第 2 章 一切都是对象

“尽管以 C++为基础，但 Java 是一种更纯粹的面向对象程序设计语言”。

无论 C++还是 Java 都属于杂合语言。但在 Java 中，设计者觉得这种杂合并不象在 C++里那么重要。杂合语言允许采用多种编程风格；之所以说 C++是一种杂合语言，是因为它支持与 C 语言的向后兼容能力。由于 C++是 C 的一个超集，所以包含的许多特性都是后者不具备的，这些特性使 C++在某些地方显得过于复杂。

Java 语言首先便假定了我们只希望进行面向对象的程序设计。也就是说，正式用它设计之前，必须先将自己的思想转入一个面向对象的世界（除非早已习惯了这个世界的思维方式）。只有做好这个准备工作，与其他 OOP 语言相比，才能体会到 Java 的易学易用。在本章，我们将探讨 Java 程序的基本组件，并体会为什么说 Java 乃至 Java 程序内的一切都是对象。

2.1 用句柄操纵对象

每种编程语言都有自己的数据处理方式。有些时候，程序员必须时刻留意准备处理的是什么类型。您曾利用一些特殊语法直接操作过对象，或处理过一些间接表示的对象吗（C 或 C++里的指针）？

所有这些在 Java 里都得到了简化，任何东西都可看作对象。因此，我们可采用一种统一的语法，任何地方均可照搬不误。但要注意，尽管将一切都“看作”对象，但操纵的标识符实际是指向一个对象的“句柄”（Handle）。在其他 Java 参考书里，还可看到有的人将其称作一个“引用”，甚至一个“指针”。可将这一情形想象成用遥控板（句柄）操纵电视机（对象）。只要握住这个遥控板，就相当于掌握了与电视机连接的通道。但一旦需要“换频道”或者“关小声音”，我们实际操纵的是遥控板（句柄），再由遥控板自己操纵电视机（对象）。如果要在房间里四处走走，并想保持对电视机的控制，那么手上拿着的是遥控板，而非电

视机。

此外，即使没有电视机，遥控板亦可独立存在。也就是说，只是由于拥有一个句柄，并不表示必须有一个对象同它连接。所以如果想容纳一个词或句子，可创建一个 String 句柄：

```
String s;
```

但这里创建的只是句柄，并不是对象。若此时向 s 发送一条消息，就会获得一个错误（运行期）。这是由于 s 实际并未与任何东西连接（即“没有电视机”）。因此，一种更安全的做法是：创建一个句柄时，记住无论如何都进行初始化：

```
String s = "asdf";
```

然而，这里采用的是一种特殊类型：字串可用加引号的文字初始化。通常，必须为对象使用一种更通用的初始化类型。

2.2 所有对象都必须创建

创建句柄时，我们希望它同一个新对象连接。通常用 new 关键字达到这一目的。new 的意思是：“把我变成这些对象的一种新类型”。所以在上面的例子中，可以说：

```
String s = new String("asdf");
```

它不仅指出“将我变成一个新字串”，也通过提供一个初始字串，指出了“如何生成这个新字串”。

当然，字串（String）并非唯一的类型。Java 配套提供了数量众多的现成类型。对我们来讲，最重要的就是记住能自行创建类型。事实上，这应是 Java 程序设计的一项基本操作，是继续本书后余部分学习的基础。

2.2.1 保存到什么地方

程序运行时，我们最好对数据保存到什么地方做到心中有数。特别要注意的是内存的分配。有六个地方都可以保存数据：

(1) 寄存器。这是最快的保存区域，因为它位于和其他所有保存方式不同的地方：处理器内部。然而，寄存器的数量十分有限，所以寄存器是根据需要由编译器分配。我们对此没有直接的控制权，也不可能在自己的程序里找到寄存器存在的任何踪迹。

(2) 堆栈。驻留于常规 RAM（随机访问存储器）区域，但可通过它的“堆栈指针”获得处理的直接支持。堆栈指针若向下移，会创建新的内存；若向上移，则会释放那些内存。这是一种特别快、特别有效的数据保存方式，仅次于寄存器。创建程序时，Java 编译器必须准确地知道堆栈内保存的所有数据的“长度”以及“存在时间”。这是由于它必须生成相应的代码，以便向上和向下移动指针。这一限制无疑影响了程序的灵活性，所以尽管有些 Java 数据要保存在堆栈里——特别是对象句柄，但 Java 对象并不放到其中。

(3) 堆。一种常规用途的内存池（也在 RAM 区域），其中保存了 Java 对象。和堆栈不同，“内存堆”或“堆”（Heap）最吸引人的地方在于编译器不必知道要从堆里分配多少存储空间，也不必知道存储的数据要在堆里停留多长的时间。因此，用堆保存数据时会得到更大的灵活性。要求创建一个对象时，只需用 new 命令编制相关的代码即可。执行这些代码时，会在堆里自动进行数据的保存。当然，为达到这种灵活性，必然会付出一定的代价：在堆里分配存储空间时会花掉更长的时间！

(4) 静态存储。这儿的“静态”（Static）是指“位于固定位置”（尽管也在RAM 里）。程序运行期间，静态存储的数据将随时等候调用。可用 static 关键字指出一个对象的特定元素是静态的。但 Java 对象本身永远都不会置入静态存储空间。

(5) 常数存储。常数值通常直接置于程序代码内部。这样做是安全的，因为它们永远都不会改变。有的常数需要严格地保护，所以可考虑将它们置入只读存储器（ROM）。

(6) 非 RAM 存储。若数据完全独立于一个程序之外，则程序不运行时仍可存在，并在程序的控制范围之外。其中两个最主要的例子便是“流式对象”和“固定对象”。对于流式对象，对象会变成字节流，通常会发给另一台机器。而对于固定对象，对象保存在磁盘中。即使程序中止运行，它们仍可保持自己的状态不变。对于这些类型的数据存储，一个特别有用的技巧就是它们能存在于其他媒体中。一旦需要，甚至能将它们恢复成普通的、基于 RAM 的对象。Java 1.1 提供了对 Lightweight persistence 的支持。未来的版本甚至可能提供更完整的方案。

