🔄 Liskov Substitution Principle (LSP) - Complete Guide

Table of Contents

📖 Introduction to LSP {#introduction}

The **Liskov Substitution Principle (LSP)** states that objects of a superclass should be replaceable with objects of its subclasses without breaking the application. In simpler terms:

> **"If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program."**

Why LSP Matters

- **Polymorphism guarantee** - Ensures that inheritance hierarchies work correctly
- **Code reliability** - Prevents unexpected behavior when using subclasses
- **Design consistency** - Maintains behavioral contracts across inheritance chains
- **Testability** - Allows confident substitution in tests and production

The 7 Core Rules of LSP

1. **Method Argument Rule** - Subclass methods can accept more general parameters
2. **Return Type Rule** - Subclass methods can return more specific types
3. **Exception Rule** - Subclass methods can throw fewer or more specific exceptions
4. **Class Invariant Rule** - Subclass must maintain parent class invariants
5. **History Constraint Rule** - Subclass cannot modify immutable properties
6. **Pre-condition Rule** - Subclass cannot strengthen pre-conditions

7. **Post-condition Rule** - Subclass cannot weaken post-conditions

🎯 Core LSP Rules {#core-rules}

Understanding these rules is crucial for creating proper inheritance hierarchies that don't violate LSP.

Quick Reference Table

Rule	What It Means	Subclass Can	Subclass Cannot
Arguments	Parameter types	Accept more general types	Require more specific types
Return Types	Return value types	Return more specific types	Return more general types
Exceptions	Exception handling	Throw fewer/more specific exceptions	Throw more/different exceptions
Invariants	Class state rules	Maintain all parent invariants	Break parent invariants
History	Immutable properties	Keep history intact	Modify immutable state
Pre-conditions	Input requirements	Weaken requirements	Strengthen requirements
Post-conditions	Output guarantees	Strengthen guarantees	Weaken guarantees

✅ Rule 1: Method Argument Rule {#argument-rule}

Rule: Subclass methods can accept **more general** parameter types than the parent class method.

Correct Implementation

java

```java
// Base media format hierarchy
abstract class MediaFormat {
    protected String extension;
    public String getExtension() { return extension; }
}

class AudioFormat extends MediaFormat {
    public AudioFormat(String ext) { this.extension = ext; }
}

class MP3Format extends AudioFormat {
    public MP3Format() { super("mp3"); }
}

// Parent class with specific parameter
class MediaPlayerCorrect {
    public void play(MP3Format mp3) {
        System.out.println("Playing MP3: " + mp3.getExtension());
    }
}

// Subclass accepts MORE GENERAL parameter - ✅ LSP Compliant
class UniversalPlayerCorrect extends MediaPlayerCorrect {
    @Override
    public void play(AudioFormat audio) { // More general than MP3Format
        System.out.println("Playing any audio: " + audio.getExtension());
    }
}
```

Why This Works

- **Parent method:** Expects `MP3Format` (specific)
- **Subclass method:** Accepts `AudioFormat` (more general)
- **Result:** Any code calling with `MP3Format` will work with both implementations
- **LSP satisfied:** Subclass can handle everything parent can, plus more

Usage Example

java

```
public class LSPArgumentDemo {
    public static void testPlayer(MediaPlayerCorrect player) {
        MP3Format mp3 = new MP3Format();
        player.play(mp3); // Works with both parent and subclass
    }

    public static void main(String[] args) {
        testPlayer(new MediaPlayerCorrect());     // ✅ Works
        testPlayer(new UniversalPlayerCorrect()); // ✅ Works - LSP satisfied
    }
}
```

✅ Rule 2: Return Type Rule {#return-type-rule}

Rule: Subclass methods can return **more specific** types than the parent class method.

Correct Implementation

```
java
```

```
// Base document class
class Document {
    protected String title;
    public Document(String title) { this.title = title; }
    public String getTitle() { return title; }
}

// Specific document type
class PDFDocument extends Document {
    public PDFDocument(String title) { super(title); }
}

// Parent class returns general type
class DocumentCreator {
    public Document createDocument(String title) {
        return new Document(title);
    }
}

// Subclass returns MORE SPECIFIC type - ✅ LSP Compliant
class PDFCreator extends DocumentCreator {
    @Override
    public PDFDocument createDocument(String title) { // More specific than Document
        return new PDFDocument(title);
    }
}
```

Why This Works

- **Parent method:** Returns `Document` (general)
- **Subclass method:** Returns `PDFDocument` (more specific)
- **Result:** Client code expecting `Document` gets `PDFDocument` which IS-A `Document`
- **LSP satisfied:** More specific return type is always safe

Usage Example

```
java
```

```
public class LSPReturnTypeDemo {
    public static void processCreator(DocumentCreator creator) {
        Document doc = creator.createDocument("Test Doc");
        System.out.println("Created: " + doc.getTitle());
        // Works regardless of actual return type
    }

    public static void main(String[] args) {
        processCreator(new DocumentCreator()); // ✅ Returns Document
        processCreator(new PDFCreator());      // ✅ Returns PDFDocument - LSP satisfied
    }
}
```

✅ Rule 3: Exception Rule {#exception-rule}

Rule: Subclass methods can throw **fewer exceptions** or **more specific exceptions** than the parent class.

