**1instanceof关键字**

需要掌握的要点：

掌握instanceof关键字的作用以及使用时机。

可以使用instanceof对对象的转型进行安全验证。

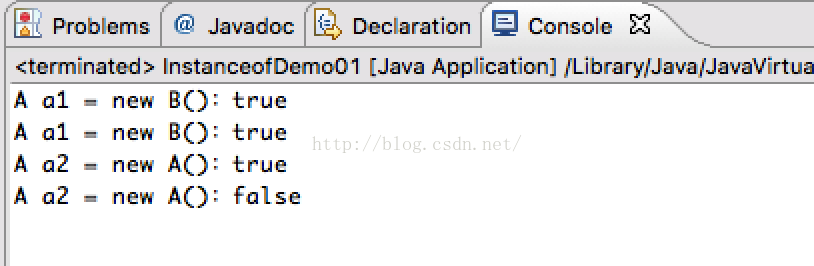
具体内容：

在Java中可以使用instanceof关键字判断一个对象到底是哪个类的实例。

**对象 instanceof 类 —>返回boolean类型**

代码验证如下：

1. class A{ *// 定义类A*
2. public void fun1(){ *// 定义fun1()方法*
3. System.out.println("A --> public void fun1(){}") ;
4. }
5. public void fun2(){
6. this.fun1() ; *// 调用fun1()方法*
7. }
8. };
9. class B extends A{
10. public void fun1(){ *// 此方法被子类覆写了*
11. System.out.println("B --> public void fun1(){}") ;
12. }
13. public void fun3(){
14. System.out.println("B --> public void fun3(){}") ;
15. }
16. };
17. public class InstanceofDemo01{
18. public static void main(String asrgs[]){
19. A a1 = new B() ; *// 通过向上转型实例化对象*
20. System.out.println("A a1 = new B()：" + (a1 instanceof A)) ;
21. System.out.println("A a1 = new B()：" + (a1 instanceof B)) ;
22. A a2 = new A() ; *// 通过向上转型实例化对象*
23. System.out.println("A a2 = new A()：" + (a2 instanceof A)) ;
24. System.out.println("A a2 = new A()：" + (a2 instanceof B)) ;
25. }
26. };

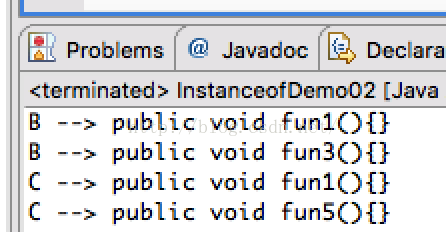


使用instanceof关键字可以做哪些事情？

从之前在对象多态性的研究代码中可以发现一个问题。

如果现在传入的实例是B类的实例，则需要调用fun3()方法，如果是C类则需要调用fun5方法。

1. class A{ *// 定义类A*
2. public void fun1(){ *// 定义fun1()方法*
3. System.out.println("A --> public void fun1(){}") ;
4. }
5. public void fun2(){
6. this.fun1() ; *// 调用fun1()方法*
7. }
8. };
9. class B extends A{
10. public void fun1(){ *// 此方法被子类覆写了*
11. System.out.println("B --> public void fun1(){}") ;
12. }
13. public void fun3(){
14. System.out.println("B --> public void fun3(){}") ;
15. }
16. };
17. class C extends A{
18. public void fun1(){ *// 此方法被子类覆写了*
19. System.out.println("C --> public void fun1(){}") ;
20. }
21. public void fun5(){
22. System.out.println("C --> public void fun5(){}") ;
23. }
24. };
25. public class InstanceofDemo02{
26. public static void main(String asrgs[]){
27. fun(new B()) ;
28. fun(new C()) ;
29. }
30. public static void fun(A a){
31. a.fun1() ;
32. if(a instanceof B){
33. B b = (B) a ;
34. b.fun3() ;
35. }
36. if(a instanceof C){
37. C c = (C) a ;
38. c.fun5() ;
39. }
40. }
41. };



在开发中一定要注意，对于向下转型操作最好增加验证，以保证转型时不会发生ClassCastException。

如果现在要增加新的子类，则肯定要修改fun方法，这样一来程序就失去了灵活性，所以在程序的开发中重点的设计应该放在父类上，只要父类设计足够合理，则会带来很大的方便。

