第 4 章 初始化和清除

“随着计算机的进步，‘不安全’的程序设计已成为造成编程代价高昂的罪魁祸首之一。”

“初始化”和“清除”是这些安全问题的其中两个。许多 C 程序的错误都是由于程序员忘记初始化一个变量造成的。对于现成的库，若用户不知道如何初始化库的一个组件，就往往会出现这一类的错误。清除是另一个特殊的问题，因为用完一个元素后，由于不再关心，所以很容易把它忘记。这样一来，那个元素占用的资源会一直保留下去，极易产生资源（主要是内存）用尽的后果。

C++为我们引入了“构建器”的概念。这是一种特殊的方法，在一个对象创建之后自动调用。Java 也沿用了这个概念，但新增了自己的“垃圾收集器”，能在资源不再需要的时候自动释放它们。本章将讨论初始化和清除的问题，以及 Java 如何提供它们的支持。

4.1 用构建器自动初始化

对于方法的创建，可将其想象成为自己写的每个类都调用一次 initialize()。这个名字提醒我们在使用对象之前，应首先进行这样的调用。但不幸的是，这也意味着用户必须记住调用方法。在 Java 中，由于提供了名为“构建器”的一种特殊方法，所以类的设计者可担保每个对象都会得到正确的初始化。若某个类有一个构建器，那么在创建对象时，Java 会自动调用那个构建器——甚至在用户毫不知觉的情况下。所以说这是可以担保的！

接着的一个问题是如何命名这个方法。存在两方面的问题。第一个是我们使

用的任何名字都可能与打算为某个类成员使用的名字冲突。第二是由于编译器的责任是调用构建器，所以它必须知道要调用是哪个方法。C++采取的方案看来是最简单的，且更有逻辑性，所以也在 Java 里得到了应用：构建器的名字与类名相同。这样一来，可保证象这样的一个方法会在初始化期间自动调用。

下面是带有构建器的一个简单的类（若执行这个程序有问题，请参考第 3 章的“赋值”小节）。

148-149 页程序

```
//: c04:SimpleConstructor.java
// Demonstration of a simple constructor.

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

public class SimpleConstructor {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock();
  }
} ///:~
```

现在，一旦创建一个对象：

```
new Rock();
```

就会分配相应的存储空间，并调用构建器。这样可保证在我们经手之前，对象得到正确的初始化。

请注意所有方法首字母小写的编码规则并不适用于构建器。这是由于构建器的名字必须与类名完全相同！

和其他任何方法一样，构建器也能使用自变量，以便我们指定对象的具体创建方式。可非常方便地改动上述例子，以便构建器使用自己的自变量。如下所示：

149 页中程序

```
//: c04:SimpleConstructor2.java
// Constructors can have arguments.

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

```
public class SimpleConstructor2 {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rock2(i);
  }
} ///:~
```

利用构建器的自变量，我们可为一个对象的初始化设定相应的参数。举个例子来说，假设类 **Tree** 有一个构建器，它用一个整数自变量标记树的高度，那么就可以象下面这样创建一个 Tree 对象：

```
tree t = new Tree(12); // 12 英尺高的树
```

若 Tree(int)是我们唯一的构建器，那么编译器不会允许我们以其他任何方式创建一个 Tree 对象。

构建器有助于消除大量涉及类的问题，并使代码更易阅读。例如在前述的代码段中，我们并未看到对 initialize()方法的明确调用——那些方法在概念上独立于定义内容。在 Java 中，定义和初始化属于统一的概念——两者缺一不可。

构建器属于一种较特殊的方法类型，因为它没有返回值。这与 void 返回值存在着明显的区别。对于 void 返回值，尽管方法本身不会自动返回什么，但仍然可以让它返回另一些东西。构建器则不同，它不仅什么也不会自动返回，而且根本不能有任何选择。若存在一个返回值，而且假设我们可以自行选择返回内容，那么编译器多少要知道如何对那个返回值作什么样的处理。

4.2 方法过载

在任何程序设计语言中，一项重要的特性就是名字的运用。我们创建一个对象时，会分配到一个保存区域的名字。方法名代表的是一种具体的行动。通过用名字描述自己的系统，可使自己的程序更易人们理解和修改。它非常象写散文——目的是与读者沟通。

我们用名字引用或描述所有对象与方法。若名字选得好，可使自己及其他人更易理解自己的代码。

将人类语言中存在细致差别的概念“映射”到一种程序设计语言中时，会出现一些特殊的问题。在日常生活中，我们用相同的词表达多种不同的含义——即词的“过载”。我们说“洗衬衫”、“洗车”以及“洗狗”。但若强制象下面这样说，就显得很愚蠢：“衬衫洗 衬衫”、“车洗 车”以及“狗洗 狗”。这是由于听众根本不需要对执行的行动作任何明确的区分。人类的大多数语言都具有很强的“冗余”性，所以即使漏掉了几个词，仍然可以推断出含义。我们不需要独一无二的标识符——可从具体的语境中推论出含义。

大多数程序设计语言（特别是 C）要求我们为每个函数都设定一个独一无二的标识符。所以绝对不能用一个名为 print()的函数来显示整数，再用另一个 print()显示浮点数——每个函数都要求具备唯一的名字。

在 Java 里，另一项因素强迫方法名出现过载情况：构建器。由于构建器的名字由类名决定，所以只能有一个构建器名称。但假若我们想用多种方式创建一个

对象呢？例如，假设我们想创建一个类，令其用标准方式进行初始化，另外从文件里读取信息来初始化。此时，我们需要两个构建器，一个没有自变量（默认构建器），另一个将字串作为自变量——用于初始化对象的那个文件的名字。由于都是构建器，所以它们必须有相同的名字，亦即类名。所以为了让相同的方法名伴随不同的自变量类型使用，“方法过载”是非常关键的一项措施。同时，尽管方法过载是构建器必需的，但它亦可应用于其他任何方法，且用法非常方便。

在下面这个例子里，我们向大家同时展示了过载构建器和过载的原始方法：

151-152 页程序

