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

2: 万物皆对象

虽然 Java 是建立在 C++之上的，但它是一个更为“纯粹”的面向对象的语言。

C++和 Java 都是混合语言，但是 Java 的软件设计师们并不认为这种混合性会像它在 C++里那么重要。混合语言能让你用多种风格进行编程；C++之所以要成为一种混合语言，是因为它必须为 C 提供向后兼容。由于 C++是 C 语言的超集，所以它包括了许多 C 语言的一些不怎么得人心的特性，从某些方面讲这些东西让 C++变得复杂得过了头。

Java 语言假定你只需要使用面向对象的编程方法。这就意味着，你必须在开始之前就把脑筋转到面向对象的世界(除非你已经身在其中了)。把精力花在这个地方的好处就是，你能用一种较其它 OOP 语言更易学易用的编程语言进行开发。本章，我们会先学习 Java 程序的基本组成，然后会看到 Java 的一切，甚至 Java 程序本身都是对象。

用 reference 操控对象

每种编程语言都有它自己的操控数据的方法。有时程序员必须时刻记着他们正在进行何种操控。你是在直接操控对象呢，还是通过一些间接的表示方法(就是 C 或 C++的指针)，用特殊的语法进行操控呢？

Java 把这一切都简化了。你可以用一种前后一致的语法，把一切都当对象来处理。虽然你把一切都“当作”对象，但实际上你所操控的那个标识符是对象的“reference”。[10]你可以把它想成一个带遥控器(reference)的电视机(对象)。只要你还拿着 reference，你就还连着电视机，但是当有人说“换台”或是“把声音调低点”的时候，你操控的实际上是那个 reference，然后再让它去和那个对象打交道。如果你想在房间里走一走，那么带上遥控器/reference就行了，不必扛着电视机。

此外，即便没有电视机，遥控器也能独立存在。也就是说，reference 并不一定要连着对象。所以，如果要保存一个单词或句子，只要创建一个 **String** 的 reference 就行了：

```
String s;
```

但是你创建的“只是”一个 reference，而不是对象。如果你现在就往 **s** 送消息，那么你将会得到一个错误(在运行的时候)，因为 **s** 实际上并没有连到任何东西(没有电视机)。比较安全的做法是，创建 reference 的时候就对它进行初始化：

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

不过这里用到了 Java 的一个特有的特性：字符串可以用以引号括起来的文字进行初始化。通常情况下，你必须用一种更为通用的方法对对象进行初始化。

你必须创建所有的对象

创建完 reference 之后，你就得往上面连新的对象了。大体上，你得用 **new** 关键词来作这件事。关键词 **new** 的意思是，“给我创建一个新的那种类型的对象。”所以在上述例程中，你可以用：

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

它不仅表示“给我创建一个新的 **String**”，而且还用字符串参数告诉它“如何”制作这个 **String**。

当然，**String** 并不是唯一的类。Java 还带了很多现成的类。更重要的是你还可以创建你自己的类。实际上，Java 编程基本上就是在创建类，而这正是本书要教给你的。

数据存在哪里

对程序运行时各部分是如何展开的——特别是内存是如何分配的，作一个直观的描述还是很有必要的。数据可以存储在以下六个地方：

1. **寄存器(registers)**。这是反映最快的存储，因为它所处位置不同：在处理器里。不过寄存器的数量非常有限，所以它是由编译器分配的。你非但不能直接控制寄存器，甚至连它存在的证据也找不到。
2. **栈(stack)**。位于“常规内存区(general random-access memory area)”里，处理器可以通过栈指针(*stack pointer*)对它进行直接访问。栈指针向下移就创建了新的存储空间，向上移就释放内存空间。这是仅次于寄存器的最快、最有效率的分配内存的方法。由于 Java 编译器必须生成能控制栈指针上移和下移的代码，所以程序编译的时候，那些将被存储在栈中的数据的大小和生命周期必须是已知的。这使得程序的灵活性收到了限制，所以尽管 Java 把某些数据——特别是对象的 reference 存放在栈里，但对象本身并没有放在栈里。
3. **堆(heap)**。这是一段“多用途的内存池”(general-purpose pool of memory，也在内存里面)，所有 Java 对象都保存在这里。同栈不同，堆的优点是，分配空间的时候，编译器无需知道该分配多少空间，或者这些数据会在堆里呆多长时间。因此使用堆的空间会比较灵活。只要你想创建对象，用 **new** 就行了，程序执行的时候自会在堆里分配空间。当然

你得为这种灵活性付出代价，分配堆的存储空间要比分配栈的慢一些(假如你能像 C++那样在栈里创建对象的话)。

4. **静态存储(static storage)**。“静态”在这里的意思就是“在固定的位置”(尽管它还是在 RAM 里面)。静态存储里面的数据在整个程序的运行期间都能访问到。你可以用 **static** 关键词来指明对象中的某个元素是静态的，但是 Java 对象本身是决不会放到静态存储中去的。
5. **固定存储(constant storage)**。常量值通常直接放在程序里，这样它们就不会被改动了，因而也更安全。有时常量还能为自己设置界限，这样在嵌入式系统中，你就能选择是不是把它们放到 ROM 里面去。
6. **非内存的存储(Non-RAM storage)**。如果数据完全独立于程序，那么即使程序不运行，它也应该也还在；即使程序失去了对数据的控制，它也仍然还在。两个最主要的例子是“流对象 (*streamed object*) ”和“persistent 对象(*persistent object*)”。大致上说，前者是一种会被送往另一台机器的，由对象转化而成的字节流；而后者则是保存在磁盘上的，能在程序中止之后仍保存对象状态的文件。这类存储的奥妙就在于，对象被转化成了某种能保存在其它介质上的东西，但是要用的时候，又能在内存里重建。Java 提供了“轻量级 persistence (*lightweight persistence*) ”的支持。未来 Java 可能会提供更为完整的 persistence 的支持。

