第 5 章 隐藏实施过程

“进行面向对象的设计时，一项基本的考虑是：如何将发生变化的东西与保持不变的东西分隔开。”

这一点对于库来说是特别重要的。那个库的用户（客户程序员）必须能依赖自己使用的那一部分，并知道一旦新版本的库出台，自己不需要改写代码。而与此相反，库的创建者必须能自由地进行修改与改进，同时保证客户程序员代码不会受到那些变动的影响。

为达到这个目的，需遵守一定的约定或规则。例如，库程序员在修改库内的一个类时，必须保证不删除已有的方法，因为那样做会造成客户程序员代码出现断点。然而，相反的情况却是令人痛苦的。对于一个数据成员，库的创建者怎样才能知道哪些数据成员已受到客户程序员的访问呢？若方法属于某个类唯一的一部分，而且并不一定由客户程序员直接使用，那么这种痛苦的情况同样是真实的。如果库的创建者想删除一种旧有的实施方案，并置入新代码，此时又该怎么办呢？对那些成员进行的任何改动都可能中断客户程序员的代码。所以库创建者处在一个尴尬的境地，似乎根本动弹不得。

为解决这个问题，Java 推出了“访问指示符”的概念，允许库创建者声明哪些东西是客户程序员可以使用的，哪些是不可使用的。这种访问控制的级别在“最大访问”和“最小访问”的范围之间，分别包括：public，“友好的”（无关键字），protected 以及 private。根据前一段的描述，大家或许已总结出作为一名库设计者，应将所有东西都尽可能保持为“private”（私有），并只展示出那些想让客户程序员使用的方法。这种思路是完全正确的，尽管它有点儿违背那些用其他语言（特别是 C）编程的人的直觉，那些人习惯于在没有任何限制的情况下访问

所有东西。到这一章结束时，大家应该可以深刻体会到 Java 访问控制的价值。

然而，组件库以及控制谁能访问那个库的组件的概念现在仍不是完整的。仍存在这样一个问题：如何将组件绑定到单独一个统一的库单元里。这是通过 Java 的 package（打包）关键字来实现的，而且访问指示符要受到类在相同的包还是在不同的包里的影响。所以在本章的开头，大家首先要学习库组件如何置入包里。这样才能理解访问指示符的完整含义。

5.1 包：库单元

我们用 import 关键字导入一个完整的库时，就会获得“包”（Package）。例如：

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

它的作用是导入完整的实用工具（Utility）库，该库属于标准 Java 开发工具包的一部分。由于 Vector 位于 java.util 里，所以现在要么指定完整名称“java.util.Vector”（可省略 import 语句），要么简单地指定一个“Vector”（因为 import 是默认的）。

若想导入单独一个类，可在 import 语句里指定那个类的名字：

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

现在，我们可以自由地使用 Vector。然而，java.util 中的其他任何类仍是不可使用的。

之所以要进行这样的导入，是为了提供一种特殊的机制，以便管理“命名空间”（Name Space）。我们所有类成员的名字相互间都会隔离起来。位于类 A 内的一个方法 f()不会与位于类 B 内的、拥有相同“签名”（自变量列表）的 f()发生冲突。但类名会不会冲突呢？假设创建一个 stack 类，将它安装到已有一个 stack 类（由其他人编写）的机器上，这时会出现什么情况呢？对于因特网中的 Java 应用，这种情况会在用户毫不知晓的时候发生，因为类会在运行一个 Java 程序的时候自动下载。

正是由于存在名字潜在的冲突，所以特别有必要对 Java 中的命名空间进行完整的控制，而且需要创建一个完全独一无二的名字，无论因特网存在什么样的限制。

迄今为止，本书的大多数例子都仅存在于单个文件中，而且设计成局部（本地）使用，没有同包名发生冲突（在这种情况下，类名置于“默认包”内）。这是一种有效的做法，而且考虑到问题的简化，本书剩下的部分也将尽可能地采用它。然而，若计划创建一个“对因特网友好”或者说“适合在因特网使用”的程序，必须考虑如何防止类名的重复。

为 Java 创建一个源码文件的时候，它通常叫作一个“编辑单元”（有时也叫作“翻译单元”）。每个编译单元都必须有一个以.java 结尾的名字。而且在编译单元的内部，可以有一个公共（public）类，它必须拥有与文件相同的名字（包括大小写形式，但排除.java 文件扩展名）。如果不这样做，编译器就会报告出错。每个编译单元内都只能有一个 public 类（同样地，否则编译器会报告出错）。那个编译单元剩下的类（如果有的话）可在那个包外面的世界面前隐藏起来，因为它们并非“公共”的（非 public），而且它们由用于主 public 类的“支撑”类组成。

编译一个.java 文件时，我们会获得一个名字完全相同的输出文件；但对于.java 文件中的每个类，它们都有一个.class 扩展名。因此，我们最终从少量

的.java 文件里有可能获得数量众多的.class 文件。如以前用一种汇编语言写过程序，那么可能已习惯编译器先分割出一种过渡形式（通常是一个.obj 文件），再用一个链接器将其与其他东西封装到一起（生成一个可执行文件），或者与一个库封装到一起（生成一个库）。但那并不是 Java 的工作方式。一个有效的程序就是一系列.class 文件，它们可以封装和压缩到一个 JAR 文件里（使用 Java 1.1 提供的 jar 工具）。Java 解释器负责对这些文件的寻找、装载和解释（注释①）。

