

程序员书库

初学者的入门宝典，程序员的百科全书



CD-ROM

10小时多媒体视频讲解

本书特色

- ※ 起点低，即使没有任何编程经验，也能通过本书掌握Java
- ※ 避免大段理论讲解，而是通过大量实例进行讲解，有很强的实践性
- ※ 对代码进行了详细注释，阅读起来非常容易，没有任何障碍
- ※ 通过现实中的事物类比Java中的概念，使读者可以很容易理解
- ※ 重点讲解Java语言的基础知识和应用，并对一些设计模式也有所介绍
- ※ 全书提供190个实例和2个综合案例，非常实用

Java

从入门到精通

高宏静 等编著



化学工业出版社

第7章 接口和内部类

与 C++ 不同，Java 并不支持多重继承。多重继承是指一个类可以继承多个类，也就是一个类可以有多个直接父类。Java 的设计者认为这样会使得类的关系过于混乱，所以 Java 并不支持多重继承。取消了多重继承使得 Java 中类的层次更加清晰，但是当需要复杂问题时却显得力不从心，于是 Java 引入了接口来弥补这点不足。

7.1 接口

接口是 Java 提供的一项非常重要的结构。它定义了一系列的抽象方法和常量，形成一个属性集合。接口定义完成后任何类都可以实现接口，而且一个类可以实现多个接口。实现接口的类必须实现接口中定义的抽象方法，具体实现细节由类自己定义。可以说接口定义了类的框架，它实际上是一种完全的抽象类。

7.1.1 接口的定义

了解什么是接口后，现在就来定义接口。接口的定义跟类的定义十分相似，只是使用的关键字不同，类的定义使用的关键字是 `class`，而接口使用的关键字是 `interface`。接口定义的形式如下。

```
修饰符 interface 接口名
{
    //接口内容
    //声明变量
    类型 变量名;
    .....
    //声明方法
    返回值类型 方法名();
    .....
}
```

接口的定义要注意以下几点。

- ❑ 接口的修饰符只能为默认的（无修饰符）或者 `public`。当修饰符为默认时，接口是包可见的，在接口所在的包之外的类不能使用接口。修饰符为 `public` 时，任何类都可以使用该接口。
- ❑ 接口的名字应该符合 Java 对标识符的规定。
- ❑ 接口内可以声明变量，接口内的变量被自动设置为共有的、静态的、最终的字段。
- ❑ 接口定义的方法都为抽象，它们被自动地设置为 `public`。
- ❑ 接口也被保存为 `.java` 文件，文件名与类名相同。

在接口中声明一个变量。

```
int i;
```

它的实际效果如下。

```
public static final i;
```

在接口定义时可以把它明确地定义为 **public static final**，但是因为字段会被自动地设置为这些类型，所以不建议再写出。下面是一个完整的接口的定义。

```
public interface Animal
{
    //接口中的变量
    int BIG_TYPE=5;
    //用接口声明方法，只有方法的声明没有具体实现
    void sleep();
    void eat();
    void breath();
}
```

7.1.2 接口的实现

接口的实现是指具体实现接口的类。接口的声明仅仅给出了抽象方法，相当于事先定义了程序的框架。实现接口的类必须要实现接口中定义的方法。实现接口的形式如下：

```
class 类名 implements 接口 1,接口 2
{
    方法 1(){
        //方法体
    }
    方法 2(){
        //方法体
    }
}
```

由关键字表示实现的接口，多个接口之间用逗号隔开。实现接口需要注意以下几点。

- ❑ 如果实现接口的类不是抽象类，它必须实现接口中定义的所有的的方法。如果该类为抽象类，在它的子类甚至子类的子类中要实现接口中定义的方法。
- ❑ 实现接口的方法时必须使用相同的方法名和参数列表。
- ❑ 实现接口类中的方法必须被声明为 **public**，因为在接口中的方法都被定义为 **public**，根据继承的原则，访问范围只能放大，不能缩小。

下面是以接口实现的，首先接口定义如下。

```
public interface Animal {
    int BIG_TYPE=5;
    void sleep();
    void eat();
    void breath();
}
```

