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

8: 接口与内部类

接口(interface)和内部类(inner class)提供了一种更为复杂的组织和控制系统中对象的方法。

比方说，C++就没有这种机制，不过聪明的程序员还是能模拟出这种效果。Java 之所以会有这个特性，是因为设计人员认为它非常重要，语言应该直接用关键词提供支持。

你已经在第 7 章学过了 **abstract** 关键词，它能让你在类里创建一个或多个没有定义的方法——你给出了接口，但是留一部分没做定义，这部分要由它的继承类来定义。而 **interface** 关键词则创建了一种完全抽象的，根本不提供实现的类。你会看到，**interface** 不仅是一种抽象类的极端表现形式，它还是一种能让你将一个对象上传到多个基类的手段，因此它提供了类似 C++的“多重继承(multiple inheritance)”的功能。

初看起来，内部类像是一种简单的隐藏代码的机制：你只是把一个类放到另一个类里。但是，你将会看到，内部类可没这么简单——它还知道该怎样同宿主类(surrounding class)打交道——因此，即使很多人还不熟悉内部类，你还是能用它写出更为优雅清晰的代码。你得过一段时间才能把内部类熟练地运用到设计之中。

接口(interface)

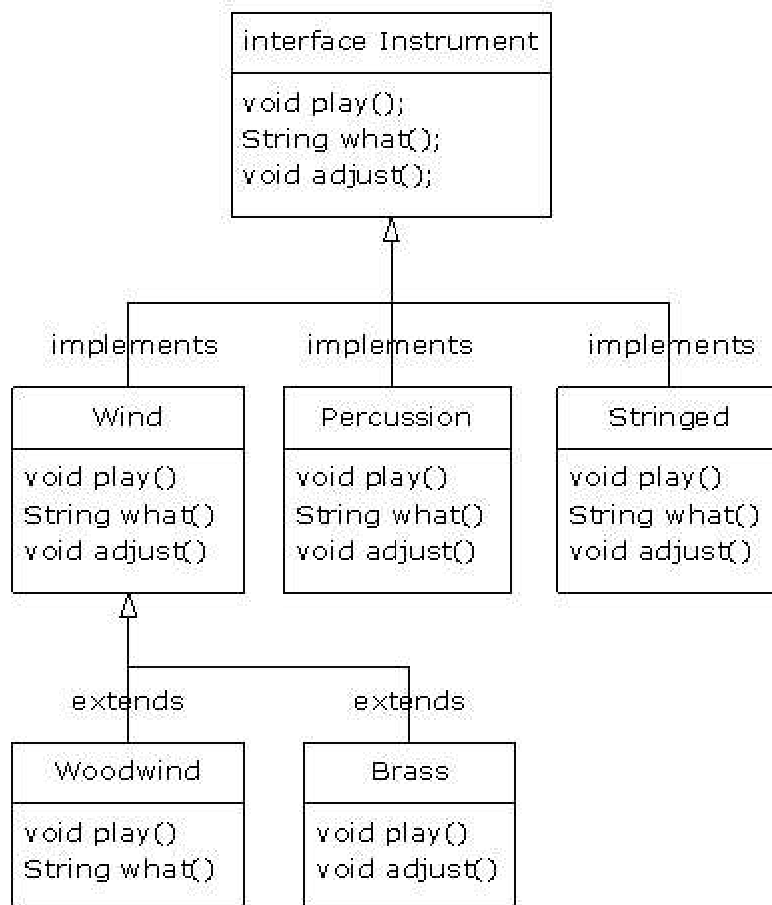
interface 关键词进一步强化了 **abstract** 的概念。你可以把它想像成“纯的”**abstract** 类。它能让开发人员定义类的形式：方法名，参数列表，返回值的类型，但是却没有方法的正文。**interface** 也可以包含数据成员，但是它天生就是 **static** 和 **final** 的。**interface** 只提供形式，不谈实现。

interface 的意思是“所有‘实现’这个接口的类都应该长这个样子。”因此任何程序，只要用到了这个 **interface** 就都知道它有那些方法可供调用了，仅此而已。因此，**interface** 会被用作定义类之间的“协议(protocol)”。(有些面向对象的语言真的用 *protocol* 关键词来作这件事。)

要想创建 **interface**，只要把 **class** 关键词换成 **interface** 就行了。跟类一样，你可以在 **interface** 关键词前面加上 **public**(只有保存在同名文件里的 **interface** 才可以加)，或者把它空着，留给它 package 权限，这样它就只能用于同一个 package 了。

要创建一个实现了某个(或者某组)**interface** 的类，就必须使用 **implements** 关键词。它的意思是，“**interface** 要告诉你‘类长什么

样子’，但是现在我要告诉你‘它是怎样『工作』的’。”除此之外，它同继承没什么两样。还是以乐器为例，下面的图演示了这种的关系：



可以从 **Woodwind** 和 **Brass** 看出，类一旦实现了某个 **interface**，它就变成了一个可以再继承下去的普通类了。

你可以把 **interface** 里的方法声明成 **public** 的，但是即便不讲，它们也是 **public** 的。所以当你 **implements** 一个 **interface** 的时候，你必须把这个 **interface** 的方法定义成 **public** 的。如果你不这么做，那它就会变成 package 权限的，这样经过继承，这些方法的访问权限就会受到限制，而这是 Java 的编译器所不允许的。

可以从修改后的 **Instrument** 例程中看到这一点。注意，编译器只允许你在 **interface** 里面声明方法。此外，虽然 **Instrument** 的方法都没有被声明成 **public** 的，但是它们自动都是 **public** 的：

```
//: c08:music5:Music5.java
// Interfaces.
package c08.music5;
import com.bruceeckel.simpletest.*;
import c07.music.Note;

interface Instrument {
```

```
  // Compile-time constant:
  int I = 5; // static & final
  // Cannot have method definitions:
  void play(Note n); // Automatically public
  String what();
  void adjust();
}

class Wind implements Instrument {
  public void play(Note n) {
    System.out.println("Wind.play() " + n);
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion implements Instrument {
  public void play(Note n) {
    System.out.println("Percussion.play() " + n);
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed implements Instrument {
  public void play(Note n) {
    System.out.println("Stringed.play() " + n);
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play(Note n) {
    System.out.println("Brass.play() " + n);
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  public void play(Note n) {
    System.out.println("Woodwind.play() " + n);
  }
  public String what() { return "Woodwind"; }
}

public class Music5 {
  private static Test monitor = new Test();
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
```


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

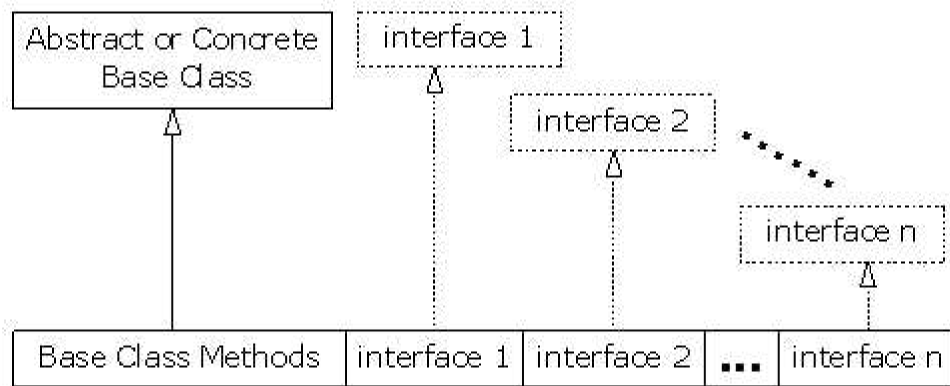
```
  public static void main(String[] args) {
    // Upcasting during addition to the array:
    Instrument[] orchestra = {
      new Wind(),
      new Percussion(),
      new Stringed(),
      new Brass(),
      new Woodwind()
    };
    tuneAll(orchestra);
    monitor.expect(new String[] {
      "Wind.play() Middle C",
      "Percussion.play() Middle C",
      "Stringed.play() Middle C",
      "Brass.play() Middle C",
      "Woodwind.play() Middle C"
    });
  }
} ///:~
```

其它代码的工作方式没变。不论是把它上传到一个叫 **Instrument** 的“普通”类，还是一个叫 **Instrument** 的 **abstract** 类，还是一个叫 **Instrument** 的 **interface**，它的工作方式都是一样的。实际上，你根本没法从 **tune( )**来判断，**Instrument** 到底是“普通”类，还是 **abstract** 类，或 **interface**。这就是它的本意：它让程序员自己去选择要在那个级别上控制对象的创建和使用。

Java 的“多重继承”

interface 不仅仅是一种“更纯”的 **abstract** 类。它还有更高一层的目的。由于 **interface** 不带任何“实现”——也就是说 **interface** 和内存无关——因此不会有谁去组绕 **interface** 之间的结合。这一点非常重要，因为有时你会遇到“**x** 既是 **a** 又是 **b**，而且还是 **c**”的情况。在 C++中，这种“将多个类的接口结合在一起”的行为被称作“多重继承 (*multiple inheritance*)”，但是由于每个类又都有它自己的实现，而这会带来很多“甩都甩不掉”的问题。Java 能让你作同样的事情，但是这时只有一个类可以有实现，因此当你合并 Java 接口的时候，就不会有这种问题了：

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



Java 并不强制你一定要去继承 **abstract** 还是“具体”的类(就是不带 **abstract** 方法的类)，但是你只能继承一个非 **interface** 的类。所有别的基类元素(base elements)都必须是 **interface**。你得把所有的接口名字都放在 **implements** 关键词后面，用逗号把它们分开。你可以根据需要，实现任意多个 **interface**；也可以将这个类上传至任何一个 **interface**。下面这段程序演示了如何将一个具体的类同几个 **interface** 合并起来，创建一个新的类：

```
//: c08:Adventure.java
// Multiple interfaces.

interface CanFight {
  void fight();
}

interface CanSwim {
  void swim();
}

interface CanFly {
  void fly();
}

class ActionCharacter {
  public void fight() {}
}

class Hero extends ActionCharacter
    implements CanFight, CanSwim, CanFly {
  public void swim() {}
  public void fly() {}
}

public class Adventure {
  public static void t(CanFight x) { x.fight(); }
  public static void u(CanSwim x) { x.swim(); }
  public static void v(CanFly x) { x.fly(); }
  public static void w(ActionCharacter x)
{ x.fight(); }
  public static void main(String[] args) {
    Hero h = new Hero();
    t(h); // Treat it as a CanFight
    u(h); // Treat it as a CanSwim
```

```
    v(h); // Treat it as a CanFly
    w(h); // Treat it as an ActionCharacter
  }
} ///:~
```

