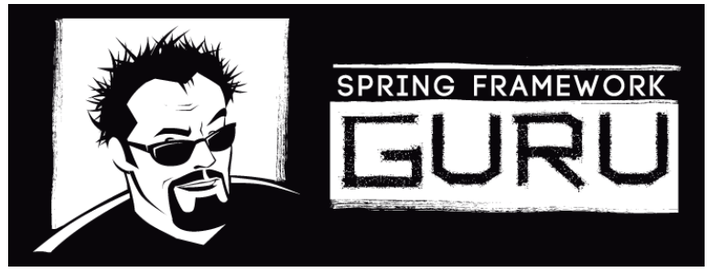


Liskov Substitution Principle

The Liskov Substitution Principle is one of the SOLID principles of object-oriented programming (Single responsibility, Open-closed, Liskov Substitution, Interface Segregation and Dependency Inversion). We have already written about the single responsibility principle, and these five principles combined are used to make object-oriented code more readable, maintainable and easier to upgrade and modify.

Liskov Substitution Principle states the following: *"in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may substitute objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)"*. Simply said, any object of some class in an object-oriented program can be replaced by an object of a child class. In order to understand this principle better, we'll make a small digression to briefly remind ourselves about the concept of inheritance and its properties, as well as subtyping, a form of polymorphism.

Inheritance, Polymorphism, Subtyping

Inheritance is a concept fairly simple to understand – it is when an object or a class are based on another object or class. When a class is "inherited" from another class, it means that the inherited class (also called subclass, or child class) contains all the characteristics of the superclass (parent class), but can also contain new properties. Let's illustrate this with a common example: if you have a class `Watch` , you can inherit from that class to get a class `PocketWatch` . A pocket watch is still a watch, it just has some additional features.

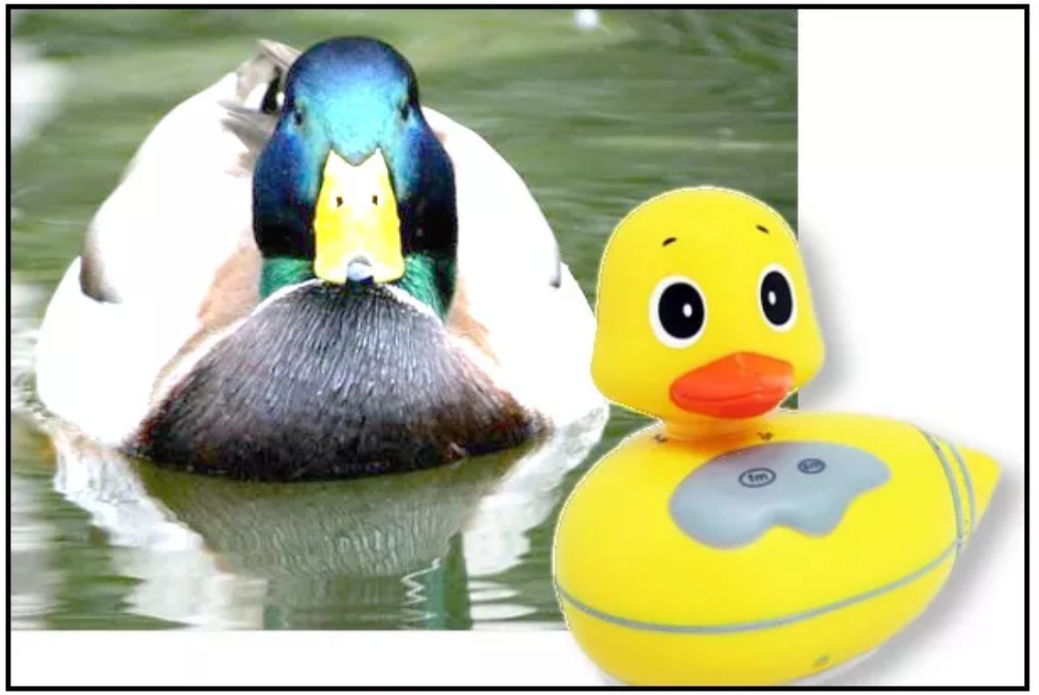
Another example would be a class called `Woman` with a child class called `Mother`. A mother is still a woman, with the addition of having a child. This brings us to the next term we should explain, which is called **polymorphism**: objects can behave in one way in a certain situation, and in another way in some other situation. In object-oriented programming, this is called *context-dependent behavior*. To use the last example: a mother, when taking a walk with her child or attending a school parent's meeting, will behave as a mother. But when she is out with her friends, at work or simply doing errands, she will behave as a woman. (As you can see, this difference is not that strict.)

