SOLID Design Principle

Table of Contents

Points to discuss:

- Overview of SOLID
- What does "S" stand for and its meaning
- What does "O" stand for and its meaning
- What does "L" stand for and its meaning
- What does "I" stand for and its meaning
- What does "D" stand for and its meaning

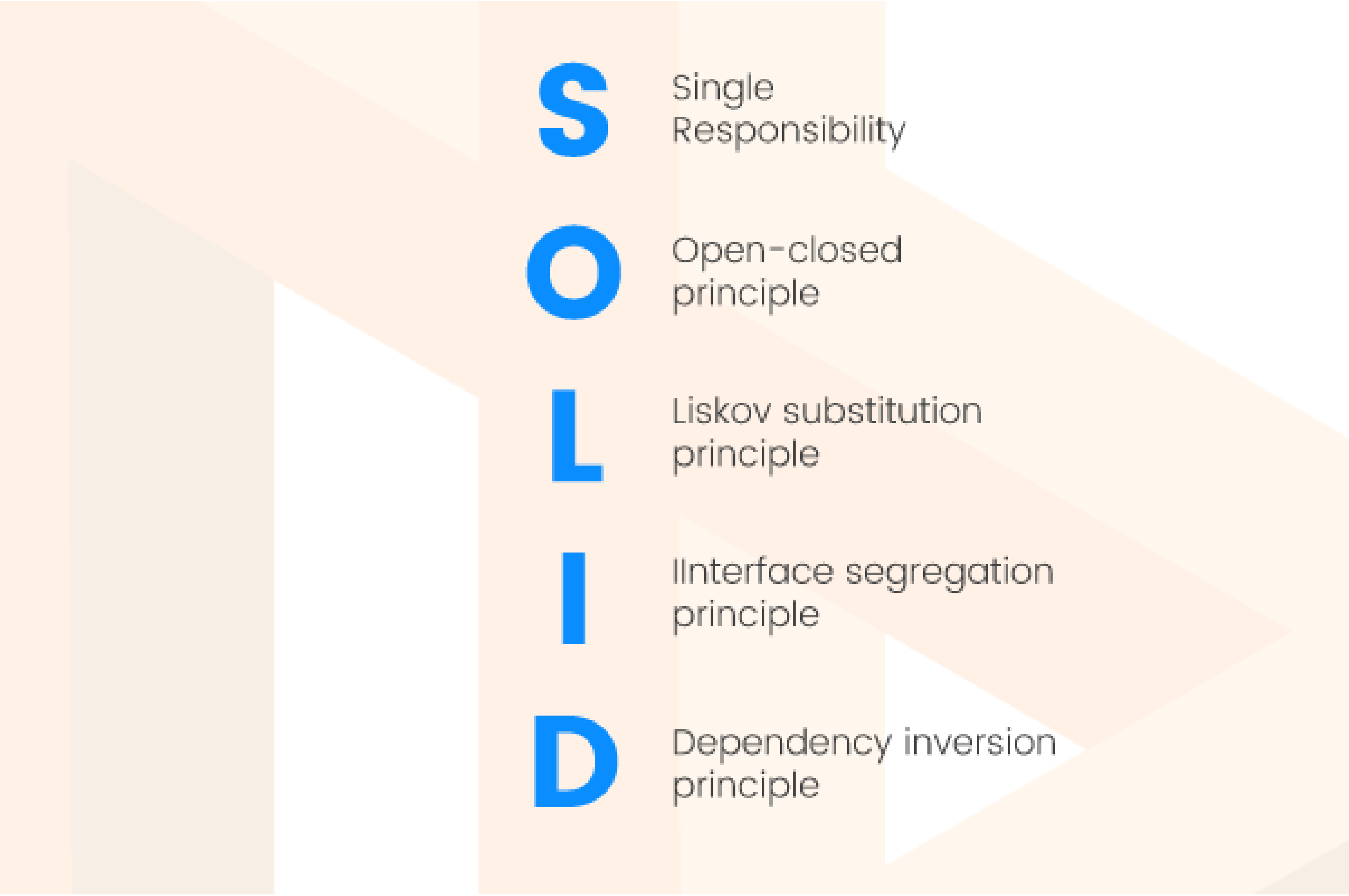
About SOLID

First introduced by the famous Computer Scientist Rober J Martin in 2000 .

But the SOLID acronym was released later by Michael Feathers.

SOLID refers to 5 design principle of OOP which has been summed up by experienced developers.

It all aims to better understanding, testing and especially maintaining.



S Single Responsibility

O Open-closed principle

L Liskov substitution principle

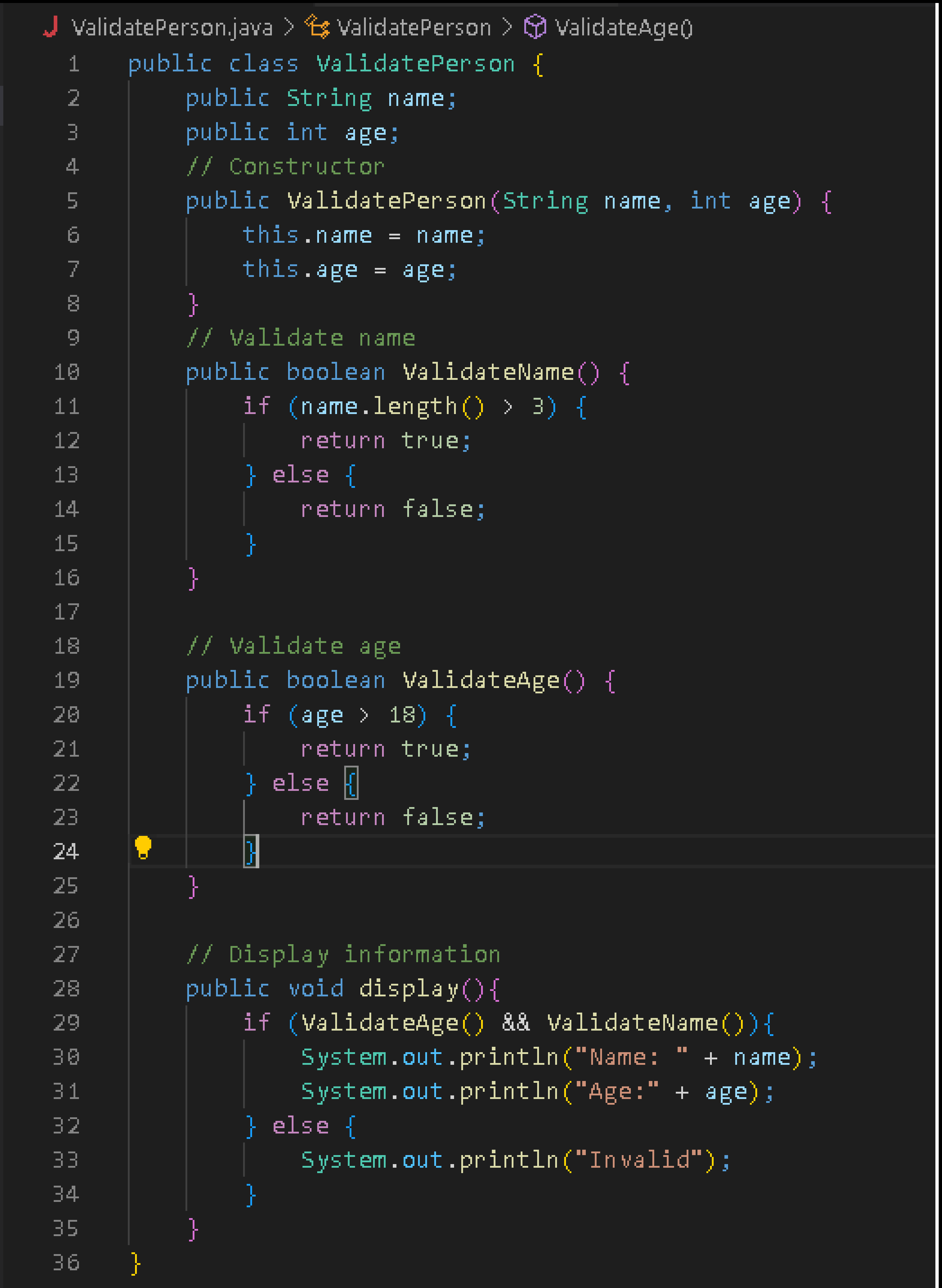
I lInterface segregation principle

D Dependency inversion principle

1. The Single Responsibility Principle

What is it ?