①：Java 并没有强制一定要使用解释器。一些固有代码的 Java 编译器可生成单独的可执行文件。

“库”也由一系列类文件构成。每个文件都有一个 public 类（并没强迫使用一个 public 类，但这种情况最很典型的），所以每个文件都有一个组件。如果想将所有这些组件（它们在各自独立的.java 和.class 文件里）都归纳到一起，那么 package 关键字就可以发挥作用）。

若在一个文件的开头使用下述代码：

```
package mypackage;
```

那么 package 语句必须作为文件的第一个非注释语句出现。该语句的作用是指出这个编译单元属于名为 mypackage 的一个库的一部分。或者换句话说，它表明这个编译单元内的 public 类名位于 mypackage 这个名字的下面。如果其他人想使用这个名字，要么指出完整的名字，要么与 mypackage 联合使用 import 关键字（使用前面给出的选项）。注意根据 Java 包（封装）的约定，名字内的所有字母都应小写，甚至那些中间单词亦要如此。

例如，假定文件名是 MyClass.java。它意味着在那个文件有一个、而且只能有一个 public 类。而且那个类的名字必须是 MyClass（包括大小写形式）：

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

现在，如果有人想使用 MyClass，或者想使用 mypackage 内的其他任何 public 类，他们必须用 import 关键字激活 mypackage 内的名字，使它们能够使用。另一个办法则是指定完整的名称：

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

import 关键字则可将其变得简洁得多：

```
import mypackage.*;
// . . .
MyClass m = new MyClass();
```

作为一名库设计者，一定要记住 package 和 import 关键字允许我们做的事情就是分割单个全局命名空间，保证我们不会遇到名字的冲突——无论有多少人使用因特网，也无论多少人用 Java 编写自己的类。

5.1.1 创建独一无二的包名

大家或许已注意到这样一个事实：由于一个包永远不会真的“封装”到单独一个文件里面，它可由多个.class 文件构成，所以局面可能稍微有些混乱。为避免这个问题，最合理的一种做法就是将某个特定包使用的所有.class 文件都置入单个目录里。也就是说，我们要利用操作系统的分级文件结构避免出现混乱局面。这正是 Java 所采取的方法。

它同时也解决了另两个问题：创建独一无二的包名以及找出那些可能深藏于目录结构某处的类。正如我们在第 2 章讲述的那样，为达到这个目的，需要将.class 文件的位置路径编码到 package 的名字里。但根据约定，编译器强迫 package 名的第一部分是类创建者的因特网域名。由于因特网域名肯定是独一无二的（由 InterNIC 保证——注释②，它控制着域名的分配），所以假如按这一约定行事，package 的名称就肯定不会重复，所以永远不会遇到名称冲突的问题。换句话说，除非将自己的域名转让给其他人，而且对方也按照相同的路径名编写 Java 代码，否则名字的冲突是永远不会出现的。当然，如果你没有自己的域名，那么必须创造一个非常生僻的包名（例如自己的英文姓名），以便尽最大可能创建一个独一无二的包名。如决定发行自己的 Java 代码，那么强烈推荐去申请自己的域名，它所需的费用是非常低廉的。

②：ftp://ftp.internic.net

这个技巧的另一部分是将 package 名解析成自己机器上的一个目录。这样一来，Java 程序运行并需要装载.class 文件的时候（这是动态进行的，在程序需要创建属于那个类的一个对象，或者首次访问那个类的一个 static 成员时），它就可以找到.class 文件驻留的那个目录。

Java 解释器的工作程序如下：首先，它找到环境变量 CLASSPATH（将 Java 或者具有 Java 解释能力的工具——如浏览器——安装到机器中时，通过操作系统进行设定）。CLASSPATH 包含了一个或多个目录，它们作为一种特殊的“根”使用，从这里展开对.class 文件的搜索。从那个根开始，解释器会寻找包名，并将每个点号（句点）替换成一个斜杠，从而生成从 CLASSPATH 根开始的一个路径名（所以 package foo.bar.baz 会变成 foo\bar\baz 或者 foo/bar/baz；具体是正斜杠还是反斜杠由操作系统决定）。随后将它们连接到一起，成为 CLASSPATH 内的各个条目（入口）。以后搜索.class 文件时，就可从这些地方开始查找与准备创建的类名对应的名字。此外，它也会搜索一些标准目录——这些目录与 Java 解释器驻留的地方有关。

为进一步理解这个问题，下面以我自己的域名为例，它是 bruceeckel.com。将其反转过来后，com.bruceeckel 就为我的类创建了独一无二的全局名称（com，edu，org，net 等扩展名以前在 Java 包中都是大写的，但自 Java 1.2 以来，这种情况已发生了变化。现在整个包名都是小写的）。由于决定创建一个名为 util 的库，我可以进一步地分割它，所以最后得到的包名如下：

```
package com.bruceeckel.util;
```

现在，可将这个包名作为下述两个文件的“命名空间”使用：

194 页中程序

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

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

创建自己的包时，要求 package 语句必须是文件中的第一个“非注释”代码。第二个文件表面看起来是类似的：

194-195 页程序

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

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

这两个文件都置于我自己系统的一个子目录中：

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

若通过它往回走，就会发现包名 com.bruceeckel.util，但路径的第一部分又是什么呢？这是由 CLASSPATH 环境变量决定的。在我的机器上，它是：