特例: primitive 类型

有一种编程时会经常用到的数据类型，会被当作特例来处理。你可以把它想成“primitive(原始)”类型。之所以要把它单独列出来，是因为用 **new** 创建对象——特别是像简单变量之类的小对象的时候，效率不是太高，因为它们都是放在堆里的。对于这类数据，Java 承袭了 C 和 C++ 的办法。也就是说，这个变量不是用 **new** 来创建的，相反，这里所创建的是一个“非 reference”的“自动”变量。这个变量保存着值，并且存储在栈中，因而效率会比较高。

Java 决定了每种 primitive 类型的大小。它不会像其它语言那样，随机器架构的不同而不同。这种变量大小的一致性是 Java 程序可移植的基础之一。

Primitive 类型	大小	最小	最大	**Wrapper 类型**
boolean	—	—	—	**Boolean**
char	16-bit	Unicode 0	Unicode 2^{16}- 1	**Character**
byte	8-bit	-128	+127	**Byte**
short	16-bit	-2^{15}	$+2^{15}-1$	**Short**
int	32-bit	-2^{31}	$+2^{31}-1$	**Integer**
long	64-bit	-2^{63}	$+2^{63}-1$	**Long**
float	32-bit	IEEE754	IEEE754	**Float**
double	64-bit	IEEE754	IEEE754	**Double**

void	—	—	—	**Void**

所有的数字类型都是带符号的，所以不用再找 unsigned 类型了。

这里没说 **boolean** 的大小；它只能存储 **true** 和 **false** 这两个值 (literal values **true** or **false**)。

Primitive 类型的“wrapper”类允许你在堆里创建一个表示这个 primitve 型数据的对象。这个对象不是 primitive 型的。例如：

```
char c = 'x';
Character C = new Character(c);
```

或者你也可以这样：

```
Character C = new Character('x');
```

为什么要这么作呢？我们会在后面的章节讲的。

高精度的数值

Java 还包括两个能进行高精度算术运算的类：**BigInteger** 和 **BigDecimal**。虽然它们也马马虎虎能被算作是“wrapper”类，但是没有哪个 primitive 类型能和这两个类相对应。

这两个类都提供了能模拟 primitive 类型的操作的方法。也就是说，你可以用 **BigInteger** 或 **BigDecimal** 完成任何 **int** 或 **float** 能完成的工作，只是你不能使用操作符，而只能调用方法。此外，由于牵涉到的东西比较多，因而速度会稍微慢些。实际上这是在用速度换精度。

BigInteger 支持任意精度的整数。也就是说，它可以精确地表示任意大的自然数，所以运算的时候不会丢失任何信息。

BigDecimal 能表示任意精度的浮点数；因此，你可以用它来进行精度要求极高的货币兑换的计算。

欲知这两个类的构造函数和方法调用方面的细节，请查阅 JDK 文档。

Java 中的数组

实际上所有编程语言都有数组。使用 C 和 C++的数组是有风险的，因为在它们眼里，数组只是一段内存。如果程序访问到数组之外的内存，或者

还未进行初始化就使用了这些内存(很常见的编程错误)，那么很有可能会发生一些无法预料的后果。

安全性是 Java 最看重的目标之一，因此许多困扰 C 和 C++程序员的问题在 Java 里面已经不复存在了。数组肯定会被初始化，而要想访问数组以外的内存也已经不可能了。边界检查的代价就是，每个数组都会多占用一些内存，而且程序运行的时候也会有些开销。不过设计者们相信，与安全性的增强和编程效率的提升相比，这点代价值。

当你创建对象数组的时候，实际上你是在创建 reference 的数组。这些 reference 会被自动地初始化为一个特殊的值：**null**。这是个关键词，意思就是“没有”。Java 一看到 **null** 就知道这个 reference 没有指向任何对象。使用 reference 之前，你必须用对象对它进行赋值，如果你试图用一个仍然是 **null** 的 reference，那么程序运行的时候就会报错。由此，Java 防止了许多常见的数组错误。

你可以创建 primitive 的数组。编译器也会进行初始化，因为它会把数组的内存全部清零。

以后章节会更详细的讲解数组。

你永远不需要清理对象

在绝大多数编程语言中，要弄清变量生命周期的概念都会耗费大量的精力。变量要生存多长时间？如果要由你来进行清理，那么应该在什么时候进行清理？在变量生命周期方面含糊其辞，会导致很多 bug。这一章会告诉你，Java 是怎样通过接管所有的清理工作来大大简化这个问题的。

作用域

绝大多数过程语言都有“作用域 (scope)”的概念。它决定了在这个作用域里定义的变量的可见性与生命周期。在 C，C++和 Java 中，作用域是由花括号**{}**的位置决定。所以，假如：

```
{
  int x = 12;
  // Only x available
  {
    int q = 96;
    // Both x & q available
  }
  // Only x available
  // q “out of scope”
}
```

在作用域中定义的变量只能用到这个作用域的结尾。

'//' 之后的这行文字都是注释。

缩进使得 Java 的代码更易读。由于 Java 是一种形式自由的语言，因此额外的空格，跳格以及回车不会对程序的结构造成任何影响。

注意，虽然在 C 和 C++里这种写法是完全合法的，但是你**不能**在 Java 里这么写：

```
{
  int x = 12;
  {
    int x = 96; // Illegal
  }
}
```

编译器会说变量 **x** 已经定义过了。于是 C 和 C++的，在更大的作用域里“隐藏”变量的能力，就被 Java 给禁了。因为 Java 的设计者们认为这会导致令人费解的程序。