可以看到，**Hero** 合并了具体的 **ActionCharacter** 类，以及 **CanFight**，**CanSwim** 和 **CanFly** 接口。当你用这种方式合并实体类(concrete class)和接口的时候，必须将实体类放在前面，然后才是接口。(否则编译器就会报错。)

注意 **ActionCharacter** 类的 **fight( )**方法。它的特征与 **interface CanFight** 的 **fight( )**方法完全相同，但是 **Hero** 没有提供 **fight( )**的定义。**interface** 的规则是这样的，你可以继承它(马上就会说到)，但是继承下来的还是 **interface**。如果你想创建一个新类型的对象，那么这个类型就必须是类，而且还得提供所有定义。尽管 **Hero** 没有明确的提供 **fight( )**的定义，但是 **ActionCharacter** 提供了，所以 **Hero** 能自动获得这个方法，并且能创建对象了。

Adventure 类有四个拿接口和实体类作参数的方法。创建出来的 **Hero** 对象可以被传给其中任何一个方法，也就是说它被依次上传给了各个 **interface**。这个过程不需要程序员编写特别的代码，这一切要归功于接口在 Java 中的设计。

上述程序告诉我们接口的真正目的：能够上传到多个基本类型(base type)。然而，使用接口的第二个理由，实际上是和“把 **abstract** 类用做基类”完全相同的：就是要禁止客户程序员去创建这个类的对象，并且重申“这只是一个接口”。这就带来了一个问题：到底是用 **interface**，还是用 **abstract** 类？**interface** 既给了你 **abstract** 类的好处，又给了你 **interface** 的好处，因此只要基类的设计里面可以不包括方法和成员变量的定义，你就应该优先使用 **interface**。实际上，如果你知道这样东西可能会是基类的话，你就应该优先考虑把它做成 **interface**，只有在不得不定义方法或成员变量的情况下，你才能把它改成 **abstract** 类，或者根据需要改成实体类。

合并接口时的名字冲突

实现多个接口的时候可能会遇到一些小问题。在上述例程中，**CanFight** 和 **ActionCharacter** 都有一个一摸一样的 **void fight( )**方法。这里没有问题，因为它们使用的是同一个方法。但是如果不是呢？下面就是一个例子：

```
//: c08:InterfaceCollision.java

interface I1 { void f(); }
interface I2 { int f(int i); }
```


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

```
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
  public void f() {}
  public int f(int i) { return 1; } // overloaded
}

class C3 extends C implements I2 {
  public int f(int i) { return 1; } // overloaded
}

class C4 extends C implements I3 {
  // Identical, no problem:
  public int f() { return 1; }
}

// Methods differ only by return type:
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} ///:~
```

这个难题要归因于覆写、实现和重载的不期而遇，以及“不能仅通过返回值来辨别重载的方法”。如果把最后两行的注释去掉，就会出现如下的错误信息：

InterfaceCollision.java:23: f( ) in C cannot implement f( ) in I1; attempting to use incompatible return type
found : int
required: void
InterfaceCollision.java:24: interfaces I3 and I1 are incompatible; both define f( ), but with different return type

而且在要合并的接口里面放上同名方法，通常也会破坏程序的可读性。所以别这么做。

用继承扩展 interface

你可以用继承，往 **interface** 里面添加新的方法，也可以用继承把多个 **interface** 合并成一个新的 **interface**。在这两种情况下，你所得到的都只是一个新的 interface，就像下面这样：

```
//: c08:HorrorShow.java
// Extending an interface with inheritance.

interface Monster {
  void menace();
}

interface DangerousMonster extends Monster {
  void destroy();
}
```


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

```java
interface Lethal {
  void kill();
}

class DragonZilla implements DangerousMonster {
  public void menace() {}
  public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
  void drinkBlood();
}

class VeryBadVampire implements Vampire {
  public void menace() {}
  public void destroy() {}
  public void kill() {}
  public void drinkBlood() {}
}

public class HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  static void w(Lethal l) { l.kill(); }
  public static void main(String[] args) {
    DangerousMonster barney = new DragonZilla();
    u(barney);
    v(barney);
    Vampire vlad = new VeryBadVampire();
    u(vlad);
    v(vlad);
    w(vlad);
  }
} ///:~
```

DangerousMonster 只是对 **Monster** 做了一点扩展，然后生成一个新的 **interface**。**DragonZilla** 则实现了这个接口。

Vampire 的语法是“接口继承(inheriting interfaces)”所独有的。通常情况下， **extends** 只能用于类，但是由于一个 **interface** 可以由多个接口拼接而成，因此创建新的 **interface** 的时候可以用 **extends** 来表示其多个“基接口(base interfaces)”。正如你所看到的，**interface** 的名字要由逗号分隔。

常量的分组

由于 **interface** 的数据成员自动就是 **static** 和 **final** 的，因此 **interface** 是一种非常方便的，创建一组常量值的工具。这点同 C 和 C++的 **enum** 很相似。例如：

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

```
//: c08:Months.java
// Using interfaces to create groups of constants.
package c08;

public interface Months {
  int
    JANUARY = 1, FEBRUARY = 2, MARCH = 3,
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
    NOVEMBER = 11, DECEMBER = 12;
} ///:~
```

注意一下，Java 的编程风格是，用全部大写字母(用下划线分隔同一个标识符里的各个单词)来表示，用常量进行初始化的 **static final** 变量。

interface 的数据成员自动就是 **public** 的，因此就不必再注明了。

你可以像对待别的 package 那样，用 **import c08.***或者 **c08.Months** 把它引进来，这样就能在这个 package 的外面用 **Months.JANUARY** 之类的表达式来使用这些常量了。当然，你得到的是一个 **int**，因此它没有像 C++的 **enum** 那样的类型安全，但是这种(很常见的)手法要比直接在程序里面用数字要好得多。(这种方法通常被成为使用“神奇数字”，并且使得代码的维护变得非常困难。)

如果你确实需要额外的类型安全，可以像这样创建一个类：[33]

```
//: c08:Month.java
// A more robust enumeration system.
package c08;
import com.bruceeckel.simpletest.*;

public final class Month {
  private static Test monitor = new Test();
  private String name;
  private Month(String nm) { name = nm; }
  public String toString() { return name; }
  public static final Month
    JAN = new Month("January"),
    FEB = new Month("February"),
    MAR = new Month("March"),
    APR = new Month("April"),
    MAY = new Month("May"),
    JUN = new Month("June"),
    JUL = new Month("July"),
    AUG = new Month("August"),
    SEP = new Month("September"),
    OCT = new Month("October"),
    NOV = new Month("November"),
    DEC = new Month("December");
  public static final Month[] month =  {
    JAN, FEB, MAR, APR, MAY, JUN,
    JUL, AUG, SEP, OCT, NOV, DEC
  };
```


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

```
  public static final Month number(int ord) {
    return month[ord - 1];
  }
  public static void main(String[] args) {
    Month m = Month.JAN;
    System.out.println(m);
    m = Month.number(12);
    System.out.println(m);
    System.out.println(m == Month.DEC);
    System.out.println(m.equals(Month.DEC));
    System.out.println(Month.month[3]);
    monitor.expect(new String[] {
      "January",
      "December",
      "true",
      "true",
      "April"
    });
  }
} ///:~
```

Month 是一个带 **private** 构造函数的 **final** 类，因此谁也不能继承它，或者创建它的实例。所有实例都是由类自己创建的，并且都是 **final static** 的：**JAN**，**FEB**，**MAR**，等等。这些对象也被用于 **month** 数组。这是一个供你遍历的 **Month** 对象的数组。你可以传一个数字给 **number( )**方法，以选取相应月份的 **Month** 对象。从 **main( )** 可以看出，这么做是类型安全的；**m** 是一个 **Month** 的对象，所以它只能被赋予 **Month**。再前一个例子中，**Month.java** 所返回的是一个 **int** 的值，因此可能会给这个表示月份的 **int** 变量赋上其它的值，因此不是非常安全。

正如在 **main( )**的最后几行所看到的，这里你还能互换地使用**==** 和 **equals( )**。之所以能这么做，是因为 **Month** 的每个值只能有一个实例。到第 11 章，你还会学到一种新的定义类的方法，用这种类创建的对象能够进行相互比较。

此外 **java.util.Calendar** 里面有一个 month 的成员。

Apache 的 Jakarta Commons 项目下还包含了一些创建枚举类型的工具，它的功能与上述例程相似，但是用起来没那么麻烦。参见 *http://jakarta.apache.org/commons* 下面的“lang”，这个包是 **org.apache.commons.lang.enum**。这个项目里面还有很多别的，可能会非常有用的类库。

初始化接口中的数据成员

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

接口所定义的数据都自动是 **static** 和 **final** 的。它们不能是“空白的 final 数据成员”，但是可以用非常量的表达式来对它们进行初始化。例如：

```
//: c08:RandVals.java
// Initializing interface fields with
// non-constant initializers.
import java.util.*;

public interface RandVals {
  Random rand = new Random();
  int randomInt = rand.nextInt(10);
  long randomLong = rand.nextLong() * 10;
  float randomFloat = rand.nextLong() * 10;
  double randomDouble = rand.nextDouble() * 10;
} ///:~
```

由于数据成员都是 **static** 的，因此装载类的时候，也就是第一次访问这些数据成员的时候进行初始化。下面就是一个简单的测试：

```
//: c08:TestRandVals.java
import com.bruceeckel.simpletest.*;

public class TestRandVals {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    System.out.println(RandVals.randomInt);
    System.out.println(RandVals.randomLong);
    System.out.println(RandVals.randomFloat);
    System.out.println(RandVals.randomDouble);
    monitor.expect(new String[] {
      "%% -?\\d+",
      "%% -?\\d+",
      "%% -?\\d\\.\\d+E?-?\\d+",
      "%% -?\\d\\.\\d+E?-?\\d+"
    });
  }
} ///:~
```