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

可以看出，CLASSPATH 里能包含大量备用的搜索路径。然而，使用 JAR 文件时要注意一个问题：必须将 JAR 文件的名字置于类路径里，而不仅仅是它所在的路径。所以对一个名为 grape.jar 的 JAR 文件来说，我们的类路径需要包括：

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

正确设置好类路径后，可将下面这个文件置于任何目录里（若在执行该程序时遇到麻烦，请参见第 3 章的 3.1.2 小节“赋值”）：

195 页程序

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

```
public class LibTest {
  public static void main(String[] args) {
    Vector v = new Vector();
    List l = new List();
  }
} ///:~
```

编译器遇到 import 语句后，它会搜索由 CLASSPATH 指定的目录，查找子目录 com\bruceeckel\util，然后查找名称适当的已编译文件（对于 Vector 是 Vector.class，对于 List 则是 List.class）。注意 Vector 和 List 内无论类还是需要的方法都必须设为 public。

1. 自动编译

为导入的类首次创建一个对象时（或者访问一个类的 static 成员时），编译器会在适当的目录里寻找同名的.class 文件（所以如果创建类 X 的一个对象，就应该是 X.class）。若只发现 X.class，它就是必须使用的那一个类。然而，如果它在相同的目录中还发现了一个 X.java，编译器就会比较两个文件的日期标记。如果 X.java 比 X.class 新，就会自动编译 X.java，生成一个最新的 X.class。

对于一个特定的类，或在与它同名的.java 文件中没有找到它，就会对那个类采取上述的处理。

2. 冲突

若通过*导入了两个库，而且它们包括相同的名字，这时会出现什么情况呢？例如，假定一个程序使用了下述导入语句：

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

由于 java.util.*也包含了一个 Vector 类，所以这会造成潜在的冲突。然而，只要冲突并不真的发生，那么就不会产生任何问题——这当然是最理想的情况，因为否则的话，就需要进行大量编程工作，防范那些可能可能永远也不会发生的冲突。

如现在试着生成一个 Vector，就肯定会发生冲突。如下所示：

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

它引用的到底是哪个 Vector 类呢？编译器对这个问题没有答案，读者也不可能知道。所以编译器会报告一个错误，强迫我们进行明确的说明。例如，假设我想使用标准的 Java Vector，那么必须象下面这样编程：

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

由于它（与 CLASSPATH 一起）完整指定了那个 Vector 的位置，所以不再需要 import java.util.*语句，除非还想使用来自 java.util 的其他东西。

5.1.2 自定义工具库

掌握前述的知识后，接下来就可以开始创建自己的工具库，以便减少或者完全消除重复的代码。例如，可为 System.out.println()创建一个别名，减少重复键入的代码量。它可以是名为 tools 的一个包（package）的一部分：

197-198 页程序

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

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

所有不同的数据类型现在都可以在一个新行输出（P.rintln()），或者不在一个新行输出（P.rint()）。

大家可能会猜想这个文件所在的目录必须从某个 CLASSPATH 位置开始，然后继续 com/bruceeckel/tools。编译完毕后，利用一个 import 语句，即可在自己系统的任何地方使用 P.class 文件。如下所示：

198 页下程序

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

public class ToolTest {
  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);
  }
} ///:~
```

所以从现在开始，无论什么时候只要做出了一个有用的新工具，就可将其加入 tools 目录（或者自己的个人 util 或 tools 目录）。

1. CLASSPATH 的陷阱

P.java 文件存在一个非常有趣的陷阱。特别是对于早期的 Java 实现方案来说，类路径的正确设定通常都是很困难的一项工作。编写这本书的时候，我引入了 P.java 文件，它最初看起来似乎工作很正常。但在某些情况下，却开始出现中断。在很长的时间里，我都确信这是 Java 或其他什么在实现时一个错误。但最后，我终于发现在一个地方引入了一个程序（即第 17 章要说明的

CodePackager.java），它使用了一个不同的类 P。由于它作为一个工具使用，所以有时候会进入类路径里；另一些时候则不会这样。但只要它进入类路径，那么假若执行的程序需要寻找 com.bruceeckel.tools 中的类，Java 首先发现的就是 CodePackager.java 中的 P。此时，编译器会报告一个特定的方法没有找到。这当然是非常令人头疼的，因为我们在前面的类 P 里明明看到了这个方法，而且根本没有更多的诊断报告可为我们提供一条线索，让我们知道找到的是一个完全不同的类（那甚至不是 public 的）。

乍一看来，这似乎是编译器的一个错误，但假若考察 import 语句，就会发现它只是说："在这里可能发现了 P"。然而，我们假定的是编译器搜索自己类路径的任何地方，所以一旦它发现一个 P，就会使用它；若在搜索过程中发现了"错误的"一个，它就会停止搜索。这与我们在前面表述的稍微有些区别，因为存在一些讨厌的类，它们都位于包内。而这里有一个不在包内的 P，但仍可在常规的类路径搜索过程中找到。

如果您遇到象这样的情况，请务必保证对于类路径的每个地方，每个名字都仅存在一个类。

5.1.3 利用导入改变行为

Java 已取消的一种特性是 C 的"条件编译"，它允许我们改变参数，获得不同的行为，同时不改变其他任何代码。Java 之所以抛弃了这一特性，可能是由于该特性经常在 C 里用于解决跨平台问题：代码的不同部分根据具体的平台进行编译，否则不能在特定的平台上运行。由于 Java 的设计思想是成为一种自动跨平台的语言，所以这种特性是没有必要的。

然而，条件编译还有另一些非常有价值的用途。一种很常见的用途就是调试代码。调试特性可在开发过程中使用，但在发行的产品中却无此功能。Alen Holub（www.holub.com）提出了利用包（package）来模仿条件编译的概念。根据这一概念，它创建了 C"断定机制"一个非常有用的 Java 版本。之所以叫作"断定机制"，是由于我们可以说"它应该为真"或者"它应该为假"。如果语句不同意你的断定，就可以发现相关的情况。这种工具在调试过程中是特别有用的。

可用下面这个类进行程序调试：

200 页程序

