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

7: 多态性

多态性是继数据抽象和继承之后的，面向对象的编程语言的第三个基本特性。

它提供了另一个层面的接口与实现的分离，也就是说把做什么和怎么做分开来。多态性不但能改善代码的结构，提高其可读性，而且能让你创建可扩展的(*extensible*)程序。所谓“可扩展”是指，程序不仅在项目最初的开发阶段能“成长”，而且还可以在需要添加新特性的时候“成长”。

“封装”通过将数据的特征与行为结合在一起，创建了一种新的数据类型。“隐藏实现”通过将细节设成 **private**，完成了接口与实现的分离。之所以要采取这种比较呆板的顺序来讲解，是要照顾那些过程语言的程序员们。但是，多态性是站在“类”的角度来处理这种逻辑上的分离的。在上一章中，你看到了，“继承”是怎样允许你将对象当作它自己的，或者它的基类的类型来处理的。这是一个很重要的功能，因为它能让你把多个类(派生自同一个基类的)当作一个类来处理，这样一段代码就能作用于很多不同的类型了。“多态方法调用(polymorphic method call)”能让类表现出各自所独有的特点，只要这些类都是从同一个基类里派生出来的就行了。当你通过基类的 reference 调用方法的时候，这些不同就会通过行为表现出来。

本章会从基础开始，通过一些简单的，只涉及多态行为的程序，来讲解多态性(也被称为动态绑定『*dynamic binding*』、后绑定『*late binding*』或运行时绑定『*run-time bingding*』)。

再访上传(upcasting)

你已经在第 6 章看到，怎样把对象当作它自己的或是它的基类的对象来使用。把对象的 reference 当作基类的 reference 来用，被称为上传(*upcasting*)。因为在继承关系图中，基类总是在上面的。

但是问题也来了。下面我们用乐器来举例。由于乐器要演奏 **Note**(音符)，所以我们在 package 里单独创建一个 **Note** 类：

```
//: c07:music:Note.java
// Notes to play on musical instruments.
package c07.music;
import com.bruceeckel.simpletest.*;

public class Note {
  private String noteName;
  private Note(String noteName) {
    this.noteName = noteName;
  }
  public String toString() { return noteName; }
```


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

```
  public static final Note
    MIDDLE_C = new Note("Middle C"),
    C_SHARP  = new Note("C Sharp"),
    B_FLAT   = new Note("B Flat");
    // Etc.
} ///:~
```

这是一个“枚举(enumeration)”类，它创建了几个固定对象以供选择。你不能再创建其它对象了，因为构造函数是 private 的。

在接下去的程序中，**Wind** 作为一种乐器继承了 **Instrument**：

```
//: c07:music:Music.java
// Inheritance & upcasting.
package c07.music;
import com.bruceeckel.simpletest.*;

public class Music {
  private static Test monitor = new Test();
  public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    tune(flute); // Upcasting
    monitor.expect(new String[] {
      "Wind.play() Middle C"
    });
  }
} ///:~
```

```
//: c07:music:Wind.java
package c07.music;

// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
  // Redefine interface method:
  public void play(Note n) {
    System.out.println("Wind.play() " + n);
  }
} ///:~
```

Music.tune( )需要一个 **Instrument** 的 reference 作参数，但是它也可以接受任何由 **Instrument** 派生出来的 reference。当 **main( )** 未经转换就把 **Wind** 的 reference 传给 **tune( )**的时候，这一切就发生了。这是完全可以可行的——因为 **Wind** 继承了 **Instrument**，因此它必须实现 **Instrument** 的接口。从 **Wind** 上传到 **Instrument** 的时

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

候，接口可能会“变窄”，但是再小也不会比 **Instrument** 的接口更小。

把对象的类型忘掉

可能你会觉得 **Music.java** 有些奇怪。为什么会有人要故意“忘掉”对象的类型呢？上传就是在做这件事情。但是，让 **tune( )**直接拿 **Wind** 的 reference 作参数好像更简单一些。但是这会有一个问题：如果采用这种方法，你就得为系统里的每个 **Instrument** 都写一个新的 **tune( )** 方法。假设我们顺着这个思路，再加一个 **Stringed** (弦乐器)和一个 **Brass** (管乐器)：

```
//: c07:music:Music2.java
// Overloading instead of upcasting.
package c07.music;
import com.bruceeckel.simpletest.*;

class Stringed extends Instrument {
  public void play(Note n) {
    System.out.println("Stringed.play() " + n);
  }
}

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

public class Music2 {
  private static Test monitor = new Test();
  public static void tune(Wind i) {
    i.play(Note.MIDDLE_C);
  }
  public static void tune(Stringed i) {
    i.play(Note.MIDDLE_C);
  }
  public static void tune(Brass i) {
    i.play(Note.MIDDLE_C);
  }
  public static void main(String[] args) {
    Wind flute = new Wind();
    Stringed violin = new Stringed();
    Brass frenchHorn = new Brass();
    tune(flute); // No upcasting
    tune(violin);
    tune(frenchHorn);
    monitor.expect(new String[] {
      "Wind.play() Middle C",
      "Stringed.play() Middle C",
      "Brass.play() Middle C"
    });
  }
} ///:~
```


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

这种做法不是不可以，但是有个重大缺陷：每次加入新的 **Instrument** 的时候，你都必须专门为这个类写一个方法。这就意味着，不但定义类的时候要多写代码，而且添加 **tune( )** 之类的方法，或者添加新的 **Instrument** 的时候，还会多出很多事情。此外，如果你忘了重载某个方法，编译器是不会报错的，于是类型处理工作就完全乱套了。

如果你可以写只一个用基类，而不是具体的派生类作参数的方法，那会不会更好一些呢？也就是，如果你可以忘掉它们都是派生类，只写同基类打交道的代码，那会不会更好呢？

这就是多态性要解决的问题。然而绝大多数从过程语言转过来的程序员们，在理解多态性的运行机制的时候，都会些问题。

问题的关键

Music.java 让人觉得费解的地方就是，运行的时候，真正产生输出的是 **Wind.play( )**。诚然，这正是我们所希望，但是它却没有告诉我们它为什么要这样运行。看看 **tune( )**方法：

```
public static void tune(Instrument i) {
  // ...
  i.play(Note.MIDDLE_C);
}
```

它接受一个 **Instrument** 的 reference 做参数。那么编译器怎么会知道这个 **Instrument** 的 reference 就指向一个 **Wind**，而不是 **Brass** 或 **Stringed** 呢？编译器不可能知道。为了能深入的理解这个问题，我们先来看看什么是“绑定(*binding*)”。

