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

www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

5: 隐藏实现

在面向对象的设计中，最关键的问题就是“将会变和不会变的东西分离开来。”

这一点对类库尤为重要。类库的使用者(客户程序员)应该能完全仰赖类库，他们知道，即使类库出了新版本，他们也不必重写代码。另一方面，类库的创建者也应该可以在确保不影响客户程序员代码的前提下，保留对类库作修正和改进的权利。

要达到上述目的，可以使用约定。比方说，类库的开发人员必须遵守：修改类的时候不删除现有的方法，因为这可能会影响客户程序员的代码。但是还有一些更棘手的问题。就拿成员数据来说，类库的开发人员又怎么知道客户程序员会使用哪些数据呢？对于那些只与类的内部实现有关的，不应该让客户程序员使用的方法来说，情况也一样。但是，如果类库的开发人员想用一种新的实现来替换旧的，那他又该怎么做呢？对类的任何修改都可能会破坏客户程序员的代码。这样，类库的开发人员就被套上了紧箍咒，什么都不能改了。

为了解决这个问题，Java 提供了访问控制符(*access specifier*)，这样类库的开发人员能告诉客户程序员，他们能用什么，不能用什么了。访问控制权限从松到紧依次是 **public**，**protected**，package 权限(也就是不给任何关键词)，以及 **private**。读了上面那段，你可能会认为，作为类库的设计者，你应该尽可能的把所有东西都做成“**private**”的，并且只公开你想让客户程序员使用的方法。完全正确！尽管对于那些用其它语言(特别是 C)编程，并且已经习惯了不受限制地访问任何东西的人来说，这么做通常是有违常理的。读过本章之后，你就会对 Java 的访问控制更有信心了。

但是，什么是组件类库(library of component)以及怎样去控制“谁能访问类库中组件”的问题还没有完全解决。还有一个问题，就是组件是怎样被捆绑成一个联系紧密的类库单元的。这是由 Java 的 **package** 关键词控制的，此外类是不是属于同一个 package，还会对访问控制符产生影响。所以，我们将从怎样将类库组件(library components)放入 package 里入手，开始本章的学习。接下来，你就能完全理解访问控制符的意思了。

package: 类库的单元

当你使用 **import** 关键词引入一个完整的类库的时候，这个 package 就能为你所用了，例如

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


www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

会把 Java 标准版里的工具类库(utility library)全都引进来。比如，**java.util** 里面有一个 **ArrayList** 类，因此你既可以用全名 **java.util.ArrayList**(这样就用不着 **import** 语句了)，也可以直接写 **ArrayList** 了(因为已经有了 **import**)。

如果你只想引入一个类，那你可以在 **import** 语句里面指名道姓地引用类了。

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

现在你就可以直接使用 **ArrayList** 而不用添加任何限定词了。但是 **java.util** 里面的其它的类就不能用了。

之所以要使用 import，是因为它提供了一种管理名字空间(*name spaces*)的机制。类的所有成员的名字都是相互独立的。**A** 类里面的 **f( )** 方法不会同 **B** 类里面，有着相同“调用特征(signiture，即参数列表)”的 **f( )**相冲突。但是类的名字呢？假设你创建了一个 **Stack** 类，并且把它装到一台已经有了一个别人写的 **Stack** 类的机器上，那又会发生什么事呢？Java 之所以要对名字空间拥有完全的控制，就是要解决这种潜在的名字冲突，并且能不受 Internet 的束缚，创建出完全唯一的名字。

到目前为止，本书所举的都是单文件的例子，而且都是在本地运行的，因此没必要使用 package。(在这种情况下，类的名字是放在“default package”的名下的。) 当然这也是一种做法，而且为了简单起见，本书的其余章节也尽可能使用这种方法。但是，如果你打算创建一个，能同机器上其它 Java 程序相互兼容的类库或程序，你就得考虑一下如何避免名字冲突了。

Java 的源代码文件通常被称为编译单元(*compilation unit* 有时也称翻译单元 *translation unit*)。每个编译单元都必须是一个以 **.java** 结尾的文件，而且其中必须有一个与文件名相同的 **public** 类 (大小写也必须相同，但是不包括 **.java** 的文件扩展名)。每个编译单元只能有一个 **public** 类，否则编译器就会报错。如果编译单元里面还有别的类，那么这些类就成了这个主要的 **public** 的类的“辅助”类了，这是因为它们都不是 **public** 的，因此对外面世界来说它们都是看不到的。

编译**.java** 文件的时候，它里面的每个类都会产生输出。其输出文件的名字就是**.java** 文件里的类的名字，但是其扩展名是**.class**。这样，写不了几个**.java** 文件就会产生一大堆**.class** 文件。如果你有过用编译语言编程的经验，那么你可能会对这个过程感到习以为常了：先用编译器生成一

www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

大堆中间文件(通常是“obj”文件)，然后再用 linker(创建可执行文件)或 librarian(创建类库)把这些中间文件封装起来。但是，Java 不是这样工作的。一个能正常工作的程序就是一大堆**.class** 文件，当然也可以(用 Java 的 **jar** 工具)把它们封装和压缩成 Java ARchive (JAR)文件。Java 解释器会负责寻找，装载和解释[26]这些文件的。

类库就是一组类文件。每个文件都有一个 **public** 类 (不是一定要有 **public** 类，但通常都是这样)，因此每个文件都代表着一个组件。如果你想把这些组件(都在它们自己的那个**.java** 和**.class** 文件里)都组织起来，那就应该用 **package** 关键词了。

