

# Java软件设计基础



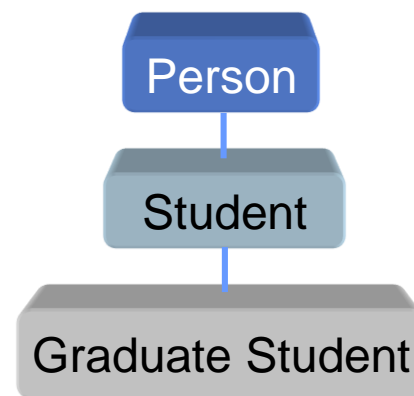


JAVA™

## 5. 类的继承

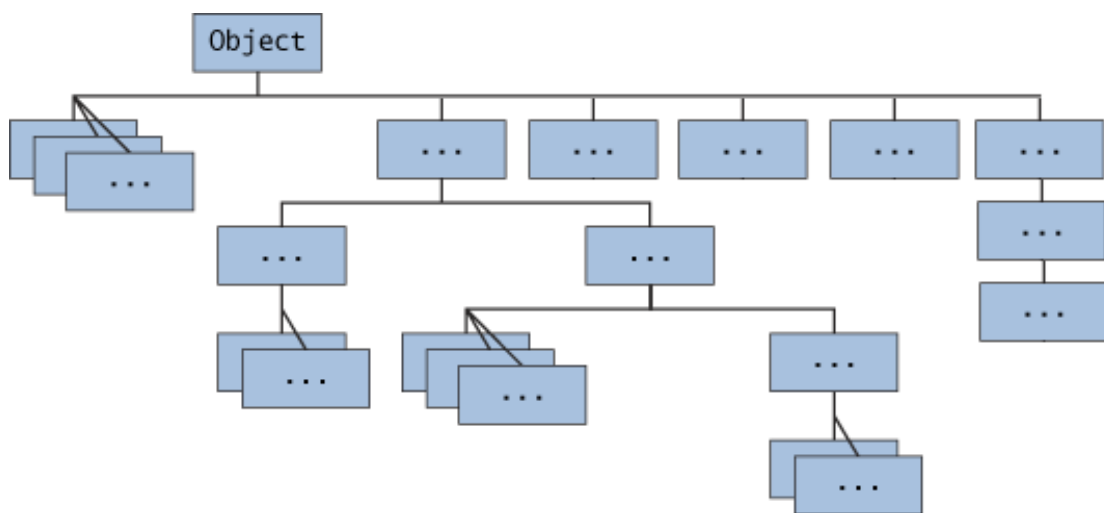
## 5.1 继承机制

- 继承是面向对象编程技术的一块基石。
  - 继承能够创建一个通用类，并定义一系列相关项目的一般特性。该类可以被更具体的类继承。
  - 每个具体的类都增加一些自己特有的东西。
- 继承是一个对象获得另一个对象的属性的过程。
  - 由继承得到的类为子类(subclass 或 childclass)，也被称为派生类(derived class)、扩展类(extended class)。
  - 被继承的类为超类(superclass)，也被称为基类(base class)或者父类(parent class)，包括所有直接或间接被继承的类；



- 类可以从另一个类的子类的子类的子类……继承而来，最终都是继承自最顶级的类Object。
- 定义在Object中的方法可被所有子类使用。

Java平台类层次结构：所有类都是Object类的后裔

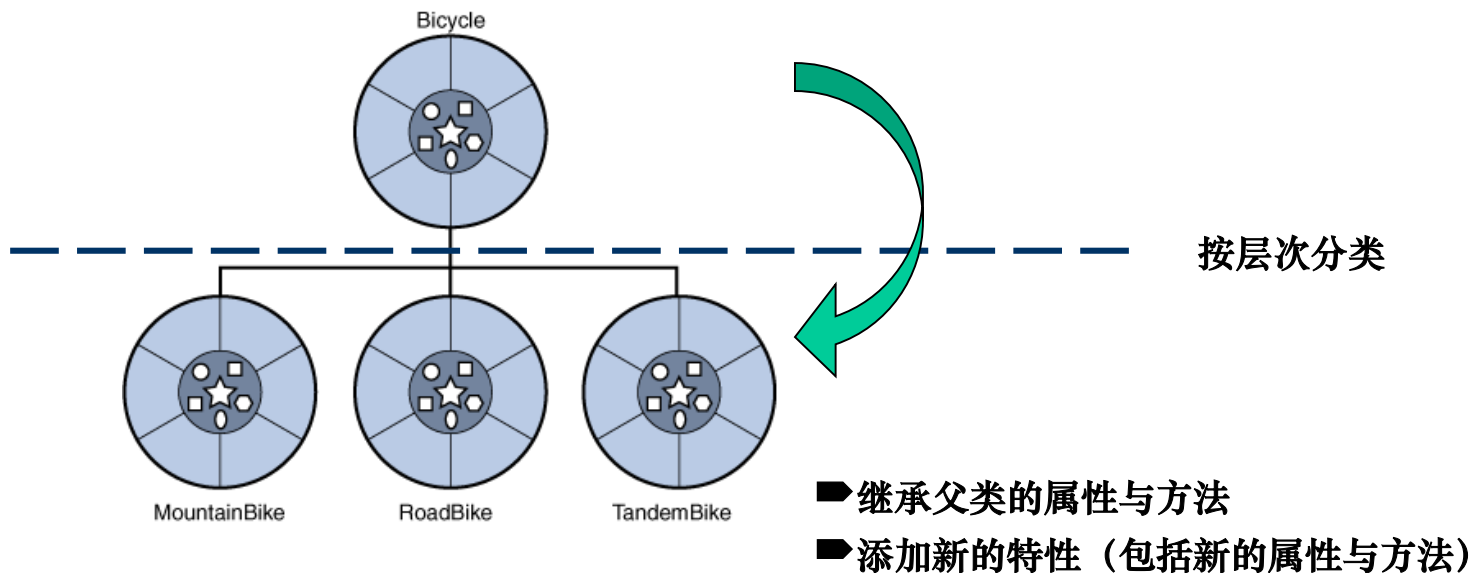


- 简单的说，类的继承性就是新的子类可以从另一个父类派生出来，并自动拥有父类的全部属性和方法。
- 子类继承父类的成员变量与方法，就如同在子类中直接声明的一样，可以被子类中自己声明的任何实例方法所调用。
- 子类继承父类的状态和行为，同时也可以修改继承自父类的状态和行为，并添加新的状态和行为。通俗的说法：子类更具有特殊性。

