Sun microsystems

SCJP认证考试指南

Sun 中国技术社区

http://developers.sun.com.cn

SCJP 认证考试指南

拥有 SCJP，职业开端与众不同

SCJP - Sun Certified Java Programmer (Sun Java 程序员认证)

Sun 公司作为 Java 语言的发明者，对全球的 Java 开发人员进行技术水平认证。该认证在国际上获得了 IT 公司的普遍认可，是目前国际上最热门的 IT 认证之一。《认证杂志》（Certificate Magazine）的权威调查结果表明：

- 持有 SCJP 认证者能够迅速获得面试机会
- 持有 SCJP 认证者的平均资薪比持有其他认证的开发人员高 21.7%
- 持有 SCJP 认证者在公司更容易获得晋升的机会

本复习资料由 J2ME 开发网编译，SUN 中国技术社区免费提供。

http://www.j2medev.com

http://developers.sun.com.cn

第 1 章 声明和访问控制

目标一 创建数组

数组

Java 中的数组跟 C/C++这些语言中的数组的语法结构很相似。但是，Java 去掉了 C/C++中的可以通过[]或者使用指针来访问元素的功能。这种在 C/C++中被普遍接受的功能虽然强大，但是也让 Bug 横行的软件更容易出现。因为 Java 不支持这种直接通过指针来操纵数据，这类的 Bug 也被消除了。

数组是一类包含被称为元素的值的对象。这就为你在程序中移动或保存一组数据以很方便的支持，并且允许你根据需要访问和改变这些值。用一个小例子来说：你可以创建一个 String 类型的数组，每一个都包含一个运动队队员名字。数组可以传送给一个需要访问每个队员名字的方法。如果一个新队员加入，其中一个老队员的名字可以被修改成新队员的名字。这就显得比 player1、player2、player3 等等很随意的不相关的变量方便很多。跟变量通过变量名来访问不同的是，元素通过从 0 开始的数字来访问。因此，你可以一个个的访问数组的每个元素。

数组跟对象很相似，它们都是用 new 关键字来创建，并且有属于主要父对象类的方法。数组可能存储简单类型或者对象的引用。

数组的每个元素必须是同一类型的。元素的类型在数组被声明时确定。如果你需要存储不同类型元素的方式，你可以选择 collection 类，collection 类是 Java2 考试中的新增的考点，我们将会在第十部分讨论它。你可以用数组来存储对象的句柄，你能像使用其它任意对象引用一样访问，摘录或者使用它。

声明但不分配空间

声明一个数组不需分配任何存储空间，它仅仅是代表你试图创建一个数组。跟 C/C++声明一个数组的明显区别就是空间的大小没有被特别标识。因此，下面的声明将会引起一个编译期错误。

```
int num[5];
```

一个数组的大小将在数组使用 new 关键字真正创建时被给定，例如：

```
int num[];
num = new int[5];
```

你可以认为命令 new 的使用跟初始化一个类的实例的使用是类似的。例子中数组名 num 说明数组大小可以是任意大小的整形数据。

同时声明和创建数组

这个例子也可以使用一行语句完成：

```
int num[] = new int[5];
```

方括号也可以放在数据类型后面或者数组名后面。下面的两种都是合法的：

```
int[] num;
int num[];
```

你可以读作：

一个名字为 num 的整型数组

一个数据类型为整型名字为 num 的数组

Java 和 C/C++数组的比较

Java 数组知道它的大小，并且 Java 语言支持对意外的移动到数组末端的保护。

如果你从 Visual Basic 背景下转到 Java 开发，并且还不习惯于一直从 0 开始计数，这点是很方便的。这也可以帮你避免一些在 C/C++程序中很难发现的错误，例如移动到了数组末端并且指向了任意内存地址。

例如，下面的程序会引起一个 ArrayIndexOutOfBoundsException 异常。

```
int[] num= new int[5];
for(int i =0; i<6; i++){
    num[i]=i*2;
    }
```

访问一个 Java 数组的标准习惯用法是使用数组的 length 成员

例如：

```
int[] num= new int[5];
for(int i =0; i<num.length; i++){
        num[i]=i*2;
}
```

