# think in java

# 传递和返回对象

## 12.1 传递句柄

将句柄传递进入一个方法时，指向的仍然是相同的对象。

|  |
| --- |
| package chapter12;  public class PassHandles {  //  static void f(PassHandles h) {  System.*out*.println("h inside f():" + h);  }  public static void main(String[] args) {  PassHandles p = new PassHandles();  System.*out*.println("p inside main():" + p);  *f*(p);  } } |

### 12.1.1 别名问题

“别名”意味着多个句柄都属兔指向同一个对象。

|  |
| --- |
| package chapter12;  public class Alias1 {   int i;   Alias1(int ii) {  i = ii;  }   public static void main(String[] args) {  Alias1 x = new Alias1(7);  // 新建一个Alias1句柄，不是把它分配给由new创建的一个新对象，而是分配给一个现有的句柄。所以句柄x的内容-即对象x指向的地址---被分配给y  Alias1 y = x;  System.*out*.println("x:" + x.i);  System.*out*.println("y:" + y.i);  System.*out*.println("Incrementing x");  x.i++;  System.*out*.println("x:" + x.i);  System.*out*.println("y:" + y.i);    }   }  package chapter12;  public class Alias2 {   int i;   Alias2(int ii) {  i = ii;  }   static void f(Alias2 handle){  handle.i++;  }   public static void main(String[] args) {  Alias2 x = new Alias2(7);  System.*out*.println("x:" + x.i);  System.*out*.println("Calling f(x)");  *f*(x);  // 方法改变了自己的参数---> 外部对象。  System.*out*.println("x:" + x.i);   }  } |

## 12.2 制作本地副本

Java中所有自变量或参数传递都是通过传递句柄进行的。

1. 参数传递过程中会自定产生别名问题
2. 不存在本地对象，只有本地句柄
3. 句柄有自己的作用域，而对象没有
4. 对象的“存在时间”在Java里不是个问题
5. 没有语言上的支持(如常量)可以防止对象被修改(以避免别名的副作用)

### 12.2.1 按值传递

按值传递，存在两种明显区别的见解：

1. Java按值传递任何东西。若将基本数据类型传递进入一个方法，会明确得到基本数据类型的一个副本。但若将一个句柄传递进入方法，得到的是句柄的副本。所以人们认为“一切”都按值传递。当然，这种说法也有一个前提：句柄肯定也会被传递。但Java的设计方案似乎有些超前，允许我们忽略（大多数时候）自己处理的是一个句柄。也就是说，它允许我们将句柄假想成“对象”，因为在发出方法调用时，系统会自动照管两者间的差异
2. Java主要按值传递（无自变量），但对象却是按引用传递的。得到这个结论的前提是句柄只是对象的一个“别名”，所以不考虑传递句柄的问题，而是直接指出“我准备传递对象”。由于将其传递进入一个方法时没有获得对象的一个本地副本，所以对象显然不是按值传递的。Sun公司似乎在某种程度上支持这一见解，因为它“保留但未实现”的关键字之一便是byvalue（按值）。但没人知道那个关键字什么时候可以发挥作用

### 12.2.2 克隆对象

Clone发昂发在基础类Object中定义成“protected”模式。在希望克隆的任何衍生类中，必须将其覆盖为“public”模式。

|  |
| --- |
| package chapter12;  public class Int {   private int i;   public Int(int ii) {  this.i = ii;  }   public void increment(){  i++;  }   public String toString(){  return Integer.*toString*(i);  } }  package chapter12;  import java.util.Enumeration; import java.util.Vector;  public class Cloning {  public static void main(String[] args) {   Vector v = new Vector();  for (int i = 0; i < 10; i++) {  v.addElement(new Int(i));  }  System.*out*.println("v:" + v);   Vector v2 = (Vector) v.clone();  for (Enumeration e = v2.elements(); e.hasMoreElements(); ) {  ((Int) e.nextElement()).increment();  }  System.*out*.println("v:" + v);   */\*\*  \* v:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  \* v:[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  \* Vector的clone方法不能自动尝试克隆Vector内包含的每个对象--由于别名问题  \* 老的Vector和克隆的Vector都包含了相同的对象。我们通常把这种情况叫作“简单复制”或者“浅层复制”  \*/* } } |

### 12.2.3 使类具备克隆能力

实现cloneable接口

### 12.2.4 成功克隆

克隆必须能够访问，所以必须将其设为public

### 12.2.5 Object.clone的效果

根类中的clone()方法负责建立正确的存储容量，并通过“按位复制”将二进制位从原始对象中复制到新对象的存储空间。也就是说，它并不只是预留存储空间以及复制一个对象——实际需要调查出欲复制之对象的准确大小，然后复制那个对象

(1) 实现Cloneable接口  
(2) 覆盖clone()  
(3) 在自己的clone()中调用super.clone()  
(4) 在自己的clone()中捕获异常  
这一系列步骤能达到最理想的效果