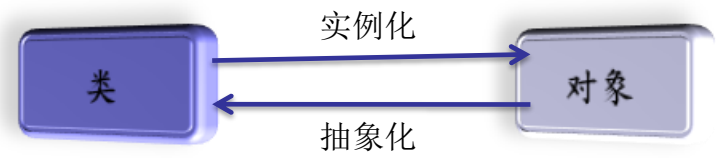
Java软件设计基础

4. 类和对象

2.1 简介

现实世界中复杂的对象是由许多小的、简单的对象组成的。客观存在的物体就是最基本的对象。不同的物体有共性，共性存在于个性之中，物体的个性又继承了共性。

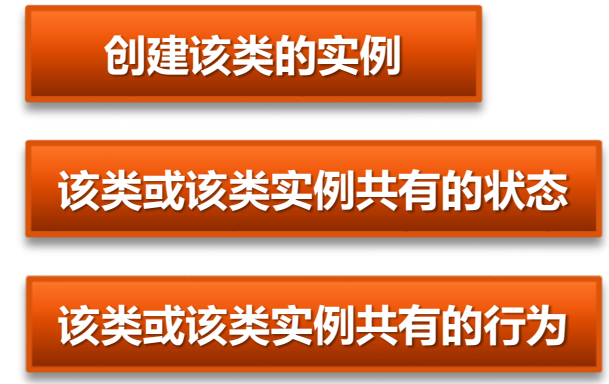
- 对象的状态属性
  - 对象蕴含着许多信息，可以用**一组状态**来表征。
- 对象的行为操作
  - 对象内部含有对数据的操作，即对象的**行为**操作。



•类(class)：用于描述客观世界里某一类对象的共同特征
•对象(object、instance)：是类的具体存在

- 类的定义

```
[修饰符]class 类名{
        零个到多个构造方法定义…
        零个到多个成员变量…
        零个到多个方法…
}
```



- 修饰符：用来说明类的属性，可分为访问控制符和非访问控制符两类。
  - 访问控制符
    - public：表示该类可以被任何对象或类来调用或引用，包括同一个包或不同包中的类访问。一个程序中只能有一个公共类。
    - 无修饰符：表示只能被该类的方法访问和修改，而不能被任何其他类、包括该类的子类来获取和引用，这是系统默认的修饰符。

- 非访问控制符
  - abstract：抽象类。表示该类是无具体对象的抽象类，这是一种特殊的类，它不能实例化，即不能用new关键字来生成该类的对象，只能由其派生子类。其抽象方法的具体实现也由继承该抽象类的子类来完成。
  - final：表示该类为最终类，不能再由它派生新的子类。

抽象类可以同常规类一样，具有数据和方法，包含抽象方法的类必须声明为抽象类。

abstract和final不能修饰同一个类。

- 成员变量定义

[修饰符]类型 成员变量名[=初始值];

成员变量也叫field，datafield，各本书翻译不同，称为域、字段、属性等，指的都是一种东西。

- 修饰符
  - 访问控制修饰符
    - 缺省访问修饰符：同friendly修饰符，修饰的成员变量可以被同一包（package）中的任何类访问；
    - public：修饰的成员变量称为公共变量，可以被项目文件中的任何方法访问。
    - protected：修饰的成员变量称为保护变量，可以被有继承关系的类自由访问，即子类可以访问它；也可以由同一个包中的其它类访问。
    - private：修饰的成员变量称为私有变量，只能被定义它的类使用，而不能被其他任何类使用。这种方式通常是最安全的，在Java程序开发中通常使用该修饰符实现数据的封装；

由于public成员变量不受限制，这易使类的对象引起不希望的修改，破坏系统的封装性，建议成员变量尽量不要使用public修饰符；

访问控制修饰符	类	子类		其它类	
		同一个包	其它包	同一个包	其它包
public	✓	✓	✓	✓	✓
private	✓	✗	✗	✗	✗
protected	✓	✓	✓	✓	✗
默认修饰符	✓	✓	✗	✓	✗

- 非访问控制修饰符
  - **final：用它修饰的成员变量就是常量**
  - **static：修饰的成员变量称为类变量或静态变量**

- 方法定义

```
[修饰符]返回值类型 方法名([形参列表]){
        方法体;
}
```

  - 修饰符：
    - **访问控制修饰符：默认，public，private，protected，相应的访问范围同成员变量的修饰符；**
    - **非访问控制修饰符：**
    - **final：最终方法，不能被覆盖或重载**
    - **abstract：抽象方法，仅有方法头而没有具体方法体的方法，所有的抽象方法必须存在于抽象类中；**
    - **static：类方法/静态方法**
    - **synchronized：被该修饰符修饰的方法，一次只能被一个线程使用，进而可以控制多个并发线程的访问；**

- 形参列表
  - 是方法的输入接口，列出了一系列参数的形式，包括个数、类型和名称，这类参数只有在运行时才分配存储单元，被称为“形式参数”。
  - 对于方法定义中的每一个参数必须合法，调用时必须由一个实参量与其对应，参量的个数、类型必须与对应的形参的个数、类型完全一致。
  - 方法也可以没有参数。
- 返回值类型
  - 可以是任意的Java类型，如果方法没有返回值，则用void表示。
  - 在区分多个同名方法时，编辑器不会考虑返回值类型，所以不能声明两个参数列表相同而返回值不同的同名方法。（）
  - 当方法使用一个类名作为返回值类型时，返回对象的类型必须是返回类型指定的类或者它的子类。