当然，这些数据都不算是接口的组成部分，相反它们保存在这个接口的静态存储区内。

接口的嵌套

接口既可以嵌套在类里，也可以嵌套在接口里面。[34]这一点揭示了许多非常有趣的特点：

```
//: c08:nesting:NestingInterfaces.java
package c08.nesting;
```


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

```
class A {
  interface B {
    void f();
  }
  public class BImp implements B {
    public void f() {}
  }
  private class BImp2 implements B {
    public void f() {}
  }
  public interface C {
    void f();
  }
  class CImp implements C {
    public void f() {}
  }
  private class CImp2 implements C {
    public void f() {}
  }
  private interface D {
    void f();
  }
  private class DImp implements D {
    public void f() {}
  }
  public class DImp2 implements D {
    public void f() {}
  }
  public D getD() { return new DImp2(); }
  private D dRef;
  public void receiveD(D d) {
    dRef = d;
    dRef.f();
  }
}

interface E {
  interface G {
    void f();
  }
  // Redundant "public":
  public interface H {
    void f();
  }
  void g();
  // Cannot be private within an interface:
  //! private interface I {}
}

public class NestingInterfaces {
  public class BImp implements A.B {
    public void f() {}
  }
  class CImp implements A.C {
    public void f() {}
  }
  // Cannot implement a private interface except
  // within that interface's defining class:
  //! class DImp implements A.D {
  //!  public void f() {}
```


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

```
  //! }
  class EImp implements E {
    public void g() {}
  }
  class EGImp implements E.G {
    public void f() {}
  }
  class EImp2 implements E {
    public void g() {}
    class EG implements E.G {
      public void f() {}
    }
  }
  public static void main(String[] args) {
    A a = new A();
    // Can't access A.D:
    //! A.D ad = a.getD();
    // Doesn't return anything but A.D:
    //! A.DImp2 di2 = a.getD();
    // Cannot access a member of the interface:
    //! a.getD().f();
    // Only another A can do anything with getD():
    A a2 = new A();
    a2.receiveD(a.getD());
  }
} ///:~
```

在类里嵌套接口的语法还是比较明显的。跟没被嵌套在类里的接口一样，它也可以是 **public** 或 package 访问权限的。而且你还可以看到，**public** 和 package 权限的接口都被实现成 **public**，package 权限，甚至 **private** 的“嵌套类(nested classes)”。

正如 **A.D** 所演示的，作为一种新的技巧，接口也可以是 **private** 的(这种语法既可用于“嵌套接口『nested interfaces』”，也可用于“嵌套类『nested classes』”)。那么 **private** 的嵌套接口又有什么好处呢？可能你会认为，它只能被实现成 **private** 的内部类，就像 **DImp** 那样，但是 **A.DImp2** 告诉我们，它也能被实现成 **public** 的类。但是 **A.DImp2** 只能被当作它自己的类型来用。你不能透露任何“它实现了一个 **private** 接口”的信息，因此实现 **private** 接口就成了一种“强制你去定义那个接口的方法，但是又不让你添加任何类型信息(也就是说不允许上传)”的手段了。

getD( )方法揭示了 **private** 接口的更深层次的困境：它是一个 **public** 方法，但却会返回一个 **private** 接口的 reference。这个返回值又该怎么用呢？可以看到，**main( )**作了好几次尝试，试图利用这个返回值，但是都失败了。唯一行得通的办法就是，把这个返回值交给一个有权用它的对象——这就是通过 **A** 的 **receivedD( )**方法。

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

接口 **E** 揭示了，接口与接口也可以相互嵌套。不过接口的规则仍然有效——特别是接口的所有元素都必须是 **public** 的这条，因此嵌套在接口中的接口也都自动是 **public** 的，它们不能是 **private** 的。

NestingInterfaces 演示了各种实现嵌套接口的办法。特别要注意的是，实现接口的时候，不一定要实现嵌套在里面的接口。同样 **private** 接口只能在定义它的类里实现。

初看起来，这些特性好像只是为“语法的一致性”服务的，但是根据我的经验，一旦你理解了一种特性，自然会找到运用它的地方。

内部类

在一个类里定义另一个类是完全可以的。这被称为“内部类(*inner class*)”。内部类是一种非常有价值的特性，它能让你在逻辑上将相互从属的类组织起来，并且在类的内部控制访问权限。但是切记，内部类和合成是截然不同的，这一点非常重要。

虽然你正在学习内部类，但却不清楚为什么要有它。等到这一节的末尾，等我们讲完内部类的语法和语义之后，你就会看到一些“能告诉你内部类有什么好处”的例子了。

创建内部类的方法同你想像的完全相同——将类的定义放到它的宿主类(surrounding class)：

```
//: c08:Parcel1.java
// Creating inner classes.

public class Parcel1 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) {
      label = whereTo;
    }
    String readLabel() { return label; }
  }
  // Using inner classes looks just like
  // using any other class, within Parcel1:
  public void ship(String dest) {
    Contents c = new Contents();
    Destination d = new Destination(dest);
    System.out.println(d.readLabel());
  }
  public static void main(String[] args) {
    Parcel1 p = new Parcel1();
    p.ship("Tanzania");
  }
```


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

```
} ///:~
```

仅就 **ship( )**而言，内部类的用法同其它类没什么两样。实际上唯一的区别就是，类的名字被嵌套在 **Parcel1** 里面了。不过，过一会你就会看到，这并不是唯一的区别。

比较常见的，还是让宿主类提供一个会返回内部类的 reference 的方法，就像这样：

```
//: c08:Parcel2.java
// Returning a reference to an inner class.

public class Parcel2 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) {
      label = whereTo;
    }
    String readLabel() { return label; }
  }
  public Destination to(String s) {
    return new Destination(s);
  }
  public Contents cont() {
    return new Contents();
  }
  public void ship(String dest) {
    Contents c = cont();
    Destination d = to(dest);
    System.out.println(d.readLabel());
  }
  public static void main(String[] args) {
    Parcel2 p = new Parcel2();
    p.ship("Tanzania");
    Parcel2 q = new Parcel2();
    // Defining references to inner classes:
    Parcel2.Contents c = q.cont();
    Parcel2.Destination d = q.to("Borneo");
  }
} ///:~
```

除非是在“宿主类(outer class)”的非 **static** 方法里面，否则无论你在那里创建内部类的对象，都必须用 *OuterClassName.InnerClassName* 的形式来表示这个对象的类型，就像 **main( )**里面那样。

内部类与上传

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

到目前为止，内部类还没有表现出什么非常惊人的特质。毕竟，如果你所追求的只是隐藏机制，那么 Java 已经有了——只要赋予类 package 权限(只能在 package 内部访问)就行了，何必要把它做成内部类。

但是，当你将它上传到基类，特别是 **interface** 的时候，就会发现，内部类还是有它自己的特性的。(实际上，将对象上传给它所实现的接口与将它上传给基类是完全相同。)这样，任何人都不能看到或者访问到内部类了——也就是 **interface** 的实现了，于是“隐藏实现”就变得轻而易举了。你所得到的，只是一个基类或 **interface** 的 reference。

首先，要用单独的文件来定义公用接口，这样它们才能“全程使用”：

```
//: c08:Destination.java
public interface Destination {
  String readLabel();
} ///:~
```

```
//: c08:Contents.java
public interface Contents {
  int value();
} ///:~
```

现在客户程序员能用 **Contents** 和 **Destination** 接口了。(要记住，**interface** 就表示它成员自动就是 **public** 的。)

当你拿到基类或 **interface** 的 reference 的时候，有可能你会没办法找出它的具体类型，就像下面所演示的：

```
//: c08:TestParcel.java
// Returning a reference to an inner class.

class Parcel3 {
  private class PContents implements Contents {
    private int i = 11;
    public int value() { return i; }
  }
  protected class PDestination implements
Destination {
    private String label;
    private PDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
  }
  public Destination dest(String s) {
    return new PDestination(s);
  }
  public Contents cont() {
    return new PContents();
```


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

```
  }
}

public class TestParcel {
  public static void main(String[] args) {
    Parcel3 p = new Parcel3();
    Contents c = p.cont();
    Destination d = p.dest("Tanzania");
    // Illegal -- can't access private class:
    //! Parcel3.PContents pc = p.new PContents();
  }
} ///:~
```

在这个例子里，**main( )**必须在另一个类里，这样才能让人看到内部类 **PContents** 在私密性方面的效果。

Parcel3 加了点新东西：内部类 **PContent** 是 **private** 的，所以除了 **Parcel3**，谁都不能访问它。**PDestination** 是 **protected**，因此除了 **Parcel3**，同属这个 package 的类(因为 **protected** 也会给予包权限)，以及 **Parcel3** 的继承类，谁都不能访问 **PDestination**。这就是说，客户程序员对这些成员的了解和访问权限是有限制的。实际上，你甚至不能将对象下传给 **private** 的内部类(或者是 **protected** 的内部类，除非你继承了这个类)，因为，正如 **class TestParcel** 所演示的，你根本就不能用这个名字。由此，“**private** 的内部类”为类的设计者们提供了一种“能彻底杜绝『用具体类型来编程所引起的依赖性问题(type-coding dependencies)』，并且完全将实现细节隐藏起来”的方法。此外，从客户程序员的角度来看，扩展 **interface** 也是毫无意义的，因为他根本没法访问 **public interface** 以外的方法。这也给了 Java 编译器一个优化代码的机会。

普通类(非内部类)是不能被定义成 **private** 或 **protected** 的；它们只可能是 **public** 或 **package** 权限的。

在方法和作用域里的内部类

迄今为止，你已经看到了内部类的主要用法。通常，你所读写的，涉及到到内部类的程序，都是些简单易懂的“很平常的”内部类。但是内部类的设计是相当完善的，还有很多别的，不太为人所知的用法可供选择；内部类可以被创建在方法内部，甚至是任意一个作用域里。这么做有两个理由：

1. 就像前面所说的，你在实现某个接口，这样你才能创建并且返回这个接口的 reference。
2. 你正在处理一个复杂的问题，需要创建一个类，但是又不想让大家都知道还有这么一个类。

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

在接下来的例程中，我们会这样修改前面所讲的程序：

1. 在方法的内部定义一个类
2. 在方法的某个作用域里定义一个类
3. 一个实现了某个接口的匿名类
4. 一个继承了“某个有着非默认构造函数”的类的匿名类
5. 一个进行数据成员初始化的匿名类
6. 一个通过实例初始化(匿名内部类不能有构造函数)来进行构建的匿名类

