

3. L - Liskov Substitution Principle (LSP)

What is the Liskov Substitution Principle (LSP)?

Code Example 1: Violating LSP

Problems with the Above Code

Code Example 2: Follows LSP

Key Benefits of the Refactored Code

Code Example 2:

LSP Violation

LSP Solution

Summary

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

```
1
2 // BAD: This design violates LSP
3 public interface Bike {
4
5     void turnOnEngine();
6
7     void turnOffEngine();
8
9     void accelerate();
10
11    void applyBrakes();
12 }
13
14 // Subclass of Bike - implements all Bike class behavior
15 public class Motorcycle implements Bike {
16     String company;
17     boolean isEngineOn;
18     int speed;
19
20     public Motorcycle(String company, int speed) {
21         this.company = company;
22         this.speed = speed;
23     }
24
25     @Override
26     public void turnOnEngine() {
27         this.isEngineOn = true; // turn on the engine!
```

```

28         System.out.println("Engine is ON!");
29     }
30
31     @Override
32     public void turnOffEngine() {
33         this.isEngineOn = false; // turn off the engine!
34         System.out.println("Engine is OFF!");
35     }
36
37     @Override
38     public void accelerate() {
39         this.speed = this.speed + 10; // increase the speed
40         System.out.println("MotorCycle Speed: " + this.speed);
41     }
42
43     @Override
44     public void applyBrakes() {
45         this.speed = this.speed - 5; // decrease the speed
46         System.out.println("MotorCycle Speed: " + this.speed);
47     }
48 }
49
50 // This class violates LSP!
51 public class Bicycle implements Bike {
52     String brand;
53     Boolean hasGears;
54     int speed;
55
56     public Bicycle(String brand, Boolean hasGears, int speed) {
57         this.brand = brand;
58         this.hasGears = hasGears;
59         this.speed = speed;
60     }
61
62     // LSP Violation: Strengthening preconditions
63     // Bicycle changes the behavior of turnOnEngine
64     @Override
65     public void turnOnEngine() {
66         throw new AssertionError("Detail Message: Bicycle has no
engine!");
67     }
68     // Bicycle changes the behavior of turnOffEngine
69     @Override
70     public void turnOffEngine() {
71         throw new AssertionError("Detail Message: Bicycle has no
engine!");
72     }
73
74     @Override
75     public void accelerate() {
76         this.speed = this.speed + 10; // increase the speed
77         System.out.println("Bicycle Speed: " + this.speed);
78     }
79
80     @Override
81     public void applyBrakes() {
82         this.speed = this.speed - 5; // decrease the speed
83         System.out.println("Bicycle Speed: " + this.speed);
84     }
85 }
86
87
88 // Usage example - demonstrates the LSP violations
89 public class Demo {
90     public static void main(String[] args) {
91         // create the objects
92         Motorcycle motorCycle = new Motorcycle("HeroHonda", 10);
93         Bicycle bicycle = new Bicycle("Hercules", true, 10);
94
95         // use the objects

```

```

96      // Works fine with Motorcycle - implements all Bike class
    behavior
97      motorcycle.turnOnEngine();
98      motorcycle.accelerate();
99      motorcycle.applyBrakes();
100     motorcycle.turnOffEngine();
101     // Client expects to be able to see the same behavior with
    Bicycle
102     bicycle.turnOnEngine(); // fails to implement Bike class
    behavior
103     bicycle.accelerate();
104     bicycle.applyBrakes();
105     bicycle.turnOffEngine(); // fails to implement Bike class
    behavior
106     }
107 }

```

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:

```

1  abstract class Bike {
2
3      // All Bikes can do these things
4      public abstract void accelerate();
5
6      public abstract void applyBrakes();
7
8  }
9  public interface Engine {
10     void turnOnEngine();
11
12     void turnOffEngine();
13 }
14
15 // Subclass of Bike - implements all Bike class behavior
16 public class Motorcycle extends Bike implements Engine {
17     String company;
18     boolean isEngineOn;
19     int speed;
20
21     public Motorcycle(String company, int speed) {
22         this.company = company;
23         this.speed = speed;
24     }
25
26     @Override
27     public void turnOnEngine() {
28         this.isEngineOn = true; // turn on the engine!
29         System.out.println("Engine is ON!");
30     }
31
32     @Override
33     public void turnOffEngine() {
34         this.isEngineOn = false; // turn off the engine!
35         System.out.println("Engine is OFF!");
36     }
37
38     public void accelerate() {
39         this.speed = this.speed + 10; // increase the speed
40         System.out.println("MotorCycle Speed: " + this.speed);
41     }