```
//: com:bruceeckel:tools:debug:Assert.java
// Assertion tool for debugging.
package com.bruceeckel.tools.debug;

public class Assert {
  private static void perr(String msg) {
    System.err.println(msg);
  }
  public final static void is_true(boolean exp) {
    if(!exp) perr("Assertion failed");
  }
  public final static void is_false(boolean exp){
    if(exp) perr("Assertion failed");
```

```
  }
  public final static void
  is_true(boolean exp, String msg) {
    if(!exp) perr("Assertion failed: " + msg);
  }
  public final static void
  is_false(boolean exp, String msg) {
    if(exp) perr("Assertion failed: " + msg);
  }
} ///:~
```

这个类只是简单地封装了布尔测试。如果失败，就显示出出错消息。在第 9 章，大家还会学习一个更高级的错误控制工具，名为“异常控制”。但在目前这种情况下，perr()方法已经可以很好地工作。

如果想使用这个类，可在自己的程序中加入下面这一行：

```
import com.bruceeckel.tools.debug.*;
```

如欲清除断定机制，以便自己能发行最终的代码，我们创建了第二个 Assert 类，但却是在一个不同的包里：

200-201 页程序

```
//: com:bruceeckel:tools:Assert.java
// Turning off the assertion output
// so you can ship the program.
package com.bruceeckel.tools;

public class Assert {
  public final static void is_true(boolean exp){}
  public final static void is_false(boolean exp){}
  public final static void
  is_true(boolean exp, String msg) {}
  public final static void
  is_false(boolean exp, String msg) {}
} ///:~
```

现在，假如将前一个 import 语句变成下面这个样子：

```
import com.bruceeckel.tools.*;
```

程序便不再显示出断言。下面是个例子：

201 页中程序

```
//: c05:TestAssert.java
// Demonstrating the assertion tool.
// Comment the following, and uncomment the
// subsequent line to change assertion behavior:
import com.bruceeckel.tools.debug.*;
```

```
// import com.bruceeckel.tools.*;

