Chapter 1

SOLID PRINCIPLES(single responsibility, open/closed, liskov substitution, interface segmented,dependency inversion)

Help us to write better code:
Avoid duplicate code
Easy to maintain
Easy to understand
Flexible software
Reduce Complexity

1) The Single Responsibility Principle (SRP) is the first principle of SOLID design principles in object-oriented programming.
   Definition
   A class should have only one reason to change - meaning it should have only one job or responsibility.
   Key Concepts
   What it means:

   Each class should focus on a single task or functionality
   If a class has multiple responsibilities, changes to one responsibility can affect the others
   Separating concerns makes code more maintainable and testable

   Example - Violation of SRP

```
javaclass Employee {
    private String name;
    private double salary;

    // Responsibility 1: Employee data management
    public void setName(String name) { this.name = name; }
    public String getName() { return name; }

    // Responsibility 2: Salary calculation
```

```
    public double calculatePay() {
        // Complex salary calculation logic
        return salary * 1.1;
    }

    // Responsibility 3: Database operations
    public void save() {
        // Database saving logic
        System.out.println("Saving employee to database");
    }

    // Responsibility 4: Reporting
    public void printReport() {
        // Report generation logic
        System.out.println("Employee Report: " + name);
    }
}
```

Example - Following SRP

```
java// Responsibility 1: Employee data
class Employee {
    private String name;
    private double salary;

    public void setName(String name) { this.name = name; }
    public String getName() { return name; }
    public double getSalary() { return salary; }
}

// Responsibility 2: Salary calculation
class PayrollCalculator {
    public double calculatePay(Employee employee) {
        return employee.getSalary() * 1.1;
    }
}

// Responsibility 3: Database operations
class EmployeeRepository {
    public void save(Employee employee) {
        System.out.println("Saving employee to database");
    }
}
```

```
// Responsibility 4: Reporting
class EmployeeReportGenerator {
  public void printReport(Employee employee) {
    System.out.println("Employee Report: " + employee.getName());
  }
}
```

---

2) The **Open/Closed Principle (OCP)** is one of the **SOLID principles** in object-oriented design.

Definition:
A software module (class, function, or component) should be:

- **Open for extension** → You can add new functionality when requirements change.
- **Closed for modification** → You should not change existing, tested, and working code.

\

Why it matters:

- Prevents breaking existing code when adding new features.
- Improves maintainability and scalability.
- Encourages writing flexible and reusable code.

Example (without OCP ❌):

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

🔴 Problem: If you add a new shape (e.g., Triangle), you must **modify** this class.

Example (with OCP ✅):

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

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

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

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

✅ Now, if you add a **Triangle**, you don't need to touch `AreaCalculator`. You just create a new class that implements `Shape`.

In short:

- **Bad design:** Keep editing the same class whenever a new requirement comes.
- **Good design (OCP):** Design the system so you only **add new code**, not change existing working code.

---

3) Liskov Substitution Principle

A subclass should be **substitutable** for its parent class without breaking the program's behavior.

👉 In other words:

- If **S** is a subclass of **T**, then objects of type **T** should be replaceable with objects of type **S without errors or unexpected results**.

⚡ **Why it matters:**

- Ensures **inheritance is meaningful** (not just code reuse).
- Prevents fragile code and unexpected bugs.
- Promotes reliability and consistency in object-oriented design.

❌ **Example that breaks LSP**

```
class Bird {
  public void fly() {
    System.out.println("Flying...");
  }
}

class Ostrich extends Bird {
  @Override
  public void fly() {
    throw new UnsupportedOperationException("Ostriches can't fly!");
  }
}
```

🔴 Problem: If a method expects a Bird and tries to call .fly(), it will crash when an Ostrich is passed.

👉 Here, **Ostrich is not a true substitute for Bird** → violates LSP.

