致读者：

我从 2002 年 7 月开始翻译这本书，当时还是第二版。但是翻完前言和介绍部分后，chinapub 就登出广告，说要出版侯捷的译本。于是我中止了翻译，等着侯先生的作品。

我是第一时间买的 这本书，但是我失望了。比起第一版，我终于能看懂这本书了，但是相比我的预期，它还是差一点。所以当 Bruce Eckel 在他的网站上公开本书的第三版的时候，我决定把它翻译出来。

说说容易，做做难。一本 1000 多页的书不是那么容易翻的。期间我也曾打过退堂鼓，但最终还是全部翻译出来了。从今年的两月初起，到 7 月底，我几乎放弃了所有的业余时间，全身心地投入本书的翻译之中。应该说，这项工作的难度超出了我的想像。

首先，读一本书和翻译一本书完全是两码事。英语与中文是两种不同的语言，用英语说得很畅的句子，翻成中文之后就完全破了相。有时我得花好几分钟，用中文重述一句我能用几秒钟读懂的句子。更何况作为读者，一两句话没搞懂，并不影响你理解整本书，但对译者来说，这就不一样了。

其次，这是一本讲英语的人写给讲英语的人的书，所以同很多要照顾非英语读者的技术文档不同，它在用词，句式方面非常随意。英语读者会很欣赏这一点，但是对外国读者来说，这就是负担了。

再有，Bruce Eckel 这样的大牛人，写了 1000 多页，如果都让你读懂，他岂不是太没面子？所以，书里还有一些很有“禅意”的句子。比如那句著名的“The genesis of the computer revolution was in a machine. The genesis of our programming languages thus tends to look like that machine.”我就一直没吃准该怎么翻译。我想大概没人能吃准，说不定 Bruce 要的就是这个效果。

这是一本公认的名著，作者在技术上的造诣无可挑剔。而作为译者，我的编程能力差了很多。再加上上面讲的这些原因，使得我不得不格外的谨慎。当我重读初稿的时候，我发现需要修改的地方实在太多了。因此，我不能现在就公开全部译稿，我只能公开已经修改过的部分。不过这不是最终的版本，我还会继续修订的。

本来，我准备到 10 月份，等我修改完前 7 章之后再公开。但是，我发现我又有点要放弃了，因此我决定给自己一点压力，现在就公开。以后，我将修改完一章就公开一章，请关注 www.wgqqh.com/shhgs/tij.html。

如果你觉得好，请给告诉我，你的鼓励是我工作的动力；如果你觉得不好，那就更应该告诉我了，我会参考你的意见作修改的。我希望能通过这种方法，译出一本配得上原著的书。

shhgs

2003 年 9 月 8 日

4:初始化与清理

随着计算机革命的进步，“不安全”的编程方式已经成为让编程变得如此昂贵的主要原因了。

这类安全性问题包括初始化(*initialization*)和清理(*cleanup*)这两项。C 程序的很多 bug 都是由程序员忘了对变量进行初始化引起的。碰到一些用户不知道该如何初始化，甚至不知道要进行初始化的类库时，情况就更是这样了。清理是一个特殊的问题，因为变量用过之后就没用了，因此会很容易把这一步给忘了。这样程序仍然保留着那些元素所占用的资源，因此资源会很快被耗尽(最常见的就是内存)。

C++引入了构造函数(*constructor*)的概念，这是一种能在对象创建的时候自动调用的方法。Java 也采纳了这种做法，此外对于那些无需再用的对象，它还有一个会自动释放其所占内存的垃圾回收器。本章探讨了初始化和清理问题，并且讲解了 Java 是如何处理这两个问题的。

用构造函数确保初始化

可以这样认为，每个类都有一个名为 **initialize( )**的方法。这个名字就暗示了它得在对象使用之前调用。不幸的是，这么做的话，用户就得记住要调用这个方法。Java 类库的设计者们可以通过提供一种被成为构造函数(*constructor*)的特殊方法，来保证每个对象都能得到初始化。如果类有构造函数，那么 Java 就会在对象刚刚创建，用户还来不及得到的时候，自动调用那个构造函数。这样初始化就有保障了。

接下来的问题是如何命名这个方法。这有两层意思。第一，名字不能与类的成员的名字相冲突。第二，由于构造函数是由编译器调用的，因此编译器必须要能知道该调用哪个方法。既然 C++的解决方案既简单又合理，那么 Java 就沿用了这种做法：构造函数的名字就是类的名字。于是初始化的时候这个方法的自动调用就变得顺理成章了。

下面是一个带构造函数的简单的类：

```
//: c04:SimpleConstructor.java
// Demonstration of a simple constructor.
import com.bruceeckel.simpletest.*;

class Rock {
  Rock() { // This is the constructor
    System.out.println("Creating Rock");
  }
}

public class SimpleConstructor {
  static Test monitor = new Test();
  public static void main(String[] args) {
```

```
    for(int i = 0; i < 10; i++)
      new Rock();
    monitor.expect(new String[] {
      "Creating Rock",
      "Creating Rock",
      "Creating Rock",
      "Creating Rock",
      "Creating Rock",
      "Creating Rock",
      "Creating Rock",
      "Creating Rock",
      "Creating Rock",
      "Creating Rock"
    });
  }
} ///:~
```

现在，创建对象的时候：

```
new Rock();
```

不但要分配内存而且还要调用构造函数。这个过程是有保障的。因此，在你能得到这个对象之前，它就已经被妥善地初始化了。

注意，编程风格要求的方法的首字母要小写的要求不适用与构造函数，因为构造函数的名字必须与类的名字完全相同。

与其他方法一样，构造函数也可以有参数，这样你就能告诉它该如何进行初始化了。前面这段程序经过简单地修改之后，也能让它的构造函数接受参数：

```
//: c04:SimpleConstructor2.java
// Constructors can have arguments.
import com.bruceeckel.simpletest.*;

class Rock2 {
  Rock2(int i) {
    System.out.println("Creating Rock number " + i);
  }
}

public class SimpleConstructor2 {
  static Test monitor = new Test();
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock2(i);
    monitor.expect(new String[] {
      "Creating Rock number 0",
      "Creating Rock number 1",
      "Creating Rock number 2",
      "Creating Rock number 3",
```

```
            "Creating Rock number 4",
            "Creating Rock number 5",
            "Creating Rock number 6",
            "Creating Rock number 7",
            "Creating Rock number 8",
            "Creating Rock number 9"
          });
      }
    } ///:~
```

构造函数的参数是一种能让你对对象的初始化提供参数的方法。比方说，**Tree** 类有一个拿整数作参数的构造函数，而这个整数又表示树的高度，那么你就可以这样来创建一个 **Tree** 对象：

```
Tree t = new Tree(12);  // 12-foot tree
```

如果 **Tree(int)** 是类仅有的构造函数，那么编译器不会让你再用其它方法创建 **Tree** 对象了。

构造函数解决了一大堆问题，提高了代码的可读性。比如，在上述的代码中，根本用不着去调用像 **initialize( )**这种，在概念上同对象的创建相分离的方法。Java 的对象创建和初始化是同一个概念——你不能要这个而不要那个。

构造函数是一种特殊的方法，它没有返回值。这一点同 **void** 型的方法有着本质上的区别，**void** 型的方法什么都不返回，但这是你决定的，你也可以让它返回些什么。而构造函数则什么都不返回，而且你别无选择(**new** 表达式确实会返回这个新创建的对象的 reference，但是构造函数本身不返回任何值)。如果构造函数能有返回值，而且你还可以选择这个值，那么编译器就要问了，它应该怎样处置这个返回值。

方法的重载

名字管理是编程语言的一项重要特性。创建对象的时候，你会给内存空间起名字。而方法是动作的名字。通过使用名字，别人就能更容易地理解和修改程序了。这就像是在写散文——其目的就是要与读者交流。

你用名字来表示对象和方法。不管是对你还是别人，好的名字能让代码读起来更简单。

但是在把人类语言映射到编程语言的时候，会碰到了一件麻烦事。同一个单词经常可以表达很多不同的意思——也就是说这个词被重载(*overload*)了。碰到这些意思只有微不足道的差别的时候，这种情况就更常见了。你会说“洗衬衫”，“洗车”，或者“给狗洗澡”。但是，如果你为了要让

听众不用去区分这些动作的区别，而在那里说“用洗衬衫的方式洗衬衫”，“用洗车的方式洗车”，或者“用给狗洗澡的方式给狗洗澡”，那就太傻了。绝大多数的人类语言都有冗余，所以即使我们省掉几个字，你也能知道他在讲些什么。我们不需要排除所有的两义性——我们可以从上下文作出判断。

绝大多数的编程语言(特别是 C)都要你为每个函数都起一个独一无二的标识符。所以你不能让一个 **print( )**专门打印整数，而让另一个 **print( )** 专门打印浮点数——每个函数都需要一个唯一的名字。

Java(以及 C++)里还有一个让我们不得不对方法进行重载的原因：构造函数。由于构造函数的名字已经由类名决定了，因此构造函数就只能有一个名字。但是假如你要用几种不同的方法来创建对象的话，那又该怎么办呢？假设你要创建这样一个类，它既能以标准的方式进行初始化，又能从文件里读取信息以进行初始化。你得有两个构造函数，一个是不需要参数的(即默认的构造函数(*default constructor*)，[19]也称为无参数(*no-arg*)构造函数)，另一个是需要一个 **String** 作参数的构造函数。而这个 **String** 就是初始化所需的文件的文件名。这两个都是构造函数，所以它们得用同一个名字——类的名字。于是，为了能让相同名字的方法使用不同的参数类型，方法的重载(*method overloading*)就变得非常重要了。重载不仅对构造函数来说是必须，而且也是一种很通用的，能用于任何方法的技巧。

