



Programming in JAVA

第5章 接口与多态

本课件以清华大学郑莉老师的课件为基础进行改编



目录

- 5.1 接口
- 5.2 塑型
- 5.3 多态的概念
- 5.4 多态的应用
- 5.5 构造方法与多态
- 5.6 内部类
- 5.7 本章小结



5.1 接口

- 接口的作用及定义
- 接口的实现
- 接口的扩展



5.1 接口

- 接口
 - 接口是描述类的部分行为的一组操作（包括操作名、参数类型和返回值类型），用于定义多个不相关类的共同行为
 - 接口的抽象程度比抽象类更进一步，其完全禁止了所有的函数定义。
 - 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

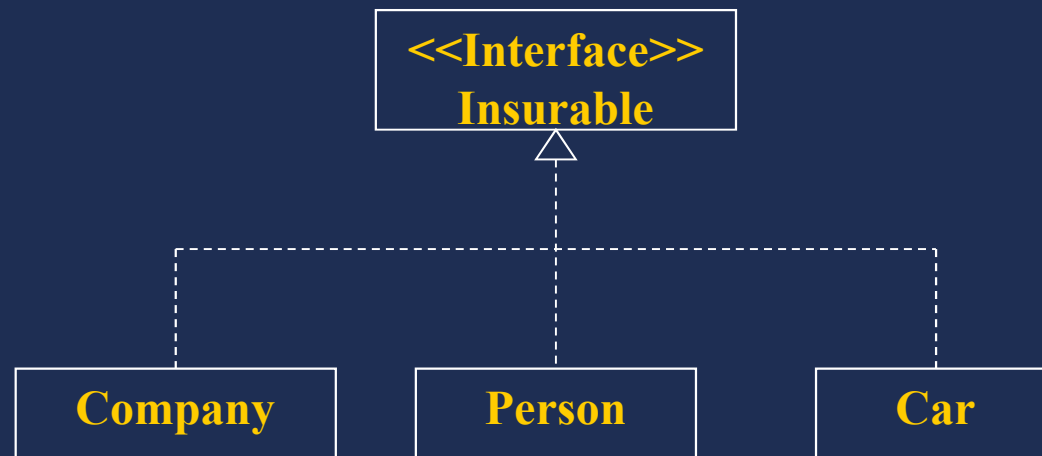


5.1.1 接口的作用及语法

——例5_1

接口

- 保险公司的例子
 - 具有车辆保险、人员保险、公司保险等多种保险业务，各种业务的保险对象不同，但在对外提供服务方面具有相似性，如都需要计算保险费(**premium**)等，因此可声明一个**Insurable** 接口，并使不同的类实现这个接口
 - 在UML图中，实现接口用带有空三角形的虚线表示



5.1.1 接口的作用及语法

接口

- 接口的语法
 - 使用关键字 `interface` 声明接口
 - 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赋初值，且此值将不能再更改，允许省略 `final` 关键字
 - 接口中的方法必须是“抽象方法”，不能有方法体，允许省略 `public` 及 `abstract` 关键字



5.1.1 接口的作用及语法

——例5_1保险接口的声明

- 例5.1中的**Insurable** 接口声明如下，可见其中的方法都是抽象方法

接口

```
public interface Insurable {  
    public int getNumber();  
    public int getCoverageAmount();  
    public double calculatePremium();  
    public Date getExpiryDate();  
}
```



5.1.1 接口的作用及语法

——例5_2

- 声明一个接口**Shape2D**，可利用它来实现二维的几何形状类**Circle**和**Rectangle**
 - 把计算面积的方法声明在接口里
 - **pi**值是常量，把它声明在接口的数据成员里

接口

```
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();
}
```



5.1.1 接口的作用及语法

接口

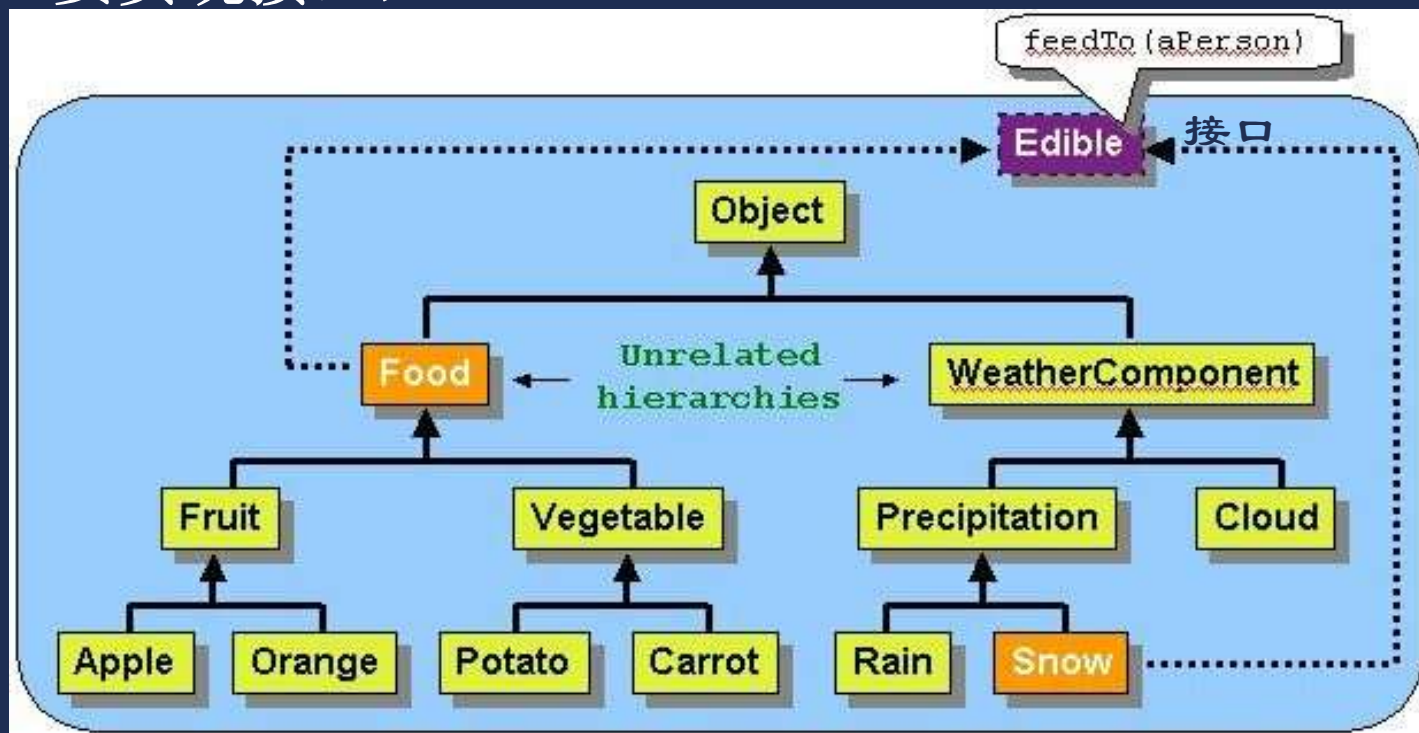
- 接口的作用
 - 是面向对象的一个重要机制
 - 实现多继承，同时免除C++中的多继承那样的复杂性
 - 建立类和类之间的“协议”
 - 把类根据其实现的功能来分别代表，而不必顾虑它所在的类继承层次；这样可以最大限度地利用动态绑定，隐藏实现细节
 - 实现不同类之间的常量共享



5.1.1 接口的作用及语法

——与抽象类的不同

抽象类描述的是本质上相同的类之间的共同行为，定义了一种**继承**关系；接口描述的是看上去不相关的类之间的共同行为，定义了一种**契约**关系（类要实现接口）。



5.1.2 实现接口

- 接口的实现
 - 接口中定义的操作需要由类来**实现**。必须和实现该接口的类联系起来，接口才有具体含义
 - 使用implements关键字，语法如下

接口

```
public class 类名称 implements 接口名
{
    /* Bodies for the interface methods */
    /* Own data and methods. */
}
```

- 必须实现接口中的所有抽象方法
- 来自接口的方法必须声明成public
- 一个类可以实现多个接口，而一个接口也可以被多个类实现，实现方式可以各不一样



5.1.2 实现接口

——例5_3

- 实现接口**Insurable**，声明汽车类实现例5.1中的**Insurable**接口，实现接口中的所有抽象方法

接
口

```
public class Car implements Insurable {  
    public int getPolicyNumber() {  
        // write code here  
    }  
    public double calculatePremium() {  
        // write code here  
    }  
    public Date getExpiryDate() {  
        // write code here  
    }  
    public int getCoverageAmount() {  
        // write code here  
    }  
    public int getMileage() { //新添加的方法  
        //write code here  
    }  
}
```



5.1.2 实现接口

——对象转型

接口

- 对象可以被转型为其所属类实现的接口类型
 - `getPolicyNumber`、`calculatePremium`是`Insurable`接口中声明的方法
 - `getMileage`是`Car`类新添加的方法，`Insurable`接口中没有声明此方法

```
Car jetta = new Car();  
Insurable item = (Insurable)jetta; //对象转型为接口类型  
item.getPolicyNumber();  
item.calculatePremium();  
item.getMileage();           // 接口中没有声明此方法，不可以  
jetta.getMileage();          // 类中有此方法，可以  
((Car)item).getMileage();    // 转型回原类，可调用此方法了
```

5.1.2 实现接口

——例5_4

- 声明**Circle**与**Rectangle**两个类实现**Shape2D**接口

接口

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

PI

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

5.1.2 实现接口

——例5_5

接
口

- 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());  
    }  
}
```

- 输出结果

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



5.1.2 实现接口

——MovableObject接口

- **MovableObject**接口定义了所有“可移动对象”能做的事情

接口

