Java语言程序设计（第2版）

第 4 章　接口与多态

郑莉

导学

<4.0>(讲课提示，不显示）

目录

接口（文字显示）

（以下文字不显示）

- 就像“纯”的抽象类，只有设计、没有实现；
- 可以同时继承多个设计；
- 可以在没有继承、组合关系的类之间抽象出共同接口；
- 可以隐藏实现细节。

类型转换（文字显示）

（以下文字不显示）

- 继承超类和实现接口的目的不仅在于重用设计和实现，而且可以规定不同类的对象有统一的对外服务接口。
- 从提供同样服务接口的角度
  - 子类对象可以充当超类的对象；
  - 实现接口类的对象可以通过接口类型的引用访问。

多态性（文字显示）

- （以下文字不显示）
- 超类和接口中设计的对外服务功能，在不同的子类或实现接口的类中可以有不同的实现，但是可以通过超类的引用或者接口的引用访问。

接口

<4.1>(讲课提示，不显示)

接口

- 可以看做是一个“纯”抽象类，它只提供一种形式，并不提供实现

接口

- 接口中可以规定方法的原型：方法名、参数列表以及返回类型，但不规定方法主体；
- 也可以包含基本数据类型的数据成员，但它们都默认为static和final。

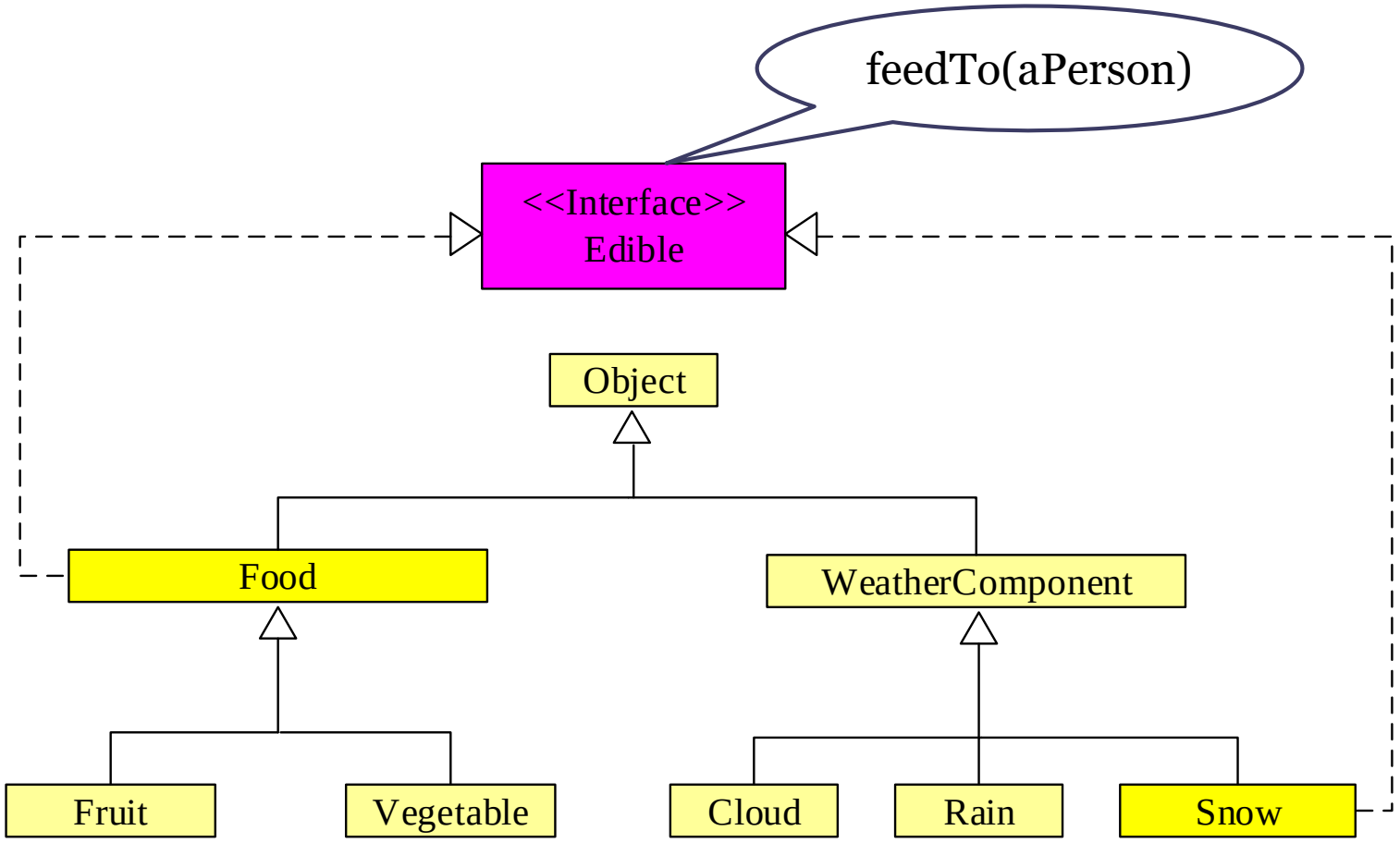
接口的作用

- 是面向对象的一个重要机制

接口的作用

- 继承多个设计。
- 建立类和类之间的“协议”
  - 将类根据其实现的功能分组用接口代表，而不必顾虑它所在的类继承层次；这样可以最大限度地利用动态绑定，隐藏实现细节；
  - 实现不同类之间的常量共享。

接口允许我们在看起来不相干的对象之间定义共同行为（此行文字不显示）



接口的语法

- 声明格式为

```
[接口修饰符] interface 接口名称 [extends 父接口名]{
    ...//方法的原型声明或静态常量
}
```

- 接口的数据成员一定要有初值，且此值将不能再更改，可以省略final关键字。
- 接口中的方法必须是“抽象方法”，不能有方法体，可以省略public及abstract关键字。

例：接口声明

- 声明一个接口Shape2D，包括π和计算面积的方法原型

```
interface Shape2D{                  //声明Shape2D接口
    final double pi=3.14;           //数据成员一定要初始化
    public abstract double area();  //抽象方法
}
```

- 在接口的声明中，允许省略一些关键字，也可声明如下

```
interface Shape2D{
   double pi=3.14;
   double area();
}
```

实现接口

- 不能用new运算符直接产生接口对象；

实现接口

- 利用接口设计类的过程，称为接口的实现，使用implements关键字，语法如下：

```
public class 类名称 implements 接口名称 {
        //在类体中实现接口的方法
        //本类声明的更多变量和方法
}
```

- 注意：
  - 必须实现接口中的所有方法；
  - 来自接口的方法必须声明成public。

例：实现接口Shape2D

```
class Circle implements Shape2D
{
    double radius;
    public Circle(double r)
    {
        radius=r;
    }
    public double area()
   {
