**Java 封装**

在面向对象程式设计方法中，封装（英语：Encapsulation）是指一种将抽象性函式接口的实现细节部份包装、隐藏起来的方法。

封装可以被认为是一个保护屏障，防止该类的代码和数据被外部类定义的代码随机访问。

要访问该类的代码和数据，必须通过严格的接口控制。

封装最主要的功能在于我们能修改自己的实现代码，而不用修改那些调用我们代码的程序片段。

适当的封装可以让程式码更容易理解与维护，也加强了程式码的安全性。

**封装的优点**

* 1. 良好的封装能够减少耦合。
* 2. 类内部的结构可以自由修改。
* 3. 可以对成员变量进行更精确的控制。
* 4. 隐藏信息，实现细节。

**Java通过private设置私有变量，通过public将变量设置成公开的。**

## 实例

让我们来看一个java封装类的例子：

EncapTest.java 文件代码：

/\* 文件名: EncapTest.java \*/ public class EncapTest{ private String name; private String idNum; private int age; public int getAge(){ return age; } public String getName(){ return name; } public String getIdNum(){ return idNum; } public void setAge( int newAge){ age = newAge; } public void setName(String newName){ name = newName; } public void setIdNum( String newId){ idNum = newId; } }

以上实例中public方法是外部类访问该类成员变量的入口。

通常情况下，这些方法被称为getter和setter方法。

因此，任何要访问类中私有成员变量的类都要通过这些getter和setter方法。

通过如下的例子说明EncapTest类的变量怎样被访问：

RunEncap.java 文件代码：

/\* F文件名 : RunEncap.java \*/

public class RunEncap{

public static void main(String args[]){

EncapTest encap = new EncapTest();

encap.setName("James");

encap.setAge(20);

encap.setIdNum("12343ms");

System.out.print("Name : " + encap.getName()+ " Age : "+ encap.getAge());

}

}

以上代码编译运行结果如下:

Name : James Age : 20

## 实现Java封装的步骤

1. 修改属性的可见性来限制对属性的访问（一般限制为private），例如：

public class Person {

private String name;

private int age;

}

这段代码中，将 **name** 和 **age** 属性设置为私有的，只能本类才能访问，其他类都访问不了，如此就对信息进行了隐藏。

2. 对每个值属性提供对外的公共方法访问，也就是创建一对赋取值方法，用于对私有属性的访问，例如：

public class Person{

private String name;

private int age; ​

public int getAge(){

return age;

} ​

public String getName(){

return name;

} ​

public void setAge(int age){

this.age = age;

} ​

public void setName(String name){

this.name = name;

}

}

采用 **this** 关键字是为了解决实例变量（private String name）和局部变量（setName(String name)中的name变量）之间发生的同名的冲突。

# Java 继承

## 继承的概念

继承是java面向对象编程技术的一块基石，因为它允许创建分等级层次的类。

继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。

### 类的继承格式

在 Java 中通过 extends 关键字可以申明一个类是从另外一个类继承而来的，一般形式如下：

## 类的继承格式

class 父类 { }

class 子类 extends 父类 { }

代码中存在重复了，导致后果就是代码量大且臃肿，而且维护性不高(维护性主要是后期需要修改的时候，就需要修改很多的代码，容易出错)，所以要从根本上解决这两段代码的问题，就需要继承，将两段代码中相同的部分提取出来组成 一个父类

公共父类：

public class Animal {

private String name;

private int id;

public Animal(String myName, int myid) {

name = myName; id = myid;

}

public void eat(){

System.out.println(name+"正在吃");

}

public void sleep(){

System.out.println(name+"正在睡");

}

public void introduction() {

System.out.println("大家好！我是" + id + "号" + name + ".");

}

}

这个Animal类就可以作为一个父类，然后企鹅类和老鼠类继承这个类之后，就具有父类当中的属性和方法，子类就不会存在重复的代码，维护性也提高，代码也更加简洁，提高代码的复用性（复用性主要是可以多次使用，不用再多次写同样的代码） 继承之后的代码：

企鹅类：

public class Penguin extends Animal {

public Penguin(String myName, int myid) {

super(myName, myid);

}

}

老鼠类：

public class Mouse extends Animal {

public Mouse(String myName, int myid) {

super(myName, myid);