Correct Implementation

```
java

// Exception hierarchy
class PaymentException extends Exception {
    public PaymentException(String message) { super(message); }
}

class ValidationException extends PaymentException {
    public ValidationException(String message) { super(message); }
}

// Parent class throws general exception
class PaymentProcessorCorrect {
    public void processPayment(double amount) throws PaymentException {
        System.out.println("Processing payment: $" + amount);
    }
}

// Subclass throws MORE SPECIFIC exception - ✅ LSP Compliant
class CreditCardProcessorCorrect extends PaymentProcessorCorrect {
    @Override
    public void processPayment(double amount) throws ValidationException { // More specific
        if (amount <= 0) throw new ValidationException("Invalid amount");
        System.out.println("Credit card payment processed: $" + amount);
    }
}
```

Why This Works

- **Parent method:** Throws `PaymentException` (general)
- **Subclass method:** Throws `ValidationException` (more specific)
- **Result:** Client code catching `PaymentException` will also catch `ValidationException`
- **LSP satisfied:** More specific exceptions are always catchable by parent exception handlers

Usage Example

java

```java
public class LSPExceptionDemo {
    public static void handlePayment(PaymentProcessorCorrect processor, double amount) {
        try {
            processor.processPayment(amount);
        } catch (PaymentException e) {
            System.out.println("Payment failed: " + e.getMessage());
            // Catches both PaymentException and ValidationException
        }
    }

    public static void main(String[] args) {
        handlePayment(new PaymentProcessorCorrect(), 100.0);       // ✅ Works
        handlePayment(new CreditCardProcessorCorrect(), -10.0);    // ✅ Works - LSP satisfied
    }
}
```

✅ Rule 4: Class Invariant Rule {#invariant-rule}

Rule: Subclass must maintain all **invariants** (unchanging conditions) of the parent class.

Correct Implementation

java

```
import java.math.BigDecimal;

// Parent class with invariant: balance cannot go below zero without permission
class BankAccountCorrect {
    protected BigDecimal balance;

    public BankAccountCorrect(BigDecimal initialBalance) {
        this.balance = initialBalance;
    }

    public boolean withdraw(BigDecimal amount) {
        if (canWithdraw(amount)) {
            balance = balance.subtract(amount);
            return true;
        }
        return false; // Invariant: maintain withdrawal rules
    }

    protected boolean canWithdraw(BigDecimal amount) {
        return balance.compareTo(amount) >= 0; // Basic rule: sufficient balance
    }

    public void deposit(BigDecimal amount) {
        balance = balance.add(amount);
    }

    public BigDecimal getBalance() { return balance; }
}

// Subclass maintains invariant while extending behavior - ✅ LSP Compliant
class OverdraftAccountCorrect extends BankAccountCorrect {
    private BigDecimal overdraftLimit;

    public OverdraftAccountCorrect(BigDecimal initialBalance, BigDecimal overdraftLimit) {
        super(initialBalance);
        this.overdraftLimit = overdraftLimit;
    }

    @Override
    protected boolean canWithdraw(BigDecimal amount) {
        // Extends the rule but maintains the invariant structure
        return balance.add(overdraftLimit).compareTo(amount) >= 0;
    }
}
```

Why This Works

- **Parent invariant:** Withdrawal must check `canWithdraw()` before proceeding
- **Subclass behavior:** Still uses `canWithdraw()` but with extended logic
- **Result:** The withdrawal process remains consistent and predictable
- **LSP satisfied:** Core behavioral contract is maintained

Usage Example

```java
public class LSPInvariantDemo {
    public static void testAccount(BankAccountCorrect account) {
        System.out.println("Initial balance: " + account.getBalance());

        boolean success = account.withdraw(new BigDecimal("50"));
        System.out.println("Withdrawal success: " + success);
        System.out.println("Final balance: " + account.getBalance());
    }

    public static void main(String[] args) {
        BankAccountCorrect regular = new BankAccountCorrect(new BigDecimal("100"));
        BankAccountCorrect overdraft = new OverdraftAccountCorrect(
            new BigDecimal("100"), new BigDecimal("200"));

        testAccount(regular);   // ✅ Works with basic rules
        testAccount(overdraft); // ✅ Works with extended rules - LSP satisfied
    }
}
```

✅ Rule 5: History Constraint Rule {#history-rule}

Rule: Subclass cannot modify properties that were **immutable** in the parent class.