        return (pi * radius * radius);
   }
}
```

```
class Rectangle implements Shape2D
{
     int width,height;
     public Rectangle(int w,int h)
     {
          width=w;
          height=h;
      }
       public double area()
       {
          return (width * height);
        }
   }
```

例：实现接口Shape2D

- 测试类

```
public class InterfaceTester {
    public static void main(String  args[]){
        Rectangle rect=new Rectangle(5,6);
        System.out.println("Area of rect = " + rect.area());
        Circle cir=new Circle(2.0);
        System.out.println("Area of cir = " + cir.area());
    }
}
```

- 运行结果

```
Area of rect = 30.0
Area of cir = 12.56
```

例：接口类型的引用变量

- 声明接口类型的变量，并用它来访问对象

```
public class VariableTester {
    public static void main(String []args)
    {
      Shape2D var1,var2;
      var1=new  Rectangle(5,6);
      System.out.println("Area of var1 = " + var1.area());
      var2=new Circle(2.0);
      System.out.println("Area of var2 = " + var2.area());
    }
}
```

- 运行结果

```
Area of var1 = 30.0
Area of var2 = 12.56
```

实现多个接口的语法

实现多个接口的语法

- Java不允许一个类有多个超类（这一行不显示）
- 一个类可以实现多个接口，通过这种机制可实现对设计的多重继承。
- 实现多个接口的语法如下

```
[类修饰符] class  类名称  implements 接口1, 接口2,  ...
{
…… 
}
```

例：通过实现接口达到多重继乘

- 声明Circle类实现接口Shape2D和Color
  - Shape2D具有常量pi与area()方法，用来计算面积；
  - Color则具有setColor方法，可用来赋值颜色；
  - 通过实现这两个接口，Circle类得以同时拥有这两个接口的成员，达到了对设计进行多重继承的目的。

```
interface Shape2D{          //声明Shape2D接口
  final double pi=3.14;     //数据成员一定要初始化
  public abstract double area();    //抽象方法
}
interface Color{
    void setColor(String str);  //抽象方法
}
```

