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

6: 复用类

Java 最令人心动的特性就是它的代码复用了。但是仅仅拷贝源代码再作修改是不能被称为“革命”的。

那是 C 之类的过程语言所采用的办法，而且也不怎么成功。就像 Java 里的一切，要解决这个问题还要靠类。你可以利用别人写好的、已经测试通过的类来创建新的类，不必一切都从零开始。

这么做的诀窍就是，要在不改动原有代码的前提下使用类。本章会介绍两种做法。第一种非常简单：在新的类里直接创建旧的类的对象。这被称为合成(*compostion*)，因为新的类是由旧的类合成而来的。你所复用的只是代码的功能，而不是它的形式。

第二种方法更为精妙。它会创建一个新的，与原来那个类同属一种类型的类。你全盘接受了旧类的形式，在没有对它做修改的情况下往里面添加了新的代码。这种神奇的做法就被称为继承(*inheritance*)。编译器会承担绝大部分的工作。继承是面向对象编程的基石，它还有一些额外的含义，对此我们会在第 7 章再做探讨。

合成与继承在语法和行为上有许多相似之处(这很好理解，因为它们都是在原有类的基础上创建新类)。你会在本章学到这些代码复用的机制。

合成所使用的语法

实际上我们已经看到很多合成的案例了。只要把对象的 reference 直接放到新的类里面就行了。假设，你要创建一个新的类，其中有几个 **String** 对象，几个 primitive 数据，以及一个别的什么类型的对象。对于非 primitive 的对象，你只要把它的 reference 放到类里就行了，但是对于 primitive，你就只能直接定义了：

```
//: c06:SprinklerSystem.java
// Composition for code reuse.
import com.bruceeckel.simpletest.*;

class WaterSource {
  private String s;
  WaterSource() {
    System.out.println("WaterSource()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class SprinklerSystem {
  private static Test monitor = new Test();
  private String valve1, valve2, valve3, valve4;
```

```
    private WaterSource source;
    private int i;
    private float f;
    public String toString() {
      return
        "valve1 = " + valve1 + "\n" +
        "valve2 = " + valve2 + "\n" +
        "valve3 = " + valve3 + "\n" +
        "valve4 = " + valve4 + "\n" +
        "i = " + i + "\n" +
        "f = " + f + "\n" +
        "source = " + source;
    }
    public static void main(String[] args) {
      SprinklerSystem sprinklers = new
  SprinklerSystem();
      System.out.println(sprinklers);
      monitor.expect(new String[] {
        "valve1 = null",
        "valve2 = null",
        "valve3 = null",
        "valve4 = null",
        "i = 0",
        "f = 0.0",
        "source = null"
      });
    }
  } ///:~
```

这两个类都定义了一个特殊的方法：**toString( )**。以后你就会知道，所有非 primitive 对象都有一个 **toString( )**方法，当编译器需要一个 **String** 而它却是一个对象的时候，编译器就会自动调用这个方法。所以当编译器从 **SprinklerSystem.toString( )**的：

```
"source = " + source;
```

中看到，你想把 **String** 同 **WaterSouce** 相加的时候，它就会说“由于 **String** 只能同 **String** 相加，因此我要调用 **source** 的 **toString( )**，因为只有这样才能把它转换成 **String**！”。 于是它就把这两个 **String** 连起来，然后再 **String** 的形式把结果返还给 **System.out.println( )**。如果你想让你写的类也具备这个功能，只要写一个 **toString( )**方法就行了。

我们已经在第 2 章讲过，当 primitive 数据作为类的成员的时候，会被自动地初始化为零。而对象的 reference 则会被初始化为 **null**，如果这时，你去调用这个对象的方法，就会得到异常。能把它打印出来而不抛出异常，这真是太好了(而且也很实用)。

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

“编译器不为 reference 准备默认对象”的这种做法，实际上也是很合乎逻辑的。因为在很多情况下，这么做会引发不必要的性能开销。如果你想对 reference 进行初始化，那么可以在以下几个时间进行：

1. 在定义对象的时候。这就意味着在构造函数调用之前，它们已经初始化完毕了。
2. 在这个类的构造函数里。
3. 在即将使用那个对象之前。这种做法通常被称为“偷懒初始化(*lazy initialization*)”。如果碰到创建对象的代价很高，或者不是每次都需要创建对象的时候，这种做法就能降低程序的开销了。

下面这段程序把这三种办法都演示一遍：

```
//: c06:Bath.java
// Constructor initialization with composition.
import com.bruceeckel.simpletest.*;

class Soap {
  private String s;
  Soap() {
    System.out.println("Soap()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class Bath {
  private static Test monitor = new Test();
  private String // Initializing at point of
definition:
    s1 = new String("Happy"),
    s2 = "Happy",
    s3, s4;
  private Soap castille;
  private int i;
  private float toy;
  public Bath() {
    System.out.println("Inside Bath()");
    s3 = new String("Joy");
    i = 47;
    toy = 3.14f;
    castille = new Soap();
  }
  public String toString() {
    if(s4 == null) // Delayed initialization:
      s4 = new String("Joy");
    return
      "s1 = " + s1 + "\n" +
      "s2 = " + s2 + "\n" +
      "s3 = " + s3 + "\n" +
      "s4 = " + s4 + "\n" +
      "i = " + i + "\n" +
      "toy = " + toy + "\n" +
      "castille = " + castille;
  }
  public static void main(String[] args) {
    Bath b = new Bath();
```

```
      System.out.println(b);
      monitor.expect(new String[] {
        "Inside Bath()",
        "Soap()",
        "s1 = Happy",
        "s2 = Happy",
        "s3 = Joy",
        "s4 = Joy",
        "i = 47",
        "toy = 3.14",
        "castille = Constructed"
      });
    }
} ///:~
```

注意，**Bath** 的构造函数会先打印一条消息再进行初始化。如果你不在定义对象的时候进行初始化，那么没人可以担保，在向这个对象的 reference 发送消息的时候，它已经被初始化了——反倒是会有异常来告诉你，它还没有初始化，。

调用 **toString( )**的时候它会先为 **s4** 赋一个值，这样它就不会未经初始化而被使用了。

继承所使用的语法

继承是 Java(也是所有 OOP 语言)不可分割的一部分。实际上当你创建类的时候，你就是在继承，要么是显式地继承别的什么类，要么是隐含地继承了标准 Java 根类，**Object**。

合成的语法很平淡，但继承就有所不同了。继承的时候，你得先声明“新类和旧类是一样的。”跟平常一样，你得先在程序里写上类的名字，但是在开始定义类之前，你还得加上 **extends** 关键词和基类(*base class*)的名字。做完这些之后，新类就会自动获得基类的全部成员和方法。下面就是一个例子：

```
//: c06:Detergent.java
// Inheritance syntax & properties.
import com.bruceeckel.simpletest.*;

class Cleanser {
  protected static Test monitor = new Test();
  private String s = new String("Cleanser");
  public void append(String a) { s += a; }
  public void dilute() { append(" dilute()"); }
  public void apply() { append(" apply()"); }
  public void scrub() { append(" scrub()"); }
  public String toString() { return s; }
  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
```