实现接口的类如下。

```
public class Tiger implements Animal
{
```

```

//实现 breath 方法
public void breath() {
    System.out.println("The tiger breath");
}
//实现 eat 方法
public void eat() {
    System.out.println("The tiger eat");
}
//实现 sleep 方法
public void sleep() {
    System.out.println("The tiger sleep");
}
public static void main(String[ ] args) {
    Tiger tiger=new Tiger();
    tiger.breath();
    tiger.eat();
    tiger.sleep();
}
}

```

程序的运行结果如下。

```

The tiger breath
The tiger eat
The tiger sleep

```

接口之间也可以有继承关系。继承接口的接口拥有它的父接口的方法，它还可以定义自己的方法，实现这个子接口的类。要实现所有这些方法，示例如下，使用上面的 **Animal** 动物接口，然后定义一个子接口哺乳动物 **Mammal**。

```

//子接口
public interface Mammal extends Animal
{
    void run();
}

```

如果类要实现 **Mammal**，它必须实现两个接口中的所有方法，下面是重新定义的类 **Tiger**。

```

public class Tiger implements Mammal
{
    //实现 breath 方法
    public void breath() {
        System.out.println("The tiger breath");
    }
    //实现 eat 方法
    public void eat() {
        System.out.println("The tiger eat");
    }
    //实现 sleep 方法
    public void sleep() {
        System.out.println("The tiger sleep");
    }
    //实现 run 方法
    public void run() {
        System.out.println("The tiger run");
    }
    public static void main(String[ ] args) {
        Tiger tiger=new Tiger();
    }
}

```

```

        tiger.breath();
        tiger.eat();
        tiger.sleep();
        tiger.run();
    }
}

```

7.1.3 接口中的应用

跟抽象类一样，接口也不可以实例化，但是可以声明接口类型的变量，它的值必须是实现了该接口的类的对象。例如：

```
Animal tiger= new Tiger();
```

通过 `tiger`，只能调用 `Animal` 中定义的方法 `eat`、`sleep`、`breath`，如果使用语句 `Mammal tiger=new Tiger();`，就可以调用在 `Mammal` 接口中定义的 `run` 方法了。

当然，通过强制类型转换可以调用所有的方法，示例程序如下，首先看两个接口的定义，其中 `Mammal` 接口继承了 `Animal` 接口。

```

interface Animal
{
    int BIG_TYPE=5;
    void sleep();
    void eat();
    void breath();
}
interface Mammal extends Animal
{
    void run();
}

```

然后是定义类 `Tiger` 和 `Fish` 实现这两个接口，`Tiger` 类直接实现 `Mammal` 接口，而 `Fish` 类实现了 `Animal` 接口。

```

class Tiger implements Mammal
{
    String name;
    public Tiger(String nm) {
        name=nm;
    }
    public void breath() {
        System.out.println(name+"The tiger breath");
    }
    public void eat() {
        System.out.println(name+"The tiger eat");
    }
    public void sleep() {
        System.out.println(name+"The tiger sleep");
    }
    public void run() {
        System.out.println(name+"The tiger run");
    }
}
class Fish implements Animal {
    String name;

```

```

    public Fish(String nm){
        name=nm;
    }
    public void breath() {
        System.out.println(name+"用腮呼吸");
    }
    public void eat() {
        System.out.println(name+"在吃水草");
    }
    public void sleep() {
        System.out.println(name+"在睁着眼睛睡觉");
    }
}

```

演示程序如下。注意这3个对象都放在 Animal 类型的变量中，而两个 Tiger 对象调用 run 方法是不同的。

```

public class AnimalDemo
{
    public static void main(String[ ] args) {
        //Animal 接口, Fish 对象
        Animal fish=new Fish("大鲨鱼");
        //Animal 接口, Tiger 对象
        Animal tiger1=new Tiger("东北虎");
        //Mammal 接口, Tiger 对象
        Mammal tiger2=new Tiger("华南虎");
        //使用 fish 调用各种方法
        fish.breath();
        fish.eat();
        fish.sleep();
        //使用 tiger1 调用各种方法
        tiger1.breath();
        tiger1.eat();
        tiger1.sleep();
        //调用 run 方法, 需要进行类型转换
        ((Tiger) tiger1).run();
        //使用 tiger2 调用各种方法
        tiger2.breath();
        tiger2.eat();
        tiger2.sleep();
        tiger2.run();
    }
}