例：通过实现接口达到多重继乘

```
class Circle implements Shape2D, Color   // 实现Circle类
{
   double radius;
   String color;
   public Circle(double r)              //构造方法
   {
      radius=r;
   }
   public double area()                 //定义area()的处理方式
   {
      return (pi*radius*radius);
   }
   public void setColor(String str)  //定义setColor()的处理方式
   {
      color=str;
      System.out.println("color="+color);
   }
}
```

例：通过实现接口达到多重继承

- 测试类

```
public class MultiInterfaceTester{
  public static void main(String args[]) {
     Circle cir;
     cir=new Circle(2.0);
     cir.setColor("blue");
     System.out.println("Area = " + cir.area());
  }
}
```

- 输出结果

```
color=blue
Area = 12.56
```

接口的扩展

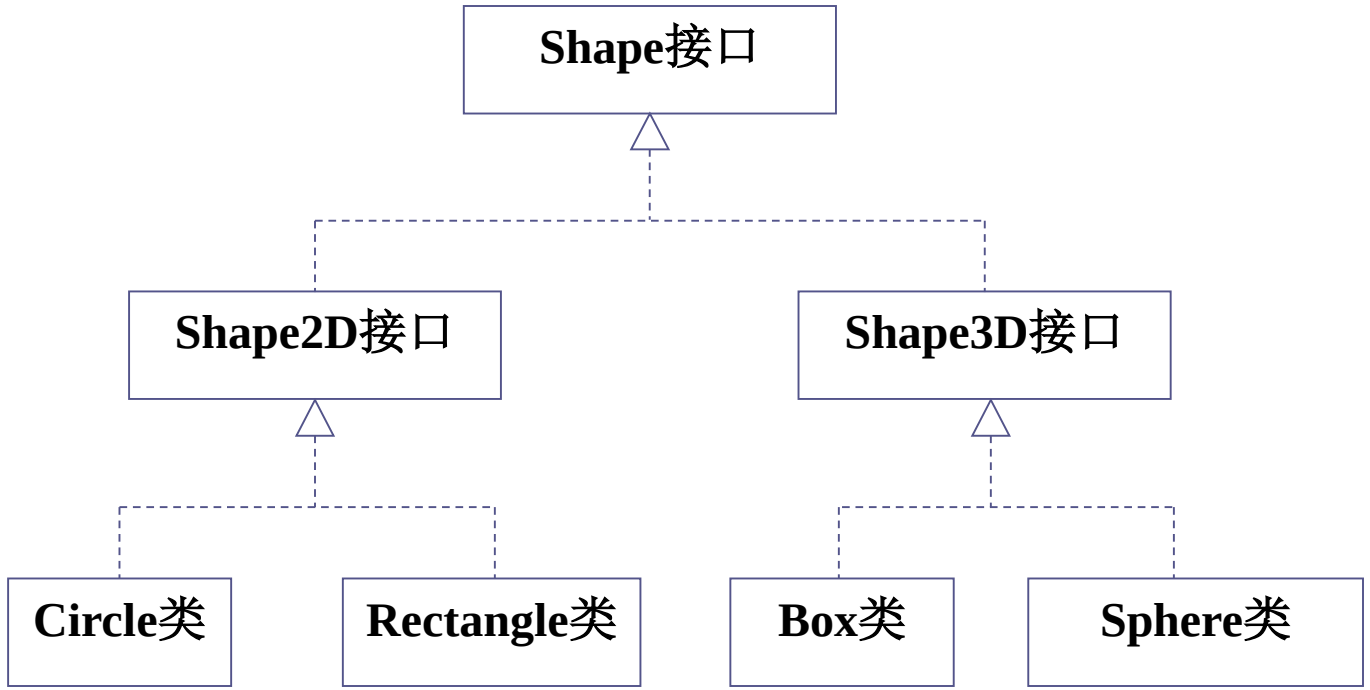
- 接口可通过扩展的技术派生出新的接口
  - 原来的接口称为超接口(super interface)；
  - 派生出的接口称为子接口(sub interface)。

接口的扩展

- 实现一个接口的类也必须实现其超接口。
- 接口扩展的语法

```
interface 子接口的名称 extends 超口的名称1，超接口的名称2，…
{
  …… 
}
```

- Shape是超接口，Shape2D与Shape3D是其子接口。Circle类及Rectangle类实现接口Shape2D，而Box类及Sphere类实现接口Shape3D （此行文字不显示）



例：接口的扩展

- 部分代码如下

```
// 声明Shape接口
interface Shape{
  double pi=3.14;
  void setColor(String str);
}
//声明Shape2D接口扩展了Shape接口
interface Shape2D extends Shape {
    double area();
}
```