尽管 **Wrapping** 是一个有着具体实现的普通类，但同时也是其继承类的共用“接口”：

```
//: c08:Wrapping.java
public class Wrapping {
  private int i;
  public Wrapping(int x) { i = x; }
  public int value() { return i; }
} ///:~
```

你会注意到，**Wrapping** 有一个带参数的构造函数，这是为了让事情变得更有趣一点。

第一个例子演示了，在方法的作用域里(而不是另一个类的作用域里)创建一个完整的类。这被称作“本地内部类(*local inner class*)”：

```
//: c08:Parcel4.java
// Nesting a class within a method.

public class Parcel4 {
  public Destination dest(String s) {
    class PDestination implements Destination {
      private String label;
      private PDestination(String whereTo) {
        label = whereTo;
      }
      public String readLabel() { return label; }
    }
    return new PDestination(s);
  }
  public static void main(String[] args) {
    Parcel4 p = new Parcel4();
    Destination d = p.dest("Tanzania");
  }
} ///:~
```

PDestination 类不是 **Parcel4** 的组成部分，相反它是 **dest( )**的组成部分。(此外，还有一点值得注意，在同一个子目录里，每个类都可以有一个 **PDestination** 内部类，这么作不会有名字冲突的问题。)因此，

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

出了 **dest( )**，谁也不能访问 **PDestination**。注意，上传发生在返回语句——**dest( )**只是送出一个 **Destination** 的 reference，它是 **PDestination** 的基类。当然，把 **PDestination** 被放在 **dest( )**里面，并不意味着 **dest( )**所返回的 **PDestination** 就不是一个有效的对象。

下面的例子演示了，怎样在任意一个作用域里嵌套内部类：

```
//: c08:Parcel5.java
// Nesting a class within a scope.

public class Parcel5 {
  private void internalTracking(boolean b) {
    if(b) {
      class TrackingSlip {
        private String id;
        TrackingSlip(String s) {
          id = s;
        }
        String getSlip() { return id; }
      }
      TrackingSlip ts = new TrackingSlip("slip");
      String s = ts.getSlip();
    }
    // Can't use it here! Out of scope:
    //! TrackingSlip ts = new TrackingSlip("x");
  }
  public void track() { internalTracking(true); }
  public static void main(String[] args) {
    Parcel5 p = new Parcel5();
    p.track();
  }
} ///:~
```

TrackingSlip 被嵌套在 **if** 语句里面。这并不意味着这个类的创建是有条件的——它会同别的东西一起编译。但是，这个类的访问范围仅限于定义它的那个作用域。除此之外，它同普通的类没什么两样。

匿名内部类

下面这段程序看上去有点奇怪：

```
//: c08:Parcel6.java
// A method that returns an anonymous inner class.

public class Parcel6 {
  public Contents cont() {
    return new Contents() {
      private int i = 11;
      public int value() { return i; }
    }; // Semicolon required in this case
  }
```


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

```
  public static void main(String[] args) {
    Parcel6 p = new Parcel6();
    Contents c = p.cont();
  }
} ///:~
```

cont( )方法将返回值的创建与“表示这个返回值的”类的定义，结合在了一起。此外这个类是匿名的；它没有名字。更糟糕的是，看上去你创建的是一个 **Contents** 对象：

```
return new Contents()
```

但是，等你看到分号的时候，你就会说，“慢！我是不是漏过了类的定义”：

```
return new Contents() {
  private int i = 11;
  public int value() { return i; }
};
```

这种奇怪的语法所要表达的意思是：“创建一个继承 **Contents** 的匿名类的对象”。**new** 语句所返回的 reference 会自动的上传到 **Contents**。实际上这段匿名内部类是如下代码的简化形式：

```
class MyContents implements Contents {
  private int i = 11;
  public int value() { return i; }
}
return new MyContents();
```

这个匿名内部类是通过默认构造函数来创建 **Contents** 的。下面这段程序演示了如果“基类所需的是一个带参数的构造函数”的话，那又该怎么做：

```
//: c08:Parcel7.java
// An anonymous inner class that calls
// the base-class constructor.

public class Parcel7 {
  public Wrapping wrap(int x) {
    // Base constructor call:
    return new Wrapping(x) { // Pass constructor
argument.
      public int value() {
```

```
        return super.value() * 47;
      }
    }; // Semicolon required
  }
  public static void main(String[] args) {
    Parcel7 p = new Parcel7();
    Wrapping w = p.wrap(10);
  }
} ///:~
```

这就是，直接将合适的参数传给基类的构造函数，就像这里，将 **x** 传给 **new Wrapping(x)**。

匿名内部类最后的分号并不表示类的正文的结束(这点不像 C++)。相反，它的意思是，这个包含匿名类的表达式结束了。因此，这个分号的作用同它在其它地方的作用是完全相同。

你也可以在定义匿名类的数据成员的时候进行初始化：

```
//: c08:Parcel8.java
// An anonymous inner class that performs
// initialization. A briefer version of Parcel4.java.

public class Parcel8 {
  // Argument must be final to use inside
  // anonymous inner class:
  public Destination dest(final String dest) {
    return new Destination() {
      private String label = dest;
      public String readLabel() { return label; }
    };
  }
  public static void main(String[] args) {
    Parcel8 p = new Parcel8();
    Destination d = p.dest("Tanzania");
  }
} ///:~
```

如果你在定义匿名内部类的时候，还要用到外面的对象，那编译就会要求你把这个参数的 reference 声明成 **final** 的，就像 **dest( )**的参数那样。如果你忘了，编译的时候就会报错。

如果你只想对数据成员进行赋值，那么这种做法还是很不错的。但是如果你想进行一些“类似构造函数所进行的”操作，那又该怎么办呢？你不能在匿名内部类里创建构造函数(因为它根本就没名字！)，但是有了“实例初始化(*instance initialization*)”，你就能在事实上创建一个匿名内部类的构造函数，就像这样：

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

```java
//: c08:AnonymousConstructor.java
// Creating a constructor for an anonymous inner
class.
import com.bruceeckel.simpletest.*;

abstract class Base {
  public Base(int i) {
    System.out.println("Base constructor, i = " + i);
  }
  public abstract void f();
}

public class AnonymousConstructor {
  private static Test monitor = new Test();
  public static Base getBase(int i) {
    return new Base(i) {
      {
        System.out.println("Inside instance
initializer");
      }
      public void f() {
        System.out.println("In anonymous f()");
      }
    };
  }
  public static void main(String[] args) {
    Base base = getBase(47);
    base.f();
    monitor.expect(new String[] {
      "Base constructor, i = 47",
      "Inside instance initializer",
      "In anonymous f()"
    });
  }
} ///:~
```

在这种情况下，变量 **i** 并一定是 **final** 的。**i** 会被传给匿名类的基类的构造函数，匿名类是绝不会直接使用它的。

下面用实例初始化来重讲一遍“parcel”这个话题。注意 **dest( )**的参数必须是 **final** 的，因为匿名类要用到它们。

```java
//: c08:Parcel9.java
// Using "instance initialization" to perform
// construction on an anonymous inner class.
import com.bruceeckel.simpletest.*;

public class Parcel9 {
  private static Test monitor = new Test();
  public Destination
  dest(final String dest, final float price) {
    return new Destination() {
      private int cost;
      // Instance initialization for each object:
      {
        cost = Math.round(price);
```


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

```
        if(cost > 100)
          System.out.println("Over budget!");
      }
      private String label = dest;
      public String readLabel() { return label; }
    };
  }
  public static void main(String[] args) {
    Parcel9 p = new Parcel9();
    Destination d = p.dest("Tanzania", 101.395F);
    monitor.expect(new String[] {
      "Over budget!"
    });
  }
} ///:~
```

在“实例初始化”部分，你可以看到一些不是以“对数据成员进行初始化的代码”的身份得到执行的代码(也就是 **if** 语句)。所以，实际上实例初始化过程就是匿名内部类的构造函数。当然，它的功能是有限的；你不能重载实例初始化，因此你只能有一个构造函数。

与宿主类的关系

讲到这里，内部类还只是一种隐藏名字和组织代码的方式，虽说这也是有用的，但还没到很吸引人的地步。但是，内部类还有一种用法。如果你创建了一个内部类，那么这个内部类的的对象，就与创建它的“宿主类的对象(enclosing object)”产生了某种关系，这样它就能访问宿主类对象的成员了——不需要任何特别的授权。此外，内部类还能访问宿主类的所有元素。[35]下面的例程演示了这一点：

```
//: c08:Sequence.java
// Holds a sequence of Objects.
import com.bruceeckel.simpletest.*;

interface Selector {
  boolean end();
  Object current();
  void next();
}

public class Sequence {
  private static Test monitor = new Test();
  private Object[] objects;
  private int next = 0;
  public Sequence(int size) { objects = new
Object[size]; }
  public void add(Object x) {
    if(next < objects.length)
      objects[next++] = x;
  }
  private class SSelector implements Selector {
    private int i = 0;
```

```
    public boolean end() { return i ==
objects.length; }
    public Object current() { return objects[i]; }
    public void next() { if(i < objects.length)
i++; }
  }
  public Selector getSelector() { return new
SSelector(); }
  public static void main(String[] args) {
    Sequence sequence = new Sequence(10);
    for(int i = 0; i < 10; i++)
      sequence.add(Integer.toString(i));
    Selector selector = sequence.getSelector();
    while(!selector.end()) {
      System.out.println(selector.current());
      selector.next();
    }
    monitor.expect(new String[] {
      "0",
      "1",
      "2",
      "3",
      "4",
      "5",
      "6",
      "7",
      "8",
      "9"
    });
  }
} ///:~
```

Sequence 只是一个带内部类的，固定长度的 **Object** 数组。你可以用 **add( )**方法往这个序列的末尾添加新的 **Object**(如果后面还有空的话)。此外你还可以利用 **Selector** 接口来提取 **Sequence** 里面的对象，这个接口包括，帮你判断是不是到了数组末尾的 **end( )**方法，帮你查找当前 **Object** 的 **current( )**方法，以及移到 **Sequence** 中的下一个 **Object** 的 **next( )**方法。由于 **Selector** 是一个 **interface**，因此每个类都可以用它自己的方法来实现这个 **interface**，而方法也可以拿这个 **interface** 作参数以创建“泛型程序(generic code)”。