对象的作用域

Java 对象的生命周期同 primitive 的不同。当你用 **new** 创建 Java 对象之后，它会晃荡到作用域外面。如果你写：

```
{
  String s = new String("a string");
} // End of scope
```

s 这个 reference 会在作用域结束之后消失。但是 **s** 所指的那个 **String** 仍然还占着内存。在这段代码里，你已经没法访问这个对象了，因为唯一指向它那个的 reference 现在已经出了作用域。在后面的章节中你还会看到，在程序的运行过程中，它是怎样传递和复制对象的 reference 的。

只要你还用得着，那些用 **new** 创建的对象就不会跑开。看来用了这种方法之后，C++里面的一大堆问题在 Java 里面已经不复存在了。C++编程中最难的问题就是，等你要用对象的时候，你根本没法从语言中获得这个对象还在不在的信息。而且更重要的是，用 C++编程，你得亲自进行对象的清理。

这就带来了一个有趣的问题。如果 Java 就这样只管造对象，不管清理对象，那么又是谁在那里阻止对象填满内存进而把程序给搞宕的呢？这确实是 C++所面临的问题。但这里就有点魔法了。Java 有一个垃圾回收器 (*gabage collector*)，它会看着所有用 **new** 创建的对象，并且还会知道

其中的哪些已经没有 reference 指着了。然后它会释放那些没有 reference 指着的对象所占据的内存，这样内存就能被新的对象用了。这就是说你永远也不必为重新申请内存而操心。你只要创建对象就行了，用完之后它们自会离开。这样就彻底解决了因程序员忘了释放内存而产生的，所谓“内存泄漏(memory leak)”的编程问题了。

创建新的数据类型：类

如果万物皆对象，那么特定类型的对象的特征和行为又是由什么决定的呢？换言之，类是由什么组成的？或许你会想应该有一个叫 “type”的关键词，当然这种想法很有道理。然而长久以来，绝大多数的面向对象的语言都用了 **class** 这个关键词，它的意思是“现在我要告诉你这种新类型的对象都有些什么特征。”**class** 关键词 (这个词太常用了，所以本书的后面部分就不再用黑体字表示了)后面跟着新类型的名字。例如：

```
class ATypeName { /* Class body goes here */ }
```

尽管类的正文部分只是一段注释 (星斜杠以及在它们中间的东西，后面会讲到的)，但是这已经是一种新的类型了。虽然没什么大用，但是你可以用 **new** 来创建这种类型的对象：

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

但是除非你为它定义一些方法，否则你没法让它去作更多的事情(也就是，你不能向它发送任何有意义的消息)。

数据成员与方法

当你定义类的时候 (Java 编程要做的就是定义类，创建这些类的对象，然后再向这些对象发送消息)，你可以往类里放两种元素：数据 (*field*，有时也被称为数据成员)，以及方法 (*method*，有时也被称为成员函数 *member function*)。数据可以是任何类型的，能通过 reference 进行操控的对象。它也可以是任何一种 primitive 数据 (这时它就不是 reference 了)。如果它是一个对象的 reference，那么你就必须用一种被称为构造函数(*constructor* 会在第 4 章详细讲述)的特殊方法对它进行初始化，这样才能确保将它连上一个真实的对象(用 **new**，就像前面看到的)。如果这是个 primitive 数据，那么你可以在定义的时候直接对它进行初始化。(过一会就能看到，reference 也可以在定义时进行初始化。)

每个对象都会保存它自己的数据成员；数据成员不能共享。下面就是一个带数据成员的类：

```
class DataOnly {
  int i;
  float f;
  boolean b;
}
```

这个类做不了任何事情，但是你可以创建这样一个对象；

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

你可以为这些数据成员赋值，但是首先你必须得知道该如何引用对象的数据成员。你必须先声明对象的 reference 的名字，然后加一个“点”，后面再跟上对象成员的名字：

```
objectReference.member
```

例如：

```
d.i = 47;
d.f = 1.1f; // ‘f’ after number indicates float
constant
d.b = false;
```

此外，对象里面还会包含一些可能会需要修改的其它对象。这么作，只要接着“往下点”就行了。就像这样：

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

DataOnly 类除了保存数据，其它什么都干不了，因为它没带方法。要想理解方法是怎样工作的，你必须先理解参数(*argument*)和返回值(*return value*)的概念。这两个概念，我们过一会就讲。

primitive 成员的缺省值

当 primitive 数据成为类的成员时，即便你不进行初始化，它也会得到一个默认的值：

Primitive 类型	缺省值
boolean	False

char	**'\u0000' (null)**
byte	**(byte)0**
short	**(short)0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**

一定要记住，只有在“变量被用作类的成员”时，Java 才能确保它获得这些默认值。这样就确保了 primitive 的类的成员肯定能得到初始化 (这是 C++做不到的)，因此也降低了产生 bug 的可能性。但是对程序来说，这些初始值可能并不正确，甚至不合法。因此最好还是自己来做初始化。

“本地”变量——也就是非类数据成员的变量，就享受不到这种保障了。因此，如果你在方法中定义了：

```
int x;
```

那么这个 **x** 可以是任意值 (这同 C 和 C++)的又相同了；它不会自动地初始化为零。在你用 **x** 之前，你得先为它赋一个值。如果你忘了，Java 这点要比 C++强的多：它会给你一个编译时(compile-time)错误，告诉你这个变量可能没有初始化。(很多 C++的编译器会对为初始化的变量发出警告，但是在 Java 中，这就成了错误。)