2.2.2 特殊情况：主要类型

有一系列类需特别对待；可将它们想象成“基本”、“主要”或者“主”（Primitive）类型，进行程序设计时要频繁用到它们。之所以要特别对待，是由于用 new 创建对象（特别是小的、简单的变量）并不是非常有效，因为 new 将对象置于“堆”里。对于这些类型，Java 采纳了与 C 和 C++相同的方法。也就是说，不是用 new 创建变量，而是创建一个并非句柄的“自动”变量。这个变量容纳了具体的值，并置于堆栈中，能够更高效地存取。

Java 决定了每种主要类型的大小。就象在大多数语言里那样，这些大小并不随着机器结构的变化而变化。这种大小的不可更改正是 Java 程序具有很强移植能力的原因之一。

主类型 大小 最小值 最大值 封装器类型

boolean 1 位 - - Boolean
char 16 位 Unicode 0 Unicode 2 的 16 次方-1 Character
byte 8 位 -128 +127 Byte（注释①）
short 16 位 -2 的 15 次方 +2 的 15 次方-1 Short（注释①）
int 32 位 -2 的 31 次方 +2 的 31 次方-1 Integer
long 64 位 -2 的 63 次方 +2 的 63 次方-1 Long
float 32 位 IEEE754 IEEE754 Float
double 64 位 IEEE754 IEEE754 Double
Void - - - Void（注释①）

①：到 Java 1.1 才有，1.0 版没有。

数值类型全都是有符号（正负号）的，所以不必费劲寻找没有符号的类型。

主数据类型也拥有自己的“封装器”（wrapper）类。这意味着假如想让堆内一个非主要对象表示那个主类型，就要使用对应的封装器。例如：

```
char c = 'x';
```

Character C = new Character('c');
也可以直接使用：
Character C = new Character('x');
这样做的原因将在以后的章节里解释。

1. 高精度数字

Java 1.1 增加了两个类，用于进行高精度的计算：BigInteger 和 BigDecimal。尽管它们大致可以划分为“封装器”类型，但两者都没有对应的“主类型”。

这两个类都有自己特殊的“方法”，对应于我们针对主类型执行的操作。也就是说，能对 int 或 float 做的事情，对 BigInteger 和 BigDecimal 一样可以做。只是必须使用方法调用，不能使用运算符。此外，由于牵涉更多，所以运算速度会慢一些。我们牺牲了速度，但换来了精度。

BigInteger 支持任意精度的整数。也就是说，我们可精确表示任意大小的整数值，同时在运算过程中不会丢失任何信息。

BigDecimal 支持任意精度的定点数字。例如，可用它进行精确的币值计算。

至于调用这两个类时可选用的构建器和方法，请自行参考联机帮助文档。

2.2.3 Java 的数组

几乎所有程序设计语言都支持数组。在 C 和 C++里使用数组是非常危险的，因为那些数组只是内存块。若程序访问自己内存块以外的数组，或者在初始化之前使用内存（属于常规编程错误），会产生不可预测的后果（注释②）。

②：在 C++里，应尽量不要使用数组，换用标准模板库（Standard TemplateLibrary）里更安全的容器。

Java 的一项主要设计目标就是安全性。所以在 C 和 C++里困扰程序员的许多问题都未在 Java 里重复。一个 Java 可以保证被初始化，而且不可在它的范围之外访问。由于系统自动进行范围检查，所以必然要付出一些代价：针对每个数组，以及在运行期间对索引的校验，都会造成少量的内存开销。但由此换回的是更高的安全性，以及更高的工作效率。为此付出少许代价是值得的。

创建对象数组时，实际创建的是一个句柄数组。而且每个句柄都会自动初始化成一个特殊值，并带有自己的关键字：null（空）。一旦 Java 看到 null，就知道该句柄并未指向一个对象。正式使用前，必须为每个句柄都分配一个对象。若试图使用依然为 null 的一个句柄，就会在运行期报告问题。因此，典型的数组错误在 Java 里就得到了避免。

也可以创建主类型数组。同样地，编译器能够担保对它的初始化，因为会将那个数组的内存划分成零。

数组问题将在以后的章节里详细讨论。

2.3 绝对不要清除对象

在大多数程序设计语言中，变量的“存在时间”（Lifetime）一直是程序员需要着重考虑的问题。变量应持续多长的时间？如果想清除它，那么何时进行？在变量存在时间上纠缠不清会造成大量的程序错误。在下面的小节里，将阐示 Java 如何帮助我们完成所有清除工作，从而极大了简化了这个问题。

2.3.1 作用域

大多数程序设计语言都提供了“作用域”（Scope）的概念。对于在作用域里定义的名字，作用域同时决定了它的“可见性”以及“存在时间”。在 C，C++ 和 Java 里，作用域是由花括号的位置决定的。参考下面这个例子：

```
{
   int x = 12;
   /* only x available */
    {
        int q = 96;
        /* both x & q available */
    }
    /* only x available */
    /* q “out of scope” */
}
```

73 页程序

作为在作用域里定义的一个变量，它只有在那个作用域结束之前才可使用。

在上面的例子中，缩进排版使 Java 代码更易辨读。由于 Java 是一种形式自由的语言，所以额外的空格、制表位以及回车都不会对结果程序造成影响。

注意尽管在 C 和 C++里是合法的，但在 Java 里不能象下面这样书写代码：

```
{
   int x = 12;
   {
       int x = 96; /* illegal */
   }
}
```