继承用来为**is-a**关系建模。不要为了重用方法而盲目的派生一个类。例如，从**Person**类派生**Tree**类毫无意义。子类和父类之间必须存在**is-a**关系。

**Java**采用单继承的特征，使得代码更加可靠，不会出现因多个父类有相同的方法或属性所带来的麻烦。

- 继承支持按层次分类的概念。



- 声明格式

[修饰符] class 子类名 extends 父类名 {类体}

- 例程：自行车类
- 自行车类作为父类

三个属性：踏板步调、速度、档位

构造方法

提供四个方法：  
设置踏板步调；  
设置档位；  
刹车减速；  
提速；

父类Bicycle

```
class Bicycle{  
    public int cadence;  
    public int speed;  
    public int gear;  
    public Bicycle(){ }  
    public Bicycle(int sC,int sS,int sG){  
        gear=sG;  
        speed=sS;  
        cadence=sC;  
    }  
    public void setCadence(int newValue){  
        cadence=newValue;  
    }  
    public void setGear(int newValue){  
        gear=newValue;  
    }  
    public void applyBrake(int decrement){  
        speed-=decrement;  
    }  
    public void speedUp(int increment){  
        speed+=increment;  
    }  
}
```

- 自行车下面还有许多子类，如山地车、公路自行车等。
- 以山地车为例

子类MountainBike

```
class MountainBike extends Bicycle{  
    public int seatHeight;  
    public MountainBike(int sC,int sS,int sG,int sSH){  
        gear=sG;  
        speed=sS;  
        cadence=sC;  
        seatHeight=sSH;  
    }  
    public void setHeight(int newValue){  
        seatHeight=newValue;  
    }  
}
```

除了拥有父类的三个属性，还有座高这一特有的属性

新增的方法：设置座高



- 说明

- 子类能够继承父类中public和protected成员变量和方法;
- 如果子类和父类在同一个包内，子类能继承父类中用默认修饰符修饰的成员变量和方法;
- 一般情况下，子类不能继承父类中的private成员变量和方法;
- 若缺省extends子句，则该类为java.lang.Object的子类。

#### 父类

```
class SuperClass
{
    public int a=1;
    protected int b=2;
    private int c=3;
    int d=4;
}
```

#### 子类

```
class SubClass extends SuperClass
{
    public void main(String args[])
    {
        SubClass o1=new SubClass();
        o1.a=1;
        o1.b=2;
        o1.c=3;
        o1.d=4;
    }
}
```

⊗ c has private access in SuperClass

## 5.2 子类的构造方法

- 构造方法

- 与属性和方法不同，子类并不“继承”父类的构造方法，只能从子类的构造方法中通过关键字super调用父类构造方法的**初始化过程**。

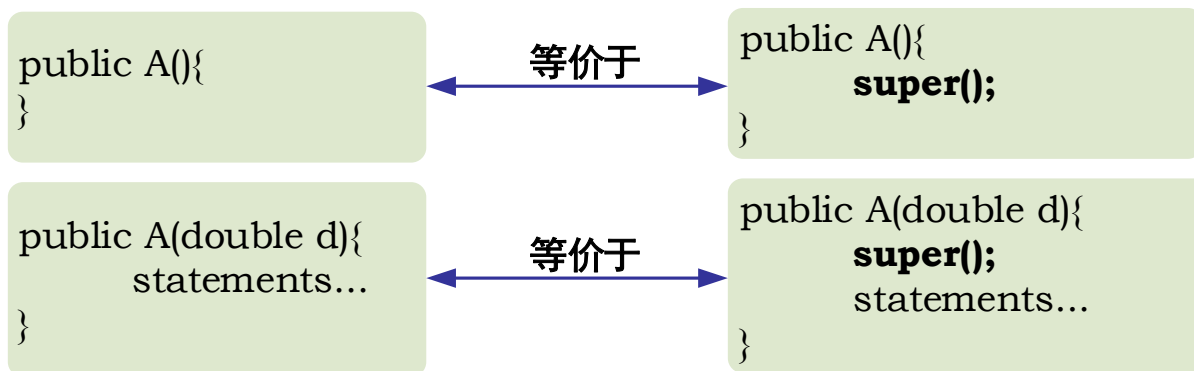
调用父类的构造方法必须使用关键字**super**，并且这个调用必须是构造方法的第一条语句。

- 例如：山地自行车的构造方法也可写成如下形式

构造方法

```
public MountainBike(int sC,int sS,int sG,int sSH){  
    super(sC,sS,sG);  
    seatHeight=sSH;  
}
```

- 调用子类的构造方法时将会沿着继承链调用所有父类的构造方法，父类的构造方法总会在子类的构造方法之前执行……以此类推，创建任何Java对象，最先执行的总是Object类的构造方法。



### 构造方法链例程

```
public class Faculty extends Employee{
    public static void main(String args[]){
        new Faculty();
    }
    public Faculty(){
        System.out.println("类Faculty的无参构造方法");
    }
}
class Employee extends Person{
    public Employee(){
        System.out.println("类Employee的无参构造方法");
    }
}
class Person{
    public Person(){
        System.out.println("类Person的无参构造方法");
    }
}
```

### 例程输出结果

类Person的无参构造方法  
类Employee的无参构造方法  
类Faculty的无参构造方法

- 根据构造方法链的原理，子类在实例化时，首先调用父类构造函数，实例化父类。之后才是子类自身实例化。
- 这就是为什么上例中会首先打印“类Person的无参构造方法”的原因。
- 将程序片段略作修改：

```
class Person{  
    public Person(String name){  
        System.out.println(name);  
    }  
}
```

当运行主程序的“new Faculty();”语句时，上述程序会报错，原因在于子类实例化之前要调用父类中的无参数构造方法，而间接父类Person中不存在无参数构造方法。

也可以使用super语句显式的调用父类的带参数构造方法来解决上述错误

```
class Employee{  
    public Employee( ){  
        super(“显式调用”);  
    }  
}
```