当你把：

```
package mypackage;
```

放到文件开头的时候 (如果要用 **package**，那么它必须是这个文件的第一个非注释的行)，你就声明了，这个编译单元是 **mypackage** 类库的组成部分。或者换一种说法，你要表达的意思是，这个编译单元的 **public** 类的名字是在 **mypackage** 的名字之下的(under the umbrella of the name **mypackage**)，任何想使用这个类的人必须使用它的全名，或者用 **import** 关键词把 **mypackage** 引进来(用前面讲的办法)。注意 Java 的约定是用全小写来表示 package 的名字，中间单词也不例外。

举例来说，假设这个文件的名字是 **MyClass.java**。于是文件里面可以有，而且只能有一个 **public** 类，而这个类的名字只能是 **MyClass** (大小写都要相同)：

```
package mypackage;
public class MyClass {
  // . . .
```

现在如果有人想要用 **MyClass**，或者 **mypackage** 里面的其它 **public** 类，那他就必须使用 **import** 关键词来引入 **mypackage** 下的名字了。还有一个办法，就是给出这个类的全名：

```
mypackage.MyClass m = new mypackage.MyClass();
```

用 **import** 可以让代码显得更清楚一点：

```
import mypackage.*;
```


www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

```
// . . .
MyClass m = new MyClass();
```

作为类库的设计者，你得记住，**package** 和 **import** 这两个关键词的作用是要把一个单独的全局名字空间分割开来，这样不论 Internet 上有多少人在用 Java 编程，你就都不会碰到名字冲突的问题了。

创建独一无二的 package 名字

可能你也发现了，由于 package 没有被真的“封装”成一个单独的文件，而 package 又是由很多 **.class** 文件组成的，因此事情就有点乱了。要解决这个问题，较为明智的做法是把所有同属一个包的 **.class** 文件都放到一个目录里；也就是利用操作系统的层次文件结构来解决这个问题。这是 Java 解决这个问题的方法之一；后面要介绍的 **jar** 程序是另一个解决办法。

将 package 的文件收进一个单独的子目录里还解决了另外两个问题：创建独一无二的 package 名字，以及帮助 Java 在复杂的目录结构中找到它们。我们已经在第 2 章讲过了，这是通过将 **.class** 文件的路径信息放到 **package** 的名字里面来完成的。Java 的约定是 **package** 名字的第一部分应该是类的创建者的 Internet 域名的反写。由于 Internet 域名的唯一性是有保证的，因此只要你遵守这个约定，**package** 的名字就肯定是唯一的，这样就不会有名字冲突的问题了。(除非你把域名让给了别人，而他又用同一个域名来写 Java 程序。)当然，如果你还没有注册域名，那你完全可以编一个(比如用你的姓和名)，然后用它来创建 package 的名字。如果你打算要发布 Java 程序，那么还是应该稍微花点精力去搞个域名。

这个技巧的第二部分是把 **package** 的名字映射到本地机器的目录，这样当你启动 Java 程序，需要装载 **.class** 文件的时候 (当程序需要创建某个类的对象，或者第一次访问那个类的 **static** 成员的时候，它会动态执行这个过程的)，它就知道该在哪个目录寻找这个 **.class** 文件了。

Java 解释器是这样工作的。首先，它要找到 CLASSPATH[27]环境变量 (这是通过操作系统设置的，有时 Java 安装程序或者 Java 工具的安装程序会为你设置)。CLASSPATH 包含了一个或多个目录，这些目录会被当作根目录供 Java 搜索**.class** 文件。从这个根目录出发，解释器会将 package 名字里的每个点都换成斜杠 (因此，根据操作系统的不同，**package foo.bar.baz** 就被转换成 **foo\bar\baz** 或 **foo/bar/baz**，或者其它可能的形式)，这样它生成了以 CLASSPATH 为根的相对路径。然后这些路径再与 CLASSPATH 里的各条记录相连。

www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

这才是 Java 用 package 的名字寻找**.class** 文件的地方。(此外，它还会根据 Java 解释器所在的位置查找一些标准目录。)

为了能讲得更清楚，就拿我的域名 **bruceeckel.com** 举例。它倒过来就是 **com.bruceeckel**，这样我写的类就有了全球唯一的名字了。(过去，com，edu，org 这些扩展，在 Java 的 package 名字里面是要大写的，但是 Java 2 作了改进，所以现在 package 的名字都是小写的。)我还可以进一步分下去，创建一个名为 **simple** 的类库，所以 package 的名字是：

```
package com.bruceeckel.simple;
```

现在，你就能用这个 package 的名字来管下面这两个文件了：

```
//: com:bruceeckel:simple:Vector.java
// Creating a package.
package com.bruceeckel.simple;

public class Vector {
  public Vector() {

System.out.println("com.bruceeckel.simple.Vector");
  }
} ///:~
```

等到你要自己写 package 的时候，你就会发现，**package** 语句必须是文件里的第一个非注释行。第二个文件看上去非常相似：

```
//: com:bruceeckel:simple:List.java
// Creating a package.
package com.bruceeckel.simple;

public class List {
  public List() {
    System.out.println("com.bruceeckel.simple.List");
  }
} ///:~
```