74 页上程序

编译器会认为变量 x 已被定义。所以 C 和 C++能将一个变量“隐藏”在一个更大的作用域里。但这种做法在 Java 里是不允许的，因为 Java 的设计者认为这样做使程序产生了混淆。

2.3.2 对象的作用域

Java 对象不具备与主类型一样的存在时间。用 new 关键字创建一个 Java 对象的时候，它会超出作用域的范围之外。所以假若使用下面这段代码：

```
{
String s = new String("a string");
} /*  作用域的终点  */
```

那么句柄 s 会在作用域的终点处消失。然而，s 指向的 String 对象依然占据

着内存空间。在上面这段代码里，我们没有办法访问对象，因为指向它的唯一一个句柄已超出了作用域的边界。在后面的章节里，大家还会继续学习如何在程序运行期间传递和复制对象句柄。

这样造成的结果便是：对于用 new 创建的对象，只要我们愿意，它们就会一直保留下去。这个编程问题在 C 和 C++里特别突出。看来在 C++里遇到的麻烦最大：由于不能从语言获得任何帮助，所以在需要对象的时候，根本无法确定它们是否可用。而且更麻烦的是，在 C++里，一旦工作完成，必须保证将对象清除。

这样便带来了一个有趣的问题。假如 Java 让对象依然故我，怎样才能防止它们大量充斥内存，并最终造成程序的“凝固”呢。在 C++里，这个问题最令程序员头痛。但 Java 以后，情况却发生了改观。Java 有一个特别的“垃圾收集器”，它会查找用 new 创建的所有对象，并辨别其中哪些不再被引用。随后，它会自动释放由那些闲置对象占据的内存，以便能由新对象使用。这意味着我们根本不必操心内存的回收问题。只需简单地创建对象，一旦不再需要它们，它们就会自动离去。这样做可防止在 C++里很常见的一个编程问题：由于程序员忘记释放内存造成的“内存溢出”。

2.4 新建数据类型：类

如果说一切东西都是对象，那么用什么决定一个“类”（Class）的外观与行为呢？换句话说，是什么建立起了一个对象的“类型”（Type）呢？大家可能猜想有一个名为“type”的关键字。但从历史看来，大多数面向对象的语言都用关键字“class”表达这样一个意思：“我准备告诉你对象一种新类型的外观”。class 关键字太常用了，以至于本书许多地方并没有用粗体字或双引号加以强调。在这个关键字的后面，应该跟随新数据类型的名称。例如：

```
class ATypeName {/*类主体置于这里}
```

这样就引入了一种新类型，接下来便可用 new 创建这种类型的一个新对象：

```
ATypeName a = new ATypeName();
```

在 ATypeName 里，类主体只由一条注释构成（星号和斜杠以及其中的内容，本章后面还会详细讲述），所以并不能对它做太多的事情。事实上，除非为其定义了某些方法，否则根本不能指示它做任何事情。

2.4.1 字段和方法

定义一个类时（我们在 Java 里的全部工作就是定义类、制作那些类的对象以及将消息发给那些对象），可在自己的类里设置两种类型的元素：数据成员（有时也叫“字段”）以及成员函数（通常叫“方法”）。其中，数据成员是一种对象（通过它的句柄与其通信），可以为任何类型。它也可以是主类型（并不是句柄）之一。如果是指向对象的一个句柄，则必须初始化那个句柄，用一种名为“构建器”（第 4 章会对此详述）的特殊函数将其与一个实际对象连接起来（就象早先看到的那样，使用 new 关键字）。但若是一种主类型，则可在类定义位置直接初始化（正如后面会看到的那样，句柄亦可在定义位置初始化）。

每个对象都为自己的数据成员保有存储空间；数据成员不会在对象之间共享。下面是定义了一些数据成员的类示例：

```
class DataOnly {
        int i;
```

```
        float f;
        boolean b;
    }
```

76 页上程序

这个类并没有做任何实质性的事情，但我们可创建一个对象：

```
DataOnly d = new DataOnly();
```

可将值赋给数据成员，但首先必须知道如何引用一个对象的成员。为达到引用对象成员的目的，首先要写上对象句柄的名字，再跟随一个点号（句点），再跟随对象内部成员的名字。即“对象句柄.成员”。例如：

```
d.i = 47;
d.f = 1.1f;
d.b = false;
```

一个对象也可能包含了另一个对象，而另一个对象里则包含了我们想修改的数据。对于这个问题，只需保持“连接句点”即可。例如：

```
myPlane.leftTank.capacity = 100;
```

除容纳数据之外，DataOnly 类再也不能做更多的事情，因为它没有成员函数（方法）。为正确理解工作原理，首先必须知道“自变量”和“返回值”的概念。我们马上就会详加解释。

1. 主成员的默认值

若某个主数据类型属于一个类成员，那么即使不明确（显式）进行初始化，也可以保证它们获得一个默认值。

主类型 默认值

```
Boolean false
Char '\u0000'(null)
byte (byte)0
short (short)0
int 0
long 0L
float 0.0f
double 0.0d
```

一旦将变量作为类成员使用，就要特别注意由 Java 分配的默认值。这样做可保证主类型的成员变量肯定得到了初始化（C++不具备这一功能），可有效遏止多种相关的编程错误。

然而，这种保证却并不适用于“局部”变量——那些变量并非一个类的字段。所以，假若在一个函数定义中写入下述代码：

```
int x;
```

那么 x 会得到一些随机值（这与 C 和 C++是一样的），不会自动初始化成零。我们责任是在正式使用 x 前分配一个适当的值。如果忘记，就会得到一条编译期

错误，告诉我们变量可能尚未初始化。这种处理正是 Java 优于 C++的表现之一。许多 C++编译器会对变量未初始化发出警告，但在 Java 里却是错误。

2.5 方法、自变量和返回值

迄今为止，我们一直用“函数”（Function）这个词指代一个已命名的子例程。但在 Java 里，更常用的一个词却是“方法”（Method），代表“完成某事的途径”。尽管它们表达的实际是同一个意思，但从现在开始，本书将一直使用“方法”，而不是“函数”。