方法调用的绑定

将方法的调用连到方法本身被称为“绑定(*binding*)”。当绑定发生在程序运行之前时(如果有的话，就是由编译器或连接器负责)被称作“前绑定(*early binding*)”。可能你从没听说过这个术语，因为面向过程的语言根本就没有这个概念。C 的编译器只允许一种方法调用，那就是前绑定。

上述例程之所以令人费解都是源于前绑定，因为当编译器只有一个 **Instrument** 的 reference 的时候，它是不知道该连到哪个方法的。

解决方案就是“后绑定(*late binding*)”，它的意思是要在程序运行的时候，根据对象的类型来决定该绑定哪个方法。后绑定也被称为“动态绑定(*dynamic binding*)”或“运行时绑定(*run-time binding*)”。如果语言实现了后绑定，那它就必须要有能在运行时判断对象类型，并且调用其

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

合适的方法的机制。也就是说，编译器还是不知道对象的类型，但是方法的调用机制会找出，并且调用正确的方法。后绑定机制会随语言的不同而不同，但是你可以设想，对象里面必定存有“它属于哪种类型”的信息。

除了 **static** 和 **final** 方法(**private** 方法隐含有 **final** 的意思)，Java 的所有的方法都采用后绑定。也就是说，通常情况下你不必考虑是不是应该采用后绑定——它是自动的。

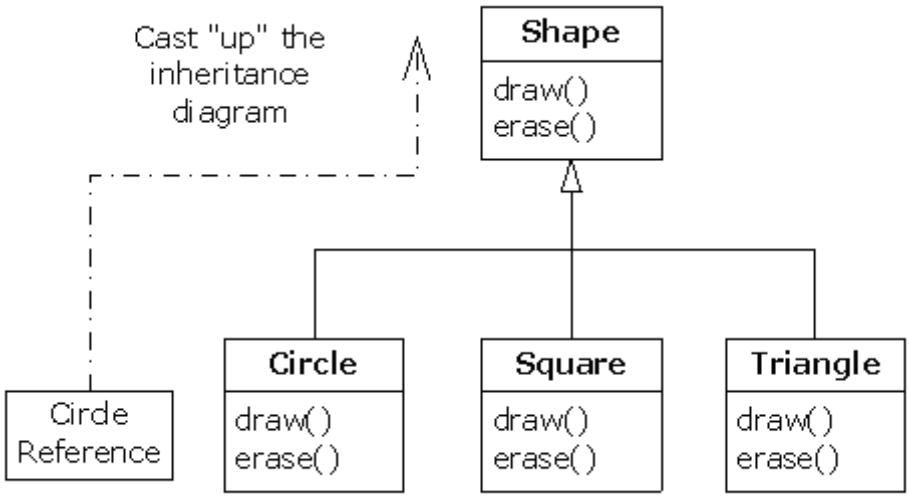
为什么要声明 **final** 方法？我们在上一章指出，这样可以禁止别人覆写那个方法。不过，更重要的可能还是要“关闭”它的动态绑定，或者更确切的说，告诉编译器这里不需要使用后绑定。这样编译器就能为 **final** 方法生成稍微高效一些的调用代码。然而在绝大多数情况下，这种做法并不会对程序的总体性能产生什么影响，因此最好还是只把 **final** 当作一种设计手段来用，而不要去考虑用它来提高性能。

产生正确的行为

一旦知道 Java 通过后绑定实现了多态的方法调用，你就可以只编写同基类打交道的代码了。因为你知道所有的派生类也能正确地使用这些代码。或者换一个说法，你“向对象发一个消息，让它自己判断该做些什么。”

“形状” 就是讲解 OOP 的一个经典的例子。它看起来直观，因此被广泛使用，但是不幸的是，这会让新手误以为 OOP 只是用来处理图像编程的，这显然不对。

在这个例子中，基类被称作 **Shape**，它有好几个派生类：**Circle**，**Square**，**Triangle**，等等。这个例子之所以好，是因为我们能很自然地说“圆形是一种形状”，而听的人也能明白。下面的继承关系图体现了这种关系：



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

下面这句就是在“上传”：

```
Shape s = new Circle();
```

这里先创建了一个 **Circle** 对象，接着马上把它的 reference 赋给了 **Shape**。看上去这像是一个错误(一种类型怎么能赋给另一种)；但是由于 **Circle** 是由 **Shape** 派生出来的，**Circle** 就是一种 **Shape**，因此这种做法非常正确。所以编译器会毫不含糊地接受这条语句，什么错都不报。

假设你调用了一个基类方法(派生类已经覆写这个方法)：

```
s.draw();
```

可能你会认为，这次应该总调用 **Shape** 的 **draw( )**了吧，因为毕竟这是 **Shape** 的 reference——编译器又怎么会知道还要做其它事情呢？但是由于实现了后绑定(多态性)，实际上它会调用 **Circle.draw( )**。

下面的例程稍微作了一些变化：

```
//: c07:Shapes.java
// Polymorphism in Java.
import com.bruceeckel.simpletest.*;
import java.util.*;

class Shape {
  void draw() {}
  void erase() {}
}

class Circle extends Shape {
  void draw() {
    System.out.println("Circle.draw()");
  }
  void erase() {
    System.out.println("Circle.erase()");
  }
}

class Square extends Shape {
  void draw() {
    System.out.println("Square.draw()");
  }
  void erase() {
    System.out.println("Square.erase()");
  }
}

class Triangle extends Shape {
  void draw() {
```


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

```
    System.out.println("Triangle.draw()");
  }
  void erase() {
    System.out.println("Triangle.erase()");
  }
}

// A "factory" that randomly creates shapes:
class RandomShapeGenerator {
  private Random rand = new Random();
  public Shape next() {
    switch(rand.nextInt(3)) {
      default:
      case 0: return new Circle();
      case 1: return new Square();
      case 2: return new Triangle();
    }
  }
}

public class Shapes {
  private static Test monitor = new Test();
  private static RandomShapeGenerator gen =
    new RandomShapeGenerator();
  public static void main(String[] args) {
    Shape[] s = new Shape[9];
    // Fill up the array with shapes:
    for(int i = 0; i < s.length; i++)
      s[i] = gen.next();
    // Make polymorphic method calls:
    for(int i = 0; i < s.length; i++)
      s[i].draw();
    monitor.expect(new Object[] {
      new TestExpression("%%
(Circle|Square|Triangle)"
        + "\\.draw\\(\\)", s.length)
    });
  }
} ///:~
```

基类 **Shape** 为继承类定义了一个共用的接口——也就是说，所有的 **Shape** 都有 **draw( )**和 **erase( )**这两个方法。派生类会覆写这两个方法，以提供各自所独有的行为。