```

程序的运行结果如下。

```

大鲨鱼用腮呼吸
大鲨鱼在吃水草
大鲨鱼在睁着眼睛睡觉
东北虎 The tiger breath
东北虎 The tiger eat
东北虎 The tiger sleep
东北虎 The tiger run
华南虎 The tiger breath
华南虎 The tiger eat
华南虎 The tiger sleep

```

华南虎 The tiger run

这个程序主要是展示通过接口来实现多态的一种方式。接口变量来存放接口实现类的对象，通过它来调用方法的时候，程序会调用“合适”的方法，过程跟继承中讲到的动态绑定很相似。

接口的另一个重要应用是用它来创建常量组，例如要用 `int` 类型的一组数来表示星期，而且这些天都是固定不需要改变的，就可以通过接口来实现。

```
//用接口来存放变量
interface WeekDays
{
    int MONDAY=1;
    int TUESDAY=2;
    int WEDNESDAY=3;
    int THURSDAY=4;
    int FRIDAY=5;
    int SATURDAY=6;
    int SUNDAY=7;
}
```

在实现了这个接口中可以直接使用这些常量，示例如下。

```
class Time implements WeekDays{
    void print(){
        System.out.println("MONDAY="+MONDAY);
        System.out.println("TUESDAY="+TUESDAY);
        System.out.println("WEDNESDAY="+WEDNESDAY);
        System.out.println("THURSDAY="+THURSDAY);
        System.out.println("FRIDAY="+FRIDAY);
        System.out.println("SATURDAY="+SATURDAY);
        System.out.println("SUNDAY="+SUNDAY);
    }
}
```

`Time` 类实现 `WeekDays` 接口，它可以直接使用接口中定义的常量。把一些固定的常量组值放在接口中定义，然后在类中实现该接口，这在编程中是常用的技巧。示例如下。

```
public class WeekDayDemo implements WeekDays
{
    public static void main(String[ ] args) {
        Time t=new Time();
        t.print();
    }
}
```

程序的运行结果如下。

```
MONDAY=1
TUESDAY=2
WEDNESDAY=3
THURSDAY=4
FRIDAY=5
SATURDAY=6
SUNDAY=7
```

7.1.4 抽象类和接口的比较

接口和抽象类是非常相像的，但它们之间是有区别的，主要区别有以下几方面。

- ❑ 一个类可以实现众多个接口，但是只能继承一个抽象类。可以说接口是取消程序语言中的多继承机制的一个衍生物，但它不完全如此。
- ❑ 抽象类可以有非抽象方法，即可以有已经实现的方法，继承它的子类可以对方法进行覆写；而接口中定义的方法必须全部为抽象方法。
- ❑ 在抽象类中定义的方法，它们的修饰符可以是 `public`、`protected`、`private`，也可以是默认值；但是在接口中定义的方法全是 `public` 的。
- ❑ 抽象类可以有构造函数，接口不能。两者都不能实例化，但是都能通过它们来存放子类对象或是实现类的对象。可以说它们都可以实现多态。

7.2 内部类

内部类是定义在其他类内部的类，内部类所在的类称为宿主类。内部类是 Java 提供的一个非常有用的特性，通过内部类的定义，可以把一些相关的类放在一起。由于内部类只能被它的宿主类使用，所以通过内部类的使用可以很好地控制类的可见性。

7.2.1 内部类的定义

首先来看一个简单的内部类的示例。在程序中定义了一个类 `Outer`，在 `Outer` 的内部定义了一个内部类。在 `Outer` 中的方法 `useInner` 中，生成一个内部类对象，通过这个对象，可以调用内部类的方法 `print`，可以发现内部类可以访问它的宿主类的变量。

```
//外部类
class Outer
{
    String out_string="String in Outer";
    void useInner(){
        Inner in=new Inner();
        in.print();
    }
    //内部类
    class Inner
    {
        void print(){
            System.out.println("Inner.print()");
            System.out.println("use\'"+out_string+"\'");
        }
    }
}
public class InnerClassDemo
{
```



```

public static void main(String[] args) {
    //创建一个对象
    Outer out=new Outer();
    //调用该类的内部类定义的方法
    out.useInner();
}

