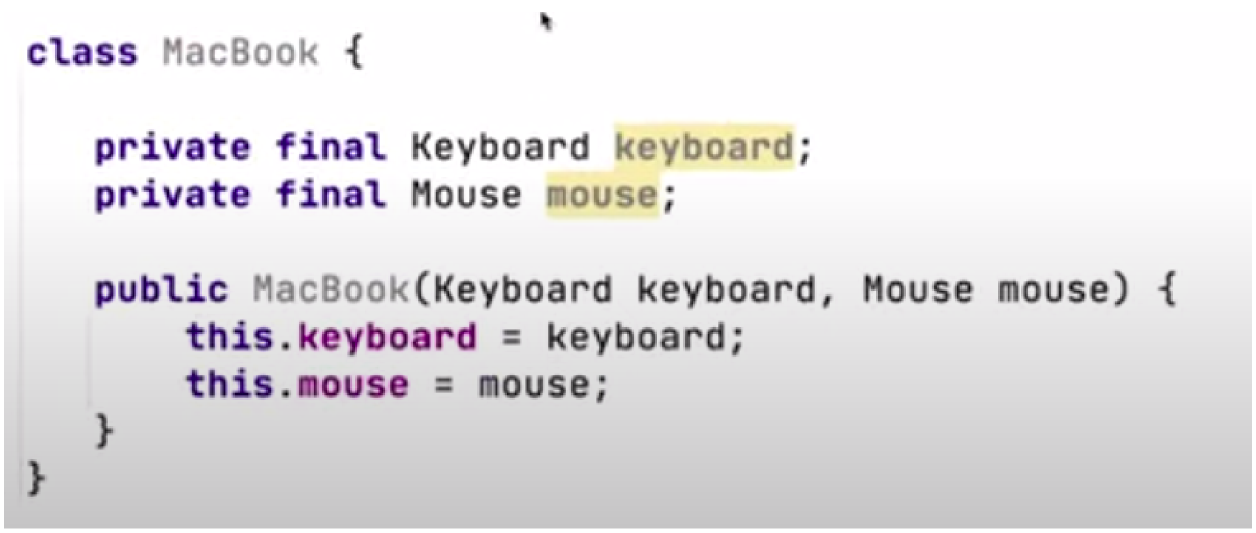


```
class MacBook {

    private final WiredKeyboard keyboard;
    private final WiredMouse mouse;

    public MacBook() {
        keyboard = new WiredKeyboard();
        mouse = new WiredMouse();
    }
}
```

Let's say, we have two interfaces (Mouse and Keyboard) both with two different classes of **Wired and Bluetooth.**

In the **MacBook** class, we created objects from the **WiredKeyboard** and **WiredMouse** classes. However, if we need to upgrade to a Bluetooth Mouse in the future, we cannot do so because we created objects from concrete classes. If we had created objects from their interfaces instead, we could have made the change.

Correct Implementation:



```
class MacBook {

    private final Keyboard keyboard;
    private final Mouse mouse;

    public MacBook(Keyboard keyboard, Mouse mouse) {
        this.keyboard = keyboard;
        this.mouse = mouse;
    }
}
```

Now, we have used interface objects in the MacBook class so that we can modify Mouse/Keyboard type whenever required.