如果一个类要生成其他子类，最好提供一个无参数的构造方法以避免编程错误。

## 5.3 隐藏 · 覆盖 · 重载

- 如果子类定义了和父类同名的实例变量，则称之为“隐藏”。
- 在正常情况下，子类里定义的方法直接访问该实例变量的时候会默认访问子类中定义的实例变量，而无法访问到父类中被隐藏的实例变量。

### 实例变量覆盖

```
class BaseClass{
    public int a=5;
}
public class SubClass extends BaseClass{
    public int a=7;
    public void accessOwner(){
        System.out.println(a);
    }
    public static void main(String[] args){
        SubClass sc=new SubClass();
        sc.accessOwner();
    }
}
```

此时只能访问到子类中定义的实例变量，打印结果为：7

- 如果想访问到父类定义的实例变量，可使用关键字super

#### 实例变量覆盖

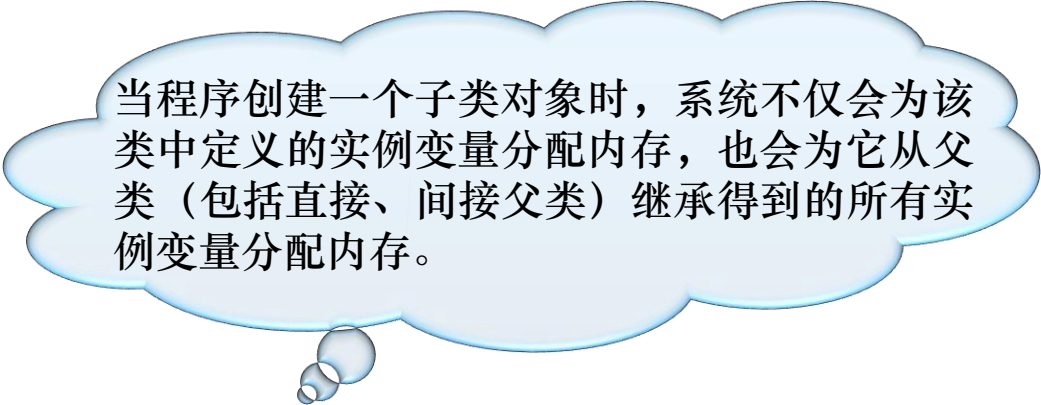
```
class BaseClass{
    public int a=5;
}
public class SubClass extends BaseClass{
    public int a=7;
    public void accessOwner(){
        System.out.println(a);
    }
    public void accessBase(){
        System.out.println(super.a);
    }
    public static void main(String[] args){
        SubClass sc=new SubClass();
        sc.accessOwner();
        sc.accessBase();
    }
}
```

如果在某个方法中访问名为a的成员变量，但没有显式指定调用者，则系统查找a的顺序为：

>>查找该方法中是否有名为a的局部变量；

>>查找当前类中是否包含名为a的成员变量；

>>查找a的直接父类是否包含名为a的成员变量，依次上溯其所有父类直至Object类，如果最终不能找到名为a的成员变量，则系统出现编程错误。



当程序创建一个子类对象时，系统不仅会为该类中定义的实例变量分配内存，也会为它从父类（包括直接、间接父类）继承得到的所有实例变量分配内存。



- 如果子类定义了和父类同名的方法以及形参列表，则称之为“覆盖”或“重写” (override)。

通过方法覆盖，子类可以重新实现父类的某些实例方法，使其具有自己的特征。

- 方法的重写需要遵循以下规则：
  - 方法名相同；形参列表相同；
  - 返回值类型应该相同或者比父类方法返回值类型范畴更小；
  - 如果有抛出异常的声明应该比父类方法声明抛出的异常类型范畴更小或相同；
  - 子类方法的访问权限应该比父类方法访问权限更大或相等；
  - 覆盖方法与被覆盖的方法要么同为实例方法，要么同为类方法

## 方法覆盖

```
class Bird{
    public void fly(){
        System.out.println("飞啊飞啊");
    }
}
public class Ostrich extends Bird{
    public void fly(){
        System.out.println("不会飞，跑啊跑啊");
    }
    public static void main(String[] args){
        Ostrich os=new Ostrich();
        os.fly();
    }
}
```

run:

不会飞，跑啊跑啊

成功构建 (总时间: 0 秒)

## 方法的覆盖

```
public class Example{
    public static void main(String args[]){
        B b=new B();
        double d=b.area();
    }
}
class A{
    protected int x=10,y=12;
    public int area(){
        return x*y;
    }
}
class B extends A{
    public double area(){
        return 3.14*x*y/4;
    }
}
```

⊘ area() in B cannot override area() in A;  
attempting to use incompatible return  
type

## 方法的覆盖

```
public class Example{
    public static void main(String args[]){
        B b=new B();
        double d=b.area();
        System.out.println(d);
    }
}
class A{
    protected int r=10;
    public double area(){
        return r*r;
    }
}
class B extends A{
    protected double area(){
        return 3.14*r*r;
    }
}
```

⊗ area() in B cannot override area() in A;  
attempting to assign weaker access  
privileges was public

- 被覆盖的父类方法可通过关键字super访问。
- 如果某个父类方法被声明为private:
  - 该方法对其子类是隐藏的，子类无法访问该方法，即无法覆盖该方法。
  - 但子类中仍然可以定义与其同名、同参数列表、同返回值的方法，但这只是重新定义了一个新的方法，并非覆盖；
- 终止覆盖
  - 为了防止利用属于系统重要信息的类来创建子类并进行方法覆盖，替换原有的类攻击系统，防止父类被覆盖，引入了终止覆盖的理念，即final修饰符外延性的使用。
  - 用final修饰符修饰父类中的方法可以避免子类来覆盖此方法。

- 重载(override)

- 一个类中可以有多多个具有相同名字的方法，由传递给它们的**不同个数和类型的参数**来决定使用哪种方法；通过重载可以定义同类的操作方法(行为)，而不需要取很多个类似又容易混淆的名字。
- 重载方法需要满足以下条件：
  - 方法名相同；
  - 方法的参数签名（即参数类型、个数、顺序）不相同；
  - 方法的返回类型可以不相同，方法的修饰符也可以不相同；
  - 不能根据方法返回类型的不同来区分重载的方法。

java.lang.Math类中的max()方法

```
public static int max(int a,int b)
```

```
public static long max(long a,long b)
```

```
public static float max(float a,float b)
```

```
public static double max(double a,double b)
```