这里的 **SSelector** 是一个提供了 **Selector** 功能的 **private** 类。你可以看到，**main( )**先创建了一个 **Sequence**，然后往里面加了一些 **String**。接着，再调用 **getSelector( )**得到了一个 **Selector**，并且用它来查询和选取 **Sequence** 的对象。

初看起来，**SSelector** 同别的内部类没什么两样。但是仔细看，你就会发现，它的方法——**end( )**，**current( )**，以及 **next( )**——所处理的 **object** 并不是 **SSelector** 的，相反它们都是其宿主类的 **private** 成员。但是内部类可以访问宿主类的方法和数据成员，就像它自己拥有的一样。从上面的例程来看，这种用法相当方便。

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

因此内部类会自动获得访问其宿主类成员的权限。那么它又是怎么做到这一点的呢？内部类里肯定会有一个指向“要负责创建它的”宿主类对象的 reference。这样，当你引用宿主类的成员的时候，就会使用那个(隐蔽的)reference 来选取成员。值得庆幸的是，编译器会为你打理这一切，但是你还是应该知道，内部类对象的创建是与宿主类对象有关的。创建内部类对象的前提就是，要获得宿主类对象的 reference，如果编译器得不到这个 reference，它就报错。绝大多数情况下，这个过程无须程序员的干预。

嵌套类

如果你不需要这种“内部类对象和宿主类对象之间的”联系，那么你可以把内部类定义成 **static** 的。这通常被称作“嵌套类(*nested class*)”。[36]要想理解 **static** 用于内部类时的意义，你就必须记住，普通的内部类对象都默认保存“它的宿主类对象，也就是创建它的那个对象的”reference。但是当你声明内部类是 **static** 的时候，情况就不是这样了。嵌套类的意思是：

1. 无须宿主类的对象就能创建嵌套类的对象。
2. 不能在嵌套类的对象里面访问非 **static** 的宿主类对象。

此外，嵌套类同普通的内部类还有一点不同。普通内部类的成员数据和方法只能到类的外围这一层，因此普通的内部类里不能有 **static** 数据，**static** 数据成员或嵌套类。但是，这些东西嵌套类里都可以有：

```
//: c08:Parcel10.java
// Nested classes (static inner classes).

public class Parcel10 {
  private static class ParcelContents implements
Contents {
    private int i = 11;
    public int value() { return i; }
  }
  protected static class ParcelDestination
  implements Destination {
    private String label;
    private ParcelDestination(String whereTo) {
      label = whereTo;
    }
    public String readLabel() { return label; }
    // Nested classes can contain other static
elements:
    public static void f() {}
    static int x = 10;
    static class AnotherLevel {
      public static void f() {}
      static int x = 10;
    }
  }
  public static Destination dest(String s) {
    return new ParcelDestination(s);
```


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

```
  }
  public static Contents cont() {
    return new ParcelContents();
  }
  public static void main(String[] args) {
    Contents c = cont();
    Destination d = dest("Tanzania");
  }
} ///:~
```

在 **main( )**里面，**Parcel10** 的对象是没什么用的；相反，你得使用普通的，选择 **static** 成员的语句来调用会返回 **Contents** 和 **Destination** 的 reference 的方法。

正如你马上将会看到得，普通(非 **static**)的内部类需要使用特殊的 **this** reference 来与宿主类对象保持联系。而嵌套类不需要这个 **this** reference，这就使得它与 **static** 方法有些相似了。

通常情况下， **interface** 里面是不能有任何代码的，但嵌套类却可以是 **interface** 的一部分。由于类是 **static** 的，因此这并不违反 **interface** 的规则——嵌套类只在接口的名字空间里：

```
//: c08:IInterface.java
// Nested classes inside interfaces.

public interface IInterface {
  static class Inner {
    int i, j, k;
    public Inner() {}
    void f() {}
  }
} ///:~
```

在本书的前面部分，我曾经建议每个类里都带一个供测试之用的 **main( )**。这种做法有一个缺点，就是编译之后，你得带着额外代码到处跑。如果这真的是一个问题，那么你可以嵌套类来保存测试代码：

```
//: c08:TestBed.java
// Putting test code in a nested class.

public class TestBed {
  public TestBed() {}
  public void f() { System.out.println("f()"); }
  public static class Tester {
    public static void main(String[] args) {
      TestBed t = new TestBed();
      t.f();
    }
  }
```


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

```
} ///:~
```

它会单独生成一个名为 **TestBed$Tester** 的 **.class** 文件(如果要运行这个程序，你可以打 **java TestBed$Tester**)。你可以用这个类来作测试，但是发布的时候可以不把它包括进去；只要在打包之前把 **TestBed$Tester.class** 删掉就行了。

引用宿主类的对象

如果你需要获取指向宿主类对象的 reference，那么可以在宿主类名字的后面加一个点再加 **this** 来表示宿主类。比如类 **Sequence.SSelctor** 里面的任何方法，都可以用 **Sequence.this** 来获取它所保存的宿主类 **Sequence** 的 reference。它会自动返回正确的类型。(这个类型是已知的，编译的时候就会作检查，因此不会给程序的运行造成额外的负担。)

有时你还要让别的对象创建它的内部类的对象。要想这么作，你就必须在 **new** 表达式里面给出宿主类对象的 reference，就像这样：

```
//: c08:Parcel11.java
// Creating instances of inner classes.

public class Parcel11 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) { label = whereTo; }
    String readLabel() { return label; }
  }
  public static void main(String[] args) {
    Parcel11 p = new Parcel11();
    // Must use instance of outer class
    // to create an instances of the inner class:
    Parcel11.Contents c = p.new Contents();
    Parcel11.Destination d = p.new
Destination("Tanzania");
  }
} ///:~
```

如果想直接创建内部类对象，你就不能想当然的用 **Parcel11** 来表示宿主类的名字，相反，你必须使用宿主类对象来创建内部类的对象：

```
Parcel11.Contents c = p.new Contents();
```


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

因此，除非你还已经创建了宿主类的对象，否则根本不可能创建内部类的对象。这是因为内部类的对象会悄悄的连到创建它的宿主类对象。但是，如果你创建的是嵌套类(**static** 的内部类)的话，那就不需要宿主类对象的 reference 了。

在多层嵌套的类里向外访问

[37]内部类的嵌套层次有多深并不是什么问题——它可以透明地访问它的各级宿主类的成员，就像这样：

```
//: c08:MultiNestingAccess.java
// Nested classes can access all members of all
// levels of the classes they are nested within.

class MNA {
  private void f() {}
  class A {
    private void g() {}
    public class B {
      void h() {
        g();
        f();
      }
    }
  }
}

public class MultiNestingAccess {
  public static void main(String[] args) {
    MNA mna = new MNA();
    MNA.A mnaa = mna.new A();
    MNA.A.B mnaab = mnaa.new B();
    mnaab.h();
  }
} ///:~
```

可以看到，**MNA.A.B** 无须任何授权就可以调用 **g( )**和 **f( )**(虽然它们都是 **private** 的)。这段例程还演示了，当你在另一个类里创建多层嵌套的内部类的对象的时候，应当使用哪种语法。“**.new**”语句指明了正确的作用域，因此你无需在调用构造函数的语句里再去限定类的名字了。

继承内部类

由于内部类的构造函数必须连到宿主类对象的 reference 上面，因此当你要继承内部类的时候，事情就有点复杂了。难就难在，这里有一个指向宿主类对象的“秘密的”reference 要进行初始化，而在派生类看来，它已经没有默认对象可连了。答案就是，使用一种专用的语法来明确地建立这种关系：

```java
//: c08:InheritInner.java
// Inheriting an inner class.

class WithInner {
  class Inner {}
}

public class InheritInner extends WithInner.Inner {
  //! InheritInner() {} // Won't compile
  InheritInner(WithInner wi) {
    wi.super();
  }
  public static void main(String[] args) {
    WithInner wi = new WithInner();
    InheritInner ii = new InheritInner(wi);
  }
} ///:~
```

可以看到 **InheritInner** 继承的只是内部类，而不是它的宿主类。但是等到要创建构造函数的时候，默认的构造函数玩不转了，你必须传给他宿主类对象的 reference。此外，你还必须在构造函数里面使用这种语法：

```java
enclosingClassReference.super();
```

这样才能提供那个必须的 reference，而程序也才能编译通过。

内部类可以被覆写吗？

假设你创建了一个内部类，然后又继承了它的宿主类，并且重新定义了那个内部类，那又会有什么结果呢？也就是说，可不可以把内部类彻底地覆写一遍？看上去这像是一种非常前卫的思想，但是像覆写宿主类的方法那样去“覆写”内部类，是不会有什么实际效果的：

```java
//: c08:BigEgg.java
// An inner class cannot be overriden like a method.
import com.bruceeckel.simpletest.*;

class Egg {
  private Yolk y;
  protected class Yolk {
    public Yolk()
{ System.out.println("Egg.Yolk()"); }
  }
  public Egg() {
    System.out.println("New Egg()");
    y = new Yolk();
  }
}

public class BigEgg extends Egg {
```


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

```
  private static Test monitor = new Test();
  public class Yolk {
    public Yolk()
{ System.out.println("BigEgg.Yolk()"); }
  }
  public static void main(String[] args) {
    new BigEgg();
    monitor.expect(new String[] {
      "New Egg()",
      "Egg.Yolk()"
    });
  }
} ///:~
```

编译器会自动生成一个默认的构造函数，而它又会调用基类的默认构造函数。可能你会认为，由于创建的是 **BigEgg**，因此应该调用覆写后的 **Yolk**，但是程序的输出告诉我们，事实并非如此。

这个例子告诉我们，当你继承宿主类的时候，内部类的戏法就到此为止了。这两个内部类是相互独立的的两个实体，它们都有自己的名字空间。但是，要想明确地继承那个内部类还是有办法的：

```
//: c08:BigEgg2.java
// Proper inheritance of an inner class.
import com.bruceeckel.simpletest.*;

class Egg2 {
  protected class Yolk {
    public Yolk()
{ System.out.println("Egg2.Yolk()"); }
    public void f()
{ System.out.println("Egg2.Yolk.f()");}
  }
  private Yolk y = new Yolk();
  public Egg2() { System.out.println("New Egg2()"); }
  public void insertYolk(Yolk yy) { y = yy; }
  public void g() { y.f(); }
}

public class BigEgg2 extends Egg2 {
  private static Test monitor = new Test();
  public class Yolk extends Egg2.Yolk {
    public Yolk()
{ System.out.println("BigEgg2.Yolk()"); }
    public void f() {
      System.out.println("BigEgg2.Yolk.f()");
    }
  }
  public BigEgg2() { insertYolk(new Yolk()); }
  public static void main(String[] args) {
    Egg2 e2 = new BigEgg2();
    e2.g();
    monitor.expect(new String[] {
      "Egg2.Yolk()",
      "New Egg2()",
```


