Liskov Substitution Principle (LSP)

Have you ever passed a subclass into a method expecting the parent class… and watched your program crash or behave in unexpected ways?

Or extended a class… only to find yourself overriding methods just to throw exceptions?

If yes, you've probably run into a violation of one of the most misunderstood object-oriented design principles: **The Liskov Substitution Principle (LSP).**

Let's understand it with a real-world example and why it breaks LSP.

Imagine you're building a system to manage different types of documents.

You start with a simple base class:

Java
Python
C++
C#
Typescript

```
class Document:
    def __init__(self, data):
        self.data = data

    def open(self):
        print("Document opened. Data:", self.data[:20] + "...")

    def save(self, new_data):
        self.data = new_data
        print("Document saved.")

    def get_data(self):
        return self.data
```

Now, a new requirement comes in:

"We need a **read-only** document type—for sensitive content like government reports or signed contracts."

You think: *A ReadOnlyDocument is still a kind of Document*, so inheritance makes sense.

So, you extend the `Document` class:

Java
Python
C++
C#
Typescript

```
class ReadOnlyDocument(Document):
    def __init__(self, data):
        super().__init__(data)

    def save(self, new_data):
        raise UnsupportedOperationException("Cannot save a read-only
document!")
```

Seems reasonable, right?

But Then Reality Hits…

Let's see how this plays out in client code:

Java
Python
C++
C#
Typescript

```
class DocumentProcessor:
    def process_and_save(self, doc, additional_info):
        doc.open()
        current_data = doc.get_data()
        new_data = current_data + " | Processed: " + additional_info
        doc.save(new_data)  # Assumes all Documents are savable
        print("Document processing complete.")

if __name__ == "__main__":
    regular_doc = Document("Initial project proposal content.")
    confidential_report = ReadOnlyDocument("Top secret government
data.")

    processor = DocumentProcessor()

    print("--- Processing Regular Document ---")
    processor.process_and_save(regular_doc, "Reviewed by Alice")
```

```
    print("\n--- Processing ReadOnly Document ---")
    try:
        processor.process_and_save(confidential_report, "Reviewed by
Bob")
    except UnsupportedOperationException as e:
        print("Error:", str(e))
```

Output:

Shell

```
--- Processing Regular Document ---
Document opened. Data: Initial project prop...
Document saved.
Document processing complete.

--- Processing ReadOnly Document ---
Document opened. Data: Top secret governme...
Error: Cannot save a read-only document!
```

Boom!The client code expected *any* `Document` to be savable. But when it received a `ReadOnlyDocument`, that assumption exploded into a runtime exception.

What Went Wrong?

At the heart of this failure is a violation of a fundamental design principle: **Liskov Substitution Principle.**

Our subtype (`ReadOnlyDocument`) cannot be seamlessly substituted for its base type (`Document`) without altering the desired behavior of the program.

If you ever find yourself overriding a method just to throw an exception, or adding subtype-specific conditions in client code—it's a red flag and you might be violating **LSP**.

Introducing the Liskov Substitution Principle (LSP)

"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 that program (correctness, task performed, etc.)." — *Barbara Liskov, 1987*

In simpler terms: **If a class `S` extends or implements class `T`, then you should be able to use `S` anywhere `T` is expected—without breaking the program's behavior or logic.**

In other words, **subtypes must honor the expectations set by their base types**. The client code shouldn't need to know or care which specific subtype it's dealing with. Everything should "just work."

Why Does LSP Matter?

- **Reliability and Predictability:** When LSP is followed, your code behaves consistently. You can substitute any subtype and still get the behavior your client code expects. No unpleasant surprises.
- **Reduced Bugs:** LSP violations often lead to conditional logic (e.g., if (obj instanceof ReadOnlyDocument)) in client code to handle subtypes differently. This is a **code smell**. It's a sign that your design is leaking abstraction. When client code has to "know" the subtype to behave correctly, you've broken polymorphism.
- **Maintainability and Extensibility:** Well-behaved hierarchies are easier to understand, maintain, and extend. You can add new subtypes without fear of breaking existing code that relies on the base type's contract.
- **True Polymorphism:** LSP is what makes polymorphism truly powerful. You can write generic algorithms that operate on a base type, confident that they will work correctly with any current or future subtype.
- **Testability:** Tests written for the base class's interface should, in theory, pass for all its subtypes if LSP is respected (at least for the shared behaviors).

Essentially, LSP helps you build systems that are:

- Easier to extend
- Less prone to bugs
- And far more resilient to change

Implementing LSP

Let's refactor our design so that subtypes like `ReadOnlyDocument` can be used **without violating the expectations** set by the base type.

The root problem was that the base class `Document` **assumed all documents are editable**, but not all documents should be. To fix this, we need to:

- Separate **editable** behavior from **read-only** behavior
- Use **interfaces or abstract types** to model capabilities explicitly

Step 1: Define Behavior Interfaces

Instead of having one base class with assumptions about mutability, let's break responsibilities apart:

Java
Python
C++
C#
Typescript

```python
class Document(ABC):
    @abstractmethod
    def open(self):
        pass

    @abstractmethod
    def get_data(self):
        pass

class Editable(ABC):
    @abstractmethod
    def save(self, new_data):
        pass
```

- `Document`: represents the ability to open and view data
- `Editable`: represents the capability to modify data

This clearly defines what each object can do—and prevents clients from assuming editability unless explicitly promised.