```

程序的运行结果如下。

```

Inner.print()
use 'String in Outer'

```

该程序主要讲解了程序的运行顺序。首先在 `main` 方法中，创建一个外部类对象，然后使用对象实例调用外部类中的 `useInner` 方法。在 `useInner` 方法中，创建了内部类对象，然后使用内部类对象实例调用内部类中的方法。在该内部类方法中，访问了外部类中的变量，从运行结果中可以看出，是可以访问的。

7.2.2 静态内部类和非静态内部类

上一小节的程序演示了一个简单的内部类的使用。内部类分为两大类，静态内部类和非静态内部类。非静态内部类如上面所示的示例，它可以调用其宿主类的所有变量和方法，并且像宿主类的其他非静态成员那样直接引用它们。静态内部类是用 `static` 修饰的类，静态内部类不能直接访问其宿主类的成员，而必须通过对象来访问。下面用一个简单的例子来说明两种内部类的区别。

```

class Outer {
    String out_string = "String in Outer";
    void useInner1() {
        Inner1 in1 = new Inner1();
        in1.print();
    }
    void useInner2() {
        Inner2 in2 = new Inner2();
        in2.print();
    }
    //非静态内部类
    class Inner1
    {
        void print() {
            System.out.println("Inner1.print()");
            //可以直接使用宿主类的变量
            System.out.println("Outer.out_string=" + out_string + "");
        }
    }
    //静态内部类
    static class Inner2
    {
        void print() {
            System.out.println("Inner2.print()");
            //System.out.println("Outer.out_string=" + out_string + "");
            //需要使用对象来方法宿主类的变量
            Outer outer = new Outer();

```

```

        System.out.println("Outer.out_string=" + outer.out_string);
    }
}
}
public class InnerClassDemo2 {
    public static void main(String[] args) {
        Outer out = new Outer();
        out.useInner1();
        out.useInner2();
    }
}
}

```

程序中 `Iner` 为非静态类，`Iner2` 为静态类。由于 `Iner2` 被标记为静态的，所以它不可以直接访问它宿主类的非静态变量，`Iner2` 中 `print` 方法中被注释掉的一句是不合法的，静态内部类必须获得宿主类的一个对象才能使用使用宿主类的变量。程序的运行结果如下。

```

Iner1.print()
Outer.out_string=String in Outer
Iner2.print()
Outer.out_string=String in Outer

```

对于宿主类访问内部类的成员变量，静态内部类的静态变量可以直接用“类名.变量名”来调用，对于静态内部类的非静态变量则要生成它的对象，利用对象来访问。非静态内部类中不包含静态变量，所以必须用非静态内部类的对象来访问。

内部类的对象的创建对于静态内部类和非静态内部类也不相同。下面程序演示了对于内部类对象的创建以及宿主类对内部类成员访问的不同。

```

class Outer
{
    String out_string = "String in Outer";
    void print(){
        //System.out.println(Iner1.this.in_string1);
        //创建内部类对象
        Iner1 in1 = new Iner1();
        //通过对象访问非静态内部类的属性
        System.out.println(in1.in_string1);
        //直接访问静态内部类的属性
        System.out.println(Iner2.in_string2);
    }
    class Iner1 {
        String in_string1="String in Iner1";
        Iner1(){
            System.out.println("Constructor of Iner1");
        }
        class InClass_In_Iner1{
        }
    }
    static class Iner2 {
        static class staticClass_In_Iner2{
        }
        static String in_string2="static String in Iner2";
        Iner2(){
            System.out.println("Constructor of Iner2");
        }
    }
}