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

```
        "Egg2.Yolk()",
        "BigEgg2.Yolk()",
        "BigEgg2.Yolk.f()"
      });
    }
  } ///:~
```

现在 **BigEgg2.Yolk** 明确的宣称它 **extends Egg2.Yolk**，并且覆写了它的方法。**insertYolk( )**方法能让 **BigEgg2** 把它自己的 **Yolk** 对象上传给 **Egg2** 的 **y** reference，这样 **g( )**所调用 **y.f( )**会使用覆写后的 **f( )**。**BigEgg2.Yolk** 的构造函数调用其基类构造函数的时候，**Egg2.Yolk( )**被第二次调用。这时你就会看到 **g( )**调用了经过覆写的 **f( )**。

本地内部类(Local inner classes)

我们前面讲过，你也可以在代码段里，通常就是方法的正文部分创建内部类。本地内部类不能有访问控制符，因为它并不属于宿主类，但是它确实可以访问当前代码段的 final 变量，以及宿主类的所有成员。下面这段程序对本地内部类和匿名内部类作了一个比较：

```
//: c08:LocalInnerClass.java
// Holds a sequence of Objects.
import com.bruceeckel.simpletest.*;

interface Counter {
  int next();
}

public class LocalInnerClass {
  private static Test monitor = new Test();
  private int count = 0;
  Counter getCounter(final String name) {
    // A local inner class:
    class LocalCounter implements Counter {
      public LocalCounter() {
        // Local inner class can have a constructor
        System.out.println("LocalCounter()");
      }
      public int next() {
        System.out.print(name); // Access local
final
        return count++;
      }
    }
    return new LocalCounter();
  }
  // The same thing with an anonymous inner class:
  Counter getCounter2(final String name) {
    return new Counter() {
      // Anonymous inner class cannot have a named
      // constructor, only an instance initializer:
      {
```


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

```
        System.out.println("Counter()");
      }
      public int next() {
        System.out.print(name); // Access local
final
        return count++;
      }
    };
  }
  public static void main(String[] args) {
    LocalInnerClass lic = new LocalInnerClass();
    Counter
      c1 = lic.getCounter("Local inner "),
      c2 = lic.getCounter2("Anonymous inner ");
    for(int i = 0; i < 5; i++)
      System.out.println(c1.next());
    for(int i = 0; i < 5; i++)
      System.out.println(c2.next());
    monitor.expect(new String[] {
      "LocalCounter()",
      "Counter()",
      "Local inner 0",
      "Local inner 1",
      "Local inner 2",
      "Local inner 3",
      "Local inner 4",
      "Anonymous inner 5",
      "Anonymous inner 6",
      "Anonymous inner 7",
      "Anonymous inner 8",
      "Anonymous inner 9"
    });
  }
} ///:~
```

Counter 会返回序列中的下一个值。程序分别用本地内部类和匿名内部类实现了这个接口，两者有着相同的行为和功能。由于本地内部类的名字在这个方法外面是没法访问的，因此用本地内部类来代替匿名内部类的唯一正当的理由就是，你需要一个有名字的构造函数，并且/或者要重载这个构造函数，因为匿名内部类只能进行实例初始化。

选择本地内部类而不是匿名内部类的唯一原因就是，你必须创建多个那种类的对象。

内部类的标识符(Inner class identifiers)

由于每个类都会生成一个**.class** 文件，以存储“该如何创建这个类的对象”的信息(这种信息会生成一种被成为 **Class** 对象的“meta-class”)，因此你可能会猜想，内部类也应该能生成保存它们的 **Class** 对象的信息的**.class** 文件。这些“文件/类”的名字都有着很严格的规

定：宿主类的名字，加上‘**$**’，再加上内部类的名字。例如 **LocalInnerClass.java** 所创建的**.class** 文件就包括：

```
Counter.class
LocalInnerClass$2.class
LocalInnerClass$1LocalCounter.class
LocalInnerClass.class
```

如果是匿名内部类，编译器就会直接用数字来表示内部类的标识符。如果内部类还嵌套在内部类里面，那么它们的名字会直接跟在宿主类后面的那个‘**$**’后面。

虽然这种内部类的命名方案十分简单，但是它却非常鲁棒，足以应付绝大多数的情况。[38]由于这是标准的 Java 命名规范，因此生成的文件自然就是与平台的无关的了。(注意，为了能让它正常运行，Java 编译器会动用各种办法来修改内部类。)

为什么要有内部类?

现在你已经看到了很多描述内部类运作方式的语法和语义了，但是这并不能回答“为什么要有内部类”这个问题。Sun 要为什么要这么麻烦地添加这个语言基本的特性呢？

通常，内部类都会继承别的什么类，或者实现某个 **interface**，而内部类里面的代码又可以操控创建它的那个宿主类。因此你可以说，内部类是一扇通往宿主类的窗口。

在内部类方面，真正击中要害的问题是：如果我要的只是 **interface** 的 reference，那为什么不让宿主类正正好好去实现这个 **interface** 呢？答案是“如果你有需要，那就一定能有办法做到。”那么，实现某个 **interface** 的内部类，同实现这个 **interface** 的宿主类相比，又有什么区别呢？答案就是，一旦牵涉到了实现，你就有可能得不到 **interface** 所提供的便利。因此内部类最吸引人的一点就是：

每个内部类都可以独立地继承某个“实现(implementation)”。因此，内部类不会受“宿主类是否已经继承了别的实现”的约束。

实际上，如果没有内部类所提供的继承多个实体类或 **abstract** 类的能力，很多设计和编程问题将会变得非常困难。因此，在如何看待内部类方面有一种观点，就把它当作彻底解决多重继承问题的办法。接口部分地解决了这个问题，但是内部类能让你真正做到 “继承多重实现(mutiple implementation inheritance)。”也就是说，内部类能让你在事实上继承多个非 **interface** 的类。

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

为了能讲得更具体一点，我们假设这样一个场景，无论如何你都必须在一个类里实现两个接口。由于继承接口的方式比较灵活，因此你有两种选择：一个单独的类，或是内部类：

```
//: c08:MultiInterfaces.java
// Two ways that a class can implement multiple
interfaces.

interface A {}
interface B {}

class X implements A, B {}

class Y implements A {
  B makeB() {
    // Anonymous inner class:
    return new B() {};
  }
}

public class MultiInterfaces {
  static void takesA(A a) {}
  static void takesB(B b) {}
  public static void main(String[] args) {
    X x = new X();
    Y y = new Y();
    takesA(x);
    takesA(y);
    takesB(x);
    takesB(y.makeB());
  }
} ///:~
```

当然，我们首先要肯定，这两种方法的思路都是很清楚的。不过，一般来说，你都可以从分析问题本质的过程中获得一些帮助，以决定该使用单独的一个类还是内部类。但是，如果没有别的条件，仅从实现的角度来看，上述这两种方法并没有什么差别。它们都能正常工作。

但是，如果你碰上的不是 **interface** 而是 **abstract** 类或实体类，而且还一定要同时实现这两个类的话，那么你就只能使用内部类了：

```
//: c08:MultiImplementation.java
// With concrete or abstract classes, inner
// classes are the only way to produce the effect
// of "multiple implementation inheritance."
package c08;

class D {}
abstract class E {}

class Z extends D {
  E makeE() { return new E() {}; }
}
```


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

```java
public class MultiImplementation {
  static void takesD(D d) {}
  static void takesE(E e) {}
  public static void main(String[] args) {
    Z z = new Z();
    takesD(z);
    takesE(z.makeE());
  }
} ///:~
```

如果你无需解决“多重实现的继承(mutiple implementation inheritance)”这类问题，那么即使没有内部类，你也完全可以解决问题。但是有了内部类，你就得到了如下的附加特性：

1. 内部类可以有多个实例，而每个又都可以有它自己的，与宿主类对象无关的状态信息。
2. 一个宿主类里可以放上好几个内部类，它们可以用各自不同的方式来实现同一个 **interface** 或继承同一个类。马上就会有这么一个例子的。
3. 内部类对象创建的时机与宿主类对象的创建没什么关系。
4. 内部类不存在什么让人头晕的“是”关系；他是一个独立的实体。

举例来说，要是 **Sequence.java** 没有使用内部类，那么你就只能说“**Sequence** 是一个 **Selector**”，于是每个 **Sequence** 里面就只能有一个 **Selector**。但是现在，你可以很方便地再定义一个 **getRSelector( )**方法，让它返回一个会倒过来访问这个序列的 **Selector**。只有内部类才能提供这种灵活性。

Closure 与回调(Closures & Callbacks)

closure 是一种能调用的对象，它记录了创建它的那个作用域信息。读过这段定义你就能看出，内部类就是一种面向对象的 closure，因为它不仅保存了宿主类的所有信息(“创建它的作用域”)，而且还自动保存指向那个宿主类对象的 reference，更何况它还有权操控这个对象的所有成员，即使它们是 **private** 的。

在要求 Java 提供指针机制的众多议论中，最具吸引力一条就是，要让它能进行回调 (*callback*)。有了回调，你就能给别的对象一段信息，然后在未来某个时点，让它反过来调用发起方对象了。随着本书的进展，你会发现这是一种非常巧妙的思路。但是，如果是用指针来实现回调，那你就只能指望程序员了，希望他们不要误用指针。正如你所看到的，Java 在这个问题上更为谨慎，因此它没有指针、

