课程目标

1、通过对节课内容的学习，了解设计原则的重要性。

2、掌握七大设计原则的具体内容。

内容定位

学习设计原则，学习设计模式的基础。在实际开发过程中，并不是一定要求所有代码都遵循设计原则，我们要考虑人力、时间、成本、质量，不是刻意追求完美，要在适当的场景遵循设计原则，体现的是一种平衡取舍，帮助我们设计出更加优雅的代码结构。

开闭原则

开闭原则（Open-Closed Principle, OCP）是指一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。所谓的开闭，也正是对扩展和修改两个行为的一个原则。强调的是用抽象构建框架，用实现扩展细节。可以提高软件系统的可复用性及可维护性。开闭原则，是面向对象设计中最基础的设计原则。它指导我们如何建立稳定灵活的系统，例如：我们版本更新，我尽可能不修改源代码，但是可以增加新功能。

在现实生活中对于开闭原则也有体现。比如，很多互联网公司都实行弹性制作息时间，规定每天工作 8 小时。意思就是说，对于每天工作 8 小时这个规定是关闭的，但是你什么时候来，什么时候走是开放的。早来早走，晚来晚走。

实现开闭原则的核心思想就是面向抽象编程，接下来我们来看一段代码：

以咕泡学院的课程体系为例，首先创建一个课程接口 ICourse：

```
public interface ICourse {
    Integer getId();
    String getName();
    Double getPrice();
}
```

整个课程生态有 Java 架构、大数据、人工智能、前端、软件测试等，我们来创建一个 Java 架构课

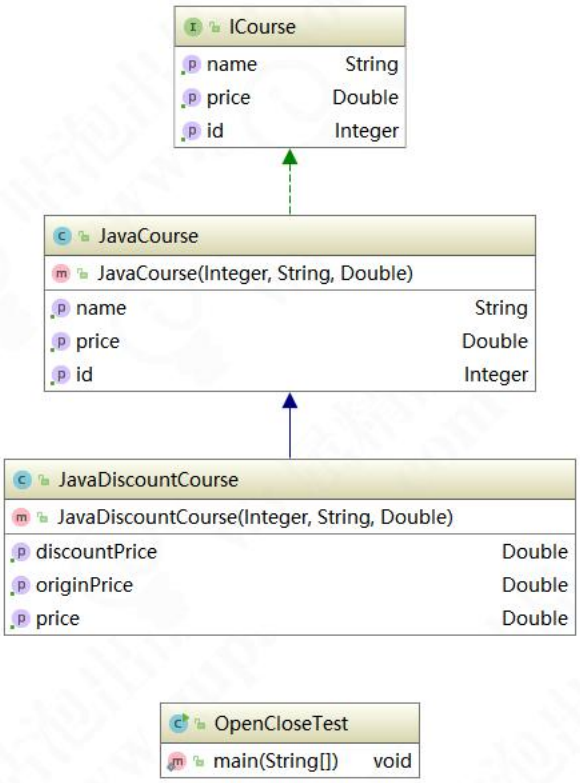
程的类 JavaCourse：

```
public class JavaCourse implements ICourse{
    private Integer Id;
    private String name;
    private Double price;
    public JavaCourse(Integer id, String name, Double price) {
        this.Id = id;
        this.name = name;
        this.price = price;
    }
    public Integer getId() {
        return this.Id;
    }
    public String getName() {
        return this.name;
    }
    public Double getPrice() {
        return this.price;
    }
}
```

现在我们要给 Java 架构课程做活动，价格优惠。如果修改 JavaCourse 中的 getPrice()方法，则会存在一定的风险，可能影响其他地方的调用结果。我们如何在不修改原有代码前提前下，实现价格优惠这个功能呢？现在，我们再写一个处理优惠逻辑的类，JavaDiscountCourse 类（思考一下为什么要叫 JavaDiscountCourse，而不叫 DiscountCourse）：

```
public class JavaDiscountCourse extends JavaCourse {
    public JavaDiscountCourse(Integer id, String name, Double price) {
        super(id, name, price);
    }
    public Double getOriginPrice(){
        return super.getPrice();
    }
    public Double getPrice(){
        return super.getPrice() * 0.61;
    }
}
```