### 主方法

```
public class MethodOverloadTest{
    public static void main(String args[]){
        MethodOverload mo=new MethodOverload();
        mo.receive(1);
        mo.receive(2,3);
        mo.receive(12.45);
        mo.receive("this is a String");
    }
}

class MethodOverload{
    void receive(int i){
        System.out.println("Receive an integer:"+i);
    }
    void receive(int i,int j){
        System.out.println("Receive two integer:"+i+","+j);
    }
    void receive(double d){
        System.out.println("Receive a double data:"+d);
    }
    void receive(String s){
        System.out.println("Receive a string:"+s);
    }
}
```

### 例程输出结果

Receive an integer:1  
Receive two integer:2,3  
Receive a double data:12.45  
Receive a string:this is a String

## 5.4 多态(Polymorphism)

- Java引用变量有两个类型，一个是编译时类型，一个是运行时类型。编译时类型由声明该变量时使用的类型决定，运行时类型由实际赋给该变量的对象决定。
- 如果编译时类型和运行时类型不一致，就可能出现多态。

多态

```
class Shape{
    String name="Shape";
    public void test(){ System.out.println("形状"); }
}
class Rectangle extends Shape{
    String name="Rectangle";
    public void test(){ System.out.println("矩形"); }
    public static void main(String[ ] args){
        Shape s=new Shape();
        Rectangle r1=new Rectangle();
        Shape r2=new Rectangle();
        r2.test();
        System.out.println(r2.name);
    }
}
```

编译时类型: **Shape**

运行时类型: **Rectangle**

在实际运行的时候执行的是子类中的test()方法



- Java允许把一个子类对象直接赋给一个父类的引用变量，无需任何类型转换，或者称为向上转型（upcasting），是由系统自动完成的。
- 上述例子中，在运行的时候其**方法行为**总是表现出子类Rectangle的行为特征，而不是父类Shape的行为特征。

• 引用变量在编译阶段只能调用其编译时类型所具有的方法（虽然其运行是按照运行时类型来调用的），这也意味着，如果编译阶段调用的方法并不是编译时类型所具有的，那么编译就无法通过。

如上例中如果在Rectangle子类中定义area()方法，而程序中出现r2.area()代码，则编译会报错“找不到符号”。

• 与方法不同的是，对象的实例变量则不具备多态性。

如上例中r2输出它的实例变量name时，并不是输出Rectangle类里定义的实例变量，而是Shape类的实例变量。

## 多态

```
class Shape{
    String name="Shape";
    public void test(){
        System.out.println("形状");
    }
}
class Rectangle extends Shape{
    int w,h;
    String name="Rectangle";
    public void test(){
        System.out.println("矩形");
    }
    public void area(){
        System.out.println(w*h);
    }
    public static void main(String[ ] args){
        Shape s=new Shape();
        Rectangle r1=new Rectangle();
        Shape r2=new Rectangle();
        r2.test();
        //r2.area();
        System.out.println(r2.name);
    }
}
```

```
run:
矩形
Shape
```

- 引用类型的强制类型转换

- 如果在编译时需要引用类型可以调用它运行时类型的方法，则需要借助类型转换运算符把它强制转换成运行时类型。

(type)variableName

- 引用类型的转换只能在具有继承关系的两个类型之间进行，否则编译会报错；
- 如果试图将父类实例转换为子类类型，则该对象必须是该子类的实例，否则将引发ClassCastException异常。

类型转换

```
public class ConversionTest{  
    public static void main(String[ ] args){  
        Object obj1=new String("abc");  
        String s1=(String)obj1; //正确  
        Object obj2=new Rectangle();  
        String s2=(String)obj2; //报错  
    }  
}
```

- instanceof操作符

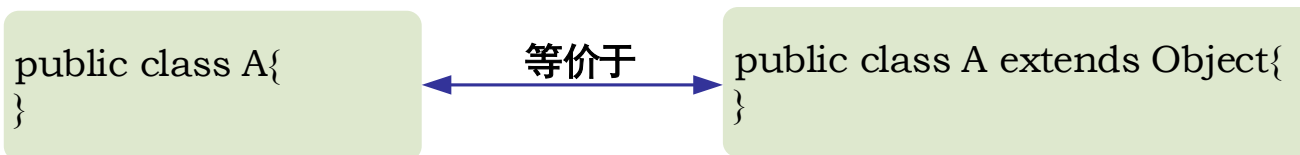
- 考虑到进行强制类型转换时可能出现异常，因此在转换前可通过 instanceof运算符来判断是否可以成功转换，例如将上述语句改为：

```
if(obj2 instanceof String)
    String s2=(String)obj2;
```

- instanceof操作符的前一个操作数通常是某个引用类型变量，后一个操作数通常是一个类（或接口），用于判断前面的对象是否是后面的类的实例，如果是，则返回true，否则返回false。

## 5.5 一个经常被覆盖的方法toString

- Java中的每个类都源于java.lang.Object类。如果一个类在定义时没有指定继承，它的父类默认是Object。



- toString()方法
  - 调用对象的toString()方法返回一个代表该对象的字符串。
  - 默认情况下，返回一个由该对象所属的类名、@和该对象十六进制的散列码组成的字符串。

toString方法

```
public class Faculty{
    public static void main(String args[]){
        Faculty f = new Faculty();
        System.out.println(f.toString());
    }
}
```

例程输出结果

Faculty@de6ced

- 在实际编写代码中常常覆盖toString()方法，使它返回一个代表该对象的易懂的字符串。

toString方法

```
public String toString(){  
    return 与该对象有关的信息;  
}
```

也可以传递对象来调用：

**System.out.println(object)**

**System.out.print(object)**

这等价于调用：

**System.out.println(object.toString())**

**System.out.print(object.toString())**

## 5.6 抽象类

- 类的前面加上abstract修饰符的即为抽象类。
- 说明
  - 抽象类和常规类一样具有成员变量、方法(普通方法和抽象方法)、构造方法；
  - 抽象方法只有方法头而没有实现，它的实现由子类提供。
  - 子类可以是抽象的，即使它的父类是具体的。