public class TestAssert {
  public static void main(String[] args) {
    Assert.is_true((2 + 2) == 5);
    Assert.is_false((1 + 1) == 2);
    Assert.is_true((2 + 2) == 5, "2 + 2 == 5");
    Assert.is_false((1 + 1) == 2, "1 +1 != 2");
  }
} ///:~
```

通过改变导入的 package，我们可将自己的代码从调试版本变成最终的发行版本。这种技术可应用于任何种类的条件代码。

5.1.4 包的停用

大家应注意这样一个问题：每次创建一个包后，都在为包取名时间接地指定了一个目录结构。这个包必须存在（驻留）于由它的名字规定的目录内。而且这个目录必须能从 CLASSPATH 开始搜索并发现。最开始的时候，package 关键字的运用可能会令人迷惑，因为除非坚持遵守根据目录路径指定包名的规则，否则就会在运行期获得大量莫名其妙的消息，指出找不到一个特定的类——即使那个类明明就在相同的目录中。若得到象这样的一条消息，请试着将 package 语句作为注释标记出去。如果这样做行得通，就可知道问题到底出在哪儿。

5.2 Java 访问指示符

针对类内每个成员的每个定义，Java 访问指示符 poublic，protected 以及 private 都置于它们的最前面——无论它们是一个数据成员，还是一个方法。每个访问指示符都只控制着对那个特定定义的访问。这与 C++存在着显著不同。在 C++中，访问指示符控制着它后面的所有定义，直到又一个访问指示符加入为止。

通过千丝万缕的联系，程序为所有东西都指定了某种形式的访问。在后面的小节里，大家要学习与各类访问有关的所有知识。首次从默认访问开始。

5.2.1 “友好的”

如果根本不指定访问指示符，就象本章之前的所有例子那样，这时会出现什么情况呢？默认的访问没有关键字，但它通常称为“友好”（Friendly）访问。这意味着当前包内的其他所有类都能访问“友好的”成员，但对包外的所有类来说，这些成员却是“私有”（Private）的，外界不得访问。由于一个编译单元（一个文件）只能从属于单个包，所以单个编译单元内的所有类相互间都是自动“友好”的。因此，我们也说友好元素拥有“包访问”权限。

友好访问允许我们将相关的类都组合到一个包里，使它们相互间方便地进行沟通。将类组合到一个包内以后（这样便允许友好成员的相互访问，亦即让它们“交朋友”），我们便“拥有”了那个包内的代码。只有我们已经拥有的代码才能

友好地访问自己拥有的其他代码。我们可认为友好访问使类在一个包内的组合显得有意义，或者说前者是后者的原因。在许多语言中，我们在文件内组织定义的方式往往显得有些牵强。但在 Java 中，却强制用一种颇有意义的形式进行组织。除此以外，我们有时可能想排除一些类，不想让它们访问当前包内定义的类。

对于任何关系，一个非常重要的问题是“谁能访问我们的‘私有’或 private 代码”。类控制着哪些代码能够访问自己的成员。没有任何秘诀可以“闯入”。另一个包内推荐可以声明一个新类，然后说：“嗨，我是 Bob 的朋友!”，并指望看到 Bob 的“protected”（受到保护的）、友好的以及“private”（私有）的成员。为获得对一个访问权限，唯一的方法就是：

(1) 使成员成为“public”（公共的）。这样所有人从任何地方都可以访问它。

(2) 变成一个“友好”成员，方法是舍弃所有访问指示符，并将其类置于相同的包内。这样一来，其他类就可以访问成员。

(3) 正如以后引入“继承”概念后大家会知道的那样，一个继承的类既可以访问一个 protected 成员，也可以访问一个 public 成员（但不可访问 private 成员）。只有在两个类位于相同的包内时，它才可以访问友好成员。但现在不必关心这方面的问题。

(4) 提供“访问器 / 变化器”方法（亦称为“获取 / 设置”方法），以便读取和修改值。这是 OOP 环境中最正规的一种方法，也是 Java Beans 的基础——具体情况会在第 13 章介绍。

5.2.2 public：接口访问

使用 public 关键字时，它意味着紧随在 public 后面的成员声明适用于所有人，特别是适用于使用库的客户程序员。假定我们定义了一个名为 dessert 的包，其中包含下述单元(若执行该程序时遇到困难，请参考第 3 章 3.1.2 小节“赋值”)：

203 页下程序

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

请记住，Cookie.java 必须驻留在名为 dessert 的一个子目录内，而这个子目录又必须位于由 CLASSPATH 指定的 C05 目录下面（C05 代表本书的第 5 章）。不要错误地以为 Java 无论如何都会将当前目录作为搜索的起点看待。如果不将一个“.”作为 CLASSPATH 的一部分使用，Java 就不会考虑当前目录。

现在，假若创建使用了 Cookie 的一个程序，如下所示：

204 页上程序

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

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

就可以创建一个 Cookie 对象，因为它的构建器是 public 的，而且类也是 public 的（公共类的概念稍后还会进行更详细的讲述）。然而，foo()成员不可在 Dinner.java 内访问，因为 foo()只有在 dessert 包内才是“友好”的。

1. 默认包

大家可能会惊讶地发现下面这些代码得以顺利编译——尽管它看起来似乎已违背了规则：

204 页下程序

```
//: c05:Cake.java
// Accesses a class in a
// separate compilation unit.

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

在位于相同目录的第二个文件里：

205 页上程序

```
//: c05:Pie.java
// The other class.
class Pie {
  void f() { System.out.println("Pie.f()");
 }
} ///:~
```

最初可能会把它们看作完全不相干的文件，然而 Cake 能创建一个 Pie 对象，

并能调用它的 f()方法！通常的想法会认为 Pie 和 f()是“友好的”，所以不适用于 Cake。它们确实是友好的——这部分结论非常正确。但它们之所以仍能在 Cake.java 中使用，是由于它们位于相同的目录中，而且没有明确的包名。Java 把象这样的文件看作那个目录“默认包”的一部分，所以它们对于目录内的其他文件来说是“友好”的。

5.2.3 private：不能接触！

private 关键字意味着除非那个特定的类，而且从那个类的方法里，否则没有人能访问那个成员。同一个包内的其他成员不能访问 private 成员，这使其显得似乎将类与我们自己都隔离起来。另一方面，也不能由几个合作的人创建一个包。所以 private 允许我们自由地改变那个成员，同时毋需关心它是否会影响同一个包内的另一个类。默认的“友好”包访问通常已经是一种适当的隐藏方法；请记住，对于包的用户来说，是不能访问一个“友好”成员的。这种效果往往能令人满意，因为默认访问是我们通常采用的方法。对于希望变成 public（公共）的成员，我们通常明确地指出，令其可由客户程序员自由调用。而且作为一个结果，最开始的时候通常会认为自己不必频繁使用 private 关键字，因为完全可以在不用它的前提下发布自己的代码（这与 C++是个鲜明的对比）。然而，随着学习的深入，大家就会发现 private 仍然有非常重要的用途，特别是在涉及多线程处理的时候（详情见第 14 章）。