Java 的“方法”决定了一个对象能够接收的消息。通过本节的学习，大家会知道方法的定义有多么简单！

方法的基本组成部分包括名字、自变量、返回类型以及主体。下面便是它最基本的形式：

返回类型 方法名(/* 自变量列表*/) {/* 方法主体 */}

返回类型是指调用方法之后返回的数值类型。显然，方法名的作用是对具体的方法进行标识和引用。自变量列表列出了想传递给方法的信息类型和名称。

Java 的方法只能作为类的一部分创建。只能针对某个对象调用一个方法（注释③），而且那个对象必须能够执行那个方法调用。若试图为一个对象调用错误的方法，就会在编译期得到一条出错消息。为一个对象调用方法时，需要先列出对象的名字，在后面跟上一个句点，再跟上方法名以及它的参数列表。亦即“对象名.方法名(自变量 1，自变量 2，自变量 3...)。举个例子来说，假设我们有一个方法名叫 f()，它没有自变量，返回的是类型为 int 的一个值。那么，假设有一个名为 a 的对象，可为其调用方法 f()，则代码如下：

```
int x = a.f();
```

返回值的类型必须兼容 x 的类型。

象这样调用一个方法的行动通常叫作“向对象发送一条消息”。在上面的例子中，消息是 f()，而对象是 a。面向对象的程序设计通常简单地归纳为“向对象发送消息”。

③：正如马上就要学到的那样，“静态”方法可针对类调用，毋需一个对象。

2.5.1 自变量列表

自变量列表规定了我们传送给方法的是什么信息。正如大家或许已猜到的那样，这些信息——如同 Java 内其他任何东西——采用的都是对象的形式。因此，我们必须在自变量列表里指定要传递的对象类型，以及每个对象的名字。正如在 Java 其他地方处理对象时一样，我们实际传递的是“句柄”（注释④）。然而，句柄的类型必须正确。倘若希望自变量是一个“字串”，那么传递的必须是一个字串。

④：对于前面提及的“特殊”数据类型 boolean，char，byte，short，int，long，，float 以及 double 来说是一个例外。但在传递对象时，通常都是指传递指向对象的句柄。

下面让我们考虑将一个字串作为自变量使用的方法。下面列出的是定义代码，必须将它置于一个类定义里，否则无法编译：

```
int storage(String s) {
return s.length() * 2;
}
```

这个方法告诉我们需要多少字节才能容纳一个特定字串里的信息（字串里的每个字符都是 16 位，或者说 2 个字节、长整数，以便提供对 Unicode 字符的支持）。自变量的类型为 String，而且叫作 s。一旦将 s 传递给方法，就可将它当作其他对象一样处理（可向其发送消息）。在这里，我们调用的是 length()方法，它是 String 的方法之一。该方法返回的是一个字串里的字符数。

通过上面的例子，也可以了解 return 关键字的运用。它主要做两件事情。首先，它意味着“离开方法，我已完工了”。其次，假设方法生成了一个值，则那个值紧接在 return 语句的后面。在这种情况下，返回值是通过计算表达式“s.length()*2”而产生的。

可按自己的愿望返回任意类型，但倘若不想返回任何东西，就可指示方法返回 void（空）。下面列出一些例子。

```
boolean flag() { return true; }
float naturalLogBase() { return 2.718; }
void nothing() { return; }
void nothing2() {}
```

若返回类型为 void，则 return 关键字唯一的作用就是退出方法。所以一旦抵达方法末尾，该关键字便不需要了。可在任何地方从一个方法返回。但假设已指定了一种非 void 的返回类型，那么无论从何地返回，编译器都会确保我们返回的是正确的类型。

到此为止，大家或许已得到了这样的一个印象：一个程序只是一系列对象的集合，它们的方法将其他对象作为自己的自变量使用，而且将消息发给那些对象。这种说法大体正确，但通过以后的学习，大家还会知道如何在一个方法里作出决策，做一些更细致的基层工作。至于这一章，只需理解消息传送就足够了。

2.6 构建 Java 程序

正式构建自己的第一个 Java 程序前，还有几个问题需要注意。

2.6.1 名字的可见性

在所有程序设计语言里，一个不可避免的问题是对名字或名称的控制。假设您在程序的某个模块里使用了一个名字，而另一名程序员在另一个模块里使用了相同的名字。此时，如何区分两个名字，并防止两个名字互相冲突呢？这个问题在 C 语言里特别突出。因为程序未提供很好的名字管理方法。C++的类（即 Java 类的基础）嵌套使用类里的函数，使其不至于同其他类里的嵌套函数名冲突。然而，C++仍然允许使用全局数据以及全局函数，所以仍然难以避免冲突。为解决这个问题，C++用额外的关键字引入了“命名空间”的概念。

由于采用全新的机制，所以 Java 能完全避免这些问题。为了给一个库生成明确的名字，采用了与 Internet 域名类似的名字。事实上，Java 的设计者鼓励程序员反转使用自己的 Internet 域名，因为它们肯定是独一无二的。由于我的域名是 BruceEckel.com，所以我的实用工具库就可命名为 com.bruceeckel.utility.foibles。反转了域名后，可将点号想象成子目录。

在 Java 1.0 和 Java 1.1 中，域扩展名 com，edu，org，net 等都约定为大写形式。所以库的样子就变成：COM.bruceeckel.utility.foibles。然而，在 Java 1.2 的开发过程中，设计者发现这样做会造成一些问题。所以目前的整个软件包都以小写字母为标准。