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

```
    System.out.println(x);
    monitor.expect(new String[] {
      "Cleanser dilute() apply() scrub()"
    });
  }
}

public class Detergent extends Cleanser {
  // Change a method:
  public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Call base-class version
  }
  // Add methods to the interface:
  public void foam() { append(" foam()"); }
  // Test the new class:
  public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    System.out.println(x);
    System.out.println("Testing base class:");
    monitor.expect(new String[] {
      "Cleanser dilute() apply() " +
      "Detergent.scrub() scrub() foam()",
      "Testing base class:",
    });
    Cleanser.main(args);
  }
} ///:~
```

这段程序能告诉我们很多东西。首先 **Cleanser** 的 **append( )**方法用 **+=** 运算符将 **Sting** 同 **s** 联接起来。Java 的设计者们“重载”了这个操作符，使之能作用于 **String**。

第二，**Cleanser** 和 **Detergent** 都有一个 **main( )**方法。你可以为每个类都创建一个 **main( )**，而且这也是一种值得提倡的编程方法，因为这样一来，测试代码就能都放进类里了。即使程序包括了很多类，它也只会调用你在命令行下给出的那个类的 **main( )**方法。(只要 **main( )**是 **public** 的就行了，至于类是不是 **public** 的，并不重要。)于是，当你输入 **java Detergent** 的时候，它就会调用 **Detergent.main( )**。虽然 **Cleanser** 不是 **public** 的，但是你也可以用 **java Cleanser** 来调用 **Cleanser.main( )**。这种往每个类里都放一个 **main( )**的做法，能让类的单元测试变得更容易一些。做完测试之后，你也不必移除 **main( )**；留下它可以供以后的测试用。

这里，**Detergent.main( )**直接调用了 **Cleanser.main( )**，并且把命令行参数原封不动地传给了它(实际上可以使用任何 **String** 数组)。

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

有一点很重要，那就是 **Cleanser** 的方法都是 **public** 的。记住，如果你不写访问控制符，成员就会被默认地赋予 package 权限，于是同一个 package 内的任何类就都能访问这些方法了。**Detergent** 没问题。但是，如果别的 package 里有一个继承了 **Cleanser** 的类，那它就只能访问 **Cleanser** 的 **public** 的成员了。(我们以后会讲，派生类可以访问基类的 **protected** 的成员。)所以继承设计方面有一条通用准则，那就是把数据都设成 **private** 的，把方法都设成 **public** 的。当然碰到特殊情况还要进行调整，但是这还是一条非常有用的准则。

注意，**Cleanser** 的接口包括了一组方法：**append()**，**dilute()**，**apply()**，**scrub()**，以及 **toString()**。由于 **Detergent** 是由 **Cleanser** 派生出来的(通过 **extends** 关键词)，所以尽管它没有明确地定义这些方法，它还是自动获得了这个接口的所有方法。由此，你可以将继承理解成类的复用。

正如 **scrub( )**所揭示的，你可以在派生类里修改一个在基类里的定义的方法。这时，你有可能要在新方法里调用基类的方法。但是你不能在 **scrub( )**里面直接调用 **scrub( )**，因为这是递归，你要的应该不是这个吧。为解决这个问题，Java 提供了一个表示当前类所继承的那个“超类(superclass)”的 **super** 关键词。于是 **super.scrub( )**就会调用基类的 **scrub( )**方法了。

继承并不会限定你只能使用基类的方法。你也可以往派生类里加进新的方法，就像往普通的类里加方法一样：直接定义就是了。**foam( )**就是一例。

从 **Degergent.main( )**可以看出，**Detergent** 对象既有 **Cleanser** 的方法，也有它自己的方法(就是 **foam()**)。

基类的初始化

现在要创建派生类对象已经不是一个类的事情了，它会牵涉到两个类——基类和派生类，因此要搞清楚它究竟是怎么创建的，就有点难度了。从局外人的角度来看，新类具备了和旧类完全相同的接口，并且还有可能会有一些它自己的方法和数据。但继承并不仅仅是拷贝基类的接口。当你创建一个派生类对象的时候，这个对象里面还有一个基类的子对象(*subobject*)。这个子对象同基类自己创建的对象没什么两样。只是从外面看来，这个子对象被包裹在派生类的对象里面。

当然，基类子对象的正确初始化也是非常重要的，而且只有一个办法能保证这一点：调用基类的构造函数来进行初始化，因为只有它才掌握怎样才能正确地进行初始化的信息和权限。Java 会让派生类的构造函数自动地调用基类的构造函数。下面这段程序就演示了它在三级继承体系下是如何运作的：

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

```
//: c06:Cartoon.java
// Constructor calls during inheritance.
import com.bruceeckel.simpletest.*;

class Art {
  Art() {
    System.out.println("Art constructor");
  }
}

class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}

public class Cartoon extends Drawing {
  private static Test monitor = new Test();
  public Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
    monitor.expect(new String[] {
      "Art constructor",
      "Drawing constructor",
      "Cartoon constructor"
    });
  }
} ///:~
```

可以看到，构造行为是从基类“向外”发展的，所以基类会在派生类的构造函数访问它之前先进行初始化。即便你不创建 **Cartoon( )**的构造函数，编译器也会为你造一个默认的构造函数，然后再由它去调用基类的构造函数。

带参数的构造函数

在上述例程中，构造函数都是默认的；也就是不带参数的。对编译器来说，调用这种构造函数会非常简单，因为根本就没有要传哪些参数的问题。但是如果类没有默认的构造函数(也就是无参数的构造函数)，或者你要调用的基类构造函数是带参数的，你就必须用 **super** 关键词以及合适的参数明确地调用基类的构造函数：

```
//: c06:Chess.java
// Inheritance, constructors and arguments.
import com.bruceeckel.simpletest.*;

class Game {
  Game(int i) {
    System.out.println("Game constructor");
  }
}
```


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

```
class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    System.out.println("BoardGame constructor");
  }
}

public class Chess extends BoardGame {
  private static Test monitor = new Test();
  Chess() {
    super(11);
    System.out.println("Chess constructor");
  }
  public static void main(String[] args) {
    Chess x = new Chess();
    monitor.expect(new String[] {
      "Game constructor",
      "BoardGame constructor",
      "Chess constructor"
    });
  }
} ///:~
```

如果你不在 **BoardGame( )**里面调用基类的构造函数，编译器就会报错说它找不到 **Game( )** 形式(译者注：即默认)的构造函数。此外，对派生类构造函数而言，调用基类的构造函数应该是它做的第一件事。(如果你做错了，编译器就会提醒你。)

捕获基类构造函数抛出的异常

我们刚说了，编译器会强制你将基类构造函数的调用放在派生类的构造函数的最前面。也就是说，在它之前不能有任何东西。等到第 9 章你就会知道，这么做会妨碍派生类的构造函数捕获基类抛出的异常。这一点有时会很不方便。