```
public interface MovableObject {  
    public boolean    start();  
    public void       stop();  
    public boolean    turn(int degrees);  
    public double     fuelRemaining();  
    public void       changeSpeed(double kmPerHour);  
}
```




5.1.2 实现接口

——MovableObject接口的实现

- Plane、Car、Train、Boat 分别实现 MovableObject 接口

接口

```
public class Plane implements MovableObject {  
    public int  seatCapacity;  
    public Company  owner;  
    public Date  lastRepairDate;  
    //实现MovableObject接口的所有方法  
    public boolean start() { //启动飞机，成功则返回true }  
    public void stop() { //停止 }  
    public boolean turn(int degrees) { //转向，成功则返回true}  
    public double fuelRemaining() { //返回燃料剩余量 }  
    public void changeSpeed(double kmPerHour) { //改变速度 }  
    //plane类自己的方法：  
    public Date getLastRepairDate() { //... }  
    public double calculateWindResistance() { //....}  
}
```



```
Plane aPlane = new Plane();
```

```
RemoteControl control=new RemoteControl(aPlane);
```

```
control.start();
```

接口

- 为 **MovableObjects** 安装遥控器(remote control)

```
public class RemoteControl {  
    private MovableObject machine;  
    RemoteControl(MovableObject m) {machine = m; }  
    //按下“启动”按钮:  
    public void start()  
    {  
        boolean okay = machine.start();  
        if (!okay) display("No Response on start");  
        //...  
    }  
}
```

- remoteControl 构造方法的形参类型为 MovableObject, 它可以是 Plane, Car, Train, Boat, 等等

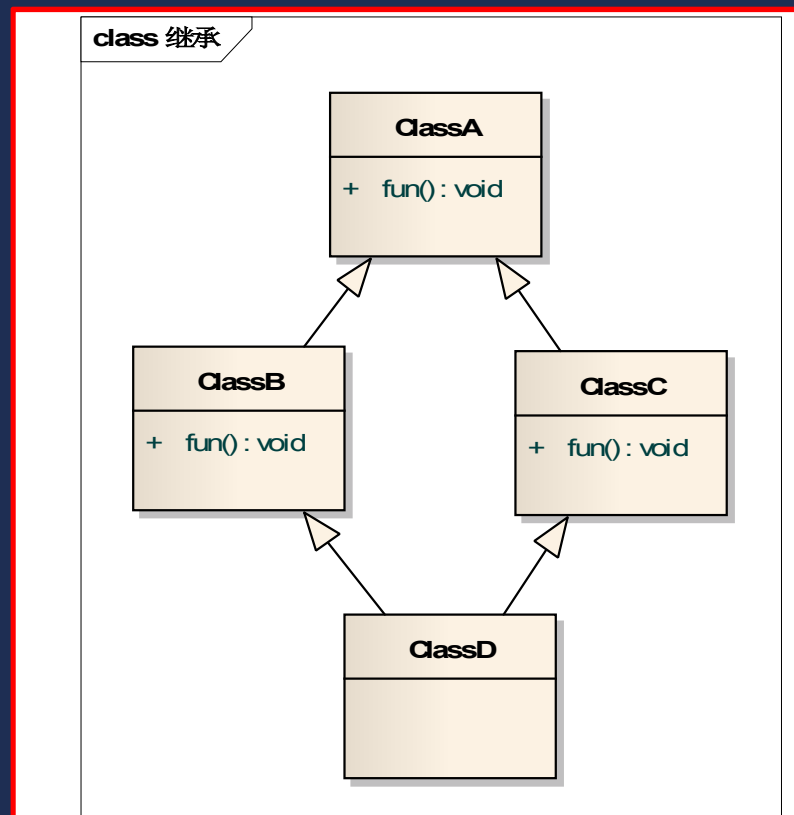


5.1.3 多重继承

- 多重继承
 - Java的设计以简单实用为导向，不允许一个类有多个父类

接口

多重继承的问题：右图中ClassD该继承哪个类的fun方法？ClassB的还是ClassC的？



5.1.3 多重继承

- 多重继承

- 但允许一个类可以实现多个接口，通过这种机制可实现多重继承
- 一个类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, ...  
{  
    ... ..  
}
```

接口



5.1.3 多重继承

——Car的例子

- Car类可以实现接口Insurable, Drivable, Sellable

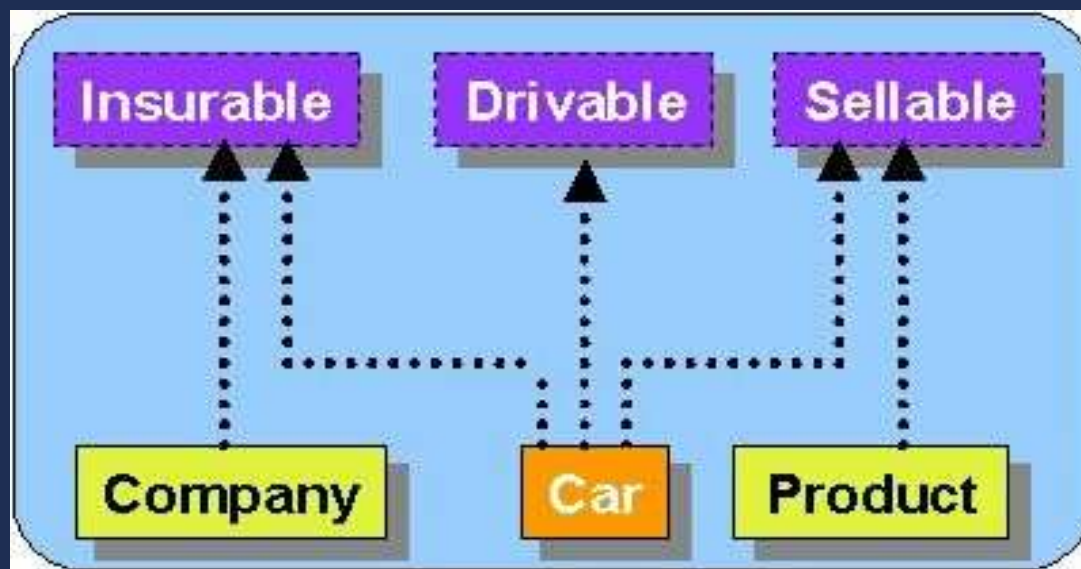
```
public class Car implements Insurable, Drivable, Sellable
```

```
{
```

```
....
```

```
}
```

接口



5.1.3 多重继承

——例5_6

接口

- 声明**Circle**类实现接口**Shape2D**和**Color**
 - **Shape2D**具有**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
}
```




5.1.3 多重继承

——例5_6

接口

```
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);
    }
}
```



5.1.4 接口的扩展

- 接口的扩展
 - 接口可通过扩展的技术派生出新的接口
 - 原来的接口称为基本接口(base interface)或父接口(super interface)
 - 派生出的接口称为派生接口(derived interface)或子接口(sub interface)
 - 派生接口不仅可以保有父接口的成员, 同时也可加入新成员以满足实际需要
 - 实现接口的类也必须实现此接口的父接口
 - 接口扩展的语法

`interface` 子接口的名称 `extends` 父接口的名称1, 父接口的名称2, ...

```
{  
    ... ..  
}
```

接
口

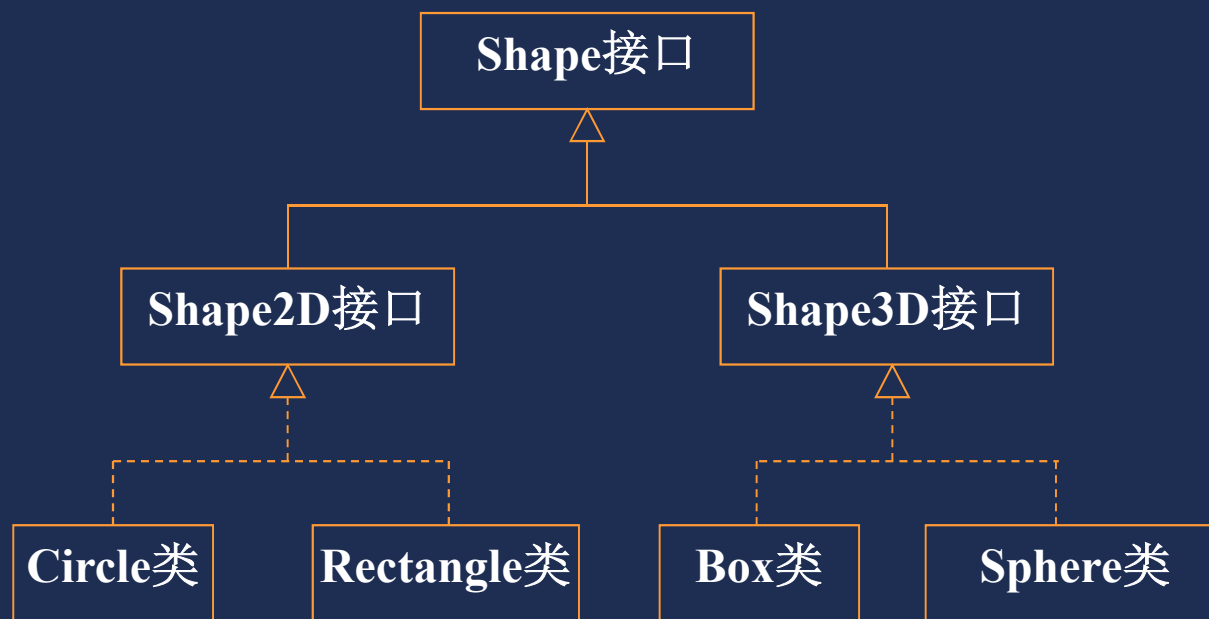


5.1.4 接口的扩展

——例5_7

- **Shape**是父接口，**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**

接口



5.1.4 接口的扩展

——例5_7

接口

- 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();  
}
```