在我的机器上这两个文件都放在这个子目录里：

```
C:\DOC\JavaT\com\bruceeckel\simple
```


www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

只要看一遍这个路径，你就会发现 package 的名字 **com.bruceeckel.simple**，但是这个路径的前面部分又是什么呢？这是由 CLASSPATH 环境变量控制的，在我的机器上，它是：

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

你会看到 CLASSPATH 可以有好几个可供选择的搜索路径。

但是，使用 JAR 文件的时候会有一点变化。除了要告诉它该到哪里去找这个 JAR 文件，你必须将文件名放到 CLASSPATH 里面。所以对名为 **grape.jar** 的 JAR 来说，CLASSPATH 应该包括：

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

设完 CLASSPATH 之后，下面这个文件就可以放在任何目录里了：

```
//: c05:LibTest.java
// Uses the library.
import com.bruceeckel.simpletest.*;
import com.bruceeckel.simple.*;

public class LibTest {
  static Test monitor = new Test();
  public static void main(String[] args) {
    Vector v = new Vector();
    List l = new List();
    monitor.expect(new String[] {
      "com.bruceeckel.simple.Vector",
      "com.bruceeckel.simple.List"
    });
  }
} ///:~
```

当编译器碰到了 **simple** 类库的 **import** 语句的时候，它就开始在 CLASSPATH 所给出的目录下搜索，先找 com\bruceeckel\simple 子目录，再找编译后的文件(**Vector** 就找 **Vector.class**，**List** 就找 **List.class**)。注意 **Vector** 和 **List** 类，以及其中要用的方法都必须是 **public** 的。

对 Java 的初学者来说，设置 CLASSPATH 曾经是一桩非常棘手的事(至少我开始的时候是这样的)，所以 Sun 在 Java 2 的 JDK 里面作了一些改进，让它变得稍微智能一些。你会发觉安装之后，即使不设置 CLASSPATH，它也能编译和运行一些基本的 Java 程序。然而要编译和

运行本书的源代码(可以从 *www.BruceEckel.com* 下载)，你就必须将这些代码的根目录加到 CLASSPATH 里面。

冲突

如果两个“*”所引入的类库都包括一个同名的类，那又会怎样呢？举例来说，假设有个程序：

```
import com.bruceeckel.simple.*;
import java.util.*;
```

由于 **java.util.*** 也包括了一个 **Vector** 类，因此这就有可能会引发冲突。然而，只要你不写会引起冲突的代码，一切会 OK——这种做法很好，因为不然的话，你得为了避免根本不可能发生的冲突而多写很多代码。

但是如果你要创建一个 **Vector** 的话，冲突就真的会来了：

```
Vector v = new Vector();
```

你指的是那个 Vector 类呢？编译器不知道，读代码的人也不知道。所以编译器就报错了，它会要你明确地指明这是哪个类。比方说，如果我要使用 Java 标准的 **Vector**，我就必须说：

```
java.util.Vector v = new java.util.Vector();
```

由于这种写法(再加上 CLASSPATH)已经能完全指明 **Vector** 的位置了，因此除非你还要使用 **java.util** 的其它类，否则就不必再使用 **import java.util.***。

一个自定义的工具类库

有了这些知识，你就能创建你自己的工具类库以减少甚至彻底消除重复代码了。假设我们要为 **System.out.println( )** 创建一个别名以减少打字的量。这可以是 **tools** package 的一部分：

```
//: com:bruceeckel:tools:P.java
// The P.rint & P.rintln shorthand.
package com.bruceeckel.tools;

public class P {
  public static void rint(String s) {
```


www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

```
    System.out.print(s);
  }
  public static void rintln(String s) {
    System.out.println(s);
  }
} ///:~
```

这种简写形式既能以加换行符的形式(**P.rintln()**)，也能以不加换行符的形式(**P.rint()**)打印 **String**。

你能猜到，这个文件一定是位于 CLASSPATH 的某个目录的 **com/bruceeckel/tools** 子目录下。编译之后，你就能在系统的任何地方用 **import** 语句引入 **P.class** 文件了：

```
//: c05:ToolTest.java
// Uses the tools library.
import com.bruceeckel.tools.*;
import com.bruceeckel.simpletest.*;

public class ToolTest {
  static Test monitor = new Test();
  public static void main(String[] args) {
    P.rintln("Available from now on!");
    P.rintln("" + 100); // Force it to be a String
    P.rintln("" + 100L);
    P.rintln("" + 3.14159);
    monitor.expect(new String[] {
      "Available from now on!",
      "100",
      "100",
      "3.14159"
    });
  }
} ///:~
```

注意，不论哪种对象，只要放进了 **String** 表达式，它就会被强制转化为这个对象的 **String** 表示形式了；在上述程序中，把空的 **String** 放进表达式就是为了达到这个目的。但是这却让我们注意到了一个有趣的现象。如果你用 **System.out.println(100)**的方式进行调用，那么它就不会把参数转换成 **String** 了。经过一番特别的重载之后，你可以也可以让 **P** 具备这样的功能(这是本章练习的要求)。

所以从现在开始，只要你写了什么新的，能派上用场的工具，你就可以把它加到你自己的 **tools** 或 **util** 目录。

使用 import 来改变程序的行为方式

www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

Java 没有实现 C 的“条件编译(*conditional compilation*)”。所谓条件编译就是，不用修改源代码，只用一个开关就能让程序产生不同的行为方式。Java 之所以要剔除这个特性，可能是因为它主要是用来解决 C 的跨平台的问题的：程序的不同部分会根据平台的不同而采用不同的编译方式。由于 Java 的初衷就是要跨平台，因此这种特性就变得多余了。