```
//: c04:Overloading.java
// Demonstration of both constructor
// and ordinary method overloading.
import java.util.*;
class Tree {
          int height;
          Tree() {
                prt("Planting a seedling");
                height = 0;
          }
          Tree(int i) {
             prt("Creating new Tree that is " + i + " feet tall");
             height = i;
           }
         void info() {
             prt("Tree is " + height + " feet tall");
         }
         void info(String s) {
             prt(s + ": Tree is " + height + " feet tall");
         }
      static void prt(String s) {       System.out.println(s);    }}
public class Overloading {
         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();
}
} ///:~
```

Tree 既可创建成一颗种子，不含任何自变量；亦可创建成生长在苗圃中的植物。为支持这种创建，共使用了两个构建器，一个没有自变量（我们把没有自变量的构建器称作“默认构建器”，注释①），另一个采用现成的高度。

①：在 Sun 公司出版的一些 Java 资料中，用简陋但很说明问题的词语称呼这

类构建器——“无参数构建器”（no-arg constructors）。但“默认构建器”这个称呼已使用了许多年，所以我选择了它。

我们也有可能希望通过多种途径调用 **info()**方法。例如，假设我们有一条额外的消息想显示出来，就使用 String 自变量；而假设没有其他话可说，就不使用。由于为显然相同的概念赋予了两个独立的名字，所以看起来可能有些古怪。幸运的是，方法过载允许我们为两者使用相同的名字。

4.2.1 区分过载方法

若方法有同样的名字，Java 怎样知道我们指的哪一个方法呢？这里有一个简单的规则：每个过载的方法都必须采取独一无二的自变量类型列表。

若稍微思考几秒钟，就会想到这样一个问题：除根据自变量的类型，程序员如何区分两个同名方法的差异呢？

即使自变量的顺序也足够我们区分两个方法（尽管我们通常不愿意采用这种方法，因为它会产生难以维护的代码）：

152-153 页程序

```
//: c04:OverloadingOrder.java
// Overloading based on the order of
// the arguments.

public class OverloadingOrder {
  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");
  }
} ///:~
```

两个 print()方法有完全一致的自变量，但顺序不同，可据此区分它们。

4.2.2 主类型的过载

主（数据）类型能从一个“较小”的类型自动转变成一个“较大”的类型。涉及过载问题时，这会稍微造成一些混乱。下面这个例子揭示了将主类型传递给过载的方法时发生的情况：

153-155 页程序

```
//: c04:PrimitiveOverloading.java
// Promotion of primitives and overloading.

public class PrimitiveOverloading {
  // boolean can't be automatically converted
  static void prt(String s) {
    System.out.println(s);
  }

  void f1(char x) { prt("f1(char)"); }
  void f1(byte x) { prt("f1(byte)"); }
  void f1(short x) { prt("f1(short)"); }
  void f1(int x) { prt("f1(int)"); }
  void f1(long x) { prt("f1(long)"); }
  void f1(float x) { prt("f1(float)"); }
  void f1(double x) { prt("f1(double)"); }

  void f2(byte x) { prt("f2(byte)"); }
  void f2(short x) { prt("f2(short)"); }
  void f2(int x) { prt("f2(int)"); }
  void f2(long x) { prt("f2(long)"); }
  void f2(float x) { prt("f2(float)"); }
  void f2(double x) { prt("f2(double)"); }

  void f3(short x) { prt("f3(short)"); }
  void f3(int x) { prt("f3(int)"); }
  void f3(long x) { prt("f3(long)"); }
  void f3(float x) { prt("f3(float)"); }
  void f3(double x) { prt("f3(double)"); }

  void f4(int x) { prt("f4(int)"); }
  void f4(long x) { prt("f4(long)"); }
  void f4(float x) { prt("f4(float)"); }
  void f4(double x) { prt("f4(double)"); }

  void f5(long x) { prt("f5(long)"); }
  void f5(float x) { prt("f5(float)"); }
  void f5(double x) { prt("f5(double)"); }

  void f6(float x) { prt("f6(float)"); }
  void f6(double x) { prt("f6(double)"); }
```

```
void f7(double x) { prt("f7(double)"); }

void testConstVal() {
   prt("Testing with 5");
   f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
}
void testChar() {
   char x = 'x';
   prt("char argument:");
   f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testByte() {
   byte x = 0;
   prt("byte argument:");
   f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testShort() {
   short x = 0;
   prt("short argument:");
   f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testInt() {
   int x = 0;
   prt("int argument:");
   f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testLong() {
   long x = 0;
   prt("long argument:");
   f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testFloat() {
   float x = 0;
   prt("float argument:");
   f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
void testDouble() {
   double x = 0;
   prt("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();
    }
} ///:~
```

若观察这个程序的输出，就会发现常数值 5 被当作一个 int 值处理。所以假若可以使用一个过载的方法，就能获取它使用的 int 值。在其他所有情况下，若我们的数据类型“小于”方法中使用的自变量，就会对那种数据类型进行“转型”处理。char 获得的效果稍有些不同，这是由于假期它没有发现一个准确的 char 匹配，就会转型为 int。

若我们的自变量“大于”过载方法期望的自变量，这时又会出现什么情况呢？对前述程序的一个修改揭示出了答案：

155-157 页程序

```
//: c04:Demotion.java
// Demotion of primitives and overloading.

public class Demotion {
  static void prt(String s) {
    System.out.println(s);
  }

  void f1(char x) { prt("f1(char)"); }
  void f1(byte x) { prt("f1(byte)"); }
  void f1(short x) { prt("f1(short)"); }
  void f1(int x) { prt("f1(int)"); }
  void f1(long x) { prt("f1(long)"); }
  void f1(float x) { prt("f1(float)"); }
  void f1(double x) { prt("f1(double)"); }

  void f2(char x) { prt("f2(char)"); }
  void f2(byte x) { prt("f2(byte)"); }
  void f2(short x) { prt("f2(short)"); }
  void f2(int x) { prt("f2(int)"); }
  void f2(long x) { prt("f2(long)"); }
  void f2(float x) { prt("f2(float)"); }

  void f3(char x) { prt("f3(char)"); }
```

```
    void f3(byte x) { prt("f3(byte)"); }
    void f3(short x) { prt("f3(short)"); }
    void f3(int x) { prt("f3(int)"); }
    void f3(long x) { prt("f3(long)"); }

    void f4(char x) { prt("f4(char)"); }
    void f4(byte x) { prt("f4(byte)"); }
    void f4(short x) { prt("f4(short)"); }
    void f4(int x) { prt("f4(int)"); }

    void f5(char x) { prt("f5(char)"); }
    void f5(byte x) { prt("f5(byte)"); }
    void f5(short x) { prt("f5(short)"); }

    void f6(char x) { prt("f6(char)"); }
    void f6(byte x) { prt("f6(byte)"); }

    void f7(char x) { prt("f7(char)"); }

    void testDouble() {
      double x = 0;
      prt("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();
    }
} ///:~
```

在这里，方法采用了容量更小、范围更窄的主类型值。若我们的自变量范围比它宽，就必须用括号中的类型名将其转为适当的类型。如果不这样做，编译器会报告出错。

大家可注意到这是一种“缩小转换”。也就是说，在造型或转型过程中可能丢失一些信息。这正是编译器强迫我们明确定义的原因——我们需明确表达想要转型的愿望。

4.2.3 返回值过载

我们很易对下面这些问题感到迷惑：为什么只有类名和方法自变量列出？为什么不根据返回值对方法加以区分？比如对下面这两个方法来说，虽然它们有同样的名字和自变量，但其实是很容易区分的：

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

若编译器可根据上下文（语境）明确判断出含义，比如在 int x=f()中，那么这样做完全没有问题。然而，我们也可能调用一个方法，同时忽略返回值；我们通常把这称为“为它的副作用去调用一个方法”，因为我们关心的不是返回值，而是方法调用的其他效果。所以假如我们象下面这样调用方法：