下面这段程序既演示了构造函数的重载，也演示了普通方法的重载：

```
//: c04:Overloading.java
// Demonstration of both constructor
// and ordinary method overloading.
import com.bruceeckel.simpletest.*;
import java.util.*;

class Tree {
  int height;
  Tree() {
    System.out.println("Planting a seedling");
    height = 0;
  }
  Tree(int i) {
    System.out.println("Creating new Tree that is "
      + i + " feet tall");
    height = i;
  }
  void info() {
    System.out.println("Tree is " + height + " feet
tall");
  }
  void info(String s) {
    System.out.println(s + ": Tree is "
      + height + " feet tall");
  }
}
```

```
public class Overloading {
  static Test monitor = new Test();
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Tree t = new Tree(i);
      t.info();
      t.info("overloaded method");
    }
    // Overloaded constructor:
    new Tree();
    monitor.expect(new String[] {
      "Creating new Tree that is 0 feet tall",
      "Tree is 0 feet tall",
      "overloaded method: Tree is 0 feet tall",
      "Creating new Tree that is 1 feet tall",
      "Tree is 1 feet tall",
      "overloaded method: Tree is 1 feet tall",
      "Creating new Tree that is 2 feet tall",
      "Tree is 2 feet tall",
      "overloaded method: Tree is 2 feet tall",
      "Creating new Tree that is 3 feet tall",
      "Tree is 3 feet tall",
      "overloaded method: Tree is 3 feet tall",
      "Creating new Tree that is 4 feet tall",
      "Tree is 4 feet tall",
      "overloaded method: Tree is 4 feet tall",
      "Planting a seedling"
    });
  }
} ///:~
```

创建 **Tree** 对象的时候，既可以不用参数，也可以用一个表示高度的 **int** 当参数。前者表示它还是一棵树苗，后者表示它已经成材了。为了能提供这种功能，程序得有一个默认的构造函数，以及一个需要拿现有高度作参数的构造函数。

或许你还想多要几种调用 **info( )**的方法。比如，想多打印些东西的时候，你可以用 **info(String)**，什么都不想说的时候，可以用 **info( )**。同一种概念要用两个名字，那肯定会很怪。所幸的是，有了方法重载，你就可以让这两者使用同一个名字。

区分经重载的方法

如果方法的名字相同，那么 Java 又该如何判断你要用哪个方法的呢？有一个很简单的规则：每一个经过重载的方法都必须有唯一的参数类型的列表。

只要稍微想一下，你就会发现这是最合情理的办法。如果不是用参数的类型，程序员又怎么来区分两个有着相同名字的方法呢？

即使是参数顺序的颠倒也足以将两个方法区分开来：(不过，通常情况下别用这种办法，因为这会产生难以维护的代码。)

```
//: c04:OverloadingOrder.java
// Overloading based on the order of the arguments.
import com.bruceeckel.simpletest.*;

public class OverloadingOrder {
  static Test monitor = new Test();
  static void print(String s, int i) {
    System.out.println("String: " + s + ", int: " +
i);
  }
  static void print(int i, String s) {
    System.out.println("int: " + i + ", String: " +
s);
  }
  public static void main(String[] args) {
    print("String first", 11);
    print(99, "Int first");
    monitor.expect(new String[] {
      "String: String first, int: 11",
      "int: 99, String: Int first"
    });
  }
} ///:~
```

两个 **print( )**有着相同的参数，但是顺序不同，而这是它们唯一不同的地方。

对 primitive 进行重载

primitive 会自动地从较小的类型升级到较大的类型。如果重载的时候又碰到这个问题，那就有点让人麻烦了。下面这段程序演示了向重载方法传 primitive 的时候，会发生什么事：

```
//: c04:PrimitiveOverloading.java
// Promotion of primitives and overloading.
import com.bruceeckel.simpletest.*;

public class PrimitiveOverloading {
  static Test monitor = new Test();
  void f1(char x) { System.out.println("f1(char)"); }
  void f1(byte x) { System.out.println("f1(byte)"); }
  void f1(short x)
{ System.out.println("f1(short)"); }
  void f1(int x) { System.out.println("f1(int)"); }
  void f1(long x) { System.out.println("f1(long)"); }
  void f1(float x)
{ System.out.println("f1(float)"); }
  void f1(double x)
{ System.out.println("f1(double)"); }

  void f2(byte x) { System.out.println("f2(byte)"); }
```

```
  void f2(short x)
{ System.out.println("f2(short)"); }
  void f2(int x) { System.out.println("f2(int)"); }
  void f2(long x) { System.out.println("f2(long)"); }
  void f2(float x)
{ System.out.println("f2(float)"); }
  void f2(double x)
{ System.out.println("f2(double)"); }

  void f3(short x)
{ System.out.println("f3(short)"); }
  void f3(int x) { System.out.println("f3(int)"); }
  void f3(long x) { System.out.println("f3(long)"); }
  void f3(float x)
{ System.out.println("f3(float)"); }
  void f3(double x)
{ System.out.println("f3(double)"); }

  void f4(int x) { System.out.println("f4(int)"); }
  void f4(long x) { System.out.println("f4(long)"); }
  void f4(float x)
{ System.out.println("f4(float)"); }
  void f4(double x)
{ System.out.println("f4(double)"); }

  void f5(long x) { System.out.println("f5(long)"); }
  void f5(float x)
{ System.out.println("f5(float)"); }
  void f5(double x)
{ System.out.println("f5(double)"); }

  void f6(float x)
{ System.out.println("f6(float)"); }
  void f6(double x)
{ System.out.println("f6(double)"); }

  void f7(double x)
{ System.out.println("f7(double)"); }

  void testConstVal() {
    System.out.println("Testing with 5");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
  }
  void testChar() {
    char x = 'x';
    System.out.println("char argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testByte() {
    byte x = 0;
    System.out.println("byte argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testShort() {
    short x = 0;
    System.out.println("short argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testInt() {
    int x = 0;
    System.out.println("int argument:");
```

```
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testLong() {
    long x = 0;
    System.out.println("long argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testFloat() {
    float x = 0;
    System.out.println("float argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  void testDouble() {
    double x = 0;
    System.out.println("double argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
  }
  public static void main(String[] args) {
    PrimitiveOverloading p =
      new PrimitiveOverloading();
    p.testConstVal();
    p.testChar();
    p.testByte();
    p.testShort();
    p.testInt();
    p.testLong();
    p.testFloat();
    p.testDouble();
    monitor.expect(new String[] {
      "Testing with 5",
      "f1(int)",
      "f2(int)",
      "f3(int)",
      "f4(int)",
      "f5(long)",
      "f6(float)",
      "f7(double)",
      "char argument:",
      "f1(char)",
      "f2(int)",
      "f3(int)",
      "f4(int)",
      "f5(long)",
      "f6(float)",
      "f7(double)",
      "byte argument:",
      "f1(byte)",
      "f2(byte)",
      "f3(short)",
      "f4(int)",
      "f5(long)",
      "f6(float)",
      "f7(double)",
      "short argument:",
      "f1(short)",
      "f2(short)",
      "f3(short)",
      "f4(int)",
      "f5(long)",
      "f6(float)",
      "f7(double)",
```

```
        "int argument:",
        "f1(int)",
        "f2(int)",
        "f3(int)",
        "f4(int)",
        "f5(long)",
        "f6(float)",
        "f7(double)",
        "long argument:",
        "f1(long)",
        "f2(long)",
        "f3(long)",
        "f4(long)",
        "f5(long)",
        "f6(float)",
        "f7(double)",
        "float argument:",
        "f1(float)",
        "f2(float)",
        "f3(float)",
        "f4(float)",
        "f5(float)",
        "f6(float)",
        "f7(double)",
        "double argument:",
        "f1(double)",
        "f2(double)",
        "f3(double)",
        "f4(double)",
        "f5(double)",
        "f6(double)",
        "f7(double)"
      });
  }
} ///:~
```

你会发现，常量 5 会被当作 **int**，所以，如果重载的方法里有拿 **int** 作参数的，它会用这个方法。在其他情况下，如果实参(译者注：真正调用方法的那个参数)的类型比形参(译者注：方法的定义中所声明的参数)的类型小，那么数据会先提升。**char** 的反映会有些不同，因为如果如果找不到匹配的 **char** 的话，它会被提升到 **int**。

如果实参比形参更大，那又会怎样呢？我们修改一下上面那段程序，让它来告诉你答案：

```
//: c04:Demotion.java
// Demotion of primitives and overloading.
import com.bruceeckel.simpletest.*;

public class Demotion {
  static Test monitor = new Test();
  void f1(char x) { System.out.println("f1(char)"); }
  void f1(byte x) { System.out.println("f1(byte)"); }
  void f1(short x)
{ System.out.println("f1(short)"); }
```

```
  void f1(int x) { System.out.println("f1(int)"); }
  void f1(long x) { System.out.println("f1(long)"); }
  void f1(float x)
{ System.out.println("f1(float)"); }
  void f1(double x)
{ System.out.println("f1(double)"); }

  void f2(char x) { System.out.println("f2(char)"); }
  void f2(byte x) { System.out.println("f2(byte)"); }
  void f2(short x)
{ System.out.println("f2(short)"); }
  void f2(int x) { System.out.println("f2(int)"); }
  void f2(long x) { System.out.println("f2(long)"); }
  void f2(float x)
{ System.out.println("f2(float)"); }

  void f3(char x) { System.out.println("f3(char)"); }
  void f3(byte x) { System.out.println("f3(byte)"); }
  void f3(short x)
{ System.out.println("f3(short)"); }
  void f3(int x) { System.out.println("f3(int)"); }
  void f3(long x) { System.out.println("f3(long)"); }

  void f4(char x) { System.out.println("f4(char)"); }
  void f4(byte x) { System.out.println("f4(byte)"); }
  void f4(short x)
{ System.out.println("f4(short)"); }
  void f4(int x) { System.out.println("f4(int)"); }

  void f5(char x) { System.out.println("f5(char)"); }
  void f5(byte x) { System.out.println("f5(byte)"); }
  void f5(short x)
{ System.out.println("f5(short)"); }

  void f6(char x) { System.out.println("f6(char)"); }
  void f6(byte x) { System.out.println("f6(byte)"); }

  void f7(char x) { System.out.println("f7(char)"); }

  void testDouble() {
    double x = 0;
    System.out.println("double argument:");
    f1(x);f2((float)x);f3((long)x);f4((int)x);
    f5((short)x);f6((byte)x);f7((char)x);
  }
  public static void main(String[] args) {
    Demotion p = new Demotion();
    p.testDouble();
    monitor.expect(new String[] {
      "double argument:",
      "f1(double)",
      "f2(float)",
      "f3(long)",
      "f4(int)",
      "f5(short)",
      "f6(byte)",
      "f7(char)"
    });
  }
} ///:~
```