✅ **Example that follows LSP**

```
interface Bird {
  void eat();
}

interface Flyable {
  void fly();
}

class Sparrow implements Bird, Flyable {
  public void eat() { System.out.println("Sparrow eats seeds."); }
  public void fly() { System.out.println("Sparrow flies high!"); }
}

class Ostrich implements Bird {
  public void eat() { System.out.println("Ostrich eats plants."); }
}
```

✅ Now, Sparrow can fly, Ostrich cannot, and no expectations are broken.
We separated the **Flyable behavior**, so each bird only implements what makes sense.

🔑 **In short:**

- **Don't promise behavior in a base class that subclasses can't deliver.**
- Subclasses must follow the rules of the parent class.
- LSP = "If it looks like a duck and quacks like a duck, it should actually behave like a duck."

---

4) Interface Segregation Principle (ISP)

👉 "Clients should not be forced to depend on methods they do not use."

That means:

Instead of having one big, fat interface, break it into smaller, more specific interfaces.

Classes should only implement the methods that are actually relevant to them.

⚡ Why it matters

Prevents classes from having "dummy" or empty method implementations.

Leads to more focused, reusable, and maintainable code.

Makes systems easier to extend without breaking unrelated parts.

❌ Bad Example (Violates ISP)

```
interface Worker {
  void work();
  void eat();
}

class Robot implements Worker {
  public void work() {
    System.out.println("Robot is working.");
  }
  public void eat() {
    // ❌ Robots don't eat!
    throw new UnsupportedOperationException("Robots don't eat!");
  }
}
```

🔴 Problem: Robot is forced to implement eat() even though it doesn't make sense.

✅ Good Example (Follows ISP)

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

interface Eatable {
   void eat();
}

class Human implements Workable, Eatable {
   public void work() {
      System.out.println("Human is working.");
   }
   public void eat() {
      System.out.println("Human is eating.");
   }
}

class Robot implements Workable {
   public void work() {
      System.out.println("Robot is working.");
   }
}
```

✅ Now, Human implements both work and eat.
✅ Robot only implements work.
👉 Each class depends only on what it needs.

🔑 In short:

ISP = "No class should be forced to implement irrelevant methods."

Break down large, general-purpose interfaces into smaller, role-specific ones.

---

5) Dependency Inversion Principle (DIP)

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

👉 "Abstractions should not depend on details. Details should depend on abstractions."

That means:

High-level code (business logic) shouldn't be tightly coupled to low-level code (implementations).

Instead, both should rely on interfaces or abstract classes.

⚡ Why it matters

Reduces tight coupling → easier to change implementations.

Makes systems more flexible and testable.

Encourages use of Dependency Injection (passing dependencies instead of creating them directly).

❌ Bad Example (Violates DIP)

```
class MySQLDatabase {
  public void save(String data) {
    System.out.println("Saving to MySQL: " + data);
  }
}

class UserService {
  private MySQLDatabase db = new MySQLDatabase(); // ❌ tightly coupled

  public void addUser(String user) {
    db.save(user);
  }
}
```

🔴 Problem: UserService depends directly on MySQLDatabase.

If you want to switch to MongoDB, you must modify UserService.

✅ Good Example (Follows DIP)

```java
interface Database {
  void save(String data);
}

class MySQLDatabase implements Database {
  public void save(String data) {
    System.out.println("Saving to MySQL: " + data);
  }
}

class MongoDBDatabase implements Database {
  public void save(String data) {
    System.out.println("Saving to MongoDB: " + data);
  }
}

class UserService {
  private Database db; // depends on abstraction, not concrete class

  public UserService(Database db) {
    this.db = db; // dependency injected
  }

  public void addUser(String user) {
    db.save(user);
  }
}
```

✅ Now:

UserService doesn't care which database is used.

You can inject MySQLDatabase, MongoDBDatabase, or any future database without modifying UserService.

🔑 In short:

DIP = High-level policies shouldn’t depend on low-level details. Both should rely on abstractions.