Correct Implementation

```java
```

```
// Parent class with history constraint on filename
class FileCorrect {
    protected String filename;  // Should remain constant after creation
    protected long size;
    protected boolean readOnly;

    public FileCorrect(String filename) {
        this.filename = filename; // Set once during construction
        this.size = 0;
        this.readOnly = false;
    }

    public void write(String data) {
        if (!readOnly) {
            size += calculateSizeIncrease(data);
            // filename never changes - history preserved
        }
    }

    protected long calculateSizeIncrease(String data) {
        return data.length();
    }

    public String getFilename() { return filename; } // Read-only access
    public long getSize() { return size; }
}

// Subclass preserves filename history - ✅ LSP Compliant
class CompressedFileCorrect extends FileCorrect {
    private double compressionRatio;

    public CompressedFileCorrect(String filename, double compressionRatio) {
        super(filename); // Filename set once, never changed
        this.compressionRatio = compressionRatio;
    }

    @Override
    protected long calculateSizeIncrease(String data) {
        return (long) (data.length() * compressionRatio);
        // filename remains unchanged - history constraint respected
    }
}
```

Why This Works

- **Parent constraint:** Filename is set once and never changes

- **Subclass behavior:** Also never changes filename, only modifies size calculation
- **Result:** Historical immutability is preserved across the inheritance hierarchy
- **LSP satisfied:** No historical properties are violated

Usage Example

java

```java
public class LSPHistoryDemo {
    public static void testFile(FileCorrect file) {
        String originalName = file.getFilename();
        System.out.println("Original filename: " + originalName);

        file.write("Some data");

        String currentName = file.getFilename();
        System.out.println("Current filename: " + currentName);
        System.out.println("Filename unchanged: " + originalName.equals(currentName));
    }

    public static void main(String[] args) {
        FileCorrect regular = new FileCorrect("document.txt");
        FileCorrect compressed = new CompressedFileCorrect("document.zip", 0.7);

        testFile(regular);    // ✅ Filename preserved
        testFile(compressed); // ✅ Filename preserved - LSP satisfied
    }
}
```

✅ Rule 6: Pre-condition Rule {#precondition-rule}

Rule: Subclass methods cannot **strengthen** (make more restrictive) the pre-conditions of parent methods.

Correct Implementation

java

```
// Parent class with strict pre-conditions
class AuthenticatorCorrect {
    public boolean authenticate(String username, String password) {
        // Strict pre-conditions
        if (username == null || username.length() < 5 ||
            password == null || password.length() < 8) {
            throw new IllegalArgumentException("Invalid credentials");
        }
        return true;
    }
}

// Subclass WEAKENS pre-conditions - ✅ LSP Compliant
class FlexibleAuthenticatorCorrect extends AuthenticatorCorrect {
    @Override
    public boolean authenticate(String username, String password) {
        // Weakens pre-conditions by providing defaults and fixes
        if (username == null) username = "guestuser";
        if (password == null) password = "default123";

        if (username.length() < 5) username += "user";
        if (password.length() < 8) password += "123";

        // Now calls parent with valid parameters
        return super.authenticate(username, password);
    }
}
```

Why This Works

- **Parent pre-condition:** Requires non-null username ≥5 chars, password ≥8 chars
- **Subclass behavior:** Accepts more inputs by providing defaults and fixes
- **Result:** Subclass is more permissive, handles cases parent would reject
- **LSP satisfied:** Client code works with broader range of inputs

Usage Example

```
java
```

```
public class LSPPreconditionDemo {
    public static void testAuth(AuthenticatorCorrect auth, String user, String pass) {
        try {
            boolean result = auth.authenticate(user, pass);
            System.out.println("Authentication successful: " + result);
        } catch (Exception e) {
            System.out.println("Authentication failed: " + e.getMessage());
        }
    }

    public static void main(String[] args) {
        AuthenticatorCorrect strict = new AuthenticatorCorrect();
        AuthenticatorCorrect flexible = new FlexibleAuthenticatorCorrect();

        // Test with invalid input
        testAuth(strict, "ab", "123");     // ✅ Throws exception
        testAuth(flexible, "ab", "123");   // ✅ Succeeds with fixes - LSP satisfied
    }
}
```

✅ Rule 7: Post-condition Rule {#postcondition-rule}

Rule: Subclass methods cannot **weaken** (make less restrictive) the post-conditions of parent methods.