例：接口的扩展

```
class Circle implements Shape2D {
    double radius;
    String color;
    public Circle(double r) {   radius=r;  }
    public double area() {
       return (pi*radius*radius);
     }
    public void setColor(String str){
       color=str;
       System.out.println("color="+color);
    }
}
public class ExtendsInterfaceTester{   //测试类
  public static void main(String []args) {
     Circle cir;
     cir=new Circle(2.0);
     cir.setColor("blue");
     System.out.println("Area = " + cir.area());
  }
}
```

例：接口的扩展

- 运行结果

  color=blue

  Area = 12.56

- 说明
  - 首先声明了父接口Shape，然后声明其子接口Shape2D；
  - 之后声明类Circle实现Shape2D子接口，因而在此类内必须明确定义setColor()与area()方法的处理方式；
  - 最后在主类中我们声明了Circle类型的变量cir并创建新的对象，最后通过cir对象调用setColor()与area()方法。

结语

- 这一节介绍了接口的概念和作用、声明和实现接口的语法，以及接口的扩展

类型转换

<4.2.1>——<4.2.2>(讲课提示，不显示)

类型转换的概念

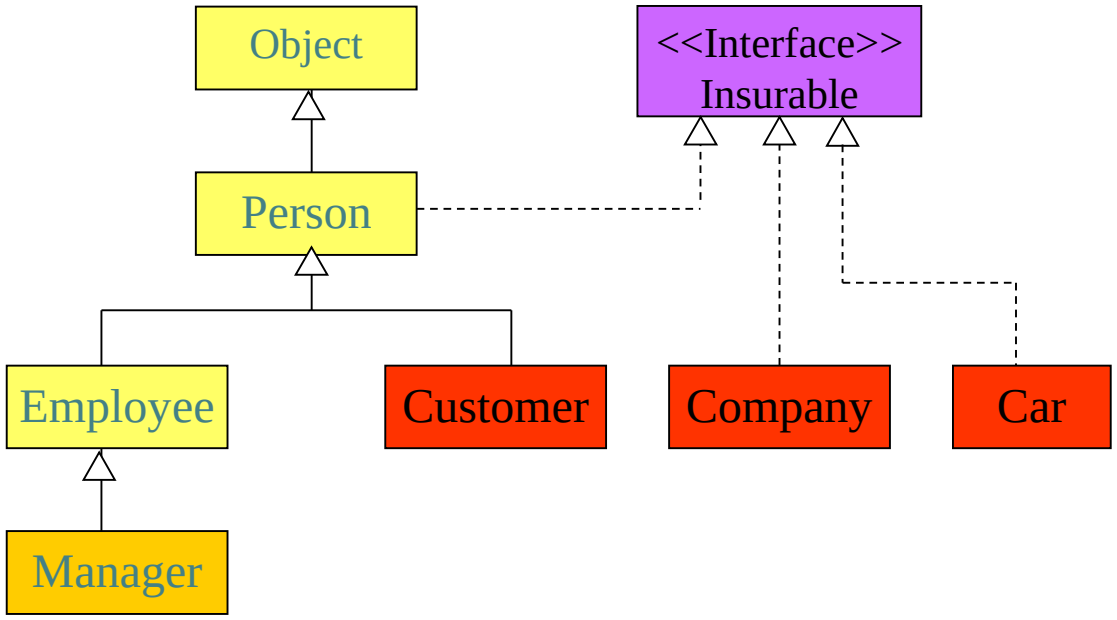
- 又称为塑型(type-casting)。
- 转换方式
  - 隐式的类型转换；
  - 显式的类型转换。
- 转换方向
  - 向上转型；
  - 向下转型。

类型转换规则

- 基本类型之间的转换
  - 将值从一种类型转换成另一种类型。
- 引用变量的类型转换
  - 将引用转换为另一类型的引用，并不改变对象本身的类型。
  - 只能被转为
    - 任何一个（直接或间接）超类的类型（向上转型）；
    - 对象所属的类（或其超类）实现的一个接口（向上转型）；
    - 被转为引用指向的对象的类型（唯一可以向下转型的情况）。
- 当一个引用被转为其超类引用后，通过他能够访问的只有在超类中声明过的方法。

类型转换举例

- Manager对象
  - 可以被塑型为Employee、Person、Object或Insurable，
  - 不能被塑型为Customer、Company或Car