这里，方法需要的是一个更小类型的 primitive 的值。如果你传的参数比它大，那么你就只能把它转换成这个类型的值。把类型的名字放到括号里就行了。不这么做，编译器就会报错。

你应该知道这是一个 *narrowing conversion*，也就是说转换过程中会丢失一些信息。这就是为什么编译器要强制你去这么做的原因了——它要告诉你这是一个 narrowing conversion。

用返回值重载

很自然，我们会问“为什么只有类的名字和方法的参数表呢？为什么不用返回值的类型来区分方法呢？”比如，要区分这样两个有着相同的名字和参数的方法是很容易的：

```
void f() {}
int f() {}
```

在 **int x = f( )**这种情况下，这种做法是行得通的。因为编译器可以根据上下文明白无误地作出判断。但是调用方法的时候是可以忽略返回值的。由于你并不关心方法调用所返回的值，而只想利用其副作用，因此这通常被称为“为利用其副作用而调用方法 (*calling a method for its side effect*)”。所以如果你这样调用方法：

```
f();
```

那么 Java 该怎样判定应该调用哪个 **f( )**呢？读代码的人又该怎样来判断呢？因此，不能用返回值来区分重载的方法。

默认的构造函数

正如我们前面提到过的，默认的构造函数(也就是“无参数”的构造函数)是一种用于创建“基本对象”的无参数的构造函数。如果你写了一个没有构造函数的类，那么编译器会自动为你创建一个默认的构造函数。例如：

```
//: c04:DefaultConstructor.java

class Bird {
  int i;
}

public class DefaultConstructor {
  public static void main(String[] args) {
    Bird nc = new Bird(); // Default!
```

```
  }
} ///:~
```

注意这行

```
new Bird();
```

尽管没有明确定义过，但是它调用了那个默认的构造函数，并且创建了一个新的对象。没有它，我们构建对象的时候就没方法可调。但是，只要定义了构造函数(不管带不带参数)，编译器就不会再自动合成默认的构造函数了：

```
class Hat {
  Hat(int i) {}
  Hat(double d) {}
}
```

现在如果你再说：

```
new Hat();
```

编译器就会报错说它没法找到匹配的构造函数。这就像，如果你不放任何构造函数，编译器会说“你肯定是需要构造函数的，所以我给你弄一个吧。”但是如果你写了一个构造函数，编译器就说“你已经写了一个构造函数，那就说明你知道你在做些什么；既然你不定义默认的构造函数，这就是说明你不需要。”

this 关键词

如果有两个类型相同的对象，**a** 和 **b**，你可能会想该如何调用这两个对象的 **f( )**方法。

```
class Banana { void f(int i) { /* ... */ } }
Banana a = new Banana(), b = new Banana();
a.f(1);
b.f(2);
```

如果只有一个叫 **f( )**的方法，那么它又该怎样判断是对象 **a** 还是 **b** 在调用它呢？

为了能让你用“发消息给对象”，这种简洁的，面向对象的语法来编写程序，编译器暗中做了许多事情。你所操控的那个对象的 reference 会被当作一个非常重要的参数传给方法 **f( )**。因此这两个调用就成了：

```
Banana.f(a,1);
Banana.f(b,2);
```

这是一种内部形式，所以你不能这样写程序，编译器也不会接受，但是它能告诉你究竟发生了些什么。

假设你想在方法里使用当前对象的 reference。由于这个 reference 是由编译器秘密传递的，因此它没有标识符。为此 Java 加了一个关键词：**this**。**this** 关键词只能用于方法内部，它负责返回调用这个方法的对象的 reference。你可以把 **this** 对象的 reference 当作任何对象的 reference。记住，如果你想在别的类的方法里调这个类的方法，那么无须使用 **this**，直接调用就是了。当前对象的 **this** 会自动地用于其它方法，因此你可以这么说：

```
class Apricot {
  void pick() { /* ... */ }
  void pit() { pick(); /* ... */ }
}
```

你可以在 **pit( )**里面写 **this.pick( )**，但是根本没这个必要。[20]编译器会自动地为你作这件事的。**this** 只应用于那些，你明确指明要使用当前对象的 reference 的特殊情况。举例来说，它常被用于 **return** 语句以返回当前对象的 reference。

```
//: c04:Leaf.java
// Simple use of the "this" keyword.
import com.bruceeckel.simpletest.*;

public class Leaf {
  static Test monitor = new Test();
  int i = 0;
  Leaf increment() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Leaf x = new Leaf();
    x.increment().increment().increment().print();
    monitor.expect(new String[] {
      "i = 3"
    });
  }
```

```
} ///:~
```

由于 **increment( )**通过 **this** 关键词返回了当前对象的 reference，因此对同一个对象进行多次操作就变得很方便了。

在构造函数里调用构造函数

当你为同一个类撰写多个构造函数的时候，为了避免重复，经常会在一个构造函数里调用另一个构造函数。你可以通过用 **this** 关键词来进行这种调用。

通常情况下，当你说 **this** 的时候，它表示“这个对象”或“当前对象”的意思， 因此无需更多解释，它就能返回当前对象的 reference。但是在构造函数里，当你传给它一串参数的时候，**this** 关键词就有不同的含义了。它会明确地调用匹配这串参数的构造函数。于是你就有了一种能直截了当地调用其它构造函数的方法了：

```
//: c04:Flower.java
// Calling constructors with "this."
import com.bruceeckel.simpletest.*;

public class Flower {
  static Test monitor = new Test();
  int petalCount = 0;
  String s = new String("null");
  Flower(int petals) {
    petalCount = petals;
    System.out.println(
      "Constructor w/ int arg only, petalCount= "
      + petalCount);
  }
  Flower(String ss) {
    System.out.println(
      "Constructor w/ String arg only, s=" + ss);
    s = ss;
  }
  Flower(String s, int petals) {
    this(petals);
//!    this(s); // Can't call two!
    this.s = s; // Another use of "this"
    System.out.println("String & int args");
  }
  Flower() {
    this("hi", 47);
    System.out.println("default constructor (no
args)");
  }
  void print() {
//! this(11); // Not inside non-constructor!
    System.out.println(
      "petalCount = " + petalCount + " s = "+ s);
  }
  public static void main(String[] args) {
```

```
    Flower x = new Flower();
    x.print();
    monitor.expect(new String[] {
      "Constructor w/ int arg only, petalCount= 47",
      "String & int args",
      "default constructor (no args)",
      "petalCount = 47 s = hi"
    });
  }
} ///:~
```

构造函数 **Flower(String s, int petals)**告诉我们，可以用 **this** 调用一个构造函数，但是不能调用两个。此外，你还得在程序的最前面调用构造函数，否则编译器就会报错。

这段程序还讲了另一种 **this** 的用法。参数 **s** 的名字和类的数据成员 **s** 的名字相同，于是就产生了两义性。你可以用 **this.s** 来解决这个问题。它表示你要用类的数据成员。这种用法在 Java 程序里很常见，而且本书也多次使用到了这种方法。

你可以在 **print( )**里看到，编译器不会让你在方法里调用构造函数，除非它自己就是构造函数。

static 的含义

掌握了 **this** 关键词之后，你就能完全理解 **static** 方法的含义了。它的意思是，这个方法没有 **this**。你不能在 **static** 方法里调用非 **static** 的方法[21] (虽然反过来是可以的)，但是你却可以不通过对象，直接对类调用 **static** 方法。实际上这正是 **static** 方法的本义。这就跟(C 里边的)全局函数一样了。然而 Java 不允许有全局函数，类的 **static** 方法只能访问其它 **static** 方法和 **static** 数据成员。

也有人说，**static** 方法有全局函数的意思，因此它不是面向对象的；你不能用 **static** 方法向对象发消息，因为它没有 **this**。这种评价还是很中肯的。如果你发现用了很多 **static** 方法，那么你大概得重新思考一下你的设计策略了。但是 **static** 很实用，而且有时你会非常需要它，因此它们是不是“真正的 OOP”，应该留给理论家们去讨论。实际上 Smalltalk 的 “类方法(class methods)”做的就是这桩事。

清理：finalization 和垃圾回收

程序员们都知道初始化的重要性，但是他们会常常忘记清理的重要性。毕竟又有谁会去清理一个 **int** 呢？但是对类库来说，简单的“用完就仍”并不总是安全的。当然，Java 有垃圾回收器，它可以回收无需再用的对象所占据的内存。现在，试想一下这样一种特殊情况：假设对象所占据的是

一些“特殊的”，不是用 **new** 分配的内存。垃圾回收器只知道该如何释放用 **new** 分配的内存，所以它不知道该如何释放这些“特殊”的内存。为了处理这种情况，Java 提供了一个叫 **finalize( )**的方法。你可以定义类的这个方法。下面就是我们猜想的它的工作方式。当垃圾回收器准备释放对象所占用的内存的时候，它会先调用 **finalize( )**，只有到下一轮，垃圾回收器才会真正释放这些内存。所以如果你选用 **finalize( )**，那么垃圾回收的时候，你就可以进行一些重要的清理工作了。

这是一个潜在的编程缺陷，因为有些程序员，特别是 C++的程序员，刚开始的时候会误认为 **finalize( )**就是 C++里面的拆构函数(*destructor*)。所谓拆构函数是指对象清理的时候必须调用的函数。但是这里一定要搞清楚 C++和 Java 的区别。(假设程序没有 bug) C++的对象总是会得到清理的(*objects always get destroyed*)，而 Java 对象并不总是会被回收的。换言之：

1.对象不一定会被垃圾回收器回收。

2.垃圾回收不是拆构 (destruction)。

记住这些能帮你省掉很多麻烦。它的意思是，如果对象在被弃用之前还必须进行一些操作的话，你必须亲自来做这些操作。Java 没有拆构函数或类似的概念，因此你只能创建一个普通方法来进行这类清理活动。举例来说，假设对象在创建的过程中把自己画到了屏幕上。只要你没写擦除图像的程序，它就永远也不会被去掉。如果你把擦除图像的功能做在 **finalize( )**里面，那么垃圾回收的时候(谁都不能保证垃圾回收器一定会启动)会先调用 **finalize( )**，于是图像会先从屏幕上消失。但是如果垃圾回收器不启动，那么图像就永远留在上面。

你可能会发现，有个对象的内存一直没被释放。这是因为程序还没有发现内存不够。如果程序运行结束垃圾回收器还没有释放过内存，那么程序退出的时候，这些内存会全部交还给操作系统。这种做法非常好，因为垃圾回收是有开销的，如果它能不启动，那么开销就能低点。

为什么要有 finalize( ) ?