把合成和继承结合起来

同时使用合成和继承的现象是很普遍的。下面这段程序演示了，怎样使用合成和继承，以及利用构造函数来进行初始化这一必不可少的步骤，来创建一个较为复杂的类：

```
//: c06:PlaceSetting.java
// Combining composition & inheritance.
import com.bruceeckel.simpletest.*;

class Plate {
  Plate(int i) {
    System.out.println("Plate constructor");
  }
}
```


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

```
class DinnerPlate extends Plate {
  DinnerPlate(int i) {
    super(i);
    System.out.println("DinnerPlate constructor");
  }
}

class Utensil {
  Utensil(int i) {
    System.out.println("Utensil constructor");
  }
}

class Spoon extends Utensil {
  Spoon(int i) {
    super(i);
    System.out.println("Spoon constructor");
  }
}

class Fork extends Utensil {
  Fork(int i) {
    super(i);
    System.out.println("Fork constructor");
  }
}

class Knife extends Utensil {
  Knife(int i) {
    super(i);
    System.out.println("Knife constructor");
  }
}

// A cultural way of doing something:
class Custom {
  Custom(int i) {
    System.out.println("Custom constructor");
  }
}

public class PlaceSetting extends Custom {
  private static Test monitor = new Test();
  private Spoon sp;
  private Fork frk;
  private Knife kn;
  private DinnerPlate pl;
  public PlaceSetting(int i) {
    super(i + 1);
    sp = new Spoon(i + 2);
    frk = new Fork(i + 3);
    kn = new Knife(i + 4);
    pl = new DinnerPlate(i + 5);
    System.out.println("PlaceSetting constructor");
  }
  public static void main(String[] args) {
    PlaceSetting x = new PlaceSetting(9);
    monitor.expect(new String[] {
      "Custom constructor",
      "Utensil constructor",
      "Spoon constructor",
```

```
      "Utensil constructor",
      "Fork constructor",
      "Utensil constructor",
      "Knife constructor",
      "Plate constructor",
      "DinnerPlate constructor",
      "PlaceSetting constructor"
    });
  }
} ///:~
```

虽然编译器会强制你对基类进行初始化，并且会要求你在构造函数的开始部分完成初始化，但是它不会检查你是不是进行了成员对象的初始化，因此你只能自己留神了。

确保进行妥善地清理

拆构函数(*destructor*)是 C++里面的概念，它是一种能在清理对象的时候自动调用的方法，Java 里面没有这种概念。原因可能是 Java 处理这类问题的时候，只是简单地把对象放到一边，然后留给垃圾回收器去处理，它不会去主动地进行清理。

在大多数情况下，这种做法也很不错，但是有时候，会遇到一些特殊的类，在清理它们的对象的时候会需要进行一些额外的操作。正如第 4 章所说的，你既不知道垃圾回收器什么时候启动，也不知道它会不会启动。所以如果要进行清理，你就必须明确地写一个专门干这件事的方法，然后告诉客户程序员们去调用这个方法。做了这些还不够——到第 9 章(“用异常处理错误”)还要讲——为了应付异常，你还要把它放到 **finally** 子句里面。

就拿计算机辅助设计系统举例，我们要在屏幕上画一点东西：

```
//: c06:CADSystem.java
// Ensuring proper cleanup.
package c06;
import com.bruceeckel.simpletest.*;
import java.util.*;

class Shape {
  Shape(int i) {
    System.out.println("Shape constructor");
  }
  void dispose() {
    System.out.println("Shape dispose");
  }
}

class Circle extends Shape {
  Circle(int i) {
    super(i);
```


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

```
    System.out.println("Drawing Circle");
  }
  void dispose() {
    System.out.println("Erasing Circle");
    super.dispose();
  }
}

class Triangle extends Shape {
  Triangle(int i) {
    super(i);
    System.out.println("Drawing Triangle");
  }
  void dispose() {
    System.out.println("Erasing Triangle");
    super.dispose();
  }
}

class Line extends Shape {
  private int start, end;
  Line(int start, int end) {
    super(start);
    this.start = start;
    this.end = end;
    System.out.println("Drawing Line: "+ start+ ", 
"+ end);
  }
  void dispose() {
    System.out.println("Erasing Line: "+ start+ ", 
"+ end);
    super.dispose();
  }
}

public class CADSystem extends Shape {
  private static Test monitor = new Test();
  private Circle c;
  private Triangle t;
  private Line[] lines = new Line[5];
  public CADSystem(int i) {
    super(i + 1);
    for(int j = 0; j < lines.length; j++)
      lines[j] = new Line(j, j*j);
    c = new Circle(1);
    t = new Triangle(1);
    System.out.println("Combined constructor");
  }
  public void dispose() {
    System.out.println("CADSystem.dispose()");
    // The order of cleanup is the reverse
    // of the order of initialization
    t.dispose();
    c.dispose();
    for(int i = lines.length - 1; i >= 0; i--)
      lines[i].dispose();
    super.dispose();
  }
  public static void main(String[] args) {
    CADSystem x = new CADSystem(47);
    try {
```

```
      // Code and exception handling...
    } finally {
      x.dispose();
    }
    monitor.expect(new String[] {
      "Shape constructor",
      "Shape constructor",
      "Drawing Line: 0, 0",
      "Shape constructor",
      "Drawing Line: 1, 1",
      "Shape constructor",
      "Drawing Line: 2, 4",
      "Shape constructor",
      "Drawing Line: 3, 9",
      "Shape constructor",
      "Drawing Line: 4, 16",
      "Shape constructor",
      "Drawing Circle",
      "Shape constructor",
      "Drawing Triangle",
      "Combined constructor",
      "CADSystem.dispose()",
      "Erasing Triangle",
      "Shape dispose",
      "Erasing Circle",
      "Shape dispose",
      "Erasing Line: 4, 16",
      "Shape dispose",
      "Erasing Line: 3, 9",
      "Shape dispose",
      "Erasing Line: 2, 4",
      "Shape dispose",
      "Erasing Line: 1, 1",
      "Shape dispose",
      "Erasing Line: 0, 0",
      "Shape dispose",
      "Shape dispose"
    });
  }
} ///:~
```

这个系统里的所有东西都是 **Shape**(而 **Shape** 本身又是 **Object**，因为它是隐含地继承自根类)。各个类在覆写 **Shape** 的 **dispose( )**方法的时候，除了用 **super** 调用基类的 **dispose( )** 之外，还要完成它自己的清理活动。具体的 **Shape** 类——**Circle**，**Triangle** 以及 **Line**——都有会把自己“画出来”的构造函数，但实际上，对象的生命周期内调用的任何方法，都可能会造成一些需要进行清理的后果。每个类都有它自己的，用来恢复内存以外的资源状态的 **dispose( )**方法。

main( )里面有两个我们要到第 9 章才会正式介绍的新关键词：**try** 和 **finally**。**try** 表示下面这段程序(由花括号限定)是一个需要给予特殊关注的受保护的区域(*guarded region*)。所谓的特殊关注就是，无论以何种方式退出 **try** 区块，都必须执行跟在这个受保护区域后面的 **finally** 子

