Java 类同其他面向对象的编程语言一样，也支持面向对象（OOP）的三个特征：

- 封装（Encapsulation）
- 继承（Inheritance）
- 多态（Polymorphism）

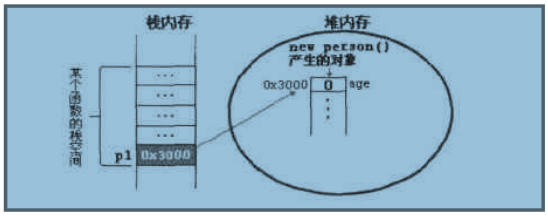
```
class Person
{
    int age;//这是一个成员变量
    void shout()
    {
       int age=60;//这是函数内部又重新定义的一个局部变量
        System.out.println("oh,my god! my age is " + age);
    }
}
```

在这里，shout 方法的 System.out.println("oh,my god! my age is " + age);语句所访问的 age 就不再是成员变量 age，而是在 shout 方法中定义的局部变量 age。

```
Person p1 = new Person();
```

等号左边以类名 Person 作为变量类型定义了一个变量 p1，来指向等号右边通过 new 关键字创建的一个 Person 类的实例对象，变量 p1 就是对象的引用句柄，对象的引用句柄是在栈中分配的一个变量，对象本身是在堆中分配的，原理同第 2 章讲过的数组是一样的。

注意：在 new 语句的类名后一定要跟着一对圆括号()，在本章的稍后部分，读者就会明白这个括号的意义。这条语句执行完后的内存状态如图 3.2 所示。



上面的程序代码，在 TestPerson.main 方法中创建了两个 Person 类的对象，并定义了两个 Person 类的对象引用句柄 p1、p2，分别指向这两个对象。接着，程序调用了 p1 和 p2 的方法和属性，p1、p2 是两个完全独立的对象，类中定义的成员变量，在每个对象都被单独实例化，不会被所有的对象共享，改变了 p1 的 age 属性，不会影响 p2 的 age 属性。调用某个对象的方法时，该方法内部所访问的成员变量，是这个对象自身的成员变量。上面程序运行的内存布局如图 3.3 所示。

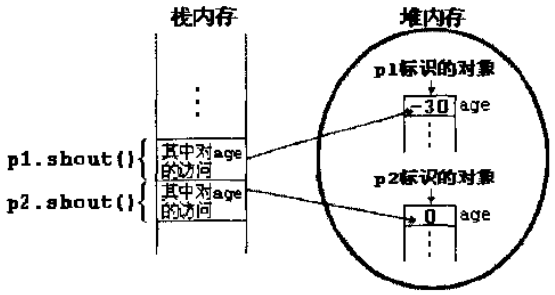


图 3.3

变成垃圾的情况

的几种程序代码，来了解对象变成垃圾的情况。

第一种情况的程序代码：

```
{
Person p1 = new Person();
......
}
```

程序执行完这个代码块后，也就是执行完这对大括号中的所有代码后，产生的 Person 对象就会变成垃圾，因为引用这个对象的句柄 p1 已超过其作用域，p1 已经无效，Person 对象就不再被任何句柄引用了。如图 3.4 所示。

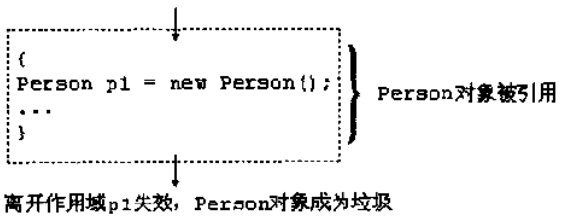


图 3.4

第二种情况的程序代码：

```
{
Person p1 = new Person();
p1 = null;
……
}
```

在执行完 p1 = null;后，即使句柄 p1 还没有超出其作用域，仍然有效，但它已经被赋值为空，也就是说 p1 不再指向任何对象，这个 Person 对象也就成了孤儿，不再被任何句柄引用，变成了垃圾。如图 3.5 所示。