内部类所提供的 closure 是一个完美的解决方案——它比指针更灵活，也更安全。下面就是例子；

```
//: c08:Callbacks.java
// Using inner classes for callbacks
import com.bruceeckel.simpletest.*;

interface Incrementable {
  void increment();
}

// Very simple to just implement the interface:
class Callee1 implements Incrementable {
  private int i = 0;
  public void increment() {
    i++;
    System.out.println(i);
  }
}

class MyIncrement {
  void increment() {
    System.out.println("Other operation");
  }
  static void f(MyIncrement mi) { mi.increment(); }
}

// If your class must implement increment() in
// some other way, you must use an inner class:
class Callee2 extends MyIncrement {
  private int i = 0;
  private void incr() {
    i++;
    System.out.println(i);
  }
  private class Closure implements Incrementable {
    public void increment() { incr(); }
  }
  Incrementable getCallbackReference() {
    return new Closure();
  }
}

class Caller {
  private Incrementable callbackReference;
  Caller(Incrementable cbh) { callbackReference =
cbh; }
  void go() { callbackReference.increment(); }
}

public class Callbacks {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    Callee1 c1 = new Callee1();
    Callee2 c2 = new Callee2();
    MyIncrement.f(c2);
    Caller caller1 = new Caller(c1);
    Caller caller2 = new
Caller(c2.getCallbackReference());
    caller1.go();
    caller1.go();
    caller2.go();
    caller2.go();
```


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

```
    monitor.expect(new String[] {
      "Other operation",
      "1",
      "2",
      "1",
      "2"
    });
  }
} ///:~
```

这段程序还进一步揭示了，“让宿主类去实现接口”和“交给内部类去做”之间的区别。从编程的角度来说，很明显，**Callee1** 更简单一些。**Callee2** 继承了 **MyIncrement**，而 **MyIncrement** 已经包括了一个它自己的 **increment( )**，而且这个方法的功能同 **Incrementable** 接口所定义的毫不相关。所以 **Callee2** 继承 **MyIncrement** 以后，就不能靠实现 **Incrementable** 接口来覆写 **increment( )**了，这就逼着你只能使用内部类来提供一个独立的实现了。此外还有一点值得注意，创建内部类的时候，别去扩展或者修改宿主类的接口。

注意，除了 **getCallbackReference( )**之外， **Callee2** 里的所有东西都是 **private** 的。**interface Incrementable** 非常重要，它是维系类与外部世界的纽带。你可以从这里看到， **interface** 是如何将接口与实现分隔开来的。

内部类 **Closure** 实现了 **Incrementable**，以此提供了一个能返回 **Caller2** 的钩子——但是这是一个安全的钩子。无论谁得到了 **Incrementable** 的 reference 都能，当然也只能调用 **increment( )**，除此之外什么都干不了(不像指针，用起来就没谱了)。

Caller 的构造函数需要一个 **Incrementable** 的 reference(虽然获取“回调 reference”的行为可能会发生在任何时候)，然后过些时候，用这个 reference 来“回调”**Callee** 类。

回调的价值在于其灵活性；你可以在程序运行的时候动态地选择应该调用哪个方法。到了第 14 章，这点就能看得更清楚了。为实现 GUI 功能，回调会到处都是。

内部类与控制框架 (Inner classes & control frameworks)

内部类还有一种更实际的用法，我把它称为“控制框架(*control framework*)”。

“应用程序框架(*application framework*)”是一个或一组为解决某种特定类型的问题而设计的类。如果想使用应用程序框架，通常情况下只要继

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

承其中的一个或多个类，再覆写某些方法就可以了。应用程序框架为你提供了一套解决问题的通用方案，而你只要覆写方法就可以根据你的特殊要求定制这个方案了 (这就是一种“模板方法(Template Method)”设计模式；见 *www.BruceEckel.com* 的 *Thinking in Patterns(with Java)*)。控制框架是应用程序框架中的一种，主要用于响应事件；如果系统的首要任务就是对事件作出响应，那么它就被称为“事件驱动系统(*event-driven system*)”。图形用户界面(GUI)是创建应用程序时要解决的最棘手的问题之一，它差不多就是完全由时间驱动的。正如你将在第 14 章看到的，Java 的 Swing 类库就是一个控制框架。它通过频繁的使用内部类，非常潇洒地解决了 GUI 的难题。

为了看看控制框架究竟是怎样用内部类来创建和使用的，我们想像一下这样一个控制框架，其任务是：只要事件“准备完毕”，它就执行那些事件。尽管“准备完毕”可以有很多意思，但这里只是指时间。接下来是一个没有包含“到底在控制什么”的信息的控制框架。这个信息会在实现“模板方法”的时候通过继承获取。

首先是描述控制事件的接口。它不是一个 **interface**，而是一个 **abstract** 类，这是因为默认的行为是根据时间来进行控制。因此，这个类里已经有了一些实现：

```
//: c08:controller:Event.java
// The common methods for any control event.
package c08.controller;

public abstract class Event {
  private long eventTime;
  protected final long delayTime;
  public Event(long delayTime) {
    this.delayTime = delayTime;
    start();
  }
  public void start() { // Allows restarting
    eventTime = System.currentTimeMillis() +
delayTime;
  }
  public boolean ready() {
    return System.currentTimeMillis() >= eventTime;
  }
  public abstract void action();
} ///:~
```

当你想让 **Event** 运行的时候，它的构造函数会获取当前时间(对象创建的那个时刻)，然后调用 **start( )**。**start( )**的任务是用当前时间加上延时算出什么时候启动这个事件。**start( )**没有被放入构造函数，相反它被单独列成一个方法，这样事件完成之后，你就可以再次启动计时器以重新使用这个 **Event** 对象。比方说，如果你想重复这个事件，只要写一个 **action( )**方法，然后让它直接调用 **start( )**就行了。

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

ready( )会告诉你什么时候可以运行 **action( )**方法了。当然，派生类可以覆写 **ready( )**，这样就能把 **Event** 改造成不是基于时间的了。

下面这段程序是一个货真价实的，可以控制和发出事件控制框架。**Event** 对象被保存在一个被称为 **ArrayList** 的容器对象中。我们要到第 11 章才开始系统地学习 **ArrayList**，现在你只要知道 **add( )**会把 **Object** 添加到 **ArrayList** 的最后，**size( )**会告诉你 **ArrayList** 里面有多少对象，**get( )**可以根据下标从 **ArrayList** 里面把元素提取出来，而 **remove( )**会根据你给出下标把元素从 **ArrayList** 里面删除。

```
//: c08:controller:Controller.java
// With Event, the generic framework for control
systems.
package c08.controller;
import java.util.*;

public class Controller {
  // An object from java.util to hold Event objects:
  private List eventList = new ArrayList();
  public void addEvent(Event c) { eventList.add(c); }
  public void run() {
    while(eventList.size() > 0) {
      for(int i = 0; i < eventList.size(); i++) {
        Event e = (Event)eventList.get(i);
        if(e.ready()) {
          System.out.println(e);
          e.action();
          eventList.remove(i);
        }
      }
    }
  }
} ///:~
```

run( ) 方法会遍历整个 **eventList**，以寻找已经 **ready( )**的 **Event** 对象。一旦它找到一个 **ready( )**的对象，它会用 **toString( )**方法把它打印出来，然后在调用这个对象的 **action( )**方法，最后再在列表里面把这个 **Event** 删除。

注意，到目前为止，你还不知道要让 **Event** 干些什么。而这就是设计的要旨——怎样才能“将会变和不会变的东西分开来”。或者，用我的话来说，“变化的向量(vector of change)”就是各种 **Event** 对象的不同行为，而你的任务是，创建各种 **Event** 的子类来表达不同的行为。

现在内部类入场了。它们能做两件事：

1. 在一个类里完整地实现整个控制框架，这样就把“实现”里的所有独一无二东西全都给封装起来了。内部类则用来表示各种解决具体问题所需的 **action( )**。

2. 内部类可以让这个实现看上去不至于太过古怪，因为它还可以直接访问宿主类的所有成员。要不是这样的话，代码就会变得非常杂乱，这样到最后你肯定会去找一个替代方案的。

想想下面这个控制框架的具体实现，它是用来控制暖房的。[39]每个动作都截然不同：开灯，浇水，开或关温控器，响铃，重新启动系统。但是控制框架可以非常简单的隔离不同的代码。内部类能让你在一个类里从同一个基类，**Event**，派生出多个继承类。针对每种行为，你都可以继承一种新的 **Event** 内部类，然后把控制代码写进 **action( )**。

与典型的应用程序框架一样，**GreenhouseControls** 类是从 **Controller** 继承下来的：

```
//: c08:GreenhouseControls.java
// This produces a specific application of the
// control system, all in a single class. Inner
// classes allow you to encapsulate different
// functionality for each type of event.
import com.bruceeckel.simpletest.*;
import c08.controller.*;

public class GreenhouseControls extends Controller {
  private static Test monitor = new Test();
  private boolean light = false;
  public class LightOn extends Event {
    public LightOn(long delayTime)
{ super(delayTime); }
    public void action() {
      // Put hardware control code here to
      // physically turn on the light.
      light = true;
    }
    public String toString() { return "Light is
on"; }
  }
  public class LightOff extends Event {
    public LightOff(long delayTime)
{ super(delayTime); }
    public void action() {
      // Put hardware control code here to
      // physically turn off the light.
      light = false;
    }
    public String toString() { return "Light is
off"; }
  }
  private boolean water = false;
  public class WaterOn extends Event {
    public WaterOn(long delayTime)
{ super(delayTime); }
    public void action() {
      // Put hardware control code here.
      water = true;
    }
    public String toString() {
      return "Greenhouse water is on";
```

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

```
      }
    }
    public class WaterOff extends Event {
      public WaterOff(long delayTime)
  { super(delayTime); }
      public void action() {
        // Put hardware control code here.
        water = false;
      }
      public String toString() {
        return "Greenhouse water is off";
      }
    }
    private String thermostat = "Day";
    public class ThermostatNight extends Event {
      public ThermostatNight(long delayTime) {
        super(delayTime);
      }
      public void action() {
        // Put hardware control code here.
        thermostat = "Night";
      }
      public String toString() {
        return "Thermostat on night setting";
      }
    }
    public class ThermostatDay extends Event {
      public ThermostatDay(long delayTime) {
        super(delayTime);
      }
      public void action() {
        // Put hardware control code here.
        thermostat = "Day";
      }
      public String toString() {
        return "Thermostat on day setting";
      }
    }
    // An example of an action() that inserts a
    // new one of itself into the event list:
    public class Bell extends Event {
      public Bell(long delayTime) { super(delayTime); }
      public void action() {
        addEvent(new Bell(delayTime));
      }
      public String toString() { return "Bing!"; }
    }
    public class Restart extends Event {
      private Event[] eventList;
      public Restart(long delayTime, Event[] eventList)
  {
        super(delayTime);
        this.eventList = eventList;
        for(int i = 0; i < eventList.length; i++)
          addEvent(eventList[i]);
      }
      public void action() {
        for(int i = 0; i < eventList.length; i++) {
          eventList[i].start(); // Rerun each event
          addEvent(eventList[i]);
```