```
f();
```

Java 怎样判断 f()的具体调用方式呢？而且别人如何识别并理解代码呢？由于存在这一类的问题，所以不能根据返回值类型来区分过载的方法。

4.2.4 默认构建器

正如早先指出的那样，默认构建器是没有自变量的。它们的作用是创建一个“空对象”。若创建一个没有构建器的类，则编译程序会帮我们自动创建一个默认构建器。例如：

158 页上程序

```
//: c04:DefaultConstructor.java

class Bird {
  int i;
}

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

对于下面这一行：

```
new Bird();
```

它的作用是新建一个对象，并调用默认构建器——即使尚未明确定义一个象这样的构建器。若没有它，就没有方法可以调用，无法构建我们的对象。然而，如果已经定义了一个构建器（无论是否有自变量），编译程序都不会帮我们自动合成一个：

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

现在，假若使用下述代码：

```
new Bush();
```

编译程序就会报告自己找不到一个相符的构建器。就好象我们没有设置任何构建器，编译程序会说：“你看来似乎需要一个构建器，所以让我们给你制造一个吧。”但假如我们写了一个构建器，编译程序就会说：“啊，你已写了一个构建器，所以我知道你想干什么；如果你不放置一个默认的，是由于你打算省略它。”

4.2.5 this 关键字

如果有两个同类型的对象，分别叫作 a 和 b，那么您也许不知道如何为这两个对象同时调用一个 f()方法：

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

若只有一个名叫 f()的方法，它怎样才能知道自己是为 a 还是为 b 调用的呢？

为了能用简便的、面向对象的语法来书写代码——亦即“将消息发给对象”，编译器为我们完成了一些幕后工作。其中的秘密就是第一个自变量传递给方法 f()，而且那个自变量是准备操作的那个对象的句柄。所以前述的两个方法调用就变成了下面这样的形式：

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

这是内部的表达形式，我们并不能这样书写表达式，并试图让编译器接受它。但是，通过它可理解幕后到底发生了什么事情。

假定我们在一个方法的内部，并希望获得当前对象的句柄。由于那个句柄是由编译器“秘密”传递的，所以没有标识符可用。然而，针对这一目的有个专用的关键字：**this**。**this 关键字**（注意只能在方法内部使用）可为已调用了其方法的那个对象生成相应的句柄。可象对待其他任何对象句柄一样对待这个句柄。但要注意，假若准备从自己某个类的另一个方法内部调用一个类方法，就不必使用 **this**。只需简单地调用那个方法即可。当前的 this 句柄会自动应用于其他方法。所以我们能使用下面这样的代码：

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

在 pit()内部，我们可以说 this.pick()，但事实上无此必要。编译器能帮我们自动完成。this 关键字只能用于那些特殊的类——需明确使用当前对象的句柄。例如，假若您希望将句柄返回给当前对象，那么它经常在 **return** 语句中使用。

160 页上程序

```
//: c04:Leaf.java
// Simple use of the "this" keyword.

public class Leaf {
  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();
  }
} ///:~
```

由于 increment()通过 this 关键字返回当前对象的句柄，所以可以方便地对同一个对象执行多项操作。

1. 在构建器里调用构建器

若为一个类写了多个构建器，那么经常都需要在一个构建器里调用另一个构建器，以避免写重复的代码。可用 this 关键字做到这一点。

通常，当我们说 this 的时候，都是指“这个对象”或者“当前对象”。而且它本身会产生当前对象的一个句柄。在一个构建器中，若为其赋予一个自变量列表，那么 this 关键字会具有不同的含义：它会对与那个自变量列表相符的构建器进行明确的调用。这样一来，我们就可通过一条直接的途径来调用其他构建器。如下所示：

160-161 页程序

```
//: c04:Flower.java
// Calling constructors with "this."

public class Flower {
  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();
    }
} ///:~
```

其中，构建器 Flower(String s,int petals)向我们揭示出这样一个问题：尽管可用 this 调用一个构建器，但不可调用两个。除此以外，构建器调用必须是我们做的第一件事情，否则会收到编译程序的报错信息。

这个例子也向大家展示了 this 的另一项用途。由于自变量 s 的名字以及成员数据 s 的名字是相同的，所以会出现混淆。为解决这个问题，可用 this.s 来引用成员数据。经常都会在 Java 代码里看到这种形式的应用，本书的大量地方也采用了这种做法。

在 print()中，我们发现编译器不让我们从除了一个构建器之外的其他任何方法内部调用一个构建器。

2. static 的含义

理解了 this 关键字后，我们可更完整地理解 static（静态）方法的含义。它意味着一个特定的方法没有 this。我们不可从一个 **static 方法**内部发出对非 static 方法的调用（注释②），尽管反过来说是可以的。而且在没有任何对象的前提下，我们可针对类本身发出对一个 static 方法的调用。事实上，那正是 static 方法最基本的意义。它就好象我们创建一个全局函数的等价物（在 C 语言中）。除了全局函数不允许在 Java 中使用以外，若将一个 static 方法置入一个类的内部，它就可以访问其他 static 方法以及 static 字段。

②：有可能发出这类调用的一种情况是我们将一个对象句柄传到 static 方法内部。随后，通过句柄（此时实际是 this），我们可调用非 static 方法，并访问非 static 字段。但一般地，如果真的想要这样做，只要制作一个普通的、非 static 方法即可。

有些人抱怨 static 方法并不是“面向对象”的，因为它们具有全局函数的某些特点；利用 static 方法，我们不必向对象发送一条消息，因为不存在 this。这可能是一个清楚的自变量，若您发现自己使用了大量静态方法，就应重新思考自己的策略。然而，static 的概念是非常实用的，许多时候都需要用到它。所以至于它们是否真的“面向对象”，应该留给理论家去讨论。事实上，即使 Smalltalk 在自己的“类方法”里也有类似于 static 的东西。

