# Liskov's Substitution Principle

(https://github.com/learn-co-curriculum/phase-1-liskovs-substitution-principle) (https://github.com/learn-co-curriculum/phase-1-liskovs-substitution-principle/issues/new)

## Learning Goals

- Recognize the meaning of strong behavioral sub-typing
- Recognize the benefits of upholding Liskov's substitution principle

## Introduction

Much work has been done in the field of Object Oriented programming, and over the last few decades, engineers have developed design patterns and principles that are meant to help keep Object Oriented code easier to understand and maintain. One principle in particular applies to inheritance and extension: Liskov's substitution principle.

In this lesson, we're going to briefly look at what Liskov's substitution principle is, how to adhere to it and why.

## Recognize the Meaning of Strong Behavioral Sub-typing

Liskov's substitution principle, also known as strong behavioral sub-typing, is the 'L' in **SOLID** (https://en.wikipedia.org/wiki/SOLID) a popular set of Object Oriented design principles.

**Liskov's substitution principle:** Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

In terms to JavaScript inheritance, **an instance of a parent class should be replaceable with an instance of a child class.**

If we follow this principle, the consequence is that properties and methods that exist on the parent will never be modified in any child. Child classes can *expand* upon what they inherited, adding methods or extra properties, do not modify what they inherited.

If you need to modify a child's inherited properties or methods, why are we inheriting them in the first place?

Below are two examples, one that violates Liskov's principle, and one that upholds it:

## Violates Substitution Principle

```
class Reptile {
  constructor(name) {
    this.name = name;
  }
  get move() {
    return `${this.name} crawls away`;
  }
}

// Lizard inherits `move` because it crawls
class Lizard extends Reptile {}

//  Snake overrides `move` because it cannot crawl
class Snake extends Reptile {
  get move() {
    return `${this.name} slithers away`;
  }
}
```

The `Snake` class is a subtype of class `Reptile` , but overrides the `move` getter because the original doesn't apply. If we created an instance of `Reptile` :

```
let tricky = new Reptile("Tricky");
```

and an instance of `Snake` :

```
let hissy = new Snake("Hissy Elliot");
```

We see that `tricky` cannot be replaced with `hissy` without changing behavior:

```
tricky.move; // => "Tricky crawls away"
hissy.move; // => "Hissy Elliot slithers away"
```

## Upholds Substitution Principle

So how do we stop violating Liskov's principle? We can either choose to not inherit from the same parent, *or* we can create a grandparent `Reptile` class. This grandparent class can still contain all shared data and behavior for all parent and child classes. Since the definition of `move` is *not* shared by all, we can move these definitions down a level, defining them in two new parent classes:

```
// all reptiles have a name
class Reptile {
  constructor(name) {
    this.name = name;
  }
}

// legless reptiles slither
class LeglessReptile extends Reptile {
  move() {
    return `${this.name} slithers away`;
  }
}

// legged reptiles crawl
class LeggedReptile extends Reptile {
  move() {
    return `${this.name} crawls away`;
  }
}
```

```
class Lizard extends LeggedReptile {}
class Snake extends LeglessReptile {}
```

Now we've got two levels of inheritance. `Reptile` sets up the `constructor` that all children and grandchildren inherit. Because `move` needs to behave differently for legged and legless reptiles, it is defined differently in both `LeglessReptile` and `LeggedReptile` .

With this structure, if an instance of `Lizard` replaced an instance of `LeggedReptile` *or* `Reptile` , it will work correctly. The same goes for an instance of `Snake` .

> Why does this work? We've removed some of behavior of `Reptile` . An instance of `Reptile` will never be required to utilize a method it hasn't defined or inherited.

## Recognize the Benefits of Upholding Liskov's Substitution Principle

Both of the above examples *work*. There is no syntax error if you choose to ignore Liskov's substitution principle. This is considered a purely *semantic* design choice.

The benefit of following this principle is that no matter how complicated inheritance gets, you can always assume that whatever a parent class has, a child class will have too. Looking at a parent class should give us some insight into the functionality of any children, grandchildren, great grandchildren, etc...

If we have chains of inheritance where children fundamentally change the data and behaviors they inherit, we can potentially introduce bugs. More importantly, this can also make your code much more complicated than it needs to be, which will make it harder to change and understand.

As per LSP, a child class may include *more* properties or use the data *differently*. If you do find that a child class needs to overwrite a method or property it inherited from its parent, consider other options - perhaps don't inherit at all.

**Note:** In general, when dealing with inheritance, the fewer levels of inheritance, the better. If you've got great grandparent, grandparent, parent and child classes, it can be difficult to figure out which class contributes what to a child. It also makes our code more difficult to change. You may not be able to modify code on a grandparent class without fundamentally changing how a parent or child class functions. Too much inheritance can make our code inflexible.

Upholding Liskov's substitution principle limits what we can do with inheritance, discouraging larger chains of inherited classes. As a side effect, our code is easier to understand and change later on.

## Conclusion

Liskov's substitution principle ensures that all subtypes may replace their types without altering the behavior of the program. An instance of a parent class should be replaceable by an instance of any of its child classes and still work as expected. The result of this pattern is a clearer, consistent inheritance pattern that leads to more organized, easier to change code.