5.1.4 接口的扩展

——例5_7

接口

```
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());  
    }  
}
```

- 运行结果
color=blue
Area = 12.56



5.2 塑型

- 塑型 (type-casting)
 - 又称为类型转换
 - 方式
 - 隐式 (自动) 的类型转换
 - 显式 (强制) 的类型转换



5.2.1 塑型的概念

塑 型

- 塑型的对象包括
 - 基本数据类型
 - 将值从一种形式转换成另一种形式
 - 引用变量
 - 将对象暂时当成更一般的对象来对待，并不改变其类型
 - 只能被塑型为
 - 任何一个父类类型
 - 对象所属的类实现的一个接口
 - 被塑型为父类或接口后，再被塑型回其本身所在的类

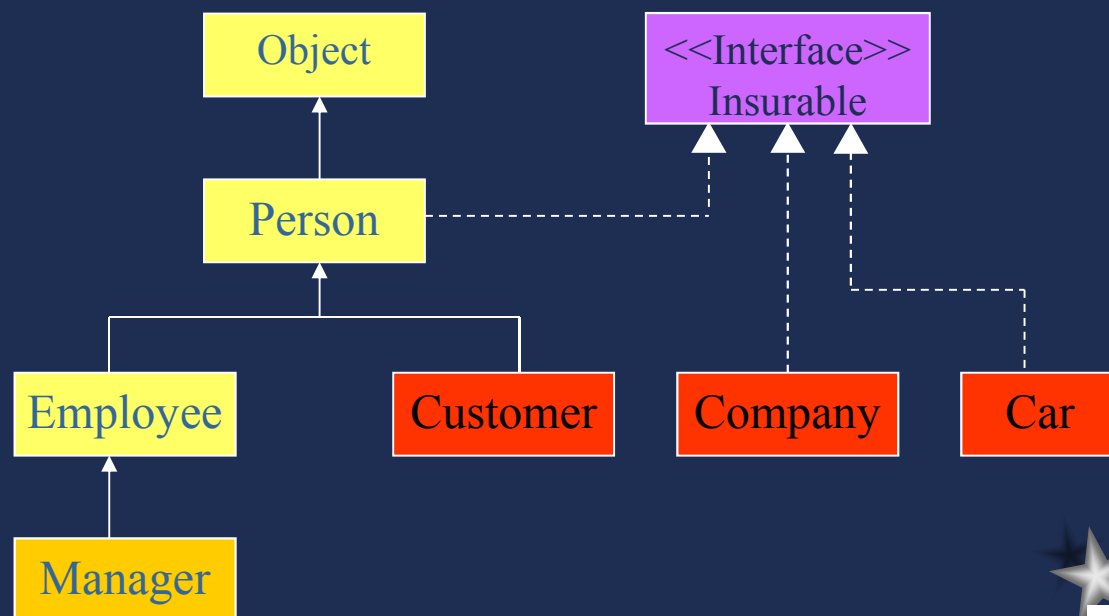


5.2.1 塑型的概念

——一个例子

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

塑
型



5.2.1 塑型的概念

——隐式(自动)的类型转换

塑
型

- 基本数据类型
 - 相容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
 - 被塑型成更一般的类
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Manager对象塑型为Employee类
 - 被塑型为对象所属类实现的接口类型
Insurable item = new Manager();



5.2.1 塑型的概念

——显式(强制)的类型转换

塑
型

- 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：还原为本来的类型

Employee emp;

Manager man;

emp = new Manager();

man = (Manager)emp; //将emp强制塑型为本来的类型



5.2.2 塑型的应用

塑 型

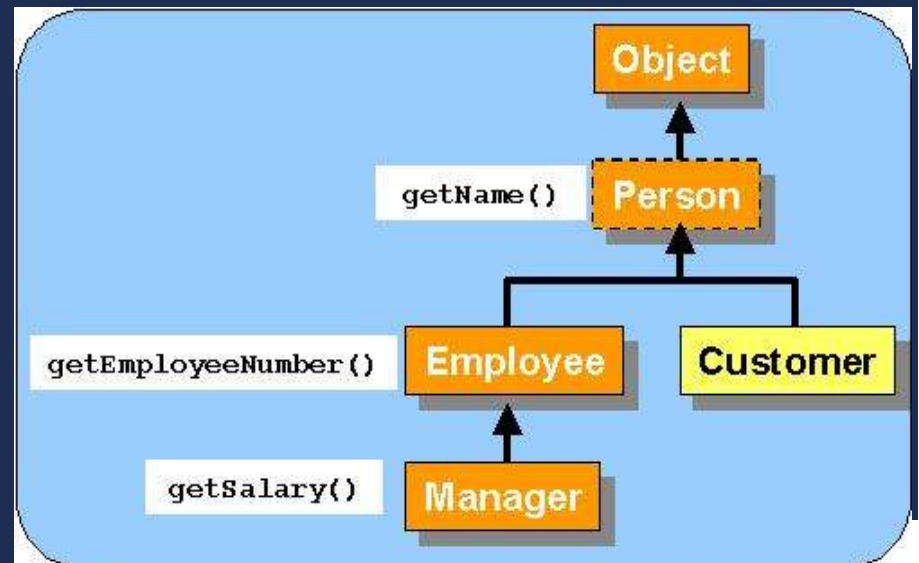
- 塑型应用的情况包括
 - 赋值转换
 - 赋值号右边的表达式类型或对象转换为左边的类型
 - 方法调用转换
 - 实参的类型转换为形参的类型
 - 算术表达式转换
 - 算术混合运算时，不同类型的项转换为相同的类型再进行运算
 - 字符串转换
 - 字符串连接运算时，如果一个操作数为字符串，一个操作数为数值型，则会自动将数值型转换为字符串

5.2.2 塑型的应用

- 当一个类对象被塑型为其父类后，它提供的方法会减少
 - 当Manager对象被塑型为Employee之后，它只能接收getName()及getEmployeeNumber()方法，不能接收getSalary()方法
 - 将其塑型为本来的类型后，又能接收getSalary()方法了

塑
型

```
Employee emp=new Manager();  
emp.getName();//ok  
emp.getEmployeeNumber();//ok  
emp.getSalary();//不允许  
((Manager)emp).getSalary();//允许
```

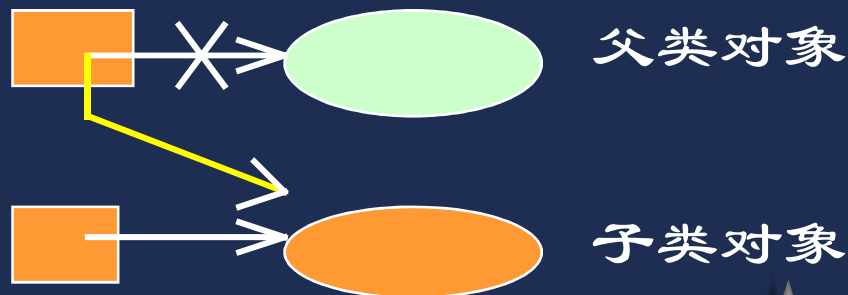


5.2.3 方法的查找

塑型

- 如果在塑型前和塑型后的类中都提供了相同的方法，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？

- 实例方法
- 类方法

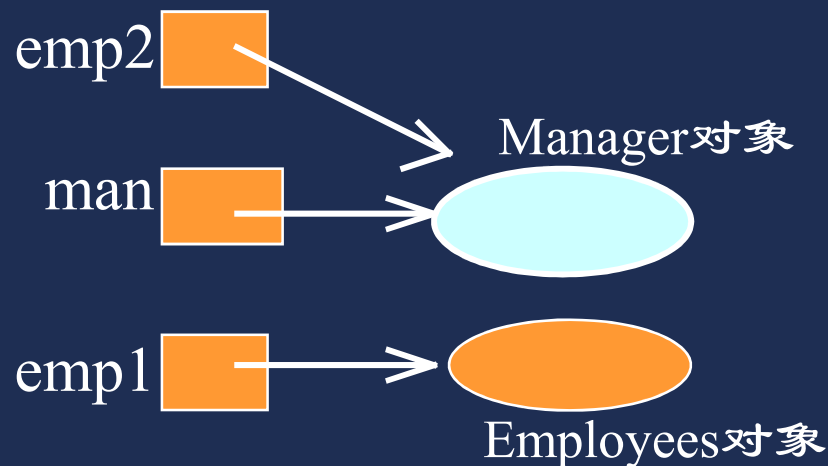
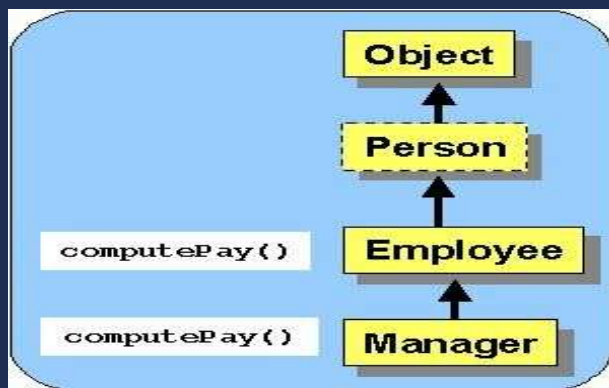