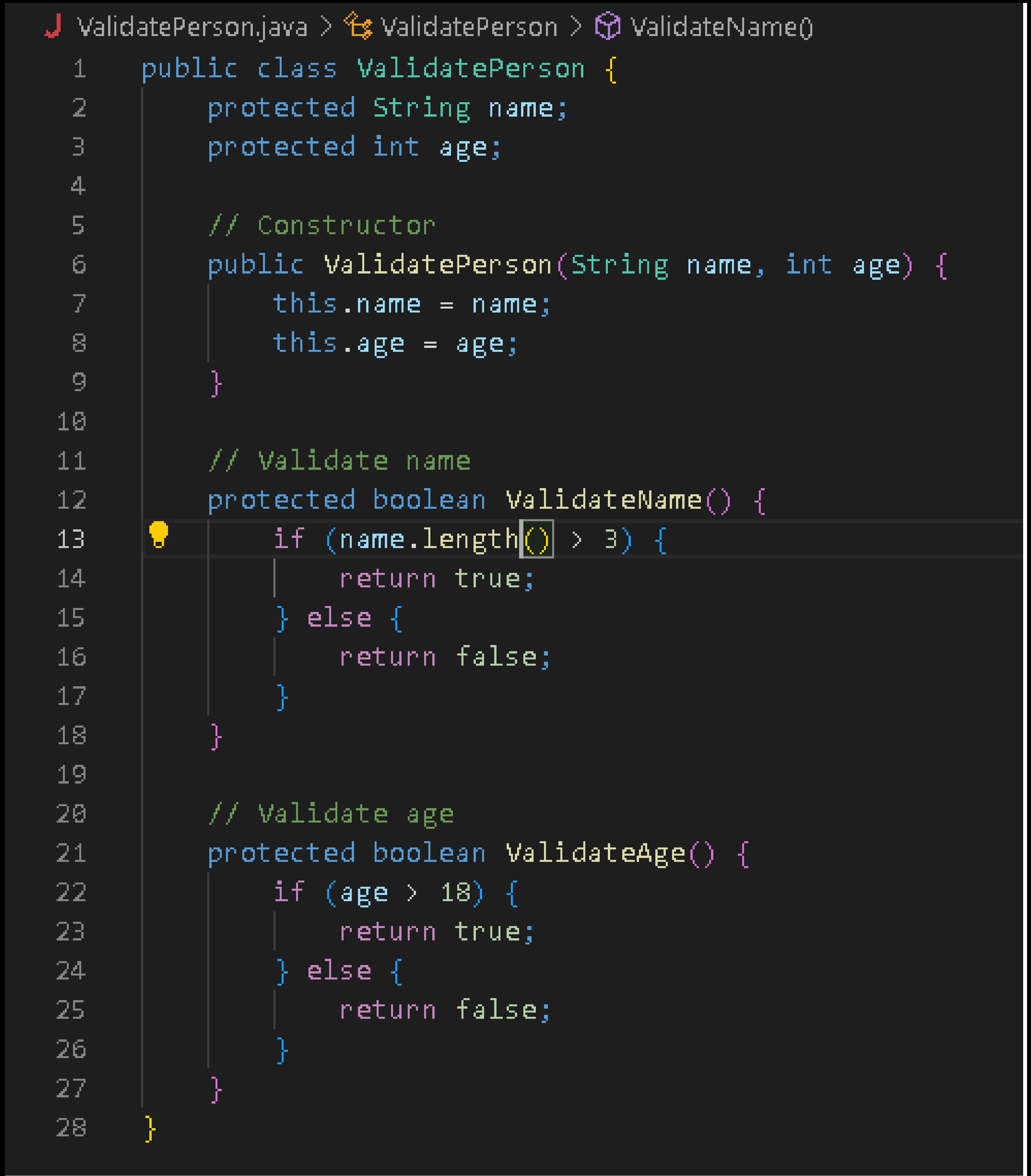
It states that a class should be responsible just for one job and therefore, it has only a single reason to change.



```java
public class ValidatePerson {
    public String name;
    public int age;
    // Constructor
    public ValidatePerson(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // Validate name
    public boolean ValidateName() {
        if (name.length() > 3) {
            return true;
        } else {
            return false;
        }
    }

    // Validate age
    public boolean ValidateAge() {
        if (age > 18) {
            return true;
        } else {
            return false;
        }
    }

    // Display information
    public void display(){
        if (ValidateAge() && ValidateName()){
            System.out.println("Name: " + name);
            System.out.println("Age:" + age);
        } else {
            System.out.println("Invalid");
        }
    }
}
```

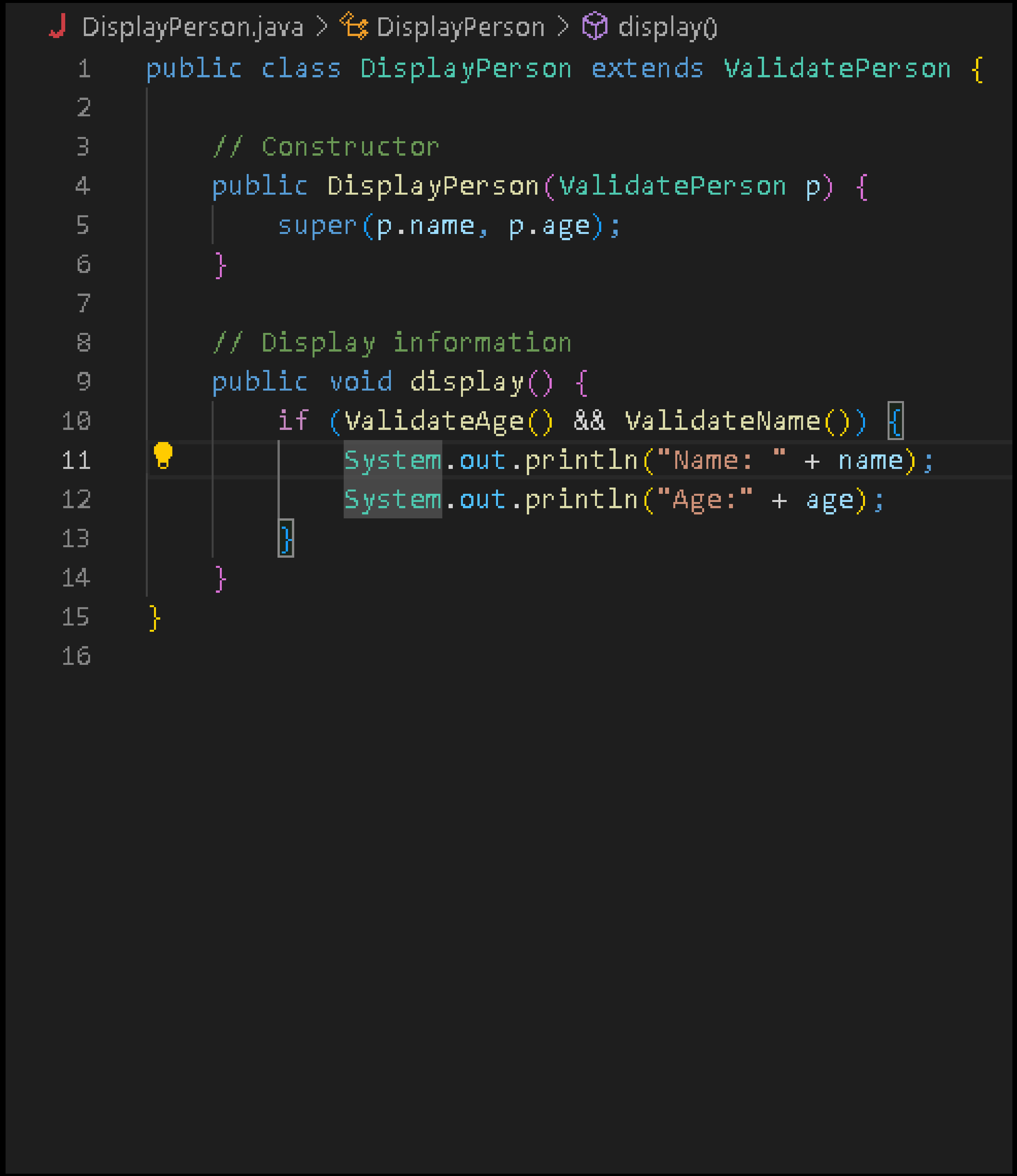
This goes against S rule

How we fix?



```java
public class ValidatePerson {
    protected String name;
    protected int age;

    // Constructor
    public ValidatePerson(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Validate name
    protected boolean ValidateName() {
        if (name.length() > 3) {
            return true;
        } else {
            return false;
        }
    }

    // Validate age
    protected boolean ValidateAge() {
        if (age > 18) {
            return true;
        } else {
            return false;
        }
    }
}
```

and



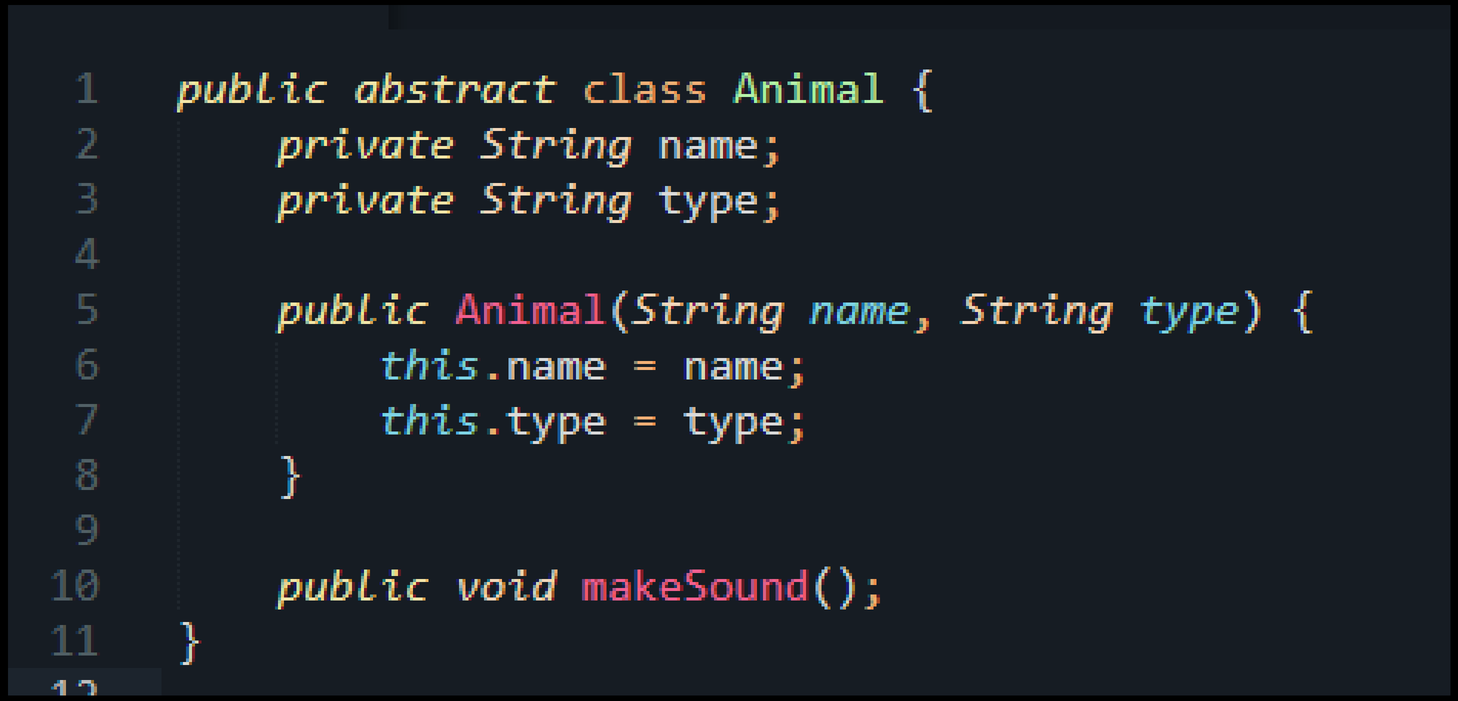
```java
public class DisplayPerson extends ValidatePerson {

    // Constructor
    public DisplayPerson(ValidatePerson p) {
        super(p.name, p.age);
    }

    // Display information
    public void display() {
        if (ValidateAge() && ValidateName()) {
            System.out.println("Name: " + name);
            System.out.println("Age:" + age);
        }
    }
}