数组知道它的大小

假如你跳过了 C/C++的对照，Java 中的数组总是知道它们的大小，这表现在 length 字段。因此，你可以通过下面的语句动态移动数组：

```
int myarray[]=new int[10];
for(int j=0; j<myarray.length;j++){
    myarray[j]=j;
}
```

注意，数组有 length 字段，而不是 length（）方法。当你开始用一组字符串的时候，你会像 s.length()这样使用字符串的 length 方法。

数组中的 length 是域（或者说特性）而不是方法。

Java 数组和 Visual Basic 数组的对照

Java 中的数组总是从 0 开始。如果使用了 Option base 声明，Visual Basic 可能从 1 开始。Java 中没有跟 Visual Basic 中可以使你不删除内容就改变数组大小的 redim preserve 命令等价

的语句。但你可以建立一个同样大小的新数组，并且复制现有元素到里面。

一个数组声明可以有多个方括号。Java 形式上不支持多维数组，但是它可以支持数组的数组，就是我们常说的嵌套数组。

C/C++中那样的多维数组和嵌套数组的最主要区别就是，每个数组不需要有同样的长度。如果你将一个数字当作一个矩阵，矩阵不一定是矩形。按照 Java 语言规范：(http://java.sun.com/docs/books/jls/html/10.doc.html#27805)

“括号里的数指明了数组嵌套的深度”

在其他语言中，就要跟数组的维度相符。因此，你可以建立一个类似于下面的形式的二维数组：

```
int i[][];
```

第一个维度可以匹配 X，第二个维度可以匹配 Y。

声明和初始化相结合

一个数组可以通过一个语句来创建并初始化，这就代替了通过数组循环来初始化的方式。这种方法很适合小数组。下面的语句创建了一个整型数组并且赋值为 0 到 4：

```
int k[]=new int[] {0,1,2,3,4};
```

注意，你没有必要确定数组元素的数量。你可能在测验中被问到下面的语句是不是正确的问题：

```
int k=new int[5] {0,1,2,3,4} //Wrong, will not compile!
```

你可以创建数组的同时确定任何数据类型，因此，你可以创建一个类似于下面形式的字符串数组：

```
String s[]=new String[] {"Zero","One","Two","Three","Four"};
System.out.println(s[0]);
```

这句将会输出 String[0]。

数组的默认值

不同于其他语言中的变量在类级别创建和本地方法级别创建有不同的动作，Java 数组总是被设定为默认值。

无论数组是否被创建了，数组中的元素总是设为默认值。因此，整型的数组总是被置 0，布尔值总是被置 false。下面的代码编译时不会出错，并且输出 0。

```
public class ArrayInit{
    public static void main(String argv[]){
        int[] ai = new int[10];
        System.out.println(ai[0]);
    }
}
```

问题

问题 1）怎样通过一个语句改变数组大小同时保持原值不变？

1) Use the setSize method of the Array class
2) Use Util.setSize(int iNewSize)
3) use the size() operator
4) None of the above

问题 2） 你想用下面的代码查找数组最后一个元素的值，当你编译并运行它的时候，会发生什么？

```
public class MyAr{
public static void main(String argv[]){
        int[] i = new int[5];
        System.out.println(i[5]);
        }
}
```

1) Compilation and output of 0
2) Compilation and output of null
3) Compilation and runtime Exception
4) Compile time error

问题 3）作为一个好的 Java 程序员，你已忘记了曾经在 C/C++中知道的关于数组大小信息的知识。如果你想遍历一个数组并停止在最后一个元素处。你会使用下面的哪一个？

1)myarray.length();
2)myarray.length;
3)myarray.size
4)myarray.size();

问题 4）你的老板为了你写出了 HelloWorld 而很高兴地为你升职了，现在她给你分配了一个新任务，去做一个踢踏舞游戏（或者我小时候玩的曲棍球游戏）。你认为你需要一个多维数组，下面哪一个能做这个工作？

1) int i =new int[3][3];
2) int[] i =new int[3][3];
3) int[][] i =new int[3][3];
4) int i[3][3]=new int[][];

问题 5）

你希望找到一个更优雅的方式给你的数组赋值而不使用 for 循环语句，下面的哪一个能做到？

1)

```
myArray{
    [1]="One";
    [2]="Two";
    [3]="Three";
 }
```

2)String s[5]=new String[] {"Zero","One","Two","Three","Four"};
3)String s[]=new String[] {"Zero","One","Two","Three","Four"};