```

```

42
43     public void applyBrakes() {
44         this.speed = this.speed - 5; // decrease the speed
45         System.out.println("MotorCycle Speed: " + this.speed);
46     }
47 }
48
49 // GOOD: Following LSP
50 // Subclass of Bike - implements all Bike class behavior
51 // As Bicycles do not have engines, we need not implement Engine
   interface
52 public class Bicycle extends Bike {
53
54     String brand;
55     Boolean hasGears;
56     int speed;
57
58     public Bicycle(String brand, Boolean hasGears, int speed) {
59         this.brand = brand;
60         this.hasGears = hasGears;
61         this.speed = speed;
62     }
63
64     @Override
65     public void accelerate() {
66         this.speed = this.speed + 10; // increase the speed
67         System.out.println("Bicycle Speed: " + this.speed);
68     }
69
70     @Override
71     public void applyBrakes() {
72         this.speed = this.speed - 5; // decrease the speed
73         System.out.println("Bicycle Speed: " + this.speed);
74     }
75 }
76
77 // Usage of LSP-compliant design
78 public class Demo {
79     public static void main(String[] args) {
80         // create the objects
81         Motorcycle motorcycle = new Motorcycle("HeroHonda", 10);
82         Bicycle bicycle = new Bicycle("Hercules", true, 10);
83
84         // use the objects
85         // Works fine with Motorcycle - implements all Bike class
   behavior
86         motorcycle.turnOnEngine();
87         motorcycle.accelerate();
88         motorcycle.applyBrakes();
89         motorcycle.turnOffEngine();
90         // Works fine with Bicycle - implements all Bike class
   behavior
91         bicycle.accelerate();
92         bicycle.applyBrakes();
93     }
94 }

```

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

```
1 public class Vehicle {
2     public Integer getNumberOfWheels() {
3         return 4;
4     }
5
6     public Boolean hasEngine() {
7         return true;
8     }
9 }
10
11 public class Motorcycle extends Vehicle {
12
13     public String getSpecifications() {
14         return "MotorCycle has " + this.getNumberOfWheels() + " wheels
and has engine: " + this.hasEngine();
15     }
16 }
17
18 public class Car extends Vehicle {
19     @Override
20     public Integer getNumberOfWheels() {
21         return 4;
22     }
23
24     public String getSpecifications() {
25         return "Car has " + this.getNumberOfWheels() + " wheels and
has engine: " + this.hasEngine();
26     }
27 }
28
29 public class Bicycle extends Vehicle {
30     public Boolean hasEngine() {
31         return null;
32     }
33 }
34
35 // Usage example - Violation of Liskov Substitution
36 public class ViolationDemo {
37     public static void main(String[] args) {
38         // Happy Flow
39         List<Vehicle> vehicleList = new ArrayList<>();
40         vehicleList.add(new Motorcycle());
41         vehicleList.add(new Car());
42         for (Vehicle vehicle : vehicleList) {
43             System.out.println(vehicle.hasEngine().toString());
44         }
45         // Add Bicycle - Violation of LSP
46         List<Vehicle> vehicleList2 = new ArrayList<>();
47         vehicleList2.add(new Motorcycle());
48         vehicleList2.add(new Car());
49         vehicleList2.add(new Bicycle());
50         for (Vehicle vehicle : vehicleList2) {
51             System.out.println(vehicle.hasEngine().toString()); //
throws NPE
52             // Client code will break for Bicycle
53         }
54     }
55 }
```

LSP Solution

```
1 public class Vehicle {
2     // Include generic methods
3     public Integer getNumberOfWheels() {
4         return 2;
5     }
6 }
```

```

6  }
7  public class Bicycle extends Vehicle {
8  }
9  public class EngineVehicle extends Vehicle {
10     public Boolean hasEngine() {
11         return true;
12     }
13 }
14 public class Motorcycle extends EngineVehicle {
15 }
16 }
17 public class Car extends EngineVehicle {
18     @Override
19     public Integer getNumberOfWheels() {
20         return 4;
21     }
22 }
23
24 public class SolutionDemo {
25     public static void main(String[] args) {
26         List<Vehicle> vehicleList = new ArrayList<>();
27         vehicleList.add(new Motorcycle());
28         vehicleList.add(new Car());
29         vehicleList.add(new Bicycle());
30         for (Vehicle vehicle: vehicleList) {
31             System.out.println(vehicle.getNumberOfWheels().toString());
32         }
33         List<EngineVehicle> vehicleList2 = new ArrayList<>();
34         vehicleList2.add(new Motorcycle());
35         vehicleList2.add(new Car());
36         // vehicleList2.add(new Bicycle()); // compile time error -
cannot add Bicycle
37         for (Vehicle vehicle: vehicleList2) {
38             // System.out.println(vehicle.hasEngine()); // compile
time error - cannot access hasEngine
39         }
40     }
41 }

```

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.