# SOLID

# What are the SOLID principles in Java?

SOLID principles are object-oriented design concepts relevant to software development. SOLID is an acronym for five other class-design principles: Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle.

| Principle | Description |
| --- | --- |
| **Single Responsibility Principle** | Each class should be responsible for a single part or functionality of the system. |
| **Open-Closed Principle** | Software components should be open for extension, but not for modification. |

| | |
|---|---|
| **Liskov Substitution Principle** | Objects of a superclass should be replaceable with objects of its sub-classes without breaking the system. |
| **Interface Segregation Principle** | No client should be forced to de-pend on methods that it does not use. |
| **Dependency Inversion Principle** | High-level modules should not de-pend on low-level modules, both should depend on abstractions. |

# Examples

# 1. Single responsibility principle

Every class in Java should have a single job to do. To be precise, there should only be one reason to change a class. Here's an example of a Java class that does not follow the single responsibility principle (SRP):

```
public class Vehicle {
    public void printDetails() {}
    public double calculateValue() {}
    public void addVehicleToDB() {}
}
```

The `Vehicle` class has three separate responsibilities: reporting, calculation, and database. By applying SRP, we can separate the above class into three classes with separate responsibilities.

```
1   public class UserSettingService
2   {
3   public void changeEmail(User user)
4   {
5   if(checkAccess(user))
6   {
7   //Grant option to change
8   }
9   }
10  public boolean checkAccess(User user)
11  {
12  //Verify if the user is valid.
13  }
14  }
```

```
public class UserSettingService
{
    public void changeEmail(User user)
   {
     if(SecurityService.checkAccess(user))
    {
//Grant option to change
    }
 }

}

public class SecurityService
{
public static boolean checkAccess(User user)
{
//check the access.
}
}
```

# 2. Open-closed principle

Software entities (e.g., classes, modules, functions) should be *open* for an extension, but *closed* for modification.

Consider the below method of the class `VehicleCalculations`:

```
public class VehicleCalculations {
    public double calculateValue(Vehicle v) {
        if (v instanceof Car) {
            return v.getValue() * 0.8;
        if (v instanceof Bike) {
            return v.getValue() * 0.5;

    }
}
```

Suppose we now want to add another subclass called `Truck`. We would have to modify the above class by adding another if statement, which goes against the Open-Closed Principle.

A better approach would be for the subclasses `Car` and `Truck` to override the `calculateValue` method:

```
public class Vehicle {
    public double calculateValue() {...}
}
public class Car extends Vehicle {
    public double calculateValue() {
        return this.getValue() * 0.8;
}
public class Truck extends Vehicle{
    public double calculateValue() {
        return this.getValue() * 0.9;
}
```

Adding another `Vehicle` type is as simple as making another subclass and extending from the `Vehicle` class.

```
public class Rectangle
{
public double length;
public double width;
}
```

```
public class AreaCalculator
{
  public double calculateRectangleArea(Rectangle rectangle)
 {
  return rectangle.length *rectangle.width;
 }
}
```

```
public class Circle
{
public double radius;
}
```

```
public class AreaCalculator

{
   public double calculateRectangleArea(Rectangle rectangle)
  {
     return rectangle.length *rectangle.width;
     }
public double calculateCircleArea(Circle circle)

    {
return (22/7)*circle.radius*circle.radius;
  }
}
```

```
public interface Shape
{
public double calculateArea();
}

public class Rectangle implements Shape
{
double length;
double width;
public double calculateArea()
{
return length * width;
}
}

public class Circle implements Shape
{
public double radius;
public double calculateArea()
{
return (22/7)*radius*radius;
}
}
```

```
public class AreaCalculator
{
public double calculateShapeArea(Shape shape)
{
return shape.calculateArea();
}
}
```

# 3. Liskov substitution principle

The **Liskov Substitution Principle (LSP)** applies to inheritance hierarchies such that derived classes must be completely substitutable for their base classes.

