3. L - Liskov Substitution Principle (LSP)

Resources

- Video → 1. SOLID Principles with Easy Examples (Hindi) | OOPs SOLID Principles - Low Level Design
- Video 1.1 → 1.1 Liskov Substitution Principle (LSP) with Solution in Java - SOLID Principles of Low Level Design

What is the Liskov Substitution Principle (LSP)?

> ***Liskov Substitution Principle(LSP)*** *states that the* ***"objects of a superclass should be replaceable with objects of its subclasses without breaking the application".***

That means, if class B is a subtype of class A, then we should be able to replace objects of A with B without breaking the behaviour of the program. Subclass should extend the capability of the parent class, not narrow it down.

Code Example 1: Violating LSP

```java

// BAD: This design violates LSP
public interface Bike {

    void turnOnEngine();

    void turnOffEngine();

    void accelerate();

    void applyBrakes();
}

// Subclass of Bike - implements all Bike class behavior
public class MotorCycle implements Bike {
    String company;
    boolean isEngineOn;
    int speed;

    public MotorCycle(String company, int speed) {
        this.company = company;
        this.speed = speed;
    }

    @Override
    public void turnOnEngine() {
        this.isEngineOn = true; // turn on the engine!
```

```java
        System.out.println("Engine is ON!");
    }

    @Override
    public void turnOffEngine() {
        this.isEngineOn = false; // turn off the engine!
        System.out.println("Engine is OFF!");
    }

    @Override
    public void accelerate() {
        this.speed = this.speed + 10; // increase the speed
        System.out.println("MotorCycle Speed: " + this.speed);
    }

    @Override
    public void applyBrakes() {
        this.speed = this.speed - 5; // decrease the speed
        System.out.println("MotorCycle Speed: " + this.speed);
    }
}

// This class violates LSP!
public class Bicycle implements Bike {
    String brand;
    Boolean hasGears;
    int speed;

    public Bicycle(String brand, Boolean hasGears, int speed) {
        this.brand = brand;
        this.hasGears = hasGears;
        this.speed = speed;
    }

    // LSP Violation: Strengthening preconditions
    // Bicycle changes the behavior of turnOnEngine
    @Override
    public void turnOnEngine() {
        throw new AssertionError("Detail Message: Bicycle has no engine!");
    }
    // Bicycle changes the behavior of turnOffEngine
    @Override
    public void turnOffEngine() {
         throw new AssertionError("Detail Message: Bicycle has no engine!");
    }

    @Override
    public void accelerate() {
        this.speed = this.speed + 10; // increase the speed
        System.out.println("Bicycle Speed: " + this.speed);
    }

    @Override
    public void applyBrakes() {
        this.speed = this.speed - 5; // decrease the speed
        System.out.println("Bicycle Speed: " + this.speed);
    }

}

// Usage example - demonstrates the LSP violations
public class Demo {
    public static void main(String[] args) {
        // create the objects
        MotorCycle motorCycle = new MotorCycle("HeroHonda", 10);
        Bicycle bicycle = new Bicycle("Hercules", true, 10);

        // use the objects
```

```
        // Works fine with MotorCycle - implements all Bike class behavior
        motorCycle.turnOnEngine();
        motorCycle.accelerate();
        motorCycle.applyBrakes();
        motorCycle.turnOffEngine();
        // Client expects to be able to see the same behavior with Bicycle
        bicycle.turnOnEngine(); // fails to implement Bike class behavior
        bicycle.accelerate();
        bicycle.applyBrakes();
        bicycle.turnOffEngine(); // fails to implement Bike class behavior
    }
}
```

Problems with the Above Code

- Not all bikes have engines(eBikes and Bicycles), but the base class forces the `turnOnEngine()` method.
- Bicycle throws an exception, violating the expected contract.
- Client code cannot safely treat all Bike subclasses uniformly.

