Liskov Substitution Principle

principles. It was created by Barbara Liskov and states:

Subtypes must be able to substitute base types.

Aside from the obvious implementation of this principle -- ensuring the subtype has the same behavior as the base type -- the implicit implementation is ensuring the subtype has the same *semantic* behavior as the base type.

Semantic is defined as:

Relating to meaning in language or logic.

And so, we create a definition of what the class name means. Typically, squares and rectangles are defined through their mathematic definition. Mathematically, a **square** is a shape with four sides in equal lengths. For rectangles, we will be using the definition of a **perfect rectangle**, meaning a shape where the top and bottom sides have equal lengths and left and right sides have equal lengths.

From this definition, we can conclude that a square's width and height must be the same while a rectangle can have a width different from its height as well as having a width equal to its height.

Let's create classes for these shapes, where we just define the width and height.

```
class Rectangle{
  public Integer width = 0;
  public Integer height = 0;


  Rectangle(Integer _width, Integer _height){
    width = _width;
    height = _height;
  }


  public Integer area(){
    return width * height;
  }
}
```

```
class Square extends Rectangle{
  Square(Integer side){
    super(side, side);
  }
}
```
view rawlsp_example_1.java hosted with ♡ by GitHub

We can see that a Square object can easily replace a Rectangle object in a given situation. All Square objects share the same behavior as Rectangle.

However, we can see the semantic difference, when we test #area().

```
@Test
public Boolean describe_area() {
  Rectangle rec = new Rectangle(5,5);
  rec.width = 6;


  Assert.assertTrue(rec.area() == 30);
}
```
view rawlsp_example_test.java hosted with ♡ by GitHub

Because a rectangle can have a different height than its width the result of #area() adheres to our semantic definition.

If we replaced Rectangle with Square the test will pass, but the Square object is no longer behaving the way we expect.

```
@Test
public Boolean describe_area() {
  Square sq = new Square(5,5);
  sq.width = 6;


  Assert.assertTrue(sq.area() == 30);
}
```
view rawlsp_example_test_2.java hosted with ♡ by GitHub

Because a square's width and height cannot be different, the Square object breaks our semantic definition. This breaks the Liskov substitute principle.

The easiest way to ensure both Rectangle and Square pass the test, while satisfying LSP, is to remove line 4.

You may be thinking, why not implement a #setWidth() function in Rectangle that can be overwritten in Square? This would cause Square to only share the interface of Rectangle instead of its behavior, making the abstraction unnecessary. By removing line 4, we can satisfy LSP and we can go even further by making width and height private.

```java
class Rectangle{
  private Integer width = 0;
  private Integer height = 0;


  Rectangle(Integer _width, Integer _height){
    width = _width;
    height = _height;
  }


  public Integer area(){
    return width * height;
  }
}


class Square extends Rectangle{
  Square(Integer side){
    super(side, side);
  }
}


/* ...assume the following is written to in a proper test class */
@Test
public Boolean describe_area() {
  Square sq = new Square(5,5);
  sq.width = 6;


  Assert.assertTrue(sq.area() == 30);
}
```

view rawlsp_example_refactor.java hosted with ♡ by GitHub

In Conclusion

When implementing inheritance, be sure to adhere to the meaning of the class outside of the implementation. This helps the class become more understandable in the grand scheme of the project.

Another technique that helps ensure classes adhere to LSP is design by contract.

URL: https://dev.to/naomidennis/a-light-introduction-liskov-substitution-principle-295a