回顾一下，简单一下类结构图：



依赖倒置原则

依赖倒置原则（Dependence Inversion Principle,DIP）是指设计代码结构时，高层模块不应该依赖底层模块，二者都应该依赖其抽象。抽象不应该依赖细节；细节应该依赖抽象。通过依赖倒置，可以减少类与类之间的耦合性，提高系统的稳定性，提高代码的可读性和可维护性，并能够降低修改程序所造成的风险。接下来看一个案例，还是以课程为例，先来创建一个类 Tom：

```
public class Tom {
    public void studyJavaCourse(){
        System.out.println("Tom 在学习 Java 的课程");
    }

    public void studyPythonCourse(){
        System.out.println("Tom 在学习 Python 的课程");
    }
}
```

来调用一下：

```
public static void main(String[] args) {
    Tom tom = new Tom();
    tom.studyJavaCourse();
    tom.studyPythonCourse();
}
```

Tom 热爱学习，目前正在学习 Java 课程和 Python 课程。大家都知道，学习也是会上瘾的。随着学习兴趣的暴涨，现在 Tom 还想学习 AI 人工智能的课程。这个时候，业务扩展，我们的代码要从底层到高层（调用层）一次修改代码。在 Tom 类中增加 studyAICourse()的方法，在高层也要追加调用。如此一来，系统发布以后，实际上是非常不稳定的，在修改代码的同时也会带来意想不到的风险。接下来我们优化代码，创建一个课程的抽象 ICourse 接口：

```
public interface ICourse {
    void study();
}
```

然后写 JavaCourse 类：

```
public class JavaCourse implements ICourse {
    @Override
    public void study() {
        System.out.println("Tom 在学习 Java 课程");
    }
}
```

再实现 PythonCourse 类：

```
public class PythonCourse implements ICourse {
    @Override
    public void study() {
        System.out.println("Tom 在学习 Python 课程");
    }
}
```

修改 Tom 类：

```
public class Tom {
    public void study(ICourse course){
        course.study();
    }
}
```

来看调用：

```
public static void main(String[] args) {
    Tom tom = new Tom();
    tom.study(new JavaCourse());
    tom.study(new PythonCourse());
}
```

我们这时候再看来代码，Tom 的兴趣无论怎么暴涨，对于新的课程，我只需要新建一个类，通过传参的方式告诉 Tom，而不需要修改底层代码。实际上这是一种大家非常熟悉的方式，叫依赖注入。注入的方式还有构造器方式和 setter 方式。我们来看构造器注入方式：

```
public class Tom {

    private ICourse course;
```

```
    public Tom(ICourse course){
        this.course = course;
    }

    public void study(){
        course.study();
    }
}
```

看调用代码：

```
public static void main(String[] args) {
    Tom tom = new Tom(new JavaCourse());
    tom.study();
}
```

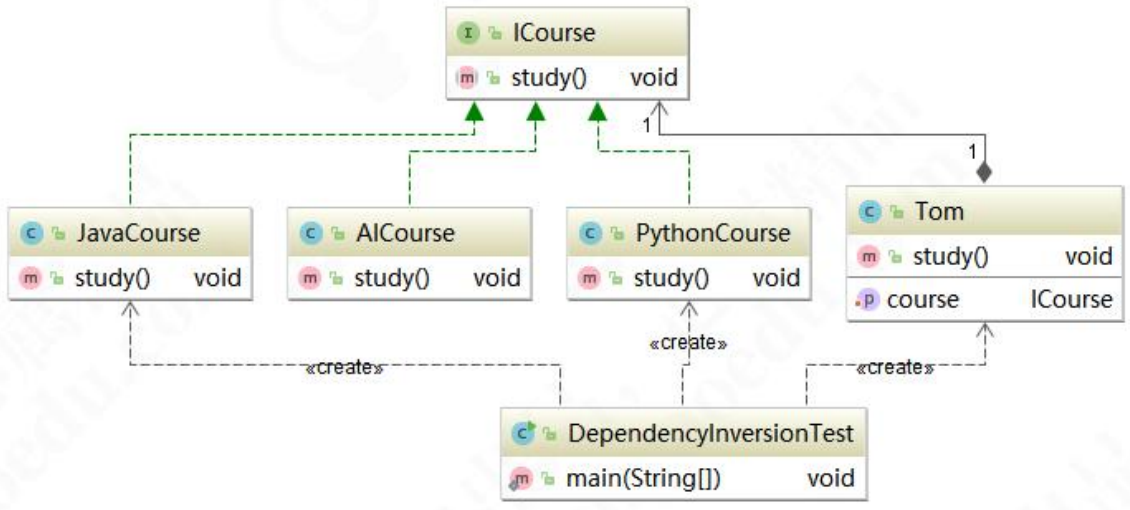
根据构造器方式注入，在调用时，每次都要创建实例。那么，如果 Tom 是全局单例，则我们就只能选择用 Setter 方式来注入，继续修改 Tom 类的代码：

```
public class Tom {
    private ICourse course;
    public void setCourse(ICourse course) {
        this.course = course;
    }
    public void study(){
        course.study();
    }
}
```