Subtyping is a concept that is not identical to polymorphism. However, the two are so tightly connected and fused together in common languages like C++, Java and C#, that the difference between them is practically non-existent. We will still give a formal definition of subtyping, though, for the sake of completeness, but the details will not be discussed in this article. *"In programming language theory, subtyping (also subtype polymorphism or inclusion polymorphism) is a form of type polymorphism in which a subtype is a datatype that is related to another datatype (the supertype) by some notion of substitutability, meaning that program elements, typically subroutines or functions, written to operate on elements of the supertype can also operate on elements of the subtype. If S is a subtype of T, the subtyping relation is often written S <: T, to mean that any term of type S can be safely used in a context where a term of type T is expected."*

This brings us to the original theme of the article – the Liskov Substitution Principle.

Liskov Substitution Principle Examples

The Liskov substitution principle, written by Barbara Liskov in 1988, states that functions that reference base classes must be able to use objects of derived (child) classes without knowing it. It's important for a programmer to notice that, unlike some other Gang of Four principles, whose breaking might result in bad, but *working code*, the violation of this principle will most likely lead to buggy or difficult to maintain code.



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Let's illustrate this in Java:

```
1  class TrasportationDevice
2  {
3     String name;
4     String getName() { ... }
5     void setName(String n) { ... }
6
7     double speed;
8     double getSpeed() { ... }
9     void setSpeed(double d) { ... }
10
11    Engine engine;
12    Engine getEngine() { ... }
13    void setEngine(Engine e) { ... }
```

```
14
15      void startEngine() { ... }
16  }
```

```
1  class Car extends TransportationDevice
2  {
3      @Override
4      void startEngine() { ... }
5  }
```

There is no problem here, right? A car is definitely a transportation device, and here we can see that it overrides the `startEngine()` method of its superclass.

Let's add another transportation device:

```
1  class Bicycle extends TransportationDevice
2  {
3      @Override
4      void startEngine() /*problem!*/
5  }
```

Everything isn't going as planned now! Yes, a bicycle **is a** transportation device, however, it does not have an engine and hence, the method `startEngine()` cannot be implemented.

These are the kinds of problems that violation of Liskov Substitution Principle leads to, and they can most usually be recognized by a method that does nothing, or even can't be implemented.

The solution to these problems is a correct **inheritance hierarchy**, and in our case we would solve the problem by differentiating classes of transportation devices with and without engines. Even though a bicycle **is a** transportation device, it doesn't have an engine. In this example our definition of transportation device is wrong. It should not have an engine.

We can refactor our `TransportationDevice` class as follows:

```
1  class TrasportationDevice
2  {
3      String name;
```

```
    String getName() { ... }
    void setName(String n) { ... }

    double speed;
    double getSpeed() { ... }
    void setSpeed(double d) { ... }
}
```

Now we can extend `TransportationDevice` for non-motorized devices.

```
class DevicesWithoutEngines extends TransportationDevice
{
    void startMoving() { ... }
}
```

And extend `TransportationDevice` for motorized devices. Here is is more appropriate to add the `Engine` object.

```
class DevicesWithEngines extends TransportationDevice
{
    Engine engine;
    Engine getEngine() { ... }
    void setEngine(Engine e) { ... }

    void startEngine() { ... }
}
```

Thus our `Car` class becomes more specialized, while adhering to the Liskov Substitution Principle.

```
class Car extends DevicesWithEngines
{
    @Override
    void startEngine() { ... }
}
```

And our `Bicycle` class is also in compliance with the Liskov Substitution Principle.

```
class Bicycle extends DevicesWithoutEngines
{
    @Override
    void startMoving() { ... }
}
```

Conclusion

Object Oriented languages such as Java are very powerful and offer you as a developer a tremendous amount of flexibility. You can misuse or abuse any language. In the Polymorphism post I explained the 'Is-A' test. If you're writing objects which extend classes, but fails the **'Is-A'** test, you're likely violating the Liskov Substitution Principle.

---

4 comments on "Liskov Substitution Principle"

Stanislav — August 7, 2015 at 4:31 am # Reply

It might be better to name DevicesWithoutEngines and DevicesWithEngines in singular form, DeviceWithoutEngines and DeviceWithEngines.

Motaz Mohammad — November 10, 2016 at 3:50 am # Reply

My Dream is to be a GURU like you . Great articular thanks for your time .

Dougie T — November 11, 2017 at 9:47 am # Reply

Great explanation. thank you.

P.S. on the first code block consider correcting the spelling for "transportation" in the "trasportationDevice" class definition.