4.3 清除：收尾和垃圾收集

程序员都知道“初始化”的重要性，但通常忘记清除的重要性。毕竟，谁需要来清除一个 int 呢？但是对于库来说，用完后简单地“释放”一个对象并非总是安全的。当然，Java 可用垃圾收集器回收由不再使用的对象占据的内存。现在考虑一种非常特殊且不多见的情况。假定我们的对象分配了一个“特殊”内存区域，没有使用 new。**垃圾收集器只知道释放那些由 new 分配的内存**，所以不知道如何释放对象的“特殊”内存。为解决这个问题，Java 提供了一个名为 **finalize()** 的方法，可为我们的类定义它。在理想情况下，它的工作原理应该是这样的：一旦垃圾收集器准备好释放对象占用的存储空间，它首先调用 finalize()，而且只有在下一次垃圾收集过程中，才会真正回收对象的内存。所以如果使用 finalize()，就可以在垃圾收集期间进行一些重要的清除或清扫工作。

但也是一个潜在的编程陷阱，因为有些程序员（特别是在 C++开发背景的）刚开始可能会错误认为它就是在 C++中为“破坏器”（Destructor）使用的 finalize()——破坏（清除）一个对象的时候，肯定会调用这个函数。但在这里有必要区分一下 C++和 Java 的区别，因为 C++的对象肯定会被清除（排开编程错误的因素），而 Java 对象并非肯定能作为垃圾被“收集”去。或者换句话说：

垃圾收集并不等于“破坏”！

若能时刻牢记这一点，踩到陷阱的可能性就会大大减少。它意味着在我们不再需要一个对象之前，有些行动是必须采取的，而且必须由自己来采取这些行动。Java 并未提供“破坏器”或者类似的概念，所以必须创建一个原始的方法，用它来进行这种清除。例如，假设在对象创建过程中，它会将自己描绘到屏幕上。如果不从屏幕明确删除它的图像，那么它可能永远都不会被清除。若在 finalize()里置入某种删除机制，那么假设对象被当作垃圾收掉了，图像首先会将自身从屏幕上移去。但若未被收掉，图像就会保留下来。所以要记住的第二个重点是：

我们的对象可能不会当作垃圾被收掉！

有时可能发现一个对象的存储空间永远都不会释放，因为自己的程序永远都接近于用光空间的临界点。若程序执行结束，而且垃圾收集器一直都没有释放我们创建的任何对象的存储空间，则随着程序的退出，那些资源会返回给操作系统。这是一件好事情，因为垃圾收集本身也要消耗一些开销。如永远都不用它，那么永远也不用支出这部分开销。

4.3.1 finalize()用途何在

此时，大家可能已相信了自己应该将 finalize()作为一种常规用途的清除方法使用。它有什么好处呢？

要记住的第三个重点是：

垃圾收集只跟内存有关！

也就是说，垃圾收集器存在的唯一原因是为了回收程序不再使用的内存。所以对于与垃圾收集有关的任何活动来说，其中最值得注意的是 finalize()方法，它们也必须同内存以及它的回收有关。

但这是否意味着假如对象包含了其他对象，finalize()就应该明确释放那些对象呢？答案是否定的——垃圾收集器会负责释放所有对象占据的内存，无论这些对象是如何创建的。它将对 finalize()的需求限制到特殊的情况。在这种情况下，我们的对象可采用与创建对象时不同的方法分配一些存储空间。但大家或许会注意到，Java 中的所有东西都是对象，所以这到底是怎么一回事呢？

之所以要使用 finalize()，看起来似乎是由于有时需要采取与 Java 的普通方法不同的一种方法，通过分配内存来做一些具有 C 风格的事情。这主要可以通过“固有方法”来进行，它是从 Java 里调用非 Java 方法的一种方式（固有方法的问题在附录 A 讨论）。C 和 C++是目前唯一获得固有方法支持的语言。但由于它们能调用通过其他语言编写的子程序，所以能够有效地调用任何东西。在非 Java 代码内部，也许能调用 C 的 malloc()系列函数，用它分配存储空间。而且除非调用了 free()，否则存储空间不会得到释放，从而造成内存“漏洞”的出现。当然，free()是一个 C 和 C++函数，所以我们需要在 finalize()内部的一个固有方法中调用它。

读完上述文字后，大家或许已弄清楚了自己不必过多地使用 finalize()。这个思想是正确的；它并不是进行普通清除工作的理想场所。那么，普通的清除工作应在何处进行呢？

4.3.2 必须执行清除

为清除一个对象，那个对象的用户必须在希望进行清除的地点调用一个清除方法。这听起来似乎很容易做到，但却与 C++“破坏器”的概念稍有抵触。在 C++中，所有对象都会破坏（清除）。或者换句话说，所有对象都“应该”破坏。若将 C++对象创建成一个本地对象，比如在堆栈中创建（在 Java 中是不可能的），那么清除或破坏工作就会在“结束花括号”所代表的、创建这个对象的作用域的末尾进行。若对象是用 new 创建的（类似于 Java），那么当程序员调用 C++的 delete 命令时（Java 没有这个命令），就会调用相应的破坏器。若程序员忘记了，那么永远不会调用破坏器，我们最终得到的将是一个内存“漏洞”，另外还包括对象的其他部分永远不会得到清除。

相反，Java 不允许我们创建本地（局部）对象——无论如何都要使用 new。但在 Java 中，没有“delete”命令来释放对象，因为垃圾收集器会帮助我们自动释放存储空间。所以如果站在比较简化的立场，我们可以说正是由于存在垃圾收集机制，所以 Java 没有破坏器。然而，随着以后学习的深入，就会知道垃圾收集器的存在并不能完全消除对破坏器的需要，或者说不能消除对破坏器代表的那种机制的需要（而且绝对不能直接调用 finalize()，所以应尽量避免用它）。若希望执行除释放存储空间之外的其他某种形式的清除工作，仍然必须调用 Java 中的