而且在开发中一定要注意以下一点：

一个类永远不要去继承一个已经实现好的类。而只能继承抽象类或实现接口。

instanceof总结：

1、instanceof用于判断一个对象是否是某个类的实例。

2、在对象向下转型之前最好使用instanceof关键字进行验证。

# 1．抽象类与接口的应用

需要掌握的有关内容如下：

掌握抽象类以及接口的实例化操作。

掌握模板设计的作用。

掌握工厂设计模式的作用。

掌握代理设计模式的作用。

掌握适配器设计模式的作用。

掌握抽象类与抽象接口的使用区别。

为抽象类与接口实例化：

在Java中可以通过对象多态性，为抽象类和接口实例化这样再使用抽象类和接口的时候就可以调用被子类中所覆写过的方法了。

之所以抽象类和接口不能直接实例化，是因为其内部包含了各个抽象方法，抽象方法本身都是未实现的方法，所以无法调用。

通过对象的多态性可以发现，子类发生了向上转型关系之后（Animal a=new cat(）)，所调用的全部方法都是被覆写过的方法。

1. abstract class A{ *// 定义抽象类A*
2. public abstract void print() ; *// 定义抽象方法print()*
3. };
4. class B extends A { *// 定义子类，继承抽象类*
5. public void print(){ *// 覆写抽象方法*
6. System.out.println("Hello World!!!") ;
7. }
8. };
9. public class AbstractCaseDemo01{
10. public static void main(String args[]){
11. A a = new B() ; *// 通过子类为抽象类实例化*
12. a.print() ;
13. }
14. };

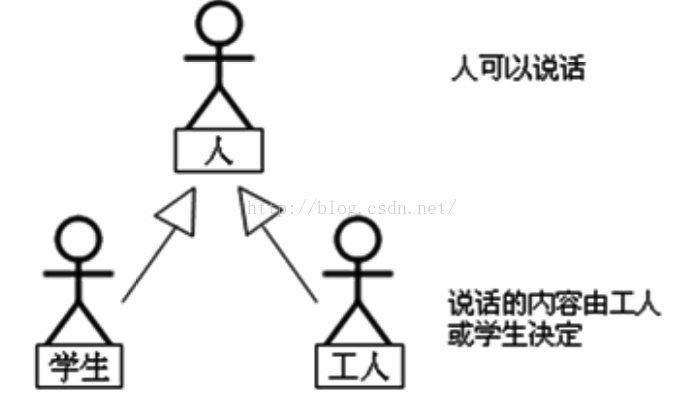
可以继续使用此概念，为一个接口实例化。

1. interface A{ *// 定义抽象类A*
2. public abstract void print() ; *// 定义抽象方法print()*
3. };
4. class B implements A { *// 定义子类，继承抽象类*
5. public void print(){ *// 覆写抽象方法*
6. System.out.println("Hello World!!!") ;
7. }
8. };
9. public class AbstractCaseDemo01{
10. public static void main(String args[]){
11. A a = new B() ; *// 通过子类为抽象类实例化*
12. a.print() ;
13. }
14. };

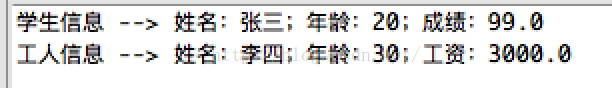
证明，如果要使用抽象类或接口，则只能按照以上的操作完成。

**抽象类的实际应用——模板设计**

场景如下：假设人分为学生和工人，学生和工人都可以说话，但是学生和工人说话的内容是不一样的，也就是说说话这个功能应该是一个具体的功能，而说话的内容就要由学生和工人来决定了。所以此时就可以使用抽象类来实现这种场景。