看调用代码：

```
public static void main(String[] args) {
    Tom tom = new Tom();
    tom.setCourse(new JavaCourse());
    tom.study();

    tom.setCourse(new PythonCourse());
    tom.study();
}
```

现在我们再来看最终的类图：



大家要切记：以抽象为基准比以细节为基准搭建起来的架构要稳定得多，因此大家在拿到需求之后，要面向接口编程，先顶层再细节来设计代码结构。

单一职责原则

单一职责（Simple Responsibility Pinciple，SRP）是指不要存在多于一个导致类变更的原因。假设我们有一个 Class 负责两个职责，一旦发生需求变更，修改其中一个职责的逻辑代码，有可能会导致另一个职责的功能发生故障。这样一来，这个 Class 存在两个导致类变更的原因。如何解决这个问题呢？我们就要给两个职责分别用两个 Class 来实现，进行解耦。后期需求变更维护互不影响。这样的设计，可以降低类的复杂度，提高类的可读性，提高系统的可维护性，降低变更引起的风险。总体来说就是一个 Class/Interface/Method 只负责一项职责。

接下来，我们来看代码实例，还是用课程举例，我们的课程有直播课和录播课。直播课不能快进和快退，录播可以可以任意的反复观看，功能职责不一样。还是先创建一个 Course 类：

```
public class Course {
    public  void study(String courseName){
        if("直播课".equals(courseName)){
            System.out.println("不能快进");
        }else{
            System.out.println("可以任意的来回播放");
        }
    }
}
```

看代码调用：

```
public static void main(String[] args) {
    Course course = new Course();
    course.study("直播课");
    course.study("录播课");
}
```

从上面代码来看，Course 类承担了两种处理逻辑。假如，现在要对课程进行加密，那么直播课和录播课的加密逻辑都不一样，必须要修改代码。而修改代码逻辑势必会相互影响容易造成不可控的风险。我们对职责进行分离解耦，来看代码，分别创建两个类 ReplayCourse 和 LiveCourse：

LiveCourse 类：

```
public class LiveCourse {
    public void study(String courseName){
```

```
        System.out.println(courseName + "不能快进看");
    }
}
```

ReplayCourse 类：

```
public class ReplayCourse {
    public  void  study(String courseName){
        System.out.println("可以任意的来回播放");
    }
}
```

调用代码：

```
public static void main(String[] args) {
    LiveCourse liveCourse = new LiveCourse();
    liveCourse.study("直播课");

    ReplayCourse replayCourse = new ReplayCourse();
    replayCourse.study("录播课");
}
```

业务继续发展，课程要做权限。没有付费的学员可以获取课程基本信息，已经付费的学员可以获得视频流，即学习权限。那么对于控制课程层面上至少有两个职责。我们可以把展示职责和管理职责分离开来，都实现同一个抽象依赖。设计一个顶层接口,创建 ICourse 接口：

```
public interface ICourse {

    //获得基本信息
    String getCourseName();

    //获得视频流
    byte[] getCourseVideo();

    //学习课程
    void studyCourse();
    //退款
    void refundCourse();
}
```