那么，如果 **finalize( )** 不能当通用的清理方法来用，那它又有什么好处呢？

要记住的第三点就是：

3. 垃圾回收只与内存有关。

也就是说，垃圾回收器的唯一目的，就是要将那些不能再为程序所用的内存恢复出来。所以与垃圾回收相关的一切活动，包括你自己写的 **finalize( )**方法，都必须与内存以及内存的释放相关。

是不是说，如果对象还包含了一些别的对象，**finalize( )** 就应该很明确地把那些对象也给释放了呢？哦，不！不论对象是以什么方式创建的，内存释放都是由垃圾回收器负责的。这样看来，只有在一些非常特殊的场合才会需要用到 **finalize( )**，这种场合就是内存是以创建对象以外的形式被分配给对象时候。但是，可能你也注意到了，Java 里的一切都是对象，所以这怎么可能呢？

看来要有 **finalize( )**的原因是，程序员还可以不用 Java 的一般方法，而用一种类似 C 的方法来分配内存。这主要是由 *native methods* 引发的，这是一种在 Java 程序里调用非 Java 程序的方法。(Native methods 在本书第二版的附录 B 中。本书的 CD ROM 里面有它的电子版，此外也可以从 www.BruceEckel.com 下载。) Native method 目前只提供了 C 和 C++的支持，但是它们可以调用别的语言，因此实际上你可以调用任何语言。在非 Java 程序里，你可能会调用 C 的 **malloc( )** 一族的函数，除非你调用 **free( )**，否则它不会释放所分配的内存，于是就会造成内存泄漏。当然，**free( )**是 C 和 C++的函数，因此你得在 **finalize( )**里，用 native method 来调用它。

看到这里，你大概也明白了，要用 **finalize( )**的机会不是太多。[22]对了；这不是通常意义上的清理活动应该发生的地方。那么通常意义上的清理活动应该在哪里进行呢？

你必须亲自进行清理

要想清理(clean up)对象，用户必须在清理的时候调用一个清理方法。这听上去很简单，但却和 C++的拆构函数在概念上有一些冲突。C++的对象都会被清除(destroyed)。或者说，对象都必须被清除。如果 C++的对象是以本地(local)的形式创建的(也就是说创建在栈里——Java 是不可能的做到的)，那么用花括号关闭这个作用域的时候，对象就被清除了。如果对象是用 **new** 创建的(和 Java 一样)，那么当程序员调用 C++的 **delete** 运算符的时候(Java 里面没有)，就会调用对象的拆构函数了。如果 C++的程序员忘了调用 **delete**，那么拆构函数就永远也不会被调用，这样就留下了一个内存的漏洞，此外对象的其余部分也不会得到清理了。这种 bug 是很难察觉的，同时它也是让 C++的程序员转向 Java 的一个重要的原因。

相反，Java 不允许你创建本地对象(local objects)——你必须用 **new**。但是 Java 里面没有释放对象的“delete”可供调用，因为垃圾回收器会替你释放内存。所以从简化问题的角度来讲，你可以认为有了垃圾回收器之后，Java 已经不再需要拆构函数了。然而，随着本书的进展，你会发现垃圾回收器并没有完全取代拆构函数。(再加上你不能直接调用 **finalize( )**，因此在这个问题并没有统一的解决方案。)如果你要进行一些释放内存以外的清理活动，那只能调用 Java 方法了，这就和 C++的拆构函数一样了，一点便宜都占不到。

记住，垃圾回收和 **finalize( )**都是靠不住的。只要 JVM 还没到快要耗尽内存的地步，它是不会浪费时间来回收垃圾以恢复内存的。

中止条件

总之，你不能指望 **finalize( )**，你得特地去创建“清理”方法，并且明明白白地调用这些方法。这样看来，只有在进行后台的内存清理的时候，**finalize( )**才派得上用场，因此绝大多数的程序员根本就用不到它。但是，**finalize( )**还有一种很有趣的用途，这时它就不要求每次都被调用到了。这就是检查对象的中止条件(*termination condition*)。[23]

当你对某个对象不再感兴趣的时候——准备要进行清理的时候——对象应该处于能安全释放其内存的状态。举例来说，如果对象是一个打开的文件，那么在垃圾回收之前，应该先关闭那个文件。如果对象里面还有尚未进行清理的东西，那么程序就潜伏下了一个很难察觉的 bug。这时就能体现 **finalize( )**的价值了。即便不是每次都能被调用，它也可以作为最后一道关口来检测这种情况。只要有一次 finalization 碰上了这个 bug，你就能查到原因，而这才是你真正关心的。

下面举一个简单的例子，告诉你该如何使用这个方法：

```
//: c04:TerminationCondition.java
// Using finalize() to detect an object that
// hasn't been properly cleaned up.
import com.bruceeckel.simpletest.*;

class Book {
  boolean checkedOut = false;
  Book(boolean checkOut) {
    checkedOut = checkOut;
  }
  void checkIn() {
    checkedOut = false;
  }
  public void finalize() {
    if(checkedOut)
      System.out.println("Error: checked out");
  }
}

public class TerminationCondition {
  static Test monitor = new Test();
  public static void main(String[] args) {
    Book novel = new Book(true);
    // Proper cleanup:
    novel.checkIn();
    // Drop the reference, forget to clean up:
    new Book(true);
    // Force garbage collection & finalization:
    System.gc();
    monitor.expect(new String[] {
      "Error: checked out"}, Test.WAIT);
  }
} ///:~
```

中止条件是所有的 **Book** 对象在被当作垃圾回收之前，应该先交还回来(check in)，但是程序员在写 **main( )**的时候犯了个错，有一本书还没还回来。如果不是用 **finalize( )**检查中止条件，要发现这个 bug 会很难。

注意，这里用 **System.gc( )**来强制进行 finalization (你可以在开放过程中用这个办法来加速调试)。但是即使不这么做，经过反复运行，也很有可能会最终被找出那本引起错误的 **Book**(假设程序分配了足够多的内存，于是启动了垃圾回收器)。

垃圾回收器是如何工作的

如果你有过编程经验，知道把对象放在堆里是比较慢的，那么你自然就会认为 Java 把所有的东西都放在堆里的这种做法也一定是很慢的。但事实上垃圾回收器可以极大的提高创建对象的速度。第一次听到这种说法的时候，你可能会觉得有些奇怪——内存的释放怎么会对内存的分配有影响——但这就是 JVM 的工作方式，而且，Java 在堆里分配内存的速度，可以同其他语言在栈里分配内存的速度相媲美。

举例来说，你可以把 C++的堆理解成一个集装箱堆场，这里面的每个对象都要负责管理它自己那块地皮。过一段时间，这块地皮就会被空出来，并且重新投入使用。但是在一些 JVM 里面，Java 的堆就有些不同了；它更像是一条每次分配新对象的时候会不断向前移动的传送带。因此，为对象分配内存会非常快。“堆指针”只是简单地向前面的处女地移，所以这实际上就同 C++的栈的分配是一样的了。(当然，还有一点额外的记帐的开销，但这点开销同搜寻内存空间相比，就不值一提了。)

现在，你可能也发现了，实际上堆不是一条传送带。如果真的这么做的话，就得非常频繁地启动虚拟内存的交换(paging memory，这对性能的影响是很大的)，进而耗尽系统资源。关键就是垃圾回收器。当它回收垃圾的时候，会把堆里面的对象全都压紧，实际上就是把“堆指针”往传送带的开头方向移，让它远离页面断层(page fault)。垃圾回收器会重新安排内存，这样就实现了一种高速的，有无限的堆空间可供分配的内存分配模型。

要理解垃圾回收器(garbage collector 简称 GC)是如何工作的，你就得先了解各种设计方案的原理。*reference counting* 是一种简单的，但却比较慢的的方案。它的意思是，每个对象都要包含一个 reference 计数器，每次有 reference 连到这个对象的时候，这个计数器就会作递增。每次有 reference 离开作用域，或是设成 **null** 的时候，这个计数器就会作递减。这样，管理 reference 计数器就成了程序运行期间的一项不大，但却无法摆脱的负担了。垃圾回收器扫描一遍所有的对象，找到 reference 计数器为零的对象，释放其内存。但是这种方案有个缺点，那

就是如果一堆垃圾对象相互引用并且连成一个环的话，那它们的 reference 计数器就不会是零了。要让垃圾回收器找出这种自我引用的对象，需要花更多的力气。Reference 计数通常被用于解释垃圾回收的技术，但是似乎没有哪种 JVM 采纳了这种设计。

在一些高效的设计方案里，垃圾回收不是基于 reference 计数的。相反，其设计思想是，所有没死的对象都应该能最终找到它在栈，或者在静态存储区里的 reference。这个关系链可以跨好几层对象。这样，如果从栈和静态存储区开始，查遍所有的 reference，你就能找出全部的还活着的对象。找到一个 reference 之后，你还得跟到它所指对象里去，然后跟着对象里面的 reference 找出它所指的对象，以此类推，直到你访遍整个网。你所路过的每个对象应该都是活着的。注意，现在就没有“自我引用的对象组(self-referential groups)”的问题了——它们干脆就找不着了，因此自动就成垃圾了。