```

2. Open - Closed Principle

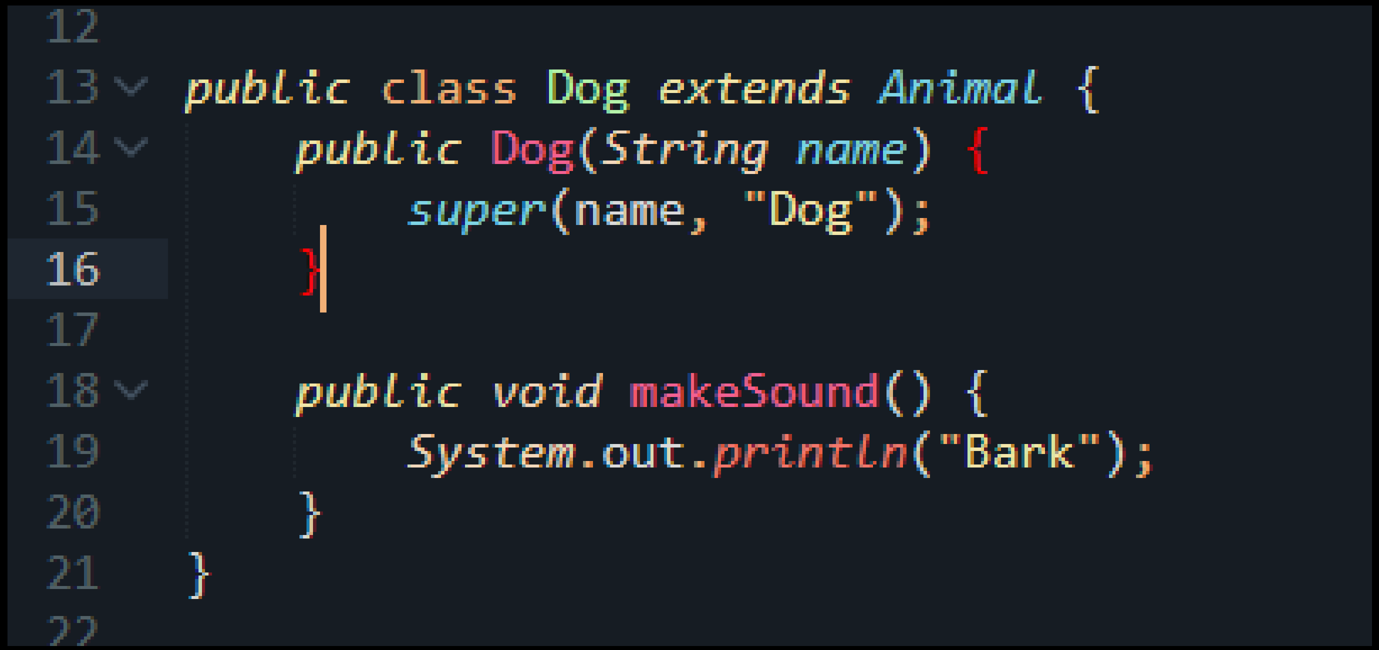
What is it ?

It states that classes can be opened for extension but should be closed to modification.

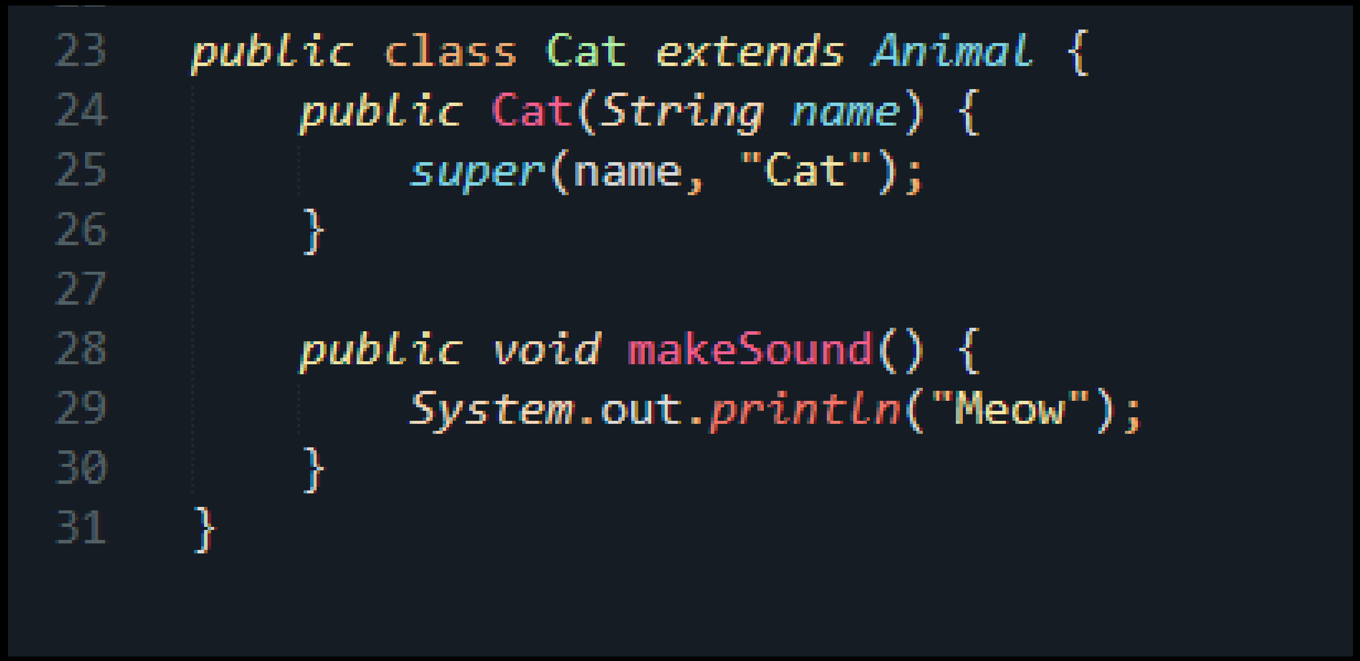
Example:



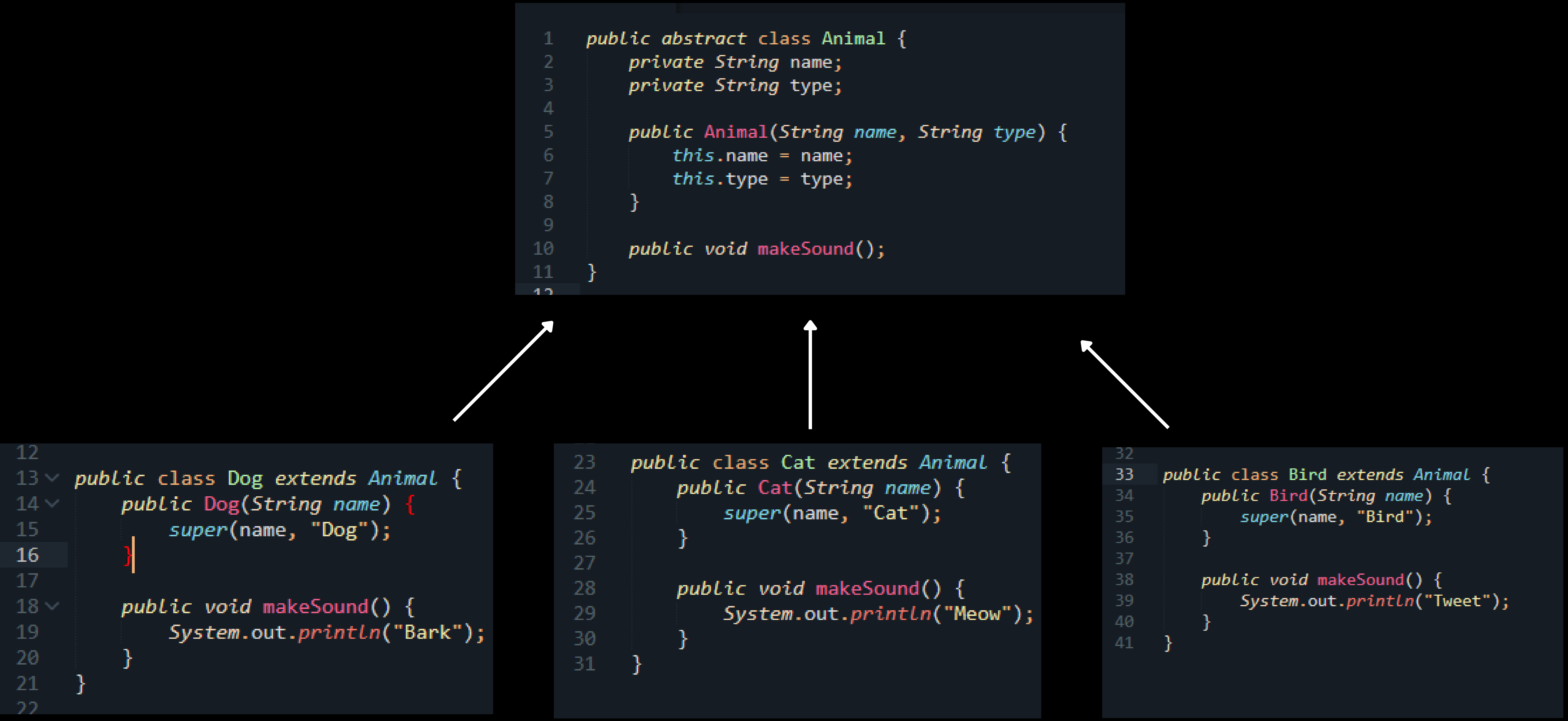
```java
public abstract class Animal {
    private String name;
    private String type;

    public Animal(String name, String type) {
        this.name = name;
        this.type = type;
    }

    public void makeSound();
}
```



```java

public class Dog extends Animal {
    public Dog(String name) {
        super(name, "Dog");
    }

    public void makeSound() {
        System.out.println("Bark");
    }
}
```



```java
public class Cat extends Animal {
    public Cat(String name) {
        super(name, "Cat");
    }

    public void makeSound() {
        System.out.println("Meow");
    }
}
```



```java
public abstract class Animal {
    private String name;
    private String type;

    public Animal(String name, String type) {
        this.name = name;
        this.type = type;
    }

    public void makeSound();
}
```