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

```
        }
        start(); // Rerun this Event
        addEvent(this);
      }
      public String toString() {
        return "Restarting system";
      }
    }
    public class Terminate extends Event {
      public Terminate(long delayTime)
  { super(delayTime); }
      public void action() { System.exit(0); }
      public String toString() { return
  "Terminating";  }
    }
  } ///:~
```

注意 **light**，**water** 和 **thermostat** 都属于宿主类 **GreenhouseControls**，而内部类无须任何特别的许可和权限就可以访问这些数据成员。此外绝大多数 **action( )**方法都牵涉到了一些硬件的控制。

绝大多数的 **Event** 类看上去都很像，但是 **Bell** 和 **Restart** 有些特别。**Bell** 响过之后还会往事件列表里面再放一个新的 **Bell** 对象，因此它会响两次。注意一下，内部类是怎样模拟多重继承的：**Bell** 和 **Restart** 具备了 **Event** 的所有方法，此外它看上去还具备了宿主类 **GreenhouseControls** 的所有方法。

Restart 拿了一个 **Event** 对象的数组，并把它加入“控制器(controller)”。由于 **Restart( )**也是一种 **Event** 对象，因此你可以在 **Restart.action( )**里面加一个 **Restart** 对象，这样系统就能定时自动重启了。

下面这个类通过创建 **GreenhouseControls** 对象，然后再添加各类 **Event** 对象，完成了系统的配置。这就是“命令(*Command*)”设计模式的一例：

```
//: c08:GreenhouseController.java
// Configure and execute the greenhouse system.
// {Args: 5000}
import c08.controller.*;

public class GreenhouseController {
  public static void main(String[] args) {
    GreenhouseControls gc = new GreenhouseControls();
    // Instead of hard-wiring, you could parse
    // configuration information from a text file
here:
    gc.addEvent(gc.new Bell(900));
    Event[] eventList = {
      gc.new ThermostatNight(0),
```


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

```
      gc.new LightOn(200),
      gc.new LightOff(400),
      gc.new WaterOn(600),
      gc.new WaterOff(800),
      gc.new ThermostatDay(1400)
    };
    gc.addEvent(gc.new Restart(2000, eventList));
    if(args.length == 1)
      gc.addEvent(
        gc.new  Terminate(Integer.parseInt(args[0])));
    gc.run();
  }
} ///:~
```

这个类要对系统进行初始化，因此会添加所有必要的事件。当然更好的办法还是不写代码，而是从文件读取。(第 12 章的练习里面有一道题会要你这样修改这个程序。)如果你在命令行下提供了参数，那么它会在经过这些毫秒之后中止这个程序(这是用来测试的)。

你应该能从这个例子里欣赏到内部类的价值了吧，特别是当它被用于控制框架的时候。但是，到第 14 章你还会看到，内部类是如何被优雅地用于描述图形用户界面的操作的。到了那时，你才算真正读完本章，你才会对内部类的价值完全信服。

总结

相比绝大多数别的 OOP 语言，接口和内部类的概念会更为复杂。比方说 C++就没类似的概念。此外，它们还解决了 C++试图用多重继承(MI)解决的问题。然而 C++的 MI 非常难用，相比之下，Java 的接口和内部类则简单了许多。

虽然，这些特性本身还是比较简单的，但是它们具体用法就属于设计的范畴了，这点同多态性非常相似。随着时间的推移，当你再碰到要选择使用接口还是内部类，或者两个都用的时候，就会好一点了。但是讲到这里，你至少应该已经熟悉它的语法和语义了。随着你越来越多的接触这些特性，最终会完全领悟它的精髓的。

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. 证明一下 **interface** 里面的数据成员默认就是 **static** 和 **final** 的。

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

2. 在它自己的 package 里面创建一个有三个方法的 interface。然后在另一个 package 里面实现这个接口。
3. 证明一下，**interface** 里面的方法自动就是 **public** 的。
4. 找到 **c07:Sandwich.java**，创建一个名叫 **FastFood** 的接口(编几个像样一点的方法)，然后修改 **Sandwich**，让它实现 **FastFood**。
5. 创建三个 **interface**，每个都要有两个方法。再写一个继承这三个 **interface** 的 **interface**，并且再添加一个新的方法。创建一个类，让它在实现这个新的 **interface** 的同时还要继承一个实体类。接下来写四个方法，每个方法都要拿一个 **interface** 做参数。在 **main( )**里面创建这样一个对象，然后传给各个方法。
6. 修改练习 5，创建一个 abstract 类，然后让派生类继承这个类。
7. 修改 **Music5.java**，加入一个 **Playable interface**。把 **play( )**方法的声明从 **Instrument** 移到 **Playable** 里面。让派生类实现 **Playable** 接口。修改 **tune( )**，让它拿 **Playable** 而不是 **Instrument** 作参数。
8. 修改第 7 章的练习 6，把 **Rodent** 改写成 **interface**。
9. 找到 **Adventure.java**，按照其它接口的形式添加一个叫 **CanClimb** 的接口。
10. 写一个要引入并使用 **Month.java** 的程序。
11. 参考 **Month.java**，创建一个一星期有几天的枚举类。
12. 在它自己的 package 里创建一个至少包括一个方法的 **interface**。加入一个实现这个 **interface** 的 **proctected** 的内部类。在第三个 package 里，继承这个类，然后用它的方法返回这个 **protected** 内部类的对象，再在返回的过程中将它上传给 **interface**。
13. 创建一个至少有一个方法的 **interface**，然后在方法里面定义一个实现这个接口的内部类，再让它返回这个 **interface** 的 reference。
    修改练习 13，但是要在方法的作用域里定义这个内部类。
14. 用匿名内部类改写练习 13。
15. 修改 **HorrorShow.java**，用匿名内部类实现 **DangerousMonster** 和 **Vampire** 接口。
16. 创建一个实现 **public interface** 的 **private** 内部类。写一个会返回这个 **private** 内部类实例的 reference 的方法，把它上传到 **interface**。试试能不能进行下传，检验一下内部类是不是完全把它隐藏起来了。
17. 创建一个带非默认构造函数(带参数的构造函数)，并且没有默认构造函数(无参数构造函数)的类。再创建一个类，其中要有一个会返回第一个的类的 reference 的方法。通过创建继承第一个类的匿名内部类的方式，创建并返回对象。
18. 创建一个带 **private** 数据成员和 **private** 方法的类。创建一个带有“能修改宿主类数据，并且能调用宿主类方法”的方法的内部类。再另一个宿主类方法里面创建一个内部类对象，再调用它的方法，然后看看宿主类对象有什么反映。
19. 用匿名内部类改写练习 19。

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

20. 创建一个包含嵌套类的类。在 **main( )**里面创建一个这种内部类的实例。
21. 创建一个包含嵌套类的 **interface**。实现这个 **interface**，并且创建一个这个嵌套类的实例。
22. 创建一个包含内部类的类，而这个内部类自己又包含一个内部类。用嵌套类重写一遍这个练习。注意一下编译器生成的**.class** 文件的名字。
23. 创建一个带内部类的类。在另一个类里创建一个这个内部类的实例。
24. 创建一个带非默认构造函数(需要参数的构造函数)的内部类的类。再创建一个带内部类的类，然后让这个内部类去继承第一个内部类。
25. 修复 **WindError.java** 中的问题。
26. 修改 **Sequence.java**，加一个 **getRSelector( )**方法，用它生成一种新的 **Selector interface** 的实现，这个 **Selector** 要能从后向前反过来访问这个序列。
27. 创建一个带三个方法的 **interface U**。创建一个带有“用创建匿名内部类方式返回 **U** 的 reference 的”方法的类 **A**。再创建一个包含一个 **U** 的数组的类 **B**。这个 **B** 应当有一个能将 **U** 的 reference 存进数组的方法，以及一个能将数组中的 reference (根据方法的参数) 设成 **null** 的方法，此外它还要有一个能遍历数组，并且调用 **U** 的方法的方法。用 **main( )**创建一组 **A** 对象以及一个 **B** 对象。用由 **A** 对象返回的 **U** 的 reference 填满 **B**。用 **B** 来回调 **A** 对象。从 **B** 里面删掉一些 **U** 的 reference。
28. 找到 **GreenhouseControls.java**，添加一个负责开关风扇(fan)的 **Event** 内部类。配置 **GreenHouseControls.java**，使之能使用这个新的 **Event** 对象。
29. 继承 **GreenhouseControls.java** 的 **GreenhouseControls** 类，添加一个负责开关水雾发生器的 **Event** 内部类。写一个新的 **GreenhouseController.java**，让它使用这个新的 **Event** 对象。
30. 证明一下内部类可以访问宿主类的 **private** 元素。看看反过来是不是也可以。

---

[33] Rich Hoffarth 的 email 给了我灵感。Joshua Bloch 在 *Effective Java* (Addison-Wesley, 2001)在第 21 条中对这个专题作了更深入的探讨。

[34]感谢 Martin Danner，他在培训班里问了这个问题。

[35]这种设计与 C++的“嵌套类(*nested classes*)”之间有着明显的区别。C++的嵌套类只是一种简单的名字隐藏的机制。它同宿主类的对象没有什么联系，也不存在什么默认的授权。

[36]大致同 C++的嵌套类相似，只是 C++的类不能像 Java 那样访问 private 的成员。

[37]还是要感谢 Martin Danner。

[38] 另一方面 **‘$’** 是 Unix shell 的特殊字符，因此显示**.class** 文件的时候，有时会遇到问题。考虑到 Sun 是一家 Unix 厂商，这就有点说不过去了。我想也许他们根本就没想过这个问题，相反他们认为你更关心的应该还是源代码文件。

[39]由于某种原因，我总是喜欢拿这个问题举例；我早期的那本 *C++ Inside & Out* 里面就有了这个例子，但是 Java 能提供一个更为潇洒的解决方案。

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