S.O.L.I.D Principles .

S - Single Resposibility Priciple.
O - Open / Closed Principle
I - Interface Segmented Principle
D - Dependency Inversion Principle

Advantage of following these Principles .

Help us to write better code:

- Avoid Duplicate code.
- Easy to maintain
- Easy to understand.
- Flexible software
- Reduce Complexity.

S Single Responsibility Principle.

- A class should have only 1 reason to change.

Here We are getting 3 change

```
package System_Design.S_O_L_I_D_Principles;

//Marker Entity
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;
    }
}

class Invoice{
    private Marker marker;
    private int quantity;
```

```
    public Invoice (Marker marker,int quantity){
        this.marker=marker;
        this.quantity=quantity;
    }
    //Change 1.
    //We want to change the logic for calaulation in price
    //we want add GST Based Calculations.

    public int calculateTotal(){
        int price = (marker.price)*this.quantity;
        return price;
    }
    //change 2
    //We want change format for Printing Invoice
    public void printInvoice(){
        //print the Invoice
    }
    //change 3
    //We also want to save data in file
    public void saveToDB(){
        //save into DATABASE
    }
}


public class Single_Responsibility_Principle {
}
```

Check For Only one responsibility

```
class Invoice{
    private Marker marker;
    private int quantity;

    public Invoice (Marker marker,int quantity){
        this.marker=marker;
        this.quantity=quantity;
    }
    //Change 1.
    //We want to change the logic for calaulation in price
    //we want add GST Based Calculations.

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

```
//only one change
class InvoiceDao {
    Invoice invoice;
    public InvoiceDao(Invoice invoice){
        this.invoice = invoice;
    }

    public void saveToDB(){
        //save To DB
    }
}
```

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

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

O Open/Closed Principle

- Open For Extension but Closed for Modifications.
- A class has already tested and working on live so we cannot modify that class we need extend their property into some other class.

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

    public void saveToDB(){
        //save To DB
    }

    public void saveToFile(){
        //save To File
    }
}
```

Below Code follow the OPEN/CLOSED Priciples

```
interface InvoiceDaoo{
    public void save(Invoice invoice);
}
```

```
class DatabaseInvoiceDao implements InvoiceDaoo{

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

```
class FileSystemInvoiceDao implements InvoiceDaoo{

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

L Liskov Subsititution Principle.

- If Class B is subtype of class A, then we should be able to replace object of A with B without breaking the behaviour of the program.
- Subclass should extend the capability of parent class not narrow it down.
- We have increase the behaviour of supplier .

```
interface Bike{
    void turnOnEngine();
    void accelerate();
}
```

```
class MotorCycle implements Bike{
    boolean isEngineOn;
    int speed;
```

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

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

```
class Bicycle implements Bike{

    //here we are narrow down the capability of Bike
    //we are changing the default behaviour of
    // object of bike by throwing exception
    @Override
    public void turnOnEngine() {
        throw new AssertionError("There is no engine bicycle")
    }

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

Example Problem 2

```
public class Vehicle{
    public Integer getNumberOfWheels(){
        return 2;
    }
    public Boolean hasEngine(){
        return true;
    }
}
```

```
public class MotorCyclee extends Vehicle{

}
```

```
 class Car extends Vehicle{
    @Override
    public Integer getNumberOfWheels(){
        return 4;
    }
}
```

```
//it's reduce the capability of Vehicle class
class Bycyclee extends Vehicle{

    @Override
    public Boolean hasEngine(){
        return null;
    }
}
```

```
public class Liskov_Subsititution_Principle {
    public static void main(String[] args){
        List<Vehicle> vehicleList = new ArrayList<>();
        vehicleList.add(new MotorCyclee());
        vehicleList.add(new Car());
        vehicleList.add(new Bycyclee());

        for(Vehicle v : vehicleList){
            System.out.println(v.hasEngine().toString());
        }
    }
}
output:
true
true
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.lang.Boolean.toString()" because the return value of
"System_Design.S_O_L_I_D_Principles.Vehicle.hasEngine()" is null
    at
System_Design.S_O_L_I_D_Principles.Liskov_Subsititution_Principle.main(Liskov_Subsititut
ion_Principle.java:87)

Process finished with exit code 1
```

Solution.

```
//solution
class Vehicle{
    public Integer getNumberOfWheels(){
        return 2;
    }
```

```
}

class EngineVehicle extends Vehicle{
    public Boolean hasEngine(){
        return true;
    }
}


class MotorCyclee extends EngineVehicle{

}

 class Car extends EngineVehicle{

    @Override
    public Integer getNumberOfWheels(){
        return 4;
    }


}

class Bycyclee extends Vehicle{

}

public class Liskov_Subsititution_Principle {
    public static void main(String[] args){
        List<Vehicle> vehicleList = new ArrayList<>();
        vehicleList.add(new MotorCyclee());
        vehicleList.add(new Car());
        vehicleList.add(new Bycyclee());

        for(Vehicle v : vehicleList){
            //compile time errpr
            //System.out.println(v.hasEngine().toString());
            System.out.println(v.getNumberOfWheels().toString());
        }
    }
}
output:
2
4
2