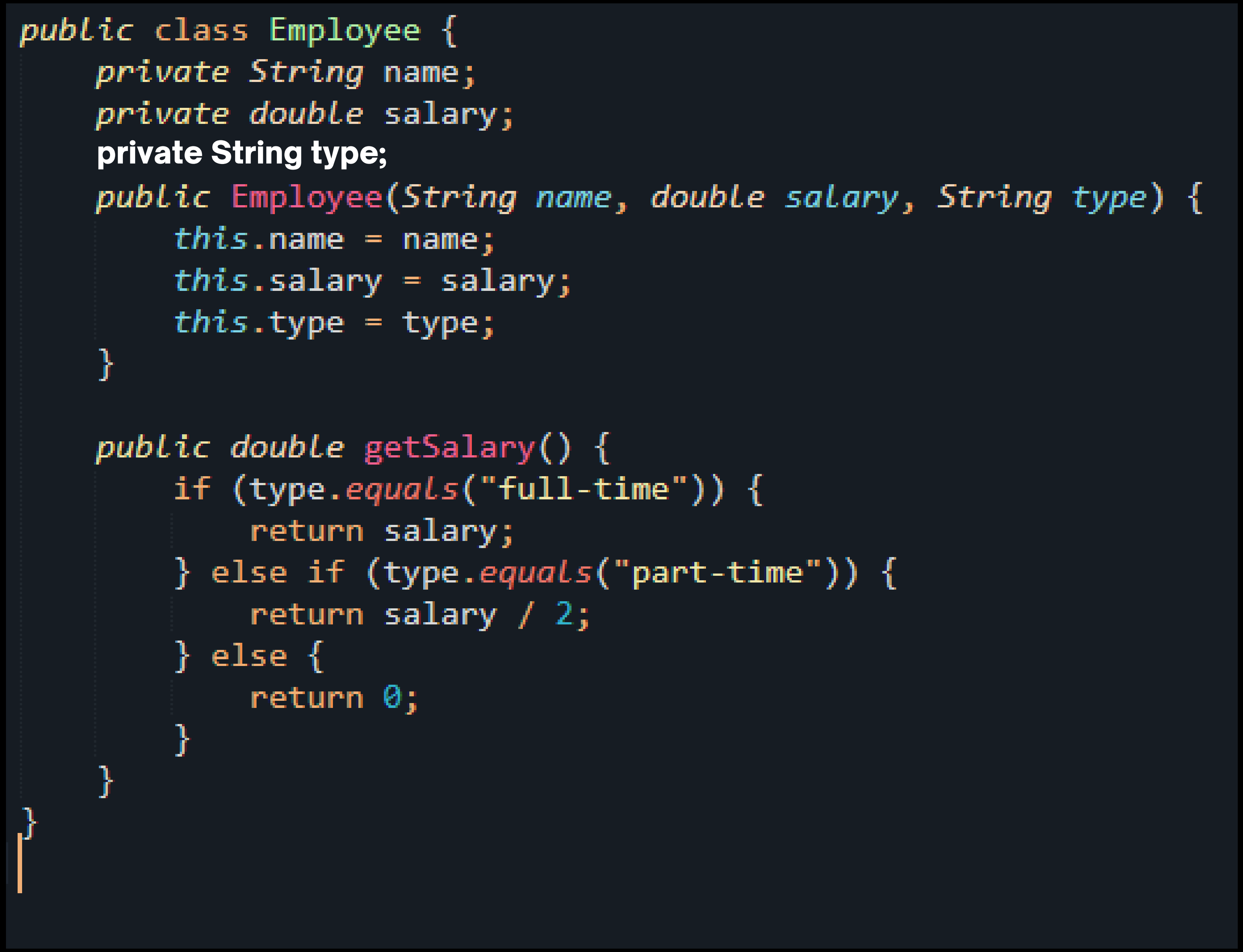
```java
public class Dog extends Animal {
    public Dog(String name) {
        super(name, "Dog");
    }

    public void makeSound() {
        System.out.println("Bark");
    }
}
```

```java
public class Cat extends Animal {
    public Cat(String name) {
        super(name, "Cat");
    }

    public void makeSound() {
        System.out.println("Meow");
    }
}
```

```java
public class Bird extends Animal {
    public Bird(String name) {
        super(name, "Bird");
    }

    public void makeSound() {
        System.out.println("Tweet");
    }
}
```

We can still extend without modifying parent class

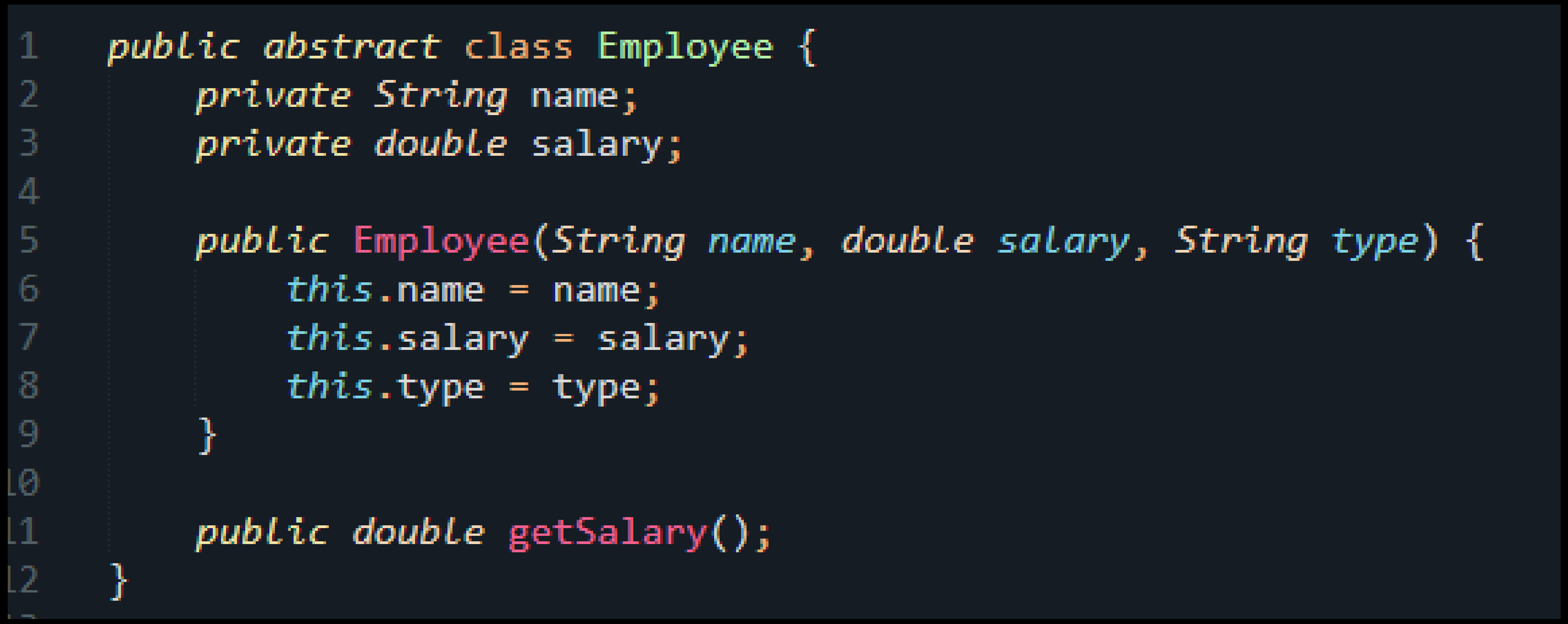
Another example:



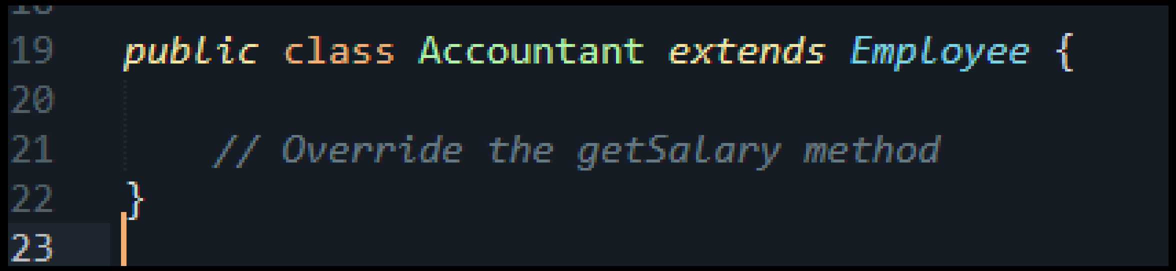
```
public class Employee {
    private String name;
    private double salary;
    private String type;
    public Employee(String name, double salary, String type) {
        this.name = name;
        this.salary = salary;
        this.type = type;
    }

    public double getSalary() {
        if (type.equals("full-time")) {
            return salary;
        } else if (type.equals("part-time")) {
            return salary / 2;
        } else {
            return 0;
        }
    }
}
```

What happens if we extend the Employee entity?