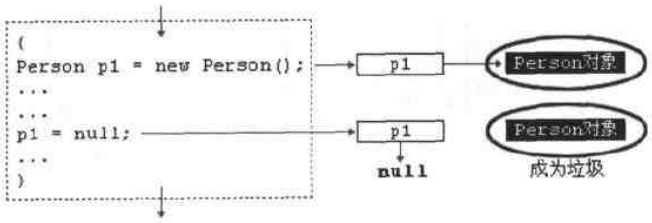


图 3.5

第三种情况的程序代码：

```
{
Person p1 = new Person();
Person p2 = p1;
p1 = null;
……
}
```

执行完 p1 = null;后，产生的 Person 对象不会变成垃圾，因为这个对象仍被 p2 所引用，直到 p2 超出其作用域而无效，产生的 Person 对象才会变成垃圾。如图 3.6 所示。

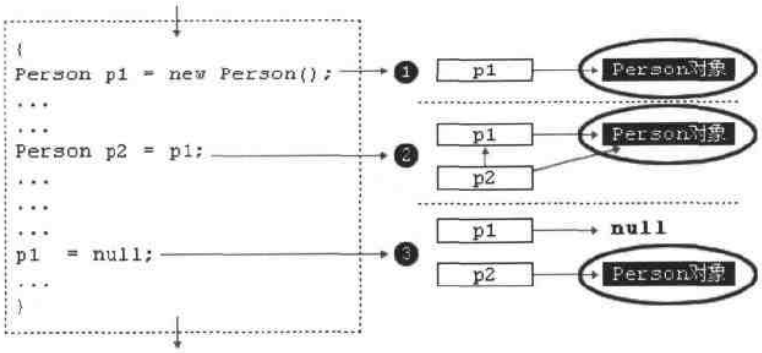


图 3.6

3.2.3 对象的比较

有两种方式可用于对象间的比较，它们是“==”运算符与 equals()方法，“==”操作符用于比较两个变量的值是否相等，equals()方法用于比较两个对象的内容是否一致。我们

```
class Compare
{
   public static void main(String[] args)
    {
       String str1 = new String("abc");
       String str2 = new String("abc");
       String str3 = str1;
       if(str1==str2)
        System.out.println("str1==str2");


           else
            System.out.println("str1!=str2");
           if(str1==str3)
            System.out.println("str1==str3");
           else
            System.out.println("str1!=str3");
       }
    }
```

程序运行结果是：

```
str1!=str2
str1==str3
```

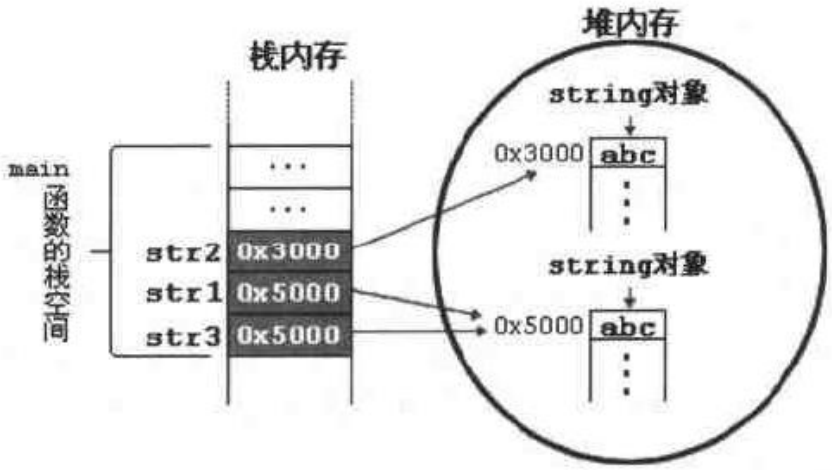


图 3.7

程序运行结果是：

```
str1 equal str2
str1 equal str3
```

多学两招：

同刚才讲的普通对象的比较方式一样，不能用“==”运算符来比较两个数组对象的内容是否相等，但数组对象本身又没有 equals 方法，那怎样比较两个数组对象的内容是否相等呢？如果还记得，在第 2 章中，曾经用过 Arrays.sort 方法对数组进行排序，为什么不去大胆设想 Arrays 里面可能提供了另外的方法，来解决比较两个数组对象的内容是否相等的问题。查看 JDK 文档，如图 3.8 所示。