非抽象类不能包含抽象方法，如果一个抽象父类的子类不能实现所有的抽象方法，它必须声明为抽象的。

抽象类不能用new操作符实例化，其定义的构造方法可在子类的构造方法中调用。

包含抽象方法的类必须是抽象的。但是允许声明没有抽象方法的抽象类。

- 例程：形状类

### 形状类

```
public class Shape{  
    private String color = "white";  
    public String getColor(){  
        return color;  
    }  
    public void setColor(String color){  
        this.color=color;  
    }  
    public double getArea(){  
        .....  
    }  
    public double getPerimeter(){  
        .....  
    }  
}
```

在某些情况下，父类只是知道其子类应该包括怎样的方法，但无法准确的知道如何实现

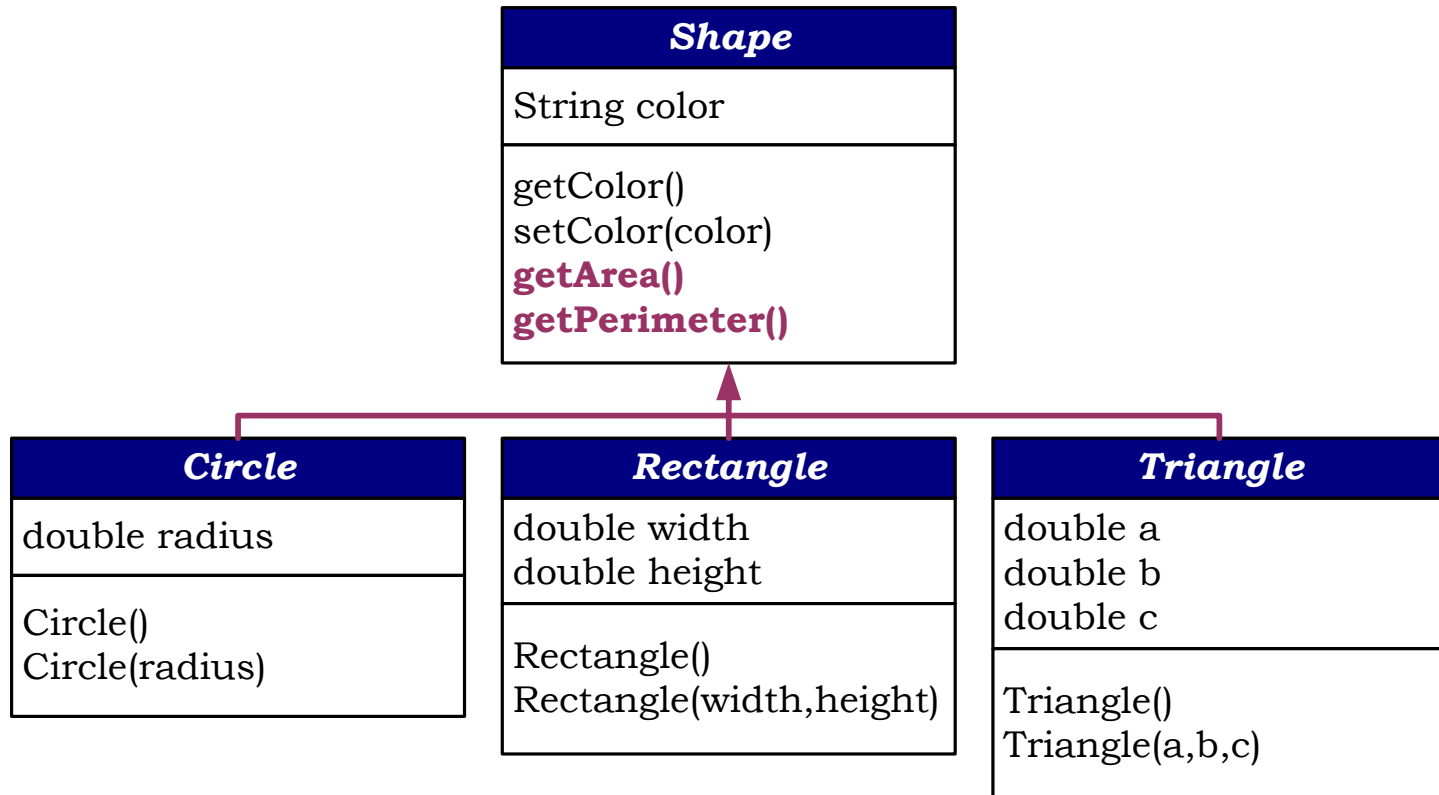


- 当涉及计算面积及周长时，其方法的实现取决于几何对象的具体类型。
- 当然，可也将所有的形状归为一类，以方法的重载来计算形状的面积及周长，但这样使得类的特征不够明显。
- 可将getArea和getPerimeter方法定义为抽象方法，这些方法将在子类中实现。

#### 形状类

```
public abstract class Shape{  
    private String color = "white";  
    public String getColor(){  
        return color;  
    }  
    public void setColor(String color){  
        this.color=color;  
    }  
    public abstract double getArea();  
    public abstract double getPerimeter();  
}
```

- 子类继承抽象父类，增加自己特有的属性，并实现抽象父类的抽象方法。



- 具体实例的创建

对象创建

```
Shape s1 = new Circle(5);  
Shape s2 = new Rectangle(10,8);
```

- 创建了一个新的圆形与新的矩形，并把它们赋值给变量s1和s2，这两个变量的编译时类型是Shape。
- 当实际运行时，使用s1.getArea() 会自动根据其运行时类型调用Circle类中实现的getArea方法——**这就是之前说过的编译时类型与运行时类型**。

抽象类具有更高层次的抽象，从多个具有相同特征的类中抽象出一个抽象类，以这个抽象类作为其子类的模板，从而避免子类涉及的随意性。子类在抽象类的基础上进行扩展、改造，但子类总体上会大致保留抽象类的行为方式。

## 5.7 接口 interface

- 对象通过暴露的方法定义它们和外部世界的交互，方法形成对象和外部世界的接口。
- 通过实现接口，类可以更加正式的说明它承诺提供的行为。接口构成了类和外部世界之间的**契约**，并且该契约在编译时由编译器强制实施。
- 如果类声明实现一个接口，那么这个接口定义的所有方法都必须出现在其源代码中，否则就不能成功编译。
- 接口不是类层次结构的一部分，但它们与类结合在一起工作。Java通过接口使得处于不同层次、甚至互不相关的类可以具有相同的行为（即实现相同的接口）。

接口不提供任何实现，只定义多个类共同的公共行为规范，这些行为是与外部交流的通道，即意味着接口里通常是一组公共方法的定义。  
一个Java源文件里最多只能有一个public接口（或类）