但是条件编译还有一些别的很有价值的用途。最常见的就是用它来调试代码。调试功能在开发版里是能用的，但在正式版里则被禁了。你可以通过修改 **package** 来切换调试版和发布版所使用的代码，来达到上述目的。这种技巧能用于任何类型的“有条件的代码(conditional code)”。

使用 **package** 的忠告

值得注意的是，每次创建 package 给它起名的时候，你也隐含地设置了一个目录结构。这个 package 必须保存在由它的名字所指示的目录里，而这个目录又必须在 CLASSPATH 下面。刚开始做 **package** 的实验的时候，可能会让人觉得有些泄气，因为除非你严格遵守了 package 的名字就是目录路径这一规则，否则即便这个类就呆在同一个目录里，你也会得到一大串莫名其妙的，告诉你找不到这类的消息。如果你得到这种消息，就先把 **package** 语句注释掉，如果它能运行了，你就知道问题出在哪里了。

Java 的访问控制符

编程的时候，**public**，**protected** 以及 **private** 这三个 Java 访问控制符，应该放在类的每个成员的定义部分的前面，不管这个成员是数据还是方法。一个访问控制符只管它所定义的这一项。这同 C++形成了鲜明的对比。C++的访问控制符会一直管下去，直到出现另一个。

每样东西都会有一个访问控制符，不是这个就是那个。下面我们就从默认的访问权限开始，学习各种访问控制符的权限。

package 访问权限

如果像本章之前的程序那样，根本就不给访问控制符，那情况又会如何呢？默认的访问权限没有关键词，但通常还是把它称为 package 权限(*package access*，有时也称为“friendly”)。它的意思是，所有同属这个 package 的类都能访问这个成员，但是对那些不属于这个 package 的类来说，这个成员就是 **private** 的了。由于编译单元——也就是源文件——只能属于一个 package，因此同一个编译单元里的各个类，自动就能通过 package 权限进行相互访问了。

package 权限能让你将相互关联的类组织成 package，这样它们之间就能很方便地进行访问了。当你把类放到 package 的时候，也就是说赋予

www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

它的 package 权限的成员以相互访问的权利的时候，你就“拥有”了这个 package 的代码。只有你的程序才能对你的其它程序进行 package 权限的访问，这是很合逻辑的。可以这么说，有了 package 权限，将类组织成 package 才变得有意义。在很多语言里，你可以用任何方式来组织文件，但是 Java 会强制你用一种更合理的方式把它们组织起来。此外，你还可能会要将一些不应该能访问当前 package 的类排除出去。

哪些代码可以访问类的成员，是由类自己控制的。没有什么能“穿墙而入”的神奇办法。另一个 package 的代码不能说“嗨，我是 **Bob** 的朋友”，然后要求看 **Bob** 的 **protected** 的，package 权限的，或者 **private** 的成员。如果你想让别人能访问到这个成员，那唯一办法就是：

1. 把这个成员做成 **public** 的。这样任何人，任何地方就都能访问到它了。
2. 不放任何访问控制符，赋予这个成员 package 权限，然后往 package 里面放其它类。这样，这个 package 的其它类就能访问这个成员了。
3. 我们会在第 6 章讲继承。届时你会看到，继承类除了能访问父类的 **public** 成员之外，还可以访问其 **protected** 成员(但是不能访问 **private** 成员)。只有当两个类都同属一个 package 的时候，它才能访问 package 成员。不过你现在还不必为此操心。
4. 提供“访问器/修改器”方法(accessor/mutator 方法，也被称为“get/set”方法)。以 OOP 的观点衡量，这是最合理的做法，而且也是 JavaBean 的基础，我们会到第 14 章再讲。

public:访问接口的权限

当你使用 **public** 关键词的时候，你的意思是：任何人，尤其是那些要使用这个类库的客户程序员，都能访问那个紧跟在 **public** 后面声明的成员。假设你定义了一个叫 **dessert** 的 package，其中有下面这个编译单元：

```
//: c05:dessert:Cookie.java
// Creates a library.
package c05.dessert;

public class Cookie {
  public Cookie() {
   System.out.println("Cookie constructor");
  }
  void bite() { System.out.println("bite"); }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

记住，**Cookie.java** 所生成的 class 文件必须保存在 CLASSPATH 项下某个目录的 **c05** 子目录(表示本书的第 5 章)的 **dessert** 子目录下。千万不要想当然地认为 Java 总是会从当前的目录开始查找。如果你不把‘**.**’放到 CLASSPATH，Java 是不会查找当前目录的。

现在，如果你用 **Cookie** 创建了一个程序：

```
//: c05:Dinner.java
// Uses the library.
import com.bruceeckel.simpletest.*;
import c05.dessert.*;

public class Dinner {
  static Test monitor = new Test();
  public Dinner() {
   System.out.println("Dinner constructor");
  }
  public static void main(String[] args) {
    Cookie x = new Cookie();
    //! x.bite(); // Can't access
    monitor.expect(new String[] {
      "Cookie constructor"
    });
  }
} ///:~
```

由于 **Cookie** 的构造函数是 **public** 的，并且 **Cookie** 这个类也是 **public** 的，因此你可以创建 **Cookie** 对象(我们过一会儿谈 **public** 类的概念。) 但是你不能在 **Dinner.java** 里面访问 **bite( )**，这是因为它是 package 权限的，只能在 **desert** 这个 package 里面访问，因此编译器会禁止你使用它。

默认的 package

或许你会觉得很奇怪，下面这段代码没有遵守规则，怎么也能编译通过：