隐式类型转换

隐式类型转换

- 基本数据类型
  - 可以转换的类型之间，存储容量低的自动向存储容量高的类型转换。
- 引用变量
  - 被转成更一般的类，例如：

```
Employee  emp;
emp = new Manager(); //将Manager类型的对象直接赋给
                     //Employee类的引用变量，系统会
                     //自动将Manage对象塑型为Employee类
```

  - 被塑型为对象所属类实现的接口类型，例如：

```
Car  jetta = new Car();
Insurable  item = jetta;
```

显式类型转换

- 基本数据类型

```
(int)871.34354;    // 结果为 871
(char)65;          // 结果为‘A’
(long)453;         // 结果为453L
```

- 引用变量

```
Employee  emp;
Manager man;
emp = new Manager();
man = (Manager)emp; //将emp显式转换为它指向的对象的类型
```

类型转换的主要应用场合

- 赋值转换
- 方法调用转换
- 算术表达式转换
- 字符串转换

类型转换的主要应用场合

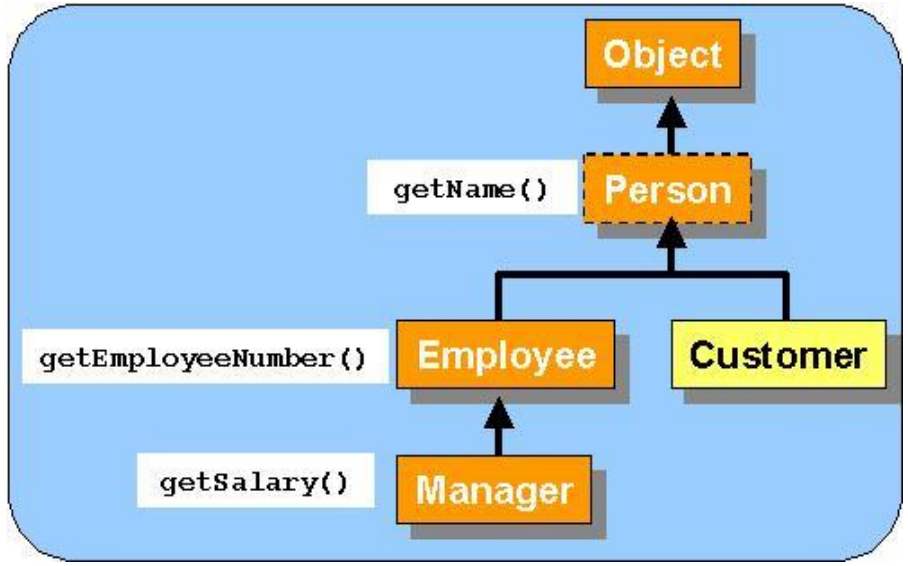
- 赋值转换
  - 赋值运算符右边的表达式或对象类型转换为左边的类型；
- 方法调用转换
  - 实参的类型转换为形参的类型；
- 算术表达式转换
  - 算数混合运算时，不同类型的操作数转换为相同的类型再进行运算；
- 字符串转换
  - 字符串连接运算时，如果一个操作数为字符串，另一个操作数为其他类型，则会自动将其他类型转换为字符串。

类型转换的应用举例

类型转换的应用举例

（以下文字不显示，只显示图，图重画一下不要蓝色底色）

- 当Manager对象被塑型为Employee之后，它只能接收getName()及getEmployeeNumber()方法（都是在其直接或间接超类中声明过的方法），不能接收getSalary()方法（ Manager类中声明的方法）；
- 将其塑型为本来的类型后，又能接收getSalary()方法了。



方法的查找

<4.2.3>(讲课提示，不显示)

方法的查找

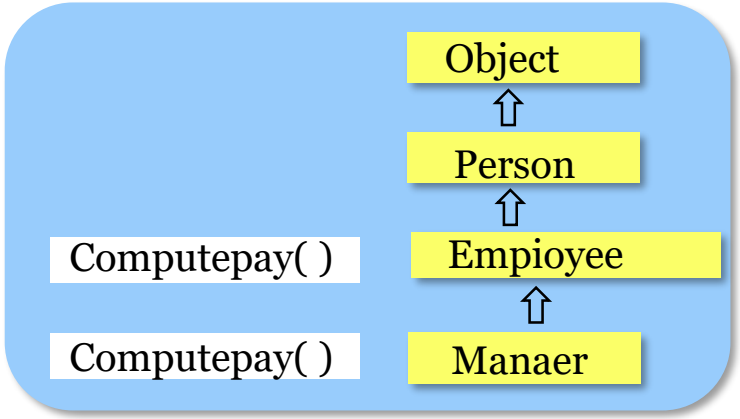
- 如果在塑型前和塑型后的类中都提供了相同原型的方法，如果将此方法发送给塑型后的对象，那么系统将会调用哪一个类中的方法？