Step 2: Implement `EditableDocument` and `ReadOnlyDocument`
Now we implement our two concrete types:

EditableDocument

Java
Python
C++
C#
Typescript

```python
class EditableDocument(Document, Editable):
    def __init__(self, data):
        self.data = data

    def open(self):
```

```
        print("Editable Document opened. Data:", self._preview())

    def save(self, new_data):
        self.data = new_data
        print("Document saved.")

    def get_data(self):
        return self.data

    def _preview(self):
        return self.data[:20] + "..."
```

ReadOnlyDocument

Java
Python
C++
C#
Typescript

```
class ReadOnlyDocument(Document):
    def __init__(self, data):
        self.data = data

    def open(self):
        print("Read-Only Document opened. Data:", self._preview())

    def get_data(self):
        return self.data

    def _preview(self):
        return self.data[:20] + "..."
```

Now:

- Both `EditableDocument` and `ReadOnlyDocument` are valid `Document` objects
- Only `EditableDocument` implements the `Editable` interface
- There's **no risk** of calling `save()` on a read-only document—it's simply **not possible**

Step 3: Refactor the Client Code

Let's update `DocumentProcessor` to act accordingly:

Java
Python
C++

C#
Typescript

```
class DocumentProcessor:
    def process(self, doc: Document):
        doc.open()
        print("Document processed.")

    def process_and_save(self, doc: Document, additional_info: str):
        if not isinstance(doc, Editable):
            raise ValueError("Document is not editable.")

        doc.open()
        current_data = doc.get_data()
        new_data = current_data + " | Processed: " + additional_info
        doc.save(new_data)
        print("Editable document processed and saved.")
```

Alternatively, use two different methods:

Java
Python
C++
C#
Typescript

```
 def process_editable_document(self, editable_doc: Editable, doc:
Document, additional_info: str):
      doc.open()
      current_data = doc.get_data()
      new_data = current_data + " | Processed: " + additional_info
      editable_doc.save(new_data)
      print("Editable document processed and saved.")
```

Usage Example:

Java
Python
C++
C#
Typescript

```
if __name__ == "__main__":
    editable = EditableDocument("Draft proposal for Q3.")
    read_only = ReadOnlyDocument("Top secret strategy.")

    processor = DocumentProcessor()

    print("--- Processing Editable Document ---")
    processor.process_and_save(editable, "Reviewed by Alice")
```

```
    print("\n--- Processing Read-Only Document ---")
    processor.process(read_only)  # This works fine
```

Common Pitfalls While Applying LSP

Understanding the Liskov Substitution Principle is one thing. Applying it correctly in the real world—**that's where the challenges begin.**

Here are some of the most common traps to watch out for:

1. The "Is-A" Linguistic Trap
Just because something *sounds* like it "is a" something else in natural language doesn't mean it's a valid subtype in code.

Take this classic example:

A **penguin** *is a* **bird**, but penguins can't fly.If your `Bird` class has a `fly()` method, and you override it in `Penguin` to throw an exception or do nothing—you've violated LSP.

The key insight: **subtyping must be based on behavior, not just taxonomy**.

2. Overriding Methods to Do Nothing or Throw Exceptions
If you find yourself writing code like this:

Java
```
@Override
public void save(String data) {
    throw new UnsupportedOperationException("Not supported.");
}
```
That's a flashing red warning light. If a subclass **cannot meaningfully implement** a method defined in the base class, it's likely **not a valid subtype**.

This leads to brittle code and runtime surprises—exactly what LSP aims to prevent.

3. Violating Preconditions or Postconditions
Changing the assumptions of a method is a subtle but dangerous LSP violation.

- **Precondition violation**: The subtype **requires more** than the base class contract promised.

- **Postcondition violation**: The subtype **delivers less** than the base class guaranteed.

These break the trust that clients place in the base class's behavior.

4. Type Checks in Client Code
Code like this is often a symptom of broken design:

Java
```
if (document instanceof ReadOnlyDocument) {
    // Special-case logic
}
```
Whenever client code has to **know the exact subtype** to behave correctly, you've violated the principle of substitution.

Polymorphism should make the client code unaware of specific subtypes. If you're relying on `instanceof`, it's time to revisit your abstraction.

5. Restricting or Relaxing Behavior Unexpectedly
Subclasses shouldn't arbitrarily **tighten or loosen** the behavior defined by the base class.

For example:

- Making a **mutable** property in the base class **immutable** in the subclass (or vice versa) can lead to subtle bugs.
- Changing validation logic in ways that break existing assumptions in client code is another LSP violation.

Consistency is key.

Common Questions About LSP

Q1: Isn't LSP just about "good inheritance"?
Yes, but it's more precise.LSP defines what *correct* behavioral inheritance actually looks like. It's not just about **reusing code**, it's about preserving **correctness and intention**.

Think of LSP as a **safety net for polymorphism**. It ensures your abstractions can scale and evolve cleanly.

Q2: What if my subclass really can't do what the base class does?

This is **exactly when you should stop and rethink your hierarchy**.

Some options:

- **Maybe it shouldn't be a subtype at all**: A `ReadOnlyDocument` that can't be saved probably shouldn't inherit from a `Document` class that supports saving.
- **Split responsibilities**: Use interfaces like `Readable`, `Editable`, etc., to model capabilities explicitly.
- **Favor composition over inheritance**: Instead of trying to "be" something, let your object **have** a capability.

Q3: Does this mean I can never use `instanceof` or casting?
Not never, but be cautious.

There are **legitimate, narrow use cases**: implementing `equals()`, serialization, certain framework hooks.

But if you're using `instanceof` to drive **business logic** or alter behavior, you're likely covering up an LSP violation.

Ask yourself:

"Am I using this because I broke polymorphism?"
If yes, revisit your design.