```

```

    }
}
public class InnerClassDemo3 {
    public static void main(String[] args) {
        //创建 Outer 类的对象
        Outer outer=new Outer();
        outer.print();
        //使用类的内部类
        Outer.Inner1 in1=new Outer().new Inner1();
        Outer.Inner2 in2=new Outer.Inner2();
        Outer.Inner1.InClass_In_Inner1 inin2=new Outer().new Inner1().new InClass_In_Inner1();
        Outer.Inner1.InClass_In_Inner1 inin1=in1.new InClass_In_Inner1();
        System.out.println(in1);
        System.out.println(in2);
        System.out.println(inin1);
        System.out.println(inin2);
    }
}

```

程序的运行结果如下。

```

Constructor of Inner1
String in Inner1
static String in Inner2
Constructor of Inner1
Constructor of Inner2
Constructor of Inner1
Outer$Inner1@14318bb
Outer$Inner2@ca0b6
Outer$Inner1$InClass_In_Inner1@10b30a7
Outer$Inner1$InClass_In_Inner1@1a758cb

```

在 Outer 的 print 方法中访问内部类的变量。

```

Inner1 in1 = new Inner1();
System.out.println(in1.in_string1);
System.out.println(Inner2.in_string2);

```

注意两种不同的访问方法。在两个类中又分别创建两个内部类，并分别构建对象。

```

Outer.Inner1 in1=new Outer().new Inner1();
Outer.Inner2 in2=new Outer.Inner2();
Outer.Inner1.InClass_In_Inner1 inin2=new Outer().new Inner1().new InClass_In_Inner1();
Outer.Inner1.InClass_In_Inner1 inin1=in1.new InClass_In_Inner1();

```

Inner1 是非静态类，它要获得它的宿主类的对象然后创建它的内部类对象，内部类的内部类则是需要进一步获得内部类的对象来创建。静态内部类直接用内部类名字访问即可。创建获得对象的类之间的层次通过\$来分隔。程序的运行结果如下。

```

Constructor of Inner1
String in Inner1
static String in Inner2
Constructor of Inner1
Constructor of Inner2
Constructor of Inner1
Outer$Inner1@14318bb
Outer$Inner2@ca0b6
Outer$Inner1$InClass_In_Inner1@10b30a7
Outer$Inner1$InClass_In_Inner1@1a758cb

```

7.2.3 局部内部类

内部类不仅可以在类中定义，也可以在方法中定义。变量如果定义在类中，叫做成员变量；如果定义在方法中，叫做局部变量。内部类也是这样的，当定义在方法中时，叫做局部内部类。演示局部内部类的示例如下。

```
public class Class_In_Method
{
    void doit(){
        //方法中定义的类
        class Class_in_method{
            Class_in_method(){
                System.out.println("Constructor of Class_in_method");
            }
        }
        new Class_in_method();
    }
    public static void main(String[ ] args) {
        Class_In_Method cim=new Class_In_Method();
        cim.doit();
    }
}
```

编译上面的程序，会产生两个.class 文件 Class_In_MethodDemo\$1Class_in_method.class、Class_In_MethodDemo.class。程序的运行结果如下。

```
Constructor of Class_in_method
```

在该程序中，首先创建了一个外部类对象，并执行外部类中的 `doit` 方法。在 `doit` 方法中定义了一个局部内部类，这时是不会执行局部内部类的。当使用 `new` 关键词创建局部内部类对象实例时，就会执行局部内部类的无参构造方法，从而出现上面的程序运行结果。

7.2.4 匿名内部类

匿名就是没有名字，匿名内部类就是没有类名的内部类。匿名内部类省去类的名字，直接构造对象，利用它可以很方便地在需要的时候构造一些只是当前需要的对象。比较下面的两个程序，程序 1 如下。

```
//类 Constant
class Constant
{
    int n;
    Constant(int i){
        n=i;
    }
}
//类 ConstantDemo
class ConstantDemo{
    //该方法获得一个 Constant 对象
    Constant getConstant(){
        return new Constant(5);
    }
}
```

```

    }
}
public class NoNameInerClass {
    public static void main(String[ ] args) {
        ConstantDemo cd=new ConstantDemo();
        System.out.println(cd.getConstant().n);;
    }
}