5.2.3 方法的查找

——实例方法

- 从对象创建时所属类开始，沿类层次向上查找

塑
型



```

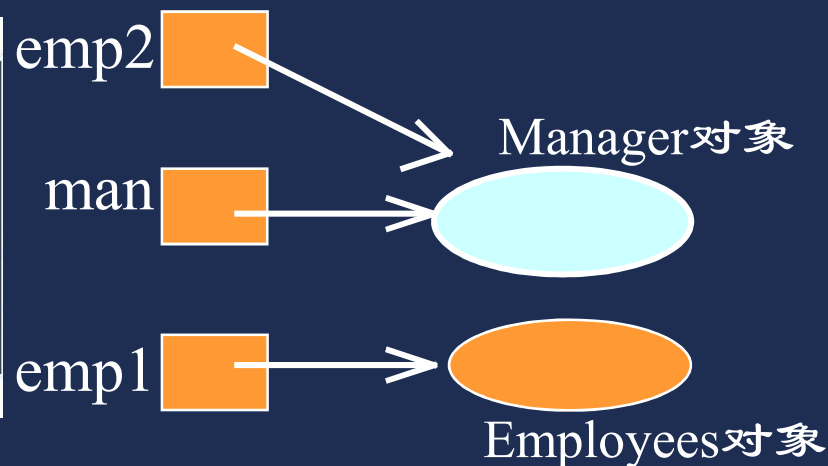
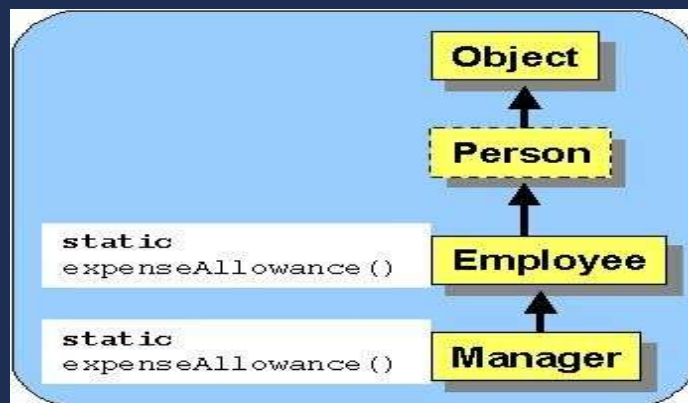
Manager man = new Manager();
Employee emp1 = new Employee();
Employee emp2 = (Employee)man;
emp1.computePay();           // 调用Employee类中的computePay() 方法
man.computePay();           // 调用Manager类中的computePay() 方法
emp2.computePay();           // 调用Manager类中的computePay() 方法
    
```

5.2.3 方法的查找

——类方法

- 总是在引用变量声明时所属的类中进行查找

塑
型



```

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



5.3 多态的概念

多态

- 多态性是一种方法，使得在多个不同的类中可以定义相同的操作，而这些操作在这些类中可以有不同的实现。例如，一个显示操作display，作用于Polygon对象是在屏幕上显示一个多边形；而同样的显示操作作用于Circle对象，则在屏幕上显示一个圆；作用于正文Text对象，则是在屏幕上显示一段正文。
- 也就是说，同样一个操作发送给不同类的对象，每个对象将根据自己所属的类中定义的该操作的执行方式进行动作，从而产生不同的结果。



5.3.1 多态的目的

多态的概念

- 多态的目的
 - 多个类型（从相同的父类型中衍生出来）可被当作同一种类型对待。用父类去代表多个子类，只让自己的代码与父类打交道。运用多态性可以简化设计、使设计灵活，易于扩展。



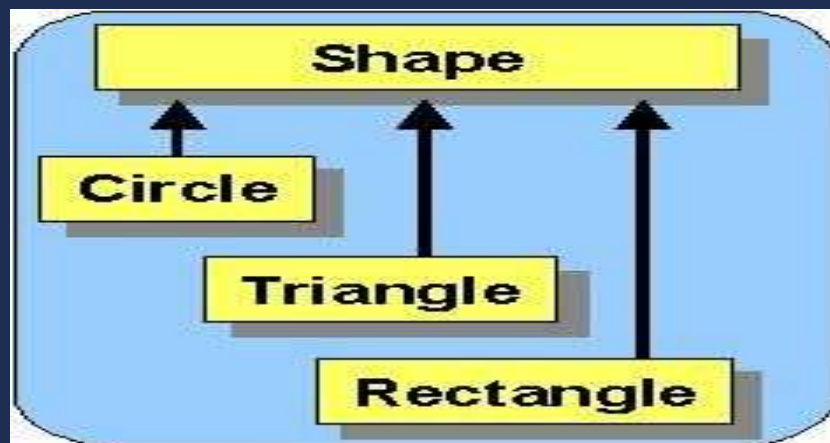
5.3.1 多态的目的

——一个例子

- 绘图

- 希望能够画出任意子类型对象的形状，可以在 **Shape** 类中声明几个绘图方法，对不同的实际对象，采用不同的画法

```
if (aShape instanceof Circle)    aShape.drawCircle();  
if (aShape instanceof Triangle) aShape.drawTriangle();  
if (aShape instanceof Rectangle)aShape.drawRectangle();
```



多
态
的
概
念

5.3.1 多态的目的

——一个例子

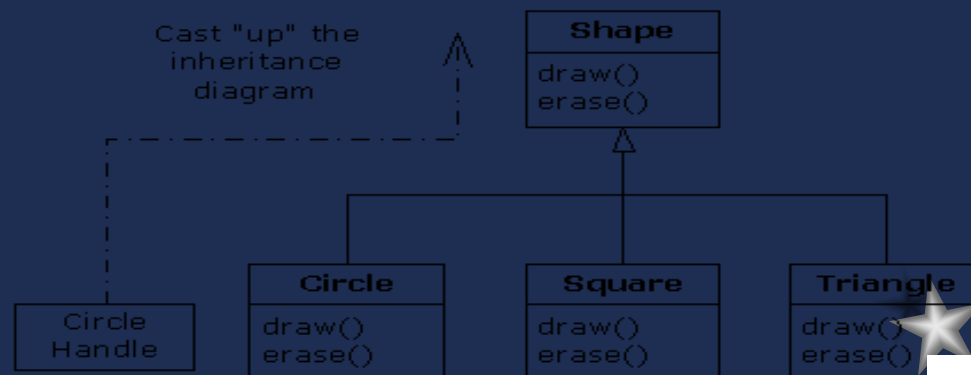
多态的概念

- 更好的方式

- 在每个子类中都声明同名的draw()方法
- 以后绘图可如下进行

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

- Circle属于Shape的一种，系统会执行自动塑型
- 当调用方法draw时，实际调用的是Circle.draw()
- 在程序运行时才进行绑定，接下来介绍绑定的概念



5.3.2 绑定的概念

多态的概念

- 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
 - 早期绑定
 - 程序运行之前执行绑定
 - 晚期绑定
 - 也叫作“动态绑定”或“运行期绑定
 - 基于对象的类别，在程序运行时执行绑定



例5-8 图形的例子：定义一个抽象类Shape以及三个子类Circle、Triangle、Square。子类覆盖父类的draw方法，因为每种特殊类型的几何形状都提供独一无二的行为

//基类Shape建立了一个通用接口

编译的时候是无法知道
s数组元素的具体类型
到底是什么，运行的时
候才能确定究竟是什么
类型，所以是动态绑定

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

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