下面是应用了 private 的一个例子：

205-206 页程序

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

class Sundae {
  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();
  }
} ///:~
```

这个例子向我们证明了使用 private 的方便：有时可能想控制对象的创建方式，并防止有人直接访问一个特定的构建器（或者所有构建器）。在上面的例子中，我们不可通过它的构建器创建一个 Sundae 对象；相反，必须调用 makeASundae()方法来实现（注释③）。

③：此时还会产生另一个影响：由于默认构建器是唯一获得定义的，而且它的属性是 private，所以可防止对这个类的继承（这是第 6 章要重点讲述的主题）。

若确定一个类只有一个“助手”方法，那么对于任何方法来说，都可以把它们设为 private，从而保证自己不会误在包内其他地方使用它，防止自己更改或删除方法。将一个方法的属性设为 private 后，可保证自己一直保持这一选项（然而，若一个句柄被设为 private，并不表明其他对象不能拥有指向同一个对象的 public 句柄。有关“别名”的问题将在第 12 章详述）。

5.2.4 protected：“友好的一种”

protected（受到保护的）访问指示符要求大家提前有所认识。首先应注意这样一个事实：为继续学习本书一直到继承那一章之前的内容，并不一定需要先理解本小节的内容。但为了保持内容的完整，这儿仍然要对此进行简要说明，并提供相关的例子。

protected 关键字为我们引入了一种名为“继承”的概念，它以现有的类为基础，并在其中加入新的成员，同时不会对现有的类产生影响——我们将这种现有的类称为“基础类”或者“基本类”（Base Class）。亦可改变那个类现有成员的行为。对于从一个现有类的继承，我们说自己的新类“扩展”（extends）了那个现有的类。如下所示：

```
class Foo extends Bar {
```

类定义剩余的部分看起来是完全相同的。

若新建一个包，并从另一个包内的某个类里继承，则唯一能够访问的成员就是原来那个包的 public 成员。当然，如果在相同的包里进行继承，那么继承获得的包能够访问所有“友好”的成员。有些时候，基础类的创建者喜欢提供一个特殊的成员，并允许访问衍生类。这正是 protected 的工作。若往回引用 5.2.2 小节“public：接口访问”的那个 Cookie.java 文件，则下面这个类就不能访问“友好”的成员：

207 页上程序

```
//: c05:ChocolateChip.java
// Can't access friendly member
// in another class.
import c05.dessert.*;

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

对于继承，值得注意的一件有趣的事情是倘若方法 foo()存在于类 Cookie 中，那么它也会存在于从 Cookie 继承的所有类中。但由于 foo()在外部的包里是“友好”的，所以我们不能使用它。当然，亦可将其变成 public。但这样一来，由于所有人都能自由访问它，所以可能并非我们所希望的局面。若象下面这样修改类 Cookie：

207 页下程序

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

那么仍然能在包 dessert 里“友好”地访问 foo()，但从 Cookie 继承的其他东西亦可自由地访问它。然而，它并非公共的（public）。

5.3 接口与实现

我们通常认为访问控制是“隐藏实施细节”的一种方式。将数据和方法封装到类内后，可生成一种数据类型，它具有自己的特征与行为。但由于两方面重要的原因，访问为那个数据类型加上了自己的边界。第一个原因是规定客户程序员哪些能够使用，哪些不能。我们可在结构里构建自己的内部机制，不用担心客户程序员将其当作接口的一部分，从而自由地使用或者“滥用”。

这个原因直接导致了第二个原因：我们需要将接口同实施细节分离开。若结构在一系列程序中使用，但用户除了将消息发给 public 接口之外，不能做其他任何事情，我们就可以改变不属于 public 的所有东西（如“友好的”、protected 以及 private），同时不要求用户对他们的代码作任何修改。

我们现在是在一个面向对象的编程环境中，其中的一个类（class）实际是指“一类对象”，就象我们说“鱼类”或“鸟类”那样。从属于这个类的所有对象都共享这些特征与行为。“类”是对属于这一类的所有对象的外观及行为进行的一种描述。

在一些早期 OOP 语言中，如 Simula-67，关键字 class 的作用是描述一种新的数据类型。同样的关键字在大多数面向对象的编程语言里都得到了应用。它其实是整个语言的焦点：需要新建数据类型的场合比那些用于容纳数据和方法的“容器”多得多。

在 Java 中，类是最基本的 OOP 概念。它是本书未采用粗体印刷的关键字之一——由于数量太多，所以会造成页面排版的严重混乱。

为清楚起见，可考虑用特殊的样式创建一个类：将 public 成员置于最开头，后面跟随 protected、友好以及 private 成员。这样做的好处是类的使用者可从上向下依次阅读，并首先看到对自己来说最重要的内容（即 public 成员，因为它们可从文件的外部访问），并在遇到非公共成员后停止阅读，后者已经属于内部实施细节的一部分了。然而，利用由 javadoc 提供支持的注释文档（已在第 2

章介绍），代码的可读性问题已在很大程度上得到了解决。

209 页上程序

```
public class X {
  public void pub1( ) { /* . . . */ }
  public void pub2( ) { /* . . . */ }
  public void pub3( ) { /* . . . */ }
  private void priv1( ) { /* . . . */ }
  private void priv2( ) { /* . . . */ }
  private void priv3( ) { /* . . . */ }
  private int i;
  // . . .
}
```

由于接口和实施细节仍然混合在一起，所以只是部分容易阅读。也就是说，仍然能够看到源码——实施的细节，因为它们需要保存在类里面。向一个类的消费者显示出接口实际是“类浏览器”的工作。这种工具能查找所有可用的类，总结出可对它们采取的全部操作（比如可以使用哪些成员等），并用一种清爽悦目的形式显示出来。到大家读到这本书的时候，所有优秀的 Java 开发工具都应推出了自己的浏览器。

5.4 类访问

在 Java 中，亦可用访问指示符判断出一个库内的哪些类可由那个库的用户使用。若想一个类能由客户程序员调用，可在类主体的起始花括号前面某处放置一个 public 关键字。它控制着客户程序员是否能够创建属于这个类的一个对象。

