

Unit 16: Liskov Substitution Principle

After this unit, the student should:

- understand the type of bugs that reckless developers can introduce when using inheritance and polymorphism
- understand the Liskov Substitution Principle and thus be aware that not all IS-A relationships should be modeled with inheritance
- know how to explicitly disallow inheritance when writing a class or disallow overriding with the `final` keyword

The Responsibility When Using Inheritance

As you have seen in [Unit 14](#), polymorphism is a powerful tool that allows a client to change the behavior of existing code written by the implementer, behind the abstraction barrier.

As Ben Parker (aka Uncle Ben) said, "With great power, comes great responsibility." The client must use overriding and inheritance carefully. Since they can affect how existing code behaves, they can easily break existing code and introduce bugs. Since the client may not have access to the existing code behind the abstraction barrier, it is often tricky to trace and debug. Furthermore, the implementer would not appreciate it if their code was working perfectly until one day, someone overriding a method causes their code to fail, even without the implementer changing anything in their code.

Ensuring this responsibility cannot be done by the compiler, unfortunately.

It thus becomes a developer's responsibility to ensure that any inheritance with method overriding does not introduce bugs to existing code. This brings us to the *Liskov Substitution Principle* (LSP), which says: "Let $\phi(x)$ be a property provable about objects x of type T . Then $\phi(y)$ should be true for objects y of type S where $S <: T$."

This is consistent with the definition of subtyping, $S <: T$, but spelled out more formally.

Let's consider the following example method, `Module::marksToGrade`, which takes in the marks of a student and returns the grade 'A', 'B', 'C', or 'F' as a `char`. How

`Module::marksToGrade` is implemented is not important. Let's look at how it is used.

```
1 void displayGrade(Module m, double marks) {  
2     char grade = m.marksToGrade(marks);  
3     if (grade == 'A')) {
```

```

4      System.out.println("well done");
5      else if (grade == 'B') {
6          System.out.println("good");
7      else if (grade == 'C') {
8          System.out.println("ok");
9      } else {
10         System.out.println("retake again");
11     }
12 }

```

Now, suppose that one day, someone comes along and create a new class `CSCUModule` that inherits from `Module`, and overrides `marksToGrade` s that it now returns only 'S' and 'U'. Since `CSCUModule` is a subclass of `Module`, we can pass an instance to `displayGrade`:

```

1  displayGrade(new CSCUModule("GEQ1000", 100));

```

and suddenly `displayGrade` is displaying `retake again` even if the student is scoring 100 marks.

We are violating the LSP here. The object `m` has the following property: `m.marksToGrade` always returns something from the set { 'A', 'B', 'C', 'F' }, that the method `displayGrade` depends on explicitly. The subclass `CSCUModule` violated that and makes `m.marksToGrade` returns 'S' or 'U', sabotaging `displayGrade` and causing it to fail.

LSP cannot be enforced by the compiler¹. The properties of an object have to be managed and agreed upon among programmers. A common way is to document these properties as part of the code documentation.

LSP Through the Lens of Testing

Another way to develop an intuition of the LSP is through the lens of testing. When we write a method, we may want to introduce test cases to check that our method is working correctly. These test cases are designed based on the *specification* of our method and not its implementation details². That is, we test based on the expected inputs and resultant outputs.

Let's look at an example. We would like to model a restaurant booking system for a restaurant chain. Consider the following `Restaurant` class. Every restaurant in the chain opens at 12 pm and closes at 10 pm, and has a singular method `canMakeReservation` which allows us to check if the restaurant is available for reservations at a certain `time`. **The requirement given is that, the system must be able to process a reservation during its opening hours.**

```

1  public class Restaurant {
2      public static final int OPENING_HOUR = 1200;

```

```

3   public static final int CLOSING_HOUR = 2200;
4
5   public boolean canMakeReservation(int time) {
6       if (time <= CLOSING_HOUR && time >= OPENING_HOUR) {
7           return true;
8       }
9       return false;
10  }
11  }

```

The method `canMakeReservation` returns `true` when the argument passed in to `time` is between 12 pm and 10 pm. Let's think about how we would test this method. Two important edge cases to test is to check if the method returns true for the stated restaurant opening and closing hours.

```

1   Restaurant r = new Restaurant();
2   r.canMakeReservation(1200) == true; // Is true, therefore test passes
3   r.canMakeReservation(2200) == true; // Is true, therefore test passes

```

Note that these are simple `jshell` tests, in software engineering modules you will learn better ways to design and formalise these tests.