```
class Triangle extends Shape {
    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); //n的取值为0~2
                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();
        }
    }
}
```

5.4 多态的应用

- 多态的应用

- 技术基础

- 向上塑型技术：一个父类的引用变量可以指向不同的子类对象
 - 动态绑定技术：运行时根据父类引用变量所指对象的实际类型执行相应的子类方法，从而实现多态性

多态的概念



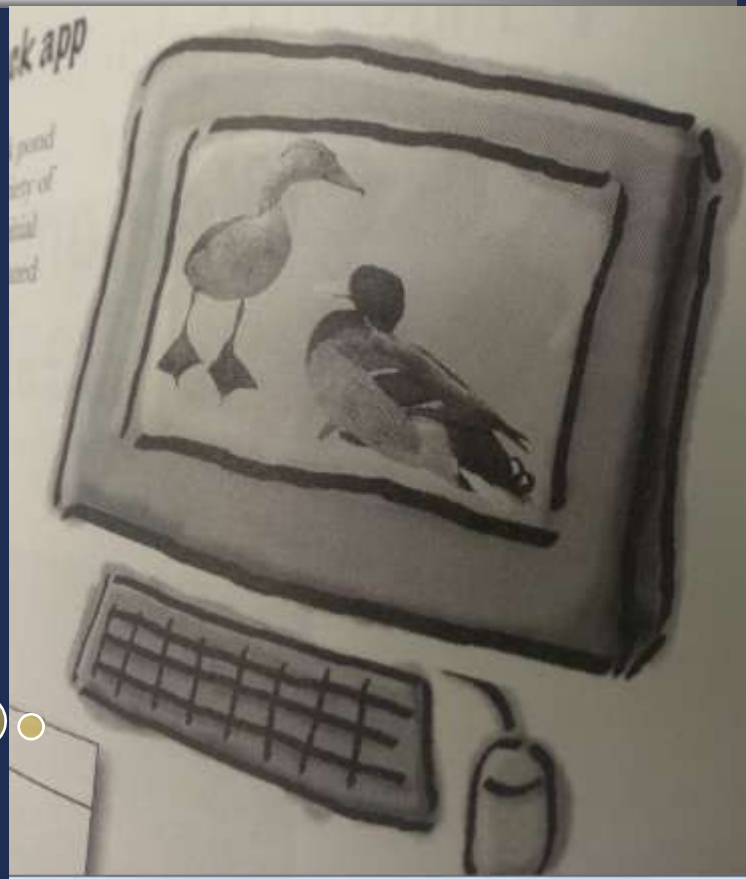
- 由于只有在运行时才能知道调用哪个类的**draw**方法，因此在每次创建一个新的图形对象时，必须记录其方法存储位置的信息，这样对于调用哪个方法的决定可以推迟到运行时完成，该技术就是动态（运行时）绑定技术。
- **Java**使用动态绑定技术，但动态绑定技术比静态绑定技术需要更多的处理过程和存储空间，因此，如果确切知道一个方法不会被重写，就使用**final**关键字，然后编译器对此方法将使用静态绑定技术。



● 案例

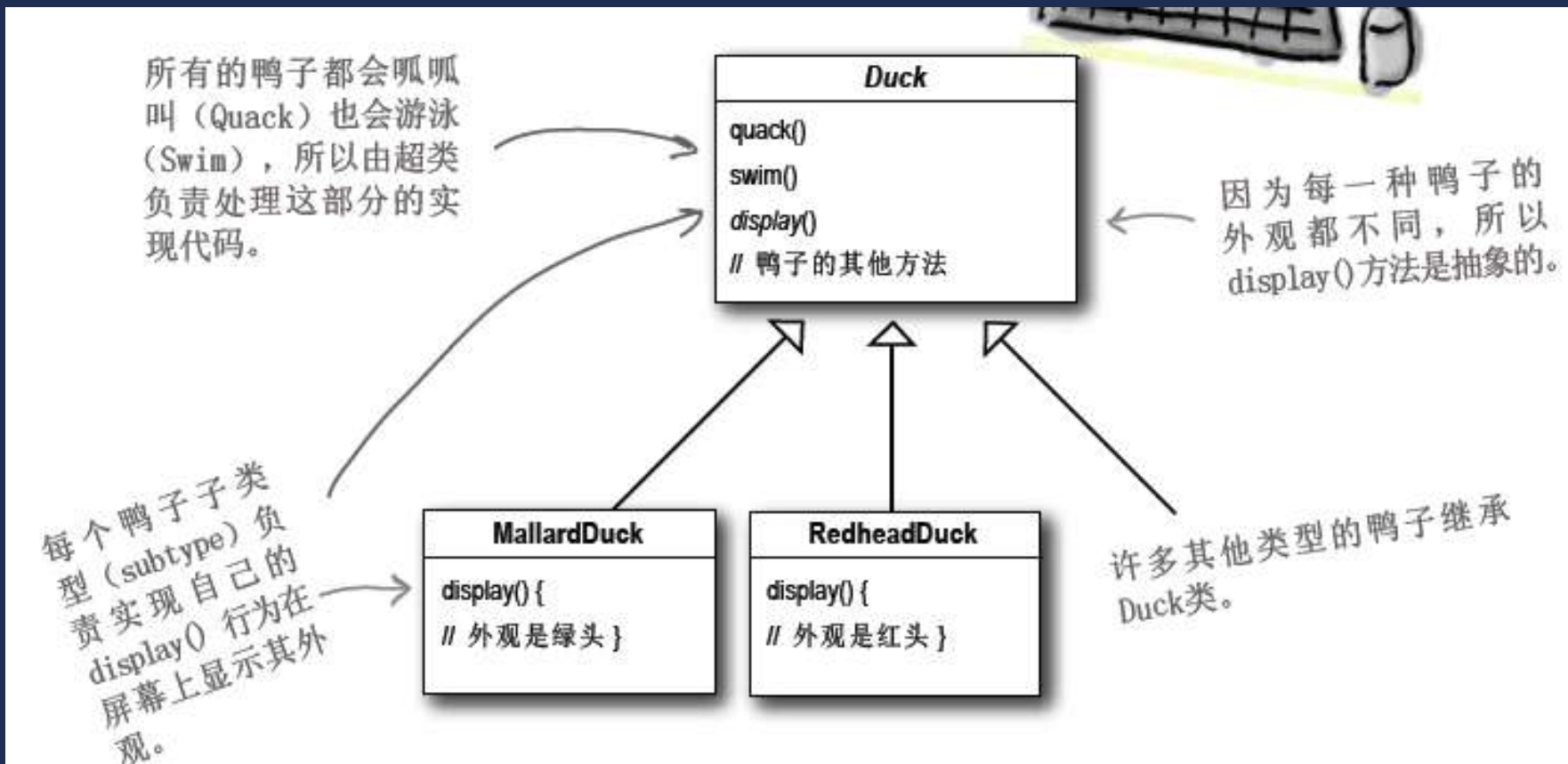
Duck App: 一个小程序，模拟一群各式各样的鸭子在池子里面游泳，并且呀呀叫。

各式各样的鸭子？但他们有一些共同的行为（游泳、叫），简单，使用继承好了！



案例：Duck app

- 最初设计：应用继承，将共性的行为（**quack**和**swim**）在超类中定义，子类重写特性的部分（**display**）



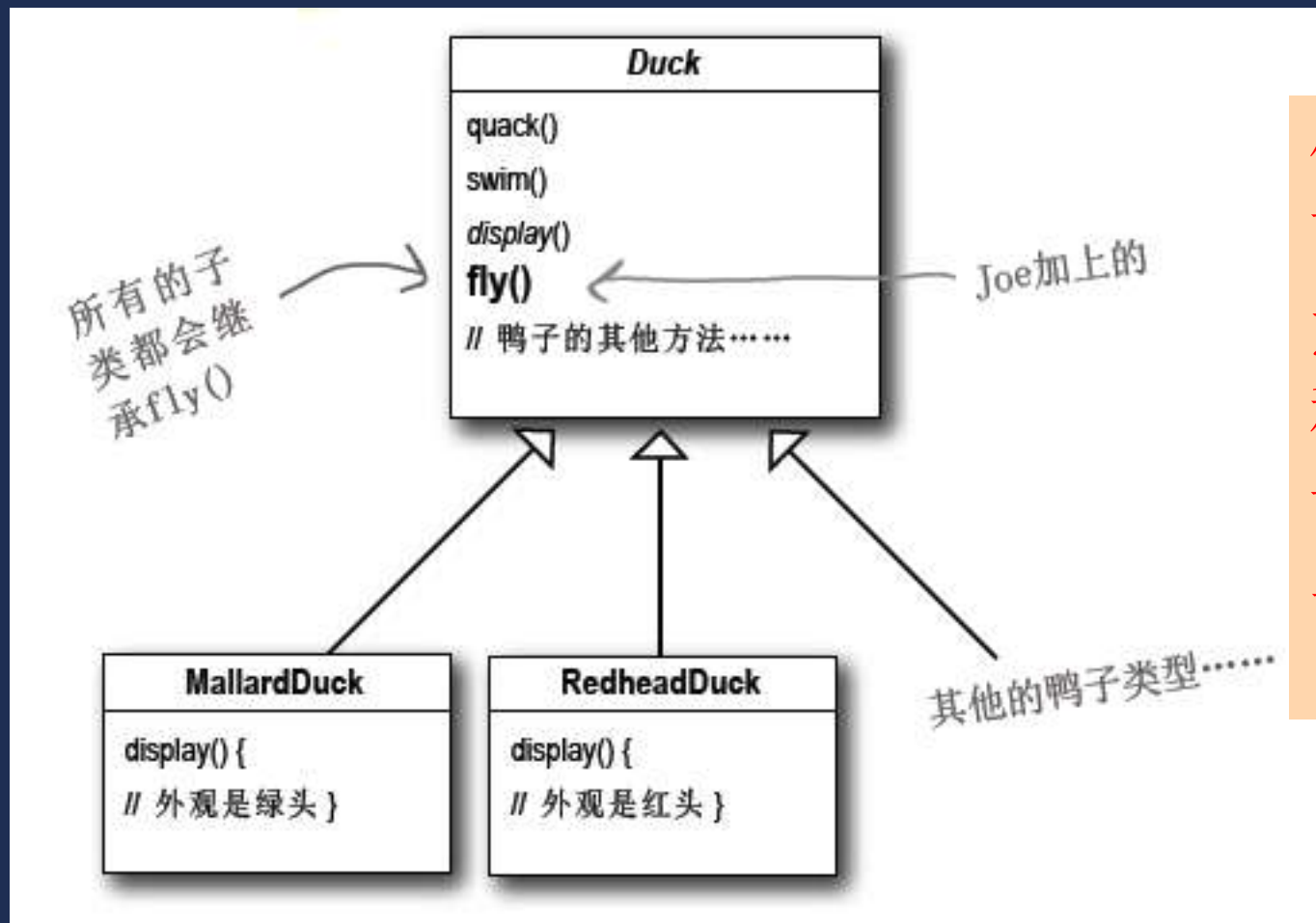
- 新需求来啦，有些鸭子会飞，请模拟出这个行为！



难不倒我！我只需要在**Duck**类中加上**fly()**方法，然后所有鸭子都会继承**fly()**。



瞧瞧这设计怎么样：



似乎合理，
可是，这就
意味着：各
种不同的鸭
子都能fly，
与事实不符！

中重写

我可以把橡皮鸭类中的
fly()方法覆盖掉，就好像
覆盖quack()的做法一样……



```

class RubberDuck
{
    quack() { // 吱吱叫 }
    display() { // 橡皮鸭 }
    fly() {
        // 覆盖，变成什么事都不做
    }
}
    
```

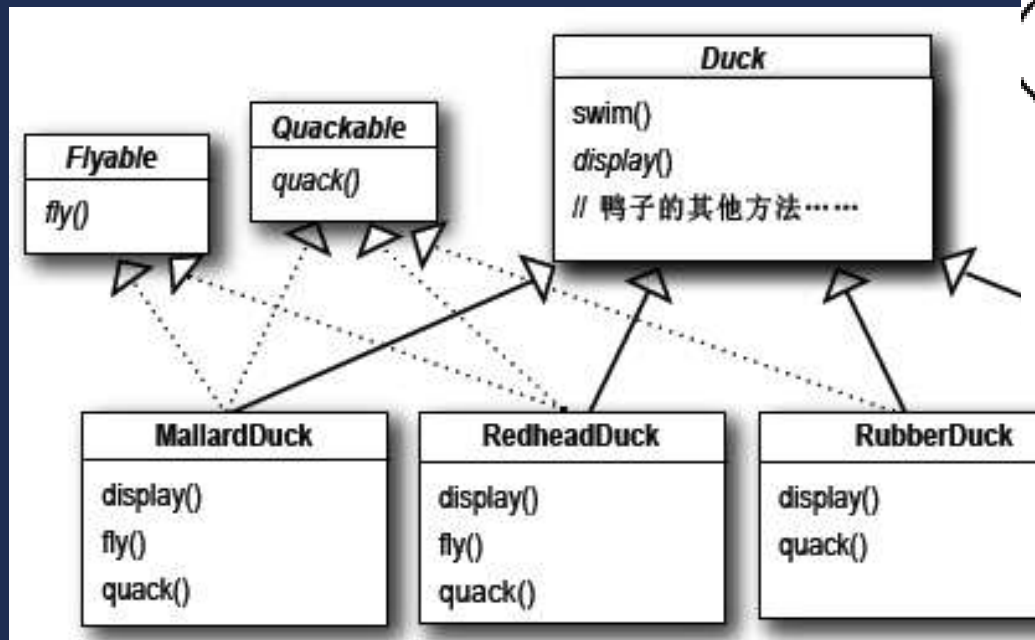
可是，如果以后我加入诱
饵鸭（DecoyDuck），又会
如何？诱饵鸭是木头假鸭，不
会飞也不会叫……