```

程序 2 如下。

```

//匿名内部类的使用
public class NoNameInerClass2 {
    Constant getConstant(){
        return new Constant(5){
            int n=5;
        };
    }
    public static void main(String[ ] args) {
        NoNameInerClass2 nnic=new NoNameInerClass2();
        System.out.println(nnic.getConstant().n);;
    }
}

```

在程序 1 中定义一个 Constant 类，然后在 ConstantDemo 中获得一个该类的对象，过程比较复杂。在程序 2 中实现相同的功能则简单许多。它使用的就是匿名内部类。匿名内部类在界面开发中经常使用到，在后面的学习中，将会重点讲解。

7.3 对象克隆

克隆技术在当今社会已经不新奇了，在 Java 中也有克隆技术，那就是对 Java 中对象的克隆。首先看下面的程序。

```

class Human {
    String name;           //人类名称
    String sex;            //性别
    int age;               //年龄
    String addr;           //地址
    Human(String name,String sex,int age,String addr){
        this.name=name;
        this.sex=sex;
        this.age=age;
        this.addr=addr;
    }
    void work(){           //工作方法
        System.out.println("我在工作");
    }
    void eat(){            //吃饭方法
        System.out.println("我在吃饭");
    }
}

```

假如有两个人他们除名字之外其他的信息都相同，可能读者会想到如下的解决方案。

```
Human zhangsan=new Human("张三","男",23,"北京");
Human lisi=zhangsan;
lisi.name="李四";
```

直接把一个对象赋给另一对象，把名字改掉就可以了，这种做法看似很正确，但是真的如此吗？示例如下。

```
public class CloneDemo1
{
    public static void main(String[] args) {
        Human zhangsan=new Human("张三","男",23,"北京");    //创建一个人类对象
        System.out.println("张三的名字: "+zhangsan.name);    //显示出名字
        //对象的赋值
        Human lisi=zhangsan;    //进行对象赋值
        lisi.name="李四";    //改变一个对象的属性
        //打印出更改之后的结果
        System.out.println("把李四的名字改为李四");
        System.out.println("李四的名字: "+lisi.name);
        System.out.println("张三的名字: "+zhangsan.name);
    }
}
```

程序的运行结果为。

```
张三的名字: 张三
把李四的名字改为李四
李四的名字: 李四
张三的名字: 李四
```

程序的结果并不像所想的那样，在改一个人的名字的时候两个人的名字都被改掉了。仔细考虑下问题出在哪？其实很简单，语句 `Human lisi=zhangsan;` 不过是把 `lisi` 也指向了 `zhangsan` 所指向的对象。`lisi` 和 `zhangsan` 实际是指向同一个对象，当然改编其中一个另一个也会改变。显然上面的做法是行不通的。在 `Object` 中提供了一个方法 `clone`，它的定义如下。

```
protected Object clone() throws CloneNotSupportedException
```

创建并返回此对象的一个副本，它在类中可以使用下面的语句。

```
super.clone()
```

调用该方法，但是得到的是一个 `Object` 对象，需要对它进行类型转换才能得到想要的对象。还有一点需要注意的是，当在类中使用 `clone` 方法时，该类需要实现 `Cloneable` 接口，并把 `clone` 方法定义为 `public`。下面的程序演示了它的使用，对 `Human` 重新进行定义。

```
class Human implements Cloneable{
    String name;    //人类名称
    String sex;    //性别
    int age;    //年龄
    String addr;    //地址
    Human(String name,String sex,int age,String addr){
        this.name=name;
        this.sex=sex;
        this.age=age;
        this.addr=addr;
    }
}
```

```

    }
    void work(){                //工作方法
        System.out.println("我在工作");
    }
    void eat(){                 //吃饭方法
        System.out.println("我在吃饭");
    }
    //对象克隆的方法
    public Object clone() {
        Human h=null;
        try {
            h= (Human)super.clone();
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return h;
    }
}

```

它的主要改变有继承 Cloneable 接口和实现 clone 方法，注意在 clone 方法中使用了下面的语句格式。

```

try{
}
catch{
}

