# 里氏替换原则的定义

里氏替换原则（Liskov Substitution Principle，LSP）由麻省理工学院计算机科学实验室的里斯科夫（Liskov）女士在 1987 年的“面向对象技术的高峰会议”（OOPSLA）上发表的一篇文章《数据抽象和层次》（Data Abstraction and Hierarchy）中提出：**继承必须确保超类拥有的性质在子类中仍然成立**（Inheritance should ensure that any property proved about supertype objects also holds for subtype objects）。也就是说：当一个子类的实例能够替换任何父类的实例时，它们之间才具有is-A关系。

里氏替换原则主要阐述了有关继承的一些原则，即什么时候应该使用继承，什么时候不应该使用继承，以及其中蕴含的原理。里氏替换原则是继承复用的基础，它反映了基类与子类之间的关系，是开闭原则的补充，是实现抽象化的具体步骤的规范。

# 里氏替换原则的含义

里氏替换原则通俗的将就是：**子类可以拓展父类的功能，但不能改变父类原有的功能。**也就是说：由父类派生出子类时，除了添加新的方法完成新增功能外，尽量不要重写父类的方法。

如果通过重写父类的方法来完成新的功能，这样写起来虽然简单，但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的概率会很大。

# 里氏替换原则的作用

1、是实现开闭原则的重要方式之一。

2、克服了继承机制中重写父类造成的可复用性变差的缺点。

3、是动作正确性的保证。即类的扩展不会给已有的系统引入新的错误，降低了代码出错的可能性。

# 里氏替换原则的实现方法

如果程序违背了里氏替换原则，则继承类的对象在基类出现的地方会出现运行错误。这时其修正方法是：取消原来的继承关系，重新设计它们之间的关系。

# 里氏替换原则的形象理解

在学习OO的时候，我们知道，一个对象时一组状态和一系列行为的组合体。状态是对象的内在特性，行为是对象的外在特性。里氏替换原则所表述的就是在同一个继承体系中的对象应该有相同的行为特征。

在这一点上，体现了OO的继承和日常生活中的继承的本质区别。举个例子：生物学中将企鹅划分为鸟类，我们以此为参照设计出这样的类和关系：“鸟”类中有个fly方法，“企鹅”类继承了这个fly方法，但是我们知道企鹅不会飞，于是我们在“企鹅”类中覆盖了fly方法，告诉调用者：“企鹅是不会飞的”，这样的设计完全符合常理，但是，它违反了里氏替换原则，企鹅是鸟类，可是企鹅却不能飞！需要注意的是，此处的“鸟”已经不再是生物学中的鸟了，它是软件中的一个类、一个抽象。

有人会说：“企鹅不能飞这很正常啊，而且这样编写的代码也能正常编译，只要在使用这个类的客户代码中添加一条判断就好了呀！”，这就是问题的所在！首先，客户代码和“企鹅”的代码很有可能不是同时设计的，在当今软件外包一层又一层的开发模式下，你甚至不知道两个模块的原产地是哪里，也就谈不上去修改客户端代码了。客户程序很可能是遗留系统的一部分，很可能已经不再维护，如果因为设计出这么一个“企鹅”而导致必须修改客户代码，谁应该为此部分工作负责呢？“修改客户代码”直接违反了开闭原则，这就是OCP原则的重要性，违反里氏替换原则将使既有的设计不能封闭。

# 其他问题

## 如何理解里氏替换原则是对开闭原则的补充？

在回答这个问题前先回想一下，什么是开闭原则——“软件实体应该对扩展开放，对修改关闭”，意思是，在软件开发的过程中，不通过修改功能模块、类、接口、方法等实体的方式来实现新的需求，而是用过扩展的方式来满足新的需求。

对类的扩展指的就是由基类派生子类，里氏替换原则对由基类派生子类的具体步骤做了规范，所以说里氏替换原则是对开闭原则的补充。

## LSP原则——关于正方形不是长方形

假设有“长方形”类：

public class Rectangle *{* private double length;  
 private double width;  
  
 public void setLength*(*double length*) {* this.length = length;  
 *}* public void setWidth*(*double width*) {* this.width = width;  
 *}* public double getArea*() {* return length \* width;  
 *}  
}*

有“正方形”类：

public class Square extends Rectangle *{* @Override  
 public void setLength*(*double length*) {* super.setLength*(*length*)*;  
 super.setWidth*(*length*)*;  
 *}* @Override  
 public void setWidth*(*double width*) {* super.setWidth*(*width*)*;  
 super.setLength*(*width*)*;  
 *}  
}*

我们都知道，长方形的面积 = 长 \* 宽，正方形因为4条边的长度相等，所以正方形的面积 = 任意边的长度 \* 任意边的长度。

上面的正方形类继承自长方形类，为了保证正方形4条边的长度相等的性质，在正方形类中覆盖了父类的setLength和setWidth方法。我们想想用户使用时可能会遇到的场景：

public static void main*(*String*[]* args*) {* Rectangle rectangle = new Rectangle*()*;  
 rectangle.setLength*(*3*)*;  
 rectangle.setWidth*(*4*)*;  
 System.*out*.println*(*"长方形的面积 = " + rectangle.getArea*())*;  
*}*

public static void main*(*String*[]* args*) {* Rectangle rectangle = new Square*()*;  
 rectangle.setLength*(*3*)*;  
 rectangle.setWidth*(*4*)*;  
 System.*out*.println*(*"长方形的面积 = " + rectangle.getArea*())*;  
*}*

在场景1中，用户设置长方形的长为3，宽为4，最终程序输出结果为“长方形的面积为12.0”，

在场景2中，用户设置“长方形”的长为3，宽为4，最终程序输出结果为“长方形的面积为16.0”。因为