4)String s[]=new String[]={"Zero","One","Two","Three","Four"};

问题 6）当你试着编译运行下面的代码的时候，可能会发生什么？

```
public class Ardec{
    public static void main(String argv[]){
    Ardec ad = new Ardec();
    ad.amethod();
    }
    public void amethod(){
    int ia1[]= {1,2,3};
    int[] ia2 =  {1,2,3};
    int  ia3[] = new int[] {1,2,3};
    System.out.print(ia3.length);
    }
}
```

1) Compile time error, ia3 is not created correctly
2) Compile time error, arrays do not have a length field
3) Compilation but no output
4) Compilation and output of 3

答案

答案 1）

4) None of the above
你不能改变一个数组的大小。你需要创建一个不同大小的临时数组，然后将原数组中的内容放进去。Java 支持能够改变大小的类的容器，例如 Vector 或者 collection 类的一个成员。

答案 2）

3) Compilation and runtime Exception
当你试着移动到数组的末端的时候，你会得到一个运行时错误。因为数组从 0 开始索引，并且最后一个元素是 i[4]而不是 i[5]。

答案 3）

2) myarray.length;

答案 4）

3) int[][] i=new int[3][3];

答案 5）

3)String s[]=new String[] {"Zero","One","Two","Three","Four"};

答案 6）

4) Compilation and output of 3
所有的数组的声明都是正确的。如果你觉得不太可能，可以自己编译这段代码。

目标二 定义类和变量

定义类，内部类，方法，实例变量，静态变量和自动（本地方法）变量，需要合适的选用允许的修饰词。（例如 public，final，static，abstract 诸如此类）。这些修饰词或者单独使用或者联合使用，定义了包的关系。

关于本目标的解释

我发现目标中用了“诸如此类”，这让我有些烦恼，我想你需要弄明白下面词的意思：
native
transient
synchronized
volatile

什么是类？

一个类的的定义把它很生硬描述为“方法和数据的集合”。它把面向对象编程出来之前的编程思想结合起来，这对理解该概念很有帮助。在类和面向对象程序设计前的主要概念是结构化程序设计。结构化程序设计的理念是程序员将复杂问题划分为小块的代码，一般称为函数或子程序。这符合“做一件很大很复杂的事情的好办法是把它分成一系列比较小但更容易管理的问题”的理念。

尽管结构化程序设计在管理复杂性方面很有用，但它不能容易的解决代码重用问题。程序员发现他们总是“重复发明”轮子。在试着对现实物理对象的思考中，程序设计方面的思想家找到了面向对象的理念（有时被称为 OO）。

举例来说，一个计算机厂商准备生产一种新型个人电脑，如果计算机厂商使用类似于程序设计的方式的话，就要求他建立新团队来设计新 CPU 芯片，新声卡，没准还需要另一个团队设计规划制造新的主板。事实上，这根本不可能出现。由于电脑组件接口的标准化，计算机厂商只需要联系配件供应商，并商议好他们要生产的新型号的说明书就行了。注意组件接口标准化的重要性。

比较 C++/VB 和 Java 的类

因为 Java 被设计成容易让 C++程序员学习的语言，因此两种语言在处理类上有很多相似的地方。C++和 Java 都有继承，多态和数据隐藏特性，并使用显式的修饰词。有一些不同也是因为使 Java 更容易学习和使用。

C++语言实现了多态继承，这样，一个类就可以比一个的父类（或基类）更强大。Java 只允许单继承，这样就只有一个父类。为了克服这个限制，Java 有一个被称作接口的特性。Java 语言的设计者确定接口能够提供多态继承的好处而没有坏处。所有 Java 类都是 Object 类的后代。

对象在 Visual Basic 中是语言设计之后才加入的想法。Visual Basic 有时被称作基于对象的语言而不是面向对象的语言。这就好像是语言的设计者认为类很酷，然后随着 VB4 的发布，他们决定加入一个新类型的模块，称它为类并且加上冒号，让它看起来更像 C++。VB 的类概念中失去了至关重要的元素：继承。微软在 VB5 中加入了跟 Java 的接口很相似的接口的概念。VB 类和 Java 类的最主要相似之处是引用的使用和 new 关键字。