在上面的帮助信息中，果然发现该类提供了用于比较数组内容的 equals（）方法。

3.3 构造函数

- 它具有与类相同的名称；
- 它不含返回值；
- 它不能在方法中用 return 语句返回一个值。

在一个类中，具有上述特征的方法就是“构造方法”。构造方法在程序设计中非常有用，它可以为类的成员变量进行初始化工作，当一个类的实例对象刚产生时，这个类的构造方法就会被自动调用，我们可以在这个方法中加入要完成初始化工作的代码。这就好像我们规定每个“人”一出生就必须先洗澡，我们就可以在“人”的构造方法中加入完成“洗澡”的程序代码，于是每个“人”一出生就会自动完成“洗澡”，程序就不必再在每个人刚出生时一个一个地告诉他们要“洗澡”了。

在构造方法里不含返回值的概念是不同于“void”的，对于“public void Person()”这样的写法就不再是构造方法，而变成了普通方法，很多人都会犯这样的错误，在定义构造方法时加了“void”，结果这个方法就不再被自动调用了。

构造方法的重载

```
class Person
{
private String name="unknown";
private int age = -1;
public Person()
{
    System.out.println("constructor1 is calling");
}
   public Person(String n)
{
       name = n;
       System.out.println("constructor2 is calling");
       System.out.println("name is"+name);
   }
public Person(String n,int a)
{
       name = n;
       age = a;
       System.out.println("constructor3 is calling");
       System.out.println("name and age is "+name+";"+age);
   }
```

```
    public void shout()
    {
        System.out.println("listen to me!! ");
    }
}
class TestPerson
{
public static void main(String[] args)
{
    Person p1=new Person();
    P1.shout();
    Person p2=new Person("Jack");
    P2.shout();
    Person p3=new Person("Tom",18);
```

这三个对象调用了不同的构造方法，可见，因为括号中传递的参数个数或类型不同，调用的构造方法也不同。

3.3.3 构造方法的一些细节

1. 在 Java 的每个类里都至少有一个构造方法，如果程序员没有在一个类里定义构造方法，系统会自动为这个类产生一个默认的构造方法，这个默认构造方法没有参数，在其方法体中也没有任何代码，即什么也不做。

下面程序的 Construct 类两种写法完全是一样的效果。

```
class Construct
{
}
class Construct
{
public Construct(){}
}
```

对于第一种写法，类虽然没有声明构造方法，但可以用 new Construct()语句来创建 Construct 类的实例对象。

由于系统提供的默认构造方法往往不能满足需求，我们可以自己定义类的构造方法来满足需要，一旦编程者为该类定义了构造方法，系统就不再提供默认的构造方法了。

3.4 this 引用句柄

持有对方的 this 引用

```
class Container
{
    Component comp;
    public void addComponent()
    {
        comp = new Component(this);//将 this 作为对象引用传递
    }
}
class Component
{
    Container myContainer;
    public Component(Container c)
    {
        myContainer = c;
    }
}
```

读者在脑子中，多想想各对象在内存中的状态，就很容易看懂上面的代码。我不希望读者学完这本书后，还要我来提醒："读代码时，不是专盯代码本身，而是要看内存状态"，但在这本书中，还得反复强调，反复地说。下面的图 3.15 是我在脑海中设想的内存状态，供读者参考。