方法、参数和返回值

很多语言 (比方说 C 和 C++)是用函数 (*function*) 这个术语来表示有名字的子程序的。对于 Java，更常见的术语是“方法 (*method*)”，也就是“怎样去做这件事”。如果你一定要把它认做是函数，那也没什么不可以。实际上这只是用词的不同，不过本书采纳了 Java 通常所用的术语，“方法”。

在 Java 里面，方法决定了对象能接受哪些消息。在本节中，你会看到定义方法竟然是如此简单。

方法的基本的组成包括方法的名字，参数，返回类型，以及方法的正文。下面就是它的基本形式：

```
returnType methodName( /* Argument list */ ) {
  /* Method body */
}
```

返回类型是指调用方法所返回的值的类型。参数列表则表示传给这个方法的数据的类型和名字。方法的名字再配合其参数列表，可以唯一地标识一个方法。

在 Java 语言里，方法只能是类的一部分。因此，你只能通过对象来调用方法，[11]因此那个对象必须要能进行调用。如果你调错了方法，那么编译的时候就会得到一个错误消息。调用对象的方法的时候，必须先给出对象的名字，然后是一个点，再跟上方法的名字和参数列表，就像这样：

```
objectName.methodName(arg1, arg2, arg3);
```

举例来说，假设有一个不需要任何参数，会返回 **int** 值的方法 **f( )**。如果你要通过 **a** 对象来调用 **f( )**，你可以这样：

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

x 的类型必须与返回值的类型相匹配。

调用方法通常被称为向对象发消息。在上述例子里，**f( )**就是消息，而 **a** 就是对象。面向对象的编程通常被简单的归纳为“向对象发消息”。

参数列表

方法的参数列表会告诉你应该向方法传哪些信息。或许你也猜到了，按照 Java“万物皆对象”的法则，这种信息应该是以对象的形式出现的。所以实际上参数列表就是传给方法的对象类型和名字。由于，不管在哪种情况下，Java 传递对象的时候，实际上是在传 reference，[12]因此 reference 的类型必须正确。如果参数说要一个 **String**，那你就必须传给它 **String**，否则编译器会报错。

看看下面这个拿 **String** 做参数的方法。这部分代码必须放到类的定义中去，这样才能编译通过：

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

这个方法会告诉你，需要多少字节才能存储这个 **String** 的信息。(为了支持 Unicode 字符集，**String** 里面的每个 **char** 都是 16 位或者说两个

字节长的。) 这个参数是 **String** 型的，名字叫 **s**。等你把 **s** 传给了这个方法，你就能像对待其它对象那样使用它了。(你可以向它发消息。) 这里，调用了 **length( )**方法，这是个 **String** 的方法；其返回值是字符串的字符数量。

你还可以看到 **return** 的用法。它会做两件事。首先，它的意思是“退出这个方法，我干完了”。其次，如果方法返回了一个值，那么这个值必须放在 **return** 的后面。这里，会先计算 **s.length( ) * 2** 表达式，再产生 **return** 的值。

你可以返回任何类型的值，但是如果你根本就不想返回，你必须标明这个方法的返回类型是 **void**。就像下面这些例子：

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

如果返回的类型是 **void**，那么 **return** 关键词就只能用来退出方法了。因此，如果方法已经结束了，那也没必要加了。你能从方法的任何地方返回，但是，无论你从那里返回，编译器都会(用错误消息)强制非 **void** 型的方法，返回一个这一类型的值。

看到这里，你会发现程序就像是一堆带着方法的对象，而方法又都拿其它对象作参数，并且向其它对象发消息。实际上也差不多，不过在以后的章节里，你会看到如何通过编写方法来完成具体的底层工作。就本章而言，知道传递消息就足够了。

构建 Java 程序

在看第一个 Java 程序之前，你必须先理解几个问题。

名字的可见性

控制名字是每个编程语言都要面对的问题。如果你在某个模块中用到了一个名字，而另一个程序员在别的什么模块里也用到了这个的名字，那么你该如何区分这两个名字，从而防止它们相互“冲撞”呢？这个问题在 C 里面特别严重，程序最后都成了无法管理的“名字海洋”了。C++(Java 的类就是基于 C++的)将函数装进了类里，这样它们就不会同属于其它类的函数相冲突了。不过 C++仍然保留了全局变量和全局函数，因此冲突仍有可能发生。为了解决这个问题，C++用 namespace 关键词引入了名字空间的概念。

Java 用了种全新的办法来解决这个问题。要想为类库找一个独一无二的名字，没有什么会比 Internet 的域名更好的了。实际上，Java 的设计者就是要你把 Internet 的域名到过来用，因为这能保证它是独一无二的。我的域名是 **BruceEckel.com**，所以我写的 foibles utility 类库的名字就是 **com.bruceeckel.utility.foibles**。把域名到过来之后，点就表示子目录了。

在 Java 1.0 和 1.1 中，域名的扩展部分，如 **com**，**edu**，**org**，**net**，都要求用大写，所以这个类库应该是：**COM.bruceeckel.utility.foibles**。但是当 Java 2 开发到一半的时候，大家发现这么做会有问题，所以现在 package 的名字又都是小写的了。

也就是说，这种机制会自动地为每个文件创建一个名字空间，而文件就生活在它自己的名字空间里。同时文件里面的类都必须使用唯一的标识符。因此你无需学习什么特别的语言特性就能解决这个问题——语言已经为你打理好了。

使用其它组件

只要程序用到了已经预先定义过的类，编译器就得知道该到那里去找这个类。当然，这个类可能是在同一个源文件里。碰到这种情况，只要直接用就是了——哪怕调用的时候这个类还没定义都无所谓 (Java 解决了“提前引用(forward referencing)”的问题，因此你不必担心)。