```

这个是关于异常的语句，读者暂且不用管它，执行 clone 方法必须放在 try、catch 语句块中。演示程序如下。

```

public class CloneDemo {
    public static void main(String[] args) {
        Human zhangsan=new Human("张三","男",23,"北京");        //创建人类对象
        Human lisi=(Human) zhangsan.clone();                      //zhangsan.getClone();
        lisi.name="李四";                                          //赋值
        System.out.println("把李四的名字改为李四");
        System.out.println("李四的名字: "+lisi.name);
        System.out.println("张三的名字: "+zhangsan.name);
    }
}

```

程序的运行结果如下。

```

把李四的名字改为李四
李四的名字: 李四
张三的名字: 张三

```

在本程序中要求是满足了，也许读者认为这样就可以了，但是这样是容易出问题的，看下面的演示程序。

```

class Addr{
    String country;                //国家
    String province;              //地区
    String city;                  //城市
    Addr(String country,String province,String city){
        this.country=country;
    }
}

```

```

        this.province=province;
        this.city=city;
    }
}
class Human1 implements Cloneable{
    String name;           //名称
    int age;               //年龄
    Addr addr;             //地址
    Human1(String name,int age,Addr addr){
        this.name=name;
        this.age=age;
        this.addr=addr;
    }
    public Object clone() {           //克隆方法
        Human1 h=null;
        try {
            h=(Human1)super.clone();
        } catch (CloneNotSupportedException e) {

            e.printStackTrace();
        }
        return h;
    }
}
}}
public class TestClone {
    public static void main(String[ ] args) {
        Addr addr=new Addr("中国","北京","朝阳区");           //创建一个地址对象
        Human1 zhangsan=new Human1("zhagnsan",24,addr);         //创建一个张三的人类对象
        Human1 lisi=(Human1)zhangsan.clone();                   //克隆出一个李四来
        System.out.println("张三的地址");                       //显示张三的地址
        System.out.println(zhangsan.addr.country+zhangsan.addr.province+zhangsan.addr.city);
        System.out.println("李四的地址");                       //显示李四的地址
        System.out.println(lisi.addr.country+lisi.addr.province+lisi.addr.city);
        lisi.addr.country="中国";                               //改变李四的地址
        lisi.addr.province="山东";
        lisi.addr.city="青岛";
        System.out.println("修改李四的地址为： 中国山东青岛");
        System.out.println("张三的地址");                       //显示张三的地址
        System.out.println(zhangsan.addr.country+zhangsan.addr.province+zhangsan.addr.city);
        System.out.println("李四的地址");                       //显示李四的地址
        System.out.println(lisi.addr.country+lisi.addr.province+lisi.addr.city);
    }
}

```

程序的运行结果如下。

```

张三的地址
中国北京朝阳区
李四的地址
中国北京朝阳区
修改李四的地址为： 中国山东青岛
张三的地址
中国山东青岛
李四的地址
中国山东青岛

```


程序中只是改变了李四的地址，却发现张三的地址也被改变了。这是怎么回事呢？这就涉及到了浅克隆的问题。

`clone` 是 `Object` 类提供的方法，`Object` 并不知道具体类的实现细节，它只是按照字段进行一个个拷贝。对于可变对象变量，它获得只是对象的地址，拷贝得到的对象中如果有变量指向可变对象，原对象和克隆得到的对象仍然所含的可变对象仍指向同一个对象，所以改变其中一个对象仍会改变另一个。注意虽然 `String` 是作为对象出现的，但是由于 `String` 的不可变性，所以在克隆中 `String` 不会产生这样的问题。修改上面的程序，如下所示。

```
public class TestClone {
    public static void main(String[] args) {
        Addr addr=new Addr("中国","北京","朝阳区");           //创建一个地址对象
        Human1 zhangsan=new Human1("zhagnsan",24,addr);        //创建一个张三的人类对象
        Human1 lisi=(Human1)zhangsan.clone();                   //克隆出一个李四来
        System.out.println("张三的地址");                       //显示张三的地址
        System.out.println(zhangsan.addr.country+zhangsan.addr.province+zhangsan.addr.city);
        System.out.println("李四的地址");                       //显示李四的地址
        System.out.println(lisi.addr.country+lisi.addr.province+lisi.addr.city);
        Addr addr1=new Addr("中国","山东","青岛");
        lisi.addr=addr1;
        System.out.println("修改李四的地址为： 中国山东青岛");
        System.out.println("张三的地址");
        System.out.println(zhangsan.addr.country+zhangsan.addr.province+zhangsan.addr.city);
        System.out.println("李四的地址");
        System.out.println(lisi.addr.country+lisi.addr.province+lisi.addr.city);
    }
}
```