一个方法。它等价于 C++的破坏器，只是没后者方便。

finalize()最有用处的地方之一是观察垃圾收集的过程。下面这个例子向大家展示了垃圾收集所经历的过程，并对前面的陈述进行了总结。

165-166 页程序

```
//: c04:Garbage.java
// Demonstration of the garbage
// collector and finalization

class Chair {
  static boolean gcrun = false;
  static boolean f = false;
  static int created = 0;
  static int finalized = 0;
  int i;
  Chair() {
    i = ++created;
    if(created == 47)
      System.out.println("Created 47");
  }
  public void finalize() {
    if(!gcrun) {
      // The first time finalize() is called:
      gcrun = true;
      System.out.println(
        "Beginning to finalize after " +
        created + " Chairs have been created");
    }
    if(i == 47) {
      System.out.println(
        "Finalizing Chair #47, " +
        "Setting flag to stop Chair creation");
      f = true;
    }
    finalized++;
    if(finalized >= created)
      System.out.println(
        "All " + finalized + " finalized");
  }
}

public class Garbage {
  public static void main(String[] args) {
    // As long as the flag hasn't been set,
```

```
        // make Chairs and Strings:
        while(!Chair.f) {
          new Chair();
          new String("To take up space");
        }
        System.out.println(
          "After all Chairs have been created:\n" +
          "total created = " + Chair.created +
          ", total finalized = " + Chair.finalized);
        // Optional arguments force garbage
        // collection & finalization:
        if(args.length > 0) {
          if(args[0].equals("gc") ||
              args[0].equals("all")) {
            System.out.println("gc():");
            System.gc();
          }
          if(args[0].equals("finalize") ||
              args[0].equals("all")) {
            System.out.println("runFinalization():");
            System.runFinalization();
          }
        }
        System.out.println("bye!");
      }
    } ///:~
```

上面这个程序创建了许多 Chair 对象，而且在垃圾收集器开始运行后的某些时候，程序会停止创建 Chair。由于垃圾收集器可能在任何时间运行，所以我们不能准确知道它在何时启动。因此，程序用一个名为 gcrun 的标记来指出垃圾收集器是否已经开始运行。利用第二个标记 f，Chair 可告诉 main()它应停止对象的生成。这两个标记都是在 finalize()内部设置的，它调用于垃圾收集期间。

另两个 static 变量——created 以及 finalized——分别用于跟踪已创建的对象数量以及垃圾收集器已进行完收尾工作的对象数量。最后，每个 Chair 都有它自己的（非 static）int i，所以能跟踪了解它具体的编号是多少。编号为 47 的 Chair 进行完收尾工作后，标记会设为 true，最终结束 Chair 对象的创建过程。

所有这些都在 main()的内部进行——在下面这个循环里：

```
while(!Chair.f) {
new Chair();
new String("To take up space");
}
```

大家可能会疑惑这个循环什么时候会停下来，因为内部没有任何改变 Chair.f

值的语句。然而，finalize()进程会改变这个值，直至最终对编号 47 的对象进行收尾处理。

每次循环过程中创建的 String 对象只是属于额外的垃圾，用于吸引垃圾收集器——一旦垃圾收集器对可用内存的容量感到“紧张不安”，就会开始关注它。

运行这个程序的时候，提供了一个命令行自变量“before”或者“after”。其中，“before”自变量会调用 System.gc()方法（强制执行垃圾收集器），同时还会调用 **System.runFinalization()**方法，以便进行收尾工作。这些方法都可在 Java 1.0 中使用，但通过使用“after”自变量而调用的 runFinalizersOnExit()方法却只有 Java 1.1 及后续版本提供了对它的支持（注释③）。注意可在程序执行的任何时候调用这个方法，而且收尾程序的执行与垃圾收集器是否运行是无关的。

③：不幸的是，Java 1.0 采用的垃圾收集器方案永远不能正确地调用 finalize()。因此，finalize()方法（特别是那些用于关闭文件的）事实上经常都不会得到调用。现在有些文章声称所有收尾模块都会在程序退出的时候得到调用——即使到程序中止的时候，垃圾收集器仍未针对那些对象采取行动。这并不是真实的情况，所以我们根本不能指望 finalize()能为所有对象而调用。特别地，finalize()在 Java 1.0 里几乎毫无用处。

前面的程序向我们揭示出：在 Java 1.1 中，收尾模块肯定会运行这一许诺已成为现实——但前提是我们明确地强制它采取这一操作。若使用一个不是“before”或“after”的自变量（如“none”），那么两个收尾工作都不会进行，而且我们会得到象下面这样的输出：

```
Created 47
```

167-168 页程序

```
Beginning to finalize after 3486 Chairs have been created
Finalizing Chair #47, Setting flag to stop Chair creation
After all Chairs have been created:
total created = 3881, total finalized = 2684
bye!
```

因此，到程序结束的时候，并非所有收尾模块都会得到调用（注释④）。为强制进行收尾工作，可先调用 System.gc()，再调用 System.runFinalization()。这样可清除到目前为止没有使用的所有对象。这样做一个稍显奇怪的地方是在调用 runFinalization()之前调用 gc()，这看起来似乎与 Sun 公司的文档说明有些抵触，它宣称首先运行收尾模块，再释放存储空间。然而，若在这里首先调用 runFinalization()，再调用 gc()，收尾模块根本不会执行。

```
//: c04:DeathCondition.java
// Using finalize() to detect an object that
// hasn't been properly cleaned up.

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 DeathCondition {
    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();
    }
} ///:~
```

④：到你读到本书时，有些 Java 虚拟机（JVM）可能已开始表现出不同的行为。

针对所有对象，Java 1.1 有时之所以会默认为跳过收尾工作，是由于它认为这样做的开销太大。不管用哪种方法强制进行垃圾收集，都可能注意到比没有额外收尾工作时较长的时间延迟。

4.4 成员初始化

Java 尽自己的全力保证所有变量都能在使用前得到正确的初始化。若被定义成相对于一个方法的“局部”变量，这一保证就通过编译期的出错提示表现出来。因此，如果使用下述代码：

```
void f() {
int i;
i++;
}
```

就会收到一条出错提示消息，告诉你 i 可能尚未初始化。当然，编译器也可为 i 赋予一个默认值，但它看起来更象一个程序员的失误，此时默认值反而会“帮倒忙”。若强迫程序员提供一个初始值，就往往能够帮他／她纠出程序里的“臭虫”。

然而，若将基本类型（主类型）设为一个类的数据成员，情况就会变得稍微

有些不同。由于任何方法都可以初始化或使用那个数据，所以在正式使用数据前，若还是强迫程序员将其初始化成一个适当的值，就可能不是一种实际的做法。然而，若为其赋予一个垃圾值，同样是非常不安全的。因此，一个类的所有基本类型数据成员都会保证获得一个初始值。可用下面这段小程序看到这些值：

169 页程序

```
//: c04:InitialValues.java
// Shows default initial values.

class Measurement {
  boolean t;
  char c;
  byte b;
  short s;
  int i;
  long l;
  float f;
  double d;
  void print() {
    System.out.println(
      "Data type      Initial value\n" +
      "boolean        " + t + "\n" +
      "char           " + c + "\n" +
      "byte           " + b + "\n" +
      "short          " + s + "\n" +
      "int            " + i + "\n" +
      "long           " + l + "\n" +
      "float          " + f + "\n" +
      "double         " + d);
  }
}

public class InitialValues {
  public static void main(String[] args) {
    Measurement d = new Measurement();
    d.print();
    /* In this case you could also say:
    new Measurement().print();
    */
  }
} ///:~
```