为控制一个类的访问，指示符必须在关键字 class 之前出现。所以我们能够使用：

```
public class Widget {
```

也就是说，假若我们的库名是 mylib，那么所有客户程序员都能访问 Widget——通过下述语句：

```
import mylib.Widget;
```

或者

```
import mylib.*;
```

然而，我们同时还要注意到一些额外的限制：

(1) 每个编译单元（文件）都只能有一个 public 类。每个编译单元有一个公共接口的概念是由那个公共类表达出来的。根据自己的需要，它可拥有任意多个提供支撑的“友好”类。但若在一个编译单元里使用了多个 public 类，编译器就会向我们提示一条出错消息。

(2) public 类的名字必须与包含了编译单元的那个文件的名字完全相符，甚至包括它的大小写形式。所以对于 Widget 来说，文件的名字必须是 Widget.java，而不应是 widget.java 或者 WIDGET.java。同样地，如果出现不符，就会报告一个编译期错误。

(3) 可能（但并常见）有一个编译单元根本没有任何公共类。此时，可按自己的意愿任意指定文件名。

如果已经获得了 mylib 内部的一个类，准备用它完成由 Widget 或者 mylib 内部的其他某些 public 类执行的任务，此时又会出现什么情况呢？我们不希望花费力气为客户程序员编制文档，并感觉以后某个时候也许会进行大手笔的修改，并将自己的类一起删掉，换成另一个不同的类。为获得这种灵活处理的能力，需要保证没有客户程序员能够依赖自己隐藏于 mylib 内部的特定实施细节。为达到这个目的，只需将 public 关键字从类中剔除即可，这样便把类变成了“友好的”（类仅能在包内使用）。

注意不可将类设成 private（那样会使除类之外的其他东西都不能访问它），也不能设成 protected（注释④）。因此，我们现在对于类的访问只有两个选择：“友好的”或者 public。若不愿其他任何人访问那个类，可将所有构建器设为 private。这样一来，在类的一个 static 成员内部，除自己之外的其他所有人都无法创建属于那个类的一个对象（注释⑤）。如下例所示：

210-211 页程序

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

④：实际上，Java 1.1 内部类既可以是“受到保护的”，也可以是“私有的”，但那属于特别情况。第 7 章会详细解释这个问题。

⑤：亦可通过从那个类继承来实现。

迄今为止，我们创建过的大多数方法都是要么返回 void，要么返回一个基本数据类型。所以对下述定义来说：

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

它最开始多少会使人有些迷惑。位于方法名（access）前的单词指出方法到底返回什么。在这之前，我们看到的都是 void，它意味着“什么也不返回”（void 在英语里是“虚无”的意思。但亦可返回指向一个对象的句柄，此时出现的就是这个情况。该方法返回一个句柄，它指向类 Soup 的一个对象。

Soup 类向我们展示出如何通过将所有构建器都设为 private，从而防止直接创建一个类。请记住，假若不明确地至少创建一个构建器，就会自动创建默认构建器（没有自变量）。若自己编写默认构建器，它就不会自动创建。把它变成 private 后，就没人能为那个类创建一个对象。但别人怎样使用这个类呢？上面的例子为我们揭示出了两个选择。第一个选择，我们可创建一个 static 方法，再通过它创建一个新的 Soup，然后返回指向它的一个句柄。如果想在返回之前对 Soup 进行一些额外的操作，或者想了解准备创建多少个 Soup 对象（可能是为了限制它们的个数），这种方案无疑是特别有用的。

第二个选择是采用“设计方案”（Design Pattern）技术，本书后面会对此进行详细介绍。通常方案叫作“独子”，因为它仅允许创建一个对象。类 Soup 的对象被创建成 Soup 的一个 static private 成员，所以有一个而且只能有一个。除非通过 public 方法 access()，否则根本无法访问它。

正如早先指出的那样，如果不针对类的访问设置一个访问指示符，那么它会自动默认为“友好的”。这意味着那个类的对象可由包内的其他类创建，但不能由包外创建。请记住，对于相同目录内的所有文件，如果没有明确地进行 package 声明，那么它们都默认为那个目录的默认包的一部分。然而，假若那个类一个 static 成员的属性是 public，那么客户程序员仍然能够访问那个 static 成员——即使它们不能创建属于那个类的一个对象。

5.5 总结

对于任何关系，最重要的一点都是规定好所有方面都必须遵守的界限或规

则。创建一个库时，相当于建立了同那个库的用户（即“客户程序员”）的一种关系——那些用户属于另外的程序员，可能用我们的库自行构建一个应用程序，或者用我们的库构建一个更大的库。

如果不制订规则，客户程序员就可以随心所欲地操作一个类的所有成员，无论我们本来愿不愿意其中的一些成员被直接操作。所有东西都在别人面前都暴露无遗。

本章讲述了如何构建类，从而制作出理想的库。首先，我们讲述如何将一组类封装到一个库里。其次，我们讲述类如何控制对自己成员的访问。

一般情况下，一个 C 程序项目会在 50K 到 100K 行代码之间的某个地方开始中断。这是由于 C 仅有一个“命名空间”，所以名字会开始互相抵触，从而造成额外的管理开销。而在 Java 中，package 关键字、包命名方案以及 import 关键字为我们提供对名字的完全控制，所以命名冲突的问题可以很轻易地得到避免。