}

}

## 继承的特性

* 子类拥有父类非private的属性，方法。
* 子类可以拥有自己的属性和方法，即子类可以对父类进行扩展。
* 子类可以用自己的方式实现父类的方法。
* Java的继承是单继承，但是可以多重继承，单继承就是一个子类只能继承一个父类，多重继承就是，例如A类继承B类，B类继承C类，所以按照关系就是C类是B类的父类，B类是A类的父类，这是java继承区别于C++继承的一个特性。
* 提高了类之间的耦合性（继承的缺点，耦合度高就会造成代码之间的联系）。

## 继承关键字

继承可以使用 extends 和 implements 这两个关键字来实现继承，而且所有的类都是继承于 java.lang.Object，当一个类没有继承的两个关键字，则默认继承object（这个类在 **java.lang** 包中，所以不需要 **import**）祖先类。

### extends关键字

在 Java 中，类的继承是单一继承，也就是说，一个子类只能拥有一个父类，所以 extends 只能继承一个类。

在 Java 中，类的继承是单一继承，也就是说，一个子类只能拥有一个父类，所以 extends 只能继承一个类。

## extends 关键字

public class Animal {

private String name;

private int id;

public Animal(String myName, String myid) {

//初始化属性值

}

public void eat() {

//吃东西方法的具体实现

}

public void sleep() {

//睡觉方法的具体实现

}

}

public class Penguin extends Animal{

}

### implements关键字

使用 implements 关键字可以变相的使java具有多继承的特性，使用范围为类继承接口的情况，可以同时继承多个接口（接口跟接口之间采用逗号分隔）。

## implements 关键字

public interface A {

public void eat();

public void sleep();

}

public interface B {

public void show();

}

public class C implements A,B {

}

### super 与 this 关键字

super关键字：我们可以通过super关键字来实现对父类成员的访问，用来引用当前对象的父类。

this关键字：指向自己的引用。

## 实例

class Animal {

void eat() {

System.out.println("animal : eat");

}

}

class Dog extends Animal {

void eat() {

System.out.println("dog : eat");

}

void eatTest() {

this.eat(); // this 调用自己的方法

super.eat(); // super 调用父类方法

}

}

public class Test {

public static void main(String[] args) {

Animal a = new Animal();

a.eat();

Dog d = new Dog();

d.eatTest();

}

}

输出结果为：

animal : eat

dog : eat

animal : eat

### final关键字

final 关键字声明类可以把类定义为不能继承的，即最终类；或者用于修饰方法，该方法不能被子类重写：

* 声明类：

final class 类名 {//类体}

* 声明方法：

修饰符(public/private/default/protected) final 返回值类型 方法名(){//方法体}

**注**:实例变量也可以被定义为 final，被定义为 final 的变量不能被修改。被声明为 final 类的方法自动地声明为 final，但是实例变量并不是 final

## 构造器

子类不能继承父类的构造器（构造方法或者构造函数），但是父类的构造器带有参数的，则必须在子类的构造器中显式地通过super关键字调用父类的构造器并配以适当的参数列表。

如果父类有无参构造器，则在子类的构造器中用super调用父类构造器不是必须的，如果没有使用super关键字，系统会自动调用父类的无参构造器。

## 实例

class SuperClass {

private int n;

SuperClass(){

System.out.println("SuperClass()");

} SuperClass(int n) {

System.out.println("SuperClass(int n)");

this.n = n;

}

}

class SubClass extends SuperClass{

private int n;

SubClass(){

super(300);

System.out.println("SubClass");

}

public SubClass(int n){

System.out.println("SubClass(int n):"+n);

this.n = n;

}

}

public class TestSuperSub{

public static void main (String args[]){

SubClass sc = new SubClass();

SubClass sc2 = new SubClass(200);