那么如果类是保存在其它文件里面的，那又该怎么做呢？ 可能你会认为编译器是非常智能的，它会三下二下就找到这个类，但问题没那么简单。设想一下，你要用一个叫“XXX”的类，但是编译器找到了好几个叫这个名字的类 (可能你作了好几个版本)。或者更糟糕。想想你写了个程序，但是编译完了之后你发觉，你往类库里面加了个同已有的类相冲突的类。

要解决这个问题，你就得排除所有会产生两义性的因素。你得用 **import** 关键词明确地告诉 Java 编译器，到底要用哪个类。**import** 告诉编译器把 package 带过来，而 package 就是类库。(在其它语言里，类库除了类之外，还可以包括函数和数据，但是 Java 的程序只能写在类里。)

大多数情况下，你都会用到那些随编译器安装的标准 Java 类库的组件。对于这些组件，你不必去记那些长的，反转过来的域名，你只要像这样：

```
import java.util.ArrayList;
```

告诉编译器，你要使用 Java 的 **ArrayList** 类。但是 **util** 里面还有很多别的类，你可能会用到其中的几个，但是又不想把它们全都列出来。这时用通配符‘*’就可以了：

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

与一个个地引入相比，这种一次引入一批的做法类更为常见。

static 关键词

通常创建类的时候，你只是在定义这类对象会长什么样子，会有什么行为。除非你用 new 创建一个那种类的对象，否则你什么也得不到。只有创建了对象，它才能分配到内存，方法也才能用。

但是碰到下述两种情况的时候，这种做法就显得无能为力了。一是，无论你要创建多少对象，甚至不创建对象，你都要有，且只要有一段保存某些数据的内存。另一种情况是，你要一个不从属任何具体对象的方法。也就是说，你要一个，即使没有创建任何对象也能调用的方法。**static** 关键词就能帮你达成上述这两个目的。当你声明某样东西是 **static** 的时候，你的意思是，这项数据或方法没有被连到类的任何一个实例之上。因此即便你从没创建过那个类的对象，你也可以调用其 **static** 方法或者访问其 **static** 数据。对于普通的非 **static** 的数据和方法，你就只能先创建对象，再访问那个对象的数据或方法了，因为非 **static** 的数据和方法必须知道它们是属于哪个对象的。当然，由于 **static** 方法无需创建任何对象就能使用，因此它们不能像普通方法那样，不说明对象的名字就直接访问它的非 static 的数据成员或方法了。(因为非 **static** 的数据成员和方法必须连到某个具体的对象上)。

有些面向对象的语言用“类数据(*class data*)”和“类方法(*class methods*)”这两个术语，来表示那些为整个类，而不是为这个哪个类的具体对象而定义的数据和方法。有时 Java 也用这两个术语。

要把数据成员或方法做成 **static** 的，只要在定义部分加上 **static** 关键词就行了。举例来说，下面这段代码就会生成并且初始化一个 **static** 的数据成员：

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

现在即使你作了两个 **StaticTest** 的对象，也只会有一个 **StaticTest.i**。这两个对象共享同一个 **i**。比如：

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

现在 **st1.i** 和 **st2.i** 的值都是 47，因为它们都引用了同一段内存。

有两种方法可以引用 **static** 变量。一是像上述例子那样，通过一个对象，比如 **st2.i**。你也可以直接用类的名字，这种写法不能用于非 **static** 的数据成员。(我们更提倡用这种方法来引用 **static** 的变量，因为它强调了变量是 **static** 的。)

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

++ 操作符表示递增这个变量。现在，**st1.i** 和 **st2.i** 的值都已经变成 48 了。

静态方法的思路也差不多。你既可以通过对象也可以用其特有的 **ClassName.method( )**句法来使用静态方法。定义 static 方法的形式也差不多：

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

现在你看到了 **StaticFun** 的 **incr( )**方法会用**++** 操作符递增 static 的数据 **i** 。你既可以像以往那样，通过对象来调用 **incr( )**：

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

也可以直接通过类来调用，因为 **incr( )** 是一个静态方法：

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

尽管 **static** 运用于数据成员的时候会根本性的改变数据创建的方式(对于非 **static** 的数据来说，每个对象都会有一份，static 的数据则变成了每个类只有一个)，但是用于方法的时候，这种改变就没那么大了。**static** 方法的一个重要用途就是让你不用创建对象就能调用方法。这一点非常重要，你会看到程序的入口点 **main( )**就用到了这个特性。

同其它方法一样，**static** 方法也可以创建或使用它本身这个类型的对象，所以 **static** 方法常常被用来管理本类型的对象，它就像是这些对象的“牧羊人”。

第一个 Java 程序

最后我们来写一个完整的程序。它会打印一个字符串，然后用 Java 标准类库的 **Date** 类来打印当前日期。

```
// 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());
  }
}
```

你得在源程序文件的开头放上 **import** 语句，这样它才能引入要用到的额外的类。注意我说“额外”。这是因为有一个类库：**java.lang**，它的类会被自动地引入所有 Java 文件。启动 Web 浏览器，查阅 Sun 的文档。(如果你还没有 JDK 文档，那赶快到 *java.sun.com* 去下载一份[13])。看 package 列表，里面是所有 Java 自带的类库。选择 **java.lang**，你会看到这个类库里边的所有的类。既然 Java 源文件都会默认地引入 **java.lang**，那么所有这些类都应该可以直接用。但是 **java.lang** 里面没有 **Date** 类，因此你必须 import 这个类。如果你不知道这个类是属于哪个 package，或者如果你想看到全部的类，你可以在 Java 文档里面选择“Tree”。现在你就可以看到所有的 Java 类了。然后你再用浏览器的“find”功能来找 **Date**。你会看到它的全名是 **java.util.Date**，于是你就知道了它属于 **util** 类库，因此要使用 **Date** 的话，你必须先 **import java.util.***。