We can now rephrase our LSP in terms of testing. A *subclass* should not break the expectations set by the *superclass*. If a class `B` is substitutable for a parent class `A` then it should be able to pass all test cases of the parent class `A`. If it does not, then it is not substitutable and the LSP is violated.

Lets now consider two subclasses of `Restaurant`, `LunchRestaurant` and `DigitalReadyRestaurant`. Our `LunchRestaurant` does not take reservations during peak hours (12 to 2 pm).

```

1   public class LunchRestaurant extends Restaurant {
2       private final int peakHourStart = 1200;
3       private final int peakHourEnd = 1400;
4
5       @Override
6       public boolean canMakeReservation(int time) {
7           if (time <= peakHourEnd && time >= peakHourStart) {
8               return false;
9           } else if (time <= CLOSING_HOUR && time >= OPENING_HOUR) {
10              return true;
11          }
12          return false;
13      }
14  }

```

As `LunchRestaurant <: Restaurant`, we can point our variable `r` to a new instance of `LunchRestaurant` and run the test cases of the parent class, as can be seen in the code below.

```

1 Restaurant r = new LunchRestaurant();
2 r.canMakeReservation(1200) == true; // Is false, therefore test fails
3 r.canMakeReservation(2200) == true; // Is true, therefore test passes

```

Whilst the second test passes, the first test does not since it falls within the peak lunch hour. Therefore `LunchRestaurant` is not substitutable for `Restaurant` and the LSP is violated. We have changed the expectation of the method in the child class.

Let's suppose the restaurant chain starts to roll out online reservation system for a subset of its restaurants. These restaurants can take reservations any time.

We create a subclass `DigitalReadyRestaurant`, as follows:

```

1 public class DigitalReadyRestaurant extends Restaurant {
2
3     @Override
4     public boolean canMakeReservation(int time) {
5         return true;
6     }
7 }

```

Similarly, as `DigitalReadyRestaurant <: Restaurant`, we can point our variable `r` to a new instance of `DigitalReadyRestaurant` and run the test cases of the parent class, as can be seen in the code below.

```

1 Restaurant r = new DigitalReadyRestaurant();
2 r.canMakeReservation(1200) == true; // Is true, therefore test passes
3 r.canMakeReservation(2200) == true; // Is true, therefore test passes

```

Both test cases pass. In fact, all test cases that pass for `Restaurant` would pass for `DigitalReadyRestaurant`. Therefore `DigitalReadyRestaurant` is substitutable for `Restaurant`. Anywhere we can use an object of type `Restaurant`, we can use `DigitalReadyRestaurant` without breaking any previously written code.

Preventing Inheritance and Method Overriding

Sometimes, it is useful for a developer to explicitly prevent a class to be inherited. Not allowing inheritance would make it much easier to argue for the correctness of programs, something that is important when it comes to writing secure programs. Both the two java classes you have seen, `java.lang.Math` and `java.lang.String`, cannot be inherited from. In Java, we use the keyword `final` when declaring a class to tell Java that we ban this class from being inherited.

```

1 final class Circle {
2     :


```

```
3 } }
```

Alternatively, we can allow inheritance but still prevent a specific method from being overridden, by declaring a method as `final`. Usually, we do this on methods that are critical for the correctness of the class.

For instance,

```
1 class Circle {  
2     :  
3     public final boolean contains(Point p) {  
4         :  
5     }  
6 }
```

-
1. We can use `assert` to check some of the properties though. 
 2. The test cases we are describing here are known as black-box tests and you will encounter these in later modules at NUS. We will not go into any further details in this module. 