```
//: c05:Cake.java
// Accesses a class in a separate compilation unit.
import com.bruceeckel.simpletest.*;

class Cake {
  static Test monitor = new Test();
  public static void main(String[] args) {
    Pie x = new Pie();
    x.f();
    monitor.expect(new String[] {
      "Pie.f()"
    });
  }
} ///:~
```


www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

这个目录里面还有一个文件：

```
//: c05:Pie.java
// The other class.

class Pie {
  void f() { System.out.println("Pie.f()"); }
} ///:~
```

刚开始的时候，你可能会把它们视作两个完全不相关的文件，所以会奇怪 **Cake** 怎么能创建 **Pie** 对象，并且调用它的 **f( )**方法的！(注意，你必须把‘**.**’放到 CLASSPATH 里面，这样文件才能顺利地编译通过。)可能你会认为 **Pie** 和 **f( )**都是 package 访问权限的，因此 **Cake** 是不能访问的。它们的确都是 package 权限的——这点没错。之所以能在 **Cake.java** 里面访问 **Pie**，是因为这两个文件都在同一个目录里面，并且都没有明确指明它是属于哪个 package 的。Java 会认为这类文件是属于这个目录的“默认 package”的，因此对这个目录里边的其它文件来说，它们就都是 package 权限的了。

private:你碰都碰不到！

private 关键词的意思是：除非是用这个类(包含这个成员的类)的方法，否则一律不得访问。同一个 package 里的其它类也不能访问 **private** 成员，所以这就有点像是在“作茧自缚”。但是另一方面，一个 package 很可能是由好几个人合作开发的，因此 **private** 能让你根据自己的需要修改那些成员，而不用担心这么做会不会对别的类产生影响。

默认的 package 权限通常已经提供了一种较为合适的隐藏效果；记住，从客户程序员的角度来看，package 权限的成员也是不能访问的。这样正好，因为默认的权限就是你经常要用的那个(而且还是你忘了设置的时候会用那个的)。于是通常情况下，你只要把那些要对客户程序员开放的成员设成 **public** 就行了。结论是，可以先不考虑大量地使用 **private**，因为即使不用，也还过得去。(这点同 C++是截然不同。)但是，始终如一地使用 **private** 还是很重要的，特别是遇到多线程的时候。(到第 13 章就知道了。)

下面是一个运用 **private** 的例子：

```
//: c05:IceCream.java
// Demonstrates "private" keyword.

class Sundae {
```


www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

```
  private Sundae() {}
  static Sundae makeASundae() {
    return new Sundae();
  }
}

public class IceCream {
  public static void main(String[] args) {
    //! Sundae x = new Sundae();
    Sundae x = Sundae.makeASundae();
  }
} ///:~
```

这里演示了一个能发挥 **private** 的特长的例子：你可能要控制对象的创建，并且阻止别人直接访问某个构造函数(或者所有的构造函数)。在上述例程中，你不能通过构造函数来创建 **Sundae** 对象；相反你必须调用 **makeASundae( )**方法来创建。[28]

只有一个方法，当你把它做成 **private** 的时候可以一点心思都不担，这就是类的“helper”方法。这样就能保证，你不会一不小心就把这个方法用到 package 的其它地方，从而造成你自己都不能修改或删除的尴尬了。方法设成 **private** 之后，这项权利就被保留下来了。

对类的 **private** 数据来说，情况也一样。除非你必须开放类的底层实现(出现这种情况的可能性要比你相像的要少的多)，否则就应该将所有的数据都设成 **private** 的。但是这并不是在说，只要类里有了一个某个对象的 **private** 的 reference，那么其它对象就不能有这个对象的 **public** 的 reference 了。(参见附录 A 的别名(aliasing)章节)

protected: 继承的访问权限

要想弄懂 **protected** 访问权限，就得先讲一点后面的东西。首先要告诉你，在讲继承(第 6 章)之前，即使你不用理解这部分内容也可以继续读下去。但是为了叙述的完整性，我们还是先简单地讲一下，再用 **protected** 举一个例子。

protected 关键词所处理的是一种被称为继承(*inheritance*)的概念，所谓继承就是选一个现成的类——我们称之为基类(*base class*)——然后在不改变已有类的前提下，往里面添加新的成员。你还可以修改已有类的成员的行为方式。要继承一个已有的类，你必须说明新的类 **extends** 一个已有的类，就像这样：

```
class Foo extends Bar {
```


www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

接下来的定义就完全相同了。

如果你创建了一个新的 package，并且其中某个类还继承了另一个 package 里面的类，那么你只能访问原先那个 package 的 public 成员。(当然如果是在同一个 package 里面继承的话，那么你还可以访问 package 权限的成员。)有时基类的创建者会希望派生类能访问某个成员，而其它类则不能访问。这就是 **protected** 要做的。**protected** 也赋予成员 package 权限——也就是说，同一个 package 里的其它类也可以访问 **protected** 元素。

如果你回上去看 **Cookie.java**，就会发现下面这个类是不能调用 package 权限的 **bite( )**的：