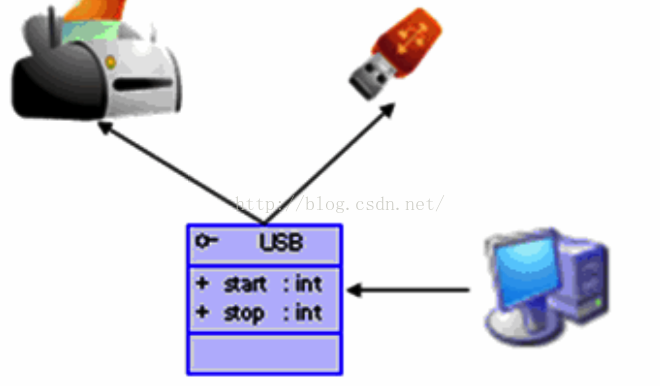


1. abstract class Person{
2. private String name ; *// 定义name属性*
3. private int age ; *// 定义age属性*
4. public Person(String name,int age){
5. this.name = name ;
6. this.age = age ;
7. }
8. public String getName(){
9. return this.name ;
10. }
11. public int getAge(){
12. return this.age ;
13. }
14. public void say(){ *// 人说话是一个具体的功能*
15. System.out.println(this.getContent()) ; *// 输出内容*
16. }
17. public abstract String getContent() ; *// 说话的内容由子类决定*
18. };
19. class Student extends Person{
20. private float score ;
21. public Student(String name,int age,float score){
22. super(name,age) ; *// 调用父类中的构造方法*
23. this.score = score ;
24. }
25. public String getContent(){
26. return "学生信息 --> 姓名：" + super.getName() +
27. "；年龄：" + super.getAge() +
28. "；成绩：" + this.score ;
29. }
30. };
31. class Worker extends Person{
32. private float salary ;
33. public Worker(String name,int age,float salary){
34. super(name,age) ; *// 调用父类中的构造方法*
35. this.salary = salary ;
36. }
37. public String getContent(){
38. return "工人信息 --> 姓名：" + super.getName() +
39. "；年龄：" + super.getAge() +
40. "；工资：" + this.salary ;
41. }
42. };
43. public class AbstractCaseDemo02{
44. public static void main(String args[]){
45. Person per1 = null ; *// 声明Person对象*
46. Person per2 = null ; *// 声明Person对象*
47. per1 = new Student("张三",20,99.0f) ; *// 学生是一个人*
48. per2 = new Worker("李四",30,3000.0f) ; *// 工人是一个人*
49. per1.say() ; *// 学生说学生的话*
50. per2.say() ; *// 工人说工人的话*
51. }
52. };



**接口的实际应用——制定标准**

接口在实际中更多的作用是用来制定标准的。比如说：U盘和打印机都可以插在电脑上使用，这是因为它们都实现了USB接口，对于电脑来说只要是符合了USB接口标准的设备就都可以插进来。



1. interface USB{ *// 定义了USB接口*
2. public void start() ; *// USB设备开始工作*
3. public void stop() ; *// USB设备结束工作*
4. }
5. class Computer{
6. public static void plugin(USB usb){ *// 电脑上可以插入USB设备*
7. usb.start() ;
8. System.out.println("=========== USB 设备工作 ========") ;
9. usb.stop() ;
10. }
11. };
12. class Flash implements USB{
13. public void start(){ *// 覆写方法*
14. System.out.println("U盘开始工作。") ;
15. }
16. public void stop(){ *// 覆写方法*
17. System.out.println("U盘停止工作。") ;
18. }
19. };
20. class Print implements USB{
21. public void start(){ *// 覆写方法*
22. System.out.println("打印机开始工作。") ;
23. }
24. public void stop(){ *// 覆写方法*
25. System.out.println("打印机停止工作。") ;
26. }
27. };
28. public class InterfaceCaseDemo02{
29. public static void main(String args[]){
30. Computer.plugin(new Flash()) ;
31. Computer.plugin(new Print()) ;
32. }
33. };



深入研究的话，接口实际上还表示将方法的视图暴漏给远程的客户端。

**工厂设计模式**

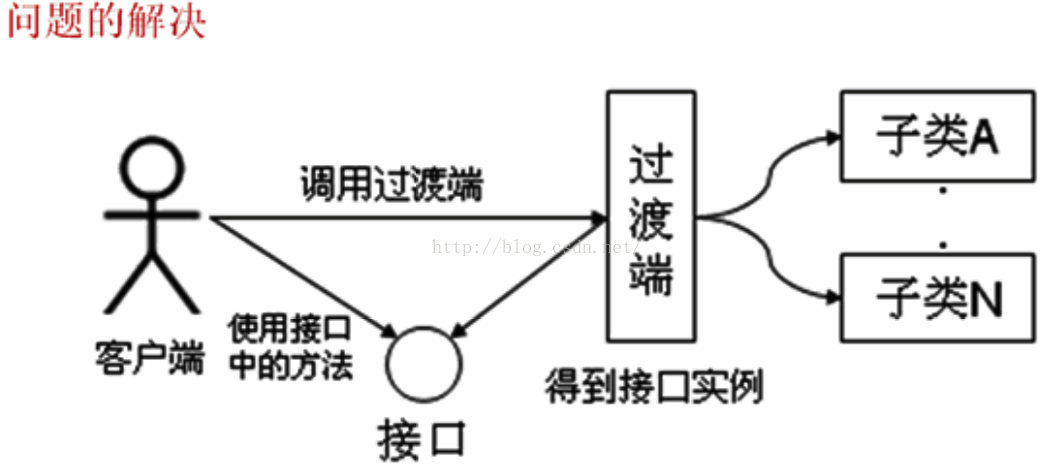
工厂设计模式，是在Java开发中最常使用的一种设计模式。

1. interface Fruit{ *// 定义一个水果接口*
2. public void eat() ; *// 吃水果*
3. }
4. class Apple implements Fruit{
5. public void eat(){
6. System.out.println("\*\* 吃苹果。") ;
7. }
8. };
9. class Orange implements Fruit{
10. public void eat(){
11. System.out.println("\*\* 吃橘子。") ;
12. }
13. };
14. public class InterfaceCaseDemo03{
15. public static void main(String args[]){
16. Fruit f = new Apple() ; *// 实例化接口*
17. f.eat() ;
18. }
19. };