```

class DecoyDuck
{
    quack() {
        // 覆盖，变成什么事都不做
    }
    display() { // 诱饵鸭 }
}
    
```

似乎合理，可是，这就意味着：以后如果有新类型的鸭子加入，如果它们不能fly或者quack，每次就得重写fly或者quack 方法！

- 不能这样！现在，需要想想办法，使得只有部分类型的鸭子能 **fly** 或者 **quack**。
- 怎么做呢？--先试试使用接口。定义两个接口 **Flyable** 和 **Quackable**，能 **fly** 的鸭子就实现 **Flyable** 接口，能 **quack** 的就实现 **Quackable** 接口



看上去还不错！可是，这就意味着：每个实现接口的类要书写自己的 **fly** 和 **quack**，无法重用代码，即使多个 **duck** 有同样的 **fly** 和 **quack** 行为！代码的无法重用带来维护的困难！

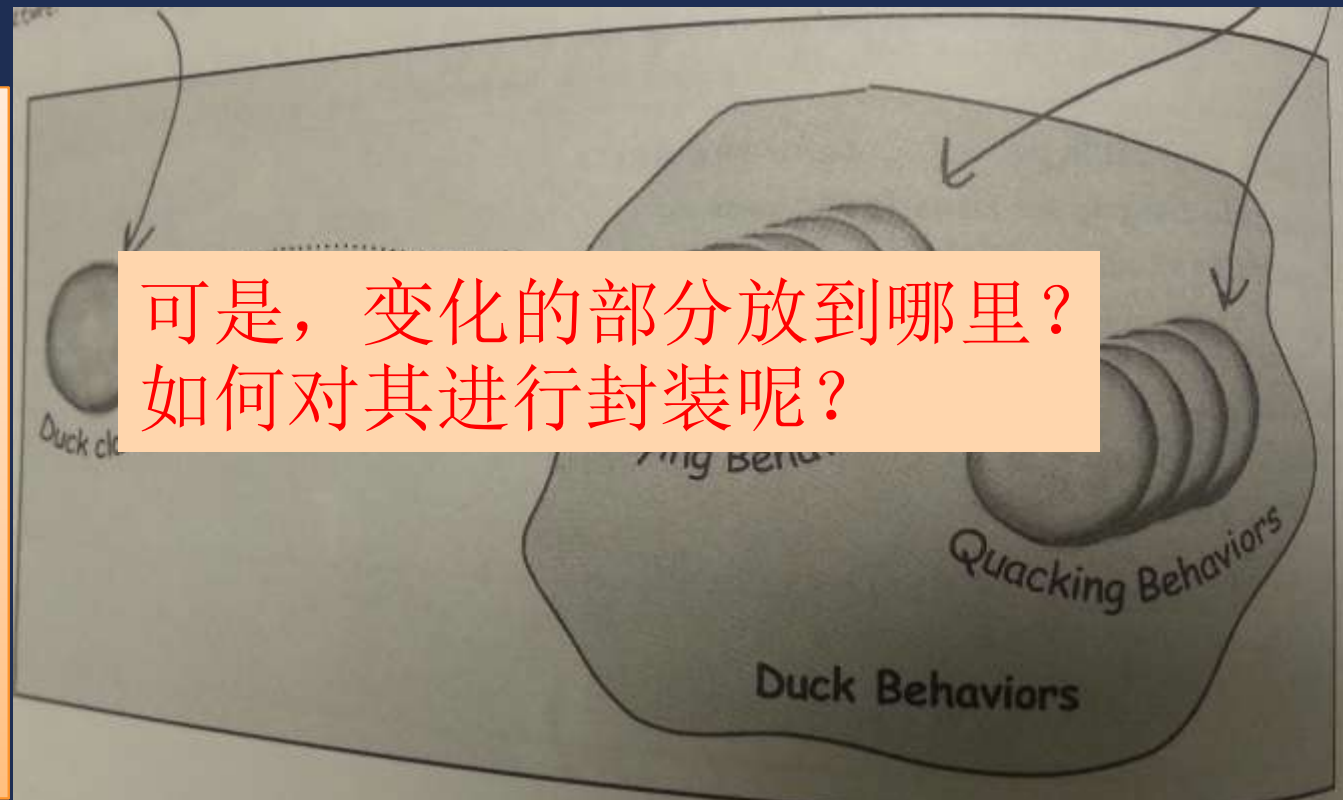
到底该怎么做？

设计高手，求助

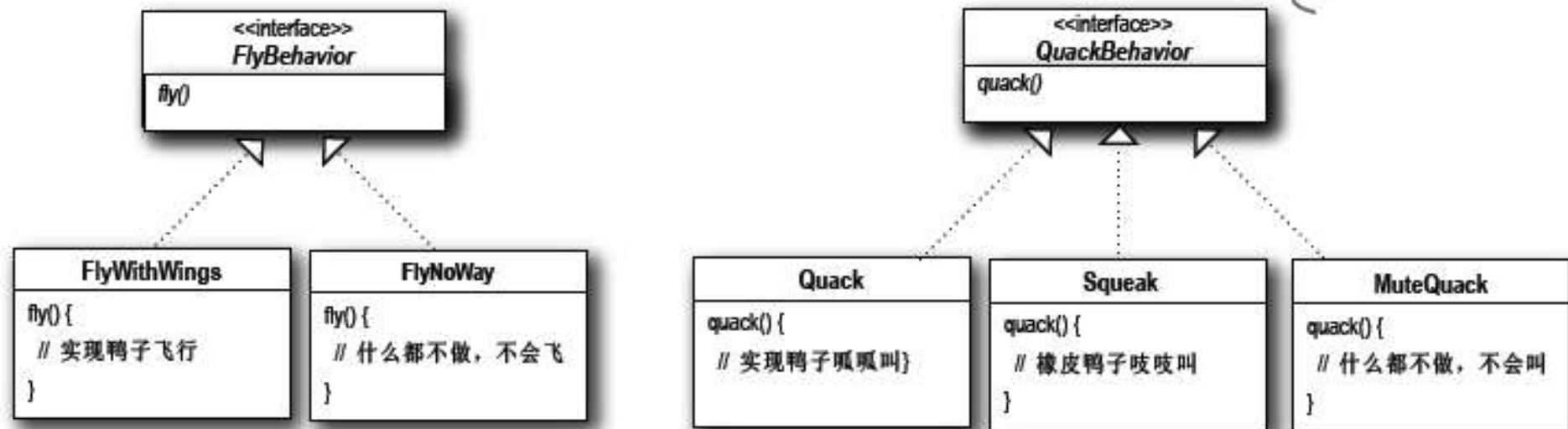


- **需要知道的设计原则**：识别设计中容易变化的部分，将变化部分与稳定的部分分离，并将变化的部分进行封装。这样将来你可以扩展和改变变化的部分，而不会对稳定部分造成影响。

在 D u c k
A P P 中，
鸭子可能会
q u a c k
和 f l y ，
也可能不会。
这就是容易
变化的部分！



- 解决方案：定义两个接口FlyBehavior和QuackBehavior，定义一些代表不同行为的子类来实现这些接口。



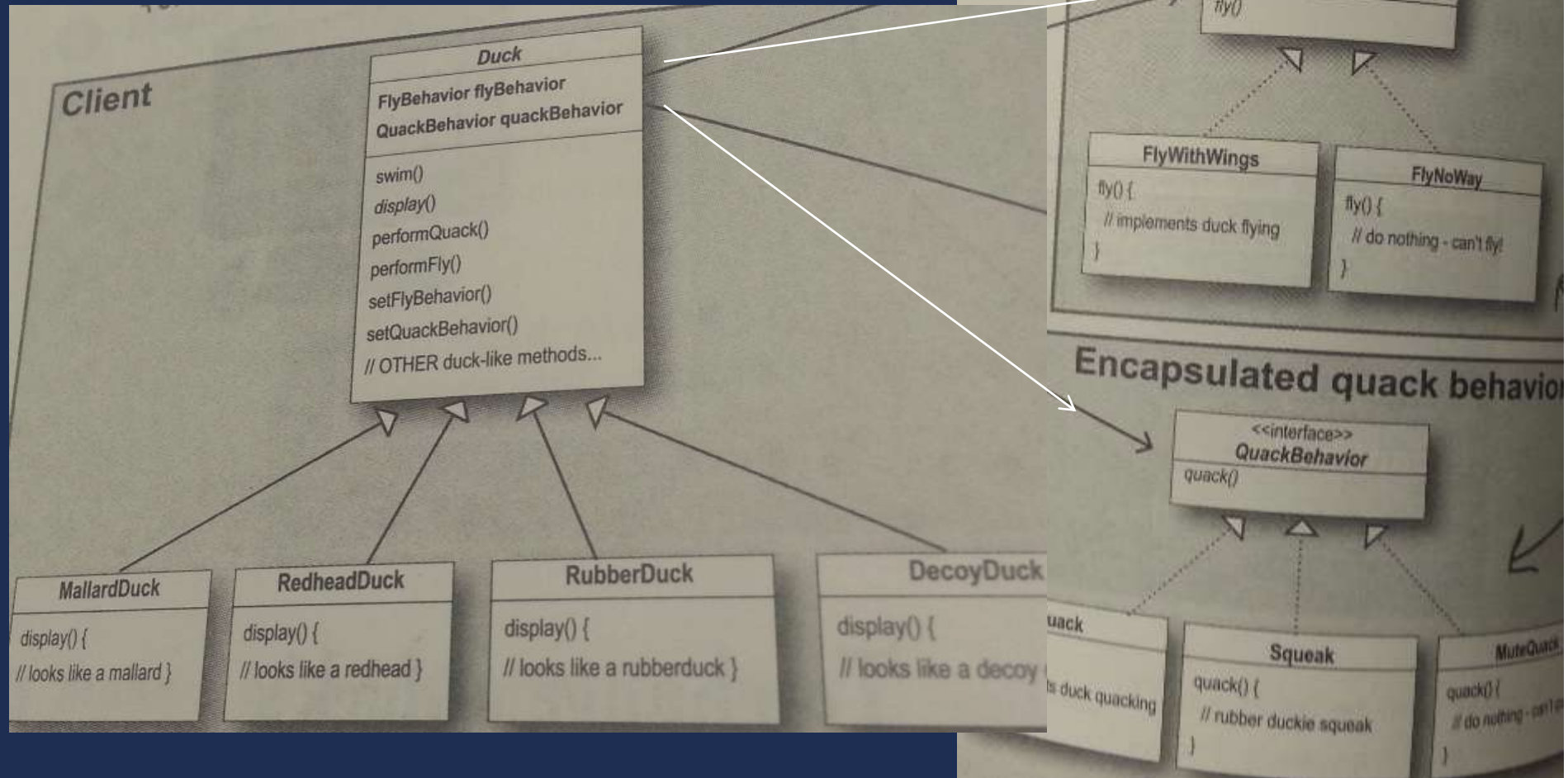
接下去，这些易变部分如何和稳定部分结合呢？

这里实现了所有有翅膀的鸭子飞行动作

真的吗

名为呱呱叫，其实不出声

- **设计原则：** 依赖于抽象、
优先使用聚合、而不是继承



- 如果新增一种鸭子，该鸭子的**quack**和之前定义的都不一样，该怎么办？
- 如果新增了一种行为（假设有的鸭子能**Song**），该怎么办？

源代码 (java)



polymorphism.rar

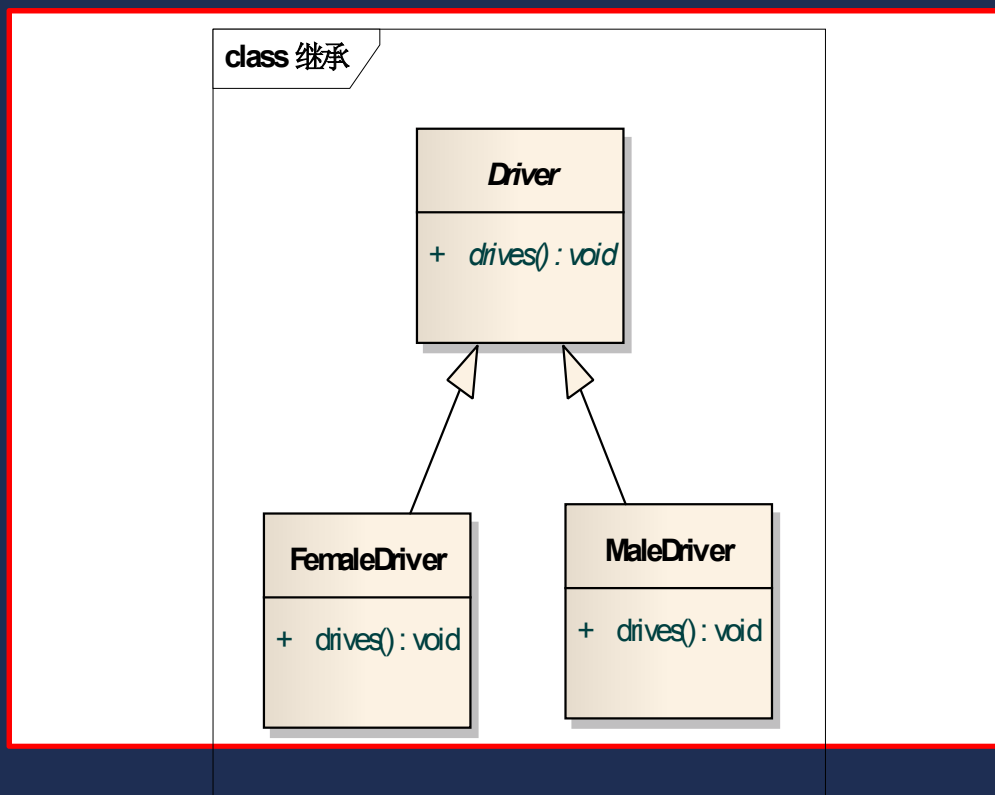


5.4 多态的应用(续)

——例5_9

- 声明一个抽象类Driver及两个子类FemaleDriver及MaleDriver

多态的概念



- 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了重写