Correct Implementation

```
java
```

```java
import java.util.*;

// Parent class with post-condition: result must be sorted
class DataSorter {
    public List<Integer> sort(List<Integer> data) {
        List<Integer> result = new ArrayList<>(data);
        Collections.sort(result);
        // Post-condition: result is sorted in ascending order
        return result;
    }
}

// Subclass STRENGTHENS post-condition - ✅ LSP Compliant
class OptimizedSorterCorrect extends DataSorter {
    @Override
    public List<Integer> sort(List<Integer> data) {
        List<Integer> result = new ArrayList<>(data);
        result.sort(Integer::compareTo); // Different algorithm, same guarantee

        // Strengthens post-condition with explicit verification
        assert isSorted(result) : "Result must be sorted";
        return result;
    }

    private boolean isSorted(List<Integer> list) {
        for (int i = 1; i < list.size(); i++) {
            if (list.get(i) < list.get(i - 1)) return false;
        }
        return true;
    }
}
```

Why This Works

- **Parent post-condition:** Returns sorted list
- **Subclass behavior:** Also returns sorted list + adds verification
- **Result:** Subclass provides same guarantee with additional assurance
- **LSP satisfied:** Post-condition is maintained or strengthened

Usage Example

```
java
```

```
public class LSPPostconditionDemo {
    public static void testSorter(DataSorter sorter) {
        List<Integer> unsorted = Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6);
        System.out.println("Input: " + unsorted);

        List<Integer> sorted = sorter.sort(unsorted);
        System.out.println("Output: " + sorted);

        // Verify post-condition
        boolean isSorted = true;
        for (int i = 1; i < sorted.size(); i++) {
            if (sorted.get(i) < sorted.get(i - 1)) {
                isSorted = false;
                break;
            }
        }
        System.out.println("Is sorted: " + isSorted);
    }

    public static void main(String[] args) {
        DataSorter basic = new DataSorter();
        DataSorter optimized = new OptimizedSorterCorrect();

        testSorter(basic);     // ✅ Returns sorted list
        testSorter(optimized); // ✅ Returns sorted list with verification - LSP satisfied
    }
}
```

📚 Summary and Best Practices {#summary}

LSP Compliance Checklist

✅ Method Signatures

- **Parameters:** Subclass can accept more general types
- **Return types:** Subclass can return more specific types
- **Exceptions:** Subclass can throw fewer or more specific exceptions

✅ Behavioral Contracts

- **Pre-conditions:** Subclass can weaken (be more permissive)
- **Post-conditions:** Subclass can strengthen (provide more guarantees)
- **Invariants:** Subclass must maintain all parent class invariants
- **History:** Subclass cannot modify immutable parent properties

Common LSP Violations to Avoid

❌ Strengthening Pre-conditions

```java
// WRONG - More restrictive than parent
class StrictProcessor extends BaseProcessor {
    @Override
    public void process(String data) {
        if (data.length() < 100) { // Parent allows shorter strings
            throw new IllegalArgumentException("String too short");
        }
        super.process(data);
    }
}
```

❌ Weakening Post-conditions

```java
// WRONG - Weaker guarantee than parent
class WeakSorter extends DataSorter {
    @Override
    public List<Integer> sort(List<Integer> data) {
        return data; // Returns unsorted data - breaks parent contract
    }
}
```

❌ Breaking Invariants

```java
// WRONG - Violates parent class invariant
class BadAccount extends BankAccount {
    @Override
    public boolean withdraw(BigDecimal amount) {
        balance = balance.subtract(amount); // No balance check - breaks invariant
        return true;
    }
}
```

Best Practices for LSP Compliance

1. Design by Contract

- Clearly define pre-conditions, post-conditions, and invariants
- Document behavioral contracts in interfaces and abstract classes
- Use assertions to verify contracts during development

2. Favor Composition Over Inheritance

- Use inheritance only when true IS-A relationships exist
- Consider composition when behavior differs significantly
- Abstract common behavior into interfaces

3. Test Substitutability

- Write tests that work with parent class references
- Verify that subclasses can replace parents in all scenarios
- Use polymorphic test methods to ensure LSP compliance

4. Interface Segregation

- Keep interfaces focused and cohesive
- Avoid large interfaces that force unnecessary dependencies
- Design for specific client needs

Real-World Benefits of LSP Compliance

Code Reliability

- Polymorphic code works predictably with any subclass
- Fewer runtime surprises and unexpected behaviors
- Easier debugging and maintenance

Design Flexibility

- New subclasses can be added without breaking existing code
- Framework and library code remains stable
- Better separation of concerns

Testing Advantages

- Unit tests can use any implementation interchangeably
- Mock objects and test doubles work seamlessly
- Integration testing is more straightforward

Final Recommendations

1. **Start with clear contracts** - Define what your base classes guarantee
2. **Test early and often** - Verify substitutability during development
3. **Document behavioral expectations** - Make contracts explicit for future developers
4. **Review inheritance hierarchies** - Regularly check for LSP violations
5. **Prefer small, focused interfaces** - Easier to maintain behavioral consistency

Remember: LSP is not just about method signatures - it's about behavioral compatibility. A subclass should be a perfect behavioral substitute for its parent class in all contexts.