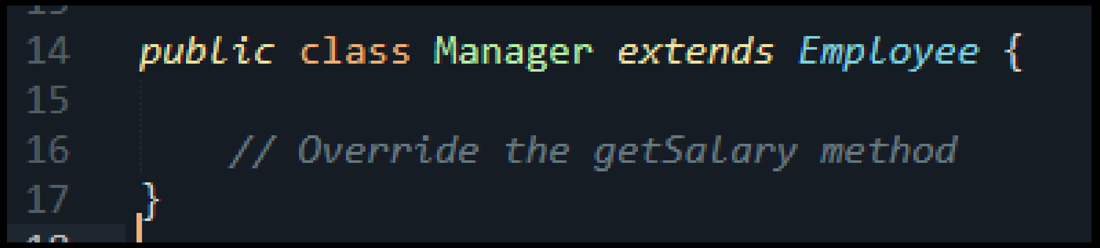
Solving method:



```java
public abstract class Employee {
    private String name;
    private double salary;

    public Employee(String name, double salary, String type) {
        this.name = name;
        this.salary = salary;
        this.type = type;
    }

    public double getSalary();
}
```



```java
public class Accountant extends Employee {

    // Override the getSalary method
}
```



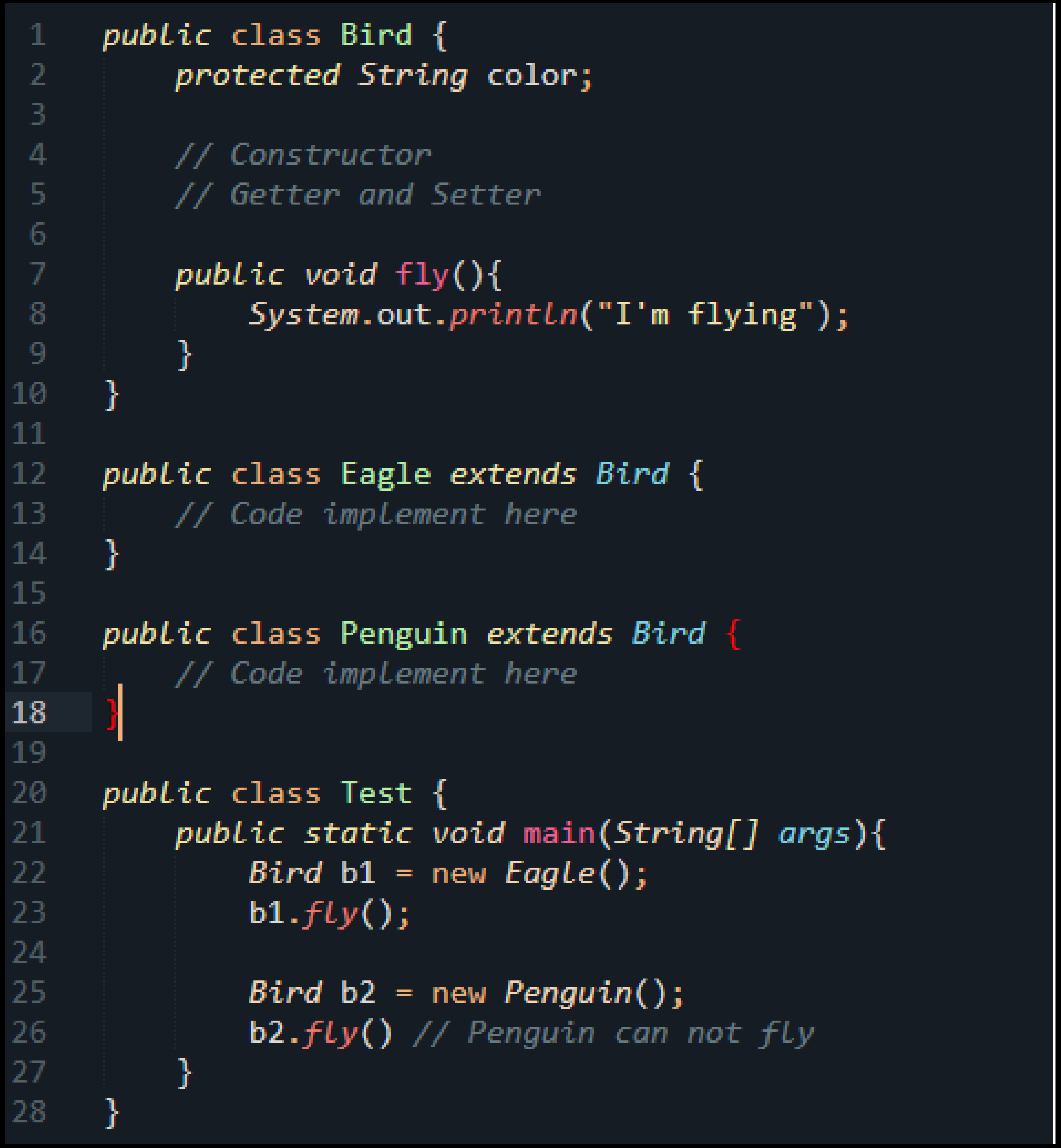
```java
public class Manager extends Employee {

    // Override the getSalary method
}
```

3. Liskov Substitution Principle

What is it ?

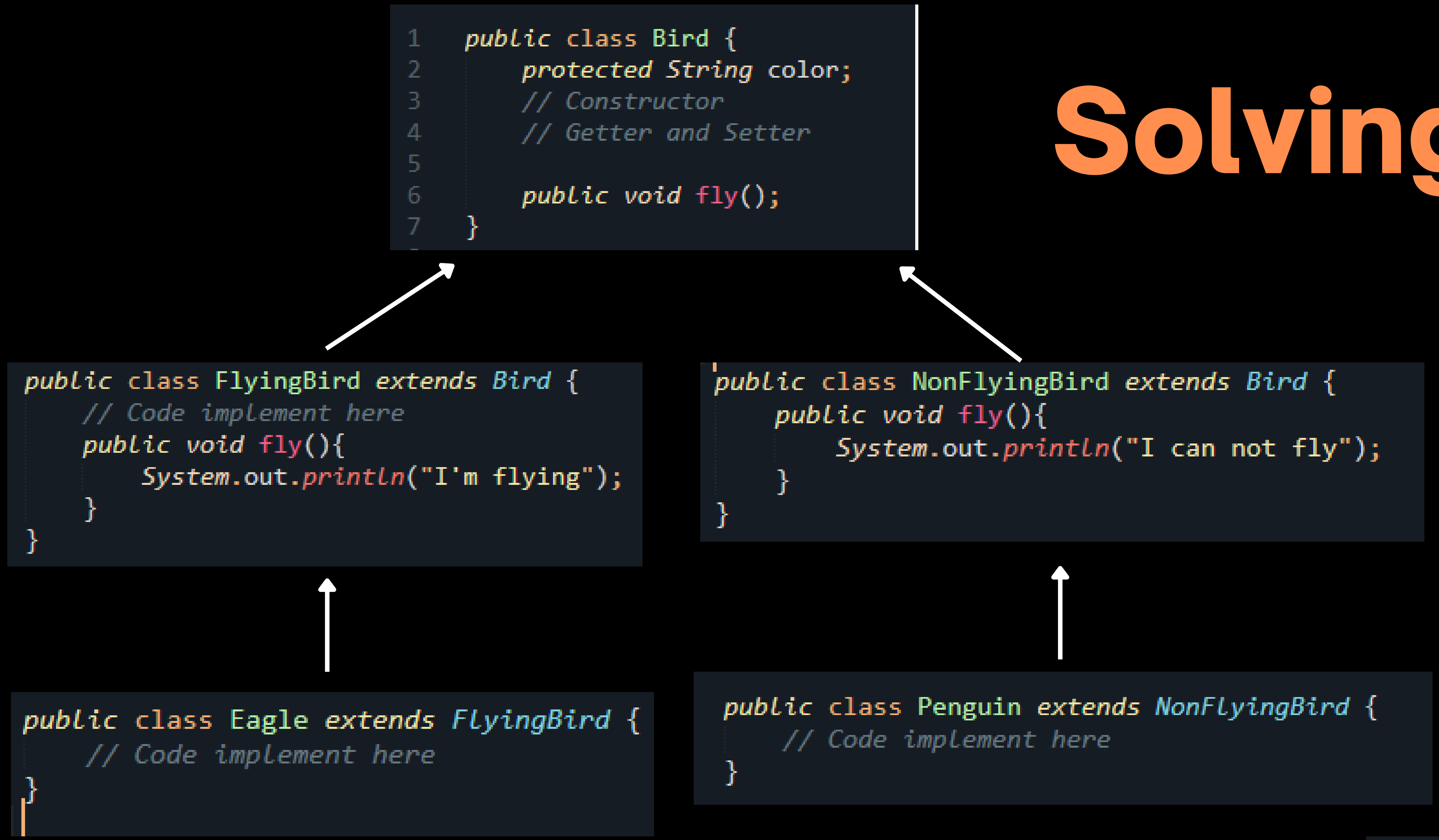
It states that subclasses should be substitutable for their parent classes.