JVM 所使用的“自适应的(*adaptive*)”垃圾回收方案就用了这种方法。这种方法有很多变形。找到对象之后，垃圾回收器会根据其具体实现来决定该怎样处理这些对象。其中有一种叫“*stop-and-copy*”的变形。就是说——基于某些很明显的原因——程序必须先停下来(这不是一种后台回收方案)。然后把活着的对象从一个堆里拷贝到另一个堆里，这样原先那个堆就成垃圾了。此外，对象被拷贝到新堆的时候是一个紧挨一个的，于是新堆就被压缩了(并且腾出了新的空间)。

当然，当对象从一个地方移到另一个地方的时候，所有指向它(实际上是 *reference*)的 reference 必须作修改。在堆或静态存储区的 reference 可以马上修改，但是还可能有一些要“逛”才能碰到的 reference 也指向这个对象。这些 reference 会在碰到的时候再作修改(可以设想在新旧地址之间有一张映射表)。

有两个问题使得这种所谓的“拷贝式回收器(copy collector)”的效率不高。第一，你得有两个堆，然后在这两个堆里来回地捣腾，这样你就得维护双倍的内存。有些 JVM 通过为堆分配大块内存，并且直接把东西从一段内存拷贝到另一段内存来解决这个问题。

第二个问题就是拷贝。一旦程序变得稳定了，它就会产生很少的，甚至根本不产生垃圾。但是，拷贝式回收器还是要把一块内存里的东西全部都拷贝到另一块内存里，这样就太浪费了。为了杜绝这种情况的发生，有些 JVM 会在检测到程序不再产生新的垃圾的时候，切换到另一种工作模式下(这就是所谓的“自适应”了)。这“另一种工作模式”就是所谓的“标识和清扫(*mark-and-sweep*)”模式，这是 Sun 早期的 JVM 所使用的方案。在一般情况下，“标识和清扫”是相当慢，但是当你知道程序只产生很少，甚至根本不产生垃圾的时候，它就变得很快了。

“标识和清扫”的思路也是从栈和静态存储出发，跟踪所有的 reference，然后找出活着的对象。但是，每次它找到一个活着的对象的

时候，它会在对象里面设置一个标记，不过它不会马上就回收对象。只有当标记过程完全结束之后，它才会开始清理。清理的时候会释放已死的对象。但是，它不进行拷贝，所以如果垃圾回收器要压缩已经成了碎片的堆的话，它就只能重排对象了。

“停止和拷贝”的意思是这种方式的垃圾回收不是在后台进行；相反，回收垃圾的时候程序必须完全停下来。在 Sun 的文档里，你会发现，很多参考资料都说垃圾回收器是一个低优先级的后台进程，但是看起来它并不是这样实现的，至少早期的 Sun JVM 不是。相反，Sun 的垃圾回收器会在内存不够的时候启动。此外，“标识和清扫”也要求程序先停下来。

正如我们先前提到过的，JVM 会大块大块地分配内存。大对象会得到一块属于它自己的内存。严格意义的“停止和拷贝”要求在释放旧堆之前，先把活着的对象都从旧堆拷贝到新堆里，这样就要占用很多内存。有了块之后，回收的时候就可以往已经废弃的块里拷贝对象了。每个块都有一个记录它是否还活着的“代计数器(*generation count*)”。通常情况下，只有上次垃圾回收之后才创建的块才会被压缩；其它块，如果在什么地方被引用过的话，会在代计数器上表现出来。这样就能应付很多短命的临时对象了。彻底的清理会定期进行——大对象就不用拷贝了(只会在代计数器上做修改)，而保存小对象的块则会被拷贝和压缩。JVM 监视着垃圾回收的效率，如果所有对象都很稳定，垃圾回收成了一种浪费的话，它就会切换到“标识和清扫”模式。同理，JVM 也会追踪“标识和清扫”的成功率，如果堆变得破碎不堪了，它就切换回“停止和拷贝”。这就是所谓的“自适应”了，所以这个方案就是很拗口的“自适应的，分代的，停止和拷贝，标识和清扫。”

JVM 还有很多别的，能提高效率的地方。最重要的，就是那个与装载器有关的，被成为 *just-in-time*(JIT)编译器的东西了。JIT 编译器会部分地，或全部地将程序编译成本机代码，这样就无须 JVM 的解释了，因此能运行得更快。装载类的时候(通常情况下，就是第一次创建那个类的对象的时候)，程序会先找到**.class** 文件，然后将字节代码(bytecode)装进内存。这时就可以直接用 JIT 编译器来编译代码了，但是它有两个缺点：一是要花一点时间，这种地方散布在程序各处，因此会累加起来；此外它还会增加可执行程序的长度(字节代码通常要比扩展后的 JIT 代码小很多)，这样就可能会导致虚拟内存的交换，这对效率的影响就严重了。还有一种所谓的“被动评估(*lazy evaluation*)”的办法，就是不到必要的时候，不作 JIT 编译。最新版 JDK 里面的 Java HotSpot 技术就采用了类似的方法，代码每次运行的时候都作一点优化，这样代码执行得越多，它的速度也就越快。

成员的初始化

为了确保变量在使用之前已经进行了初始化，Java 做得还要彻底。如果变量是在方法内部定义的局部变量，这种保障就表现为编译时的错误信息。所以这种程序：

```
void f() {
  int i;
  i++; // Error -- i not initialized
}
```

会得到一个错误信息，告诉你 **i** 可能还没有初始化。当然编译器原本可以给 **i** 一个缺省值，但这看上去更像是程序员犯的错误，而给了缺省值之后反而会把这个错误给掩盖了。强制程序员提供初始化的值更像是在找 bug。

但是，如果这个 primitive 是类的成员数据，那么情况就有些不同了。由于任何方法都可能初始化或用到这个数据，因此要求用户在使用数据之前就对它进行适当的初始化，就显得不那么现实了。然而把垃圾值留给它也是不安全的，所以类的 primitive 类型的数据都能确保获得一个初始值。这些值就在下面的例程里：

```
//: c04:InitialValues.java
// Shows default initial values.
import com.bruceeckel.simpletest.*;

public class InitialValues {
  static Test monitor = new Test();
  boolean t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  void print(String s) { System.out.println(s); }
  void printInitialValues() {
    print("Data type      Initial value");
    print("boolean        " + t);
    print("char           [" + c + "]");
    print("byte           " + b);
    print("short          " + s);
    print("int            " + i);
    print("long           " + l);
    print("float          " + f);
    print("double         " + d);
  }
  public static void main(String[] args) {
    InitialValues iv = new InitialValues();
    iv.printInitialValues();
    /* You could also say:
    new InitialValues().printInitialValues();
    */
    monitor.expect(new String[] {
```

```
        "Data type      Initial value",
        "boolean        false",
        "char           [" + (char)0 + "]",
        "byte           0",
        "short          0",
        "int            0",
        "long           0",
        "float          0.0",
        "double         0.0"
      });
  }
} ///:~
```

你能看到，尽管没有进行赋值，但它们已经自动地初始化了(**char** 的值是 0，打印出来就是空格)。所以至少你不会有要用未经初始化的值来工作的危险。

以后你会看到，如果在类里定义了一个对象的 reference，但是却没有对它进行初始化，那么那个 reference 会被赋上一个特殊的值 **null** (这也是 Java 的关键词)。

指定初始化

如果要给变量赋上初始化值，那又该怎样做呢？一个简单的方法就是在定义类的变量的时候直接给它赋值。(注意，你不能在 C++里这么做，不过很多 C++的新手就是不信这个邪。)下面是 **InitialValues** 类的成员数据的定义部分，我们作了修改，让它在定义数据的时候提供初始化的值：

```
class InitialValues {
  boolean b = true;
  char c = 'x';
  byte B = 47;
  short s = 0xff;
  int i = 999;
  long l = 1;
  float f = 3.14f;
  double d = 3.14159;
  //. . .
```

你也可以用这个办法对非 primitive 的对象进行初始化。如果 **Depth** 是一个类，那么你可以这样创建变量并且进行初始化：

```
class Measurement {
  Depth d = new Depth();
  // . . .
```

如果你还没给 **d** 初始值就急着用了，那么就会得到一个被称为异常(*excpetion*，第九章讲)的运行时错误。

你甚至可以调用方法来获取初始化的值：

```
class CInit {
  int i = f();
  //...
}
```

当然，这个方法也可以有参数，但是这些参数不能是类的其它尚未初始化的成员。因此你可以这样做：

```
class CInit {
  int i = f();
  int j = g(i);
  //...
}
```

但是不能这样做：

```
class CInit {
  int j = g(i);
  int i = f();
  //...
}
```

对编译器来说，在这个地方对向前引用(forward referencing)表示不满是很合适的，因为这与初始化的顺序有关，而且程序也不是这样编译的。

用这种方法来进行初始化既简单有直接。但是它有局限性，**InitialValues** 类的所有对象都会得到同一个初始化值。有时这正是你想要的，但是更多时候，你需要更大的灵活性。

用构造函数进行初始化

可以用构造函数来进行初始化，这种方法能给编程带来更大的灵活性，因为你可以调用方法，并且在运行时进行操作以确定初始化的值。但是有一件事要记住：你并没有排除自动的初始化，它在构造函数运行之前就已经完成了。所以，举例来说，如果程序是：

```
class Counter {
  int i;
  Counter() { i = 7; }
  // . . .
```

i 先会被初始化为 0，然后才是 7。对于 primitive 和对象的 reference 而言都是这样，甚至是定义的时就给初始值的成员数据也不例外。由此可知，编译器不会强制你只能在构造函数的什么地方进行初始化，或者在使用之间进行初始化——初始化已经有了保障。[24]

