内容提要

- 继承
- 终结类与终结方法
- 抽象类与抽象方法
- 接口
- 多态

接口

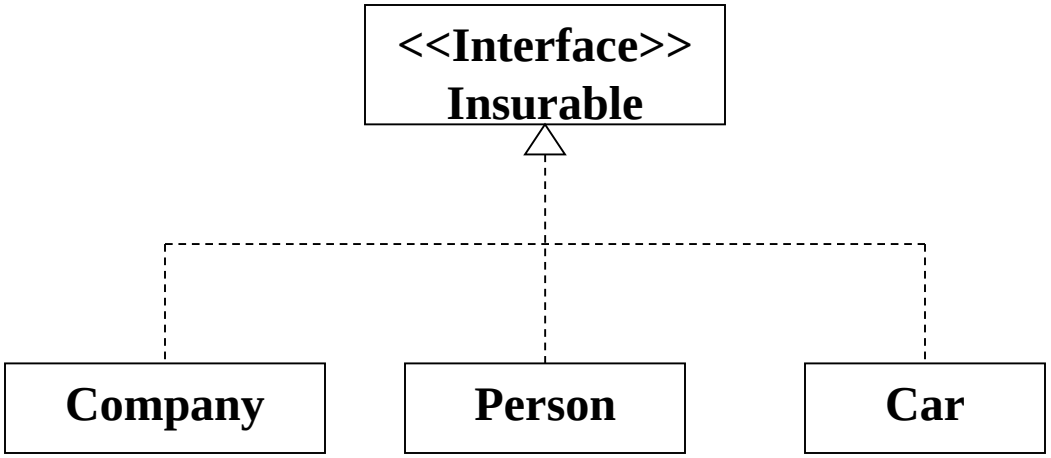
- 与抽象类一样都是定义多个类的共同属性
- 使抽象的概念更深入了一层，是一个“纯”抽象类，它只提供一种形式，并不提供实现
- 允许创建者规定方法的基本形式：方法名、参数列表以及返回类型，但不规定方法主体
- 也可以包含基本数据类型的数据成员，但它们都默认为static和final

接口

- 接口的作用
  - 是面向对象的一个重要机制
  - Java中的类只能有一个父类，但可以通过接口实现多继承，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（实现一种多态），隐藏实现细节
    - 实现不同类之间的常量共享

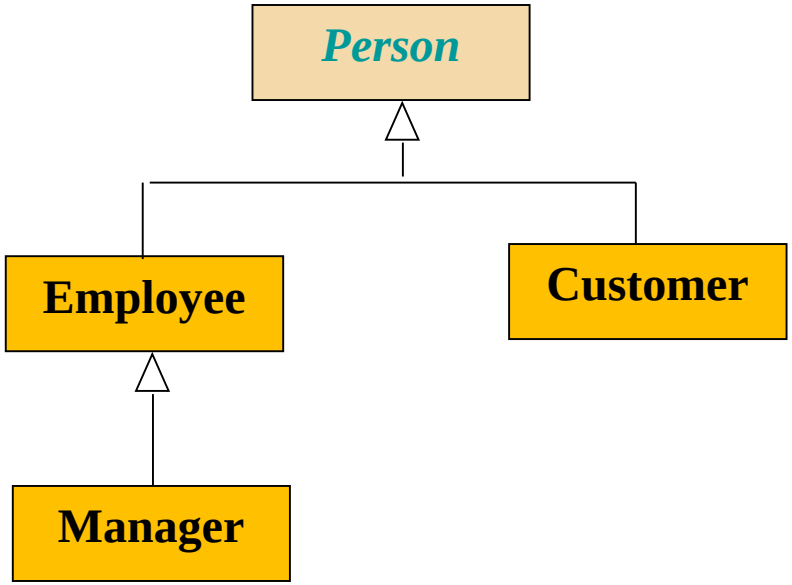
接口

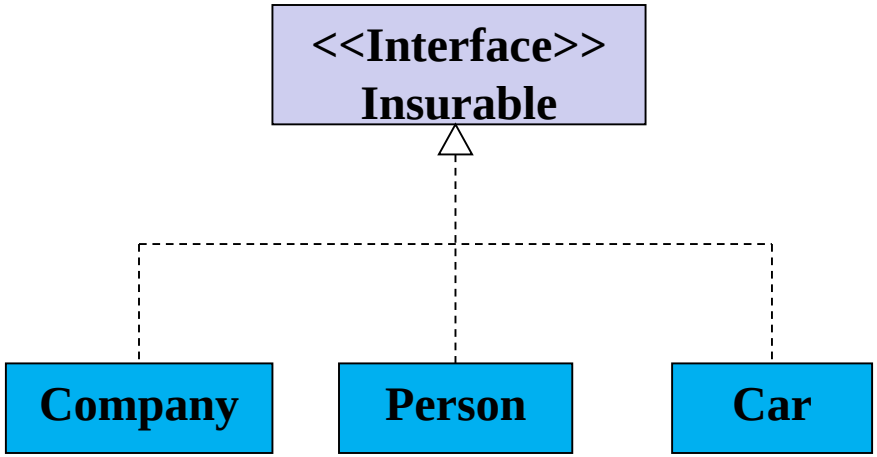
- 接口的作用
  - 保险公司的例子
    - 具有车辆保险、人员保险、公司保险等多种保险业务，在对外提供服务方面具有相似性，如都需要计算保险费(premium)等，因此可声明一个Insurable接口
    - 在UML图中，实现接口用带有空三角形的虚线表示