这样的代码可以使用吗？有问题吗？

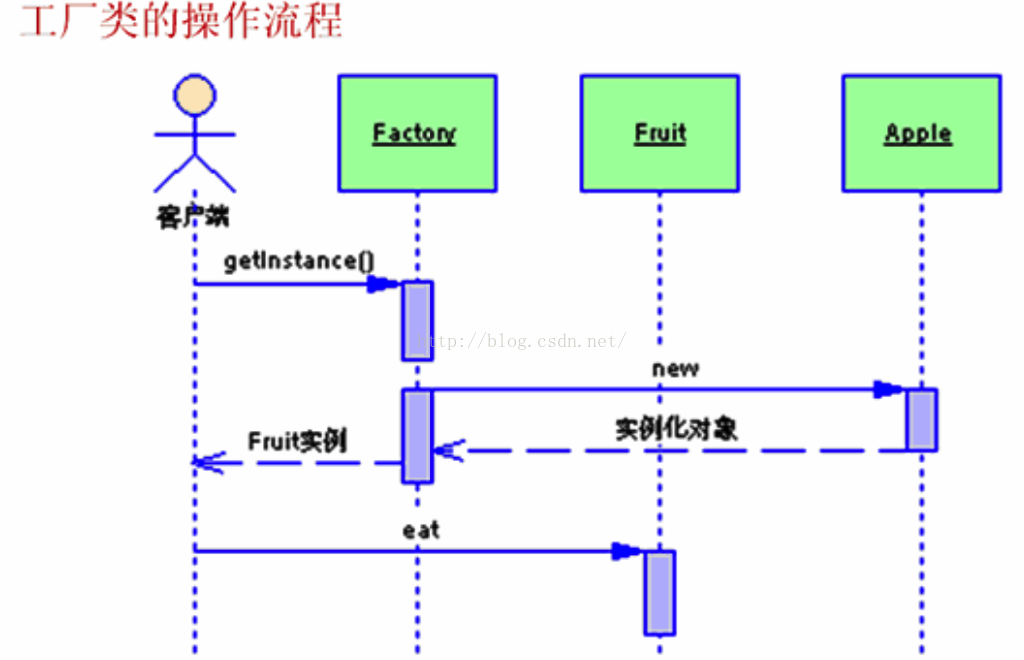
分析：主方法：应该表示一个客户端，主方法的代码越少越好。此时，直接在主方法中指定了要操作的子类，如果要更换子类，则肯定要修改客户端。就表示跟特定的子类紧密耦合在一起了。

JVM原理：程序—>JVM—>操作系统。



过渡端在程序中就称为工厂设计。

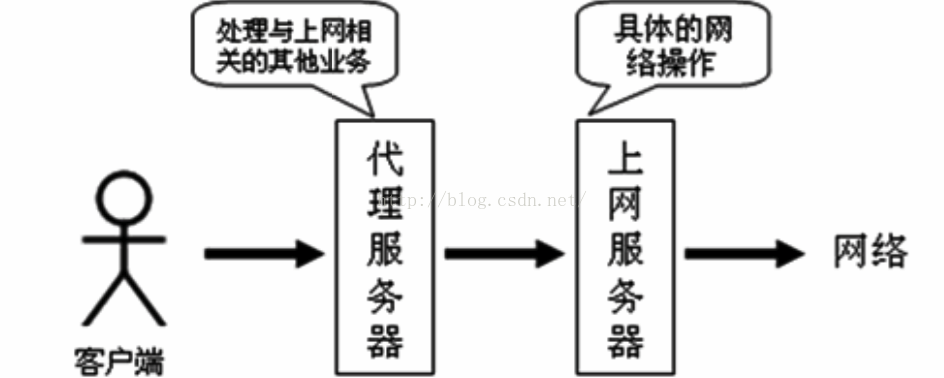
1. interface Fruit{ *// 定义一个水果接口*
2. public void eat() ; *// 吃水果*
3. }
4. class Apple implements Fruit{
5. public void eat(){
6. System.out.println("\*\* 吃苹果。") ;
7. }
8. };
9. class Orange implements Fruit{
10. public void eat(){
11. System.out.println("\*\* 吃橘子。") ;
12. }
13. };
14. class Factory{ *// 定义工厂类*
15. public static Fruit getInstance(String className){
16. Fruit f = null ;
17. if("apple".equals(className)){ *// 判断是否要的是苹果的子类*
18. f = new Apple() ;
19. }
20. if("orange".equals(className)){ *// 判断是否要的是橘子的子类*
21. f = new Orange() ;
22. }
23. return f ;
24. }
25. };
26. public class InterfaceCaseDemo05{
27. public static void main(String args[]){
28. Fruit f = Factory.getInstance(args[0]) ; *// 实例化接口*
29. if(f!=null){ *// 判断是否取得实例*
30. f.eat() ;
31. }
32. }
33. };