Consider a typical example of a `Square` derived class and `Rectangle` base class:

```
public class Rectangle {
    private double height;
    private double width;
    public void setHeight(double h) { height = h; }
    public void setWidht(double w) { width = w; }
    ...
}
public class Square extends Rectangle {
    public void setHeight(double h) {
        super.setHeight(h);
        super.setWidth(h);
    }
    public void setWidth(double w) {
        super.setHeight(w);
        super.setWidth(w);
    }
}
```

The above classes do not obey LSP because you cannot replace the `Rectangle` base class with its derived class `Square`. The `Square` class has extra constraints, i.e., the height and width must be the same. Therefore, substituting `Rectangle` with `Square` class may result in unexpected behavior.

```java
public class Rectangle
{
private int length;
private int breadth;
public int getLength()
{
return length;
}
public void setLength(int length)
{
this.length = length;
}
public int getBreadth()
{
return breadth;
}
public void setBreadth(int breadth)
{
this.breadth = breadth;
}
public int getArea()
{
return this.length * this.breadth;
}
}
```

```java
public class Square extends Rectangle
{

public void setBreadth(int breadth)
{
super.setBreadth(breadth);
super.setLength(breadth);
}

public void setLength(int length)
{
super.setLength(length);
super.setBreadth(length);
}
}
```

```
public class LSPDemo
{
public void calculateArea(Rectangle r)
{
r.setBreadth(2);
r.setLength(3);
assert r.getArea() == 6 : printError("area", r);
assert r.getLength() == 3 : printError("length", r);
assert r.getBreadth() == 2 : printError("breadth", r);
}
private String printError(String errorIdentifer, Rectangle r)
{
return "Unexpected value of " + errorIdentifer + " for instance of " + r.getCl
}
public static void main(String[] args)
{
LSPDemo lsp = new LSPDemo();
// An instance of Rectangle is passed
lsp.calculateArea(new Rectangle());
// An instance of Square is passed
lsp.calculateArea(new Square());
}
}
```

# 4. Interface segregation principle

The **Interface Segregation Principle (ISP)** states that clients should not be forced to depend upon interface members they do not use. In other words, do not force any client to implement an interface that is irrelevant to them.

Suppose there's an interface for vehicle and a `Bike` class:

```java
public interface Vehicle {
    public void drive();
    public void stop();
    public void refuel();
    public void openDoors();
}
public class Bike implements Vehicle {

    // Can be implemented
    public void drive() {...}
    public void stop() {...}
    public void refuel() {...}

    // Can not be implemented
    public void openDoors() {...}
}
```

As you can see, it does not make sense for a `Bike` class to implement the `openDoors()` method as a bike does not have any doors! To fix this, ISP proposes that the interfaces be broken down into multiple, small cohesive interfaces so that no class is forced to implement any interface, and therefore methods, that it does not need.

```java
public interface RestaurantInterface
{
  public void acceptOnlineOrder();
  public void takeTelephoneOrder();
  public void payOnline();
  public void walkInCustomerOrder();
  public void payInPerson();
}
```

```java
public class OnlineClientImpl implements RestaurantInterface
{
public void acceptOnlineOrder()
{
//logic for placing online order
}
public void takeTelephoneOrder()
{
//Not Applicable for Online Order
throw new UnsupportedOperationException();
}
public void payOnline()
{
//logic for paying online
}
public void walkInCustomerOrder()
{
//Not Applicable for Online Order
throw new UnsupportedOperationException();
}
public void payInPerson() {
//Not Applicable for Online Order
throw new UnsupportedOperationException();
}
}
```

# 5. Dependency inversion principle

The **Dependency Inversion Principle (DIP)** states that we should depend on abstractions (interfaces and abstract classes) instead of concrete implementations (classes). The abstractions should not depend on details; instead, the details should depend on abstractions.

Consider the example below. We have a `Car` class that depends on the concrete `Engine` class; therefore, it is not obeying DIP.

```
public class Car {
    private Engine engine;
    public Car(Engine e) {
        engine = e;
    }
    public void start() {
        engine.start();
    }
}
public class Engine {
   public void start() {...}
}
```

The code will work, for now, but what if we wanted to add another engine type, let's say a diesel engine? This will require refactoring the `Car` class. However, we can solve this by introducing a layer of abstraction. Instead of `Car` depending directly on `Engine`, let's add an interface:

```
public interface Engine {
    public void start();
}
```

Now we can connect any type of `Engine` that implements the Engine interface to the `Car` class:

```
public class Car {
    private Engine engine;
    public Car(Engine e) {
        engine = e;
    }
    public void start() {
        engine.start();
    }
}
public class PetrolEngine implements Engine {
   public void start() {...}
}
public class DieselEngine implements Engine {
   public void start() {...}
}
```

```
public class Student
{
    private Address address;
    public Student()
   {
     address = new Address();
    }
}
```

In the above example, Student class requires an Address object and it is responsible for initializing and using the Address object. If Address class is changed in future then we have to make changes in Student class also. This makes the tight coupling between Student and Address objects. We can resolve this problem using the dependency inversion design pattern. i.e. Address object will be implemented independently and will be provided to Student when Student is instantiated by using constructor-based or setter-based dependency inversion.