我们可以把这个接口拆成两个接口，创建一个接口 ICourseInfo 和 ICourseManager：

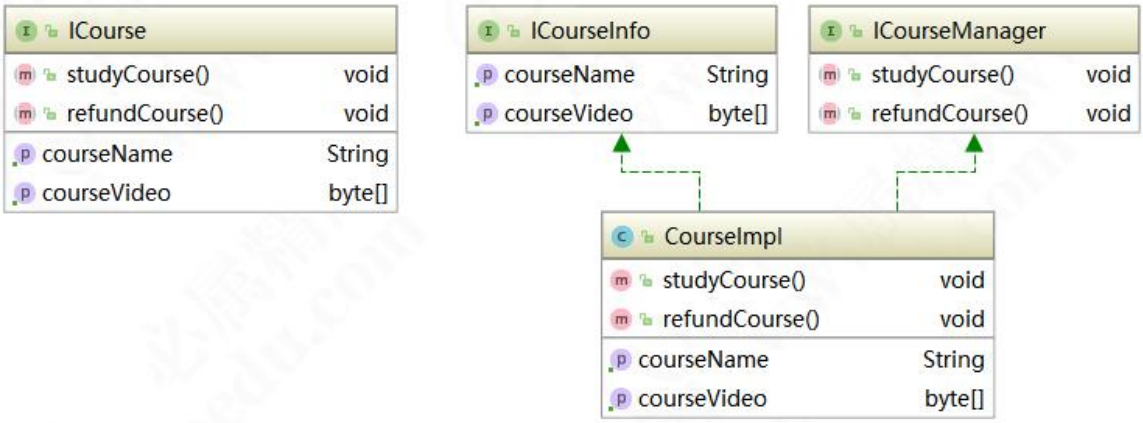
ICourseInfo 接口：

```
public interface ICourseInfo {
    String getCourseName();
    byte[] getCourseVideo();
}
```

ICourseManager 接口：

```
public interface ICourseManager {
    void studyCourse();
    void refundCourse();
}
```

来看一下类图：



下面我们来看一下方法层面的单一职责设计。有时候，我们为了偷懒，通常会把一个方法写成下面这样：

```
private void modifyUserInfo(String userName,String address){
    userName = "Tom";
    address = "Changsha";
}
```

还可能写成这样：

```
private void modifyUserInfo(String userName,String... fileds){
        userName = "Tom";
//        address = "Changsha";
}
private void modifyUserInfo(String userName,String address,boolean bool){
    if(bool){

    }else{

    }

    userName = "Tom";
    address = "Changsha";
}
```

显然，上面的 modifyUserInfo()方法中都承担了多个职责，既可以修改 userName,也可以修改 address，甚至更多，明显不符合单一职责。那么我们做如下修改，把这个方法拆成两个：

```
private void modifyUserName(String userName){
    userName = "Tom";
}
private void modifyAddress(String address){
    address = "Changsha";
}
```

这修改之后，开发起来简单，维护起来也容易。但是，我们在实际开发中会项目依赖，组合，聚合这些关系，还有还有项目的规模，周期，技术人员的水平，对进度的把控，很多类都不符合单一职责。但是，我们在编写代码的过程，尽可能地让接口和方法保持单一职责，对我们项目后期的维护是有很大

帮助的。

接口隔离原则

接口隔离原则（Interface Segregation Principle, ISP）是指用多个专门的接口，而不使用单一的总接口，客户端不应该依赖它不需要的接口。这个原则指导我们在设计接口时应当注意一下几点：

1、一个类对一类的依赖应该建立在最小的接口之上。

2、建立单一接口，不要建立庞大臃肿的接口。

3、尽量细化接口，接口中的方法尽量少（不是越少越好，一定要适度）。

接口隔离原则符合我们常说的高内聚低耦合的设计思想，从而使得类具有很好的可读性、可扩展性和可维护性。我们在设计接口的时候，要多花时间去思考，要考虑业务模型，包括以后有可能发生变更的地方还要做一些预判。所以，对于抽象，对业务模型的理解是非常重要的。下面我们来看一段代码，写一个动物行为的抽象：

IAnimal 接口：

```
public interface IAnimal {
    void eat();
    void fly();
    void swim();
}
```

Bird 类实现：

```
public class Bird implements IAnimal {
    @Override
    public void eat() {}
    @Override
    public void fly() {}
    @Override
    public void swim() {}
}
```

Dog 类实现：

```
public class Dog implements IAnimal {
    @Override
    public void eat() {}
    @Override
    public void fly() {}
    @Override
    public void swim() {}
}
```

可以看出，Bird 的 swim()方法可能只能空着，Dog 的 fly()方法显然不可能的。这时候，我们针对不同动物行为来设计不同的接口，分别设计 IEatAnimal，IFlyAnimal 和 ISwimAnimal 接口，来看代码：

