Liskove Substitution Principle was written by Barbara Liskov in 1988.

The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass.

An overriding method of a subclass needs to accept the same input parameter values as the method of the superclass. That means you can implement less restrictive validation rules, but you are not allowed to enforce stricter ones in your subclass. Otherwise, in case of polymorphic invocation, it may cause an **exception**, if it gets called with an object of the subclass.

Similar rules apply to the return value of the method. The return value of a method of the subclass needs to comply with the same rules

as the return value of the method of the superclass. You can only decide to apply even stricter rules by returning a specific subclass of the defined return value, or by returning a subset of the valid return values of the superclass.

```
01 class Bird {
02   public void fly(){}
03   public void eat(){}
04 }
05 class Crow extends Bird {}
06 class Ostrich extends Bird{
07   fly(){
08     throw new UnsupportedOperationException();
09   }
10 }
11
12 public BirdTest{
```

```
13   public static void
main(String[] args){
14     List<Bird> birdList = new
ArrayList<Bird>();
15     birdList.add(new Bird());
16     birdList.add(new Crow());
17     birdList.add(new
Ostrich());
18     letTheBirdsFly ( birdList
);
19   }
20   static void letTheBirdsFly (
List<Bird> birdList ){
21     for ( Bird b : birdList )
{
22       b.fly();
23     }
24   }
25}
```

What will happen to the above code? As soon as an Ostrich instance is passed, it blows up!!! **Here the sub type is not replaceable for the super type.**

So this is a violation of LSP.

How do we fix such issues?

In the above scenario we can factor out the fly feature into- Flight and NonFlight birds.

```
1 class Bird{
2   public void eat(){}
3 }
4 class FlightBird extends Bird{
5   public void fly()()
6 }
7 class NonFlight extends Bird{}
```

So instead of dealing with Bird, we can deal with 2 categories of birds- Flight and NonFlight.

i.e.

```
20    static void letTheBirdsFly (
List<FlyingBird> birdList ){
21      for ( FlyingBird b :
birdList ) {
```

```
22        b.fly();
23                                    }
```

Another Example:

```
class MyThread extends
Thread
{
   public void run()
   {
      // some code
   }
}

Thread t1=new Thread();
t1.start();  // here
Thread's run will be
invoked

t1=new MyThread();
t1.start();  // here
MyThread's run will be
invoked
```

So here we've seen that run() still executes but with different implementation, so behaviour is same, implementation is different.

Violation of LSP

```
class Mythread extends Thread
{
    public void start()
    {
        System.out.println(“Hello World”);
    }
}

Thread t1=new Thread();
```

t1.start(); // Thread registration with Schedular

t1=new Mythread();

t1.start(); // Hello world - doesn’t have any meaning

Solution

```
class Mythread extends Thread
{
    public void start()
    {
        super.start();
```

```
            // some genuine code
        }
    }
```