```
//: c05:ChocolateChip.java
// Can't use package-access member from another
package.
import com.bruceeckel.simpletest.*;
import c05.dessert.*;

public class ChocolateChip extends Cookie {
  private static Test monitor = new Test();
  public ChocolateChip() {
   System.out.println("ChocolateChip constructor");
  }
  public static void main(String[] args) {
    ChocolateChip x = new ChocolateChip();
    //! x.bite(); // Can't access bite
    monitor.expect(new String[] {
      "Cookie constructor",
      "ChocolateChip constructor"
    });
  }
} ///:~
```

继承有一个有趣的特性，就是如果 **Cookie** 类里一个 **bite( )**方法，那么所有继承 **Cookie** 的类里也都有 **bite( )**方法。但是 **bite( )**是 package 权限的，并且在另一个 package 里面，因此我们没法用。当然你可以把它做成 **public** 的，但是这样一来任何人都可以访问这个方法了，而这又不是你所希望的。但是，如果你这样修改 **Cookie**：

```
public class Cookie {
  public Cookie() {
    System.out.println("Cookie constructor");
  }
  protected void bite() {
    System.out.println("bite");
  }
}
```


www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

那么在 **dessert** package 里，**bite( )**仍然是 package 权限的，但是继承 **Cookie** 的类也能访问它了。但它却不是 **public** 的。

接口(Interface)与实现(implementation)

访问权限通常被称为“隐藏实现(*implementation hiding*)”。在将数据和方法集成到了类里的同时，完成“隐藏实现”，这种做法常被称为封装(*encapsulation*)。[29] 其结果就是数据类型有了特征和行为。

有两个重要的原因要让我们为数据类型设置边界。首先就是要告诉客户程序员，他们能使用哪些东西，不能用哪些东西。你可以在系统里构建自己的内部机制，这样就不必担心客户程序员会一不小心就把这部分东西当作接口来用了。

这一点又直接牵涉到了第二个原因，这就是接口与实现的分离。如果这种结构被用于一组程序，那么客户程序员除了能向 **public** 接口发送消息之外就什么也做不了，这样你就能自由地修改那些非 **public** 的 (包括 package 权限，**protected** 或 **private**)的成员，而不用担心会破坏客户代码了。

现在，我们是在面向对象编程的世界中，**class** 的意思实际上是指“一类对象”，就像你在说鱼类或鸟类。所有属于这一类的对象都有某些共同的特征或行为。类就是在描述这些对象是什么样子的，是怎样工作的。

在最早的 OOP 语言，Simula-67 中，关键词 **class** 是用来描述一种新的数据类型的。绝大多数的面向对象的语言都沿用了这个关键词。它点明了 OOP 语言的关键：创建一种新的数据类型，而不仅仅是把数据和方法做在一个模块里。

类是 Java 的 OOP 概念的基础。它也是本书不用粗体表示的关键词之一——要把像“class”这样出现频率极高的词做这种处理会是非常烦人的。

为了让代码显得更有条理，可能你选用这种风格，就是将 **public** 成员都放在类的开头，接下来是 **protected** 成员，然后是 package 权限的，最后是 **private** 成员。这样做的好处就是，当用户从上到下读代码的时候，会先看到对他们最重要的东西(就是能在文件以外访问的 **public** 成员)。而当他们遇到非 **public** 成员的时候，就会知道这是类的内部实现部分，这样就可以不读下去了。

```
public class X {
  public void pub1() { /* . . . */ }
  public void pub2() { /* . . . */ }
```


www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

```
  public void pub3() { /* . . . */ }
  private void priv1() { /* . . . */ }
  private void priv2() { /* . . . */ }
  private void priv3() { /* . . . */ }
  private int i;
  // . . .
}
```

由于接口和实现仍然是混在一起的，因此这种写法只能部分地减轻读者的负担。也就是说，你还得读源代码，也就是其实现部分，因为它就在类里。此外 javadoc(第 2 章讲的)生成的注释文档也大大降低了客户程序员要读源代码的必要性。实际上，向用户展示接口应该是“类浏览器(*class browser*)”的工作。所谓类浏览器是一种工具，它能找出所有的类，并且告诉你，应该用什么方法来使用这些类(比如可以用哪些成员)。类浏览器已经成为优秀的 Java 开发工具所必不可少的组成部分了。

类的访问权限

Java 的访问控制符还能用于类，这时它会决定，用户能够使用类库里的哪些类。如果要允许客户程序员使用一个类，你可以用 **public** 关键词来定义这个类。它会控制，客户程序员能否创建这个类的对象。

要想控制类的访问权限，控制符必须放在 **class** 关键词的前面。因此你应该这样写：