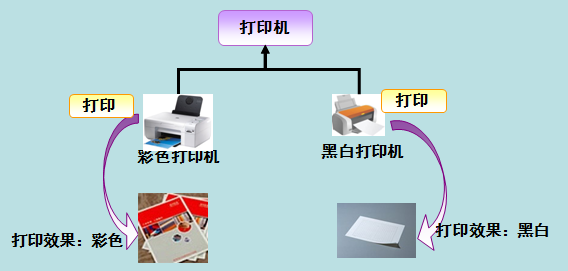
}

}

**Java 多态**

多态是同一个行为具有多个不同表现形式或形态的能力。

多态就是同一个接口，使用不同的实例而执行不同操作，如图所示：



多态性是对象多种表现形式的体现。

*现实中，比如我们按下 F1 键这个动作：*

* 如果当前在 Flash 界面下弹出的就是 AS 3 的帮助文档；
* 如果当前在 Word 下弹出的就是 Word 帮助；
* 在 Windows 下弹出的就是 Windows 帮助和支持。

*同一个事件发生在不同的对象上会产生不同的结果。*

**多态的优点**

* 1. 消除类型之间的耦合关系
* 2. 可替换性
* 3. 可扩充性
* 4. 接口性
* 5. 灵活性
* 6. 简化性

**多态存在的三个必要条件**

* 继承
* 重写
* 父类引用指向子类对象

比如：

Parent p = new Child();

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，再去调用子类的同名方法。

多态的好处：可以使程序有良好的扩展，并可以对所有类的对象进行通用处理。

以下是一个多态实例的演示，详细说明请看注释：

public class Test {

public static void main(String[] args) {

show(new Cat()); // 以 Cat 对象调用 show 方法

show(new Dog()); // 以 Dog 对象调用 show 方法

Animal a = new Cat(); // 向上转型

* 1. eat(); // 调用的是 Cat 的 eat

Cat c = (Cat)a; // 向下转型 c.work(); // 调用的是 Cat 的 work

}

public static void show(Animal a) {

1. eat(); // 类型判断

if (a instanceof Cat) {

// 猫做的事情

Cat c = (Cat)a; c.work(); 4

} else if (a instanceof Dog) {

// 狗做的事情

Dog c = (Dog)a; c.work();

}

}

}

abstract class Animal {

abstract void eat();

}

class Cat extends Animal {

public void eat() {

System.out.println("吃鱼");

}

public void work() {

System.out.println("抓老鼠");

}

}

class Dog extends Animal {

public void eat() {

System.out.println("吃骨头");

}

public void work() {

System.out.println("看家");

}

}

## 多态的实现方式

### 方式一：重写：

这个内容已经在上一章节详细讲过，就不再阐述，详细可访问：[Java 重写(Override)与重载(Overload)](http://www.runoob.com/java/java-override-overload.html)。

### 方式二：接口

* 1. 生活中的接口最具代表性的就是插座，例如一个三接头的插头都能接在三孔插座中，因为这个是每个国家都有各自规定的接口规则，有可能到国外就不行，那是因为国外自己定义的接口类型。
* 2. java中的接口类似于生活中的接口，就是一些方法特征的集合，但没有方法的实现。具体可以看 [java接口](http://www.runoob.com/java/java-interfaces.html) 这一章节的内容。

# Java 重写(Override)与重载(Overload)

## 重写(Override)

重写是子类对父类的允许访问的方法的实现过程进行重新编写, 返回值和形参都不能改变。**即外壳不变，核心重写！**

重写的好处在于子类可以根据需要，定义特定于自己的行为。 也就是说子类能够根据需要实现父类的方法。

重写方法不能抛出新的检查异常或者比被重写方法申明更加宽泛的异常。例如： 父类的一个方法申明了一个检查异常 IOException，但是在重写这个方法的时候不能抛出 Exception 异常，因为 Exception 是 IOException 的父类，只能抛出 IOException 的子类异常。

在面向对象原则里，重写意味着可以重写任何现有方法。实例如下：

## TestDog.java 文件代码：

class Animal{ public void move(){ System.out.println("动物可以移动"); } } class Dog extends Animal{ public void move(){ System.out.println("狗可以跑和走"); } } public class TestDog{ public static void main(String args[]){ Animal a = new Animal(); // Animal 对象 Animal b = new Dog(); // Dog 对象 a.move();// 执行 Animal 类的方法 b.move();//执行 Dog 类的方法 } }

以上实例编译运行结果如下：

动物可以移动

狗可以跑和走

在上面的例子中可以看到，尽管b属于Animal类型，但是它运行的是Dog类的move方法。

这是由于在编译阶段，只是检查参数的引用类型。

然而在运行时，Java虚拟机(JVM)指定对象的类型并且运行该对象的方法。

因此在上面的例子中，之所以能编译成功，是因为Animal类中存在move方法，然而运行时，运行的是特定对象的方法。

思考以下例子：

## TestDog.java 文件代码：

class Animal{

public void move(){

System.out.println("动物可以移动");