Process finished with exit code 0
```

I Interface Segmented Principle.

- Interfaces should be such, that client should implement unnecessary functions they do not need.

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

Not following the Interface Segmentation

```
class waiter implements RestaurantEmployee{

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

    @Override
    public void serveCustomers() {
        //ues and here is my implementation
        System.out.println("Serving the customer");

    }

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

Fowllowing the Interface Segeration.

```
interface WaiterInterface{
   void serveToCustomers();
   void takeOrder();

}

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

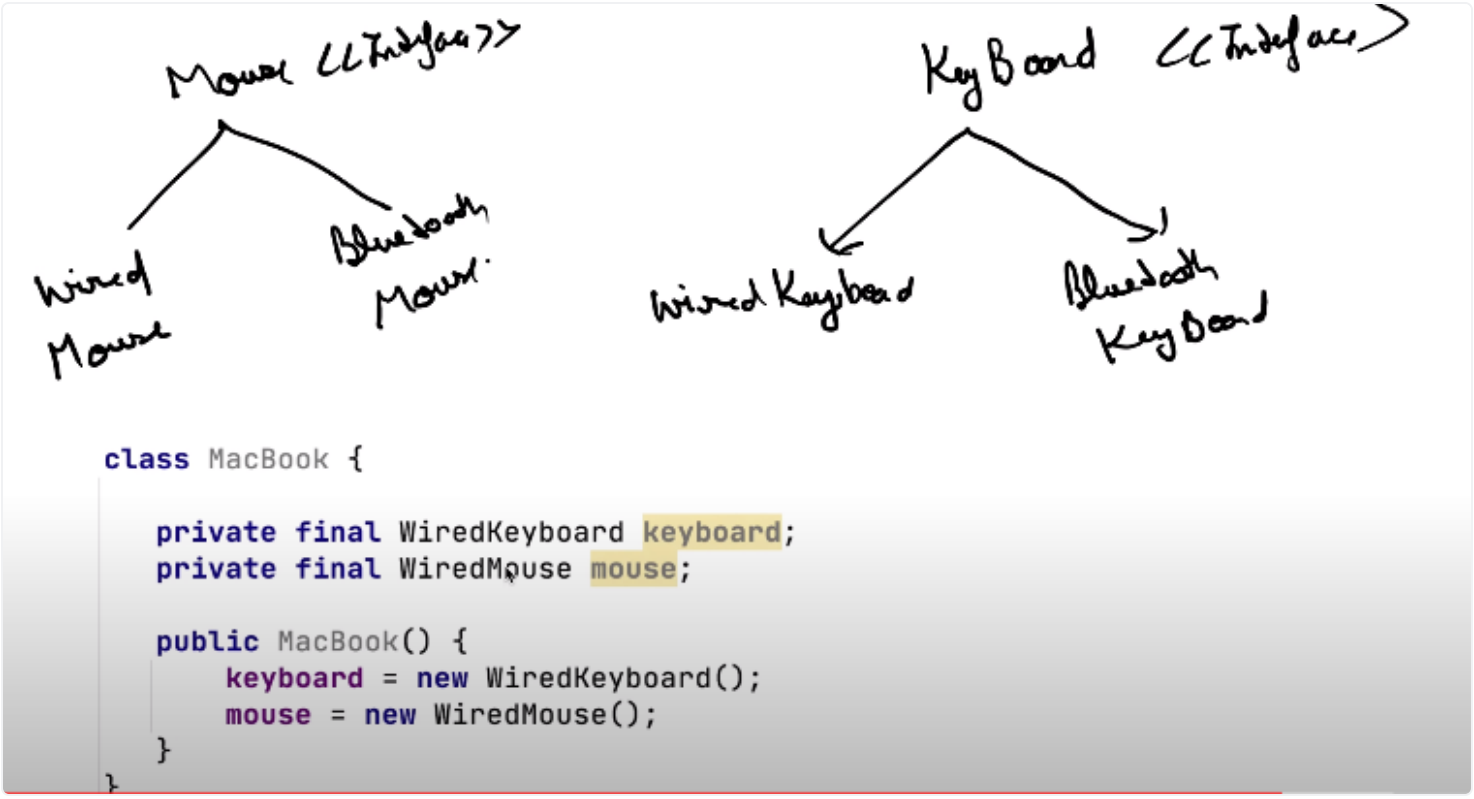
```
class OnlyWaiter implements WaiterInterface{

    @Override
    public void serveToCustomers() {
        System.out.println("Serving To customers");
    }

    @Override
    public void takeOrder() {
        System.out.println("Taking order from Customers");
    }
}
```

D Dependency Inversion Principle.

- Class should depend on interfaces rather than concrete class




```
class MacBook {

    private final WiredKeyboard keyboard;
    private final WiredMouse mouse;

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

```java
class MacBook {

    private final Keyboard keyboard;
    private final Mouse mouse;

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