IEatAnimal 接口：

```
public interface IEatAnimal {
    void eat();
}
```

IFlyAnimal 接口：

```
public interface IFlyAnimal {
    void fly();
}
```

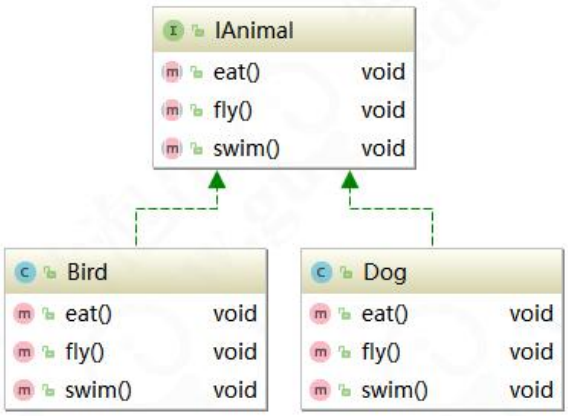
ISwimAnimal 接口：

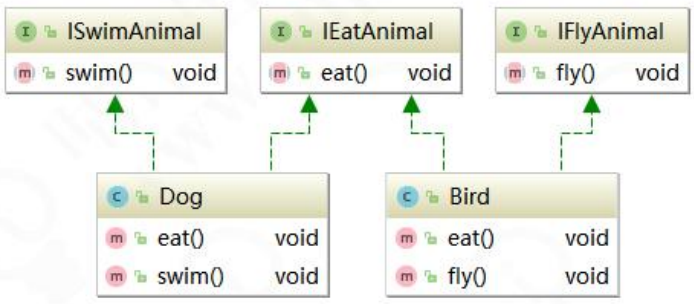
```
public interface ISwimAnimal {
    void swim();
}
```

Dog 只实现 IEatAnimal 和 ISwimAnimal 接口：

```
public class Dog implements ISwimAnimal,IEatAnimal {
    @Override
    public void eat() {}
    @Override
    public void swim() {}
}
```

来看下两种类图的对比，还是非常清晰明了的：





迪米特法则

迪米特原则（Law of Demeter LoD）是指一个对象应该对其他对象保持最少的了解，又叫最少知道原则（Least Knowledge Principle,LKP），尽量降低类与类之间的耦合。迪米特原则主要强调只和

朋友交流，不和陌生人说话。出现在成员变量、方法的输入、输出参数中的类都可以称之为成员朋友类，而出现在方法体内部的类不属于朋友类。

现在来设计一个权限系统，TeamLeader 需要查看目前发布到线上的课程数量。这时候，TeamLeader 要找到员工 Employee 去进行统计，Employee 再把统计结果告诉 TeamLeader。接下来我们还是来看代码：

Course 类：

```
public class Course {
}
```

Employee 类：

```
public class Employee{
    public void checkNumberOfCourses(List<Course> courseList){
        System.out.println("目前已发布的课程数量是：" + courseList.size());
    }
}
```

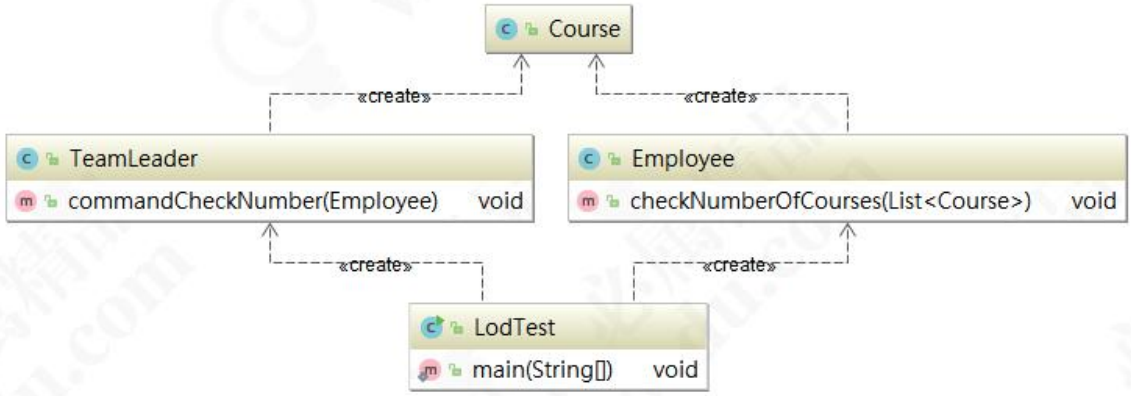
TeamLeader 类：

```
public class TeamLeader{
    public void commandCheckNumber(Employee employee){

        List<Course> courseList = new ArrayList<Course>();
        for (int i= 0; i < 20 ;i ++){
            courseList.add(new Course());
        }
        employee.checkNumberOfCourses(courseList);
    }
}
```