Java 中类的角色

类是 Java 的心脏，所有的 Java 代码都在一个类里。Java 里没有自由独立代码的概念，甚至最简单的 HelloWorld 应用都是包含在类里被创建的。为了指出一个类是另一个类的派生类，我们使用 extend 关键字。如果 extend 关键字没有被使用，这个类将是基类 Object 派生的。这可以使它有一些基本的功能，比如打印自己的名字和其他一些在线程中可能需要用到的功能。

类的最简单特性

定义一个类至少需要 class 关键字，类名和一对花括号。如：

```
class classname {}
```

如果不是有特别作用的类，它在语法上是正确的（我很惊讶的发现，当我举例说明继承时，我定义了一个跟着类似的类）。

通常，一个类还会包括一个访问修饰符，放在关键字 class 前面，还会有程序体放在花括号之间。下面的是一个更好的类模版：

```
public class classname{
    //Class body goes here
}
```

创建一个简单的 HelloWorld 类

这里有一个简单的 HelloWorld 程序，它将会向控制台输出“hello world”。

```
public class HelloWorld{
public static void main(String argv[]){
        System.out.println("Hello world");
        }
}//End class definition
```

关键字 public 是一个可见的修饰符，指明了这个类对于其他类来说都是可见的。一个文件只有一个外部类可以声明为 public。内部类将会隐藏在任意位置。如果你在一个文件中定义了多于一个的 public 类，将会发生一个编译期错误。注意，Java 对每一部分都是很敏感的，包含这个类的文件名字必须是 HelloWorld.java。当然，这跟微软平台虽然保护但是却忽略文件的大小写有些差别。

关键字 class 指明了一个将被定义的类，并且类名是 HelloWorld。左花括号表明类的开

始。注意，类结束的右花括号后面没有分号。注释语句
//End class definition
使用了 C/C++中同样允许的单行类型。Java 也能够识别/**/的注释模式。

创建一个类的实例

上面描述的 HelloWorld 应用例子很浅显的告诉了你所能创建的最简单的应用，但是它漏掉了使用类时至关重要的元素，那就是关键字 new 的使用，new 指出了一个类的新实例的创建。在 HelloWorld 应用中，因为只有 System.out.println 这个唯一的 static 方法，并且不需要类使用 new 关键字创建，因此创建新实例不是必要的。static 方法只能访问 static 变量。可以稍微改进一下 HelloWorld 应用，下面举例说明一个类的新实例的创建。

```
public class HelloWorld2{
        public static void main(String argv[]){
                HelloWorld2 hw = new HelloWorld2();
                hw.amethod();
        }
        public void amethod(){
                System.out.println("Hello world");
        }
}
```

上面的代码通过这行代码创建了自己的一个新实例。

```
HelloWorld2 hw = new HelloWorld2();
```

这是使用类创建新实例的一个基本语法。注意类的名字怎样出现了两次。第一个指明了类的引用的数据类型。这需要它不能和 new 关键字所修饰真正的类的名字相同。这个类实例的名字是 hw。这仅仅是给变量选择的名字。这里有一个命名习惯，一个类的实例名以小写字母开头，而类的名字以大写字母开头。

创建方法

在上一个例子 HelloWorld2 中，一个 Java 中的方法跟 C/C++中的函数和 Visual Basic 中的子程序很相似。上例中名字为 amethod 的方法和本例中的 amethod 方法被声明为 public，这说明它可以在任何地方被访问。它有一个返回值 void，表明没有值返回。并且括号中也是空的，表明它没有参数。

同样的方法可以从下面几种方式之中选择：

```
private void amethod(String s)
private void amethod(int i, String s)
protected void amethod(int i)
```

这些例子说明了一些典型的方法签名。使用关键字 private 和 protected 说明它们将会在别处隐藏。

Java 方法和其他像 C 这样的非面向对象语言的方法的区别是 Java 方法属于类。这表明

它们通过点号指明代码属于哪个类的实例来调用。(static 方法是一个例外，但我们现在无需担心)

因此在 HelloWorld 中 amethod 通过下面的语句调用

```
HelloWorld hw = new HelloWorld();
hw.amethod();
```