输入结果如下：

170 页上程序

Data type	Initial value
boolean	false
char	
byte	0
short	0
int	0
long	0
float	0.0
double	0.0

其中，Char 值为空（NULL），没有数据打印出来。

稍后大家就会看到：在一个类的内部定义一个对象句柄时，如果不将其初始化成新对象，那个句柄就会获得一个空值。

4.4.1 规定初始化

如果想自己为变量赋予一个初始值，又会发生什么情况呢？为达到这个目的，一个最直接的做法是在类内部定义变量的同时也为其赋值（注意在 C++里不能这样做，尽管 C++的新手们总“想”这样做）。在下面，Measurement 类内部的字段定义已发生了变化，提供了初始值：

170 页下程序

```
class Measurement {
  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;
  //. . .
```

亦可用相同的方法初始化非基本（主）类型的对象。若 Depth 是一个类，那么可象下面这样插入一个变量并进行初始化：

```
class Measurement {
Depth o = new Depth();
boolean b = true;
// . . .
```

若尚未为 o 指定一个初始值，同时不顾一切地提前试用它，就会得到一条运行期错误提示，告诉你产生了名为异常 Exception）的一个错误（在第 9 章详述）。

甚至可通过调用一个方法来提供初始值：

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

当然，这个方法亦可使用自变量，但那些自变量不可是尚未初始化的其他类成员。因此，下面这样做是合法的：

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

但下面这样做是非法的：

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

这正是编译器对“向前引用”感到不适应的一个地方，因为它与初始化的顺序有关，而不是与程序的编译方式有关。

这种初始化方法非常简单和直观。它的一个限制是类型 Measurement 的每个对象都会获得相同的初始化值。有时，这正是我们希望的结果，但有时却需要盼望更大的灵活性。

4.4.2 构建器初始化

可考虑用构建器执行初始化进程。这样便可在编程时获得更大的灵活程度，因为我们可以在运行期调用方法和采取行动，从而“现场”决定初始化值。但要注意这样一件事情：不可妨碍自动初始化的进行，它在构建器进入之前就会发生。因此，假如使用下述代码：

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

那么 i 首先会初始化成零，然后变成 7。对于所有基本类型以及对象句柄，这种情况都是成立的，其中包括在定义时已进行了明确初始化的那些一些。考虑到这个原因，编译器不会试着强迫我们在构建器任何特定的场所对元素进行初始化，或者在它们使用之前——初始化早已得到了保证（注释⑤）。

⑤：相反，C++有自己的“构建器初始模块列表”，能在进入构建器主体之前进行初始化，而且它对于对象来说是强制进行的。参见《Thinking in C++》。

1. 初始化顺序

在一个类里，初始化的顺序是由变量在类内的定义顺序决定的。即使变量定义大量遍布于方法定义的中间，那些变量仍会在调用任何方法之前得到初始化——甚至在构建器调用之前。例如：

172-173 页程序

```
//: c04:OrderOfInitialization.java
// Demonstrates initialization order.

// 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 {
  public static void main(String[] args) {
    Card t = new Card();
    t.f(); // Shows that construction is done
  }
} ///:~
```

在 Card 中，Tag 对象的定义故意到处散布，以证明它们全都会在构建器进入或者发生其他任何事情之前得到初始化。除此之外，t3 在构建器内部得到了重新

初始化。它的输入结果如下：

173 页中程序

```
Tag(1)
Tag(2)
Tag(3)
Card()
Tag(33)
f()
```

因此，t3 句柄会被初始化两次，一次在构建器调用前，一次在调用期间（第一个对象会被丢弃，所以它后来可被当作垃圾收掉）。从表面看，这样做似乎效率低下，但它能保证正确的初始化——若定义了一个过载的构建器，它没有初始化 t3；同时在 t3 的定义里并没有规定“默认”的初始化方式，那么会产生什么后果呢？

2. 静态数据的初始化

若数据是静态的（**static**），那么同样的事情就会发生；如果它属于一个基本类型（主类型），而且未对其初始化，就会自动获得自己的标准基本类型初始值；如果它是指向一个对象的句柄，那么除非新建一个对象，并将句柄同它连接起来，否则就会得到一个空值（NULL）。

如果想在定义的同时进行初始化，采取的方法与非静态值表面看起来是相同的。但由于 static 值只有一个存储区域，所以无论创建多少个对象，都必然会遇到何时对那个存储区域进行初始化的问题。下面这个例子可将这个问题说更清楚一些：

174-175 页程序

```
//: c04:StaticInitialization.java
// Specifying initial values in a
// class definition.

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 {
    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);
    }
    static Table t2 = new Table();
    static Cupboard t3 = new Cupboard();
} ///:~
```

Bowl 允许我们检查一个类的创建过程，而 Table 和 Cupboard 能创建散布于类定义中的 Bowl 的 static 成员。注意在 static 定义之前，Cupboard 先创建了一个非 static 的 Bowl b3。它的输出结果如下：

175 页中程序

```
    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 初始化只有在必要的时候才会进行。如果不创建一个 Table 对象，而且永远都不引用 Table.b1 或 Table.b2，那么 static Bowl b1 和 b2 永远都不会创建。然而，只有在创建了第一个 Table 对象之后（或者发生了第一次 static 访问），它们才会创建。在那以后，static 对象不会重新初始化。

初始化的顺序是首先 static（如果它们尚未由前一次对象创建过程初始化），接着是非 static 对象。大家可从输出结果中找到相应的证据。

在这里有必要总结一下对象的创建过程。请考虑一个名为 Dog 的类：

(1) 类型为 Dog 的一个对象首次创建时，或者 Dog 类的 static 方法 / static 字段首次访问时，Java 解释器必须找到 Dog.class（在事先设好的类路径里搜索）。

(2) 找到 Dog.class 后（它会创建一个 Class 对象，这将在后面学到），它的所有 static 初始化模块都会运行。因此，static 初始化仅发生一次——在 Class 对象首次载入的时候。

(3) 创建一个 new Dog()时，Dog 对象的构建进程首先会在内存堆（Heap）里为一个 Dog 对象分配足够多的存储空间。

(4) 这种存储空间会清为零，将 Dog 中的所有基本类型设为它们的默认值（零用于数字，以及 boolean 和 char 的等价设定）。

(5) 进行字段定义时发生的所有初始化都会执行。

(6) 执行构建器。正如第 6 章将要讲到的那样，这实际可能要求进行相当多的操作，特别是在涉及继承的时候。

3. 明确进行的静态初始化

Java 允许我们将其他 static 初始化工作划分到类内一个特殊的“static 构建从句”（有时也叫作“静态块”）里。它看起来象下面这个样子：

176 页下程序

```
class Spoon {
  static int i;
```

```
  static {
    i = 47;
  }
  // . . .
```

尽管看起来象个方法，但它实际只是一个 static 关键字，后面跟随一个方法主体。与其他 static 初始化一样，这段代码仅执行一次——首次生成那个类的一个对象时，或者首次访问属于那个类的一个 static 成员时（即便从未生成过那个类的对象）。例如：

176-177 页程序

```
//: c04:ExplicitStatic.java
// Explicit static initialization
// with the "static" clause.