测试代码：

```
public static void main(String[] args) {
    TeamLeader teamLeader = new TeamLeader();
    Employee employee = new Employee();
    teamLeader.commandCheckNumber(employee);
}
```

写到这里，其实功能已经都已经实现，代码看上去也没什么问题。根据迪米特原则，TeamLeader 只想要结果，不需要跟 Course 产生直接的交流。而 Employee 统计需要引用 Course 对象。TeamLeader 和 Course 并不是朋友，从下面的类图就可以看出来：



下面来对代码进行改造：

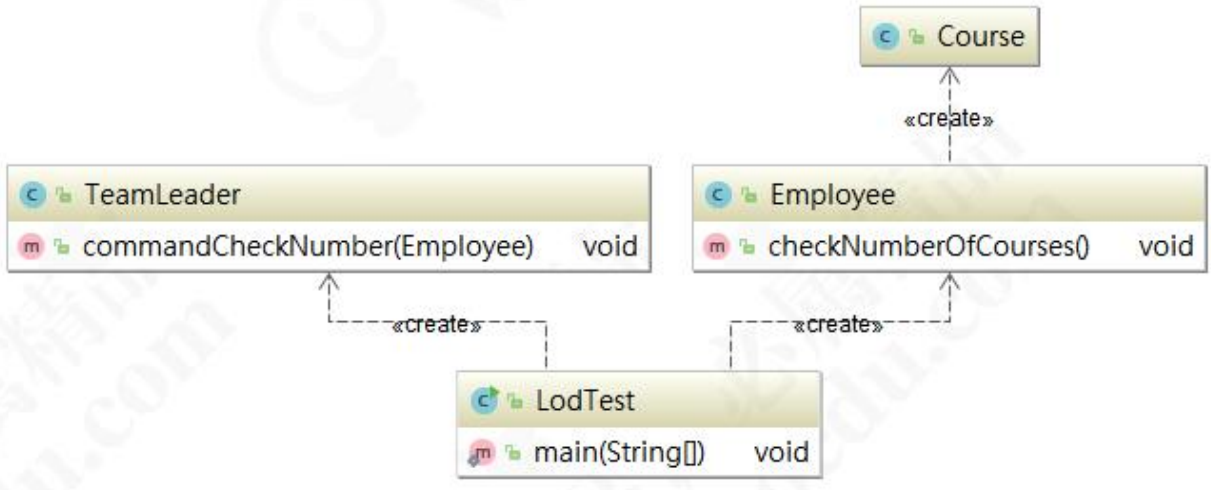
Employee 类：

```
public class Employee {

    public void checkNumberOfCourses(){
        List<Course> courseList = new ArrayList<Course>();
        for (int i= 0; i < 20 ;i ++){
            courseList.add(new Course());
        }
        System.out.println("目前已发布的课程数量是："+courseList.size());
    }
}
```

TeamLeader 类：

```
public class TeamLeader {

    public void commandCheckNumber(Employee employee){
        employee.checkNumberOfCourses();
    }
}
```

再来看下面的类图，Course 和 TeamLeader 已经没有关联了。



学习软件设计原则，千万不能形成强迫症。碰到业务复杂的场景，我们需要随机应变。

里氏替换原则

里氏替换原则（Liskov Substitution Principle,LSP）是指如果对每一个类型为 T1 的对象 o1,都有类型为 T2 的对象 o2,使得以 T1 定义的所有程序 P 在所有的对象 o1 都替换成 o2 时，程序 P 的行为没有发生变化，那么类型 T2 是类型 T1 的子类型。

定义看上去还是比较抽象，我们重新理解一下，可以理解为一个软件实体如果适用一个父类的话，那一定是适用于其子类，所有引用父类的地方必须能透明地使用其子类的对象，子类对象能够替换父类对象，而程序逻辑不变。根据这个理解，我们总结一下：

引申含义：子类可以扩展父类的功能，但不能改变父类原有的功能。

1、子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。

2、子类中可以增加自己特有的方法。

3、当子类的方法重载父类的方法时，方法的前置条件（即方法的输入/入参）要比父类方法的输入参数更宽松。

4、当子类的方法实现父类的方法时（重写/重载或实现抽象方法），方法的后置条件（即方法的输出/返回值）要比父类更严格或相等。

在前面讲开闭原则的时候埋下了一个伏笔，我们记得在获取折后时重写覆盖了父类的 getPrice()方法，增加了一个获取源码的方法 getOriginPrice()，显然就违背了里氏替换原则。我们修改一下代码，不应该覆盖 getPrice()方法，增加 getDiscountPrice()方法：

```
public class JavaDiscountCourse extends JavaCourse {
    public JavaDiscountCourse(Integer id, String name, Double price) {
        super(id, name, price);
    }
    public Double getDiscountPrice(){
        return super.getPrice() * 0.61;
    }
}
```