初始化的顺序

对类而言，初始化的顺序是由变量在类的定义里面的顺序所决定的。变量的定义可能会分散在类定义的各个地方，并且与方法的定义相互交错，但是变量的初始化会先于任何方法，甚至是构造函数的调用。举个例子：

```
//: c04:OrderOfInitialization.java
// Demonstrates initialization order.
import com.bruceeckel.simpletest.*;

// When the constructor is called to create a
// Tag object, you'll see a message:
class Tag {
  Tag(int marker) {
    System.out.println("Tag(" + marker + ")");
  }
}

class Card {
  Tag t1 = new Tag(1); // Before constructor
  Card() {
    // Indicate we're in the constructor:
    System.out.println("Card()");
    t3 = new Tag(33); // Reinitialize t3
  }
  Tag t2 = new Tag(2); // After constructor
  void f() {
    System.out.println("f()");
  }
  Tag t3 = new Tag(3); // At end
}

public class OrderOfInitialization {
  static Test monitor = new Test();
  public static void main(String[] args) {
    Card t = new Card();
    t.f(); // Shows that construction is done
    monitor.expect(new String[] {
      "Tag(1)",
      "Tag(2)",
      "Tag(3)",
      "Card()",
      "Tag(33)",
      "f()"
    });
  }
} ///:~
```

我们故意把 **Tag** 对象的定义分散到 **Card** 的各个地方，想以此证明初始化会在启动构造函数，或者发生别的什么事之前完成。此外，构造函数还重新对 **t3** 进行了一番初始化。

你可以从程序的输出中看到，**t3** 的 reference 被初始化了两次：一次是在构造函数调用之前。(这前一个对象被扔掉了，因此会被当作垃圾回收。)这种做法的效率好像不怎么样，但是却能保证进行适当的初始化——试想一下，如果你还重载了一个不对 **t3** 进行初始化的构造函数，而定义 **t3** 的时候又没有提供“缺省”的初始化值，那又会怎样呢？

静态数据的初始化

对 **static** 数据来说，故事是相同的：如果它是一个 primitive，而你又没有初始化，那么它会得到一个标准的 primitive 的初始化值。如果它是一个对象的 reference，那么除非你创建一个新的对象并把 reference 连到那个对象，否则它就是 **null**。

如果是在定义的时候初始化，那就和非 **static** 数据没什么两样了。无论创建多少对象，**static** 数据只能有一份。但是当你要对 **static** 数据进行初始化的时候问题就来了。我们还是用程序来把这个问题讲清楚：

```
//: c04:StaticInitialization.java
// Specifying initial values in a class definition.
import com.bruceeckel.simpletest.*;

class Bowl {
  Bowl(int marker) {
    System.out.println("Bowl(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}

class Table {
  static Bowl b1 = new Bowl(1);
  Table() {
    System.out.println("Table()");
    b2.f(1);
  }
  void f2(int marker) {
    System.out.println("f2(" + marker + ")");
  }
  static Bowl b2 = new Bowl(2);
}

class Cupboard {
  Bowl b3 = new Bowl(3);
  static Bowl b4 = new Bowl(4);
  Cupboard() {
    System.out.println("Cupboard()");
    b4.f(2);
  }
  void f3(int marker) {
```

```
    System.out.println("f3(" + marker + ")");
  }
  static Bowl b5 = new Bowl(5);
}

public class StaticInitialization {
  static Test monitor = new Test();
  public static void main(String[] args) {
    System.out.println("Creating new Cupboard() in
main");
    new Cupboard();
    System.out.println("Creating new Cupboard() in
main");
    new Cupboard();
    t2.f2(1);
    t3.f3(1);
    monitor.expect(new String[] {
      "Bowl(1)",
      "Bowl(2)",
      "Table()",
      "f(1)",
      "Bowl(4)",
      "Bowl(5)",
      "Bowl(3)",
      "Cupboard()",
      "f(2)",
      "Creating new Cupboard() in main",
      "Bowl(3)",
      "Cupboard()",
      "f(2)",
      "Creating new Cupboard() in main",
      "Bowl(3)",
      "Cupboard()",
      "f(2)",
      "f2(1)",
      "f3(1)"
    });
  }
  static Table t2 = new Table();
  static Cupboard t3 = new Cupboard();
} ///:~
```

Bowl 能让你看到类的创建，而 **Table** 和 **Cupboard** 还在其类的定义里分散创建了 **static** 的 **Bowl** 类型的成员。注意，**Cupboard** 在定义 **static** 的成员之前，先创建了一个非 **static** 的 **Bowl b3**。

你可以从程序的输出看出，**static** 成员只会在需要的时候进行初始化。如果你没有创建 **Table** 对象，你就永远不可能用到 **Table.b1** 或 **Table.b2**，因此也不会去创建 **static** 的 **Bowl b1** 和 **b2**。只有创建了第一个 **Table** 对象之后(或者第一次访问 **static** 成员的时候)，它们才会被初始化。此后，**static** 对象就不会再作初始化了。

如果先前没有创建过这种对象，因而其 **static** 的成员尚未初始化的话，初始化会先处理其 **static** 成员，再处理非 **static** 的对象。输出已经证实了这一点。

对创建对象的步骤作一个总结是很有帮助的。就用 **Dog** 举例吧：

1. 第一次创建 **Dog** 类的对象(构造函数实际上是 **static** 方法)，或者第一次访问 **Dog** 类的 **static** 的方法或字段的时候，Java 解释器会要搜寻 classpath，找到 **Dog.class**。
2. 装载了 **Dog.class** 之后，(我们以后会学到，创建了 **Class** 对象之后)，会对所有的 **static** 数据进行初始化。这样第一个装载 **Class** 对象的时候，会先进行 **static** 成员的初始化。
3. 用 **new Dog( )**创建新对象的时候，**Dog** 对象的构建进程会先在堆里为对象分配足够的内存。
4. 这块内存先被清零，这样就自动地把 **Dog** 对象的 primitive 类型的成员赋上缺省的值(对于数字就是零，或者是相应的 **boolean** 和 **char**)，将 reference 设成 **null**。
5. 执行定义成员数据时所作的初始化。
6. 执行构造函数。正如你将在第 6 章看到的，这可能会牵涉到相当多的活动，特别是有继承的时候。

显式的静态初始化

Java 能让你用特殊的“**static** 子句”把其它的静态初始化语句全都组织起来(有时被称为 *static* 块(*static block*))。它的形式是这样的：

```
class Spoon {
  static int i;
  static {
    i = 47;
  }
  // . . .
```

它看上去像是一个方法，但实际上只是一段跟在 **static** 关键词后面的代码。同别的 **static** 的初始化一样，这些代码只会执行一次：第一次创建这个类的对象，或是第一次访问这个类的 **static** 的成员的时候(即便你从没创建过那个类的对象)。举例来说：

```
//: c04:ExplicitStatic.java
// Explicit static initialization with the "static"
clause.
import com.bruceeckel.simpletest.*;

class Cup {
  Cup(int marker) {
    System.out.println("Cup(" + marker + ")");
  }
  void f(int marker) {
```

```
    System.out.println("f(" + marker + ")");
  }
}

class Cups {
  static Cup c1;
  static Cup c2;
  static {
    c1 = new Cup(1);
    c2 = new Cup(2);
  }
  Cups() {
    System.out.println("Cups()");
  }
}

public class ExplicitStatic {
  static Test monitor = new Test();
  public static void main(String[] args) {
    System.out.println("Inside main()");
    Cups.c1.f(99);  // (1)
    monitor.expect(new String[] {
      "Inside main()",
      "Cup(1)",
      "Cup(2)",
      "f(99)"
    });
  }
  // static Cups x = new Cups();  // (2)
  // static Cups y = new Cups();  // (2)
} ///:~
```

无论是通过标为(1)的那行程序访问 **static** 的 **c1** 对象，还是把(1)注释掉，让它去运行标为(2)的那行，**Cups** 的 **static** 初始化程序都会得到执行。如果把(1)和(2)同时注释掉，**Cups** 的 **static** 初始化就不会发生了。此外，激活一行还是两行(2)代码，实际上都没关系；静态初始化只运行一次。

非静态的实例初始化

Java 提供了一种类似的语法来初始化对象的非 **static** 的变量。下面就是示例：

```
//: c04:Mugs.java
// Java "Instance Initialization."
import com.bruceeckel.simpletest.*;

class Mug {
  Mug(int marker) {
    System.out.println("Mug(" + marker + ")");
  }
  void f(int marker) {
    System.out.println("f(" + marker + ")");
  }
}
```

```
public class Mugs {
  static Test monitor = new Test();
  Mug c1;
  Mug c2;
  {
    c1 = new Mug(1);
    c2 = new Mug(2);
    System.out.println("c1 & c2 initialized");
  }
  Mugs() {
    System.out.println("Mugs()");
  }
  public static void main(String[] args) {
    System.out.println("Inside main()");
    Mugs x = new Mugs();
    monitor.expect(new String[] {
      "Inside main()",
      "Mug(1)",
      "Mug(2)",
      "c1 & c2 initialized",
      "Mugs()"
    });
  }
} ///:~
```

你可以看到实例初始化(instance initialization)的语句：

```
{
  c1 = new Mug(1);
  c2 = new Mug(2);
  System.out.println("c1 & c2 initialized");
}
```

除了没有 static 关键词，它同静态初始化没什么两样。这种语法为匿名内部类(*anonymous inner class* 见第八章)的初始化提供了必不可少的支持。

数组的初始化

在 C 语言里，初始化数组是一件既常容易出错又冗长乏味的事。C++用所谓的“集合初始化(*aggregate initialization*)”大大增强了这个过程的安全性。[25]由于 Java 的一切都是对象，因而它没有 C++的“集合(aggregates)”。它也有数组，并且提供了数组的初始化。

数组只是一个列在同一个标识符名下的简单序列，这个序列既可以是 primitive 的，也可以是同一种类型的对象的。数组是通过由方括号**[ ]**括起来的数组下标(*indexing operator*)来定义和使用的。要定义一个数组，只要直接在类型的名字后面加上一对空的方括号就行了：

```
int[] a1;
```

你也可以将方括号放到标识符的后面，这样做的意思是相同的：

```
int a1[];
```

这种写法是 C 和 C++程序员们所预想的。但是前一种写法可能更好一些，因为它在说这个类型是“一个 int 型的数组。”本书采纳了这种写法。

编译器不允许你告诉它数组有多大。这让我们又回到了“reference”这个话题。现在你所得到的只是一个数组的 reference，而这个数组的内存还没分配。要为这个数组创建存储空间，你就必须进行初始化。对数组来说，初始化可以在程序的任何地方进行，但是你也可以用一种特殊的初始化语句，让它在创建数组的时候进行初始化。这种特殊的初始化就是用一对花括号把值括起来。在这种情况下，内存分配就会由编译器(和使用 **new** 是等效的)来处理了。例如：