Is it true?



```java
public class Bird {
    protected String color;

    // Constructor
    // Getter and Setter

    public void fly(){
        System.out.println("I'm flying");
    }
}

public class Eagle extends Bird {
    // Code implement here
}

public class Penguin extends Bird {
    // Code implement here
}

public class Test {
    public static void main(String[] args){
        Bird b1 = new Eagle();
        b1.fly();

        Bird b2 = new Penguin();
        b2.fly() // Penguin can not fly
    }
}
```

Solving method



```java
public class Bird {
    protected String color;
    // Constructor
    // Getter and Setter

    public void fly();
}
```

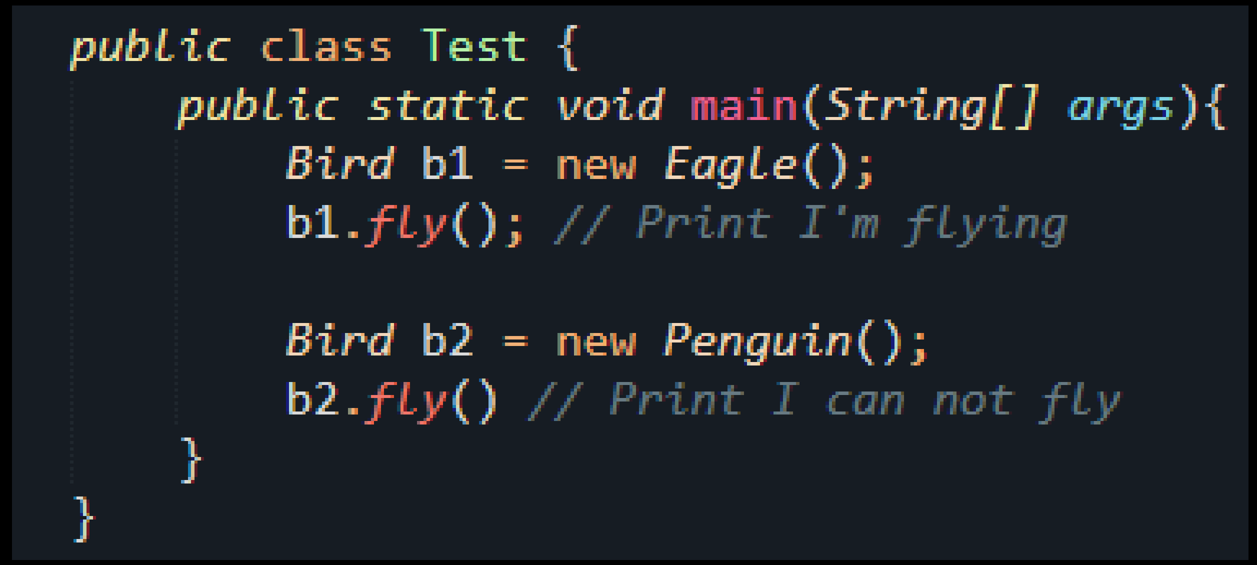
```java
public class FlyingBird extends Bird {
    // Code implement here
    public void fly(){
        System.out.println("I'm flying");
    }
}
```

```java
public class NonFlyingBird extends Bird {
    public void fly(){
        System.out.println("I can not fly");
    }
}
```

```java
public class Eagle extends FlyingBird {
    // Code implement here
}
```

```java
public class Penguin extends NonFlyingBird {
    // Code implement here
}
```

Test again:



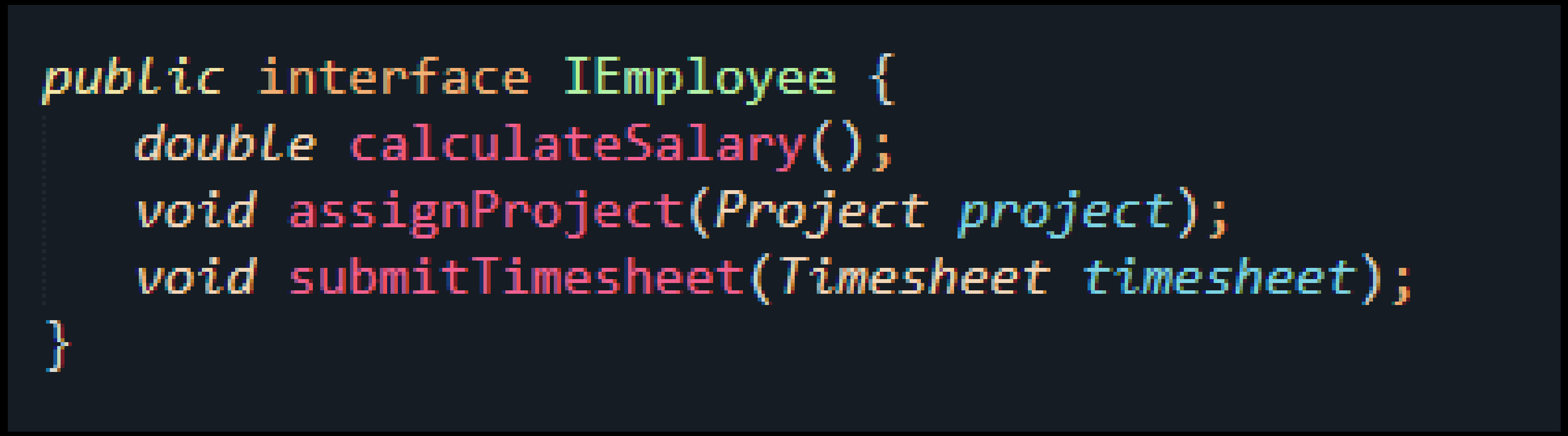
```java
public class Test {
    public static void main(String[] args){
        Bird b1 = new Eagle();
        b1.fly(); // Print I'm flying

        Bird b2 = new Penguin();
        b2.fly() // Print I can not fly
    }
}
```

4. Interface segreration principle

What is it ?

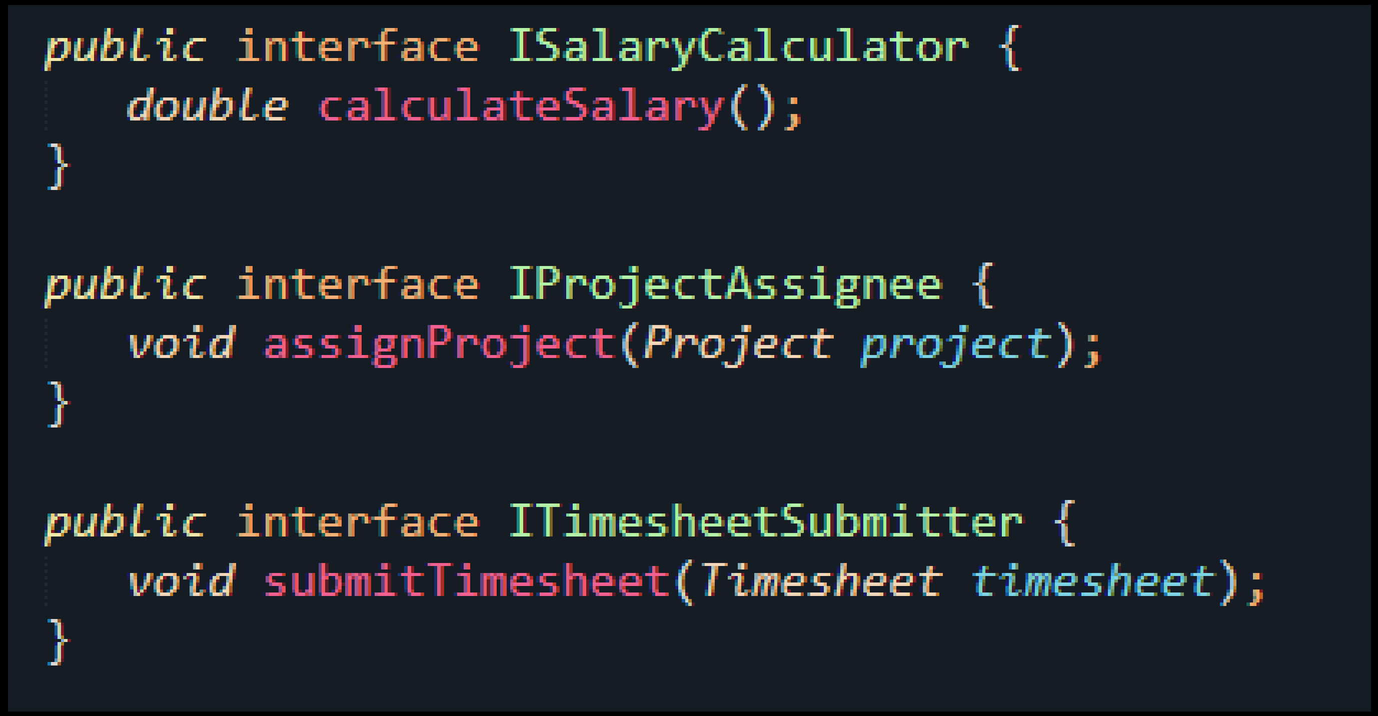
It states that instead of initializing a super big interface class with hundreds of methods, we should seperate it into smaller one with specific purpose.

Bad

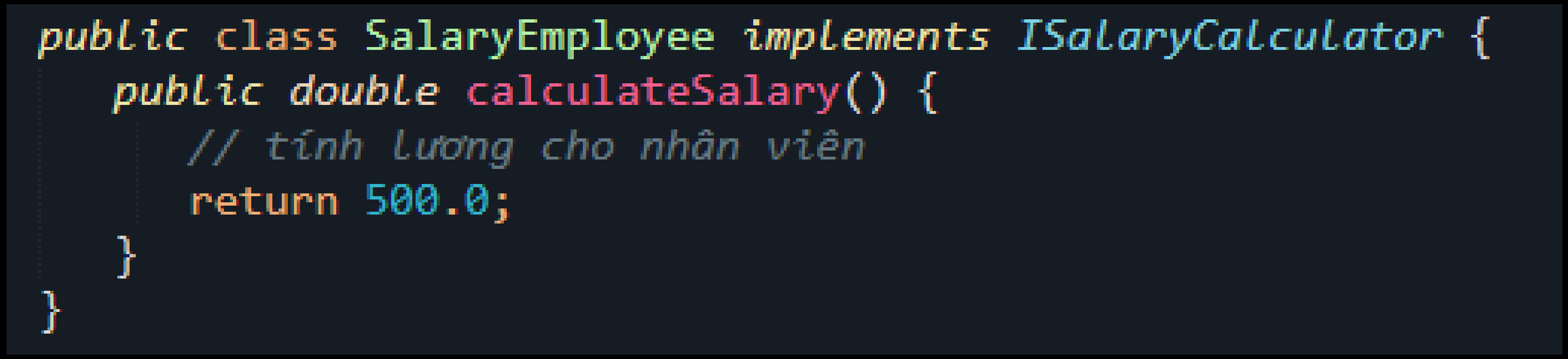


```java
public interface IEmployee {
    double calculateSalary();
    void assignProject(Project project);
    void submitTimesheet(Timesheet timesheet);
}
```