使用里氏替换原则有以下优点：

1、约束继承泛滥，开闭原则的一种体现。

2、加强程序的健壮性，同时变更时也可以做到非常好的兼容性，提高程序的维护性、扩展性。降低需求变更时引入的风险。

现在来描述一个经典的业务场景，用正方形、矩形和四边形的关系说明里氏替换原则，我们都知道正方形是一个特殊的长方形，那么就可以创建一个长方形父类 Rectangle 类：

```
public class Rectangle {
    private long height;
    private long width;

    public long getHeight() {
        return height;
    }

    public void setHeight(long height) {
        this.height = height;
    }

    public long getWidth() {
        return width;
    }

    public void setWidth(long width) {
        this.width = width;
    }
}
```

创建正方形 Square 类继承长方形：

```
public class Square extends Rectangle {
    private long length;

    public long getLength() {
        return length;
    }

    public void setLength(long length) {
        this.length = length;
    }

    @Override
    public long getHeight() {
        return getLength();
    }

    @Override
    public void setHeight(long height) {
        setLength(height);
    }

    @Override
    public long getWidth() {
        return getLength();
    }
```

```
    @Override
    public void setWidth(long width) {
        setLength(width);
    }
}
```

在测试类中创建 resize()方法，根据逻辑长方形的宽应该大于等于高，我们让高一直自增，知道高等于宽变成正方形：

```
    public static void resize(Rectangle rectangle){
        while (rectangle.getWidth() >= rectangle.getHeight()){
            rectangle.setHeight(rectangle.getHeight() + 1);
            System.out.println("Width:" +rectangle.getWidth() +",Height:" + rectangle.getHeight());
        }
        System.out.println("Resize End,Width:" +rectangle.getWidth() +",Height:" + rectangle.getHeight());
    }
```

测试代码：

```
public static void main(String[] args) {
    Rectangle rectangle = new Rectangle();
    rectangle.setWidth(20);
    rectangle.setHeight(10);
    resize(rectangle);
}
```

运行结果：

```
Width:20,Height:11
Width:20,Height:12
Width:20,Height:13
Width:20,Height:14
Width:20,Height:15
Width:20,Height:16
Width:20,Height:17
Width:20,Height:18
Width:20,Height:19
Width:20,Height:20
Width:20,Height:21
Resize End,Width:20,Height:21
```

发现高比宽还大了，在长方形中是一种非常正常的情况。现在我们再来看下面的代码，把长方形 Rectangle 替换成它的子类正方形 Square，修改测试代码：

```
public static void main(String[] args) {
    Square square = new Square();
    square.setLength(10);
    resize(square);
}
```

这时候我们运行的时候就出现了死循环，违背了里氏替换原则，将父类替换为子类后，程序运行结果没有达到预期。因此，我们的代码设计是存在一定风险的。里氏替换原则只存在父类与子类之间，约

束继承泛滥。我们再来创建一个基于长方形与正方形共同的抽象四边形 Quadrangle 接口：

```
public interface QuadRangle {
    long getWidth();
    long getHeight();
}
```

修改长方形 Rectangle 类：

```
public class Rectangle implements QuadRangle {
    private long height;
    private long width;

    public long getHeight() {
        return height;
    }

    public void setHeight(long height) {
        this.height = height;
    }

    public long getWidth() {
        return width;
    }

    public void setWidth(long width) {
        this.width = width;
    }
}
```

修改正方形类 Square 类：

```
public class Square implements QuadRangle {
    private long length;

    public long getLength() {
        return length;
    }

    public void setLength(long length) {
        this.length = length;
    }

    public long getWidth() {
        return length;
    }

    public long getHeight() {
        return length;
    }
}
```

此时，如果我们把 resize()方法的参数换成四边形 Quadrangle 类，方法内部就会报错。因为正方形 Square 已经没有了 setWidth()和 setHeight()方法了。因此，为了约束继承泛滥，resize()的方法参数只能用 Rectangle 长方形。当然，我们在后面的设计模式课程中还会继续深入讲解。