Code Example 2: Follows LSP

Here's the refactored code that follows the Liskov Substitution Principle:

```java
abstract class Bike {

    // All Bikes can do these things
    public abstract void accelerate();

    public abstract void applyBrakes();

}
public interface Engine {
    void turnOnEngine();

    void turnOffEngine();
}

// Subclass of Bike - implements all Bike class behavior
public class MotorCycle extends Bike implements Engine {
    String company;
    boolean isEngineOn;
    int speed;

    public MotorCycle(String company, int speed) {
        this.company = company;
        this.speed = speed;
    }

    @Override
    public void turnOnEngine() {
        this.isEngineOn = true; // turn on the engine!
        System.out.println("Engine is ON!");
    }

    @Override
    public void turnOffEngine() {
        this.isEngineOn = false; // turn off the engine!
        System.out.println("Engine is OFF!");
    }

    public void accelerate() {
        this.speed = this.speed + 10; // increase the speed
        System.out.println("MotorCycle Speed: " + this.speed);
    }
```

```java

    public void applyBrakes() {
        this.speed = this.speed - 5; // decrease the speed
        System.out.println("MotorCycle Speed: " + this.speed);
    }
}

// GOOD: Following LSP
// Subclass of Bike - implements all Bike class behavior
// As Bicycles do not have engines, we need not implement Engine interface
public class Bicycle extends Bike {

    String brand;
    Boolean hasGears;
    int speed;

    public Bicycle(String brand, Boolean hasGears, int speed) {
        this.brand = brand;
        this.hasGears = hasGears;
        this.speed = speed;
    }

    @Override
    public void accelerate() {
        this.speed = this.speed + 10; // increase the speed
        System.out.println("Bicycle Speed: " + this.speed);
    }

    @Override
    public void applyBrakes() {
        this.speed = this.speed - 5; // decrease the speed
        System.out.println("Bicycle Speed: " + this.speed);
    }
}

// Usage of LSP-compliant design
public class Demo {
    public static void main(String[] args) {
        // create the objects
        MotorCycle motorCycle = new MotorCycle("HeroHonda", 10);
        Bicycle bicycle = new Bicycle("Hercules", true, 10);

        // use the objects
        // Works fine with MotorCycle - implements all Bike class behavior
        motorCycle.turnOnEngine();
        motorCycle.accelerate();
        motorCycle.applyBrakes();
        motorCycle.turnOffEngine();
        // Works fine with Bicycle - implements all Bike class behavior
        bicycle.accelerate();
        bicycle.applyBrakes();
    }
}
```

Key Benefits of the Refactored Code

- **Prevents Fragile Code**: LSP helps avoid situations where subclass behavior violates the expectations set by the parent class.
- **Improves Flexibility**: By designing classes that can be substituted without breaking the parent contract.
- **Improved maintainability**: Changes to subclasses don't break existing code.
- **Promotes Code Reusability**: LSP facilitates the reuse of base classes and the creation of new subclasses without requiring modifications to existing code.

Code Example 2:

LSP Violation

```java
public class Vehicle {
    public Integer getNumberOfWheels() {
        return 4;
    }

    public Boolean hasEngine() {
        return true;
    }
}

public class MotorCycle extends Vehicle {

    public String getSpecifications() {
        return "MotorCycle has " + this.getNumberOfWheels() + " wheels and has engine: " + this.hasEngine();
    }
}

public class Car extends Vehicle {
    @Override
    public Integer getNumberOfWheels() {
        return 4;
    }

    public String getSpecifications() {
        return "Car has " + this.getNumberOfWheels() + " wheels and has engine: " + this.hasEngine();
    }
}

public class Bicycle extends Vehicle {
    public Boolean hasEngine() {
        return null;
    }
}

// Usage example - Violation of Liskov Substitution
public class ViolationDemo {
    public static void main(String[] args) {
        // Happy Flow
        List<Vehicle> vehicleList = new ArrayList<>();
        vehicleList.add(new MotorCycle());
        vehicleList.add(new Car());
        for (Vehicle vehicle : vehicleList) {
            System.out.println(vehicle.hasEngine().toString());
        }
        // Add Bicycle - Violation of LSP
        List<Vehicle> vehicleList2 = new ArrayList<>();
        vehicleList2.add(new MotorCycle());
        vehicleList2.add(new Car());
        vehicleList2.add(new Bicycle());
        for (Vehicle vehicle : vehicleList2) {
            System.out.println(vehicle.hasEngine().toString()); // throws NPE
            // Client code will break for Bicycle
        }
    }
}
```

LSP Solution

```java
public class Vehicle {
    // Include generic methods
    public Integer getNumberOfWheels() {
        return 2;
    }
```

```java
}
public class Bicycle extends Vehicle {
}
public class EngineVehicle extends Vehicle {
    public Boolean hasEngine() {
        return true;
    }
}
public class MotorCycle extends EngineVehicle {

}
public class Car extends EngineVehicle {
    @Override
    public Integer getNumberOfWheels() {
        return 4;
    }
}

public class SolutionDemo {
    public static void main(String[] args) {
        List<Vehicle> vehicleList = new ArrayList<>();
        vehicleList.add(new MotorCycle());
        vehicleList.add(new Car());
        vehicleList.add(new Bicycle());
        for (Vehicle vehicle: vehicleList) {
            System.out.println(vehicle.getNumberOfWheels().toString());
        }
        List<EngineVehicle> vehicleList2 = new ArrayList<>();
        vehicleList2.add(new MotorCycle());
        vehicleList2.add(new Car());
        // vehicleList2.add(new Bicycle()); // compile time error - cannot add Bicycle
        for (Vehicle vehicle: vehicleList2) {
            // System.out.println(vehicle.hasEngine()); // compile time error - cannot access hasEngine
        }
    }
}
```

Summary

LSP is vital because it upholds the integrity of your class hierarchy, ensuring that extending functionality or creating new subclasses won't disrupt the program's existing behavior. It keeps your code clean, modular, and safe to evolve.