接口

- 接口与抽象类的异同
  - 接口本质上是一种特殊的抽象类，目的是用来实现多继承。接口与抽象类都是声明多个类的共同属性。
  - 它们的不同之处在于，接口允许看起来不相干的类之间定义共同行为。





接口：声明

- 接口的语法

[接口修饰符] interface 接口名称 [extends 父接口列表]{

公有的静态常量声明;

公有的抽象方法声明;

}

– 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public、final及static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

接口：声明

【例】保险接口的声明

- Insurable 接口声明如下，可见其中的方法都是抽象方法

```
public interface Insurable {
    public int getNumber();                  // 返回保险编号
    public int getCoverageAmount();  // 返回保险额度
    public double calculatePremium(); // 计算保险费用
    public Date getExpiryDate();       // 返回保险到期日
}
```

接口：声明

【例】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

```
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：

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

接口：实现

- 接口的实现
  - 接口不能用new运算符直接产生对象，必须利用其特性设计新的类，再用新类来创建对象
  - 利用接口设计类的过程，称为接口的实现，使用implements关键字

```
public class 类名称 implements 接口名称{
        接口方法的实现;
        类自己的数据成员和方法成员;
}
```

  - 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

接口：实现

【例】汽车类实现Insurable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
        }
    }
```

实现接口中的所有抽象方法

接口：实现

- 对象与接口之间的转型
  - 对象可以被转型为其所属类实现的接口类型（窄->宽）
  - 对象转换后接口也可以转回原类（宽->窄）
  - 例如：

```
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
```

接口：实现

【例】声明Circle与Rectangle两个类实现Shape2D接口

```
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);
      }
   }
```

接口：实现

【例】声明Circle与Rectangle两个类实现Shape2D接口

```
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
```

接口：实现

- 声明接口类型的变量，并用它来访问对象

```
public class InterfaceTester {
   public static void main(String  args[]){
      Shape2D cir, rect; //接口类型变量
      rect=new  Rectangle(5,6);
      System.out.println("Area of rect = " + rect.area());
      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

- 接口类型的变量作为参数传递

```
static void printArea(Shape2D shape){ // 接口类型的参数
        System.out.println("Area of shape = " + shape.area());
}
```

接口：实现

【例】为所有“可移动对象”能做的事情规定MovableObject接口，Plane、Car、 Train、 Boat类 分别实现 该接口。并为MovableObject接口安装遥控器(remote control)，它可以遥控Plane等可移动对象。

```
public interface  MovableObject { // 定义接口
   public boolean    start(); //启动，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
}
```

接口：实现

- Plane, Car, Train, Boat类分别实现MovableObject接口

```
public class Plane implements  MovableObject {
   public int   seatCapacity;
   public Company  owner;
   public Date  lastRepairDate;
   //实现MovalbleObject接口的所有方法
   public boolean start() {  //… }
   public void stop() {  //…  }
   public boolean turn(int degrees) {  //…  }
   public double fuelRemaining() {  //…  }
   public void changeSpeed(double kmPerHour) { // …  }
   //plane类自己的方法:
   public Date getLastRepairDate() {   //...   }
   public double calculateWindResistance() {  //.... }
}
```

接口：实现

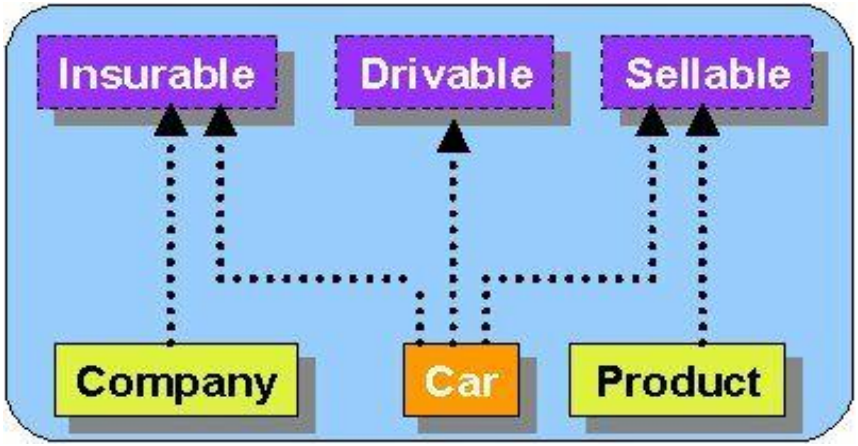
- 为MovableObject安装遥控器(remote control)，用于遥控某个可移动对象。