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

在标记为(1)的行内访问 static 对象 c1 的时候，或在行(1)标记为注释，同时(2)行不标记成注释的时候，用于 Cups 的 static 初始化模块就会运行。若(1)和(2)都被标记成注释，则用于 Cups 的 static 初始化进程永远不会发生。

4. 非静态实例的初始化

针对每个对象的非静态变量的初始化，Java 1.1 提供了一种类似的语法格式。下面是一个例子：

177-178 页程序

```
//: c04:Mugs.java
// Java "Instance Initialization."
class Mug {
  Mug(int marker) {
  System.out.println("Mug(" + marker + ")");
  }
  void f(int marker) {
          System.out.println("f(" + marker + ")");
  }
}
public class Mugs {
      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();
      }
} ///:~
```

大家可看到实例初始化从句：

178 页下程序

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

它看起来与静态初始化从句极其相似，只是 static 关键字从里面消失了。为支持对“匿名内部类”的初始化（参见第 7 章），必须采用这一语法格式。

4.5 数组初始化

在 C 中初始化数组极易出错，而且相当麻烦。C++通过“集合初始化”使其更安全（注释⑥）。Java 则没有象 C++那样的“集合”概念，因为 Java 中的所有东西都是对象。但它确实有自己的数组，通过数组初始化来提供支持。

数组代表一系列对象或者基本数据类型，所有相同的类型都封装到一起——采用一个统一的标识符名称。数组的定义和使用是通过方括号索引运算符进行的（[]）。为定义一个数组，只需在类型名后简单地跟随一对空方括号即可：

```
int[] al;
```

也可以将方括号置于标识符后面，获得完全一致的结果：

```
int al[];
```

这种格式与 C 和 C++程序员习惯的格式是一致的。然而，最“通顺”的也许还是前一种语法，因为它指出类型是“一个 int 数组”。本书将沿用那种格式。

编译器不允许我们告诉它一个数组有多大。这样便使我们回到了“句柄”的问题上。此时，我们拥有的一切就是指向数组的一个句柄，而且尚未给数组分配任何空间。为了给数组创建相应的存储空间，必须编写一个初始化表达式。对于数组，初始化工作可在代码的任何地方出现，但也可以使用一种特殊的初始化表达式，它必须在数组创建的地方出现。这种特殊的初始化是一系列由花括号封闭起来的值。存储空间的分配（等价于使用 new）将由编译器在这种情况下进行。例如：

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

那么为什么还要定义一个没有数组的数组句柄呢？

```
int[] a2;
```

事实上在 Java 中，可将一个数组分配给另一个，所以能使用下述语句：

```
a2 = a1;
```

我们真正准备做的是复制一个句柄，就象下面演示的那样：

180 页上程序

```
//: c04:Arrays.java
// Arrays of primitives.

public class Arrays {
  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]);
  }
} ///:~
```

大家看到 a1 获得了一个初始值，而 a2 没有；a2 将在以后赋值——这种情况下是赋给另一个数组。

这里也出现了一些新东西：所有数组都有一个本质成员（无论它们是对象数组还是基本类型数组），可对其进行查询——但不是改变，从而获知数组内包含了多少个元素。这个成员就是 length。与 C 和 C++类似，由于 Java 数组从元素 0 开始计数，所以能索引的最大元素编号是“length-1”。如超出边界，C 和 C++会“默默”地接受，并允许我们胡乱使用自己的内存，这正是许多程序错误的根源。然而，Java 可保留我们不受这一问题的损害，方法是一旦超过边界，就生成一个运行期错误（即一个“异常”，这是第 9 章的主题）。当然，由于需要检查每个数组的访问，所以会消耗一定的时间和多余的代码量，而且没有办法把它关闭。这意味着数组访问可能成为程序效率低下的重要原因——如果它们在关键的场合进行。但考虑到因特网访问的安全，以及程序员的编程效率，Java 设计人员还是应该把它看作是值得的。

程序编写期间，如果不知道在自己的数组里需要多少元素，那么又该怎么办呢？此时，只需简单地用 new 在数组里创建元素。在这里，即使准备创建的是一个基本数据类型的数组，new 也能正常地工作（new 不会创建非数组的基本类型）：

180-181 页程序

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

public class ArrayNew {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args) {
    int[] a;
    a = new int[pRand(20)];
    System.out.println(
      "length of a = " + a.length);
    for(int i = 0; i < a.length; i++)
      System.out.println(
        "a[" + i + "] = " + a[i]);
  }
} ///:~
```

由于数组的大小是随机决定的（使用早先定义的 pRand()方法），所以非常明显，数组的创建实际是在运行期间进行的。除此以外，从这个程序的输出中，大家可看到基本数据类型的数组元素会自动初始化成“空”值（对于数值，空值就是零；对于 char，它是 null；而对于 boolean，它却是 false）。

当然，数组可能已在相同的语句中定义和初始化了，如下所示：

```
int[] a = new int[pRand(20)];
```

若操作的是一个非基本类型对象的数组，那么无论如何都要使用 new。在这里，我们会再一次遇到句柄问题，因为我们创建的是一个句柄数组。请大家观察封装器类型 Integer，它是一个类，而非基本数据类型：

181-182 页程序

```
//: c04:ArrayClassObj.java
// Creating an array of nonprimitive objects.
import java.util.*;
public class ArrayClassObj {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;  }
  public static void main(String[] args) {
    Integer[] a = new Integer[pRand(20)];
    System.out.println(
      "length of a = " + a.length);
      for(int i = 0; i < a.length; i++) {
        a[i] = new Integer(pRand(500));
        System.out.println(
        "a[" + i + "] = " + a[i]);
      }
  }
} ///:~
```

在这儿，甚至在 new 调用后才开始创建数组：

