SOLID Principles in Java

Here's a breakdown of the SOLID principles with corresponding Java code examples:

1. Single Responsibility Principle (SRP)

- **Definition:** A class should have only one reason to change. In other words, a class should have only one job.
- **Bad Example:**

```
class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public void saveToDatabase() {
        // Code to save user to database
        System.out.println("Saving user to database: " + name);
    }

    public void sendEmail() {
        // Code to send email
        System.out.println("Sending email to: " + email);
    }
}
```

    - Problem: The User class has two responsibilities: managing user data and sending emails. If the email sending logic changes, the User class needs to be modified, violating SRP.
- **Good Example:**

```
class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public void saveToDatabase() {
```

```
        // Code to save user to database
        System.out.println("Saving user to database: " + name);
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }
}

class EmailService {
    public void sendEmail(String email, String message) {
        // Code to send email
        System.out.println("Sending email to " + email + ": " + message);
    }
}
```

- Solution: The EmailService class handles the email sending responsibility, separating it from the User class.

2. Open/Closed Principle (OCP)

- **Definition:** Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
- **Bad Example:**

```
class Shape {
    private int type; // 1 for rectangle, 2 for circle
    private int width;
    private int height;
    private int radius;

    public Shape(int type, int width, int height, int radius) {
        this.type = type;
        this.width = width;
        this.height = height;
        this.radius = radius;
    }

    public double getArea() {
        if (type == 1) {
            return width * height;
```

```
        } else if (type == 2) {
            return Math.PI * radius * radius;
        }
        return 0;
    }
}
```

- Problem: If you want to add a new shape, you have to modify the Shape class, violating OCP.

- **Good Example:**

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

class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double getArea() {
        return width * height;
    }
}

class Circle implements Shape {
    private double radius;

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

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

```
//To calculate area:
class AreaCalculator{
  public double calculateArea(Shape shape){
    return shape.getArea();
  }
}
```

- Solution: Define an interface Shape and create classes for each shape (e.g., Rectangle, Circle) that implement the interface. To add a new shape, you create a new class, without modifying the existing ones.

3. Liskov Substitution Principle (LSP)

- **Definition:** Subtypes must be substitutable for their base types.
- **Bad Example:**

```
class Rectangle {
  protected double width;
  protected double height;

  public void setWidth(double width) {
    this.width = width;
  }

  public void setHeight(double height) {
    this.height = height;
  }

  public double getArea() {
    return width * height;
  }
}

class Square extends Rectangle {
  @Override
  public void setWidth(double width) {
    this.width = width;
    this.height = width; // Square's width and height are always equal
  }

  @Override
  public void setHeight(double height) {
    this.width = height;
    this.height = height; // Square's width and height are always equal
  }
```

```
}

// Client code
public class Main {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle();
        rect.setWidth(5);
        rect.setHeight(10);
        System.out.println(rect.getArea()); // Output: 50

        // Substituting Square for Rectangle
        Rectangle square = new Square();
        square.setWidth(5);
        square.setHeight(10);  // This violates the behavior of a square
        System.out.println(square.getArea()); // Output: 100. Should be 5*5 = 25
    }
}
```

    - Problem: The Square class violates LSP because it changes the behavior of the setWidth and setHeight methods. Client code that works correctly with Rectangle may produce unexpected results when used with Square.

- **Good Example:**

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

class Rectangle implements Shape{
    protected double width;
    protected double height;

    public Rectangle(double width, double height){
        this.width = width;
        this.height = height;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public void setHeight(double height) {
        this.height = height;
    }
```

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

class Square implements Shape {
    private double side;

    public Square(double side) {
        this.side = side;
    }

    public void setSide(double side) {
        this.side = side;
    }

    @Override
    public double getArea() {
        return side * side;
    }
}

public class Main {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(5, 10);
        System.out.println(rect.getArea());

        Square square = new Square(5);
        System.out.println(square.getArea());
    }
}
```

- Solution: The Square and Rectangle are defined independently, and both implement the Shape interface. There is no inheritance relationship.