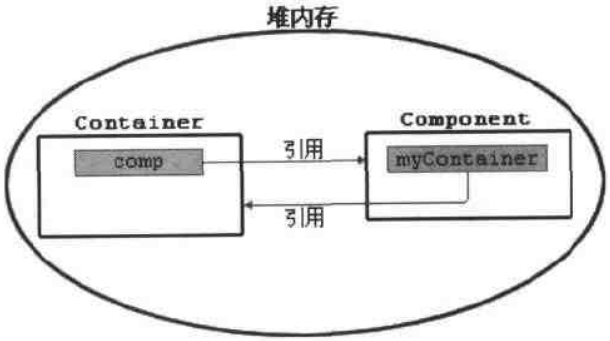


图 3.15

这就是通过 this 引用把当前的对象作为一个参数传递给其他的方法和构造方法的应用。

通过 this 的引用把当前的对象 作为参数传递

3.5 与垃圾回收有关的知识

3.5.1 finalize 方法

finalize()方法是 Object 类的一个方法，任何一个类都从 Object 那继承了这个方法，

finalize()方法的作用就不如 C++中的析构方法那么重要了。finalize()方法是在对象被当成垃圾从内存中释放前调用，而不是在对象变成垃圾前调用，垃圾回收器的启用不由程序员控制，也无规律可循，并不会一产生了垃圾，它就被唤起，甚至有可能到程序终止，它都没有启动的机会。因此这并不是一个很可靠的机制，所以，我们无法保证每个对象的 finalize()方法最终都会被调用。我们只要了解一下 finalize()方法的作用就行了，不要期望 finalize()方法去帮我们做“需要可靠完成”的工作。

调用的时间 是由 java 自己决定 不可靠

3.5.2 System.gc 的作用

Java 的垃圾回收器被执行的偶然性有时候也会给程序运行带来麻烦，比如说在一个对象成为垃圾时需要马上被释放，或者程序在某段时间内产生大量垃圾时，释放垃圾占据的内存空间似乎成了一件棘手的事情，如果垃圾回收器不被启动，finalize()方法也不会被调用。为此，Java 里提供了一个 System.gc()方法，使用这个方法可以强制启动垃圾回收器来回收垃圾，就像我们主动给环卫局打电话，通知他们提前来清扫垃圾的道理是一样的。我

3.6 函数的参数传递

3.6.1 基本数据类型的参数传递

我们看看下面的程序代码：

```
class PassValue
    {
        public static void main(String [] args)
        {
            int x = 5;
            change(x);
            System.out.println(x);
        }
        public static void change(int x)
        {
            x = 3;
        }
    }
```

请问：上面的程序打出的结果是 3 还是 5 呢？我们通过下面的图 3.16 来描述 change 方法被调用的内存状况。

显然，change 方法从开始到结束的过程中并没有改变 main 方法中的 x 的值，所以打印出来的结果应该是 5。可见，基本类型的变量作为实参传递，并不能改变这个变量的值。

3.6.2 引用数据类型的参数传递

```
{
    int x ;
    public static void main(String [] args)
    {
        PassRef obj = new PassRef();
        obj.x = 5;
        change(obj);
        System.out.println(obj.x);
    }
    public static void change(PassRef obj)
    {
        obj.x=3;
    }
}
```


图 3.16

上面的程序打印出来的结果是多少呢？编译运行一下，打印的结果是 3，这是为什么呢？我们通过图 3.17 来看一下发生了什么。

main 方法中的 obj 值没有改变，这和图 3.16 中的过程是一样的，所以还是指向那个对象，但指向对象的内容已在 change 方法中被改变。change 方法中的 obj（这里用 A 标记）就好比 main 方法中的 obj（这里用 B 标记）的别名，对 A 所引用的对象的任何操作就是对 B 所引用的对象的操作。例如有人名叫张小二，他的绰号是“五狗子”，说“五狗子”怎么怎么的，其实就是对张小二说三道四。

3.7 static 关键字

3.7.1 静态变量

注意：我们不能把任何方法体内的变量声明为静态，如下面这样是不行的：

```
fun()
{
  static int i = 0;
}
```

3.7.2 静态方法

我们有时也希望不必创建对象就可以调用某个方法，换句话说也就是使该方法不必和对象绑在一起。要实现这样的效果，只需要在类中定义的方法前加上 static 关键字即可，我们称这种方法为静态成员方法。同静态成员变量一样，可以用类名直接访问静态成员方法，也可以用类的实例对象来访问静态成员方法，还可以在类的非静态的成员方法中像访问其他非静态方法一样去访问这个静态方法，如下面的程序代码：

对于 system。Out。println 的理解

关键字说明类的属性和方法不属于类的某个实例对象，在前面的多个例子程序中反复用到的 System.out.println()语句，其中，System 是一个类名，out 是 System 类的一个静态成员变量，println()方法则是 out 所引用的对象的方法。System.gc()语句中的 gc()也是 System 类的一个静态方法。