方法的查找

- 实例方法的查找
- 类方法的查找

实例方法的查找

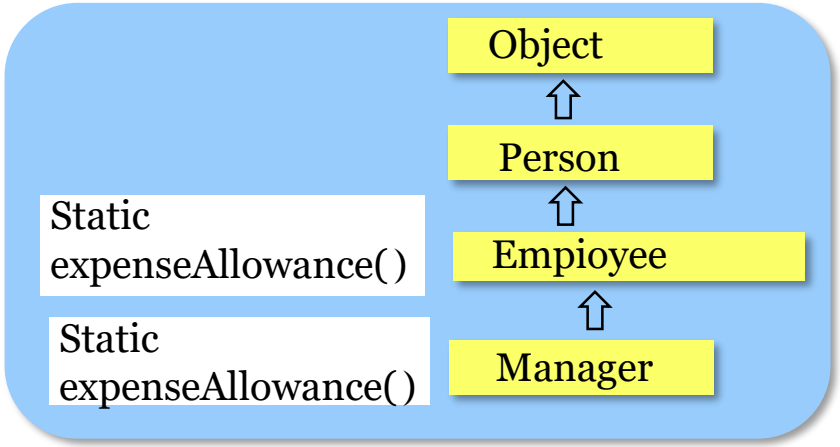
- 从对象创建时的类开始，沿类层次向上查找。




```
Manager  man = new Manager();
Employee  emp1 = new Employee();
Employee  emp2 = (Employee)man;
emp1.computePay();       // 调用Employee类中的computePay()方法
man.computePay();        // 调用Manager类中的computePay()方法
emp2.computePay();       // 调用Manager类中的computePay()方法
```

类方法的查找

- 总是在引用变量声明时所属的类中进行查找。




```
Manager  man = new Manager();
Employee emp1 = new Employee();
Employee emp2 = (Employee)man;
man.expenseAllowance();          //in Manager
emp1.expenseAllowance();         //in Employee
emp2.expenseAllowance();         //in Employee!!!
```

多态的概念

<4.3>(讲课提示，不显示)

多态的概念

- 多态是指不同类型的对象可以响应相同的消息

多态的概念

- 超类对象和从相同的超类派生出来的多个子类的对象，可被当作同一种类型的对象对待；
- 实现同一接口不同类型的对象，可被当作同一种类型的对象对待；
- 可向这些不同的类型对象发送同样的消息，由于多态性，这些不同类的对象响应同一消息时的行为可以有所差别。
- 例如
  - 所有的Object类的对象都响应toString()方法
  - 所有的BankAccount类的对象都响应deposit()方法

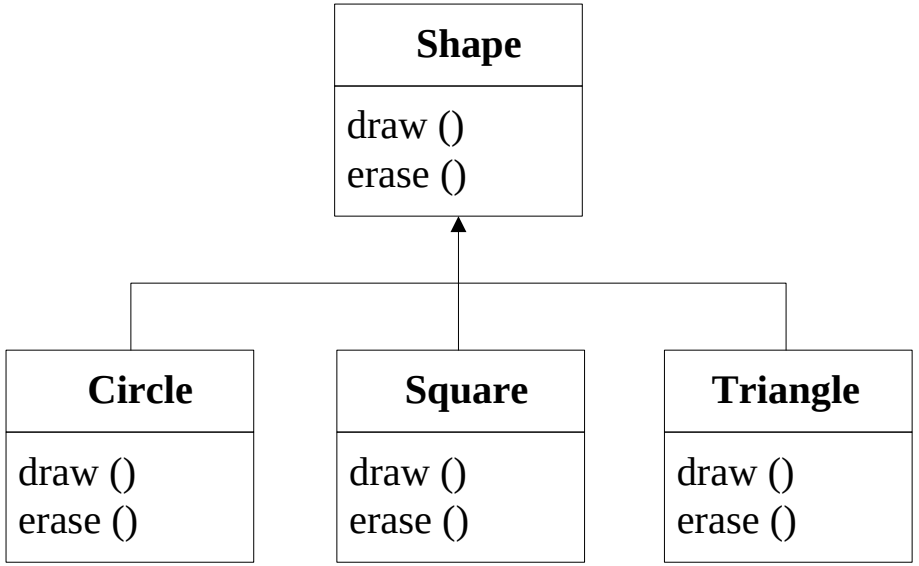
多态的目的

- 使代码变得简单且容易理解；
- 使程序具有很好的可扩展性。

例：图形类

- 在Shape 类中声明一个绘图方法。
- 在每个子类中都覆盖draw()、erase()方法。
- 以后绘图可如下进行：

```
Shape s = new Circle();
s.draw(); //实际调用的是Circle对象的draw()
```



