✅ 1. S – Single Responsibility Principle (SRP)

🔹 Definition:

A class should have only one responsibility or one reason to change.

Ex: **A chef only cooks**, a **waiter only serves**, and a **cashier only bills** — each has one job.

🎯 Purpose:

Makes code clean and focused

Easier to update or test one thing at a time

🔍 Example (Problem):

public class Restaurant {

public void bill() {  
 System.*out*.println("Cashier is billing the order.");  
}

public void cook() {  
 System.*out*.println("Chef is cooking");  
}

public void serve() {  
 System.*out*.println("Waiter is serving food.");  
}

public static void main(String[]args){

Restaurant r = new Restaurant();

r.bill()

r.cook();

r.serve();  
 }  
}

🔴 Problem: One class is doing too many things.

✅ Fixed Example (SRP):

public class Cashier {  
 public void bill() {  
 System.*out*.println("Cashier is billing the order.");  
 }  
}

public class Chef {  
 public void cook() {  
 System.*out*.println("Chef is cooking");  
 }  
}

public class Waiter {  
 public void serve() {  
 System.*out*.println("Waiter is serving food.");  
 }  
}

public class Restaurant {  
 public static void main(String[]args){  
  
 Chef chef = new Chef();  
 Waiter waiter = new Waiter();  
 Cashier cashier = new Cashier();  
  
 chef.cook();  
 waiter.serve();  
 cashier.bill();  
 }  
}

Each class has one job only ✅

✅ 2. O – Open/Closed Principle (OCP)

🔹 Definition:

**A class should be open for extension but closed for modification.**  
This means you can **add new functionality** by **extending the class**, not by **changing existing code**.

🎯 Purpose:

Add new features without touching old code

Prevent breaking working code

🔍 Example (Bad OCP):

public class NotificationService {

public void notifyUser(String type, String message) {

if (type.equals("email")) {

System.out.println("Sending Email: " + message);

} else if (type.equals("sms")) {

System.out.println("Sending SMS: " + message);

}

// ❌ Every time a new type is added, we need to modify this method

else if (type.equals("push")) {

System.out.println("Sending Push Notification: " + message);

}

}

}

#Main :

public class Phone {

public static void main(String[] args) {

NotificationService service = new NotificationService();

service.notifyUser("email", "Welcome without OCP!");

service.notifyUser("sms", "This breaks OCP!");

service.notifyUser("push", "Push added by modifying existing code!");

}

}

**🚫 Problems With This Approach:**

1. **Breaks OCP**:  
   You have to **modify NotificationService every time** a new notification type is added.
2. **Tightly Coupled**:  
   All logic is crammed into a single method, making the class harder to test and maintain.
3. **No Polymorphism**:  
   The system doesn't use interfaces or abstraction. So you lose the benefits of **extensibility** and **code reusability**.
4. **Error-Prone**:  
   Mistyped strings like "Email", "EMAIL", or "e-mail" could break the logic silently.

✅ Fixed Example (OCP):

public interface Notification {  
 void send(String message);  
}

public class EmailNotification implements Notification{  
 public void send(String message) {  
 System.*out*.println("Sending Email: " + message);  
 }  
}

public class SmsNotification implements Notification{  
 public void send(String message) {  
 System.*out*.println("Sending SMS: " + message);  
 }  
}

public class NotificationService {  
 public void notifyUser(Notification notification, String message) {  
 notification.send(message);  
 }  
}

public class Phone {  
 public static void main(String[] args) {  
 NotificationService service = new NotificationService();  
  
 service.notifyUser(new EmailNotification(), "Welcome to OCP!");  
 service.notifyUser(new SmsNotification(), "OCP makes code flexible!");  
 }  
}

✅ Now we can add SmsNotification etc. without changing the Existing class

✅ 3. L – Liskov Substitution Principle (LSP)

🔹 Definition:

Child classes must be replaceable for their parent class without breaking the program.

🎯 Purpose:

Subclasses should behave like the base class

Prevent incorrect or broken inheritance

🔍 Example (Bad LSP):

java

Copy

Edit

class Bird {

public void fly() {

System.out.println("Bird can fly");

}

}

class Ostrich extends Bird {

public void fly() {

throw new UnsupportedOperationException("Ostrich can't fly");

}

}

🔴 Problem: Ostrich is a bird, but it can't fly. Using it will break code.

✅ Fixed Example (LSP):

java

Copy

Edit

class Bird {}

class FlyingBird extends Bird {

public void fly() {

System.out.println("Flying bird");

}

}

class Parrot extends FlyingBird {}

class Ostrich extends Bird {}

Now, only birds that can fly use FlyingBird class ✅

✅ 4. I – Interface Segregation Principle (ISP)

🔹 Definition:

Don’t force classes to implement methods they don’t use.

🎯 Purpose:

Keep interfaces small and specific

Avoid unnecessary code in classes

🔍 Example (Bad ISP):

java

Copy

Edit

interface Animal {

void walk();

void fly();

void swim();

}

class Dog implements Animal {

public void walk() {}

public void fly() {} // Dog can't fly ❌

public void swim() {}

}

🔴 Problem: Dog is forced to implement fly()

✅ Fixed Example (ISP):

java

Copy

Edit

interface Walkable {

void walk();

}

interface Flyable {

void fly();

}

interface Swimmable {

void swim();

}

class Dog implements Walkable, Swimmable {

public void walk() {}

public void swim() {}

}

Now Dog implements only what it actually uses ✅

✅ 5. D – Dependency Inversion Principle (DIP)

🔹 Definition:

Depend on abstractions, not concrete classes.

🎯 Purpose:

Make high-level code independent from low-level code

Easier to swap or test components

🔍 Example (Bad DIP):

java

Copy

Edit

class LightBulb {

void turnOn() {}

}

class Switch {

private LightBulb bulb = new LightBulb();

public void operate() {

bulb.turnOn();

}

}

🔴 Problem: Switch is tightly bound to LightBulb

✅ Fixed Example (DIP):

java

Copy

Edit

interface Switchable {

void turnOn();

void turnOff();

}

class LightBulb implements Switchable {

public void turnOn() {}

public void turnOff() {}

}

class Fan implements Switchable {

public void turnOn() {}

public void turnOff() {}

}

class Switch {

private Switchable device;

public Switch(Switchable device) {

this.device = device;

}

public void operate() {

device.turnOn();

}

}

Now you can use Switch with LightBulb, Fan, etc. ✅

🧠 Summary:

Principle Meaning Java Example

SRP One class = one job Split Invoice into 3 classes

OCP Extend behavior, don’t modify it Add new services without changing Notification

LSP Subclass should replace parent Ostrich should not extend FlyingBird

ISP Don’t force unwanted methods Dog shouldn’t implement fly()

DIP Depend on interface, not class Switch works with any Switchable device