在 HelloWorld 类中创建的其他实例中，方法被类的每个实例所调用。每个类的实例将能够访问它自己的变量。因此下面的代码将调用不同实例的 amethod 方法

```
HelloWorld hw = new HelloWorld();
HelloWorld hw2 = new HelloWorld();
hw.amethod();
hw2.amethod();
```

类的两个实例 hw 和 hw2 可能访问不同的变量。

自动局部变量

自动变量是方法变量。它们在方法代码开始运行时生效，并在方法结束时失效。因为它们只能在方法内可见，因此临时操作数据时比较有用。如果你希望一个值在方法被调用时保持，你需要将变量创建在类级别。

一个自动变量将“屏蔽”类级别的变量。

因此，下面的代码将打印 99 而不是 10

```
public class Shad{
public int iShad=10;
public static void main(String argv[]){
                    Shad s = new Shad();
                    s.amethod();
          }//End of main
          public void amethod(){
                    int iShad=99;
                    System.out.println(iShad);
          }//End of amethod
}
```

修饰语和封装

修饰符的可见性是 Java 封装机制的一部分。封装允许分离方法执行的接口。修饰符的可见性是 Java 封装机制至关重要的部分。封装允许分离方法执行的接口。带来的好处就是类内部的代码的细节可以被改变，同时不影响其他对象的使用。这是面向对象设计(最后不得不在某处使用这个词)的一个关键概念。

封装一般用找回或更新 private 类的变量值的方法的形式。这些方法一般是 accessor 或 mutator 方法。访问方法找回值而设置方法改变值。命名惯例是这些方法名类似于 setFOO 改变值，getFOO 得到值。注意，使用 set 和 get 来命名的方法比仅仅使程序员感到方便更重要，并且是 Javabean 系统的重要组成部分。不过我们的测试还没有涉及到 Javabean 的内容。

举一个例子，你有一个变量用来存储学生的年龄。你可能简单的用一个 public 的整型变量来存储。

```
int iAge;
```

接下来，当你的应用程序交付使用后，你可能会发现你的某些学生可能有超过 200 岁的记录，还有小于 0 岁的记录。你需要一段代码来检查错误条件。所以当你的程序改变年龄的值的时候，你用 if 语句来检查范围。

```
if(iAge > 70){
        //do something
        }
if (iAge <3){
        //do something
}
```

当你正在做这些的时候，你漏掉了一些使用过 iAge 变量的代码，所以你被召回了，因为你可能有一个 19 岁的学生，但是你的记录里却是 190 岁。

面向对象使用封装处理了这样的问题，就是创建一个访问包含年龄值的 private 域的方法，名字类似于 setAge 和 getAge。setAge 方法可能有一个整型的参数并且更新年龄的 private 值，getAge 方法没有参数但从 private 的年龄域返回值。

```
public void setAge(int iStudentAge){
        iAge = iStudentAge;
}
public int getAge(){
        return iAge;
}
```

开始，我们也许认为这么长的代码来做一小段代码就能完成的工作没有意义，但是，当这些方法能够满足你的需求时，可以帮你做更多的 iAge 域的确认工作，同时不会影响已经在使用这些信息的代码。

通过这样的代码执行处理方式，实际的程序代码行可以改变，而外面的部分（接口）保持不变。

Private（私有）

私有变量仅仅在创建它的类内部可见。这意味着它们在子类里不可见。这使变量除了当前类之外，绝缘于其他方法的修改。像是修饰语和封装里描述的，这对于将接口与接口实现分离开很有帮助。

```
class Base{
private int iEnc=10;
public void setEnc(int iEncVal){
      if(iEncVal < 1000){
         iEnc=iEncVal;
          }else
          System.out.println("Enc value must be less than 1000");
         //Or Perhaps thow an exception
        }//End if
```

```
}

public class Enc{
public static void main(String argv[]){
                    Base b = new Base();
                    b.setEnc(1001);
          }//End of main
}
```

public（共有）

public 修饰符可以应用于变量（域）或者类。它可能是你学习 Java 过程中最先接触的修饰符。想想 HelloWorld.Java 程序中被这样声明的类的代码

```
public class HelloWorld
```

这是因为 Java 虚拟机仅仅在一个声明为 public 的类中查找神奇的 main 启动方法。

```
public static void main(String argv[])
```