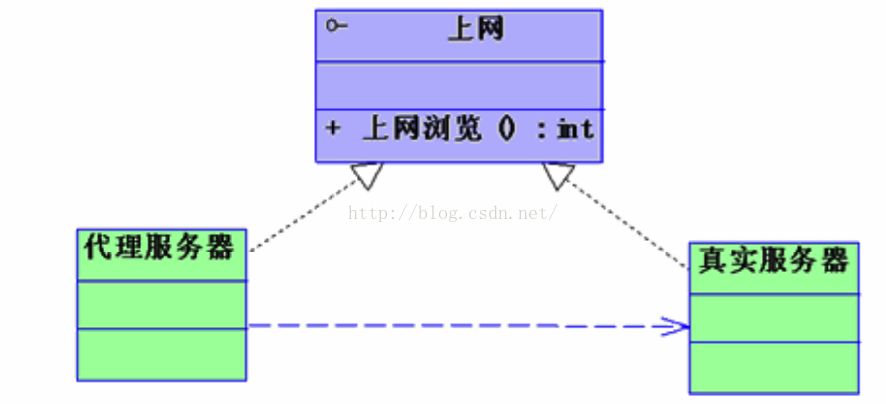
**代理模式**

设计模式—代理设计

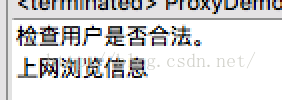
代理设计也是在Java开发中使用较多的一种设计模式，所谓的代理设计就是指一个代理主题来操作真实主题，真实主题执行具体的业务操作，而代理主题负责其他相关业务的处理，就好比在生活中经常使用到的代理上网那样，客户通过网络代理连接网络，由代理服务器完成用户权限，访问限制等上网操作相关的操作。