回过头去我们再选 **java.lang**，然后 **System**，你会看到 **System** 类有好几个数据成员，其中 **out** 是 **static** 的 **PrintStream** 对象。既然是 **static** 的，那么你就无需创建了。**out** 对象永远存在，因此你可以直接使用。至于你能让 **out** 做些什么，这是由它的类型，**PrintStream** 决定的。**PrintStream** 以超链接的形式列在旁边，因此可以很方便地把它点开，这里有可以调用的 **PrintStream** 方法的清单。里面还真不少，以后我们会详细介绍。现在我们只对 **println( )** 感兴趣，它的意思是“把给你的东西，加个换行符之后打印到控制台上。”所以写 Java 程序的时候，你随时都可以 **System.out.println("things")**；只要你想打印。

类的名字同文件名相同。当你创建这种独立程序的时候，文件里面必须要有一个同文件名相同的类。(否则编译器就会报错。) 而那个方法必须要有一个具备下列特征(signature)的 **main( )**：

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

public 关键词的意思是，这个方法可以被外部世界调用(第 5 章再作详细介绍。) **main( )**的参数是一个 **String** 对象的数组。虽然这个程序没有用到 **args**，但是 Java 编译器还是会要求你把它填进去，因为它是用来存储命令行参数的。

输出日期的这行比较有意思：

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

它的参数是一个正在创建中的 **Date** 对象，而创建这个对象的目的就是要把它的值(会自动转换成 **String**)传给 **println( )**。这句一结束，**Date** 就不需要了，因此垃圾回收器会随时过来把它清理掉。我们不用担心清理的问题。

编译和运行

要编译和运行这段以及本书的其它程序，你必须先有一个 Java 的编程环境。有很多第三方的开发环境，但本书假定你使用的是 Sun 免费发布的 Java Developer's Kit (JDK)。如果你用的是别的开发系统，[14]那么你该参考它的文档，以决定该如何编译和运行程序。

直接到 *java.sun.com*。你会在那里找到能指导你，下载和安装你所属的平台的 JDK 的信息和链接。

一旦 JDK 安装完毕，你就得在这台机器上设定的路径信息了，这样它才能找到 **javac** 和 **java**。下载本书的源代码(上 *www.BruceEckel.com* 去找)，再把它解开。它会为本书的每一章都创建一个子目录。到 **c02** 目录，然后输入：

```
javac HelloDate.java
```

这条命令敲下去之后应该什么反映也没有。只要你得到什么错误信息，那就说明你还没正确地安装 JDK，于是你应该重新检查一下。

如果什么都没发生，提示符又重新出现了，那你就可以输入：

```
java HelloDate
```

于是你就能从输出中看到这个消息和日期了。

这就是编译和运行本书程序的步骤。不过你还会看到本书的源代码里，每一章都有一个 **build.xml** 的文件。它包含了能自动编译这章代码的“ant”命令。我们会到第 15 章再详细讲解“编译文件(buildfile)”和 Ant (包括应该到哪里下载)，但是如果你已经安装了 Ant (从 *http://jakarta.apache.org/ant* 下载)，那你就可以在命令下用‘**ant**’命令来编译和运行各章程序了。如果还没有安装 Ant，那你可以手动地输入 **javac** 和 **java** 了。

注释和嵌入式的文档

Java 有两种注释方式。一种是由 C++所继承的传统的 C 的注释风格。这些注释由 **/*** 开始，可以跨很多行，用 ***/** 表示结束。提醒一下，很多程序员会在每个注释行前面再加个 ***** ，所以你会常常看到：

```
/* This is a comment
 * that continues
 * across lines
 */
```

但是**/*** 和 ***/**之间的部分会被忽略掉，所以这和下面这种写法没什么区别：

```
/* This is a comment that
continues across lines */
```

第二种是 C++的注释风格。这是一种单行注释，它由 **//** 开始，一直到本行结束。由于简单易用，因此这种注释被广为传播。你无需找完 **/** 之后再找 ***** (只要按两下就行了)，也不用去关闭注释。所以你经常会看到：

```
// This is a one-line comment
```

注释文档

Java 有一些很好的想法，其中一条就是，写代码并不是唯一重要的事情——制作文档至少和它同等重要。但是文档工作的难点在于维护。如果文档与程序是相互分离的，那么每次修改完程序之后，你还得跟着改文档，这就很麻烦了。解决方案似乎很简单，把代码同文档联系起来。最简单的办法是把它们放到同一个文件里。但是要完成这个步骤，你就得用一种特殊的语法来标注文档，然后用一个工具把注释提取出来，再形成能用的格式。这就是 Java 作的。

这个提取注释的工具就是 *javadoc*，它会随 JDK 一道安装。它用了一些 Java 编译器的技术来查找你放在程序里面的特殊的注释标记。它不仅能提取这些标注的信息，而且还会把注释周围的类名和方法名都给提取出来。这样，你就能用最少的工作量来生成一份还过得去的文档了。

Javadoc 会输出 HTML 文件，这样你就能用 Web 浏览器看了。这样，你就能只创建和维护一份源文件，然后让 javadoc 自动生成文档了。由于有了 javadoc，我们就有了创建文档的标准，于是我们就有理由要求所有的 Java 类库都必须带有文档。

此外，如果你想对这些信息进行一些特殊处理(比如输出另一种格式)，你还可以编写你自己的，被成为 *doclets* 的 javadoc 处理程序。Doclets 会在第十五章作介绍。