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

句。(对于异常处理来说，会有很多非正常退出 **try** 区块的情况。)这里 **finally** 的意思是“不论发生什么事情，你都必须调用 **x** 的 **dispose( )**”。我们会在第 9 章再详细解释这两个关键词。

注意，在清理方法中，如果子对象之间有依赖关系，那么你还要留意其基类和成员对象的清理方法的调用顺序。总之， 这个顺序同 C++的编译器要求的拆构函数的执行顺序是一样的：先按照创建对象的相反顺序进行类的清理。(一般来说，这要求留着基类对象以供访问。)然后调用基类的清理方法，就像这里所做的。

在很多情况下，清理并不是什么问题；把它留给垃圾回收器就行了。但是如果你要自己做的话，那就只能辛苦一点了，而且还要格外小心，因为在垃圾回收方面，谁都帮不上你。垃圾回收器可能永远也不会启动。即便它启动了，你也没法控制它的回收顺序。最好不要依赖垃圾回收器去做任何与内存回收无关的事情。如果你要进行清理，一定要自己写清理方法，别去用 **finalize( )**。

名字的遮盖

如果 Java 的基类里有一个被重载了好几次的方法，那么在派生类里重新定义那个方法，是不会把基类里定义的任何一个给遮盖掉的(这点同 C++不同)。因此，无论方法是在这一层还是在其基类定义的，重载都能起作用：

```
//: c06:Hide.java
// Overloading a base-class method name in a derived
class
// does not hide the base-class versions.
import com.bruceeckel.simpletest.*;

class Homer {
  char doh(char c) {
    System.out.println("doh(char)");
    return 'd';
  }
  float doh(float f) {
    System.out.println("doh(float)");
    return 1.0f;
  }
}

class Milhouse {}

class Bart extends Homer {
  void doh(Milhouse m) {
    System.out.println("doh(Milhouse)");
  }
}

public class Hide {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    Bart b = new Bart();
```

```
      b.doh(1);
      b.doh('x');
      b.doh(1.0f);
      b.doh(new Milhouse());
      monitor.expect(new String[] {
        "doh(float)",
        "doh(char)",
        "doh(float)",
        "doh(Milhouse)"
      });
    }
  } ///:~
```

可以看到，尽管 **Bart** 又重载了一遍，但 **Homer** 所重载的方法，在 **Bart** 里依然有效(在 C++里这么做的话，就会把基类方法全都隐藏起来了)。到下一章你就会知道，在派生类里用相同的参数列表，相同的返回类型来覆写方法的这种做法，实在是太普通了。否则就太乱了(这也是为什么 C++不允许你这么做的原因——要防止你去做可能会是错的事)。

用合成还是继承

合成与继承都能让你将子对象植入新的类(合成是显式的，继承是隐含的)。也许你想了解一下这两者有什么区别，以及该如何进行选择。

一般来说，合成用于新类要使用旧类的功能，而不是其接口的场合。也就是说，把对象嵌进去，用它来实现新类的功能，但是用户看到的是新类的接口，而不是嵌进去的对象的接口。因此，你得在新类里嵌入 **private** 的旧类对象。

有时，让用户直接访问新类的各个组成部分也是合乎情理的；这就是说，将成员对象定义成 **public**。成员对象各自都有“隐藏实现”的机制，因此这么做也是安全的。如果用户知道你用了哪些零件，那么接口对他们来说就变得更简单了。**car** 对象就是一个好例子：

```
//: c06:Car.java
// Composition with public objects.

class Engine {
  public void start() {}
  public void rev() {}
  public void stop() {}
}

class Wheel {
  public void inflate(int psi) {}
}

class Window {
  public void rollup() {}
  public void rolldown() {}
```

```
}

class Door {
  public Window window = new Window();
  public void open() {}
  public void close() {}
}

public class Car {
  public Engine engine = new Engine();
  public Wheel[] wheel = new Wheel[4];
  public Door
    left = new Door(),
    right = new Door(); // 2-door
  public Car() {
    for(int i = 0; i < 4; i++)
      wheel[i] = new Wheel();
  }
  public static void main(String[] args) {
    Car car = new Car();
    car.left.window.rollup();
    car.wheel[0].inflate(72);
  }
} ///:~
```

由于在这个例子里，**car** 的各个组成部分(不仅仅是其底层实现的一部分)还是一个分析问题的过程，因而将成员定义成 **public** 的，有助于客户程序员理解该如何使用这个类，由此也降低了这个类自身的开发难度。但是要记住这只是一个特列，通常情况下，你都应该将成员数据定义成 **private** 的。

继承则是要对已有的类做一番改造，以此获得一个特殊版本。简而言之，你要将一个较为抽象的类改造成能适用于某些特定需求的类。稍微想一下就会知道，用 vehicle(车辆)对象来合成一个 car(轿车)是毫无意义的——car 不包含 vehicle，它本来就是 vehicle。继承要表达的是一种“是(*is-a*)”关系，而合成表达要表达的是“有(*has-a*)”关系。

protected

现在你已经知道继承了，因此关键词 **protected** 也有意义了。在理想情况下 **private** 关键词已经够用了。但是在实际的项目中，你有时会碰到，要让一些东西对外部世界隐藏，但是却要对它的继承类开放。**protected** 关键词就是这种实用主义的体现。它的意思是“对用户而言，它是 **private** 的，但是如果你想继承这个类，或者开发一个也属于这个 **package** 的类的话，就可以访问它了。”(Java 的 **protected** 也提供 package 的权限。)

最好的做法是，将数据成员设成 **private** 的；你应该永远保留修改底层实现的权利。然后用 **protected** 权限的方法来控制继承类的访问权限：

```
//: c06:Orc.java
// The protected keyword.
import com.bruceeckel.simpletest.*;
import java.util.*;

class Villain {
  private String name;
  protected void set(String nm) { name = nm; }
  public Villain(String name) { this.name = name; }
  public String toString() {
    return "I'm a Villain and my name is " + name;
  }
}

public class Orc extends Villain {
  private static Test monitor = new Test();
  private int orcNumber;
  public Orc(String name, int orcNumber) {
    super(name);
    this.orcNumber = orcNumber;
  }
  public void change(String name, int orcNumber) {
    set(name); // Available because it's protected
    this.orcNumber = orcNumber;
  }
  public String toString() {
    return "Orc " + orcNumber + ": " +
super.toString();
  }
  public static void main(String[] args) {
    Orc orc = new Orc("Limburger", 12);
    System.out.println(orc);
    orc.change("Bob", 19);
    System.out.println(orc);
    monitor.expect(new String[] {
      "Orc 12: I'm a Villain and my name is
Limburger",
      "Orc 19: I'm a Villain and my name is Bob"
    });
  }
} ///:~
```

可以看到 **change( )**调用了 **set( )**，因为它是 **protected** 的。 此外还要注意一下 **Orc** 的 **toString( )**方法，它用到了基类的 **toString( )** 方法。