```
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接口，它可以是Plane, Car, Train, Boat等。

接口：用接口实现多继承

- Java的设计以简单实用为导向，不允许一个类有多个直接父类，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：用接口实现多继承

【例】声明Circle类实现接口Shape2D和Color

- Shape2D具有pi属性与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{                   //声明Color接口
   void setColor(String str);  //抽象方法
}
```

接口：用接口实现多继承

```
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);
   }
}
```

接口：用接口实现多继承

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

运行结果：

```
color=Blue
Area = 12.566370614359172
```

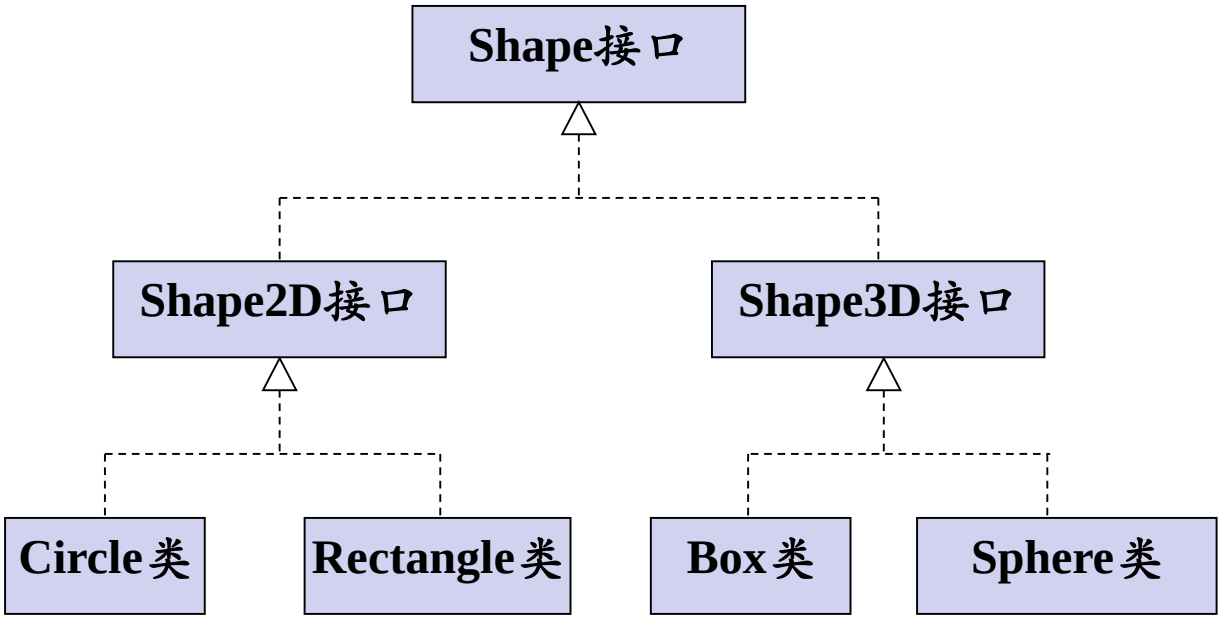
接口：接口的扩展

- 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问题的需要
- 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 extends 父接口1，父接口2，…
{ 加入新接口成员 }
```

接口：接口的扩展

【例】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：接口的扩展

【例】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

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

接口：接口的扩展

【例】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

```
class Circle implements Shape2D {
    double radius;
    String color;
    public Circle(double r) {   radius=r;  }
    public double area() {  //实现接口的父接口的方法
      return (pi*radius*radius);
     }
    public void setColor(String str){ //实现接口的方法
      color=str;
      System.out.println("color="+color);
    }
}
```

塑型(类型转换)