Java 的这种特殊机制意味着所有文件都自动存在于自己的命名空间里。而且一个文件里的每个类都自动获得一个独一无二的标识符（当然，一个文件里的类名必须是唯一的）。所以不必学习特殊的语言知识来解决这个问题——语言本身已帮我们照顾到这一点。

2.6.2 使用其他组件

一旦要在自己的程序里使用一个预先定义好的类，编译器就必须知道如何找到它。当然，这个类可能就在发出调用的那个相同的源码文件里。如果是那种情况，只需简单地使用这个类即可——即使它直到文件的后面仍未得到定义。Java 消除了“向前引用”的问题，所以不要关心这些事情。

但假若那个类位于其他文件里呢？您或许认为编译器应该足够“联盟”，可以自行发现它。但实情并非如此。假设我们想使用一个具有特定名称的类，但那个类的定义位于多个文件里。或者更糟，假设我们准备写一个程序，但在创建它的时候，却向自己的库加入了一个新类，它与现有某个类的名字发生了冲突。

为解决这个问题，必须消除所有潜在的、纠缠不清的情况。为达到这个目的，要用 import 关键字准确告诉 Java 编译器我们希望的类是什么。import 的作用是指示编译器导入一个“包”——或者说一个“类库”（在其他语言里，可将“库”想象成一系列函数、数据以及类的集合。但请记住，Java 的所有代码都必须写入一个类中）。

大多数时候，我们直接采用来自标准 Java 库的组件（部件）即可，它们是与编译器配套提供的。使用这些组件时，没有必要关心冗长的保留域名；举个例子来说，只需象下面这样写一行代码即可：

```
import java.util.Vector;
```

它的作用是告诉编译器我们想使用 Java 的 Vector 类。然而，util 包含了数量众多的类，我们有时希望使用其中的几个，同时不想全部明确地声明它们。为达到这个目的，可使用“*”通配符。如下所示：

```
import java.util.*;
```

需导入一系列类时，采用的通常是这个办法。应尽量避免一个一个地导入类。

2.6.3 static 关键字

通常，我们创建类时会指出那个类的对象的外观与行为。除非用 new 创建那个类的一个对象，否则实际上并未得到任何东西。只有执行了 new 后，才会正式生成数据存储空间，并可使用相应的方法。

但在两种特殊的情形下，上述方法并不堪用。一种情形是只想用一个存储区域来保存一个特定的数据——无论要创建多少个对象，甚至根本不创建对象。另

一种情形是我们需要一个特殊的方法，它没有与这个类的任何对象关联。也就是说，即使没有创建对象，也需要一个能调用的方法。为满足这两方面的要求，可使用 static（静态）关键字。一旦将什么东西设为 static，数据或方法就不会同那个类的任何对象实例联系到一起。所以尽管从未创建那个类的一个对象，仍能调用一个 static 方法，或访问一些 static 数据。而在这之前，对于非 static 数据和方法，我们必须创建一个对象，并用那个对象访问数据或方法。这是由于非 static 数据和方法必须知道它们操作的具体对象。当然，在正式使用前，由于 static 方法不需要创建任何对象，所以它们不可简单地调用其他那些成员，同时不引用一个已命名的对象，从而直接访问非 static 成员或方法（因为非 static 成员和方法必须同一个特定的对象关联到一起）。

有些面向对象的语言使用了“类数据”和“类方法”这两个术语。它们意味着数据和方法只是为作为一个整体的类而存在的，并不是为那个类的任何特定对象。有时，您会在其他一些 Java 书刊里发现这样的称呼。

为了将数据成员或方法设为 static，只需在定义前置和这个关键字即可。例如，下述代码能生成一个 static 数据成员，并对其初始化：

```
class StaticTest {
Static int i = 47;
}
```

现在，尽管我们制作了两个 StaticTest 对象，但它们仍然只占据 StaticTest.i 的一个存储空间。这两个对象都共享同样的 i。请考察下述代码：

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

此时，无论 st1.i 还是 st2.i 都有同样的值 47，因为它们引用的是同样的内存区域。

有两个办法可引用一个 static 变量。正如上面展示的那样，可通过一个对象命名它，如 st2.i。亦可直接用它的类名引用，而这在非静态成员里是行不通的（最好用这个办法引用 static 变量，因为它强调了那个变量的“静态”本质）。

```
StaticTest.i++;
```

其中，++运算符会使变量增值。此时，无论 st1.i 还是 st2.i 的值都是 48。

类似的逻辑也适用于静态方法。既可象对其他任何方法那样通过一个对象引用静态方法，亦可用特殊的语法格式“类名.方法()”加以引用。静态方法的定义是类似的：

```
class StaticFun {
static void incr() { StaticTest.i++; }
}
```

从中可看出，StaticFun 的方法 incr()使静态数据 i 增值。通过对象，可用典型的方法调用 incr()：

```
StaticFun sf = new StaticFun();
sf.incr();
```

或者，由于 incr()是一种静态方法，所以可通过它的类直接调用：

```
StaticFun.incr();
```

尽管是“静态”的，但只要应用于一个数据成员，就会明确改变数据的创建

方式（一个类一个成员，以及每个对象一个非静态成员）。若应用于一个方法，就没有那么戏剧化了。对方法来说，static 一项重要的用途就是帮助我们在不必创建对象的前提下调用那个方法。正如以后会看到的那样，这一点是至关重要的——特别是在定义程序运行入口方法 main()的时候。