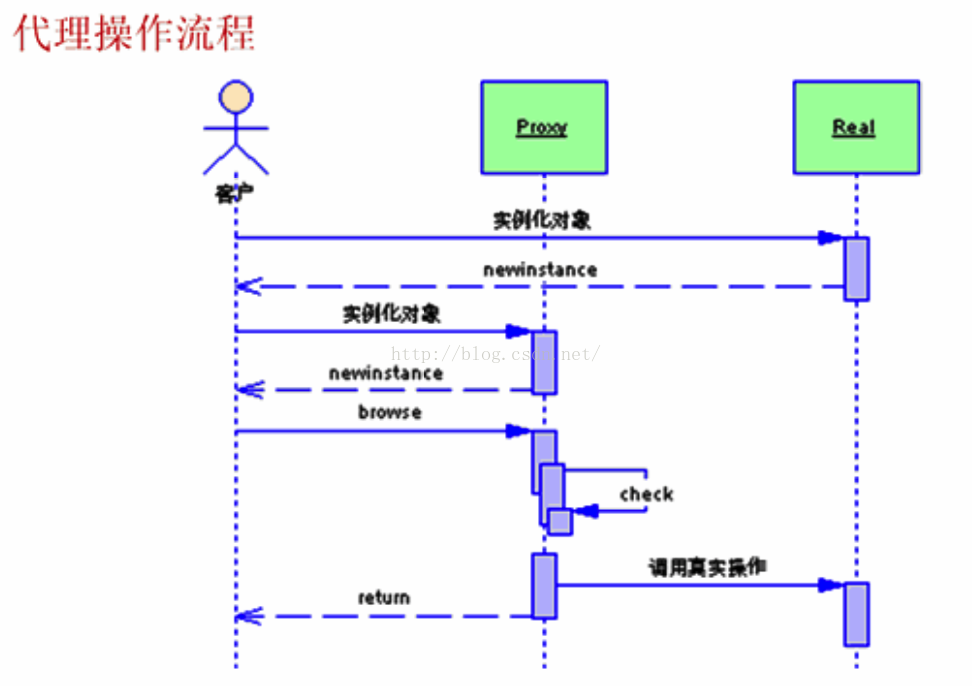


分析结果：不管是代理操作也好，真实操作也好，其共同目的就是上网，所以用户关心的只是如何上网，至于里面是如何操作的用户并不关心。



1. interface Network{
2. public void browse() ; *// 浏览*
3. }
4. class Real implements Network{
5. public void browse(){
6. System.out.println("上网浏览信息") ;
7. }
8. };
9. class Proxy implements Network{
10. private Network network ; *// 代理对象*
11. public Proxy(Network network){
12. this.network = network ;
13. }
14. public void check(){
15. System.out.println("检查用户是否合法。") ;
16. }
17. public void browse(){
18. this.check() ;
19. this.network.browse() ; *// 调用真实的主题操作*
20. }
21. };
22. public class ProxyDemo{
23. public static void main(String args[]){
24. Network net = null ;
25. net = new Proxy(new Real()) ;*// 指定代理操作*
26. net.browse() ; *// 客户只关心上网浏览一个操作*
27. }
28. };



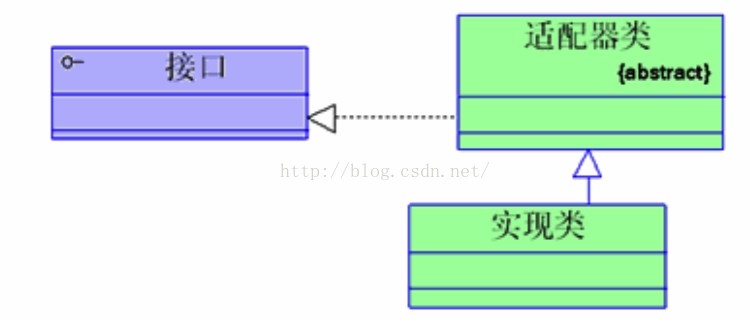


**适配器设计**

适配器设计主要在Java的图形界面上使用最多。

设计模式——适配器设计

对于Java程序来说，如果一个类要实现一个接口，则肯定要覆写此接口中的全部抽象方法，那么如果，此时一个接口中定义的抽象方法过多，但是在子类中又用不到这么多的抽象方法，则肯定很麻烦，所以此时就需要一个中间的过渡，但是此过渡又不希望被直接使用，所以将此过渡类定义成抽象类最合适，即一个接口首先被一个抽象类（此抽象类通常被称为适配器类），并在此抽象类中实现若干方法（方法体为空）则以后的子类直接继承此抽象类，就可以有选择的覆写所需要的方法。必须实现的方法在适配器中定义成抽象方法，可选择实现的方法定义成方法体为空的方法即可。



1. interface Window{ *// 定义Window接口，表示窗口操作*
2. public void open() ; *// 打开*
3. public void close() ; *// 关闭*
4. public void activated() ; *// 窗口活动*
5. public void iconified() ; *// 窗口最小化*
6. public void deiconified();*// 窗口恢复大小*
7. }
8. abstract class WindowAdapter implements Window{
9. public void open(){} ; *// 打开*
10. public void close(){} ; *// 关闭*
11. public void activated(){} ; *// 窗口活动*
12. public void iconified(){} ; *// 窗口最小化*
13. public void deiconified(){};*// 窗口恢复大小*
14. };
15. class WindowImpl extends WindowAdapter{
16. public void open(){
17. System.out.println("窗口打开。") ;
18. }
19. public void close(){
20. System.out.println("窗口关闭。") ;
21. }
22. };
23. public class AdapterDemo{
24. public static void main(String args[]){
25. Window win = new WindowImpl() ;
26. win.open() ;
27. win.close() ;
28. }
29. };