RandomShapeGenerator 是一个“工厂”，每次调用它的 **next( )** 方法的时候，它都会随机选取一个 **Shape** 对象，然后返回这个对象的 reference。要注意，上传就发生在 **return** 语句，它拿到的都是 **Circle**，**Square** 或是 **Triangle**，而它返回的都是 **Shape**。因此每次调用 **next( )**的时候，你都没法知道返回值的具体类型，因为传给你的永远是普通的 **Shape** reference。

main( )创建了一个 **Shape** reference 的数组，然后用 **RandomShapeGenerator.next( )**把它填满。这时，你只知道它们

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

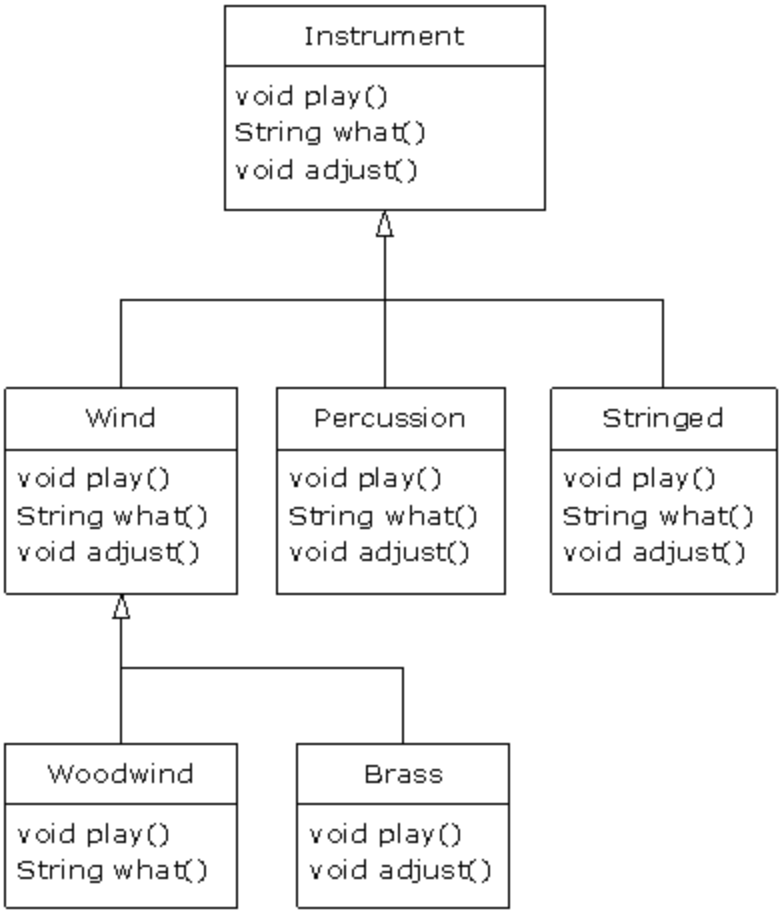
都是 **Shape**，至于更具体的，就不得而知了(编译器也一样)。但是一旦程序运行，当你遍历数组，逐个地调用它们的 **draw( )**方法的时候，你就会发现， **draw( )**的行为魔法般地变成了各个具体类型的正确行为了。

之所以要随机选择形状，是想做得道地一点，让你相信编译器在编译的时候也不知道该选用哪个方法。所有的 **draw( )**都必须使用后绑定。

可扩展性

现在，我们再回到乐器的例子。由于有了多态性，你就可以根据需要，往系统里添加任意多个新类型，而不用担心还要修改 **tune( )**方法了。在一个设计良好的 OOP 程序中，绝大多数方法都会和 **tune( )**一样，只跟基类接口打交道。这种程序是可扩展的，因为你可以通过“让新的数据类型继承通用的基类”的方法，来添加新的功能。而那些与基类接口打交道的方法，根本不需要作修改就能适应新的类。

就拿乐器为例，想想该怎样往基类里加新的方法，并且通过继承产生一些新的类。下面就是关系图：



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

这些新的类都能同原先未作修改的 **tune( )**方法协同工作。即便是在 **tune( )**保存在另一个文件，而 **Instrument** 的接口已经加入了新方法的情况下，它也能无须重新编译而正常工作。下面就是这个关系图的实现：

```
//: c07:music3:Music3.java
// An extensible program.
package c07.music3;
import com.bruceeckel.simpletest.*;
import c07.music.Note;

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

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

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

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

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

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


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

```
public class Music3 {
  private static Test monitor = new Test();
  // Doesn't care about type, so new types
  // added to the system still work right:
  public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
  }
  public static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  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"
    });
  }
} ///:~
```

新的方法是 **what( )**和 **adjust( )**。前者会返回一个描述这个类的 **String**；而后者则会为每件乐器调音。

当你向 **main( )**的 **orchestra** 数组放东西的时候，它们都会被自动地上传到 **Instrument**。

可以看到，**tune( )**方法对周遭所发生的变化一无所知，但是却能正常工作。这正是我们所希望的，多态性应该提供的功能。程序的变动不会影响到它不应该影响的部分。换言之，对程序员来说，多态性是一项非常重要的技术，它能让你“将会变和不会变的东西分隔开来。”

错误：“覆写”private 的方法

你可能会很无辜地尝试去作下面这类事情：

```
//: c07:PrivateOverride.java
// Abstract classes and methods.
import com.bruceeckel.simpletest.*;

public class PrivateOverride {
```


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

```
  private static Test monitor = new Test();
  private void f() {
    System.out.println("private f()");
  }
  public static void main(String[] args) {
    PrivateOverride po = new Derived();
    po.f();
    monitor.expect(new String[] {
      "private f()"
    });
  }
}

class Derived extends PrivateOverride {
  public void f() {
    System.out.println("public f()");
  }
} ///:~
```

可能你预想的输出会是“**public f( )**”，但是 **private** 方法自动就是 **final** 的，而且会对派生类隐藏。因此这里 **Derived** 的 **f( )**是一个全新的方法；甚至连重载都不算，因为 **Derived** 根本看不到基类的 **f( )**。

结论就是，只有非 **private** 的方法才能被覆写。但是你得留意那些看上去像是在覆写 **private** 方法的程序，它们不会产生编译错误，但是很可能会不按你的计划运行。说得再彻底一些，别用基类的 **private** 方法的名字去命名派生类的方法。

抽象类和抽象方法

在这些乐器例子中，**Instrument** 基类的方法都是些“样子货”。如果这些方法真的被调用了，那程序就有问题了。实际上 **Instrument** 的意图是要为所有由它派生出来的类创建一个公共的接口。

要创建这种公共接口的唯一原因就是，各个子类要用它自己的方式来实现这个接口。它定义了一个基本的形式，你可以说这是所有的派生类所共有的。还有一种说法，就是 **Instrument** 是一个“抽象的基类(*abstract base class* 或者简化为抽象类 *abstract class*)”。当你想要通过一个公共的接口来操控一组类的时候，就可以使用抽象类了。通过动态绑定机制，那些符合方法特征的派生类方法将会得到调用。(正如你在上一节看到的，如果方法名同基类的相同，而参数列表不同，那就变成重载了，这大概不是你想要的结果吧。)

如果你有一个像 **Instrument** 这样的 abstract class，那么这种类的对象是没什么意思的。也就是说 **Instrument** 只是用来定义接口，而不是具体实现，因此创建 **Instrument** 对象没有意义，更何况你还可能要禁止用户这么作。要做到这点，可以让 **Instrument** 的方法打印错误信

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

息，但是这样一来就把问题留到运行时了，因此用户端需要进行详尽的测试。更好的办法还是在编译时发现这个问题。

Java 提供了一种被称为“抽象方法(*abstract method*)”的机制来解决这个问题。[32]这是一种尚未完成的方法；这种方法只有声明，没有正文。下面就是抽象方法的声明：