一个 public 类有全局的作用范围，一个实例可以在程序内部或外部的任意位置创建。任何文件中只能有一个非内部类可以用 public 关键字定义。如果你用 public 关键字在一个文件中定义了超过一个非内部类，编译器将会报错。

使用 public 修饰符定义一个变量可以使它在任何位置适用。使用方法如下：

```
public int myint =10;
```

如果你希望创建一个可以在任何地方修改的变量，你可以将它声明为 public。你可以使用类似于调用方法那样的点号来访问它。

```
class Base {
     public int iNoEnc=77;
}
public class NoEnc{
public static void main(String argv[]){
                    Base b = new Base();
                    b.iNoEnc=2;
                    System.out.println(b.iNoEnc);
          }//End of main
}
```

注意，并不建议你对代码的接口和执行不加分隔的使用。如果你想改变 iNoEnc 的数据类型，你必须修改执行改变代码的每一部分。

protected（保护）

protected 有一点古怪。一个 protected 变量在类，子类和同一个包内部可见，但不是全部可见。限制就是它在包内部的可见性可能超过你的预期。在同一路径下的类都是被默认为在一个包内，因此，protected 类将会可见。这就意味着一个 protected 变量会比一个没有任何访问修饰符的变量更有可见性。

一个没有访问修饰符定义的变量称为它有默认的可见性。默认可见性是说一个变量可以

在类内部可见，而包内的其他类中均不可见，不在同一个包的子类内也不可见。

静态的（static）

虽然 static 可以起到可见性修饰符的作用，但它不是直接的可见性修饰符。static 修饰符可以应用于内部类，方法和变量。功能代码经常放在 static 方法中，例如 Math 类有完整的功能方法，如：random，sin 和 round。基本数据类型的包装类 Integer，Double 等等也有 static 方法处理包装过的基本数据类型，如返回符合字符串“2”的 int 值。

标记一个变量为 static 表明每个类只能有一个副本存在。这是与普通的情况相区别。一般情况下，一个类的每个实例都有一个整型变量的副本。在下面的非 static int 例子中，三个实例中的 int iMyVal 都有对应各自实例的不同值。

```
class MyClass{
        public int iMyVal=0;
}

public class NonStat{
public static void main(String argv[]){
        MyClass m1 = new MyClass();
        m1.iMyVal=1;
        MyClass m2 = new MyClass();
        m2.iMyVal=2;
        MyClass m3 = new MyClass();
        m3.iMyVal=99;
        //This will output 1 as each instance of the class
        //has its own copy of the value iMyVal
        System.out.println(m1.iMyVal);
        }//End of main
}
```

下面的例子说明了当你有包含 static 整型数的类的多个实例时会发生什么

```
class MyClass{
        public static   int iMyVal=0;
}
public class Stat{
public static void main(String argv[]){
            MyClass m1 = new MyClass();
            m1.iMyVal=0;
            MyClass m2 = new MyClass();
            m2.iMyVal=1;
            MyClass m3 = new MyClass();
            m2.iMyVal=99;
            //Because iMyVal is static, there is only one
            //copy of it no matter how many instances
            //of the class are created /This code   will
```

```
            //output a value of 99
            System.out.println(m1.iMyVal);
        }//End of main
}
```

你必须要忍受这样的事实，你不能在一个 static 方法内部访问一个非 static 变量。因此，下面的代码会引起一个编译时错误

```
public class St{
int i;
public static void main(String argv[]){
        i = i + 2;//Will cause compile time error
        }
}
```

一个 static 方法不能在一个子类中重写为非 static 方法

一个 static 方法不能在一个子类中重写为非 static 方法。同样，一个非 static（普通的）方法也不能在子类中重写为 static 方法。但是同样的规则对方法重载没有作用。下面的代码在它尝试重写类的方法为非 static 方法 amethod 时将会引起一个错误。

```
class Base{
        public static void amethod(){
        }
}
public class Grimley extends Base{
        public void amethod(){}//Causes a compile time error
}
```

IBM Jikes 编译器会产生下面的错误

```
Found 1 semantic error compiling "Grimley.java":

    6.          public void amethod(){}
                                <------->

*** Error: The instance method "void amethod();"
cannot override the static method "void amethod();"
declared in type "Base"
```

static 方法不能在子类中重写，但是可以被隐藏

在我的模拟测验中，我有一个问题问到 static 方法是否可以被重写，答案是不能，但是引来了大量的 email，很多人举例说明 static 方法被重写了。在子类中，重写过程包括的不仅仅是简单的替代一个方法。它还包括运行时决定哪个方法被调用取决于它的引用类型。