```
public abstract class Driver
{
    public Driver( ) { }
    public abstract void drives( );
}
```




5.4 多态的应用(续)

——例5_9

多
态
的
概
念

```
public class FemaleDriver extends Driver {  
    public FemaleDriver( ) { }  
    public void drives( ) {  
        System.out.println("A Female driver drives a vehicle.");  
    }  
}  
  
public class MaleDriver extends Driver {  
    public MaleDriver( ) { }  
    public void drives( ) {  
        System.out.println("A male driver drives a vehicle.");  
    }  
}
```



5.4 多态的应用(续)

——例5_9

多
态
的
概
念

```
public class Test1
{
    static public void main(String [ ] args)
    {
        Driver a = new FemaleDriver( );
        Driver b = new MaleDriver( );
        a.drives( );
        b.drives( );
    }
}
```

- 运行结果

A Female driver drives a vehicle.

A male driver drives a vehicle.



5.4 多态的应用(续)

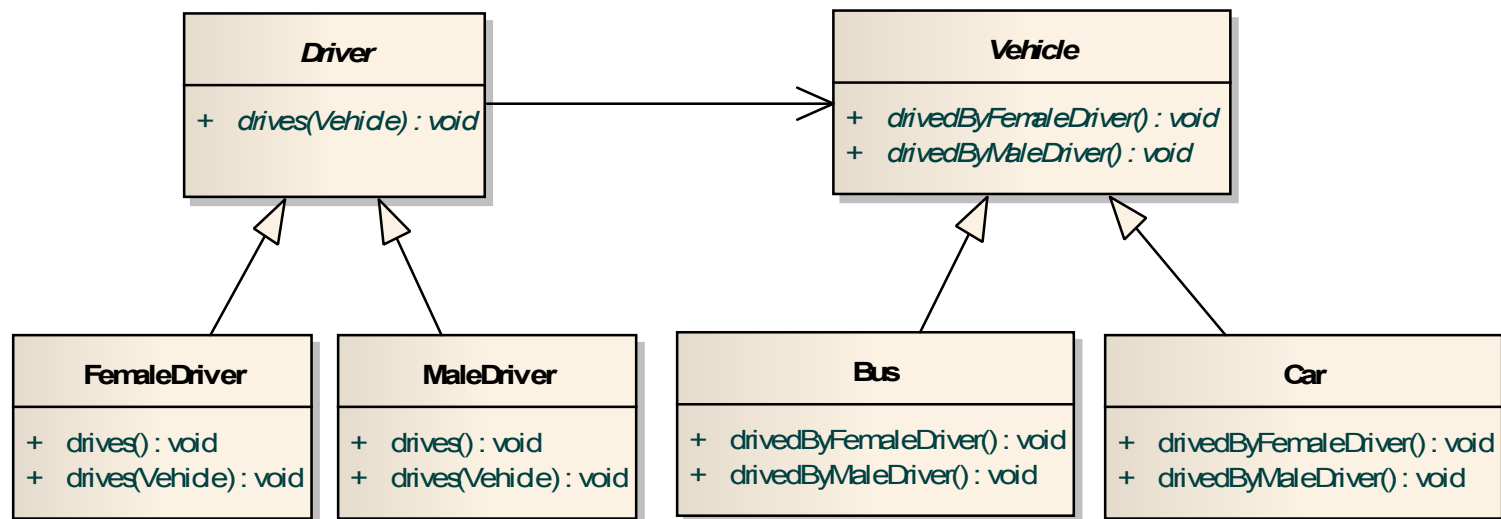
——例5_9改进

多
态
的
概
念

- 所有类都放在**drive**包中
 - 试想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
 - 对前面的**drives**方法进行改进, 使其接收一个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



class 继承



5.4 多态的应用(续)

——例5_9改进

多
态
的
概
念

- 测试代码可改写如下:

```
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);  
    }  
}
```

- 并希望输出下面的结果
A female driver drives a Car.
A male driver drives a bus.



5.4 多态的应用(续)

——例5_9改进

多
态
的
概
念

- **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 drivenByFemaleDriver();
    public abstract void drivenByMaleDriver();
}
```



5.4 多态的应用(续)

——例5_9改进

多
态
的
概
念

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



5.4 多态的应用(续)

——例5_9改进

- 对**FemaleDriver**及**MaleDriver**类中的**drives**方法进行改进，在**drives**方法的定义体中不直接输出结果，而是调用**Bus**及**Car**类中的相应方法

多
态
的
概
念

```
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(); }  
}
```



5.4 多态的应用(续)

——例5_9改进运行结果

多
态
的
概
念

- 运行结果

A Female driver drives a car.