此种设计思路，在Java的图形界面编程上使用的非常多，但是在JavaEE的开发中并不常见。

**内部类的拓展**

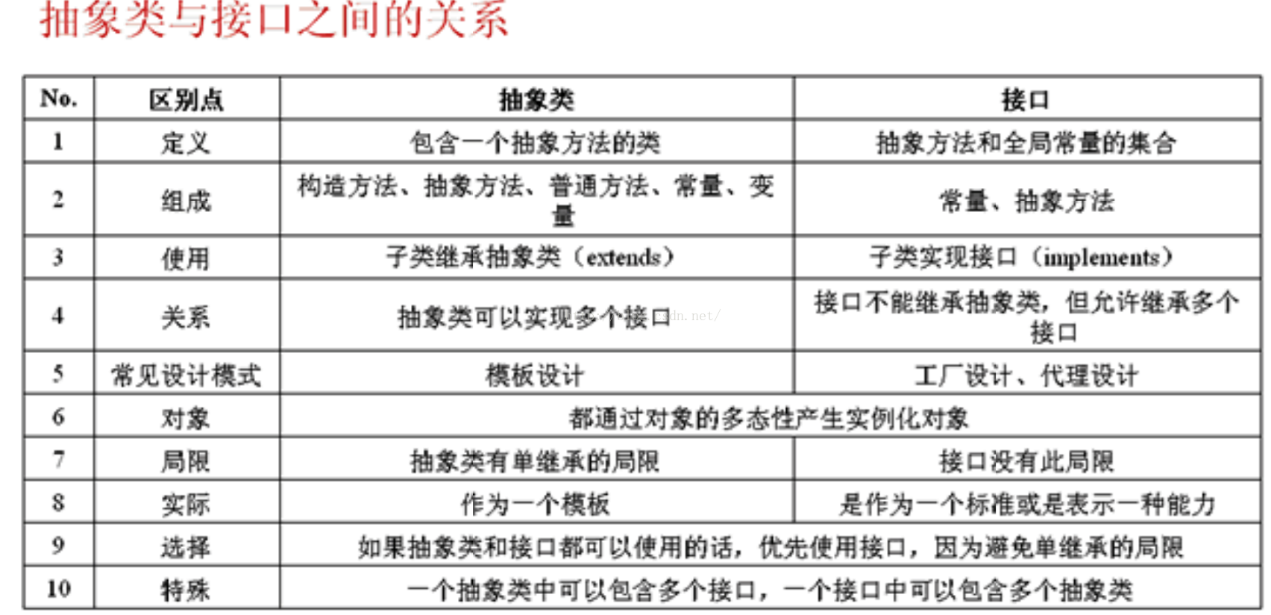
实际上抽象类中也可以包含一个接口。

1. abstract class A{ *// 定义抽象类*
2. public abstract void printA() ; *// 抽象方法*
3. interface B{ *// 定义内部接口*
4. public void printB() ; *// 定义抽象方法*
5. }
6. };
7. class X extends A{ *// 继承抽象类*
8. public void printA(){
9. System.out.println("HELLO --> A") ;
10. }
11. class Y implements B{ *// 定义内部类实现内部接口*
12. public void printB(){
13. System.out.println("HELLO --> B") ;
14. }
15. };
16. };
17. public class InnerExtDemo01{
18. public static void main(String args[]){
19. A.B b = new X().new Y() ;
20. b.printB() ;
21. }
22. };

反之，在一个接口中也可以定义一个抽象类。

1. interface A{ *// 定义接口*
2. public void printA() ; *// 抽象方法*
3. abstract class B{ *// 定义内部抽象类*
4. public abstract void printB() ; *// 定义抽象方法*
5. }
6. };
7. class X implements A{ *// 实现接口*
8. public void printA(){
9. System.out.println("HELLO --> A") ;
10. }
11. class Y extends B{ *// 继承抽象类*
12. public void printB(){
13. System.out.println("HELLO --> B") ;
14. }
15. };
16. };
17. public class InnerExtDemo02{
18. public static void main(String args[]){
19. A.B b = new X().new Y() ;
20. b.printB() ;
21. }
22. };

但是从实际的开发角度上将，此种设计并不常见，因为代码的结构有些混乱了。



重要提示：一个类永远不要去继承一个已经实现好的类，要么继承抽象类，要么实现接口，如果接口和抽象类同时都可以使用的话，那么优先使用接口，可以避免单继承。

总结：

1、抽象类和接口的实例化：通过对象多态性。

2、抽象类表示一个模板，接口制定的是一个标准。

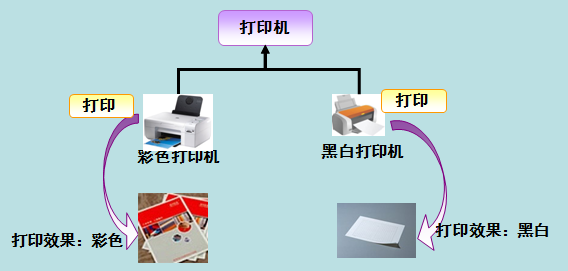
3、常见的设计模式：模板设计，工厂设计、代理设计、适配器设计。

# 2．Java 多态

**多态。 如果说父类中有这个属性跟方法，子类有重写过，那么调用的是子类中的属性跟方法。 如果父类中没有这个属性跟方法，那么子类调用就会出错。 如果父类有这个属性跟方法，而子类没有，则调用的是父类的属性跟方法。**

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(一般继承抽象类，接口)
* 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：