绑定的概念

- 绑定
  - 指将一个方法调用同一个方法主体连接到一起

- 根据绑定时期的不同，可分为
  - 早绑定：程序运行之前执行绑定
  - 晚绑定：也叫作“动态绑定”或“运行期绑定，是基于对象的类别，在程序运行时执行绑定

例：动态绑定

- 仍以绘图为例，所有类都放在binding包中
  - 超类Shape建立了一个通用接口

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

  - 子类覆盖了draw方法，为每种特殊的几何形状都提供独一无二的行为

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

例：动态绑定

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

例：动态绑定

- 对动态绑定进行测试如下

```
public class BindingTester{
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        int n;
        for(int i = 0; i < s.length; i++) {
            n = (int)(Math.random() * 3);
            switch(n) {
                case 0: s[i] =  new Circle(); break;
                case 1: s[i] =  new Square(); break;
                case 2: s[i] =  new Triangle();
            }
        }
        for(int i = 0; i < s.length; i++)   s[i].draw();
    }
}
```

例：动态绑定

- 运行结果

```
Square.draw()
Triangle.draw()
Circle.draw()
Triangle.draw()
Triangle.draw()
Circle.draw()
Square.draw()
Circle.draw()
Triangle.draw()
```

- 说明
  - 在主方法的循环体中，每次随机生成一个Circle、Square或者Triangle对象；
  - 编译时无法知道s数组元素指向的实际对象类型，运行时才能确定类型，所以是动态绑定。

结语

- 这一节介绍了多态性的概念和语法。
- 多态的技术基础
  - 向上转型技术
  - 动态绑定技术

多态的应用举例

<4.4>(讲课提示，不显示)

例：二次分发

- 有不同种类的交通工具(vehicle)，如公共汽车(bus)及小汽车(car)，由此可以声明一个抽象类Vehicle及两个子类Bus及Car；
- 声明一个抽象类Driver及两个子类FemaleDriver及MaleDriver；
- 在Diver类中声明了抽象方法drives，在两个子类中对这个方法进行覆盖；
- drives接收一个Vehicle类的参数，当不同类型的交通工具被传送到此方法时，可以输出具体的交通工具；
- 所有类都放在drive包中。

例：二次分发

- 测试代码：

```
public class DriverTest {
    static public void main(String [ ] args) {
        Driver a = new FemaleDriver( );
        Driver b = new MaleDriver( );
        Vehicle x = new Car( );
        Vehicle y = new Bus( );
        a.drives(x);
        b.drives(y);
    }
}
```

- 希望输出下面的结果

```
A female driver drives a Car.
A male driver drives a bus.
```

例：二次分发

- Vehicle及其子类声明如下

```
public abstract class Vehicle
{
    private String type;
    public Vehicle( ) { }
    public Vehicle(String s) { type = s; }
    public abstract void drivedByFemaleDriver();
    public abstract void drivedByMaleDriver();
}
```

例：二次分发

```
public class Bus extends Vehicle {
        public Bus( ) { }
        public void drivedByFemaleDriver()
        { System.out.println("A female driver drives a bus."); }
        public void drivedByMaleDriver()
        { System.out.println("A male driver drives a bus."); }
}
public class Car extends Vehicle {
        public Car( ) { }
        public void drivedByFemaleDriver()
        { System.out.println("A Female driver drives a car."); }
        public void drivedByMaleDriver()
        { System.out.println("A Male driver drives a car."); }
}
```

例：二次分发

- Driver 及其子类FemaleDriver和MaleDriver

```
public abstract class Driver {
    public Driver() { }
    public abstract void drives(Vehicle v );
}
public class FemaleDriver extends Driver{
    public FemaleDriver( ) { }
    public void drives(Vehicle v){ v.drivedByFemaleDriver(); }
}
public class MaleDriver extends Driver{
    public MaleDriver( ) { }
    public void drives(Vehicle v){ v.drivedByMaleDriver(); }
}
```

例：二次分发

- 说明
  - 这种技术称为二次分发(“double dispatching”)，即对输出消息的请求被分发两次；
  - 首先根据驾驶员的类型被发送给一个类；
  - 之后根据交通工具的类型被发送给另一个类。