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

那么为什么要在没有数组的情况下，定义一个数组的 reference 呢？

```
int[] a2;
```

此外，Java 还可以将一个数组赋值给另一个，因此你可以说；

```
a2 = a1;
```

你真正在做的是复制 reference，正如这里所演示的：

```
//: c04:Arrays.java
// Arrays of primitives.
import com.bruceeckel.simpletest.*;

public class Arrays {
  static Test monitor = new Test();
  public static void main(String[] args) {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.length; i++)
```

```
      a2[i]++;
    for(int i = 0; i < a1.length; i++)
      System.out.println(
        "a1[" + i + "] = " + a1[i]);
    monitor.expect(new String[] {
      "a1[0] = 2",
      "a1[1] = 3",
      "a1[2] = 4",
      "a1[3] = 5",
      "a1[4] = 6"
    });
  }
} ///:~
```

你可以看到，**a1** 给了一组初始化的值，但是 **a2** 没有；后来又对 **a2** 进行了赋值——在这个例子里，就是另外一个数组。

这里有点新的东西：所有数组 (不论它是 primitive 的还是对象的) 都有一个可供查询的内部成员——但是你不能修改——它会告诉你这个数组有多少元素。这个成员就是 **length**。由于 Java 的数组，同 C 和 C++的一样，是从零开始计算的，因此你能访问的最大的数组下标是 **length - 1**。如果是 C 和 C++的数组，即使你过了界，它也会安安静静地接受这个指令，让后放你到内存里乱窜一气，许多臭名昭著的 bug 就是由此产生的。但是 Java 能让你免受这些问题的困扰。一旦你出了界，就会引发一个运行时错误(异常 *exception*，这是第九章的主题)。当然，每次访问数组的时候都要花时间检查，而且代码也会长些，但是你没法把它关了，也就是说，如果效率非常重要，那么访问数组就可能会成为拖累程序效率的一个因素。但是为了 Internet 的安全和程序员的编程效率，Java 的设计者们认为这个代价还是值得的。

如果写程序的时候还不知道数组要有多少元素，那该怎么办呢？你可以直接用 **new** 来创建数组的元素。这时，甚至能用 **new** 创建 primitive 的数组(**new** 不能用于创建非数组的 primitive)：

```
//: c04:ArrayNew.java
// Creating arrays with new.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class ArrayNew {
  static Test monitor = new Test();
  static Random rand = new Random();
  public static void main(String[] args) {
    int[] a;
    a = new int[rand.nextInt(20)];
    System.out.println("length of a = " + a.length);
    for(int i = 0; i < a.length; i++)
      System.out.println("a[" + i + "] = " + a[i]);
    monitor.expect(new Object[] {
      "%% length of a = \\d+",
```

```
        new TestExpression("%% a\\[\\d+\\] = 0",
a.length)
    });
  }
} ///:~
```

这个例子的 **expect( )**语句里面有些新东西：**TestExpression** 类。**TestExpression** 对象会接受一个表达式，它既可以是普通的表达式，也可以是正则表达式，这里所演示的。此外它还要一个表示这个表达式会重复的次数的 **int**。**TestExpression** 不仅能帮你减少不必要的重复代码，而且能在运行时判断其重复的次数，就像这个例子。

数组的大小是由 **Random.nextInt( )**方法随机选择的，这个方法会返回一个从零到那个参数之间的值。由于它是随机产生的，因此很清楚数组实际上是在运行时创建的。此外，程序的输出也证明了，primitive 类型的数组元素会被初始化为“空”值。(对于数字和 **char** 就是零，对于 **boolean** 就是 **false**。)

当然，数组的定义和初始化也可以合二为一：

```
int[] a = new int[rand.nextInt(20)];
```

如果可以的话，这应该是优先选用的方法。

如果你要处理的不是 primitive 数组，而是对象数组，那你就必须使用 **new** 了。这时，reference 的问题又来了，因为你创建的是一个 reference 的数组。想一下，**Integer** 这个 wrapper 类，它可不是 primitive：

```
//: c04:ArrayClassObj.java
// Creating an array of nonprimitive objects.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class ArrayClassObj {
  static Test monitor = new Test();
  static Random rand = new Random();
  public static void main(String[] args) {
    Integer[] a = new Integer[rand.nextInt(20)];
    System.out.println("length of a = " + a.length);
    for(int i = 0; i < a.length; i++) {
      a[i] = new Integer(rand.nextInt(500));
      System.out.println("a[" + i + "] = " + a[i]);
    }
    monitor.expect(new Object[] {
      "%% length of a = \\d+",
      new TestExpression("%% a\\[\\d+\\] = \\d+",
a.length)
```

```
    });
  }
} ///:~
```

这时，数组的创建甚至可以放在 **new** 的后面：

```
Integer[] a = new Integer[rand.nextInt(20)];
```

这只是一个 reference 的数组，所以只要还没有创建新的 **Integer** 对象来初始化这些 reference，初始化的过程就没有结束：

```
a[i] = new Integer(rand.nextInt(500));
```

然而，如果你忘了创建对象，那么程序运行的时候，当你要用到数组的空位置的时候，就会得到一个异常。

我们来看看打印语句里面的 **String** 对象的构成。你会看到，**Integer** 对象的 reference 会被自动地转换成表示这个值的 **String**。

也可以用一个由花括号括起来的对象列表来初始化对象数组。它有两种形式：

```
//: c04:ArrayInit.java
// Array initialization.

public class ArrayInit {
  public static void main(String[] args) {
    Integer[] a = {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
    Integer[] b = new Integer[] {
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
  }
} ///:~
```

第一种形式能用于很多场合，但是它的局限性较大，因为数组的大小在编译的时候就已经确定了。这各列表的最后一个逗号是可选的。(这个特性能使长列表的维护工作变得简单一些。)

第二种形式提供了一种简便的，能达到 C 的可变参数列表(*variable argument lists*，也被称为 C 的“varargs”)的效果的，创建和调用方法的语法。它既能用于参数数量未知的场合，也能用于参数类型未知的场合。由于所有的类都追根溯源继承自公共的 **Object** 根类(随着本书的进展，你会对这个主题有更深的理解)，你可以创建一个拿 **Object** 数组作参数的方法，然后这样调用：

```
//: c04:VarArgs.java
// Using array syntax to create variable argument
lists.
import com.bruceeckel.simpletest.*;

class A { int i; }

public class VarArgs {
  static Test monitor = new Test();
  static void print(Object[] x) {
    for(int i = 0; i < x.length; i++)
      System.out.println(x[i]);
  }
  public static void main(String[] args) {
    print(new Object[] {
      new Integer(47), new VarArgs(),
      new Float(3.14), new Double(11.11)
    });
    print(new Object[] {"one", "two", "three" });
    print(new Object[] {new A(), new A(), new A()});
    monitor.expect(new Object[] {
      "47",
      "%% VarArgs@\\p{XDigit}+",
      "3.14",
      "11.11",
      "one",
      "two",
      "three",
      new TestExpression("%% A@\\p{XDigit}+", 3)
    });
  }
} ///:~
```

你会看到 **print( )**拿一个 **Object** 数组作参数，它会遍历整个数组，再把对象都打印出来。标准的 Java 类库能生成很容易理解的输出，但是这里所创建的 **A** 和 **VarArgs** 类的对象，会先打印类的名字，然后是‘@’符号，以及一个表示 16 进制数的正则表达式，**\p{XDigit}**。最后那个‘+’表示会有一个或多个 16 进制的数字。于是其默认的动作(假设你不定义类的 **toString( )**方法。我们以后会讲这个方法的。)就是打印类的名字，然后是对象的地址。

多维数组

Java 能让你很容易地创建多维数组：

```
//: c04:MultiDimArray.java
// Creating multidimensional arrays.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class MultiDimArray {
  static Test monitor = new Test();
  static Random rand = new Random();
  public static void main(String[] args) {
    int[][] a1 = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    for(int i = 0; i < a1.length; i++)
      for(int j = 0; j < a1[i].length; j++)
        System.out.println(
          "a1[" + i + "][" + j + "] = " + a1[i][j]);
    // 3-D array with fixed length:
    int[][][] a2 = new int[2][2][4];
    for(int i = 0; i < a2.length; i++)
      for(int j = 0; j < a2[i].length; j++)
        for(int k = 0; k < a2[i][j].length; k++)
          System.out.println("a2[" + i + "][" + j +
"][" +
            k + "] = " + a2[i][j][k]);
    // 3-D array with varied-length vectors:
    int[][][] a3 = new int[rand.nextInt(7)][][];
    for(int i = 0; i < a3.length; i++) {
      a3[i] = new int[rand.nextInt(5)][];
      for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[rand.nextInt(5)];
    }
    for(int i = 0; i < a3.length; i++)
      for(int j = 0; j < a3[i].length; j++)
        for(int k = 0; k < a3[i][j].length; k++)
          System.out.println("a3[" + i + "][" + j +
"][" +
            k + "] = " + a3[i][j][k]);
    // Array of nonprimitive objects:
    Integer[][] a4 = {
      { new Integer(1), new Integer(2)},
      { new Integer(3), new Integer(4)},
      { new Integer(5), new Integer(6)},
    };
    for(int i = 0; i < a4.length; i++)
      for(int j = 0; j < a4[i].length; j++)
        System.out.println("a4[" + i + "][" + j +
            "] = " + a4[i][j]);
    Integer[][] a5;
    a5 = new Integer[3][];
    for(int i = 0; i < a5.length; i++) {
      a5[i] = new Integer[3];
      for(int j = 0; j < a5[i].length; j++)
        a5[i][j] = new Integer(i * j);
    }
    for(int i = 0; i < a5.length; i++)
      for(int j = 0; j < a5[i].length; j++)
        System.out.println("a5[" + i + "][" + j +
            "] = " + a5[i][j]);
    // Output test
    int ln = 0;
```

```
    for(int i = 0; i < a3.length; i++)
      for(int j = 0; j < a3[i].length; j++)
        for(int k = 0; k < a3[i][j].length; k++)
          ln++;
    monitor.expect(new Object[] {
      "a1[0][0] = 1",
      "a1[0][1] = 2",
      "a1[0][2] = 3",
      "a1[1][0] = 4",
      "a1[1][1] = 5",
      "a1[1][2] = 6",
      new TestExpression(
        "%% a2\\[\\d\\]\\[\\d\\]\\[\\d\\] = 0", 16),
      new TestExpression(
        "%% a3\\[\\d\\]\\[\\d\\]\\[\\d\\] = 0", ln),
      "a4[0][0] = 1",
      "a4[0][1] = 2",
      "a4[1][0] = 3",
      "a4[1][1] = 4",
      "a4[2][0] = 5",
      "a4[2][1] = 6",
      "a5[0][0] = 0",
      "a5[0][1] = 0",
      "a5[0][2] = 0",
      "a5[1][0] = 0",
      "a5[1][1] = 1",
      "a5[1][2] = 2",
      "a5[2][0] = 0",
      "a5[2][1] = 2",
      "a5[2][2] = 4"
    });
  }
} ///:~
```