程序的运行结果如下。

```
张三的地址
中国北京朝阳区
李四的地址
中国北京朝阳区
修改李四的地址为： 中国山东青岛
张三的地址
中国北京朝阳区
李四的地址
中国山东青岛
```

因为在程序中重新构造了一个 `Addr` 对象，并把它赋值给 `lisi` 的 `Addr`，所以改变是成功的，得到了想要的结果。但是这样的话在使用的时候不够方便，较好的解决方案是也能够对 `Addr` 类实现 `clone` 方法。完整的程序如下。

```
class Addr implements Cloneable{
    String country;           //表示地址的国家
    String province;          //表示地址的地区
    String city;              //表示地址的城市
    Addr(String country,String province,String city){
        this.country=country;
        this.province=province;
        this.city=city;
    }
    public Object clone() {    //克隆方法
```

```

        Addr addr=null;
        try {
            addr=(Addr)super.clone();
        } catch (CloneNotSupportedException e) {

            e.printStackTrace();
        }
        return addr;
    }
}
class Human1 implements Cloneable{
    String name;                //名称
    int age;                    //年龄
    Addr addr;                  //地址
    Human1(String name,int age,Addr addr){
        this.name=name;
        this.age=age;
        this.addr=addr;
    }
    public Object clone() {
        Human1 h=null;
        try {
            h=(Human1)super.clone();
            h.addr=(Addr) this.addr.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return h;
    }
}
public class TestClone {
    public static void main(String[ ] args) {
        Addr addr=new Addr("中国","北京","朝阳区");                //创建一个地址对象
        Human1 zhangsan=new Human1("zhagnsan",24,addr);              //创建一个张三的人类对象
        Human1 lisi=(Human1)zhangsan.clone();                        //克隆出一个李四来
        System.out.println("张三的地址");                            //显示张三的地址
        System.out.println(zhangsan.addr.country+zhangsan.addr.province+zhangsan.addr.city);
        System.out.println("李四的地址");                            //显示李四的地址
        System.out.println(lisi.addr.country+lisi.addr.province+lisi.addr.city);
        lisi.addr.country="中国";                                    //改变李四的地址
        lisi.addr.province="山东";
        lisi.addr.city="青岛";
        System.out.println("修改李四的地址为： 中国山东青岛");
        System.out.println("张三的地址");
        System.out.println(zhangsan.addr.country+zhangsan.addr.province+zhangsan.addr.city);
        System.out.println("李四的地址");
        System.out.println(lisi.addr.country+lisi.addr.province+lisi.addr.city);
    }
}

```

程序的运行结果如下。

```

张三的地址
中国北京朝阳区
李四的地址
中国北京朝阳区

```

```
修改李四的地址为：中国山东青岛
张三的地址
中国北京朝阳区
李四的地址
中国山东青岛
```

7.4 小结

在本章中主要介绍了接口和内部类，并在最后介绍了对象克隆的相关知识。接口是 Java 实现程序灵活性和多态性的重要手段，读者应该注意区分接口和抽象类直接的不同点和相同点，以加强面向对象思想的理解。

内部类是 Java 提供了一种特殊的语法结构，它包括静态内部类和非静态内部类以及局部内部类。通过本章的学习，读者应该能掌握它们的基本用法，并尝试在合适的场合使用，提高程序开发效率。内部类在事件处理的时候使用特别方便，这将在后面的内容体现。

对象克隆是一种非常有用的操作，所以在 Object 类中就定义了这个方法，不过由于用户创建的对象复杂性，用户应该创建自己的 clone 方法覆盖该方法。