下面只是对 javadoc 的基础知识做一下介绍。JDK 文档里有完整的介绍，你可以到 *java.sun.com* 去下载一份(注意一下，JDK 没有包括文档；你必须单独下载其文档)。解开之后，查看“tooldocs“子目录，(或者点击“tooldocs”链接)。

语法

所有 javadoc 命令都由**/**** 注释符开始。结束的标记同普通注释符相同，也是 ***/** 。Javadoc 有两种主要的用法：嵌入式的 HTML 和用“文档标记(doc tags)”。“独立的文档标记(*Standalone doc tags*)”是一些由‘**@**’打头的命令，它们会被放在注释行的开头。(但是打头的那个‘*****’则被忽略了。)“内嵌式的文档标记(*inline doc tags*)”则可以出现在任何地方，它也是由‘**@**’开始的，但是要用花括号括起来。

注释文档根据它所注释的内容，分成三“类”：类，变量和方法。也就是说，类的注释一定要出现在类定义的前面；变量注释要出现在变量定义的前面；而方法注释则要出现在方法定义的前面。注释和定义之间不能有任何东西，就像下面那样：

```
/** A class comment */
public class DocTest {
  /** A variable comment */
  public int i;
  /** A method comment */
  public void f() {}
}
```

注意，javadoc 只会处理 **public** 和 **protected** 的成员的注释文档。它不会去管 **private** 和 package 访问权限(第 5 章讲)的成员的注释。(不过你也可以使用 **–private** 标记，将 **private** 成员也包括进去。)这种

做法是有道理的，因为从客户程序员的角度来看，他们只能访问到这个文件的 **public** 和 **protected** 成员。不过输出会包括所有 **class** 的注释。

这些代码会生成 HTML 文件，它会同其它 Java 文档一样，有着相同的标准格式，因此用户查找类的时候就会觉得很舒服。你应该打一遍上述那段程序，然后用 javadoc 处理一下，看看会输出什么样的 HTML 文件。

嵌入式的 HTML

Javadoc 会把 HTML 命令送到生成的 HTML 文档里面。这样你就能充分利用 HTML 的功能了；不过这个功能主要还是用来排版的，就像这样：

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

还可以像写其它 Web 文档那样，用 HTML 来为普通文本排版：

```
/**
* You can <em>even</em> insert a list:
* <ol>
* <li> Item one
* <li> Item two
* <li> Item three
* </ol>
*/
```

注意，在文档注释里面，每行开头的星号和后面跟的空格都会被 javadoc 抛掉。Java 为一切都重排格式，这样它才能符合标准的文档格式。不要在嵌入的 HTML 里面使用像**<h1>** 或 **<hr>**之类的抬头标记，因为 javadoc 会插入它自己的抬头，而你插入的会干扰其工作的。

所有类型的注释文档——类，变量，以及方法——都支持嵌入的 HTML。

标记举例

下面是一些制作文档时用的 javadoc 的标记。在用 javadoc 之前，你应该先去看看 JDK 文档，对 javadoc 的各种功能有一个全面的了解。

@see: 引用其它类

@see 能让你引用文档中的其它类。Javadoc 会用**@see** 标记生成链接到其它文档的 HTML 链接。它的形式是：

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

每一行都会在生成的文档里面加上一个“See Also”的超链接。Javadoc 不会去检查你所给出的超链接，因此也无法保证它们都是有效的。

{@link *package.class#member label*}

与**@see** 很相似，只是它是内嵌式的文档标记，而且 *label* 是作为超链接的文本而不是“See Also”出现的。

{@docRoot}

返回以文档目录为根的相对路径。用于链接目录中的页面。

{@inheritDoc}

把与这个类最近的那个基类的文档继承到当前文档中。

@version

用法是：

```
@version version-information
```

你可以在 **version-information** 里面填任何你觉得合适的东西。在命令行下用 **–version** 参数调用 javadoc 之后，就能让生成带版本信息的 HTML 文档了。

@author

用法是：

```
@author author-information
```

其中 **author-information** 应该是你的姓名，当然也可以包括 email 地址，以及其它信息。在命令行下用 **–author** 标记调用 javadoc，就能生成带作者信息的 HTML 文档了。

你可以放多个作者标记，但是必须连续放。在 HTML 文件里，所有作者信息都放在一起。

@since

这个标记能让你标明，程序是从哪一版开始提供某种特性的。在 Java 的 HTML 文档里面，你会看到它是用来表示 JDK 的版本的。

@param

用于标注方法，其用法是：

```
@param parameter-name description
```

其中 **parameter-name** 表示方法的参数列表，而 **description** 是一段可以分成好几行的文本。只有碰到新的文档标记， description 才算结束。**@param** 标记可以有好几个，每个标记表示一个参数。

@return

用于标注方法，其用法是：

```
@return description
```

其中的 **description** 会告诉你返回值的信息。它可能会跨好几行。

@throws

异常会放到第九章讲。简单地说，异常就是方法运行失败的时候，会被“抛出来”的对象。虽然每次失败只会抛出一个异常，但是一个方法可以抛出很多种异常，因此要把所有异常都写出来。异常标记的用法是：

```
@throws fully-qualified-class-name description
```

其中的 **fully-qualified-class-name** 必须是方法所声明的异常的名字，而 **description** (可以有好几行长)则会告诉你，是什么原因导致方法产生这个异常。