```
public class Widget {
```

如果你的类库的名字是 **mylib**，那么客户程序员就能这样使用 **Widget** 了：

```
import mylib.Widget;
```

或者

```
import mylib.*;
```

但是还有一些额外的限制：

www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

1. 每个编译单元(文件)只能有一个 **public** 类。这么做的意思是，每个编译单元只能有一个公开的接口，而这个接口就由其 **public** 类来表示。你可以根据需要，往这个文件里面添加任意多个提供辅助功能的 package 权限的类。但是如果这个编译单元里面有两个或两个以上的 **public** 类的话，编译器就会报错。
2. **public** 类的名字必须和这个编译单元的文件名完全相同，包括大小写。所以对 **Widget** 类，文件名必须是 **Widget.java**，不能是 **widget.java** 或 **WIDGET.java**。如果你不遵守，编译器又要报错了。
3. 编译单元里面可以没有 **public** 类，虽然这种情况不常见，但却是可以的。这时，你就能随意为文件起名字了。

如果 **mylib** 里面还有一个要为 **Widget** 或 **mylib** 的其它 **public** 类提供服务的类，那你又该怎么做呢？你不想为客户程序员写文档，因为你知道，可能过段时间你就会用一个新的类来替换它了。要想能获得这种灵活性，你就必须确保客户程序员不能利用 **mylib** 的内部实现来编程。为了达成这个目标，你只要将 **public** 关键词从类里删掉就行了，这样它就是 package 权限的了。(于是类只能用于 package 内部了。)

创建 package 权限的类时，将类的成员定义成 **private**，仍然会是很明智的——你应该尽量地将成员都设成 **private** 的——但是通常情况下，还是应该将方法的访问权限设成和类的一样(也就是说，方法也设成 package 的)。由于 package 权限的类只会用于 package 的内部，因此实在没办法的时候，只要将这些方法设成 **public** 的就行了。碰到这种情况的时候，编译器会通知你的。

注意，类不能是 **private**(这样除了这个类自己，其它人都不能访问了)或 **protected** 的。[30]因此类只有两种访问权限：package 权限和 **public**。如果你不希望别人访问这个类，你可以将它的构造函数做成 **private** 的，这样除你之外，没人可以创建那个类的对象了。而你则可以使用一个 **static** 方法来创建对象。下面就是一例：

```
//: c05:Lunch.java
// Demonstrates class access specifiers. Make a
class
// effectively private with private constructors:

class Soup {
  private Soup() {}
  // (1) Allow creation via static method:
  public static Soup makeSoup() {
    return new Soup();
  }
  // (2) Create a static object and return a
reference
  // upon request.(The "Singleton" pattern):
  private static Soup ps1 = new Soup();
  public static Soup access() {
    return ps1;
```


www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

```
  }
  public void f() {}
}

class Sandwich { // Uses Lunch
  void f() { new Lunch(); }
}

// Only one public class allowed per file:
public class Lunch {
  void test() {
    // Can't do this! Private constructor:
    //! Soup priv1 = new Soup();
    Soup priv2 = Soup.makeSoup();
    Sandwich f1 = new Sandwich();
    Soup.access().f();
  }
} ///:~
```

迄今为止，绝大多数的方法都是 **void** 或返回 primitive 类型的，所以刚看到这个定义：