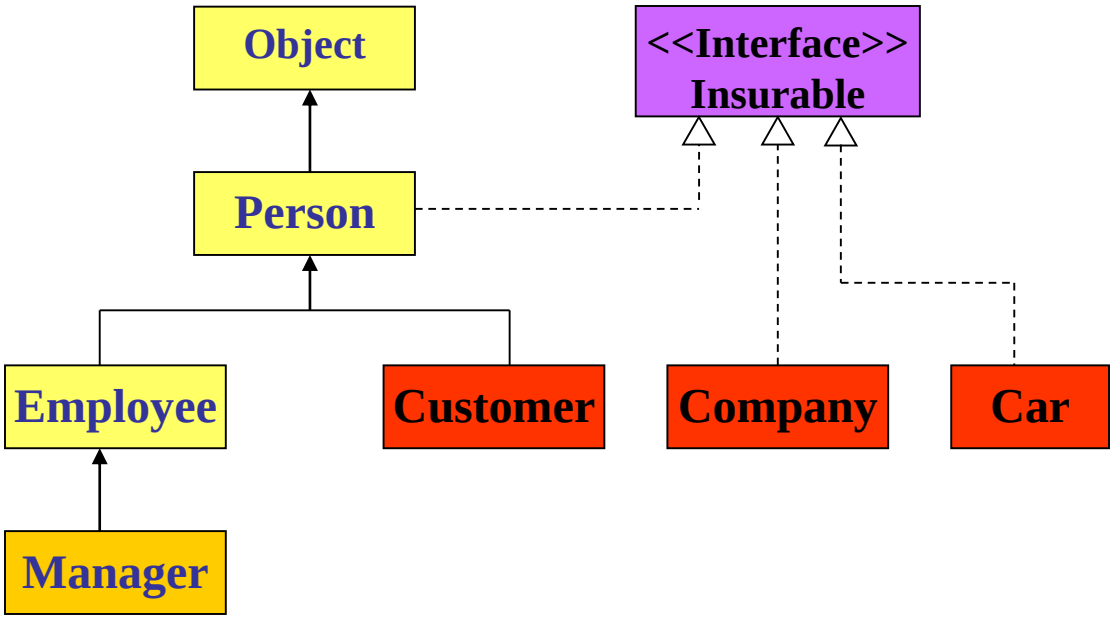
- 塑型的概念
  - 塑型(type-casting)，又称为类型转换
  - 方式
    - 隐式(自动)的类型转换
    - 显式(强制)的类型转换

塑型(类型转换)

- 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

塑型(类型转换)

- 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(类型转换)

- 隐式(自动)的类型转换
  - 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(); //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

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

塑型(类型转换)

- 显式(强制)的类型转换
  - 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：还原为本来的类型

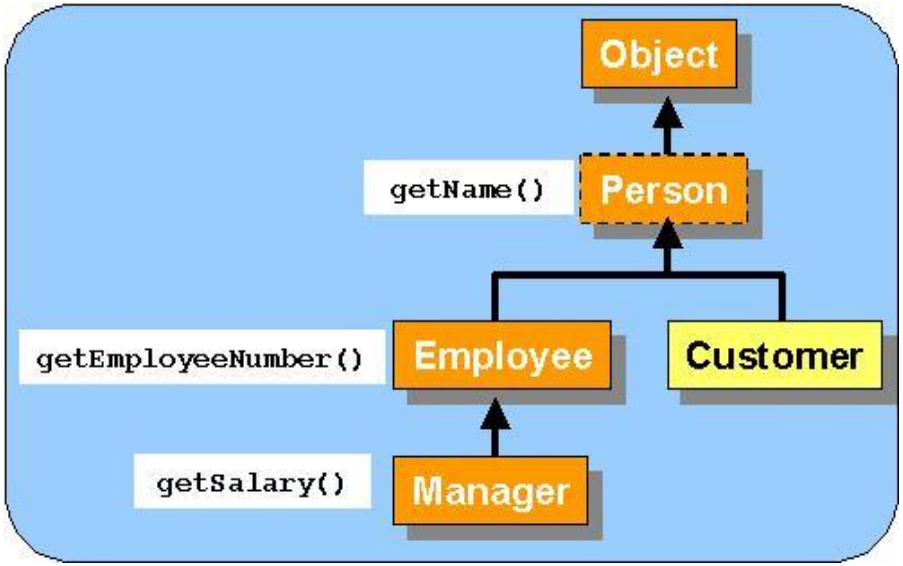
```
Employee  emp;
Manager man;
emp = new Manager();
man = (Manager)emp; //将emp强制塑型为本来的类型
```

塑型(类型转换)

- 塑型应用的场合，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
    - 算数混合运算时，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

塑型(类型转换)

- 当一个类对象被塑型为其父类后，它提供的方法会减少。例如：
  - 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