## Test.java 文件代码：

1. 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 a.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) { a.eat(); // 类型判断 if (a instanceof Cat) { // 猫做的事情 Cat c = (Cat)a; c.work(); } 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"); } }
2. 执行以上程序，输出结果为：

吃鱼

抓老鼠

吃骨头

看家

吃鱼

抓老鼠

## 虚函数

虚函数的存在是为了多态。

Java 中其实没有虚函数的概念，它的普通函数就相当于 C++ 的虚函数，动态绑定是Java的默认行为。如果 Java 中不希望某个函数具有虚函数特性，可以加上 final 关键字变成非虚函数。

### 重写

我们将介绍在 Java 中，当设计类时，被重写的方法的行为怎样影响多态性。

我们已经讨论了方法的重写，也就是子类能够重写父类的方法。

当子类对象调用重写的方法时，调用的是子类的方法，而不是父类中被重写的方法。

要想调用父类中被重写的方法，则必须使用关键字 **super**。

## Employee.java 文件代码：

/\* 文件名 : Employee.java \*/ public class Employee { private String name; private String address; private int number; public Employee(String name, String address, int number) { System.out.println("Employee 构造函数"); this.name = name; this.address = address; this.number = number; } public void mailCheck() { System.out.println("邮寄支票给： " + this.name + " " + this.address); } public String toString() { return name + " " + address + " " + number; } public String getName() { return name; } public String getAddress() { return address; } public void setAddress(String newAddress) { address = newAddress; } public int getNumber() { return number; } }

假设下面的类继承Employee类：

## Salary.java 文件代码：

/\* 文件名 : Salary.java \*/ public class Salary extends Employee { private double salary; // 全年工资 public Salary(String name, String address, int number, double salary) { super(name, address, number); setSalary(salary); } public void mailCheck() { System.out.println("Salary 类的 mailCheck 方法 "); System.out.println("邮寄支票给：" + getName() + " ，工资为：" + salary); } public double getSalary() { return salary; } public void setSalary(double newSalary) { if(newSalary >= 0.0) { salary = newSalary; } } public double computePay() { System.out.println("计算工资，付给：" + getName()); return salary/52; } }

现在我们仔细阅读下面的代码，尝试给出它的输出结果：

## VirtualDemo.java 文件代码：

/\* 文件名 : VirtualDemo.java \*/ public class VirtualDemo { public static void main(String [] args) { Salary s = new Salary("员工 A", "北京", 3, 3600.00); Employee e = new Salary("员工 B", "上海", 2, 2400.00); System.out.println("使用 Salary 的引用调用 mailCheck -- "); s.mailCheck(); System.out.println("\n使用 Employee 的引用调用 mailCheck--"); e.mailCheck(); } }

以上实例编译运行结果如下：

Employee 构造函数

Employee 构造函数

使用 Salary 的引用调用 mailCheck --

Salary 类的 mailCheck 方法

邮寄支票给：员工 A ，工资为：3600.0

使用 Employee 的引用调用 mailCheck--

Salary 类的 mailCheck 方法

邮寄支票给：员工 B ，工资为：2400.0

### 例子解析

* 实例中，实例化了两个 Salary 对象：一个使用 Salary 引用 s，另一个使用 Employee 引用 e。
* 当调用 s.mailCheck() 时，编译器在编译时会在 Salary 类中找到 mailCheck()，执行过程 JVM 就调用 Salary 类的 mailCheck()。
* 因为 e 是 Employee 的引用，所以调用 e 的 mailCheck() 方法时，编译器会去 Employee 类查找 mailCheck() 方法 。
* 在编译的时候，编译器使用 Employee 类中的 mailCheck() 方法验证该语句， 但是在运行的时候，Java虚拟机(JVM)调用的是 Salary 类中的 mailCheck() 方法。

以上整个过程被称为虚拟方法调用，该方法被称为虚拟方法。

Java中所有的方法都能以这种方式表现，因此，重写的方法能在运行时调用，不管编译的时候源代码中引用变量是什么数据类型。

## 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)](https://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](https://www.runoob.com/java/java-interfaces.html) 这一章节的内容。

### 方式三：抽象类和抽象方法

详情请看 [Java抽象类](https://www.runoob.com/java/java-abstraction.html) 章节。