## • 接口的声明

```
[修饰符] interface 接口名 [extends 父接口列表]{  
    常量定义;  
    抽象方法定义;  
}
```

- 除此之外，接口中还可包含嵌套类型的定义，从Java8开始，可定义默认方法、类方法；
- 接口不能被实例化，只能被类实现，或者被其他接口扩展；
- 接口中所有成员都是public访问权限（可省略该修饰符）；
- 对于常量来说，系统还会为其自动增加static和final修饰符；
- 对于抽象方法，系统默认都采用abstract修饰符。

## 接口范例

```
public interface Output{
```

```
    int MAX=50;
```

等同于使用public static final修饰

```
    void out();
```

```
    void getData(String msg);
```

抽象方法，等同于使用public abstract修饰

```
    default void print(String... msgs){
```

默认方法，等同于public修饰

```
        for(String msg:msgs){
```

```
            System.out.println(msg);
```

```
        }
```

```
    }
```

```
    static String staticTest(){
```

类方法，等同于public修饰

```
        return "static method";
```

```
    }
```

```
}
```

接口支持多继承，子接口会获得父接口里定义的所有抽象方法、常量。

- 接口可用于声明引用类型变量，当使用接口来声明引用类型变量时，该引用类型变量必须指向到该接口实现类的对象。
- 接口的最主要用途就是被实现类实现，实现接口的类可以获得接口里定义的常量、方法，格式如下：

```
[修饰符]class 类名 [extends 父类名] implements 接口1[,接口2]{  
    .....  
}
```

- 如果实现某个接口的类不是抽象类或者该抽象类的子类，则都必须实现指定接口的所有抽象方法，抽象方法在实现时要修饰为public，并且要求方法的参数列表、名字和返回类型与接口定义中的完全一致。

#### 接口的实现

```
class F implements collection{  
    void add(Object obj){……}  
    float sum(float x,float y){……}  
    int currentCount(){……}  
}
```

#### 接口的声明

```
interface collection{  
    void add(Object obj);  
    float sum(float x,float y);  
    int currentCount();  
}
```

- 熟悉的栗子：
  - 定义接口Shape2D，指明类需要实现的方法：计算二维图形的面积和周长。

#### 接口的声明

```
interface Shape2D{  
    final double PI=3.14;  
    void area();  
    void perimeter();  
}
```



- 构造两种二维图形的类：矩形类和圆形类，实现该Shape2D接口

#### 接口的实现

```
class Rectangle implements Shape2D{
    int width,height;
    public Rectangle(int w,int h){
        width=w;
        height=h;
    }
    public void area(){
        System.out.println(width*height);
    }
    public void perimeter(){
        System.out.println(2*width+2*height);
    }
}
```

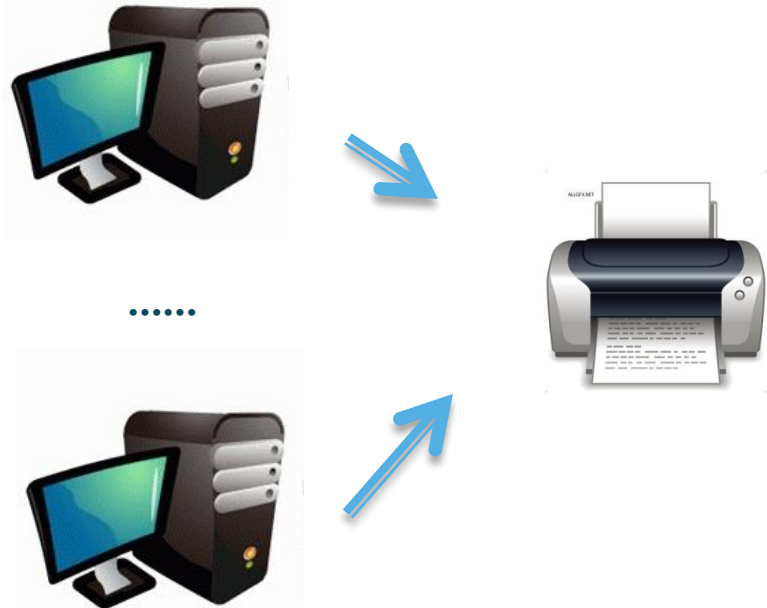
#### 接口的实现

```
class Circle implements Shape2D{
    int radius;
    public Circle(int r){
        radius=r;
    }
    public void area(){
        System.out.println(PI*radius*radius);
    }
    public void perimeter(){
        System.out.println(2*PI*radius);
    }
}
```

抽象类和接口都可以用于模拟共同特征，一般来说**strong is-a relationship**应该用类来继承模拟；**weak is-a relationship**是类属关系，指对象拥有某种属性，可用接口来模拟。例如，职员和人的关系应该用类继承模拟；字符串都可以用来比较，则可以使**String**类实现**Comparable**接口。

