# SOLID Principle

## S → Single-responsibility principle

This principle states that "**A class should have only one reason to change**" which means every class should have a single responsibility or single job or single purpose. In other words, a class should have only one job or purpose within the software system.

### A class should have only one reason to change.

❌ **Bad Example: One class doing multiple things**

```
class Report {
    public String generateReport() {
        return "Report Data";
    }

    public void saveToFile(String content) {
        System.out.println("Saving to file: " + content);
    }

    public void sendEmail(String content) {
        System.out.println("Sending email: " + content);
    }
}
```

✅ **Good Example: Split responsibilities**

```
class Report {
    public String generateReport() {
        return "Report Data";
    }
}
```

```
class FileSaver {
    public void saveToFile(String content) {
        System.out.println("Saving to file: " + content);
    }
}

class EmailSender {
    public void sendEmail(String content) {
        System.out.println("Sending email: " + content);
    }
}
```

---

## O → Open/Closed Principle (OCP)

This principle states that "**Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification**" which means you should be able to extend a class behavior, without modifying it.

### Classes should be open for extension but closed for modification.

❌ **Bad: Modify class every time a new shape is added**

```
class AreaCalculator {
    public double calculate(Object shape) {
        if (shape instanceof Circle) {
            return Math.PI * ((Circle) shape).radius * ((Circle) shape).radius;
        } else if (shape instanceof Rectangle) {
            return ((Rectangle) shape).length * ((Rectangle) shape).width;
        }
        return 0;
    }
}
```

✅ **Good: Use abstraction (extend without modify)**

```
interface Shape {
    double area();
}

class Circle implements Shape {
    double radius;
    Circle(double radius) { this.radius = radius; }
    public double area() { return Math.PI * radius * radius; }
}

class Rectangle implements Shape {
    double length, width;
    Rectangle(double length, double width) { this.length = length; this.width =
width; }
    public double area() { return length * width; }
}

class AreaCalculator {
    public double calculate(Shape shape) {
        return shape.area();
    }
}
```

---

## L → The Liskov Substitution Principle (LSP)

The Liskov substitution principle is one of the most important principles to adhere to in object-oriented programming (OOP). It was introduced by the computer scientist Barbara Liskov in 1987 in a paper she co-authored with Jeannette Wing.

The principle states that child classes or subclasses must be substitutable for their parent classes or super classes. In other words, the child class must be able to replace the parent class. This has the advantage of letting you know what to expect from your code.

## Child classes should be replaceable for their parent class without breaking functionality.

❌ **Bad Example: Subclass breaks behavior**

```
class Bird {
    void fly() {
        System.out.println("I can fly");
    }
}

class Ostrich extends Bird {
    @Override
    void fly() {
        throw new UnsupportedOperationException("Ostrich cannot fly");
    }
}
```

✅ **Good Example: Correct hierarchy**

```
interface Bird { }

interface FlyableBird extends Bird {
    void fly();
}

class Sparrow implements FlyableBird {
    public void fly() {
        System.out.println("Sparrow flying");
    }
}

class Ostrich implements Bird {
    // no fly() method because it can't fly
}
```

# I → Interface Segregation Principle (ISP)

This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that "**do not force any client to implement an interface which is irrelevant to them**". Here your main goal is to focus on avoiding fat interface and give preference to many small client-specific interfaces. You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.

Let's understand Interface Segregation Principle using an example:

> Suppose if you enter a restaurant and you are pure vegetarian. The waiter in that restaurant gave you the menu card which includes vegetarian items, non-vegetarian items, drinks, and sweets.

- In this case, as a customer, you should have a menu card which includes only vegetarian items, not everything which you don't eat in your food. Here the menu should be different for different types of customers.
- The common or general menu card for everyone can be divided into multiple cards instead of just one. Using this principle helps in reducing the side effects and frequency of required changes.

## Do not force a class to implement interfaces it doesn’t use.

❌ **Bad: One large interface**

```
interface Worker {
    void work();
    void eat();
}
class Robot implements Worker {
    public void work() {}
    public void eat() {} // Robots don't eat!
}
```

✅ **Good: Split into smaller interfaces**

```
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class Human implements Workable, Eatable {
    public void work() {}
    public void eat() {}
}

class Robot implements Workable {
    public void work() {}
}
```

## D → Dependency Inversion Principle (DIP)

The principle states that high-level modules should not depend on low-level modules. Instead, they should both depend on abstractions. Additionally, abstractions should not depend on details, but details should depend on abstractions.

### High-level modules should not depend on low-level modules. Both should depend on abstractions.

❌ **Bad: High-level class depends on concrete class**

```
class Keyboard { }
class Monitor { }

class Computer {
    private Keyboard keyboard;
    private Monitor monitor;
```

```
    public Computer() {
        this.keyboard = new Keyboard();
        this.monitor = new Monitor();
    }
}
```

✅ **Good: Depend on Interfaces (Abstractions)**

```
interface Keyboard { }
interface Monitor { }

class MechanicalKeyboard implements Keyboard { }
class LEDMonitor implements Monitor { }

class Computer {
    private Keyboard keyboard;
    private Monitor monitor;

    public Computer(Keyboard keyboard, Monitor monitor) {
        this.keyboard = keyboard;
        this.monitor = monitor;
    }
}
```

---

# 🎯 Final Summary

| Principle | Key Idea |
|---|---|
| SRP | One class = One job |
| OCP | Extend behavior, don't modify existing code |
| LSP | Subclasses must behave like their base class |

| Principle | Key Idea |
| --- | --- |
| ISP | Many small, specific interfaces are better than one big interface |
| DIP | Code should depend on interfaces, not concrete implementations |