打印程序用了 **length**，这样它就不用绑在固定长度的数组上了。

第一个例子演示了 primitive 的多维数组。你用花括号标出多维数组的每个向量：

```
int[][] a1 = {
  { 1, 2, 3, },
  { 4, 5, 6, },
};
```

各组方括号都会把你带进下一级数组。

第二个例子演示了用 **new** 来分配一个三维数组。这时数组是一次性分配的：

```
int[][][] a2 = new int[2][2][4];
```

但是第三段程序揭示了，组成数组矩阵的各个向量都可以是任意长度的：

```
int[][][] a3 = new int[rand.nextInt(7)][][];
for(int i = 0; i < a3.length; i++) {
  a3[i] = new int[rand.nextInt(5)][];
  for(int j = 0; j < a3[i].length; j++)
    a3[i][j] = new int[rand.nextInt(5)];
}
```

第一个 **new** 创建了一个随机长度的数组，它决定了第一维的长度，而其它两维则未作决定。**for** 循环的第二个 **new** 填满了各个元素，但是其第三维还是悬而未决，这个问题留给了第三个 **new**。

你可以从输出看出，如果你不给数组明确地初始值的话，它的值会被自动地初始化为零。

你可以用类似的风格来处理非 primitive 的对象数组。这个演示在第四个例子里面，它用花括号把许多 **new** 表达式给组织起来：

```
Integer[][] a4 = {
  { new Integer(1), new Integer(2)},
  { new Integer(3), new Integer(4)},
  { new Integer(5), new Integer(6)},
};
```

第五个例子演示了如何一段一段地创建非 primitive 的对象数组：

```
Integer[][] a5;
a5 = new Integer[3][];
for(int i = 0; i < a5.length; i++) {
  a5[i] = new Integer[3];
  for(int j = 0; j < a5[i].length; j++)
    a5[i][j] = new Integer(i*j);
}
```

用 **i*j** 只是为了往 **Integer** 里面放一些有意思的值。

总结

构造函数，这种工于心计的初始化机制，应该给了你很强的暗示，告诉你 Java 是多么关注这个问题。当 C++的发明人 Bjarne Stroustrup 还在设计 C++语言的时候，他就在最初几份关于 C 在编程效率的观察报告中，单独作了份关于不正确的变量初始化而引起的大量的编程问题的报告。这种 bug 是很难找的，此外这一问题也适用于不正确的清理。由于

构造函数能让你确保进行适当的初始化和清理(编译器不会让你未调用构造函数就创建对象)，因此你就有了完全的控制和安全。

在 C++里，拆构过程(destruction)是相当重要的，因为用 **new** 创建的对象必须被明确的清理掉。由于 Java 的垃圾回收器会自动释放所有对象的内存，因而在很多场合下，它根本就用不到清理方法(不过本章也讲了，真正要用的时候，你就只能自己写了)。在那些不需要拆构函数的行为的场合，Java 的垃圾回收器能在大大简化编程工作的同时，还为你奉上你梦寐以求的管理内存方面的安全保障。有些垃圾回收器甚至可以回收图像，文件句柄之类的其它资源。但是，垃圾回收器的开销是很大的，而且由于 Java 的解释器从来就比较慢，因此它对性能的影响还很难评估。尽管 Java 在性能方面已经取得了长足的进步，速度问题仍然是妨碍它进入某些编程领域的最大障碍。

由于要确保所有对象都能被正确地创建，构造函数实际上要比这里讲的更复杂。特别是，当你用合成(*composition*)和继承(*inheritance*)创建新的类的时候，这种保障应该仍然有效，这时会用到一些别的语句来提供支持。在以后的章节中，你会学到合成，继承，以及它们是如何影响对象的创建的。

练习

只要付很小一笔费用就能从 *www.BruceEckel.com* 下载名为 *The Thinking in Java Annotated Solution Guide* 的电子文档，这上面有一些习题的答案。

1. 创建一个带默认构造函数(不带参数的构造函数)的类，这个构造函数应该能打印消息。创建一个这个类的对象。
2. 重载练习 1 中的构造函数，让它拿一个 **String** 作参数，然后把这个字符串同消息一同打印出来。
3. 创建一个练习 2 所定义的类的对象的数组，但是不要创建对象，也不要为数组赋值。运行程序的时候观察一下，它是不是会打印调用构造函数所产生的初始化信息。
4. 修改练习 3，创建一组对象，并且把它连到 reference 的数组。
5. 创建一个 String 对象的数组，然后为每个元素赋值。用 **for** 循环打印数组。
6. 创建一个重载 **bark( )**方法的 **Dog** 类。这个方法要对各种 primitive 类型进行重载，并且还要根据调用的版本打印出是叫(barking)还是嚎(howling)。写一个 **main( )**方法，把所有的版本都调用一遍。
7. 修改练习 6，让它有两个重载方法，这两个方法应该有两个(类型不同)顺序相反的参数。检验一下它是不是能正常工作。
8. 创建一个不带构造函数的类，然后用 **main( )**创建一个那个类的对象，以检验它会不会自动生成一个默认的构造函数。

9. 创建一个有两个方法的类。在第一个方法理，调用两次第二各方法：第一次不用 **this**，第二次用 **this**。
10. 创建一个有两个(重载)构造函数的类。在第一个构造函数里用 **this** 调用第二个构造函数。
11. 创建一个带 **finalize( )**方法的类，这个方法应该能打印消息。在 **main( )**里面，创建一个这个类的对象。解释一下程序是怎样运行的。
12. 修改练习 11 的程序，让它肯定能调用 **finalize( )** 。
13. 创建一个名为 **Tank** 的类。它能填满，也能排空，还有中止条件。中止条件就是，清理对象的时候，油箱应该是空的。写一个检查这个中止条件的 **finalize( )**方法。用 **main( )**测试各种 **Tank** 被用过之后的情景。
14. 创建一个包含未作初始化的 **int** 和 **char** 的类，打印其值以检验 Java 是不是会做默认的初始化。
15. 创建一个未作初始化的 **String** reference。检验一下，Java 是不是会把它初始化为 **null**。
16. 创建一个带 **String** 成员的类，在定义的时候就进行初始化，再创建一个在构造函数里进行初始化的类。这两者有什么区别？
17. 创建一个带 **static String** 成员的类，一个在定义的时候进行初始化，另一个在 **static** 块里进行初始化的类。给这个类添一个会打印这两个成员的 **static** 方法，看看它们在使用之前是不是进行了初始化。
18. 创建一个带 **String** 成员的类，用“实例初始化”进行初始化。举一个这种特性的用途(本书所讲的不算)。
19. 写一个会创建并初始化一个两维 **double** 数组的方法。数组的大小要由方法的参数决定，初始化的值得在方法参数所给出的最小值和最大值之间。创建第二个方法，让它打印第一个方法生成的数组。在 **main( )**里创建并打印几个不同大小的数组，以此来测试这两个方法。
20. 用三维数组重写练习 19。
21. 将 **ExplicitStatic.java** 中用(1)标注的那行注释掉，看看它是不是不调用静态初始化语句了。再将用(2)标注的那两行中的一行恢复回来，看看它是不是又调用静态初始化语句。最后将另一行用(2)标注的代码恢复回来，看看静态初始化是不是只执行一次。

---

[19]在 Sun 发表的一些 Java 文档中，他们使用了一个有些奇怪但却很直观的名字“no-arg constructors (无参数的构造函数)”。但是“default constructor (默认的构造函数)”这个术语已经用了很多年了，所以这里我仍然用这个。

[20]有些人会执着的将 **this** 放到每个方法调用和成员数据的 reference 前面，他们声称这样做能使代码“更清晰更明确。”别这么做。我们之所以要使用高级语言的原因就是要让它替我们作一些事情。如果你把 **this** 放到一些没必要的地方，那么读程序的人就会被搞得晕头转向，因为别的程序员不会把 **this** 到处乱放的。遵循一种一致而简明的编程风格可以节省时间和金钱。

[21]假设可以传一个对象的 reference 给 **static** 方法的话，那么就可以通过这个 reference(实际上就是 **this**)调用别的非 static 的方法了，并且访问非 static 的字段了。但是如果你要达到这个目的的话，通常应该使用普通的，非 static 的方法。

[22] Joshua Bloch 在题为“回避 finalize( ) (avoid finalizers)”一节中讲得更绝：“Finalize( )是无法预料的，常常是危险的，总之是多余的。” *Effective Java*，第 20 页，(Addison-Wesley 2001)。

[23]这个术语是由 Bill Venners (www.artima.com)在一期他同我合开的培训班上发明的。

[24]相反，C++有构造函数的初始化列表(*constructor initializer list*)。它会在对象进入构造函数的主体之前对它强制进行初始化的。见 *Thinking in* C++，第二版(本书的 CD ROM 里面有，此外也可以到 *www.BruceEckel.com* 下载。)

[25] 若要完整的了解 C++的总体初始化，请参见 *Thinking in C++*，第二版。