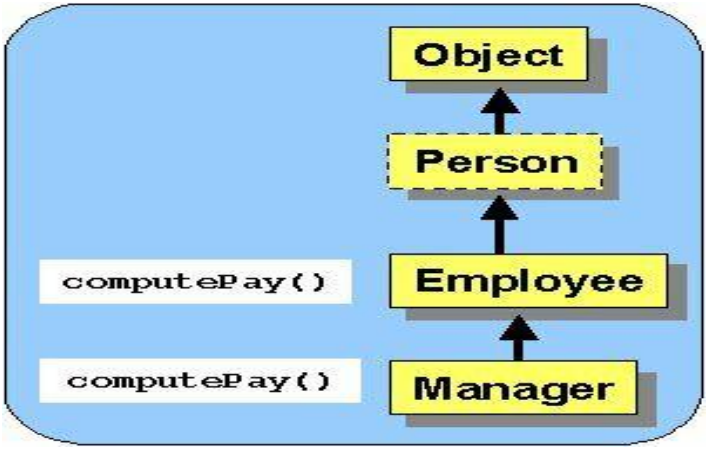
塑型(类型转换)

- 塑型后同名方法的查找
  - 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?
    - 实例方法的查找：从对象创建时的类开始，沿类层次向上查找
    - 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

塑型(类型转换)

【例】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

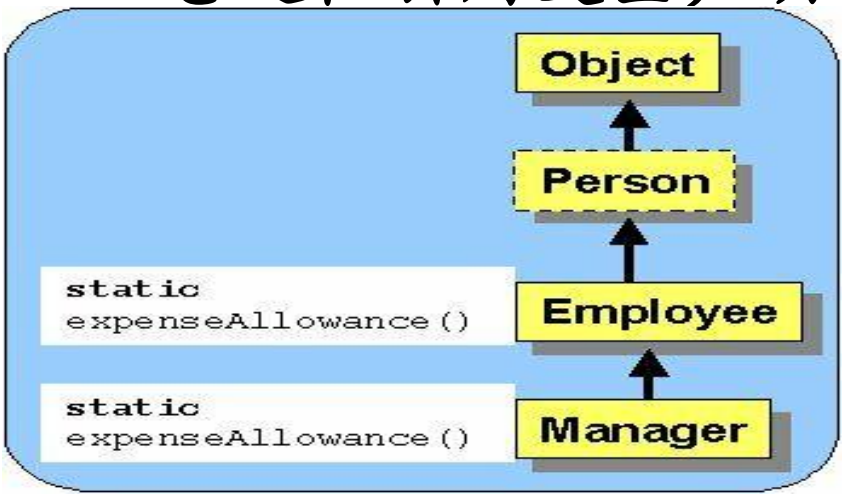



```
Manager  man = new Manager();
Employee emp1 = new Employee();
Employee emp2 =man;
emp1.computePay();  // 调用Employee类中的computePay()方法
man.computePay();   // 调用Manager类中的computePay()方法
emp2.computePay();  // 调用Manager类中的computePay()方法
```

塑型(类型转换)

【例】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.expenseAllowance();
Employee.expenseAllowance();
Employee.expenseAllowance();
```

但是静态方法的标准用用法是

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

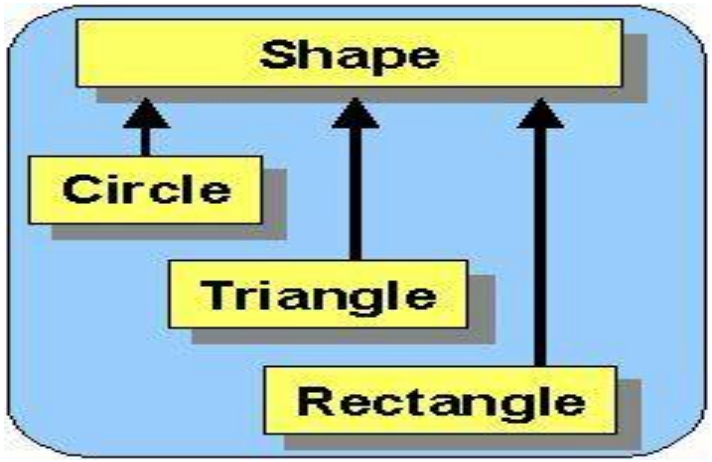
多态

- 多态是指不同类型的对象可以响应相同的消息
- 从相同的基类派生出来的多个类型可被当作同一种类型对待，可对这些不同的类型进行同样的处理，由于多态性，这些不同派生类对象响应同一方法时的行为是有所差别的
- 例如
  - 所有的Object类的对象都响应toString()方法
  - 所有的Shape2D的对象都响应area()方法
- 多态的目的
  - 所有的对象都可被塑型为相同的类型，响应相同的消息
  - 使代码变得简单且容易理解
  - 使程序具有很好的“扩展性”。例如新增一个Shape2D的子类，主程序代码几乎不用改变。

多态

- 一个例子：绘图——直接的方式
  - 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();
```