Good



```java
public interface ISalaryCalculator {
    double calculateSalary();
}

public interface IProjectAssignee {
    void assignProject(Project project);
}

public interface ITimesheetSubmitter {
    void submitTimesheet(Timesheet timesheet);
}
```



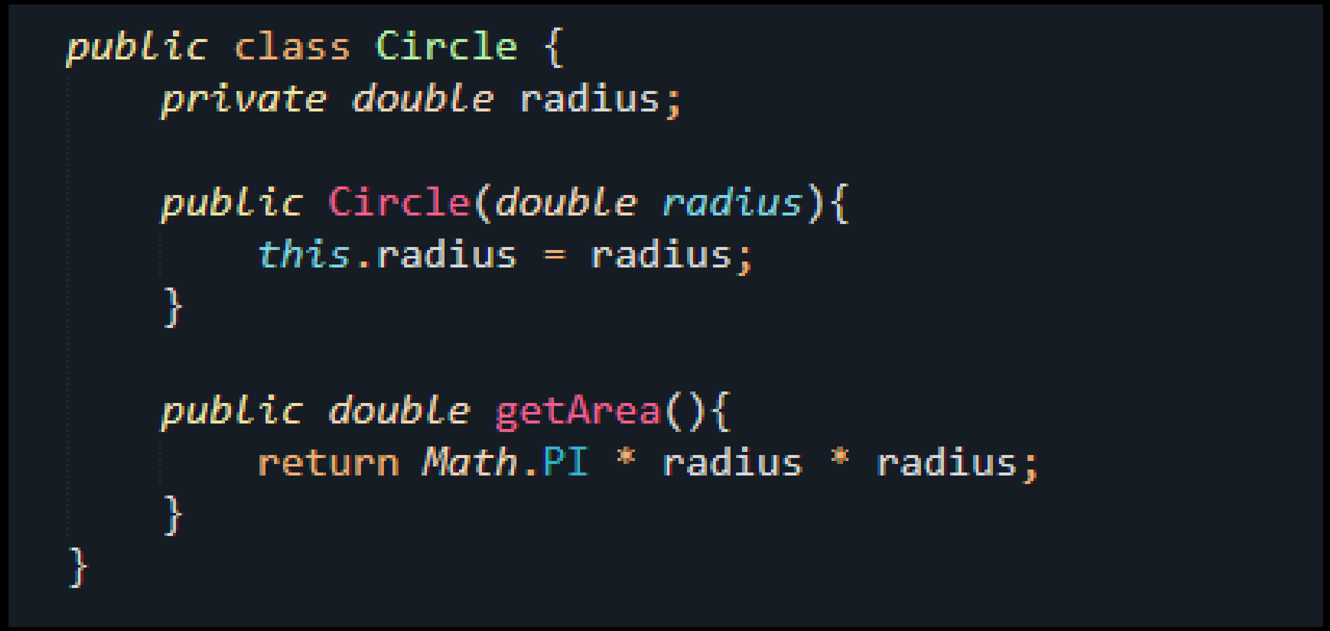
```java
public class SalaryEmployee implements ISalaryCalculator {
    public double calculateSalary() {
        // tính lương cho nhân viên
        return 500.0;
    }
}
```

5. Dependency Inversion Principle

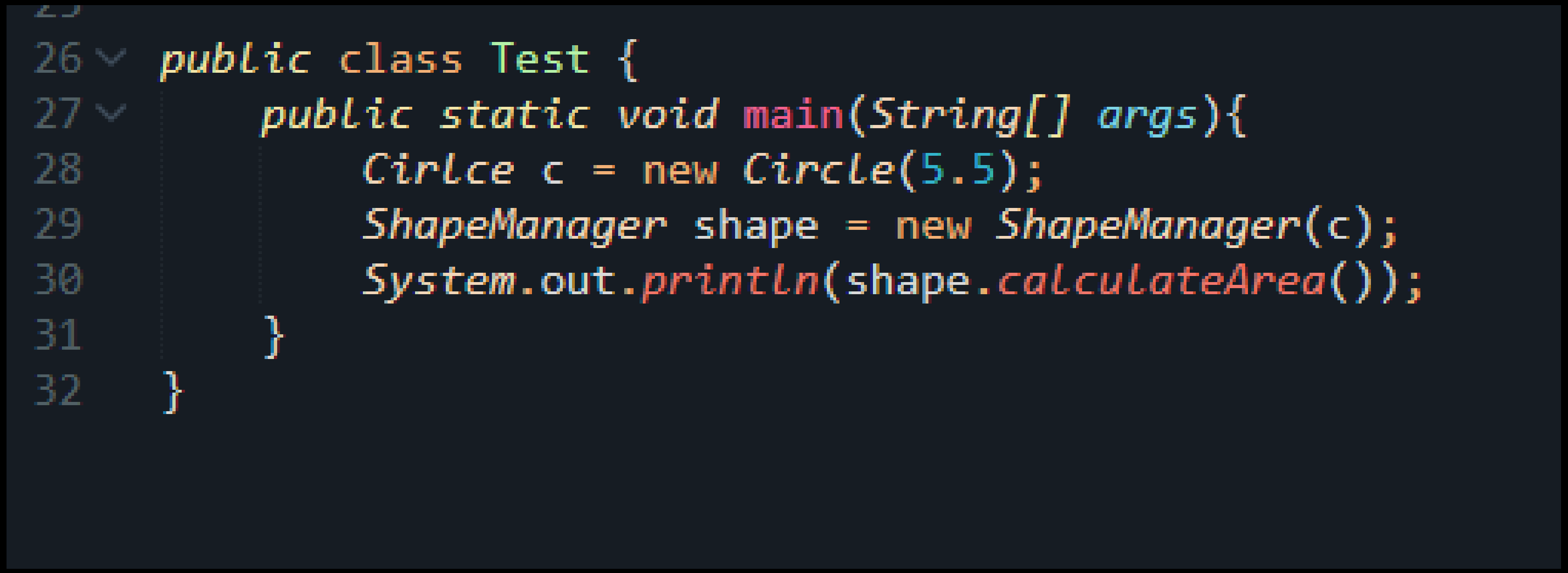
What is it ?

It states that higher-level module should not depend on lower-level module, alternatively both of them should rely on abstraction (interface class).

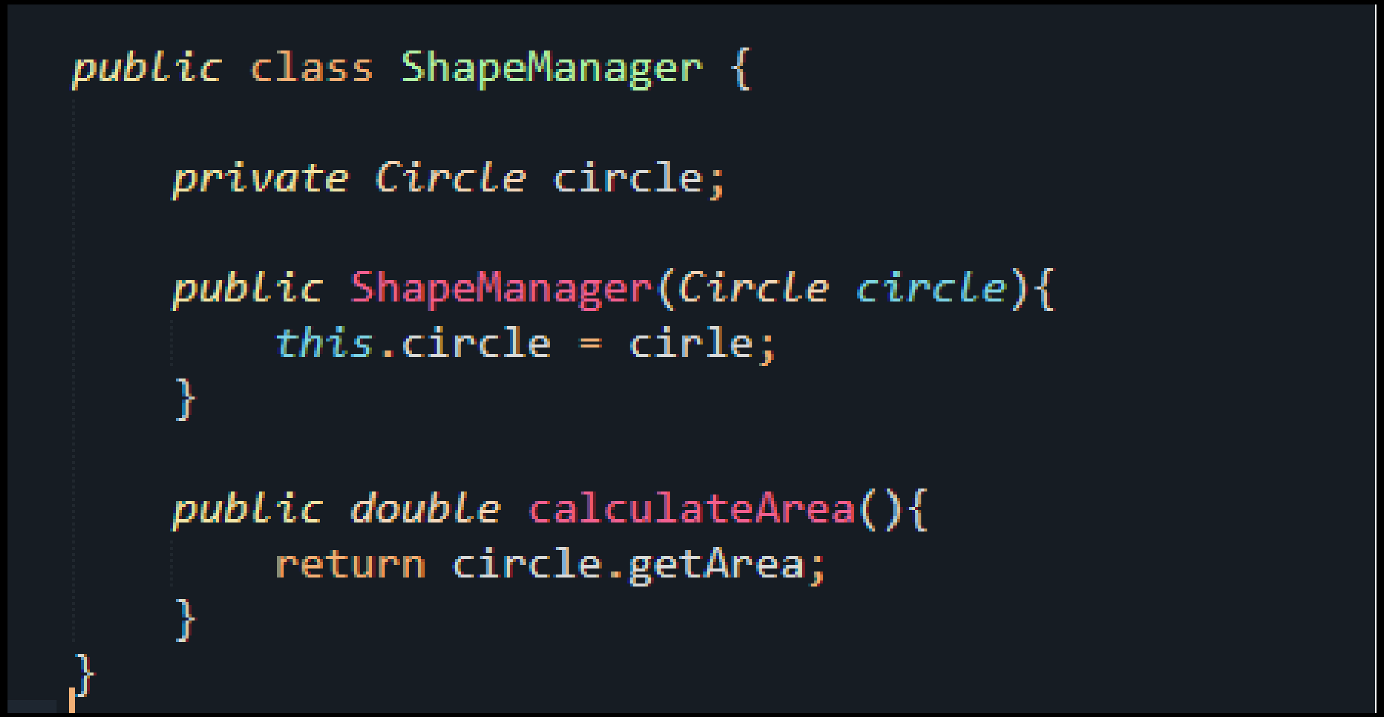
Example:



```java
public class Circle {
    private double radius;

    public Circle(double radius){
        this.radius = radius;
    }

    public double getArea(){
        return Math.PI * radius * radius;
    }
}
```