渐进式的开发

继承的优点之一就是，它支持渐进式的开发(*incremental develop*)。添加新的代码的时候，不会给老代码带来 bug；实际上新的 bug 全都被圈在新代码里。通过继承已有的，已经能正常工作的类，然后再添加一些数据成员和方法(以及重新定义一些原有的方法)，你可以不去修改那些可能还有人在用的老代码，因而也就不会造成 bug 了。一旦发现了 bug，你

就知道它肯定是在新代码里。相比要去修改老代码，新代码会短很多，读起来也更简单。

类的隔离竟会如此彻底，这真是太令人惊讶了。你甚至不需要源代码就能进行复用。最多就是 import 一个 package。(对于继承和合成而言都是这样。)

你得明白，程序开发就像人的学习一样，是一个渐进的过程。不论你作过多少分析，不实际做项目的话，还是得不到答案。如果你能摒弃像建玻璃摩天楼那样毕其功于一役的开发方式，而采用类似生物进化的，让那个项目逐步的“增长”的开发方式，那么你就会获得更大的成功——以及更多的及时反馈。

尽管在试验阶段，继承是一种非常有用的技术，但是当项目进入稳定阶段之后，你就得用一种新的眼光来审视类的继承体系了，你要把它压缩成一个合乎逻辑的结构。记住，在这些错综复杂的关系后面，继承实质上是在表达这样一种关系：“新的类是一种旧的类”。程序不应该围着 bit 转，它应该从问题空间出发，通过创建和操控形形色色的对象来表达一种解决问题的方法。

上传

继承最重要的特征不在于它为新类提供了方法，而是它表达了新类同基类之间的关系。这种关系可以被归纳为一句话“新类就是一种原有的类。”

这并不是在空口说白话——语言直接给了支持。比方说，表示乐器的基类叫 **Instrument**，然后有一个叫 **Wind** 的派生类。继承的意思就是基类有的方法派生类都有，因此送给基类的消息也可以送给派生类。如果 **Instrument** 有一个 **play( )**方法，那么 **Wind** 也有。也就是说，你可以很有把握地说，**Wind** 对象也是一种 **Instrument**。下面这段程序演示了编译器是怎样支持这种观念的：

```
//: c06:Wind.java
// Inheritance & upcasting.
import java.util.*;

class Instrument {
  public void play() {}
  static void tune(Instrument i) {
    // ...
    i.play();
  }
}

// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
  public static void main(String[] args) {
    Wind flute = new Wind();
```

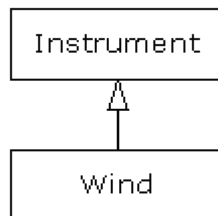

www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

```
    Instrument.tune(flute); // Upcasting
  }
} ///:~
```

这个例子里的 **tune( )**方法很有趣。它需要 **Instrument** 的 reference 作参数，但是 **Wind.main( )**给了它一个 **Wind** 的 reference。我们知道 Java 的类型检查是很挑剔的，因此如果你不知道 **Wind** 对象就是一种 **Instrument** 对象，而且除了 **Wind** 之外 **tune( )**没有别的 **Instrument** 可调，你就会觉得很困惑，为什么接受 **Instrument** 的方法也可以接受 **Wind**。**tune( )**的代码可以作用于 **Instrument**，以及 **Instrument** 的派生类，而将 **Wind** 的 reference 转换成 **Instrument** 的 reference 的这种做法就被称为“上传(*upcasting*)”。

为什么叫“上传”？

这个术语是有讲法的，它缘于类的继承关系图的传统画法：将根置于顶端，然后向下发展(当然，你也可以按照你的习惯来画。)**Wind.java** 的继承关系图就是：



把派生类传给基类就是沿着继承图往上送，因此被称为“上传(*upcasting*)”。上传总是安全的，因为你是把一个较具体的类型转换成较为一般的类型。也就是说派生类是基类的超集(superset)。它可能会有一些基类所没有的方法，但是它最少要有基类的方法。在上传过程中，类的接口只会减小，不会增大。这就是为什么编译器会允许你不作任何明确的类型转换或特殊表示就进行上传的原因了。

你也可以进行反向传递，这被称为“下传(*downcasting*)”，但是这时就会有问题了。我们会在第 10 章再作讲解。

合成还是继承，再探讨

在面相对象的编程中，最常见的编程和使用代码的方式还是将数据和方法简单地封装成类，然后再使用那个类的对象。你也可以通过合成，在现有的类的基础上创建新的类。继承则不太常用。所以，虽然在 OOP 的学习中，继承占有很重要的地位，但这并不是在说你可以到处滥用。相反，运

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

用继承的时候，你应该尽可能的保守，只有在它能带来很明显的好处的时候，你才能用。在判断该使用合成还是继承的时候，有一个最简单的办法，就是问一下你是不是会把新类上传给基类。如果你必须上传，那么继承就是必须的，如果不需要上传，那么就该再看看是不是应该用继承了。下一章(多态性)会讲为什么要用上传，但是如果你还记得要问自己“我需要上传吗？”，那么你就有了一件能帮你判断该使用合成还是继承的好工具了。

final 关键词

Java 的关键词 **final** 的含义会根据上下文略有不同，但是总的来说，它的意思都是“这样东西不允许改动”。你可能会出于两点考虑不想让别人作改动：设计和效率。由于这两个原因差别很大，因此很可能会误用 **final** 关键词。

下面的几节会讨论 **final** 的三种用途：数据(data)，方法(method)和类(class)。

Final 的数据

很多编程语言都有通知编译器“这是段『常量(constant)』 数据”的手段。常量能用于下列两种情况：

1. 可以是“编译时的常量(*compile-time constant*)”，这样就再也不能改了。
2. 也可以是运行时初始化的值，这个值你以后就不想再改了。

如果是编译时的常量，编译器会把常量放到算式里面；这样编译的时候就能进行计算，因此也就降低了运行时的开销。在 Java 中这种常量必须是 primitive 型的，而且要用 **final** 关键词表示。这种常量的赋值必须在定义的时候进行。

一个既是 **static** 又是 **final** 的数据成员会只占据一段内存，并且不可修改。

当 **final** 不是指 primitive，而是用于对象的 reference 的时候，意思就有点搞了。对 primitive 来说，**final** 会将这个值定义成常量，但是对于对象的 reference 而言，**final** 的意思则是这个 reference 是常量。初始化的时候，一旦将 reference 连到了某个对象，那么它就再也不能指别的对象了。但是这个对象本身是可以修改的；Java 没有提供将某个对象作成常量的方法。(但是你可以自己写一个类，这样就能把类当做常量了。)这种局限性也体现在数组上，因为它也是一个对象。