```
abstract void f();
```

包含抽象方法的类被称为“抽象类(*abstract class*)”。如果类包含一个或多个抽象方法，那么这个类就必须被定义成 **abstract** 的。(否则编译器就会报错了。)

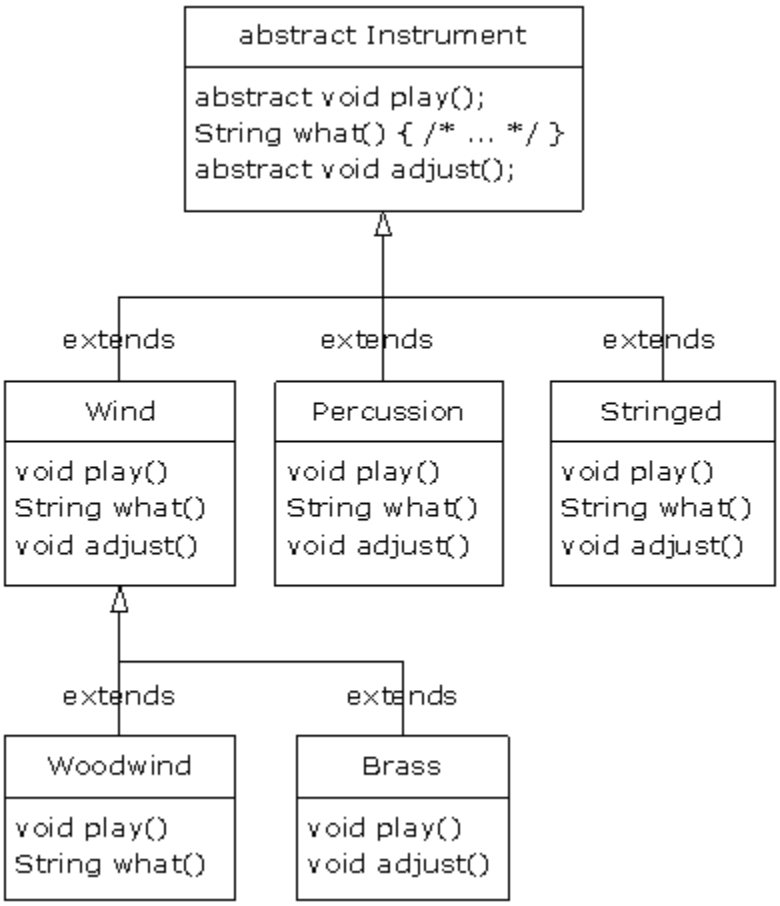
既然抽象类是尚未完成的类，那么如果有人想要创建抽象类的对象的话，编译器又打算怎么做呢？编译器没法安全地创建抽象类对象，所以它会报错。由此，编译器保证了抽象类的纯粹性，而你也不用担心它会被误用了。

如果你继承了抽象类，而且还打算创建这个新类的对象，那你就必须实现基类所定义的全部抽象方法。如果你不这么做(你确实可以选择不这么做)，那么这个继承下来的类就也成了抽象类了，编译器会强制你用 **abstract** 关键词来声明这个类的。

创建一个不包含 **abstract** 方法的 **abstract** 类，是完全可以的。这种技巧可以用于“不必创建 **abstract** 的方法，但是又要禁止别人创建这个类的对象”的场合。

Instrument 类可以很容易的被修改成 **abstract** 类。由于抽象类并不要求它的所有方法都是抽象的，因此只要把几个方法定义成 **abstract** 的就行了。下面就是设计方案：

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



就是用 **abstract** 类和 **abstract** 方法来修改 orchestra (管弦乐团)的例子：

```
//: c07:music4:Music4.java
// Abstract classes and methods.
package c07.music4;
import com.bruceeckel.simpletest.*;
import java.util.*;
import c07.music.Note;

abstract class Instrument {
  private int i; // Storage allocated for each
  public abstract void play(Note n);
  public String what() {
    return "Instrument";
  }
  public abstract void adjust();
}

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

class Percussion extends Instrument {
```


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

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

class Stringed extends 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 Music4 {
  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]);
  }
  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"
    });
```


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

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

可以看到，除了基类之外，别的什么都没改。

由于它明确了类的抽象性，并且告诉用户和编译器该如何使用，因此 **abstract** 的类和方法能帮你解决很多问题。

构造函数与多态性

跟往常一样，构造函数总是与众不同，牵涉到多态性的时候也不例外。尽管构造函数不是多态的(实际上它们都是 **static** 方法，只是声明的时候没有直说)，但是能理解“它在复杂的类系和多态的环境下是如何工作的”仍然十分重要。一旦理解了这个问题，你就能避开很多让人不舒服的纠缠。

构造函数的调用顺序

我们先是在第 4 章简要介绍了构造函数的调用顺序，后来又在第 6 章作了进一步的讲解，但是那时我们还没有介绍多态性。

在创建派生类对象的过程中，基类的构造函数总是先得到调用，这样一级一级的追溯上去，每个基类的构造函数都会被调用。这种做法是很合乎情理的，因为构造函数有一个特殊的任务：它要知道对象是不是被正确地创建了。派生类只能访问它自己的成员，它看不到基类的成员(因为它们通常都是 **private** 的)。只有基类的构造函数才知道怎样初始化它的成员，同时也只有它才有权限进行初始化。因此“把所有的构造函数都调用一遍”就变得非常重要了，否则对象就没法创建了。这就是为什么编译器会强制每个派生类都要调用其基类的构造函数的原因了。如果你不在派生类的构造函数里明确地调用基类的构造函数，那编译器就会悄悄的调用那个默认的构造函数。如果没有默认构造函数，编译器就会报错。(要是类没有构造函数，编译器会自动为你准备一个默认构造函数。)