}

}

class Dog extends Animal{

public void move(){

System.out.println("狗可以跑和走");

}

public void bark(){

System.out.println("狗可以吠叫");

}

}

public class TestDog{

public static void main(String args[]){

Animal a = new Animal(); // Animal 对象

Animal b = new Dog(); // Dog 对象

1. move();

// 执行 Animal 类的方法

1. move();

//执行 Dog 类的方法

1. bark();

}

}

以上实例编译运行结果如下：

TestDog.java:30: cannot find symbol

symbol : method bark()

location: class Animal

b.bark();

^

该程序将抛出一个编译错误，因为b的引用类型Animal没有bark方法。

## 方法的重写规则

* 参数列表必须完全与被重写方法的相同；
* 返回类型必须完全与被重写方法的返回类型相同；
* 访问权限不能比父类中被重写的方法的访问权限更低。例如：如果父类的一个方法被声明为public，那么在子类中重写该方法就不能声明为protected。
* 父类的成员方法只能被它的子类重写。
* 声明为final的方法不能被重写。
* 声明为static的方法不能被重写，但是能够被再次声明。
* 子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为private和final的方法。
* 子类和父类不在同一个包中，那么子类只能够重写父类的声明为public和protected的非final方法。
* 重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。
* 构造方法不能被重写。
* 如果不能继承一个方法，则不能重写这个方法。

## Super关键字的使用

当需要在子类中调用父类的被重写方法时，要使用super关键字。

## TestDog.java 文件代码：

class Animal{

public void move(){

System.out.println("动物可以移动");

}

}

class Dog extends Animal{

public void move(){

super.move(); // 应用super类的方法

System.out.println("狗可以跑和走");

}

}

public class TestDog{

public static void main(String args[]){

Animal b = new Dog(); // Dog 对象

1. move(); //执行 Dog类的方法

}

}

以上实例编译运行结果如下：

动物可以移动

狗可以跑和走

## 重载(Overload)

重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。

每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

最常用的地方就是构造器的重载。

**重载规则:**

* 被重载的方法必须改变参数列表(参数个数或类型不一样)；
* 被重载的方法可以改变返回类型；
* 被重载的方法可以改变访问修饰符；
* 被重载的方法可以声明新的或更广的检查异常；
* 方法能够在同一个类中或者在一个子类中被重载。
* 无法以返回值类型作为重载函数的区分标准。

### 实例

## Overloading.java 文件代码：

public class Overloading {

public int test(){

System.out.println("test1");

return 1;

}

public void test(int a){

System.out.println("test2");

}

//以下两个参数类型顺序不同

public String test(int a,String s){

System.out.println("test3");

return "returntest3";

}

public String test(String s,int a){

System.out.println("test4");

return "returntest4";

}

public static void main(String[] args){

Overloading o = new Overloading();

System.out.println(o.test());

o.test(1);

System.out.println(o.test(1,"test3"));

System.out.println(o.test("test4",1));

}

}

## 重写与重载之间的区别

|  |  |  |
| --- | --- | --- |
| **区别点** | **重载方法** | **重写方法** |
| 参数列表 | 必须修改 | 一定不能修改 |
| 返回类型 | 可以修改 | 一定不能修改 |
| 异常 | 可以修改 | 可以减少或删除，一定不能抛出新的或者更广的异常 |
| 访问 | 可以修改 | 一定不能做更严格的限制（可以降低限制） |

## 总结

方法的重写(Overriding)和重载(Overloading)是java多态性的不同表现，重写是父类与子类之间多态性的一种表现，重载可以理解成多态的具体表现形式。

* (1)方法重载是一个类中定义了多个方法名相同,而他们的参数的数量不同或数量相同而类型和次序不同,则称为方法的重载(Overloading)。
* (2)方法重写是在子类存在方法与父类的方法的名字相同,而且参数的个数与类型一样,返回值也一样的方法,就称为重写(Overriding)。
* (3)方法重载是一个类的多态性表现,而方法重写是子类与父类的一种多态性表现。