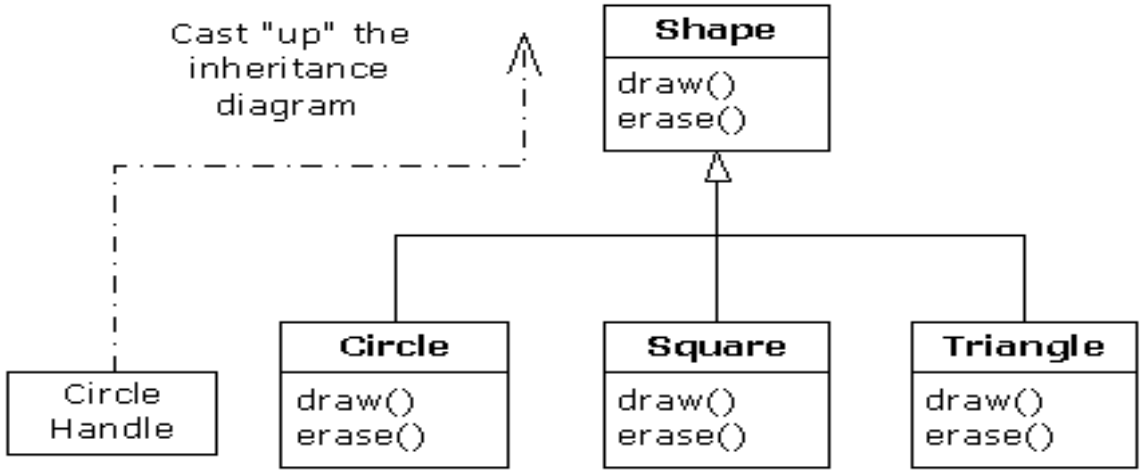


多态

- 一个例子：绘图——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（接下来将介绍绑定）



多态：绑定

- 绑定的概念
  - 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（即编译时）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

多态：绑定

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

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

  - 派生类覆盖了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()"); }
}
```

多态：绑定

- 仍以绘图为例，所有类都放在binding包中

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

多态：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();
    }
}
```