我们来看看下面这个例子，通过它我们可以了解合成，继承，以及多态性对于对象创建的影响：

```
//: c07:Sandwich.java
// Order of constructor calls.
package c07;
import com.bruceeckel.simpletest.*;

class Meal {
  Meal() { System.out.println("Meal()"); }
}
```


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

```
class Bread {
  Bread() { System.out.println("Bread()"); }
}

class Cheese {
  Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
  Lettuce() { System.out.println("Lettuce()"); }
}

class Lunch extends Meal {
  Lunch() { System.out.println("Lunch()"); }
}

class PortableLunch extends Lunch {
  PortableLunch()
{ System.out.println("PortableLunch()");}
}

public class Sandwich extends PortableLunch {
  private static Test monitor = new Test();
  private Bread b = new Bread();
  private Cheese c = new Cheese();
  private Lettuce l = new Lettuce();
  public Sandwich() {
    System.out.println("Sandwich()");
  }
  public static void main(String[] args) {
    new Sandwich();
    monitor.expect(new String[] {
      "Meal()",
      "Lunch()",
      "PortableLunch()",
      "Bread()",
      "Cheese()",
      "Lettuce()",
      "Sandwich()"
    });
  }
} ///:~
```

这段程序用很多类创建了一个复杂的类，这些类都有构造函数。**Sandwich** 是最重要的类，它有三层继承关系(如果你把隐含的继承 **Object** 也算上的话，实际上是四层)，还有三个成员对象。你可以通过 **main( )**的输出观察到 **Sandwich** 对象的创建过程。也就是说复杂对象的构造函数的调用顺序是这样的：

1. 调用基类的构造函数。这是一个递归过程，因此会先创建继承体系的根，然后是下一级派生类，以此类推，直到最后一个继承类的构造函数。
2. 成员对象按照其声明的顺序进行初始化。
3. 执行继承类的构造函数的正文。

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

构造函数的调用顺序是非常重要的。继承的前提就是你能看到基类，并且还能访问它的 **public** 和 **protected** 成员。也就是说，轮到处理派生类的时候， 所有的基类成员都应该已经准备好了。对于普通方法，构造过程已经结束了，因此对象已经完全建好了。但是对构造函数来说，这还只是假设。为了确保这一点，只能让构造函数先调用基类的构造函数。这样，当你执行派生类构造函数的时候，基类成员就都已经初始化完毕了，可以供你访问了。“能用构造函数访问数据成员”也是“只要有可能，就在定义类的成员对象(也就是用合成的方式放进去的对象)的时候就进行初始化”的原因(比如上述程序中的 **b**，**c**，以及 **l**)。如果你遵循了这种做法，那么当前对象的所有“基类成员(base class members)”，以及它自己的成员对象的初始化就有保证了。但不幸的是，这不是一条万能法则。下一节就会讲到。

继承与清理

即便新类既有合成又有继承，绝大多数情况下，你都无须担心清理的问题；子对象通常都可以交由垃圾回收器去处理。如果真的需要进行清理，那就只能辛苦一点，为新类创建一个 **dispose( )**方法了(我特地选了这个名字，不过你也可以选一个你觉得更好的名字)。而且在继承情况下，如果垃圾回收过程中还要作一些特殊的处理，那你还必须在派生类里覆写基类的 **dispose( )**。当你编写派生类的 **dispose( )**的时候，要记住第一件事就是调用基类的 **dispose( )**，这点非常重要。因为不这样做的话，基类就不会得到清理。下面的例程演示了这点：

```
//: c07:Frog.java
// Cleanup and inheritance.
import com.bruceeckel.simpletest.*;

class Characteristic {
  private String s;
  Characteristic(String s) {
    this.s = s;
    System.out.println("Creating Characteristic " +
s);
  }
  protected void dispose() {
    System.out.println("finalizing Characteristic "
+ s);
  }
}

class Description {
  private String s;
  Description(String s) {
    this.s = s;
    System.out.println("Creating Description " + s);
  }
  protected void dispose() {
    System.out.println("finalizing Description " +
s);
  }
}
```


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

```
class LivingCreature {
  private Characteristic p = new Characteristic("is
alive");
  private Description t =
    new Description("Basic Living Creature");
  LivingCreature() {
    System.out.println("LivingCreature()");
  }
  protected void dispose() {
    System.out.println("LivingCreature dispose");
    t.dispose();
    p.dispose();
  }
}

class Animal extends LivingCreature {
  private Characteristic p= new Characteristic("has
heart");
  private Description t =
    new Description("Animal not Vegetable");
  Animal() {
    System.out.println("Animal()");
  }
  protected void dispose() {
    System.out.println("Animal dispose");
    t.dispose();
    p.dispose();
    super.dispose();
  }
}

class Amphibian extends Animal {
  private Characteristic p =
    new Characteristic("can live in water");
  private Description t =
    new Description("Both water and land");
  Amphibian() {
    System.out.println("Amphibian()");
  }
  protected void dispose() {
    System.out.println("Amphibian dispose");
    t.dispose();
    p.dispose();
    super.dispose();
  }
}

public class Frog extends Amphibian {
  private static Test monitor = new Test();
  private Characteristic p = new
Characteristic("Croaks");
  private Description t = new Description("Eats
Bugs");
  public Frog() {
    System.out.println("Frog()");
  }
  protected void dispose() {
    System.out.println("Frog dispose");
    t.dispose();
```


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

```
    p.dispose();
    super.dispose();
  }
  public static void main(String[] args) {
    Frog frog = new Frog();
    System.out.println("Bye!");
    frog.dispose();
    monitor.expect(new String[] {
      "Creating Characteristic is alive",
      "Creating Description Basic Living Creature",
      "LivingCreature()",
      "Creating Characteristic has heart",
      "Creating Description Animal not Vegetable",
      "Animal()",
      "Creating Characteristic can live in water",
      "Creating Description Both water and land",
      "Amphibian()",
      "Creating Characteristic Croaks",
      "Creating Description Eats Bugs",
      "Frog()",
      "Bye!",
      "Frog dispose",
      "finalizing Description Eats Bugs",
      "finalizing Characteristic Croaks",
      "Amphibian dispose",
      "finalizing Description Both water and land",
      "finalizing Characteristic can live in water",
      "Animal dispose",
      "finalizing Description Animal not Vegetable",
      "finalizing Characteristic has heart",
      "LivingCreature dispose",
      "finalizing Description Basic Living Creature",
      "finalizing Characteristic is alive"
    });
  }
} ///:~
```