这里有一个例子的代码，看起来显示了一个 static 方法被重写了

```
class Base{
    public static void stamethod(){
    System.out.println("Base");
    }
}
public class ItsOver extends Base{
    public static void main(String argv[]){
```

```
    ItsOver so = new ItsOver();
    so.stamethod();
    }
    public static void stamethod(){
    System.out.println("amethod in StaOver");
    }
}
```

这段代码会被编译并且输出"amethod in StaOver"

本地的（native）

native 修饰符仅仅用来修饰方法，指明代码体不是用 Java 而是用 C 或 C++所写。native 方法经常为平台的特殊目的所写，例如访问某些 Java 虚拟机不支持的硬件。另一个原因是为了需要获得更好的性能。

一个 native 方法以一个分号结尾，而不是代码块。例如下面的代码将会调用一个可能用 C++所写的外部程序:

```
public native void fastcalc();
```

抽象（abstract）

粗略的看一下 abstract 修饰符显得很容易，但是也会漏掉它的一些隐含内容。属于那种主考者很喜欢问一些狡猾的关于那类修饰符的问题。

abstract 修饰符可以被用在类和方法上。当用在方法上时，表明方法会没有方法体（也就是没有花括号的部分），并且代码只能在子类执行时运行。但是，还有一些关于何时何处你能拥有 abstract 方法的限制和包含这类方法的类的规则。如果一个类有一个或多个 abstract 方法，或者继承了不准备运行的 abstract 方法，则它必须声明为 abstract。另外一个情况是，如果一个类实现了接口但是不准备运行接口的每个方法。但这种情况很少见。如果一个类有 abstract 方法，则它需要声明为 abstract 类不要认为一个 abstract 类不能有非 abstract 方法而感到心烦意乱。任何从 abstract 类继承而来的类都要实现基类的 abstract 方法，或者声明自身为 abstract 类。这些规则倾向于问你为什么想要创建 abstract 方法？

abstract 类对于类的设计者很有用。它使类的设计者能够创建应当被实现的方法的原型，但是真正的实现留给以后使用这个类的人。下面的例子是一个包含 abstract 方法的 abstract 类。再次注意，类必须被声明为 abstract，否则会出现编译时错误。

下面的类是 abstract 类，它会被正确编译并打印输出字符串

```
public abstract class abstr{
public static void main(String argv[]){
        System.out.println("hello in the abstract");
        }
        public abstract int amethod();
}
```

常量（final）

final 修饰符可以用在类，方法和变量上。它跟遗传关系的意思很相近，因此很容易记忆。一个 final 类可能从不被继承。另外一种想法是，一个 final 类不能作为父类。任何 final 类中的方法自动成为 final 方法。如果你不希望别的程序员“弄乱你的代码”，这是一个有效的方法。另一个好处就是效率，编译器对于一个 final 方法的工作很少。这些内容在 Core Java 的第一卷中有提及。

final 修饰符表明方法不能被重写。因此，如果你在子类中有一个同样签名的方法的话，你会得到一个编译时错误。

下面的例子说明对一个类使用 final 修饰符。这段代码将会打印字符串"amethod"

```
final class Base{
public void amethod(){
        System.out.println("amethod");
        }
}
public class Fin{
public static void main(String argv[]){
        Base b = new Base();
        b.amethod();
        }
}
```

一个 final 变量的值不能被改变，并且必须在一定的时刻赋值。这跟其他语言中的 constant 的思想比较相似。

同步的（Synchronized）

synchronized 关键字被用来保证不只有一个的线程在同一时刻访问同一个代码块。参看第七部分关于线程的内容来了解更多的关于它的运行的知识。

瞬时（Transient）

transient 修饰符是不常用的修饰符之一。它表明一个变量在序列化过程中不能被写出。

不稳定的（Volatile）

你可能对 volatile 关键字有疑问。最坏的情况就是你确认它真的是一个 Java 关键字。根据 Barry Boone 所说“它告诉编译器一个变量可能在线程异步时被改变”

接受它是 Java 语言的一部分，然后去担心别的吧。