结语

- 这一节通过一个简单的二次分发的例子演示了多态性的一个应用

构造方法与多态性

<4.5>(讲课提示，不显示)

- 构造方法与其他方法是有区别的；
- 构造方法并不具有多态性，但仍然有必要理解在构造方法中调用多态方法会发生什么。

构造子类对象时构造方法的调用顺序

- 首先调用超类的构造方法，这个步骤会不断重复下去，首先被执行的是最远超类的构造方法；
- 执行当前子类对象的构造方法体其他语句。

例：构造方法的调用顺序

- 构建一个点类Point，一个球类Ball，一个运动的球类MovingBall继承自Ball

```
public class Point {
    private double xCoordinate;
    private double yCoordinate;
    public Point ( ) { }
    public Point(double x, double y) {
        xCoordinate = x;
        yCoordinate = y;
    }
    public String toString( )          {
        return "(" + Double.toString(xCoordinate) + ", "
            + Double.toString(yCoordinate) + ")";
    }
}
```

例：构造方法的调用顺序

```
public class Ball {
        private Point center;      //中心点
        private double radius;     //半径
        private String colour;     //颜色
        public Ball( ) { }
        public Ball(double xValue, double yValue, double r) {
           center = new Point(xValue, yValue); //调用Point中的构造方法
           radius = r;
        }
        public Ball(double xValue, double yValue, double r, String c) {
           this(xValue, yValue, r); // 调用三个参数的构造方法
           colour = c;
        }
        public String toString( ) {
           return "A ball with center " + center.toString( ) + ", radius "
               + Double.toString(radius) + ", colour " + colour;
        }
}
```

例：构造方法的调用顺序

```
public class MovingBall extends Ball {
        private double speed;
        public MovingBall( ) { }
        public MovingBall(double xValue, double yValue, double r, String c, double s) {
          super(xValue, yValue, r, c);
          speed = s;
        }
        public String toString( ) {
         return super.toString( ) + ", speed " + Double.toString(speed);
    }
}
```

- 子类不能直接存取父类中声明的私有数据成员，super.toString调用父类Ball的toString方法输出类Ball中声明的属性值

例：构造方法的调用顺序

```
public class Tester{
    public static void main(String args[]){
    MovingBall mb = new MovingBall(10,20,40,"green",25);
    System.out.println(mb);
    }
}
```

- 运行结果

  A ball with center (10.0, 20.0), radius 40.0, colour green,
  speed 25.0

例：构造方法的调用顺序

- 构造方法调用的顺序为

MovingBall(double xValue, double yValue, double r, String c, double s)

Ball(double xValue, double yValue, double r, String c)

Ball(double xValue, double yValue, double r)

Point(double x, double y)

构造方法中如果调用多态方法会发生什么?

以下文字不显示：

- 在构造方法内调用准备构造的那个对象的动态绑定方法
  - 被调用方法要操纵的成员可能尚未得到正确的初始化
  - 可能造成一些难于发现的程序错误

例：构造方法中调用多态方法

- 在Glyph中声明一个抽象方法，并在构造方法内部调用之

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

例：构造方法中调用多态方法

```
class RoundGlyph extends Glyph {
    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 {
    public static void main(String[] args) {
        new RoundGlyph(5);
    }
}
```

例：构造方法中调用多态方法

- 运行结果

  Glyph() before draw()

  RoundGlyph.draw(), radius = 0

  Glyph() after draw()

  RoundGlyph.RoundGlyph(), radius = 5

- 说明
  - 在Glyph中，draw()方法是抽象方法，在子类RoundGlyph中对此方法进行了覆盖。Glyph的构造方法调用了这个方法；
  - 从运行的结果可以看到：当Glyph的构造方法调用draw()时，radius的值甚至不是默认的初始值1，而是0。

实现构造方法的注意事项：

- 用尽可能少的动作把对象的状态设置好；
- 如果可以避免，不要调用任何方法；
- 在构造方法内唯一能够安全调用的是在超类中具有final属性的那些方法（也适用于private方法，它们自动具有final属性）。这些方法不能被覆盖，所以不会出现前述的潜在问题。

结语

- 这一节介绍了创建子类对象时，构造方法的调用次序，以及在构造方法种条用多态方法可能出现的潜在问题。

第4章小结

- 本章主要内容内容
  - 接口作用及语法
  - 类型转换的概念及应用
  - 多态的概念及应用
  - 构造方法的调用顺序及其中的多态方法