- 面向接口的编程-简单工厂模式

- 例子：程序中有多个Computer类需要组合输出设备Printer



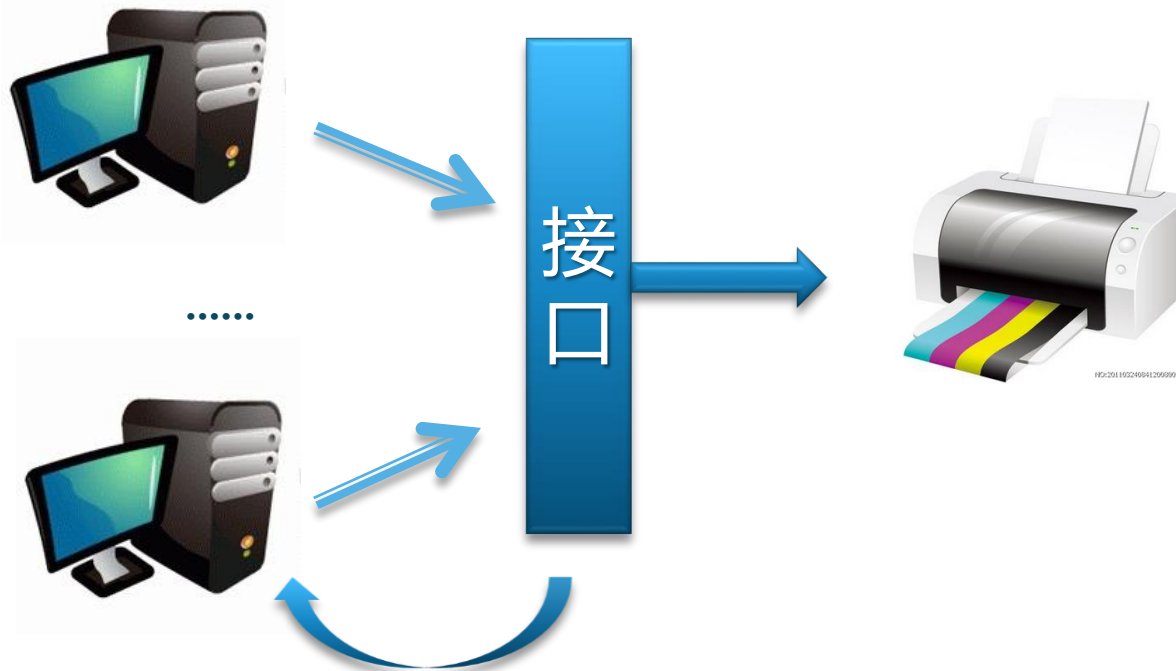
## 范例

```
class Printer{  
    final int MAX_CACHE_LINE=50;  
    private String[ ] printData=new String[MAX_CACHE_LINE];  
    Private int dataNum=0;  
    public void out(){.....} //将打印队列中的信息输出 ;  
    public void getData(String s){.....}  
    //将信息添加到打印队列 , 如果队列已满 , 则添加失败  
}
```

当有多个类以这样的方式和Printer组合后,一旦Printer变更为其输出设备,则需要对多个类做修改

```
public class Computer{  
    private Printer out;  
    public Computer(Printer out){  
        this.out=out;  
    }  
    public void keyIn(String s){out.getData(msg);}  
    public void print(){out.out();}  
}
```

- 建议的模式——接口



Computer不直接组合Printer类，而是组合由接口提供的输出设备的对象，将Computer类和Printer类完全分离。

**class Printer implements Output{**

private String[ ] printData=new String[MAX\_CACHE\_LINE];

Private int dataNum=0;

public void out(){.....} //将打印队列中的信息输出 ;

public void getData(String s){.....}

//将信息添加到打印队列 , 如果队列已满 , 则添加失败

}

**public class Computer{**

private **Output out;**

与Printer分离, 而与接口组合

public **Computer(Output out){**

this.out=out;

}

public void keyIn(String s){out.getData(msg);}

public void print(){out.out();}

}

```

public class OutputFactory{
    public Output getOutput(){
        return new Printer();
    }

    public static void main(String[ ] args){
        OuputFactory of=new OutputFactory();
        Computer c=new Computer(of.getOutput);
        c.keyIn("待输出内容1");
        c.print();
    }
}

```

在该方法中返回Output实现类的对象  
当Printer被ColorPrinter类取代时，只需要将这里改写为：

```
return new ColorPrinter();
```

由一个Output工厂来负责生成Output类的对象用以和Computer类组合

## 5.8 泛型

- 假设需要一个坐标类Point，在这个类中需要可以对以下三种类型的数据做保存：
  - 整形；浮点型；字符串型
  - 必须有一种数据类型可以保存这三类数据

### 泛型

类属性或者方法的参数在定义数据类型时，可以直接使用**一个标记**进行占位，在具体使用时才设置其对应的实际数据类型，这样当设置的数据类型出现错误后，就可以在编译时检测出来。

```
class Point<T>{
    private T x;
    private T y;
    public void setX(T x){
        this.x=x;
    }
    public void setY(T y){
        this.y=y;
    }
    public T getX(){
        return x;
    }
    public T getY(){
        return y;
    }
}
```

```
Point<Integer> p=new Point<>();
p.setX(10);
p.setY(20);
int x=p.getX();
int y=p.getY();
System.out.println(x+","+y);
```

```
Point<String> p=new Point<>();
p.setX("东经20度");
p.setY("北纬100度");
String x=p.getX();
String y=p.getY();
System.out.println(x+","+y);*/
```



- 即便是同一个类，由于设置泛型类型不同，其对象表示的含义也不同，因此不能直接进行引用操作

```
class Point<T>{  
    private T x,y;  
    public void setXY(T x,T y){  
        this.x=x;  
        this.y=y;  
    }  
    public T getX(){  
        return x;  
    }  
    public T getY(){  
        return y;  
    }  
}
```

```
public class TestDemo{  
    public static void main(String[] args){  
        Point<Integer> p=new Point<>();  
        p.setXY(10,20);  
        add(p);  
    }  
    public static void add(Point<Integer> temp){  
        System.out.println(temp.getX()+temp.getY());  
    }  
}
```

```

public class TestDemo{
    public static void main(String[] args){
        Point<String> p=new Point();
        p.setXY("10","20");
        add(p);
    }
    public static void add(Point<Integer> temp){
        System.out.println(temp.getX()+temp.getY());
    }
}

```

22: 错误: 不兼容的类型: Point<String>无法转换为Point<Integer>  
 add(p);  
 ^

```

public static void add(Point<String> temp){
    System.out.println(temp.getX()+" "+temp.getY());
}

```

错误。  
 方法重载要求参数类型不同，但是对于泛型则没有要求，会报错（方法已定义）

- 通配符 “?” ——用来接收任何泛型类的对象

```
public static void add(Point<?> temp){  
    System.out.println(temp.getX()+","+temp.getY());  
}
```

需要注意的是，“?”设置的泛型类型只表示可以取出，但是不能修改。

- “?” 还有两个子通配符
  - ? extends 类：设置泛型上限（在声明、方法参数中使用）
    - ? extends Bicycle ->可以设置参数为Bicycle或其子类(MountainBike等)
  - ? super 类：设置泛型下限（方法参数中使用）
    - ? super String ->可以设置参数为String及其父类

```
class Message<T extends Number>{
    private T msg;
    public void setMsg(T msg){
        this.msg=msg;
    }
    public T getMsg(){
        return msg;
    }
}

public class NumberTest{
    public static void main(String[] args){
        Message<Integer> m1=new Message<>();
        m1.setMsg(10);
        fun(m1);
    }
    public static void fun(Message<? extends Number> temp){
        System.out.println(temp.getMsg());
    }
}
```