@deprecated

这是用来表示程序的某个特性已经被新的，改进过的特性所替代了。Deprecate 标记建议用户不再使用这个特性，因为未来这个特性可能会

从程序里面取出。客户程序使用了 **@deprecated** 标记的方法会导致编译器报一个警告消息。

文档举例

下面还是原先那个 Java 程序，只是加上了文档注释：

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

/** The first Thinking in Java example program.
 * Displays a string and today's date.
 * @author Bruce Eckel
 * @author 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());
  }
} ///:~
```

第一行用的是我自己的标记‘**//:**’，这个标记是专门用来注释源文件的路径信息(在这里，**c02** 表示第 2 章) 及文件名的。[15]最后一行也有注释，这次是‘**///:~**’，它表示源代码到此结束，这样经过编译器的检查并且运行通过之后，它就能被自动地加进本里。

编程风格

Code Conventions for the Java Programming Language[16] 所推荐的风格是，用第一个字母大写来表示这是类的名字。如果类名由多个单词组成，那么它们应该连在一起(也就是说，名字里面不要有下划线)，并且其中的每个单词的第一个字母要大写，就像这样：

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

这种风格有时被称为“camel-casing”。其它东西：方法，数据成员(成员变量)，对象的 reference，其推荐的风格是，除了首字母小写，其它部分与类名完全相同。例如：

```
class AllTheColorsOfTheRainbow {
```

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

用户也要打同样长的名字，所以行行好。

你会看到，在 Java 类库的代码里面，Sun 放花括号的位置同本书的完全相同。

总结

本章的目标只是想让你知道怎样写简单的 Java 程序。你对这个语言及其基本概念已经有了一个大致的了解。但是，迄今为止，你看到的代码都是在说“干这个，然后干那个，再做别的什么”。但是怎样才能让程序自己决定呢？举例来说，“如果这么做会得到红色，就做这些；否则，就做那些”。下一章我们就讲 Java 是如何完成这些基本的编程任务的。

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. 参照本章的 **HelloDate.java** 编写一个能打印“hello, world”的程序。这个类只需要一个方法(就是执行程序的时候要用到的“main”。)记着，它必须是 **static** 的，而且还要有参数，虽然我们不会用这些参数。用 **javac** 编译程序，用 **java** 运行程序。如果你不是用 JDK，而是别的什么开发环境，那么就去看看它是怎样编译和运行程序的。
2. 找出与 **ATypeName** 相关的程序片断，把它改写成能编译，能运行的程序。
3. 将 **DataOnly** 改写成能编译能运行的程序。
4. 修改练习 3，使其能对 **DataOnly** 的数据赋值，然后用 **main( )**打印这些值。
5. 编写一个能调用本章所定义的 **storage( )**方法的程序。
6. 将 **StaticFun** 改写成能运行的程序。
7. 编写一个能打印三个命令行参数的程序。要做到这点，你必须用数组下标来查询 **String** 数组。
8. 将 **AllTheColorsOfTheRainbow** 改写成能编译能运行的程序。
9. 找出第二个，也就是带注释文档的那个 **HelloData.java**。运行 **javadoc**，然后用 Web 浏览器看看它生成的文档。

10. 将 docTest 改写成能编译的程序，然后运行 **javadoc**。用 Web 浏览器检查一下其生成的文档。
11. 在练习 10 的文档上添加 HTML 格式的列表。
12. 为练习 1 的程序添加注释文档。用 **javadoc** 生成注释文档，然后用 Web 浏览器查看文档。
13. 找出第四章的 **Overloading.java** 例程，加上 javadoc 文档。用 **javadoc** 提取注释文档，生成 HTML 文件，然后用 Web 浏览器观察。

---

[10] 这里可能会有分歧。有些人说“很明显，这是指针，”但实际上他讲的是其背后的实现。从语法角度上讲，相比指针，Java 的 reference 与 C++的引用(reference)更接近。由于 C++的引用同 Java 的 reference 有一些非常重大的区别，所以我在本书的第一版中，自创了一个新的术语，“handle”。我是从 C++转过来的，我觉得 C++程序员可能是 Java 最大的听众群，所以我不想把他们搞胡涂。在第二版中，我决定还是用大家都接受的“reference”术语。不管是谁，要从 C++转过来，需要适应的东西绝不止 reference 术语这一样，因此他应该全心全意地投入。但是，居然会有人对“reference”都表示反对。我在一本书中读到这么一段，“要说 Java 支持 by reference 传递，那就大错特错了”，因为 Java 对象的标识符 (根据那个作者)实际上是“object references”。而且(他继续道)所有的东西实际上是以 by value 方式传递的。所以你不是在传 reference， 而是“以 by value 方式在传 object reference”。为了追求精确，他选择了如此拗口的说法。但是我觉得我在没有破坏任何东西的前提下，简化了这一概念(当然吹毛求疵的人会说我在撒谎，不过我觉得我是在提供了一种更恰当的抽象。)

[11] 你很快就会学到，**static** 方法的调用不是通过对象，而是通过类的。

[12] 除了前面提到的“特殊”数据类型，**boolean**，**char**，**byte**，**short**，**int**，**long**，**float**，以及 **double**。总之，虽然你想传对象，但是真正被传递的是对象的 reference。

[13] 本书的 CD 没有包括 Sun 的 Java 编译器和文档。它们像是会定期更新。所以你还是自己去下载吧，这样才能得到最新的版本。

[14] IBM 的“jikes”编译器就是一种很常见的非主流系统，它要比 Java 的 javac 快出不少。

[15] 起初我用 Python(见 www.Python.org)写了个程序，它会根据这些信息将代码文件提取出来，然后放入合适的子目录中，再创建 makefile。在这一版中，所有的文件都保存在 CVS(Concurrent Version System)里面了，它会通过 VBA 的宏，自动合成到这本书里。从代码维护的角度而言，新方法似乎更方便，这主要还是因为 CVS。

[16] http://java.sun.com/docs/codeconv/index.html。为了照顾书的篇幅和课堂演示的时间限制，我就没法照搬这些准则了。