```
  public static Soup access() {
    return ps1;
  }
```

的时候，会有点不知所云。方法名字(**access**)前面的那个单词会告诉你，这个方法应该返回什么类型的数据。到目前为止，我们看到最多的是 **void**，它的意思是什么都不返回。但是你也可以让它返回一个对象的 reference，而这就是这段程序的意思。这个方法会返回一个 **Soup** 对象的 reference。

class Soup 演示了，怎样用 **private** 构造函数来禁止用户直接创建某个类的对象。记住，要是你一个构造函数都不写的话，编译器就会为你合成一个默认的构造函数(即无参数的构造函数)。写了默认的构造函数之后，它也不会再为你创建了。构造函数定义成 **private** 之后，就没人能创建那个类的对象了。那么它又该怎样使用呢？上面的例子给出了两种方法。第一种，就是定义一个会创建 **Soup** 对象，并且会返回其 reference 的 **static** 方法。如果你想先做一些操作，再返回对象的 reference，或者要计算一下 **Soup** 对象的数量(可能是为了限制其数量)，那么这种做法还是很有用的。

第二种方法就是我们所说的设计模式(*design pattern*)。设计模式要在的 Thinking in Patterns(*with Java*)这本书里讲，这本书也可以到 *www.BruceEckel.com* 去下载。这里用到的是被称为“singleton”的

www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

模式，因为它只允许你创建一个这种类的对象。这个对象是被当作 **Soup** 类的 **static private** 成员来创建的，因此它有且只有一个对象，而且除非是通过 **public** 的 **access( )**方法，否则没法获取。

正如我们前面所提到的，如果你不写类的访问控制符，那么它就默认是 package 权限的。也就是说，package 中的任何一个类都能创建这个类的对象，但是 package 以外的类就不行了。(记住，同一个目录里面的其它文件，只要没有包含 **package** 语句，就都会被默认为是这个目录的 package 权限的。)但是如果这个类有一个 **public** 的 **static** 成员，那么即便客户程序员不能创建那个类的对象，他们也还可以访问这个 **static** 的成员。

总结

不论参与各方有什么利害关系，能有一个为各方所尊重的界限是很重要的。当你创建类库的时候，你就与类库的使用者，也就是客户程序员们，建立了一种关系。他们也是程序员，他们要用类库来组建一个应用程序，或者在你的类库的基础上构建一个更大的类库。

要是没有规则的话，即便你不想让客户程序员们去直接操控类的某些成员，你也没法去阻止他们。一切都暴露在外面，什么遮盖也没有。

本章主题是怎样用类来构建类库：首先是怎样将类封装成类库，然后是，类是怎样控制它的成员的访问权限的。

曾经有人做过评估，说 C 的项目在达到 50,000 到 100,000 行的时候就开始崩溃了，因为 C 只有一个“名字空间”，而名字冲突会引发额外的管理的负担。但是 Java 语言的 **package** 关键词，package 的命名规范，以及 **import** 关键词，能让你对名字实施完全的控制，因此可以很容易的化解名字冲突。

有两个原因促使我们要对类的成员进行访问权限方面的控制。一是，要禁止用户去碰他们不该碰的东西，也就是那些不属于供用户解决问题之用的接口，而是属于涉及类的内部运作的工具。因此我们说将方法和字段定义成 **private** 的，是对用户提供的一种服务，因为他们能直接了解什么是重要的，什么是可以不去理会的。这简化了他们对类的理解。

第二个也是最重要的原因就是，要让类库的设计者们能在不惊动客户程序员的前提下修改类库的内部运行方式。可能你会先用一种思路来创建类，然后发现重新规划一下能让它跑得更快。如果接口和实现被分得很清楚，而且访问控制也做得很好，那么你就能在做到这一点了。

Java 的访问控制符赋予类的开发人员一种很有价值的能力。用户能够很清楚地知道，哪些是他们可以用的，哪些是他们可以忽略的。但是更重要

www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

的是，这是一种能确保用户不会依赖类的内部实现来编程的手段。作为类的创建者，如果你理解了这一点，就可以随心所欲的修改类的底层实现了，因为你知道客户程序员们无须修改他们的代码；因为他们根本访问不到这部分。

当你能获得了修改类的底层实现的能力的时候，你就有权利随时修改类的设计了。同样你也有了犯错误的权利。无论计划如何周详，设计如何精巧，你都会犯错误。当你知道即使犯了错误也没什么大不了的时候，你就会变得更富实验精神了。于是，你会学得更快，项目也能完成得更早。

类的公开接口是用户实实在在看到的部分，因此在分析和设计阶段，它是类是否“正确”的决定因素。即使是这样，你还是可以做修改的。如果第一次没能给出正确的接口，那以后还可以添加，只是不能删除客户程序员已经用到过的东西了。

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. 写一段会创建 **ArrayList** 对象的程序，不要用 **import java.util.***。
2. 找到“package：类库的单元”一节中与 **mypackage** 有关的代码片断，将它改写成一组能编译运行的 Java 源文件。
3. 找到“冲突”一节的代码片断，将它改写成一个程序，然后验证一下，看看冲突是不是真的会发生。
4. 对本章所定义的 **P** 类做一般化处理，重载 **rint( )**和 **rintln( )**方法，使之能处理各种 Java 数据类型。
5. 创建一个有 **public**，**private**，**protected**，和 package 访问权限的数据的类。创建一个这个类的对象，然后看看，当你要访问这些数据的时候，编译器都会给一些什么消息。提醒一下，同一个目录里面的其它类也是“默认”package 的一部分。
6. 创建一个有 **protected** 数据的类。然后在同一个源文件里创建一个类，这个类要有一个能操控第一个类的 **protected** 数据的方法。
7. 修改“protected：继承的访问权限”一节中的 **Cookie** 类。验证一下，**bite( )**不是 **public** 的。
8. 在“类的访问权限”一节中，找到讲述 **mylib** 和 **Widget** 的代码。然后把这个类库写出来，再写一个不属于 **mylib** package 的类，然后在这个类里创建一个 **Widget** 对象。
9. 创建一个新的目录并且把它加到 CLASSPATH 中。把 **P.class** 文件(编译 **com.bruceeckel.tools.P.java** 生成的)拷贝到这个目录里，然

www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

后修改文件名，里面 **P** 类的名称，以及其方法名。(你可以让它多输出些东西，看看它是怎样工作的。)在另一个目录里面创建一个要使用这个新类的类。

10. 参照 **Lunch.java** 写一个 **ConnectionManager** 类，然后用它去管理一组固定数量的 **Connection** 对象。要做到，客户程序员不能直接创建，而只能通过 **ConnectionManager** 的 **static** 方法来获取 **Connection** 对象。当 **ConnectionManage** 无对象可分配的时候，它会返回 **null** 的 **reference**。用 **main( )**做测试。
11. 在 c05/local 目录创建下面这个文件(假设 CLASSPAH 里面有这个目录)： Create the following file in the c05/local directory (presumably in your CLASSPATH):

```
// c05:local:PackagedClass.java
package c05.local;
class PackagedClass {
  public PackagedClass() {
    System.out.println("Creating a packaged class");
  }
}
```

然后在 c05 以外的目录里创建下面这个文件：

```
// c05:foreign:Foreign.java
package c05.foreign;
import c05.local.*;
public class Foreign {
   public static void main (String[] args) {
      PackagedClass pc = new PackagedClass();
   }
}
```

解释一下编译器为什么会报错。把这个 **Foreign** 类做成 **c05.local** package 的会不会有什么不同？

---

[26] Java 也不是非要用解释器不可。有一些 Java 本地代码编译器能生成单独的可执行的文件。

[27]环境变量要用大写(CLASSPATH)。

[28]这种做法还会产生一个后果：由于只定义了一个默认的构造函数，而且还是 **private** 的，因此要继承这个类就变得不可能了。(这是第 6 章的内容。)

[29]但是人们常会把封装只理解为“隐藏实现”。

www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com

[30]实际上内部类(*inner class*)可以是 private 或 protected，但这是特例。我们会在第 7 章再作讨论。

www.wgqqh.com/shhgs/tij.html
email: shhgs@sohu.com