和其他任何方法一样，static 方法也能创建自己类型的命名对象。所以经常把 static 方法作为一个“领头羊”使用，用它生成一系列自己类型的“实例”。

2.7 我们的第一个 Java 程序

最后，让我们正式编一个程序（注释⑤）。它能打印出与当前运行的系统有关的资料，并利用了来自 Java 标准库的 System 对象的多种方法。注意这里引入了一种额外的注释样式：“//”。它表示到本行结束前的所有内容都是注释：

```
// HelloDate.java
import java.util.*;

public class HelloDate {
   public static void main(String[] args) {
         System.out.println("Hello, it's: ");
        System.out.println(new Date());
     }
}
```

84 页程序

⑤：在某些编程环境里，程序会在屏幕上一切而过，甚至没机会看到结果。可将下面这段代码置于 main()的末尾，用它暂停输出：

```
try {
Thread.currentThread().sleep(5 * 1000);
} catch(InterruptedException e) {}
}
```

它的作用是暂停输出 5 秒钟。这段代码涉及的一些概念要到本书后面才会讲到。所以目前不必深究，只知道它是让程序暂停的一个技巧便可。

在每个程序文件的开头，都必须放置一个 import 语句，导入那个文件的代码里要用到的所有额外的类。注意我们说它们是“额外”的，因为一个特殊的类库会自动导入每个 Java 文件：java.lang。启动您的 Web 浏览器，查看由 Sun 提供的用户文档（如果尚未从 http://www.java.sun.com 下载，或用其他方式安装了 Java 文档，请立即下载）。在 packages.html 文件里，可找到 Java 配套提供的所有类库名称。请选择其中的 java.lang。在“Class Index”下面，可找到属于那个库的全部类的列表。由于 java.lang 默认进入每个 Java 代码文件，所以这些类在任何时候都可直接使用。在这个列表里，可发现 System 和 Runtime，我们在 Property.java 里用到了它们。java.lang 里没有列出 Date 类，所以必须导入另一个类库才能使用它。如果不清楚一个特定的类在哪个类库里，或者想检视所有的类，可在 Java 用户文档里选择“Class Hierarchy”（类分级结构）。在

Web 浏览器中，虽然要花不短的时间来建立这个结构，但可清楚找到与 Java 配套提供的每一个类。随后，可用浏览器的“查找”（Find）功能搜索关键字“Date”。经这样处理后，可发现我们的搜索目标以 java.util.Date 的形式列出。我们终于知道它位于 util 库里，所以必须导入 java.util.*；否则便不能使用 Date。

观察 packages.html 文档最开头的部分（我已将其设为自己的默认起始页），请选择 java.lang，再选 System。这时可看到 System 类有几个字段。若选择 out，就可知道它是一个 static PrintStream 对象。由于它是“静态”的，所以不需要我们创建任何东西。out 对象肯定是 3，所以只需直接用它即可。我们能对这个 out 对象做的事情由它的类型决定：PrintStream。PrintStream 在说明文字中以一个超链接的形式列出，这一点做得非常方便。所以假若单击那个链接，就可看到能够为 PrintStream 调用的所有方法。方法的数量不少，本书后面会详细介绍。就目前来说，我们感兴趣的只有 println()。它的意思是“把我给你的东西打印到控制台，并用一个新行结束”。所以在任何 Java 程序中，一旦要把某些内容打印到控制台，就可条件反射地写上 System.out.println("内容")。

类名与文件是一样的。若象现在这样创建一个独立的程序，文件中的一个类必须与文件同名（如果没这样做，编译器会及时作出反应）。类里必须包含一个名为 main()的方法，形式如下：

```
public static void main(String[] args) {
```

其中，关键字“public”意味着方法可由外部世界调用（第 5 章会详细解释）。main()的自变量是包含了 String 对象的一个数组。args 不会在本程序中用到，但需要在这个地方列出，因为它们保存了在命令行调用的自变量。

程序的第一行非常有趣：

```
System.out.println(new Date());
```

请观察它的自变量：创建 Date 对象唯一的目的就是将它的值发送给 println()。一旦这个语句执行完毕，Date 就不再需要。随之而来的“垃圾收集器”会发现这一情况，并在任何可能的时候将其回收。事实上，我们没太大的必要关心“清除”的细节。

第二行调用了 System.getProperties()。若用 Web 浏览器查看联机用户文档，就可知道 getProperties()是 System 类的一个 static 方法。由于它是“静态”的，所以不必创建任何对象便可调用该方法。无论是否存在该类的一个对象，static 方法随时都可使用。调用 getProperties()时，它会将系统属性作为 Properties 类的一个对象生成（注意 Properties 是“属性”的意思）。随后的的句柄保存在一个名为 p 的 Properties 句柄里。在第三行，大家可看到 Properties 对象有一个名为 list()的方法，它将自己的全部内容都发给一个我们作为自变量传递的 PrintStream 对象。

main()的第四和第六行是典型的打印语句。注意为了打印多个 String 值，用加号（+）分隔它们即可。然而，也要在这里注意一些奇怪的事情。在 String 对象中使用时，加号并不代表真正的“相加”。处理字串时，我们通常不必考虑“+”的任何特殊含义。但是，Java 的 String 类要受一种名为“运算符过载”的机制的制约。也就是说，只有在随同 String 对象使用时，加号才会产生与其他任何地方不同的表现。对于字串，它的意思是“连接这两个字串”。

但事情到此并未结束。请观察下述语句：

```
System.out.println("Total Memory = "
+ rt.totalMemory()
```

```
+ " Free Memory = "
+ rt.freeMemory());
```

其中，totalMemory()和 freeMemory()返回的是数值，并非 String 对象。如果将一个数值“加”到一个字串身上，会发生什么情况呢？同我们一样，编译器也会意识到这个问题，并魔术般地调用一个方法，将那个数值（int，float 等等）转换成字串。经这样处理后，它们当然能利用加号“加”到一起。这种“自动类型转换”亦划入“运算符过载”处理的范畴。