下面这段程序演示了 **final** 的数据成员：

```java
//: c06:FinalData.java
// The effect of final on fields.
import com.bruceeckel.simpletest.*;
import java.util.*;

class Value {
  int i; // Package access
  public Value(int i) { this.i = i; }
}

public class FinalData {
  private static Test monitor = new Test();
  private static Random rand = new Random();
  private String id;
  public FinalData(String id) { this.id = id; }
  // Can be compile-time constants:
  private final int VAL_ONE = 9;
  private static final int VAL_TWO = 99;
  // Typical public constant:
  public static final int VAL_THREE = 39;
  // Cannot be compile-time constants:
  private final int i4 = rand.nextInt(20);
  static final int i5 = rand.nextInt(20);
  private Value v1 = new Value(11);
  private final Value v2 = new Value(22);
  private static final Value v3 = new Value(33);
  // Arrays:
  private final int[] a = { 1, 2, 3, 4, 5, 6 };
  public String toString() {
    return id + ": " + "i4 = " + i4 + ", i5 = " + i5;
  }
  public static void main(String[] args) {
    FinalData fd1 = new FinalData("fd1");
    //! fd1.VAL_ONE++; // Error: can't change value
    fd1.v2.i++; // Object isn't constant!
    fd1.v1 = new Value(9); // OK -- not final
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // Object isn't constant!
    //! fd1.v2 = new Value(0); // Error: Can't
    //! fd1.v3 = new Value(1); // change reference
    //! fd1.a = new int[3];
    System.out.println(fd1);
    System.out.println("Creating new FinalData");
    FinalData fd2 = new FinalData("fd2");
    System.out.println(fd1);
    System.out.println(fd2);
    monitor.expect(new String[] {
      "%% fd1: i4 = \\d+, i5 = \\d+",
      "Creating new FinalData",
      "%% fd1: i4 = \\d+, i5 = \\d+",
      "%% fd2: i4 = \\d+, i5 = \\d+"
    });
  }
} ///:~
```

由于 **VAL_ONE** 和 **VAL_TWO** 都是在编译时赋值的 **final** primitive，因而它们都能被用作编译时的常量，这两者在所有重大的方

面完全相同。**VAL_THREE** 则用了一种更常见的方式来定义常量：**public**，所以即使是在 package 的外面也能用，**static** 强调它只有这一个数据，而 **final** 表示这是一个常量。注意，通常约定，被初始化为常量值的 **final static** 的 primitive 的名字全都用大写，词与词之间用下划线分开。(这就同 C 的常量很相似了，实际上这个约定就是从 C 那里拿来的。)同样要知道 **i5** 的值在编译的时候是不知道的，因此不需要大写。

不能仅从某样东西是 **final** 的，就判断说“它的值在编译的时候就已经确定了”。这一点可以从 **i4** 和 **i5** 的初始化上得到证实。它们都使用随机生成的数字来进行初始化。这段例程还演示了将 **fianl** 值做成 **static** 和非 **static** 的区别。这种差别只在程序执行初始化的时候才能显现出来，因为编译器处理“编译时的值(compile-time values)”的方式是相同的。(假设不存在优化的话。)运行程序的时候就能看出差别了。注意，**fd1** 和 **fd2** 的 **i4** 值是不同的，而创建第二个 **FinalData** 对象的时候 **i5** 的值是不变的。这是因为它是 **static** 的，因而它是在装载类的时候，而不是创建对象的时候进行初始化的。

v1 到 **v3** 的变量演示了 **final** refeerence 的含义。正如你在 **main( )** 中所看到的，不会因为 **v2** 是 **final** 的就不让它修改对象的值。因为 **final** 的是 reference，它的意思是你不能把 **v2** 绑到其它对象上。对于数组也是这个意思，因为它也是一种 reference。(我不知道有什么办法把数组本身做成 **final** 的。)看来把 reference 作成 **final** 的不如把 primitive 作成 **final** 的有用。

空白的 final 数据 (Blank finals)

Java 能让你创建“空白的 final 数据(*blank finals*)”，也就是说把数据成员声明成 **final** 的，但却没给初始化的值。碰到这种情况，你必须先进行初始化，再使用空白的 final 数据成员，而且编译器会强制你这么做。不过，空白的 final 数据也提供了一种更为灵活的运用 **final** 关键词方法，比方说，现在对象里的 **final** 数据就能在保持不变性的同时又有所不同了。下面就是一例：

```java
//: c06:BlankFinal.java
// "Blank" final fields.

class Poppet {
  private int i;
  Poppet(int ii) { i = ii; }
}

public class BlankFinal {
  private final int i = 0; // Initialized final
  private final int j; // Blank final
  private final Poppet p; // Blank final reference
  // Blank finals MUST be initialized in the
constructor:
  public BlankFinal() {
    j = 1; // Initialize blank final
```


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

```
    p = new Poppet(1); // Initialize blank final
reference
  }
  public BlankFinal(int x) {
    j = x; // Initialize blank final
    p = new Poppet(x); // Initialize blank final
reference
  }
  public static void main(String[] args) {
    new BlankFinal();
    new BlankFinal(47);
  }
} ///:~
```

你一定得为 **final** 数据赋值，要么是在定义数据的时候用一个表达式赋值，要么是在构造函数里面进行赋值。为了确保 **final** 数据在使用之前已经进行了初始化，这一要求是强制性的。

Final 的参数

Java 允许你在参数表中声明参数是 **final** 的，这样参数也编程 **final** 了。也就是说，你不能在方法里让参数 reference 指向另一个对象了：

```
//: c06:FinalArguments.java
// Using "final" with method arguments.

class Gizmo {
  public void spin() {}
}

public class FinalArguments {
  void with(final Gizmo g) {
    //! g = new Gizmo(); // Illegal -- g is final
  }
  void without(Gizmo g) {
    g = new Gizmo(); // OK -- g not final
    g.spin();
  }
  // void f(final int i) { i++; } // Can't change
  // You can only read from a final primitive:
  int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    FinalArguments bf = new FinalArguments();
    bf.without(null);
    bf.with(null);
  }
} ///:~
```

f( )和 **g( )**演示了把 primitive 参数做成 **final** 的效果：你可以读，但是不能改参数。这种功能好像也没什么大用，也许也不是你所需要的。

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

Final 方法

使用 **final** 方法的目的有二。第一，为方法上“锁”，禁止派生类进行修改。这是出于设计考虑。当你希望某个方法的功能，能在继承过程中被保留下来，并且不被覆写，就可以使用这个方法。

第二个原因就是效率。如果方法是 **final** 的，那么编译器就会把调用转换成“内联的(*inline*)”。当编译器看到要调用 **final** 方法的时候，它就会(根据判断)舍弃普通的，“插入方法调用代码的”编译机制(将参数压入栈，然后跳去执行要调用的方法的代码，再跳回来清空栈，再处理返回值)，相反它会用方法本身的拷贝来代替方法的调用。当然如果方法很大，那么程序就会膨胀得很快，于是内联也不会带来什么性能的改善，因为这种改善相比程序处理所耗用的时间是微不足道的。Java 的设计者们暗示过，Java 的编译器有这个功能，可以智能地判断是不是应该将 **final** 方法做成内联的。不过，最好还是把效率问题留给编译器和 JVM 去处理，而只把 **final** 用于要明确地禁止覆写的场合。[31]

final 和 private

private 方法都隐含有 **final** 的意思。由于你不能访问 **private** 的方法，因此你也不能覆写它。你可以给 **private** 方法加一个 **final** 修饰符，但是这样做什么意义也没有。