合成复用原则

合成复用原则（Composite/Aggregate Reuse Principle,CARP）是指尽量使用对象组合(has-a)/聚合(contanis-a)，而不是继承关系达到软件复用的目的。可以使系统更加灵活，降低类与类之间的耦合度，一个类的变化对其他类造成的影响相对较少。

继承我们叫做白箱复用，相当于把所有的实现细节暴露给子类。组合/聚合也称之为黑箱复用，对类以外的对象是无法获取到实现细节的。要根据具体的业务场景来做代码设计，其实也都需要遵循 OOP 模型。还是以数据库操作为例，先来创建 DBConnection 类：

```
public class DBConnection {
    public String getConnection(){
        return "MySQL 数据库连接";
    }
}
```

创建 ProductDao 类：

```
public class ProductDao{
    private DBConnection dbConnection;
    public void setDbConnection(DBConnection dbConnection) {
        this.dbConnection = dbConnection;
    }
    public void addProduct(){
        String conn = dbConnection.getConnection();
        System.out.println("使用"+conn+"增加产品");
    }
}
```

这就是一种非常典型的合成复用原则应用场景。但是，目前的设计来说，DBConnection 还不是一种抽象，不便于系统扩展。目前的系统支持 MySQL 数据库连接，假设业务发生变化，数据库操作层要支持 Oracle 数据库。当然，我们可以在 DBConnection 中增加对 Oracle 数据库支持的方法。但是违背了开闭原则。其实，我们可以不必修改 Dao 的代码，将 DBConnection 修改为 abstract，来看代码：

```
public abstract class DBConnection {
    public abstract String getConnection();
}
```

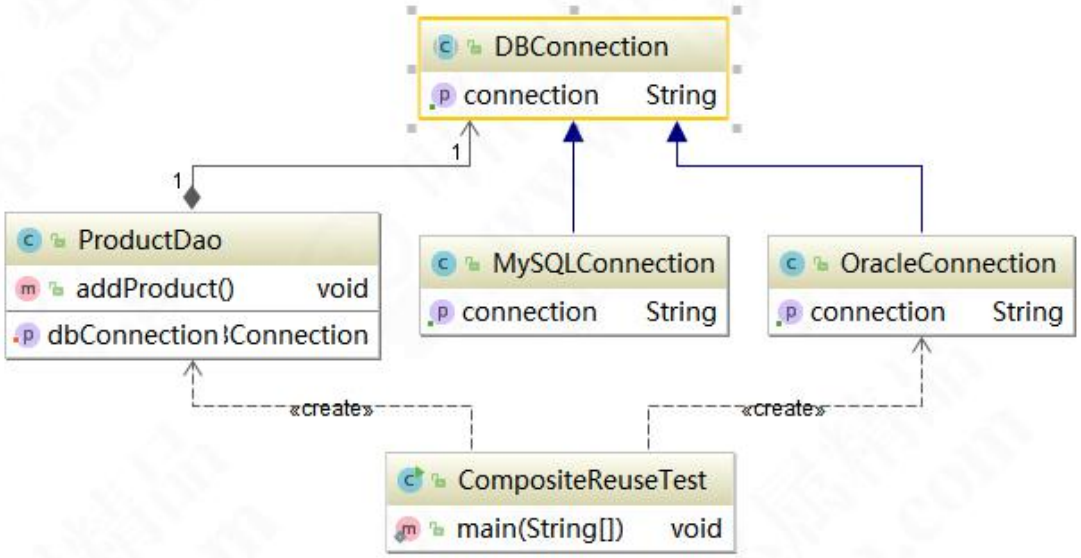
然后，将 MySQL 的逻辑抽离：

```
public class MySQLConnection extends DBConnection {
    @Override
    public String getConnection() {
        return "MySQL 数据库连接";
    }
}
```

再创建 Oracle 支持的逻辑：

```
public class OracleConnection extends DBConnection {
    @Override
    public String getConnection() {
        return "Oracle 数据库连接";
    }
}
```

具体选择交给应用层，来看一下类图：



设计原则总结

学习设计原则，学习设计模式的基础。在实际开发过程中，并不是一定要求所有代码都遵循设计原则，我们要考虑人力、时间、成本、质量，不是刻意追求完美，要在适当的场景遵循设计原则，体现的是一种平衡取舍，帮助我们设计出更加优雅的代码结构。