```
Integer[] a = new Integer[pRand(20)];
```

它只是一个句柄数组，而且除非通过创建一个新的 Integer 对象，从而初始化了对象句柄，否则初始化进程不会结束：

```
a[i] = new Integer(pRand(500));
```

但若忘记创建对象，就会在运行期试图读取空数组位置时获得一个“异常”错误。

下面让我们看看打印语句中 String 对象的构成情况。大家可看到指向 Integer 对象的句柄会自动转换，从而产生一个 String，它代表着位于对象内部的值。

亦可用花括号封闭列表来初始化对象数组。可采用两种形式，第一种是 Java 1.0 允许的唯一形式。第二种（等价）形式自 Java 1.1 才开始提供支持：

182-183 页程序

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

这种做法大多数时候都很有用，但限制也是最大的，因为数组的大小是在编译期间决定的。初始化列表的最后一个逗号是可选的（这一特性使长列表的维护变得更加容易）。

数组初始化的第二种形式（Java 1.1 开始支持）提供了一种更简便的语法，可创建和调用方法，获得与 C 的“变量参数列表”（C 通常把它简称为“变参表”）一致的效果。这些效果包括未知的参数（自变量）数量以及未知的类型（如果这样选择的话）。由于所有类最终都是从通用的根类 Object 中继承的，所以能创建一个方法，令其获取一个 Object 数组，并象下面这样调用它：

183 页中程序

```
//: c04:VarArgs.java
// Using the array syntax to create
// variable argument lists.

class A { int i; }

public class VarArgs {
    static void f(Object[] x) {
        for(int i = 0; i < x.length; i++)
            System.out.println(x[i]);
    }
    public static void main(String[] args) {
        f(new Object[] {
                new Integer(47), new VarArgs(),
                new Float(3.14), new Double(11.11) });
        f(new Object[] {"one", "two", "three" });
        f(new Object[] {new A(), new A(), new A()});
    }
} ///:~
```

此时，我们对这些未知的对象并不能采取太多的操作，而且这个程序利用自动 String 转换对每个 Object 做一些有用的事情。在第 11 章（运行期类型标识或

RTTI），大家还会学习如何调查这类对象的准确类型，使自己能对它们做一些有趣的事情。

4.5.1 多维数组

在 Java 里可以方便地创建多维数组：

184-185 页程序

```
//: c04:MultiDimArray.java
// Creating multidimensional arrays.
import java.util.*;
public class MultiDimArray {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
   }
  static void prt(String s) {
    System.out.println(s);
  }
  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++)
        prt("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++)
          prt("a2[" + i + "][" + j + "][" + k + "] = " + a2[i][j][k]);
    // 3-D array with varied-length vectors:
    int[][][] a3 = new int[pRand(7)][][];
    for(int i = 0; i < a3.length; i++) {
      a3[i] = new int[pRand(5)][];
      for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pRand(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++)
          prt("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++)
          prt("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++)
          prt("a5[" + i + "][" + j +"] = " + a5[i][j]);
    }
  } ///:~
```

用于打印的代码里使用了 length，所以它不必依赖固定的数组大小。

第一个例子展示了基本数据类型的一个多维数组。我们可用花括号定出数组内每个矢量的边界：

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

每个方括号对都将我们移至数组的下一级。

第二个例子展示了用 new 分配的一个三维数组。在这里，整个数组都是立即分配的：

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

但第三个例子却向大家揭示出构成矩阵的每个矢量都可以有任意的长度：

186 页上程序

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

对于第一个 new 创建的数组，它的第一个元素的长度是随机的，其他元素的

长度则没有定义。for 循环内的第二个 new 则会填写元素，但保持第三个索引的未定状态——直到碰到第三个 new。

根据输出结果，大家可以看到：假若没有明确指定初始化值，数组值就会自动初始化成零。

可用类似的表式处理非基本类型对象的数组。这从第四个例子可以看出，它向我们演示了用花括号收集多个 new 表达式的能力：

186 页中程序

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

第五个例子展示了如何逐渐构建非基本类型的对象数组：

186 页下程序

```
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);
    }
```

i*j 只是在 Integer 里置了一个有趣的值。

4.6 总结

作为初始化的一种具体操作形式，构建器应使大家明确感受到在语言中进行初始化的重要性。与 C++的程序设计一样，判断一个程序效率如何，关键是看是否由于变量的初始化不正确而造成了严重的编程错误（臭虫）。这些形式的错误很难发现，而且类似的问题也适用于不正确的清除或收尾工作。由于构建器使我们能保证正确的初始化和清除（若没有正确的构建器调用，编译器不允许对象创建），所以能获得完全的控制权和安全性。

在 C++中，与“构建”相反的“破坏”（Destruction）工作也是相当重要的，因为用 new 创建的对象必须明确地清除。在 Java 中，垃圾收集器会自动为所有对象释放内存，所以 Java 中等价的清除方法并不是经常都需要用到的。如果不需要类似于构建器的行为，Java 的垃圾收集器可以极大简化编程工作，而且在内存的管理过程中增加更大的安全性。有些垃圾收集器甚至能清除其他资源，比如图形和文件句柄等。然而，垃圾收集器确实也增加了运行期的开销。但这种开销到底造成了多大的影响却是很难看出的，因为到目前为止，Java 解释器的总体运行速度仍然是比较慢的。随着这一情况的改观，我们应该能判断出垃圾收集器的

开销是否使 Java 不适合做一些特定的工作（其中一个问题是垃圾收集器不可预测的性质）。

由于所有对象都肯定能获得正确的构建，所以同这儿讲述的情况相比，构建器实际做的事情还要多得多。特别地，当我们通过“创作”或“继承”生成新类的时候，对构建的保证仍然有效，而且需要一些附加的语法来提供对它的支持。大家将在以后的章节里详细了解创作、继承以及它们对构建器造成的影响。

4.7 练习

(1) 用默认构建器创建一个类（没有自变量），用它打印一条消息。创建属于这个类的一个对象。

(2) 在练习 1 的基础上增加一个过载的构建器，令其采用一个 String 自变量，并随同自己的消息打印出来。

(3) 以练习 2 创建的类为基础上，创建属于它的对象句柄的一个数组，但不要实际创建对象并分配到数组里。运行程序时，注意是否打印出来自构建器调用的初始化消息。

(4) 创建同句柄数组联系起来的对象，最终完成练习 3。

(5) 用自变量“before”，“after”和“none”运行程序，试验 Garbage.java。重复这个操作，观察是否从输出中看出了一些固定的模式。改变代码，使 System.runFinalization()在 System.gc()之前调用，再观察结果。