这个继承体系中的每个类都有一个 **Characteristic** 和一个 **Description** 类型的成员对象。它们也应该得到清理。对象与对象之间有可能会有依赖关系，因此清理的顺序应该与初始化的顺序相反。对数据成员而言，这就是说它们的清理顺序应该与声明的顺序相反(因为数据的初始化是按照声明的顺序进行的)。对基类而言(它采用了 C++拆构函数的形式)，你应该先进行派生类的清理，再进行基类的清理。这是因为派生类的清理可能需要调用某些基类的方法，也就是说要留着基类，因此它不能过早地被清除掉。你可以从程序的输出看出，**Frog** 对象各部分的清理顺序，正好与它们创建的顺序相反。

这个例程表明，尽管你不会老是进行清理，但是真的要做的时候，还是要非常小心的。

多态方法在构造函数中的行为

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

构造函数的调用顺序也带来了一个有趣的难题。如果构造函数调用了一个动态绑定的方法，而这个方法又属于那个正在创建中的对象，那它会产生什么样的效果呢？如果是普通方法，你猜也可以猜到它会怎么做：由于不知道这个对象应该算是基类还是派生类的，因此动态绑定会在运行时进行解析。出于一致性的考虑，你可能会认为构造函数也应该这么作。

事实并非完全如此。如果你在构造函数里面调用了动态绑定的方法，那么它会使用那个覆写后的版本。但是这个“效果”会有些出人意料，因此会成为一些不易排查的 bug 的藏身之处。

从理论上讲，构造函数的任务就是创建对象(这可不是什么轻而易举的事)。从构造函数的角度来看，对象可能只创建了一半——你只知道基类对象已经初始化了，但是你还不知道它会派生出什么类。但是动态绑定的方法调用，会从“外面”把手伸进“类系(inheritance hierarchy)”。它调用的是派生类的方法。如果你在构造函数里面这么做的话，你就可能调用了一个“会访问尚未初始化的成员”的方法了——这注定会出问题。

你可以从下面这个例子理解这个问题：

```
//: c07:PolyConstructors.java
// Constructors and polymorphism
// don't produce what you might expect.
import com.bruceeckel.simpletest.*;

abstract class Glyph {
  abstract void draw();
  Glyph() {
    System.out.println("Glyph() before draw()");
    draw();
    System.out.println("Glyph() after draw()");
  }
}

class RoundGlyph extends Glyph {
  private int radius = 1;
  RoundGlyph(int r) {
    radius = r;
    System.out.println(
      "RoundGlyph.RoundGlyph(), radius = " + radius);
  }
  void draw() {
    System.out.println(
      "RoundGlyph.draw(), radius = " + radius);
  }
}

public class PolyConstructors {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    new RoundGlyph(5);
    monitor.expect(new String[] {
      "Glyph() before draw()",
      "RoundGlyph.draw(), radius = 0",
      "Glyph() after draw()",
```


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

```
        "RoundGlyph.RoundGlyph(), radius = 5"
      });
    }
} ///:~
```

Glyph 的 **draw( )**方法是 **abstract** 的，所以这种设计就是要叫别人来覆写的。实际上，你必须在 **RoundGlyph** 里覆写 **draw( )**。**Glyph** 的构造函数调用了 **draw( )**，而这个调用最后落到了 **RoundGlyph.draw( )**，看来一切都正常。但是当你查看输出的时候，你就会发现，**Glyph** 的构造函数调用 **draw( )**的时候，**radius** 的值还没有被设成缺省的初始值 1。它还是 0。如果是真的编程的话，这可能会是屏幕上画的一个点，甚至是什么都不画，而你呢，会在那里瞪大了眼睛，费力地排查到底那里出了错。

在上面章节介绍的初始化顺序并不完整，而缺失的部分才是这个谜团的关键。真正的初始化过程是这样的：

1. 在进行其它工作之前，分配给这个对象的内存会先被初始化为两进制的零。
2. 正如前面一直在所说的，先调用基类的构造函数。这时会调用被覆写的 **draw( )**方法(是的，在调用 **RoundGlyph** 的构造函数之前调用)，这时它发现，由于受第一步的影响，**radius** 的值还是零。
3. 数据成员按照它们声明的顺序进行初始化。
4. 调用派生类的构造函数的正文。

这样做有个好处，就是它不会留一堆垃圾，最起码把所有的东西都初始化为零了(无论是哪种类型的，都是零)。这其中也包括用合成嵌进去的对象，这些 reference 都被设成了 **null**。所以如果你忘了对 reference 进行初始化，运行的时候就会抛出异常。所有的东西都是零，这样查看输出的时候多少会有点线索。

但是另一方面，你可能会对程序运行的结果大吃一惊。你的设计完美无缺，但是程序运行的结果就是那样错得离谱，而且编译器还不报错。(在这种情况下，C++的编译器会有一些比较理性的反映。)这类 bug 很容易会被忽略，而且要花很长的时间发现。

结论就是，一个好的构造函数应该，“用最少的工作量把对象的状态设置好，而且要尽可能地避免去调用方法。”构造函数唯一能安全调用的方法，就是基类的 **final** 方法。(这一条也适用 **private** 方法，因为它自动就是 **final** 的。)它们不会被覆写，因此也不会产生这种意外的行为。

用继承来进行设计

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

一旦理解了多态性，你就会觉得所有东西应该都是继承下来的，因为多态性实在是太聪明了。但是这样做会加重设计的负担；实际上，如果你一遇到“要用已有的类来创建新类”的情况就想到要用继承的话，事情就会毫无必要地变得复杂起来了。

较好的办法还是先考虑合成，特别是当你不知道该继承哪个类的时候。合成并不强求你把设计搞成一个类系。此外它还更灵活，因为使用合成的时候，你可以动态地选择成员的类型(以及它们的行为)，而使用继承的话，就得在编译时指明对象的确切类型。下面这段程序就演示了这一点：

```
//: c07:Transmogrify.java
// Dynamically changing the behavior of an object
// via composition (the "State" design pattern).
import com.bruceeckel.simpletest.*;

abstract class Actor {
  public abstract void act();
}

class HappyActor extends Actor {
  public void act() {
    System.out.println("HappyActor");
  }
}

class SadActor extends Actor {
  public void act() {
    System.out.println("SadActor");
  }
}

class Stage {
  private Actor actor = new HappyActor();
  public void change() { actor = new SadActor(); }
  public void performPlay() { actor.act(); }
}

public class Transmogrify {
  private static Test monitor = new Test();
  public static void main(String[] args) {
    Stage stage = new Stage();
    stage.performPlay();
    stage.change();
    stage.performPlay();
    monitor.expect(new String[] {
      "HappyActor",
      "SadActor"
    });
  }
} ///:~
```

Stage 对象包含了一个 **Actor** 的 reference，而这个 reference 又被初始化为 **HappyActor**。也就是说 **performPlay( )**会有一些特殊的