许多 Java 著作都在热烈地辩论“运算符过载”（C++的一项特性）是否有用。目前就是反对它的一个好例子！然而，这最多只能算编译器（程序）的问题，而且只是对 String 对象而言。对于自己编写的任何源代码，都不可能使运算符“过载”。

通过为 Runtime 类调用 getRuntime()方法，main()的第五行创建了一个 Runtime 对象。返回的则是指向一个 Runtime 对象的句柄。而且，我们不必关心它是一个静态对象，还是由 new 命令创建的一个对象。这是由于我们不必为清除工作负责，可以大模大样地使用对象。正如显示的那样，Runtime 可告诉我们与内存使用有关的信息。

2.8 注释和嵌入文档

Java 里有两种类型的注释。第一种是传统的、C 语言风格的注释，是从 C++继承而来的。这些注释用一个“/*”起头，随后是注释内容，并可跨越多行，最后用一个“*/”结束。注意许多程序员在连续注释内容的每一行都用一个“*”开头，所以经常能看到象下面这样的内容：

```
/* 这是
* 一段注释，
* 它跨越了多个行
*/
```

但请记住，进行编译时，/*和*/之间的所有东西都会被忽略，所以上述注释与下面这段注释并没有什么不同：

```
/* 这是一段注释，
它跨越了多个行 */
```

第二种类型的注释也起源于 C++。这种注释叫作“单行注释”，以一个“//”起头，表示这一行的所有内容都是注释。这种类型的注释更常用，因为它书写时更方便。没有必要在键盘上寻找“/”，再寻找“*”（只需按同样的键两次），而且不必在注释结尾时加一个结束标记。下面便是这类注释的一个例子：

```
// 这是一条单行注释
```

2.8.1 注释文档

对于 Java 语言，最体贴的一项设计就是它并没有打算让人们为了写程序而

写程序——人们也需要考虑程序的文档化问题。对于程序的文档化，最大的问题莫过于对文档的维护。若文档与代码分离，那么每次改变代码后都要改变文档，这无疑会变成相当麻烦的一件事情。解决的方法看起来似乎很简单：将代码同文档“链接”起来。为达到这个目的，最简单的方法是将所有内容都置于同一个文件。然而，为使一切都整齐划一，还必须使用一种特殊的注释语法，以便标记出特殊的文档；另外还需要一个工具，用于提取这些注释，并按有价值的形式将其展现出来。这些都是 Java 必须做到的。

用于提取注释的工具叫作 javadoc。它采用了部分来自 Java 编译器的技术，查找我们置入程序的特殊注释标记。它不仅提取由这些标记指示的信息，也将毗邻注释的类名或方法名提取出来。这样一来，我们就可用最轻的工作量，生成十分专业的程序文档。

javadoc 输出的是一个 HTML 文件，可用自己的 Web 浏览器查看。该工具允许我们创建和管理单个源文件，并生动生成有用的文档。由于有了 jvadoc，所以我们能够用标准的方法创建文档。而且由于它非常方便，所以我们能轻松获得所有 Java 库的文档。

2.8.2 具体语法

所有 javadoc 命令都只能出现于“/**”注释中。但和平常一样，注释结束于一个“*/”。主要通过两种方式来使用 javadoc：嵌入的 HTML，或使用“文档标记”。其中，“文档标记”（Doc tags）是一些以“@”开头的命令，置于注释行的起始处（但前导的“*”会被忽略）。

有三种类型的注释文档，它们对应于位于注释后面的元素：类、变量或者方法。也就是说，一个类注释正好位于一个类定义之前；变量注释正好位于变量定义之前；而一个方法定义正好位于一个方法定义的前面。如下面这个简单的例子所示：

```
/**  一个类注释  */
public class docTest {
/**  一个变量注释  */
public int i;
/**  一个方法注释  */
public void f() {}
}
```

注意 javadoc 只能为 public（公共）和 protected（受保护）成员处理注释文档。“private”（私有）和“友好”（详见 5 章）成员的注释会被忽略，我们看不到任何输出（也可以用-private 标记包括 private 成员）。这样做是有道理的，因为只有 public 和 protected 成员才可在文件之外使用，这是客户程序员的希望。然而，所有类注释都会包含到输出结果里。

上述代码的输出是一个 HTML 文件，它与其他 Java 文档具有相同的标准格式。因此，用户会非常熟悉这种格式，可在您设计的类中方便地“漫游”。设计程序时，请务必考虑输入上述代码，用 javadoc 处理一下，观看最终 HTML 文件的效果如何。

2.8.3 嵌入 HTML

javadoc 将 HTML 命令传递给最终生成的 HTML 文档。这便使我们能够充分利用 HTML 的巨大威力。当然，我们的最终动机是格式化代码，不是为了哗众取宠。下面列出一个例子：

```
/**
* <pre>
* System.out.println(new Date());
* </pre>
*/
```

亦可象在其他 Web 文档里那样运用 HTML，对普通文本进行格式化，使其更具条理、更加美观：

```
/**
*  您<em>甚至</em>可以插入一个列表：
* <ol>
* <li>  项目一
* <li>  项目二
* <li>  项目三
* </ol>
*/
```

注意在文档注释中，位于一行最开头的星号会被 javadoc 丢弃。同时丢弃的还有前导空格。javadoc 会对所有内容进行格式化，使其与标准的文档外观相符。不要将<h1>或<hr>这样的标题当作嵌入 HTML 使用，因为 javadoc 会插入自己的标题，我们给出的标题会与之冲撞。