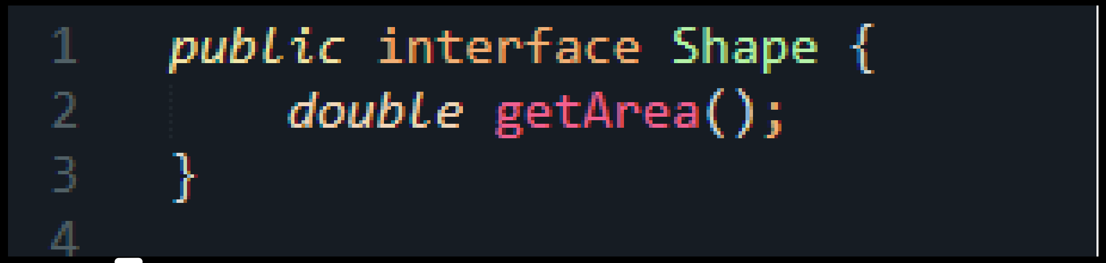


```java
public class Test {
    public static void main(String[] args){
        Cirlce c = new Circle(5.5);
        ShapeManager shape = new ShapeManager(c);
        System.out.println(shape.calculateArea());
    }
}
```

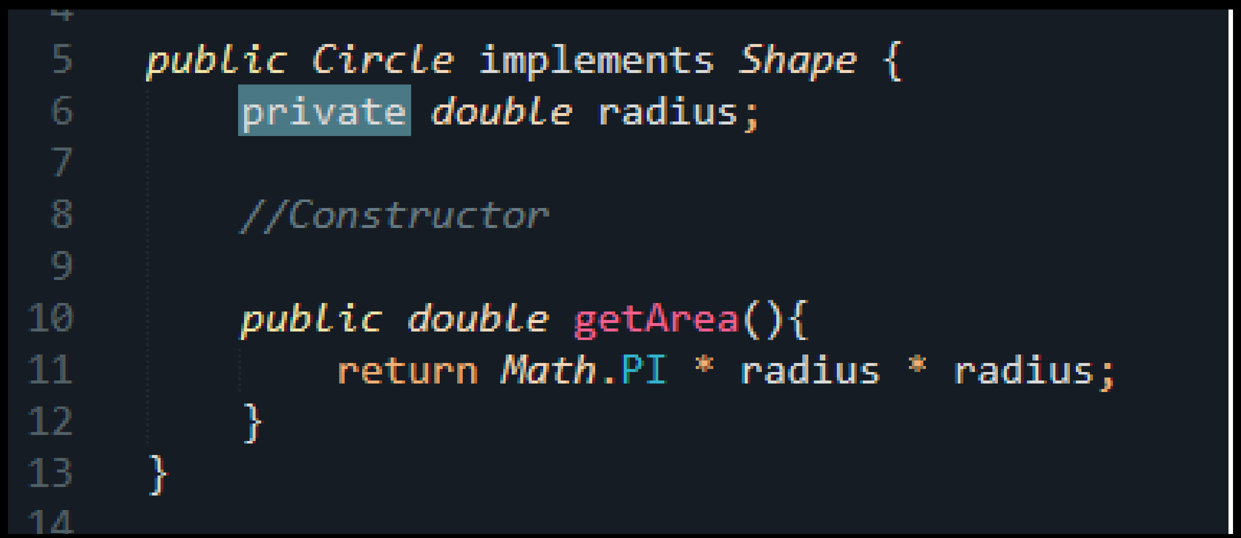


```java
public class ShapeManager {

    private Circle circle;

    public ShapeManager(Circle circle){
        this.circle = cirle;
    }

    public double calculateArea(){
        return circle.getArea;
    }
}
```

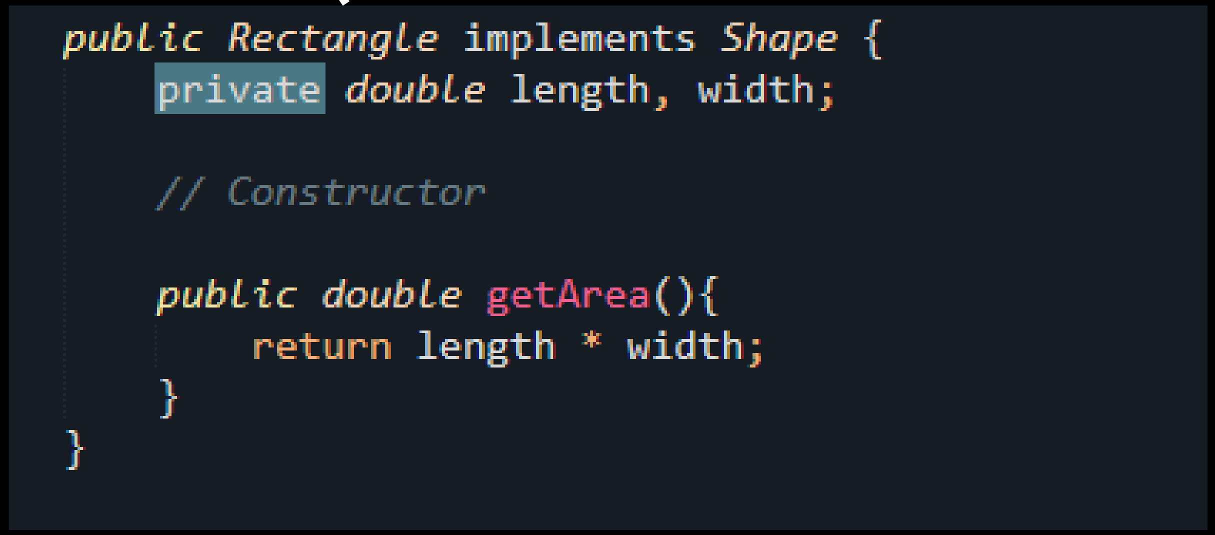
What happens if we extend the ShapeManager to calculate more geometries?



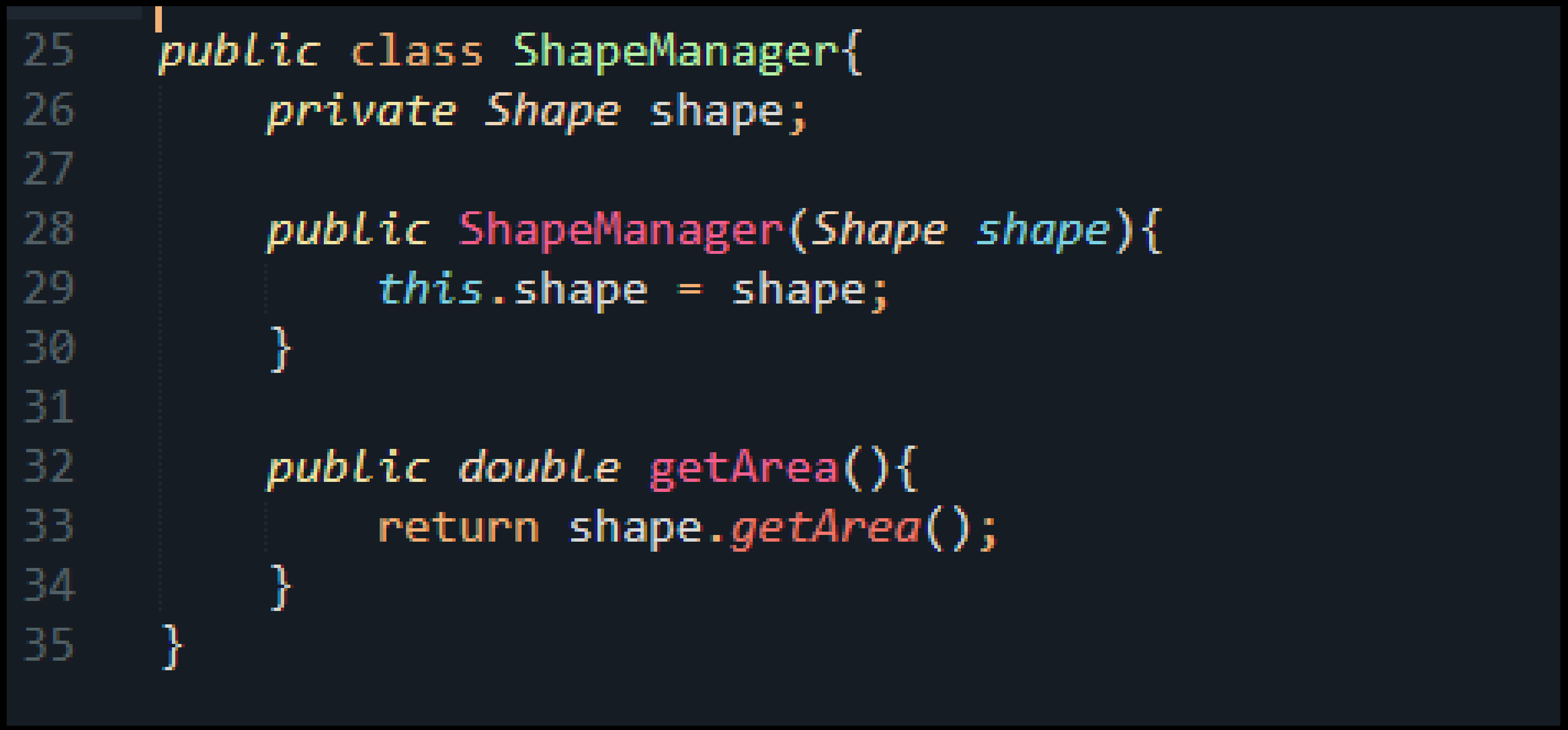
```java
public interface Shape {
    double getArea();
}
```



```java
public Circle implements Shape {
    private double radius;

    //Constructor

    public double getArea(){
        return Math.PI * radius * radius;
    }
}
```



```java
public Rectangle implements Shape {
    private double length, width;

    // Constructor

    public double getArea(){
        return length * width;
    }
}
```



```java
public class ShapeManager{
    private Shape shape;

    public ShapeManager(Shape shape){
        this.shape = shape;
    }

    public double getArea(){
        return shape.getArea();
    }
}
```

Here, we see that ShapeManager no longer depend on lower level module, it now working with interface class

In conclusion, we should keepthis set of rules in mind when designing, programming so that it can be more readable, extendable and maintainable

Thanks for listenning