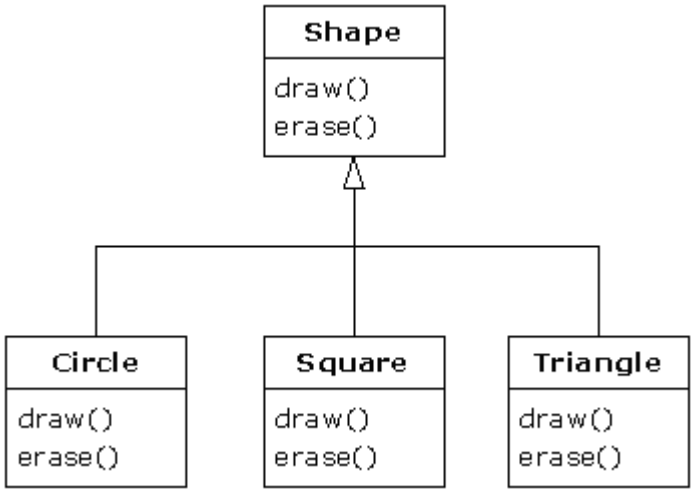
www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

行为。但是程序运行时 **Actor** 的 reference 可以连到另一个对象上，因此可以用 **SadActor** 对象来替换它，于是 **performPlay( )**的行为就发生了变化。这样你就在运行时获得了高度的灵活性。(这也被称为“状态模式(*State Pattern*)”。参见 *www.BruceEckel.com* 所刊载的 *Thinking in Pattern* (Java 版))。反观继承，它不能让你在运行时继承不同的类；这个问题在编译的时候就已经定下来了。

有一条一般准则“使用继承来表示行为的不同，而用成员数据来表示不同的状态。”上述例程同时体现这两者；两个派生类用来表示 **act( )**方法的不同，而 **Stage** 则使用合成来表示状态的变化。在这种情况下，状态的不同会导致行为的不同。

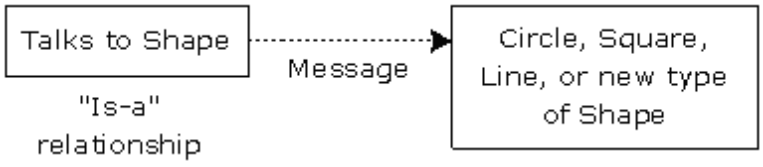
纯继承与扩展

看来，研究继承的最好方式，还是用“纯继承”的方式创建一个类系。也就是说，派生类仅覆写基类或 **interface** 里有的方法。就像下面这张图：



由于类是由其接口所决定的，因此这种关系被称为纯的“是”关系。继承保证了所有派生类都至少拥有基类的接口。而如果你采纳了这张图，那么继承类的接口就不会比基类的更大。

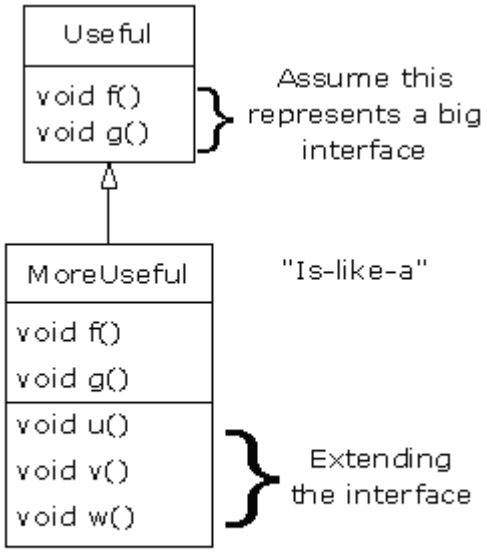
可以把这想像成“完全替代(*pure substitution*)”，因为完全可以用派生类的对象来替换基类的对象，而且你这样做的时候根本不需要任何子类的信息：



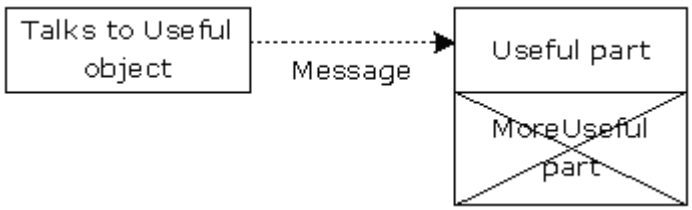
www.wgqqh.com/shhgs/tij.html
email:shhgs@sohu.com

也就是说，由于有着相同的接口，基类可以接受任何发送给派生类的消息。 你所要做的，只是将派生类的对象上传，然后就不再需要知道这个对象是什么类型的了。所有的东西都交由多态性去处理。

看到这里，你会觉得纯的“是”关系才是唯一合理的方案，而其它方案都不过是一些思路混乱，并且前后矛盾的东西。这也是一个误区。如果你用这种思路考虑问题的话，很快就会发现事实并非如此，对某些问题而言，扩展接口(看到了吗，**extends** 关键词就是在鼓励你去这么做)是一个完美的解决方案。这种关系可以用“像是(is-like-a)”这个术语来表示，因为派生类“像”基类——它有着相同的基本接口——但是它还有一些其它特性，需要实现一些额外的方法：



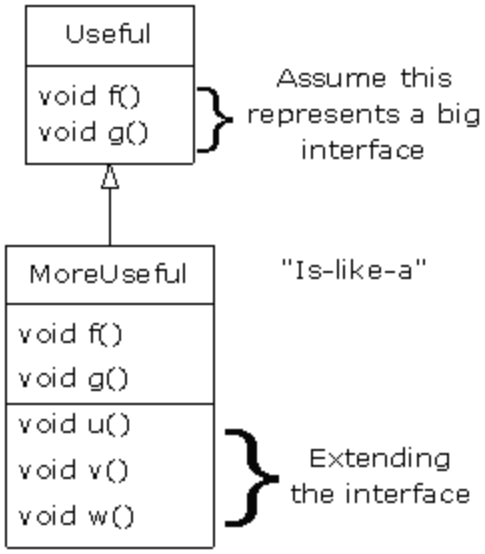
虽然这也是一种有用并且合理的方式(根据情况)，但是它有一个缺点。不能通过基类访问派生类的扩展接口，所以上传之后，你就无法调用新的方法了：



如果你不使用上传，那么什么问题都没有。但是通常情况下，你都会碰到“要重新发现这个对象的确切类型”的情况，这样你才能使用那种类型的扩展方法。下面一节会告诉你该怎么做。