有两方面的原因要求我们控制对成员的访问。第一个是防止用户接触那些他们不应碰的工具。对于数据类型的内部机制，那些工具是必需的。但它们并不属于用户接口的一部分，用户不必用它来解决自己的特定问题。所以将方法和字段变成“私有”（private）后，可极大方便用户。因为他们能轻易看出哪些对于自己来说是最重要的，以及哪些是自己需要忽略的。这样便简化了用户对一个类的理解。

进行访问控制的第二个、也是最重要的一个原因是：允许库设计者改变类的内部工作机制，同时不必担心它会对客户程序员产生什么影响。最开始的时候，可用一种方法构建一个类，后来发现需要重新构建代码，以便达到更快的速度。如接口和实施细节早已进行了明确的分隔与保护，就可以轻松地达到自己的目的，不要求用户改写他们的代码。

利用 Java 中的访问指示符，可有效控制类的创建者。那个类的用户可确切知道哪些是自己能够使用的，哪些则是可以忽略的。但更重要的一点是，它可确保没有任何用户能依赖一个类的基础实施机制的任何部分。作为一个类的创建者，我们可自由修改基础的实施细节，这一改变不会对客户程序员产生任何影响，因为他们不能访问类的那一部分。

有能力改变基础的实施细节后，除了能在以后改进自己的设置之外，也同时拥有了“犯错误”的自由。无论当初计划与设计时有多么仔细，仍然有可能出现一些失误。由于知道自己能相当安全地犯下这种错误，所以可以放心大胆地进行更多、更自由的试验。这对自己编程水平的提高是很有帮助的，使整个项目最终能更快、更好地完成。

一个类的公共接口是所有用户都能看见的，所以在进行分析与设计的时候，这是应尽量保证其准确性的最重要的一个部分。但也不必过于紧张，少许的误差仍然是允许的。若最初设计的接口存在少许问题，可考虑添加更多的方法，只要保证不删除客户程序员已在他们的代码里使用的东西。

5.6 练习

(1) 用 public、private、protected 以及“友好的”数据成员及方法成员创建一个类。创建属于这个类的一个对象，并观察在试图访问所有类成员时会获得哪种类型的编译器错误提示。注意同一个目录内的类属于“默认”包的一部分。

(2) 用 protected 数据创建一个类。在相同的文件里创建第二个类，用一个方法操纵第一个类里的 protected 数据。

(3) 新建一个目录，并编辑自己的 CLASSPATH，以便包括那个新目录。将 P.class 文件复制到自己的新目录，然后改变文件名、P 类以及方法名（亦可考虑添加额外的输出，观察它的运行过程）。在一个不同的目录里创建另一个程序，令其使用自己的新类。

(4) 在 c05 目录（假定在自己的 CLASSPATH 里）创建下述文件：

214 页程序

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

然后在 c05 之外的另一个目录里创建下述文件：

214-215 页程序

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

解释编译器为什么会产生一个错误。将 Foreign（外部）类作为 c05 包的一部分改变了什么东西吗？

Exercises

1. Write a program that creates an ArrayList object without explicitly importing java.util.*.

2. In the section labeled “package: the library unit,” turn the code fragments concerning mypackage into a compiling and running set of Java files.

In the section labeled “Collisions,” take the code fragments and turn them into a program, and verify that collisions do in fact occur.

Generalize the class P defined in this chapter by adding all the overloaded versions of rint( ) and rintln( ) necessary to handle all the different basic Java types.

Change the import statement in TestAssert.java to enable and disable the assertion mechanism.

Create a class with public, private, protected, and “friendly” data members and method members. Create an object of this class and see what kind of compiler messages you get when you try to access all the class members. Be aware that classes in the same directory are part of the “default” package.

Create a class with protected data. Create a second class in the same file with a method that manipulates the protected data in the first class.

Change the class Cookie as specified in the section labeled “protected: ‘sort of friendly.’” Verify that bite( ) is not public.

In the section titled “Class access” you’ll find code fragments describing mylib and Widget. Create this library, then create a Widget in a class that is not part of the mylib package.

Create a new directory and edit your CLASSPATH to include that new directory. Copy the P.class file (produced by compiling com.bruceeckel.tools.P.java) to your new directory and then change the names of the file, the P class inside and the method names. (You might also want to add additional output to watch how it works.) Create another program in a different directory that uses your new class.

Following the form of the example Lunch.java, create a class called ConnectionManager that manages a fixed array of Connection objects. The client programmer must not be able to explicitly create Connection objects, but can only get them via a static method in ConnectionManager. When the ConnectionManager runs out of objects, it returns a null reference. Test the classes in main( ).

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

Then create the following file in a directory other than c05:

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

} ///:~

Explain why the compiler generates an error. Would making the Foreign class part of the c05.local package change anything?

---

[33] There's nothing in Java that forces the use of an interpreter. There exist native-code Java compilers that generate a single executable file.

[34] There's another effect in this case: Since the default constructor is the only one defined, and it's private, it will prevent inheritance of this class. (A subject that will be introduced in Chapter 6.)

[35] However, people often refer to implementation hiding alone as encapsulation.

[36] Actually, an inner class can be private or protected, but that's a special case. These will be introduced in Chapter 7.

[37] You can also do it by inheriting (Chapter 6) from that class.

[Previous Chapter] [Short TOC] [Table of Contents] [Index] [Next Chapter]

英文版主页 | 中文版主页 | 详细目录 | 关于译者