4. Interface Segregation Principle (ISP)

- **Definition:** A client should not be forced to depend on methods it does not use.
- **Bad Example:**

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

```
}

class HumanWorker implements Worker {
    @Override
    public void work() {
        System.out.println("Human is working");
    }

    @Override
    public void eat() {
        System.out.println("Human is eating");
    }
}

class RobotWorker implements Worker {
    @Override
    public void work() {
        System.out.println("Robot is working");
    }

    @Override
    public void eat() {
        // Robot doesn't eat, but we still have to implement this method.
        System.out.println("Robot cannot eat");
    }
}
```

    - Problem: The RobotWorker class is forced to implement the eat() method, even though robots don't eat. This violates ISP.

- **Good Example:**

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

interface Eatable {
    void eat();
}

class HumanWorker implements Workable, Eatable {
    @Override
    public void work() {
        System.out.println("Human is working");
```

```
    }

    @Override
    public void eat() {
        System.out.println("Human is eating");
    }
}

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

    - Solution: Separate the Worker interface into Workable and Eatable interfaces. Now, RobotWorker only needs to implement the Workable interface.

5. Dependency Inversion Principle (DIP)

- **Definition:**
    - 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.
- **Bad Example:**

```
class MySQLDatabase {
    public void connect() {
        System.out.println("Connecting to MySQL database");
    }
}

class UserManager {
    private MySQLDatabase db;

    public UserManager() {
        this.db = new MySQLDatabase();
    }

    public void addUser() {
        db.connect();
        System.out.println("Adding user");
    }
}
```

    - Problem: The UserManager class depends directly on the MySQLDatabase class (a concrete implementation). If you want to switch to a different database, you have to modify UserManager.
- **Good Example:**

```
interface Database {
    void connect();
}

class MySQLDatabase implements Database {
    @Override
    public void connect() {
        System.out.println("Connecting to MySQL database");
    }
}

class OracleDatabase implements Database {
    @Override
    public void connect() {
        System.out.println("Connecting to Oracle database");
    }
}

class UserManager {
    private Database db;

    public UserManager(Database db) {
        this.db = db;
    }

    public void addUser() {
        db.connect();
        System.out.println("Adding user");
    }
}

public class Main {
    public static void main(String[] args) {
        //Can use either MySQL or Oracle, depending on the desired behavior
        Database mySqlDb = new MySQLDatabase();
        UserManager userManager1 = new UserManager(mySqlDb);
        userManager1.addUser();
```

```
        Database oracleDb = new OracleDatabase();
        UserManager userManager2 = new UserManager(oracleDb);
        userManager2.addUser();
    }
}
```

- Solution: The UserManager class depends on the Database interface (an abstraction), not on a specific database implementation. This allows you to easily switch between databases without modifying UserManager.

Benefits of Using SOLID Principles

Here are some key benefits of adhering to SOLID principles in your Java code:

- **Increased Maintainability:** SOLID principles lead to code that is easier to understand, modify, and debug. Each class has a single, well-defined purpose, reducing the complexity of changes.
- **Enhanced Reusability:** When classes are designed with single responsibilities and adhere to principles like LSP and DIP, they can be reused in different parts of the application or in other projects.
- **Improved Scalability:** SOLID code is more adaptable to new features and changing requirements. Adding new functionality can often be done by extending existing code rather than modifying it, minimizing the risk of introducing bugs.
- **Greater Flexibility:** Dependency inversion and interface segregation promote loose coupling, making it easier to change dependencies and adapt to different environments or frameworks.
- **Better Testability:** SOLID code is easier to test because of its modularity and clear separation of concerns. You can write unit tests for individual classes with greater confidence.
- **Reduced Complexity:** By breaking down complex systems into smaller, more manageable components, SOLID principles help to reduce the overall complexity of the codebase.
- **Fewer Bugs:** When code is well-structured and follows SOLID principles, it is less prone to errors and unexpected behavior. Changes in one part of the code are less likely to have unintended consequences in other parts.