多态：绑定

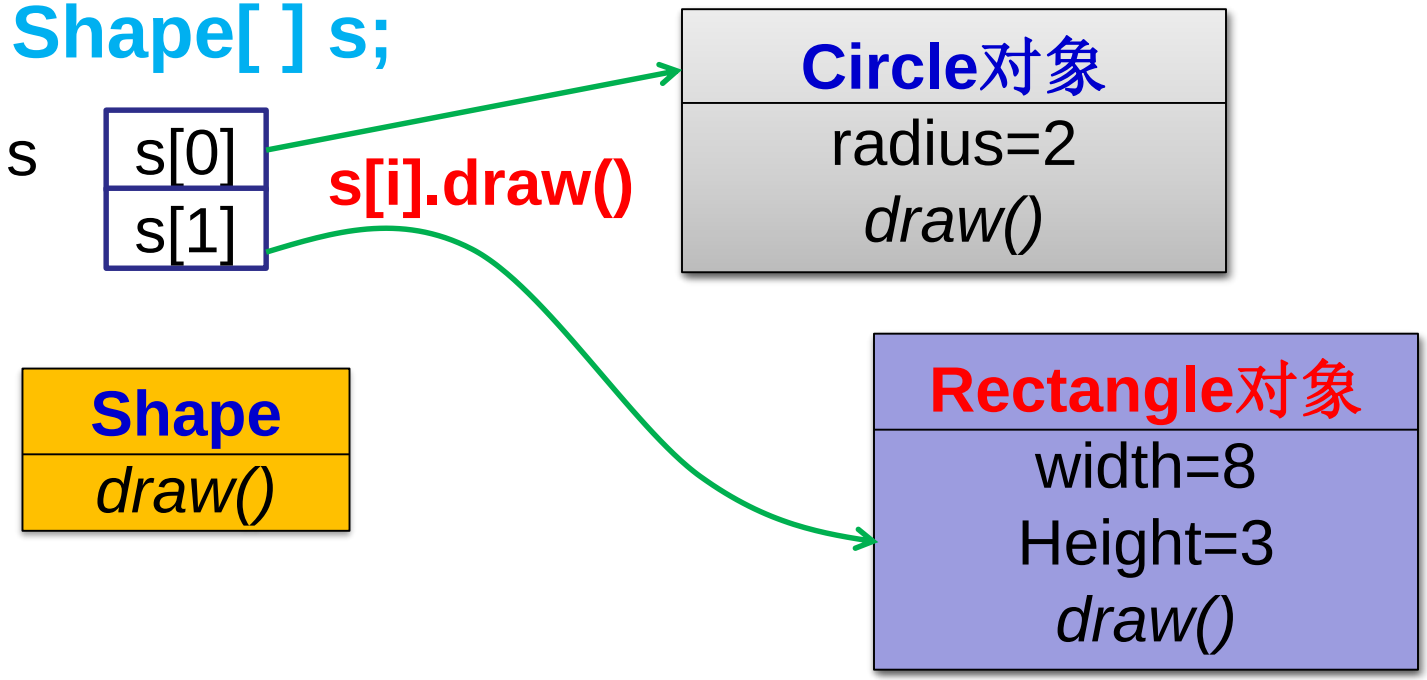
- 某次运行的结果：

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

- 说明
  - 编译时无法知道s数组元素的具体类型，运行时才能确定类型，所以是动态绑定
  - 在主方法的循环体中，每次随机生成指向一个Circle、Square或者Triangle的引用

多态：应用

- 多态应用的技术基础
  - 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：应用

- 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了重写

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

多态：应用

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

多态：应用

```
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.

多态：应用

- 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

多态：应用

- 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.
```

多态：应用

- 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.
```

多态：应用

- 一种似乎可行的方案：在驾驶员类的drives方法中判断交通工具的类型，输出相应的信息：

```
public class FemaleDriver extends Driver {
    public FemaleDriver( ) { }
    public void drives(Vehicle v ) {// 注：父类也相应改变
      if(v instanceof Car)
        __.println("A Female driver drives a car.");
      else if(v instanceof Bus)
        __.println("A Female driver drives a bus.");
       ……
      else
        __.println("A Female driver drives a boat.");
    }
}
```

但每增加一种新的交通工具，就要修改FemaleDriver类。因此，这种方案不利于软件的扩充，不是十分可取！

多态：应用

- 更好的方案：在交通工具类中提供两种方法，分别输出驾驶信息（该交通工具被男人驾驶或被女人驾驶）。
- 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();
}
```

多态：应用

- Vehicle及其子类声明如下：

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

多态：应用

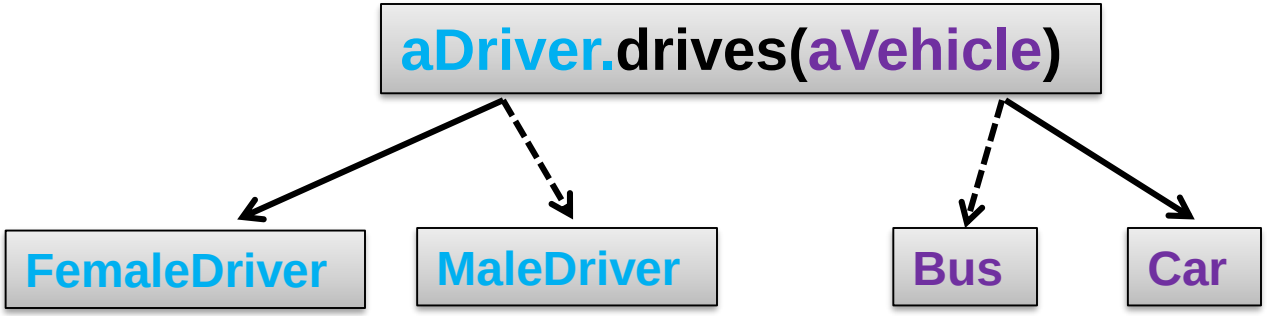
- 对抽象类Driver及子类FemaleDriver、MaleDriver类中的drives方法进行改进，在drives方法的定义体中不直接输出结果，而是调用Bus及Car类中的相应方法

```
public abstract class Driver {
    public Driver() { }
    public abstract void drives(Vehicle v ); // 传入Vehicle类型参数
}
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(); }
}
```

多态：应用

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给一个类
    - 之后根据交通工具的类型被发送给另一个类



增加新的交通工具时，无需修改驾驶人员类。

内部类

- 在另一个类或方法的定义中定义的类
- 可访问其外层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)

内部类

- 例如，定义一个包裹类，其中包含两个内部类

```
public class Parcel1 { // 包裹
    class Contents { //  内容--- 内部类
        private int i = 11;
        public int value() { return i; }
    }
    class Destination  { // 目的地---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());
    }
```

内部类

- 例如，定义一个包裹类，其中包含两个内部类

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

- 说明：
  - 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

内部类

- 外层类的方法可以返回内部类的引用变量

```
public class Parcel2{ // 包裹
   class Contents { //  内容--- 内部类
      private int i = 11;
      public int value() { return i; }
   }
   class Destination  { // 目的地---内部类
      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();  }
```

内部类

- 外层类的方法可以返回内部类的引用变量

```
public void ship(String dest) {
  Contents c = new Contents();
  Destination d = new Destination(dest);
  System.out.println(d.readLabel());
}
public static void main(String[] args) {
  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( )方法返回部内类Destination的引用
  - cont( )方法返回内部类Contents的引用

内部类：实现接口

- 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或者接口的一个引用
- 例子