所有类型的注释文档——类、变量和方法——都支持嵌入 HTML。

2.8.4 @see：引用其他类

所有三种类型的注释文档都可包含@see 标记，它允许我们引用其他类里的文档。对于这个标记，javadoc 会生成相应的 HTML，将其直接链接到其他文档。格式如下：

```
@see  类名
@see  完整类名
@see  完整类名#方法名
```

每一格式都会在生成的文档里自动加入一个超链接的“See Also”（参见）条目。注意 javadoc 不会检查我们指定的超链接，不会验证它们是否有效。

2.8.5 类文档标记

随同嵌入 HTML 和@see 引用，类文档还可以包括用于版本信息以及作者姓名的标记。类文档亦可用于“接口”目的（本书后面会详细解释）。

1. @version

格式如下：

@version 版本信息

其中，“版本信息”代表任何适合作为版本说明的资料。若在 javadoc 命令行使用了“-version”标记，就会从生成的 HTML 文档里提取出版本信息。

2. @author

格式如下：

@author 作者信息

其中，“作者信息”包括您的姓名、电子函件地址或者其他任何适宜的资料。若在 javadoc 命令行使用了“-author”标记，就会专门从生成的 HTML 文档里提取出作者信息。

可为一系列作者使用多个这样的标记，但它们必须连续放置。全部作者信息会一起存入最终 HTML 代码的单独一个段落里。

2.8.6 变量文档标记

变量文档只能包括嵌入的 HTML 以及@see 引用。

2.8.7 方法文档标记

除嵌入 HTML 和@see 引用之外，方法还允许使用针对参数、返回值以及违例的文档标记。

1. @param

格式如下：

@param 参数名 说明

其中，“参数名”是指参数列表内的标识符，而“说明”代表一些可延续到后续行内的说明文字。一旦遇到一个新文档标记，就认为前一个说明结束。可使用任意数量的说明，每个参数一个。

2. @return

格式如下：

@return 说明

其中，“说明”是指返回值的含义。它可延续到后面的行内。

3. @exception

有关“违例”（Exception）的详细情况，我们会在第 9 章讲述。简言之，它们是一些特殊的对象，若某个方法失败，就可将它们“扔出”对象。调用一个方法时，尽管只有一个违例对象出现，但一些特殊的方法也许能产生任意数量的、不同类型的违例。所有这些违例都需要说明。所以，违例标记的格式如下：

@exception 完整类名 说明

其中，“完整类名”明确指定了一个违例类的名字，它是在其他某个地方定义好的。而“说明”（同样可以延续到下面的行）告诉我们为什么这种特殊类型的违例会在方法调用中出现。

4. @deprecated

这是 Java 1.1 的新特性。该标记用于指出一些旧功能已由改进过的新功能取代。该标记的作用是建议用户不必再使用一种特定的功能，因为未来改版时可能摒弃这一功能。若将一个方法标记为@deprecated，则使用该方法时会收到编译器的警告。

2.8.8 文档示例

下面还是我们的第一个 Java 程序，只不过已加入了完整的文档注释：

```
//: c02:HelloDate.java
import java.util.*;

/** The first Thinking in Java example program.
 * Displays a string and today's date.
 * @author Bruce Eckel
 * @author http://www.BruceEckel.com
 * @version 2.0
*/
public class HelloDate {
  /** Sole entry point to class & application
   * @param args array of string arguments
   * @return No return value
   * @exception exceptions No exceptions thrown
  */
  public static void main(String[] args) {
    System.out.println("Hello, it's: ");
    System.out.println(new Date());
  }
} ///:~
```

92 页程序

第一行：

//: Property.java

采用了我自己的方法：将一个“:”作为特殊的记号，指出这是包含了源文件名字的一个注释行。最后一行也用这样的一条注释结尾，它标志着源代码清单的结束。这样一来，可将代码从本书的正文中方便地提取出来，并用一个编译器检查。这方面的细节在第 17 章讲述。

2.9 编码样式

一个非正式的 Java 编程标准是大写一个类名的首字母。若类名由几个单词构成，那么把它们紧靠到一起（也就是说，不要用下划线来分隔名字）。此外，每个嵌入单词的首字母都采用大写形式。例如：

```
class AllTheColorsOfTheRainbow { // ...}
```

对于其他几乎所有内容：方法、字段（成员变量）以及对象句柄名称，可接受的样式与类样式差不多，只是标识符的第一个字母采用小写。例如：

```
class AllTheColorsOfTheRainbow {
int anIntegerRepresentingColors;
void changeTheHueOfTheColor(int newHue) {
// ...
}
// ...
}
```

当然，要注意用户也必须键入所有这些长名字，而且不能输错。

2.10 总结

通过本章的学习，大家已接触了足够多的 Java 编程知识，已知道如何自行编写一个简单的程序。此外，对语言的总体情况以及一些基本思想也有了一定程度的认识。然而，本章所有例子的模式都是单线形式的“这样做，再那样做，然后再做另一些事情”。如果想让程序作出一项选择，又该如何设计呢？例如，“假如这样做的结果是红色，就那样做；如果不是，就做另一些事情”。对于这种基本的编程方法，下一章会详细说明在 Java 里是如何实现的。

2.11 练习

(1) 参照本章的第一个例子，创建一个“Hello，World”程序，在屏幕上简单地显示这句话。注意在自己的类里只需一个方法（“main”方法会在程序启动时执行）。记住要把它设为 static 形式，并置入自变量列表——即使根本不会用到这个列表。用 javac 编译这个程序，再用 java 运行它。

(2) 写一个程序，打印出从命令行获取的三个自变量。

(3) 找出 Property.java 第二个版本的代码，这是一个简单的注释文档示例。请对文件执行 javadoc，并在自己的 Web 浏览器里观看结果。

(4) 以练习(1)的程序为基础，向其中加入注释文档。利用 javadoc，将这个注释文档提取为一个 HTML 文件，并用 Web 浏览器观看。