这个问题有可能会造成混乱，因为即使你覆写了一个 **private** 方法(它隐含有 **final** 的意思)，看上去它还是可以运行的，而且编译器也不会报错：

```
//: c06:FinalOverridingIllusion.java
// It only looks like you can override
// a private or private final method.
import com.bruceeckel.simpletest.*;

class WithFinals {
  // Identical to "private" alone:
  private final void f() {
    System.out.println("WithFinals.f()");
  }
  // Also automatically "final":
  private void g() {
    System.out.println("WithFinals.g()");
  }
}

class OverridingPrivate extends WithFinals {
  private final void f() {
    System.out.println("OverridingPrivate.f()");
  }
  private void g() {
    System.out.println("OverridingPrivate.g()");
  }
}
```

```
class OverridingPrivate2 extends OverridingPrivate {
  public final void f() {
    System.out.println("OverridingPrivate2.f()");
  }
  public void g() {
    System.out.println("OverridingPrivate2.g()");
  }
}

public class FinalOverridingIllusion {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    OverridingPrivate2 op2 = new
OverridingPrivate2();
    op2.f();
    op2.g();
    // You can upcast:
    OverridingPrivate op = op2;
    // But you can't call the methods:
    //! op.f();
    //! op.g();
    // Same here:
    WithFinals wf = op2;
    //! wf.f();
    //! wf.g();
    monitor.expect(new String[] {
      "OverridingPrivate2.f()",
      "OverridingPrivate2.g()"
    });
  }
} ///:~
```

只有是基类接口里的东西才能被“覆写”。也就是说，对象应该可以被上传到基类，然后再调用同一个方法(这一点要到下一章才能讲得更清楚。)如果方法是 **private** 的，那它就不属于基类的接口。它只能算是被类隐藏起来的，正好有着相同的名字的代码。如果你在派生类里创建了同名的 **public** 或 **protected**，或 package 权限的方法，那么它们同基类中可能同名的方法，没有任何联系。你并没有覆写那个方法，你只是创建了一个新的方法。由于 **private** 方法是无法访问的，实际上是看不见的，因此这么作除了会影响类的代码结构，其它什么意义都没有。

Final 类

把整个类都定义成 **final** 的(把 **final** 关键词放到类的定义部分的前面)就等于在宣布，你不会去继承这个类，你也不允许别人去继承这个类。换言之，出于类的设计考虑，它再也不需要作修改了，或者从安全角度出发，你不希望它再生出子类。

```
//: c06:Jurassic.java
// Making an entire class final.

class SmallBrain {}
```

```
final class Dinosaur {
  int i = 7;
  int j = 1;
  SmallBrain x = new SmallBrain();
  void f() {}
}

//! class Further extends Dinosaur {}
// error: Cannot extend final class 'Dinosaur'

public class Jurassic {
  public static void main(String[] args) {
    Dinosaur n = new Dinosaur();
    n.f();
    n.i = 40;
    n.j++;
  }
} ///:~
```

注意，**final** 类的数据可以是 **final** 的，也可以不是 **final** 的，这要由你来决定。无论类是不是 **final** 的，这一条都适用于“将 **final** 用于数据的”场合。但是，由于 **final** 类禁止了继承，覆写方法已经不可能了，因此所有的方法都隐含地变成 **final** 了。你可以为 **final** 类的方法加一个 **final** 修饰符，但是这一样没什么意义。

小心使用 final

看来，设计类的时候将方法定义成 **final** 的，会是一个很明智的决定。可能你会觉得没人会要覆写你的方法。有时确实是这样。

但是你这么假设的时候一定要非常谨慎。一般来说，要事先预想“类会怎样被复用”是非常困难的，特别是对那些很通用的类来说。如果你把类定义成 **final** 的，那么很可能会发生这种情况，由于你没有料到这个类还能被这么使用，其它项目的程序员就没法通过继承来复用这个类了。

标准 Java 类库就是一个活生生的例子。特别是 Java 1.0/1.1 的 **Vector** 类，这个类曾被广泛使用，如果不是为了追求效率(天晓得提高了多少)而把它的所有方法都做成 **final** 的话，它的用途可能会更广。这个类太有用了，因此应该很容易想到会有人要继承它并且覆写其中的方法，但是类的设计者们不知怎么搞的，认定这么作是不对的。有两个理由使得这种想法变得非常具有讽刺意味。首先，**Stack** 是继承自 **Vector** 的，也就是说 **Stack** 就是 **Vector**，但是在逻辑上这种说法并不正确。第二，**Vector** 的很多重要的方法，比如 **addElement( )**以及 **elementAt( )**，都是 **synchronized**。正如你会在第 11 章看到的，这样作会造成很严重的性能下降，并且完全抵消 **final** 所带来的优化。这更让我们相信了，程序员在猜测该在哪里作优化的时候总是犯错。在标准类库里面放进如此笨拙的设计实在是太糟了，但是大家还都不得不迁就。

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

(所幸得是 Java 2 的容器类库用 **ArrayList** 替换了 **Vector**，而它的工作方式要文雅了许多。但不幸的是，还有人在用老的容器类库写新程序。)

再看看 **Hashtable**，也是很有意思的。它是 Java 1.0/1.1 标准类库里的另一个重要的类，它没有任何 **final** 方法。曾几何时，我在本书中说过，这些类很明显都是由一群不相干的人设计出来的。(还有一个证据，你可以比较一下 **Hashtable** 和 **Vector** 的方法的名字，前者的要简洁许多。)这绝对应该是类库的使用者们不应该看出来的东西。如果设计缺乏连贯性，用户就得受苦——这又是在强调设计和代码复查的重要性了。

初始化与类的装载

在较传统的编程语言中，程序启动的时候都是一次装载所有的东西，然后进行初始化，接下来再开始执行。这些语言必须仔细的控制初始化的过程，这样 **static** 数据的初始化才不至于会产生问题。就拿 C++为例，如果一个 **static** 数据要依赖另一个 **static** 的数据，而它又没有初始化的话，问题就来了。

Java 采用了一种新的装载模式，因此没有这种问题。Java 的所有东西都是对象，因此很多事情都变得简单了，这就是一例。下一章你还会学的更具体。编译之后每个类都保存在它自己的文件里。不到需要的时候，这个文件是不会装载的。总之你可以说“类的代码会在它们第一次使用的时候装载”。类的装载通常都发生在第一次创建那个类的对象的时候，但是访问 **static** 数据或 **static** 方法的时候也会装载。

第一次使用 **static** 数据的时候也是进行初始化的时候。装载的时候，**static** 对象和 **static** 代码段会按照它们字面的顺序(也就是在程序中出现的顺序)进行初始化。当然 **static** 数据只会初始化一次。

继承情况下的初始化

了解一下包括继承在内的初始化的过程将是非常有益的，这样就能有个总体的了解。看看下面这段代码：

```
//: c06:Beetle.java
// The full process of initialization.
import com.bruceeckel.simpletest.*;

class Insect {
  protected static Test monitor = new Test();
  private int i = 9;
  protected int j;
  Insect() {
    System.out.println("i = " + i + ", j = " + j);
    j = 39;
  }
  private static int x1 =
    print("static Insect.x1 initialized");
```