```
abstract class AContents {
    abstract public int value();
}

interface IDestination {
    String readLabel();
}
```

内部类：实现接口

```
public class Parcel3 {
    private class PContents extends AContents {
        private int i = 11;
        public int value() { return i; }
    }
```

私有的内部类Pcontents对外不可见，但实现了抽象类Contents，因此其他类可以通过AContents变量引用它。

```
 protected class PDestination implements IDestination {
        private String label;
        private PDestination(String whereTo) { label = whereTo;}
        public String readLabel() { return label; }
    }
```

保护的内部类PDestination对外不可见，但实现了接口IDestination，因此其他类可以通过IDestination变量引用它。

```
    public IDestination dest(String s) { return new PDestination(s); }
    public AContents cont() { return new PContents(); }
}
```

内部类：实现接口

```
class Test {   // 外部的测试类
    public static void main(String[] args) {
        Parcel2 p2=new Parcel2();
       // 默认的Parcel2.Contents类在类Test中仍然可见
        Parcel2.Contents c2=p2.cont();
        System.out.println(c2.value());

       Parcel3 p3 = new Parcel3();
       // Parcel3.PContents c3 ;  Parcel3.PDestination d3;
       // 私有的、保护的内部类在类Test中不可见
       // 但通过基类、接口，可以引用它们

       IDestination d3 = p3.dest("Tanzania");
       System.out.println(d3.readLabel());
       AContents c3=p3.cont();
       System.out.println(c3.value());   }
}
```

内部类：方法中的内部类

- 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

内部类：方法中的内部类

```
public class Parcel4 {
    public IDestination dest(String s) {
        class PDestination implements IDestination {
            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();
        IDestination d = p.dest("Tanzania");
        // PDestination d2; // PDestination 无法解析为类型
        __.println(d.readLabel()); //通过接口访问内部类
    }
}
```

内部类：匿名的内部类

- 匿名类的使用场合：
  - 在创建对象时只使用一次，且要产生的新类须继承一个父类或实现一个接口时
  - GUI程序处理事件时多用
- 特点：
  - 本身没有名称，因此不存在构造方法
  - 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
- 声明语法：

  new 父类名 或 接口名 ()

  { 类体 }

内部类：匿名的内部类

- 匿名类示例：分别基于继承的方式和实现接口的方式

```
public class FatherClass { //定义了一个父类
    void showFC() {
        System.out.println("调用了FatherClass的方法！");
    }
}

public interface MyInterface { // 定义了一个接口
    void showMI();
}
```

内部类：匿名的内部类

```
public class AnonymityClassTester {
    public static void main(String[] args) {
        FatherClass f;
        f=new FatherClass(){
            void showFC(){System.out.println("调用了匿名子类重写父类FatherClass的方法！");}
        };
        f.showFC(); //父类引用调用匿名子类重写的showFC()方法

        MyInterface mi;
        mi=new MyInterface(){
            public void showMI(){System.out.println("调用了匿名类实现接口MyInterface的方法！");}
        };
        mi.showMI(); //接口引用调用匿名类实现的showMI()方法
        }
}
```

内部类：匿名的内部类

- 示例：GUI程序中处理事件时常用匿名类

```
//WindowAdapter为接收窗口事件的抽象适配器类。此类中的方法为空。此类存在的目的是方便创建侦听器对象。
// 给窗体添加监听器
addWindowListener(  new WindowAdapter( ) {
                        public void windowClosing(WindowEvent e){
                           System.exit(0);     }
                    }
                );
```

```
addWindowListener( new cls() );
class cls extends WindowAdapter {
        public void windowClosing(WindowEvent e){
            System.exit(0);   }
}
```

习 题

1. 设计图形类Shape及其子类：
   ① Shape类具有表示图形中心坐标的受保护的centerX和centerY属性，且规定其子类要具有计算并返回图形面积的方法area。
   ② 其子类包括圆类Circle和矩形类Rectangle。
   ③ 为这些类设计恰当的属性、构造方法和set/get方法。
   ④ 对你设计的类进行测试，其中包括创建若干个圆和矩形对象，然后计算它们的总面积。（提示：创建Shape类型的数组s，数组元素可以引用Shape类的子类对象，例如s[0]=new Circle(5.0)或s[0]=new Circle(10.0, 20.0, 5.0), 计算面积则调用s[0].area( )）
2. 在上一题的基础上，定义一个接口IPrint：
   ① 它只有一个方法display( )，用于显示图形信息。
   ② Circle和Rectangle类实现该接口。
3. 创建一个窗体，为窗体添加窗体监听器（使用内部类）。

谢谢大家！