- 泛型接口

- 在定义接口的实现类时继续设置泛型，在实例化接口实现类的对象时所设置的泛型类型就是接口中使用的泛型类型。

```
interface IMessage<T>{  
    public void print(T t);  
}  
  
class MessageImp<S> implements IMessage<S>{  
    public void print(S t){  
        System.out.println(t);  
    }  
}  
  
public class InterfaceTest{  
    public static void main(String[] args){  
        IMessage<String> s=new MessageImp<String> ();  
        s.print("hello");  
    }  
}
```

- 在实现类中不设置泛型，为接口明确地定义一个泛型类型。

```
interface IMessage<T>{  
    public void print(T t);  
}  
  
class MessageImp implements IMessage<String>{  
    public void print(String t){  
        System.out.println(t);  
    }  
}  
  
public class InterfaceTest{  
    public static void main(String[] args){  
        IMessage<String> s=new MessageImp();  
        s.print("hello");  
    }  
}
```

- 泛型方法

- 可以在方法的参数或者返回值定义泛型
- 泛型方法不一定要在泛型类中定义

```
public class funTest{  
    public static void main(String[] args){  
        String s=fun("hello");  
        System.out.println(s.length());  
    }  
  
    public static <T> T fun(T t){  
        return t;  
    }  
}
```

## 5.8 包-package

- 包是组织一组相关类和接口的名称空间。

- **Java平台提供的数量庞大的类库(包的集合)，可以用于应用程序的开发，它的包代表和通用编程相关的最常见的任务；各种软件厂商、个人开发者也会提供成千上万各种用途的类。**
- **包用于提供类的多层命名空间，用于解决类的命名冲突、类文件管理等问题。**

- 包的作用

- 如同文件夹的概念一样，不同的包中的类可以同名，这样可以避免命名的冲突；
- 使得功能相关的类易于查找和使用，同一包中的类和接口通常是功能相关的；
- 可提供一种访问权限的控制机制，一些访问权限以包为访问范围。



- 定义

```
package 包名1[.包名2[.包名3...]];
```

- 在定义类和接口的源文件的第一行使用package语句，就指明了该文件中定义的类和接口属于第一条语句定义的包；
- 包可以带路径，形成与Java开发系统文件结构相同的层次关系。
- 包嵌套的层次可以用来保证包名的唯一性。

包的定义

```
package Family;  
class Father{.....}  
class Mother{.....}  
class Son{.....}
```

依次把Father类、Mother类、Son类装入包Family

- 不同程序文件内的类也可同属于一个包，只要在这些程序文件前加上同一个包的声明；

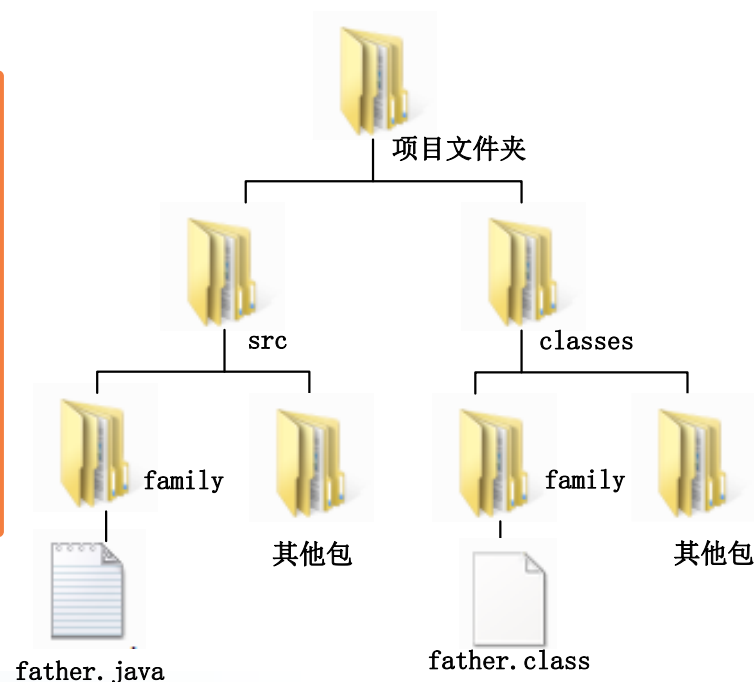
程序1

```
package Animals;  
class Cat{……}
```

程序2

```
package Animals;  
class Dog{……}
```

**Java建议位于包中的类，在文件系统中也有与包名层次相同的目录结构，这是为了解决类文件在文件系统中的存储冲突，同时也是为了方便为设置classpath时指定使用该类需要到哪个目录去寻找。**



## • 包的引用

- 将类和接口组织成包的目的是为了能够更有效的使用包中的已经具备一定功能的类。
- 装载使用已编译好的包的方式：

- 在要引用的类名前带上包名：

装载方式1

```
Animals.Cat cat=new Animals.Cat();
```

- 在文件开头使用import引入包中的类：

装载方式2

```
import Animals.Cat;  
class Check{  
    Cat cat=new Cat();  
}
```

- 在文件前使用import引入包中所有的类：

装载方式2

```
import Animals.*;  
class Check{  
    Cat cat=new Cat();  
    Dog dog=new Dog();  
}
```

- 使用\*时表示要从包中引入所有的类，也只能是该包中的类(不能指包)；

包的导入

```
import java.awt.*;
```

- 如果想使用awt包下面的event包中的类，需要明确写出：

```
import java.awt.event.*;
```

- Java编译器为所有程序自动引入包java.lang，所以不必显式引入。
- 在某些极端情况下程序需要使用不同包下面的同名类，那么在使用import导入后，在程序中仍然需要写明该类的全名。

- 静态导入

- JDK1.5以后增加了静态导入的语法，用于导入指定类的某个或全部静态成员变量、方法。

- 语法

`import static 包名1[.包名2[.包名3...]].类名.静态变量|静态方法;`

`import static 包名1[.包名2[.包名3...]].类名.*;`

#### 静态导入

```
import static java.lang.Math.*;
public class StaticImport{
    public static void main(String[] args){
        System.out.println(Math.PI);
        System.out.println(Math.sqrt(256));
    }
}
```

使用**import**可以省略写包名，  
使用**import static**可以连类名都省略，而直接使用该类下面定义的方法和类变量。