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

```
  static int print(String s) {
    System.out.println(s);
    return 47;
  }
}

public class Beetle extends Insect {
  private int k = print("Beetle.k initialized");
  public Beetle() {
    System.out.println("k = " + k);
    System.out.println("j = " + j);
  }
  private static int x2 =
    print("static Beetle.x2 initialized");
  public static void main(String[] args) {
    System.out.println("Beetle constructor");
    Beetle b = new Beetle();
    monitor.expect(new String[] {
      "static Insect.x1 initialized",
      "static Beetle.x2 initialized",
      "Beetle constructor",
      "i = 9, j = 0",
      "Beetle.k initialized",
      "k = 47",
      "j = 39"
    });
  }
} ///:~
```

当你用 Java 运行 **Beetle** 的时候，第一件事就是访问了 **Beetel.main( )**(这是一个 **static** 方法)，于是装载器(loader)就会为你寻找经编译的 **Beetle** 类的代码(也就是 **Beetle.class** 文件)。在装载的过程中，装载器注意到它有一个基类(也就是 **extends** 所要表示的意思)，于是它再装载基类。不管你创不创建基类对象，这个过程总会发生。(试试看，把创建对象的那句注释掉，看看会有什么结果。)

如果基类还有基类，那么这第二个基类也会被装载，以此类推。下一步，它会执行“根基类(root base class)”(这里就是 **Insect**)的 **static** 初始化，然后是下一个派生类的 **static** 初始化，以此类推。这个顺序非常重要，因为派生类的“静态初始化(即前面讲的 **static** 初始化)”有可能要依赖基类成员的正确初始化。

现在所有必要的类都已经装载结束，可以创建对象了。首先，对象里的所有的 primitive 都会被设成它们的缺省值，而 reference 也会被设成 **null**——这个过程是一瞬间完成的，对象的内存会被统一地设置成“两进制的零(binary zero)”。然后调用基类的构造函数。调用是自动发生的，但是你可以使用 **super** 来指定调用哪个构造函数(也就是 **Beetle( )** 构造函数所做的第一件事)。基类的构造过程以及构造顺序，同派生类的相同。基类构造函数运行完毕之后，会按照各个变量的字面顺序进行初始化。最后会执行构造函数的其余部分。

总结

继承和合成都能让你在已有的类的基础上创建新的类。但是通常情况下，合成是把已有的类当作新类底层实现的一部分来复用，而继承则是复用其接口。由于派生类拥有基类的接口，因此它可以被上传(*upcast*)到基类，正如你将再下一章看到的，这点对于多态性是非常重要。

尽管面向对象的编程会反复强调继承，但是当你着手设计的时候，通常情况下还是应该先考虑合成，只有在必要的时候才使用继承。合成会更灵活。此外，还可以让成员使用继承类的对象，这样你就能在运行时更换这些成员的具体类型，及其行为了。于是，合成后的对象的行为方式也能得以改变了。

设计系统的时候，你的目标是要找到或者创建一组这样的类，它们每个都有具体的用途，并且都不是太大(塞了太多功能，复用起来就不方便了)，当然也不能太小了(功能不足的话就不能独立完成任务了)。

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. 创建两个带默认构造函数(空的参数类列表)的类 **A** 和 **B**。再创建一个继承 **A** 的 **C** 类，**C** 类里要有一个 **B** 类型的成员对象。不要创建 **C** 的构造函数。创建一个 **C** 类的对象，然后观察一下其运行结果。
2. 修改练习 1 的程序，使 **A** 和 **B** 都有带参数的构造函数。再为 **C** 写一个构造函数，然后让它执行全部的初始化工作。
3. 先创建一个简单的类。然后在第二个类里，定义一个第一个类的对象的 reference。用“偷懒初始化(lazy initialization)”来实例化这个对象。
4. 写一个继承 **Detergent** 的新类。覆写其 **scrub( )**方法，再加一个新的 **sterilize( )**方法。
5. 找到 **Cartoon.java**，将 **Cartoon** 类的构造函数注释掉，看看会有什么效果，再解释一下为什么。
6. 找到 Chess.java，将 Chess 类的构造函数注释掉，看看会有什么效果，再解释一下为什么。
7. 试着证明编译器会为你创建一个默认的构造函数。
8. 试着证明基类的构造函数(a)总是会被调用(b)会在调用派生类的构造函数之前调用。
9. 创建一个只有非默认构造函数的基类，以及一个既有默认构造函数又有非默认构造函数的派生类。在派生类的构造函数里调用基类的构造函数。

10. 创建一个名为 **Root**，并且包含 **Component1**，**Component2**，以及 **Component3** 这三个类(也要由你来写)的实例的类。写一个继承 **Root** 的 **Stem** 类，它也要包含这三个“component”。所有的类都应该有能打印类的消息的默认构造函数。
11. 修改练习 10，使得每个类都只有一个非默认的构造函数。
12. 为练习 11 中的各个类添加合适的 **dispose( )**方法。
13. 创建一个重载三次方法的类。写一个新的，继承这个类，并且添加一个新的重载方法的类，然后演示一下，这四个方法都是可以访问的。
14. 找到 **Car.java**，往 **Engin** 里面添加一个 **service( )**方法，然后在 **main( )**里面调用这个方法。
15. 在 package 里面创建一个类。这个类应该包括一个 **protected** 的方法。在这个类外面，调用这个 **protected** 方法，然后解释一下为什么会出现这种情况。然后，继承这个类，再在继承类的方法里面调用这个 **protected** 的方法。
16. 创建一个名为 **Amphibian** 的类。然后继承下一个 **Frog** 类。在基类里面适当地放一些方法。用 **main( )**创建一个 **Frog**，再上传给 **Amphibian**，看看，这些方法是不是还能继续用。
17. 修改练习 16，让 **Frog** 覆写基类中定义的方法(使用相同的方法特征，但是要重新定义)。看看 **main( )**会有什么效果。
18. 创建一个带 static final 成员和 final 成员的类，看看这两者有什么区别。
19. 创建一个带“空白的 **final** 的”reference 的类。所有的构造函数都要对这个 **final** 数据进行初始化。证明一下，“**final** 在使用前必须进行初始化”以及“一旦初始化之后就不能再修改了”，这两点是有保障的。
20. 创建一个带 **final** 方法的类。继承这个类，并试着去覆写这个类。
21. 创建一个 **final** 类，并试着去继承这个类。
22. 试着证明“类只会装载一次”。证明“第一次创建对象的实例”，以及“访问 **static** 的成员”都能引发类的装载。
23. 找到 **Beetle.java**，按照现有类的格式创建一个具体的继承类。跟踪并且解释程序的输出。

---

[31]不要的陷入过早优化代码的陷阱。假如你有了一个能正常工作但是运行速度很慢的系统，那能不能用 **final** 解决问题还是件很难说的事。不过我们会在第 15 章介绍 profiling，这是一个能帮你改善程序运行速度的工具。

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