在使用类的静态方法时，要注意以下几点：

为什么 main 函数中 调用其他的 一定要先创 建 实 例 。

一个静态方法。

在使用类的静态方法时，要注意以下几点：

（1）在静态方法里只能直接调用同类中其他的静态成员（包括变量和方法），而不能直接访问类中的非静态成员。这是因为，对于非静态的方法和变量，需要先创建类的实例对象后才可使用，而静态方法在使用前不用创建任何对象。

（2）静态方法不能以任何方式引用 this 和 super 关键字（super 关键字在第 4 章讲解）。与上面的道理一样，因为静态方法在使用前不用创建任何实例对象，当静态方法被调用时，this 所引用的对象根本就没有产生。

（3）main()方法是静态的，因此 JVM 在执行 main 方法时不创建 main 方法所在的类的实例对象，因而在 main()方法中，不能直接访问该类中的非静态成员，必须创建该类的一个实例对象后，才能通过这个对象去访问类中的非静态成员，这种情况，在以后的例子中会多次碰到。

3.7.3 静态代码块

一个类中可以使用不包含在任何方法体中的静态代码块（static block），当类被载入时，静态代码块被执行，且只被执行一次，静态块经常用来进行类属性的初始化。如下面的程序代码：

```
{
static String country;
static
{
        country = "china";
        System.out.println("StaticCode is loading"|;
}
}
```

3.7.4 单态设计模式

****

*****

完全不会了解和掌握的人的成功经验。

所谓类的单态设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法。

能存在一个对象实例，并且该类只提供一个取得其对象实例的方法。

如果要让类在一个虚拟机中只能产生一个对象，首先必须将类的构造方法的访问权限设置为 private，这样，就不能用 new 操作符在类的外部产生类的对象了，但在类内部仍可以产生该类的对象。因为在类的外部开始还无法得到类的对象，只能调用该类的某个静态方法以返回类内部创建的对象，静态方法只能访问类中的静态成员变量，所以，指向类内部产生的该类对象的变量也必须定义成静态的。下面是一个单态类的程序例子：

```
public class TestSingle
{
    private static final TestSingle onlyOne=new TestSingle();
    public static TestSingle getTestSingle()
    {
        return onlyOne;
        }
    private TestSingle(){}
}
```

对于上面的程序，我们在外面只能调用 TestSingle.getTestSingle()方法获得 TestSingle 的对象。一些学员告诉我，有些用人单位对于考查应聘者这样的知识点乐此不疲，希望刚才这句话，能对那些想通过学习 Java 编程去找工作的读者有所帮助。

3.8 内 部 类

事务的两种不同叫法，因为思考方式的问题，反而把问题变得复杂了。嵌套类可以直接访问嵌套它的类的成员，包括 private 成员，但是，嵌套类的成员却不能被嵌套它的类直接访问。

非常容易明白作者下面的结论了：当一个类中的程序代码要用到另外一个类的实例对象，而另外一个类中的程序代码又要访问第一个类中的成员，将另外一个类做成第一个类的内部类，程序代码就要容易编写得多，这样的情况在实际应用中非常之多！

如果函数的局部变量（函数的形参也是局部变量），内部类的成员变量，外部类的成员变量重名，我们应该按下面的程序代码所使用的方式来明确指定我们真正要访问的变量。

```
public class Outer
{
   private int size;
   public class Inner
    {
       private int size;
       public void doStuff( int size)
        {
           size++; // 引用的是 doStuff 函数的形参
           this.size++; //引用的是 Inner 类中的成员变量
             Outer.this.size++; // 引用的 Outer 类中的成员变量
     }
   }
}
```

3.8.2 内部类如何被外部引用

内部类也可以通过创建对象从外部类之外被调用，只要将内部类声明为 public 即可，请看下面的程序：

```
class Outer
{
    private int size=10;
    public class Inner
    {
        public void doStuff()
        {
            System.out.println(++size);
        }
    }
}
    public class TestInner
    {
    public static void main( String[] args)
    {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.doStuff();
    }
    }
```

程序中，内部类 Inner 被声明为 public，在外部就可以创建其外部类 Outer 的实例对象，再通过 Outer 类的实例对象创建 Inner 类的实例对象，就可以使用 Inner 类的实例对象来调用内部类 Inner 中的方法了。