A male driver drives a bus.

- 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給一个类
 - 之后根据交通工具的类型被发送給另一个类



5.5 构造方法与多态

- 构造方法与多态
 - 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，但仍然非常有必要理解构造方法如何在复杂的分级结构中随同多态性一同使用的情况



5.5.1 构造方法的调用顺序

构造方法与多态

- 构造方法的调用顺序
 - 调用基类的构造方法。这个步骤会不断重复下去，首先得到构建的是分级结构的根部，然后是下一个派生类，等等。直到抵达最深一层的派生类
 - 按声明顺序调用成员初始化模块
 - 调用派生构造方法



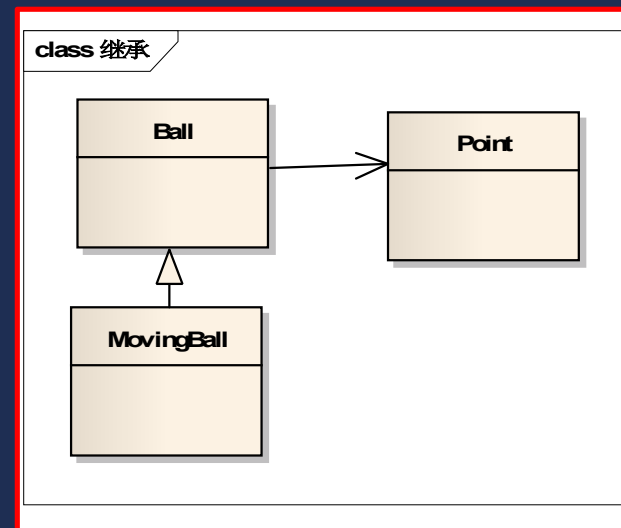
5.5.1 构造方法的调用顺序(续)

——例5_10

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

构造方法与多态

```
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) + ")";
    }
}
```




5.5.1 构造方法的调用顺序(续)

——例5_10

构造
方法
与
多
态

```
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;  
    }  
}
```



5.5.1 构造方法的调用顺序(续)

——例5_10

构造
方法
与
多
态

```
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

5.5.1 构造方法的调用顺序(续)

——例5_10运行结果

构造
方法
与
多
态

```
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



5.5.1 构造方法的调用顺序(续)

——例5_10说明

- 上面的代码中，构造方法调用的顺序为

构造
方法
与
多
态

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)



5.5.1 构造方法的调用顺序(续)

——例5_11

构造
方法
与
多
态

- 构造方法的调用顺序举例2

```
class Meal { //饭类
    Meal() { System.out.println("Meal()"); }
}
class Bread { //面包类
    Bread() { System.out.println("Bread()"); }
}
class Cheese { //奶酪类
    Cheese() { System.out.println("Cheese()"); }
}
class Lettuce { //莴苣类
    Lettuce() { System.out.println("Lettuce()"); }
}
```



5.5.1 构造方法的调用顺序(续)

——例5_11

构造
方法
与
多
态

```
class Lunch extends Meal { //午餐类继承自饭类
    Lunch() {System.out.println("Lunch()");}
}
class PortableLunch extends Lunch {
    PortableLunch() { System.out.println("PortableLunch()"); }
}
public class Sandwich extends PortableLunch {
    Bread b = new Bread();
    Cheese c = new Cheese();
    Lettuce l = new Lettuce();
    Sandwich(){System.out.println("Sandwich()");}
    public static void main(String[] args) { new Sandwich(); }
}
```



5.5.1 构造方法的调用顺序(续)

——例5_11运行结果

构造方法与多态

- 输出结果

Meal()

Lunch()

PortableLunch()

Bread()

Cheese()

Lettuce()

Sandwich()

- 说明

- 当我们在构造派生类的时候，必须能假定基类的所有成员都是有效的。在构造方法内部，必须保证使用的所有成员都已初始化。因此唯一的办法就是首先调用基类构造方法，然后在进入派生类构造方法之前，初始化所有能够访问的成员



5.5.2 构造方法中的多态方法

构造方法与多态

- 构造方法中的多态方法
 - 在构造方法内调用准备构造的那个对象的动态绑定方法
 - 会调用位于派生类里的一个方法
 - 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



5.5.2 构造方法中的多态方法(续)

——例

5_12

构造
方法
与
多
态

- 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()");  
    }  
}
```



5.5.2 构造方法中的多态方法(续)


——例

5_12

构造
方法
与
多
态

```
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);
    }
}
```



5.5.2 构造方法中的多态方法(续)

——例5_12运行

结果

构造方法与多态

- 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

5.5.2 构造方法中的多态方法

构造方法与多态

- 定义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基类中具有**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，所以不会出现上述潜在的问题



5.6 内部类

- 内部类
 - 在另一个类或方法的定义中定义的类
 - 可访问其外部类中的所有数据成员和方法成员
 - 可对逻辑上相互联系的类进行分组
 - 对于同一个包中的其他类来说，能够隐藏
 - 可非常方便地编写事件驱动程序
 - 声明方式
 - 命名的内部类：可在类的内部多次使用
 - 匿名内部类：可在new关键字后声明内部类，并立即创建一个对象
 - 假设外层类名为Myclass，则该类的内部类名为
 - Myclass\$c1.class (c1为命名的内部类名)
 - Myclass\$1.class (表示类中声明的第一个匿名内部类)



5.6 内部类

——Parcel1.java

内
部
类

```
public class Parcel1 {  
    class Contents { //内部类  
        private int i = 11;  
        public int value() { return i; }  
    }  
    class Destination { //内部类  
        private String label;  
        Destination(String whereTo) { label = whereTo; }  
        String readLabel() { return label; }  
    }  
    public void ship(String dest) {  
        Contents c = new Contents();  
        Destination d = new Destination(dest);  
        System.out.println(d.readLabel());  
    }  
}
```



5.6 内部类

——Parcel1.java

内部类

```
public static void main(String[] args) {  
    Parcel1 p = new Parcel1();  
    p.ship("Tanzania");  
}  
}
```

- 说明

- 在Parcel1类中声明了两个内部类Contents、Destination
- 在ship方法中生成两个内部类对象，并调用了内部类中声明的一个方法



5.6 内部类

——Parcel2.java

- 外部类的方法可以返回内部类的引用变量

内部类

```
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(); }  
}
```

5.6 内部类

——Parcel2.java

内
部
类

```
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();  
    Parcel2.Contents c = q.cont();  
    Parcel2.Destination d = q.to("Borneo");  
}  
}
```

- 说明

- to()方法返回内部类Destination的引用
- cont()方法返回内部类Contents的引用



5.6 内部类

——内部类实现接口

内部类

- 内部类实现接口
 - 可以完全不被看到，而且不能被调用
 - 可以方便实现“隐藏实现细则”。你所能得到的仅仅是指向基类(base class)或者接口的一个引用
- 例子

```
abstract class Contents {  
    abstract public int value();  
}  
  
interface Destination {  
    String readLabel();  
}
```



5.6 内部类

——Parcel3.java

内
部
类

```
public class Parcel3 {  
    private class PContents extends 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(); }  
}
```

5.6 内部类

——Parcel3测试

内
部
类

```
class Test {  
    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 c = p.new PContents();  
    }  
}
```

- 说明

- 内部类PContents实现了抽象类Contents
- 内部类PDestination实现了接口Destination
- 外部类Test中不能声明对private的内部类的引用



5.6 内部类

——方法中的内部类

内部类

- 在方法内定义一个内部类
 - 为实现某个接口，产生并返回一个引用
 - 为解决一个复杂问题，需要建立一个类，而又不想它为外界所用



5.6 内部类

——Parcel4.java

内
部
类

```
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");  
        }  
    }  
}
```



5.6 内部类

——作用域中的内部类

内部类

```
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();  
        }  
    }  
    public void track() { internalTracking(true); }  
    public static void main(String[] args) {  
        Parcel5 p = new Parcel5();  
        p.track();  
    }  
}
```



5.6 内部类

——匿名的内部类

内
部
类

```
public class Parcel6 {  
    public Contents cont() {  
        return new Contents() {  
            private int i = 11;  
            public int value() { return i; }  
        };  
    }  
    public static void main(String[] args) {  
        Parcel6 p = new Parcel6();  
        Contents c = p.cont();  
    }  
}
```



5.6 内部类

——匿名的内部类

- 基类需要一个含参数的构造方法
- 例子

内部类

```
public class Parcel7 {  
    public Wrapping wrap(int x) {  
        return new Wrapping(x) {  
            public int value() { return super.value() * 47; }  
        };  
    }  
    public static void main(String[] args) {  
        Parcel7 p = new Parcel7();  
        Wrapping w = p.wrap(10);  
    }  
}
```



5.6 内部类

——匿名的内部类

- 匿名内部类对象的初始化
- 例子

内
部
类

```
public class Parcel8 {  
    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");  
    }  
}
```



5.6 内部类

——匿名的内部类

- 通过实例初始化构造匿名内部类
- 例子:

```
public class Parcel9 {  
    public Destination dest(final String dest, final float price) {  
        return new Destination() {  
            private int cost;  
            { cost = Math.round(price);  
              if(cost > 100)  
                  System.out.println("Over budget!");  
            } //不能重载  
            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);  
    }  
}
```

内
部
类



5.7 本章小结

- 本章内容

- 接口作用及语法
- 塑型的概念及应用
- 多态的概念及引用
- 构造方法的调用顺序及其中的多态方法
- 内部类的有关知识

- 本章要求

- 理解接口、塑型、多态的概念并能熟练应用
- 熟练掌握构造方法的调用顺序，理解编写时需要注意的问题
- 掌握内部类的语法结构及其应用场合