下传与运行时的类型鉴别

由于“上传 *upcast*”(沿着继承关系向上移)之后，类的具体信息丢了，因此“用『下传(*downcast*)』——也就是沿着继承关系重新向下移——来提取类型的信息”就成了顺理成章的事了。你知道上传总是安全的；基类的接口不可能比派生类的更大。因此它肯定能收到那些通过基类接口发送的消息。但是碰到下传的时候，你就不能肯定“这个形状是不是圆了”(只是举个例子)。它可以是一个三角型，一个矩形或是其它什么形状。



要想解决这个问题，必须要有办法能够确保下传是正确的，这样你就不会把对象误传给另一个类型了，于是也不会向它发送什么它不能接受的消息了。这是相当不安全的。

某些语言(像 C++)需要经过使用特别处理，才能安全地进行下传。但是 Java 类型传递都要经过检查！所以，尽管看上去只是用一对括号作了些普通的的类型转换，但是运行的时候，系统会对这些转换作检查，以确保它确实是你想要转换的类型。如果不是，你就会得到一个 **ClassCastException**。这种运行时的类型检查被称为“运行时的类型鉴别(*run-time type identification* 缩写为 RTTI)”。下面的例程演示了 RTTI 的行为：

```
//: c07:RTTI.java
// Downcasting & Run-Time Type Identification (RTTI).
// {ThrowsException}

class Useful {
  public void f() {}
  public void g() {}
}

class MoreUseful extends Useful {
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
```


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

```
}

public class RTTI {
  public static void main(String[] args) {
    Useful[] x = {
      new Useful(),
      new MoreUseful()
    };
    x[0].f();
    x[1].g();
    // Compile time: method not found in Useful:
    //! x[1].u();
    ((MoreUseful)x[1]).u(); // Downcast/RTTI
    ((MoreUseful)x[0]).u(); // Exception thrown
  }
} ///:~
```

正如图表所示，**MoreUseful** 扩展了 **Useful** 的接口。但是由于它是继承的，因此可以将它上传给 **Useful**。可以看到，当 **main( )**对数组 **x** 进行初始化的时候，就进行了上传。由于数组中的两个对象都是 **Useful** 的，因此你可以向它们发送 **f( )**和 **g( )**消息，但是如果你调用了 **u( )**(只有 **MoreUseful** 才有)，编译的时候就会报错。

如果你想访问 **MoreUseful** 对象的扩展接口，你就得先下传。如果类型正确，这个操作就会成功。否则，你就会得到 **ClassCastException**。你不必为这个异常编写什么特殊的代码，因为它表示这是程序员犯的错误，而这种错误可能发生在程序的任意地方。

RTTI 要比直接转换更复杂。举例来说，你可以在下传之前先看一看“这个对象是哪种类型的”。我们会在第 10 章，用整章的篇幅研究 Java 的运行时类型鉴别。

总结

多态性的意思是“不同的形式”。在面向对象的编程中，你会有“一张相同的脸”(基类的公共接口)和很多“不同的使用这张脸的方式”：各个版本的动态绑定方法。

你在本章也看到了，如果不理解数据抽象和继承的话，是根本不可能理解多态性的，更不用说创建多态性的例子了。多态性是一种不能孤立的看待的特性(不像 **switch** 语句)，相反只有放在类关系的“大背景”下，它才有用武之地。人们通常会把它同 Java 的那些非面向对象的特性相混淆，比如方法的重载，它常常会被当作面向对象的特性介绍给大家。千万别上当：不是后绑定的，就不是多态性。

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

要想在编程中有效地使用多态性，以及面向对象的技术，那你就必须扩展你的编程视野，不能只关注单个类的数据成员和消息，而是要去理解类与类之间的共同性，以及它们之间的关系。虽然这个要求很高，但是这种努力是值得的，因为它能加速程序的开发，改善代码的逻辑组织，使得程序更易于扩展，同时维护代码也变得更方便了。

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. 为 **Shapes.java** 的基类加一个能打印消息的方法，但是别在派生类里覆写这个方法。预测一下会有什么结果。现在，再在一个派生类里覆写这个类，但是别在其它派生类里覆写，看看结果会怎样。最后，在所有的派生类里覆写这个方法。
2. 向 **Shape.java** 添加一个新的类，然后用 **main( )**检查一下，看看是不是像和旧类一样，多态性在新类上也一样能正常工作。
3. 修改 **Music3.java**，让 **what( )**成为 **Object** 的 **toString( )**方法。试着用 **System.out.println( )**打印 **Instrument** 对象(不要用类型转换)。
4. 往 **Music3.java** 里面加一种新的 **Instrument**，看看多态性是不是也会对新类型正常工作。
5. 修改 **Music3.java**，让它以 **Shapes.java** 的方式随机创建 **Instrument** 对象。
6. 创建一个 **Rodent**(啮齿动物)类系：**Mouse**，**Gerbil**，**Hamster**，等等。在基类里定义所有 **Rodent** 所共有的方法，然后在派生类里根据 **Rodent** 的具体类型覆写这些方法，以提供不同的行为。创建一个 **Rodent** 的数组，用各种具体的 **Rodent** 填满这个数组，然后调用基类的方法，看看程序运行的结果。
7. 修改练习 6，将 **Rodent** 改写成 **abstract** 的类。只要有可能，就把 Rodent 的方法作成抽象方法。
8. 创建一个不含任何 **abstract** 方法的 **abstract** 类，然后验证一下，你是不是不能创建这个类的实例。
9. 往 **Sandwich.java** 里面添加 **Pickle**(酸黄瓜)类。
10. 修改练习 6，用它来演示基类与派生类的初始化顺序。再往基类和派生类里同时加入成员对象，然后看看它们在创建过程中的初始化顺序。
11. 创建一个有两个方法的基类。用第一个方法调用第二个方法。继承这个类，并覆写第二个方法。创建第二个类的对象，并上传到其基类，然后调用第一个方法。说说看，会有什么结果。
12. 创建一个带 **abstract print( )**方法的基类，然后在派生类里覆写这个方法。覆写后的方法要能打印在派生类里定义的 **int** 变量的值。定义这个变量的时候，赋给它一个非零的值。在基类的构造函数里，调用这

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

个方法。用 **main( )**创建一个派生类的对象，然后调用 **print( )**方法。解释一下为什么会有这个结果。

13. 根据 **Transmogrify.java** 创建一个 **Starship** 类，这个类里要包含一个 **AlertStatus** 的 reference，而这个 **AlertStatus** 要能表示三种不同的状态。写一个切换状态的方法。
14. 创建一个不带方法的 **abstract** 类。继承这个类，并且添加一个方法。再创建一个会将基类的 reference 下传到派生类，并且调用这个方法的 **static** 方法。用 **main( )**检验一下，看看是不是能正常工作。然后在基类里把这个方法声明成 **abstract** 的，这样就不需要下传了。

---

[32]对于 C++的程序员来说，这就是 C++的“纯虚函数(*pure virtual function*)”。

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