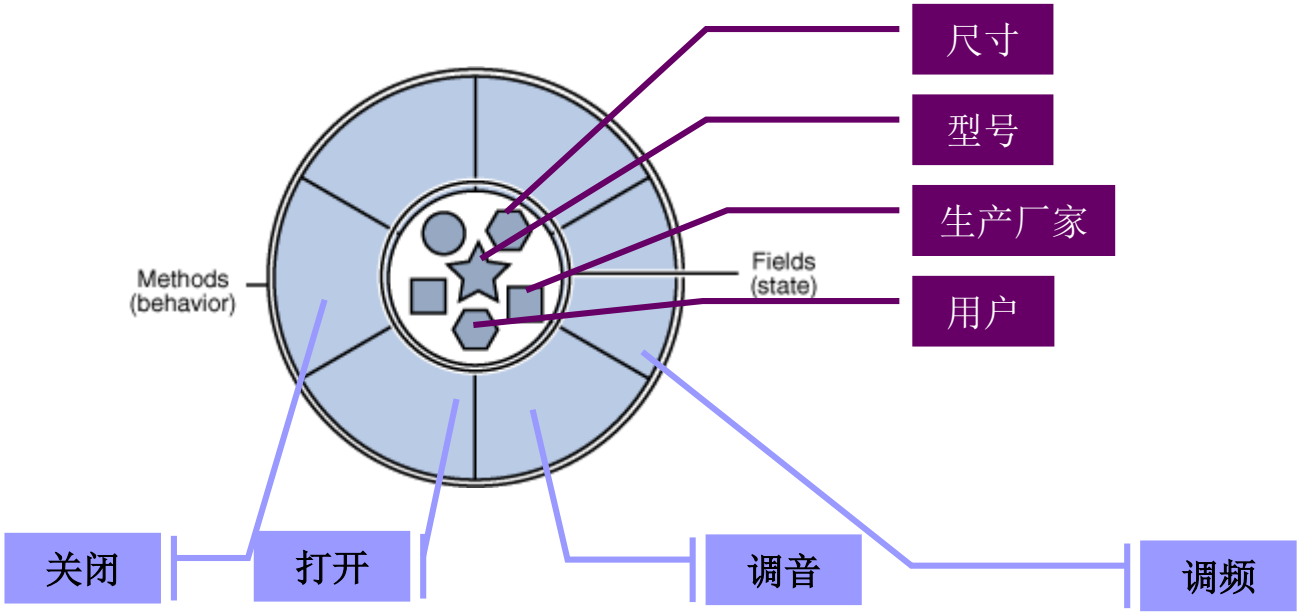
- 构造方法(构造器)定义

```
[修饰符]构造方法名([形参列表]){
        方法体;
}
```

  - 修饰符
    - **默认，public，private，protected，相应的访问范围同成员变量的修饰符；**
  - 构造方法名
    - **必须和所在类型相同**

构造方法既不能定义返回值类型，也不能使用void声明没有返回值。如果一个构造方法定义了返回值类型(包括void)，编译时不会报错，但会被当初普通方法处理，而不会被当成构造方法。

- 第一个类的定义
  - 收音机



收音机类

```
class Radio{
        float size;
        String type;
        String manufacturer
        String user;
        Radio(…){
                ……
        }
        void turnOn(){
                ……
        }
        void turnOff(){}{
                ……
        }
        changeVolume(…){
                ……
        }
        changeChannel(…){
                ……
        }
}
```

- 一个更具体的例子——点类

类定义示例

```
class Cpoint{
        private int x,y;
        public Cpoint(int a,int b){
                x=a;y=b;
        }
        public void setPoint(int a,int b){
                x=a;y=b;
        }
        public int getx(){
                return x;
        }
        public int gety(){
                return y;
        }
        public String getString() {
                return “【x=”+x+”,y=”+y+”】”;
        }
}
```

成员变量

构造方法

成员方法

2.2 使用类

- 创建和使用对象
  - 一个类可以创建多个实例，实例的生成包括**声明、实例化**和**初始化。**
  - 对象的生命周期为：创建→使用→销毁回收。
  - 语法格式

```
类名 对象名;
对象名=new 类名([参数列表]);
```

```
类名 对象名=new 类名([参数列表]);
```

对象生成语句

```
Person Mary;
Mary = new Person();
Person John = new Person();
```

  - 通过new关键字调用构造方法创建该类的实例。
    - **为该对象分配了引用内存空间；**
    - **在new运算符完成操作前，构造方法立刻自动调用，用来完成初始化该实例；**

- Java允许一个类中有若干个构造方法，但这些构造方法的参数必须不同。多个参数列表不同的同名方法，称为“重载”；
- 仅当程序中没有显式的编写构造方法时，Java才会为该类创建一个默认的、无参构造方法。

- 例子-创建实例（矩形对象的创建）
  - 矩形类的属性：
    - 矩形长
    - 矩形高
    - 矩形左上角所在坐标点
  - 坐标点类的属性
    - 横坐标
    - 纵坐标

矩形属性

```
public int width=0;
public int height=0;
public Point center;
```

点属性

```
public int x=0;
public int y=0;
```

- 坐标点类的构造方法

点类构造方法

```
public Point(int a,int b){x=a;y=b;}
```

- 矩形类的构造方法

矩形类构造方法

```
public Rectangle(){center = new Point(0,0);}
```

```
public Rectangle(Point p){center = p;}
```

```
public Rectangle(int w,int h){
        center=new Point(0,0);
        width=w;
        height=h;
}
```

```
public Rectangle(Point p,int w,int h){
        center = p;
        width=w;
        height=h;
}
```

- 根据需要编写成员方法

计算矩形面积

```
public int area(){return width*height;}
```

计算矩形周长

```
public int lengthOfRec(){return 2*(width+height);}
```

- 使用矩形类创建实例

```
Rectangle r1;
r1=new Rectangle(5,4);
Rectangle r2=new Rectangle(new Point(1,1),3,3);
```

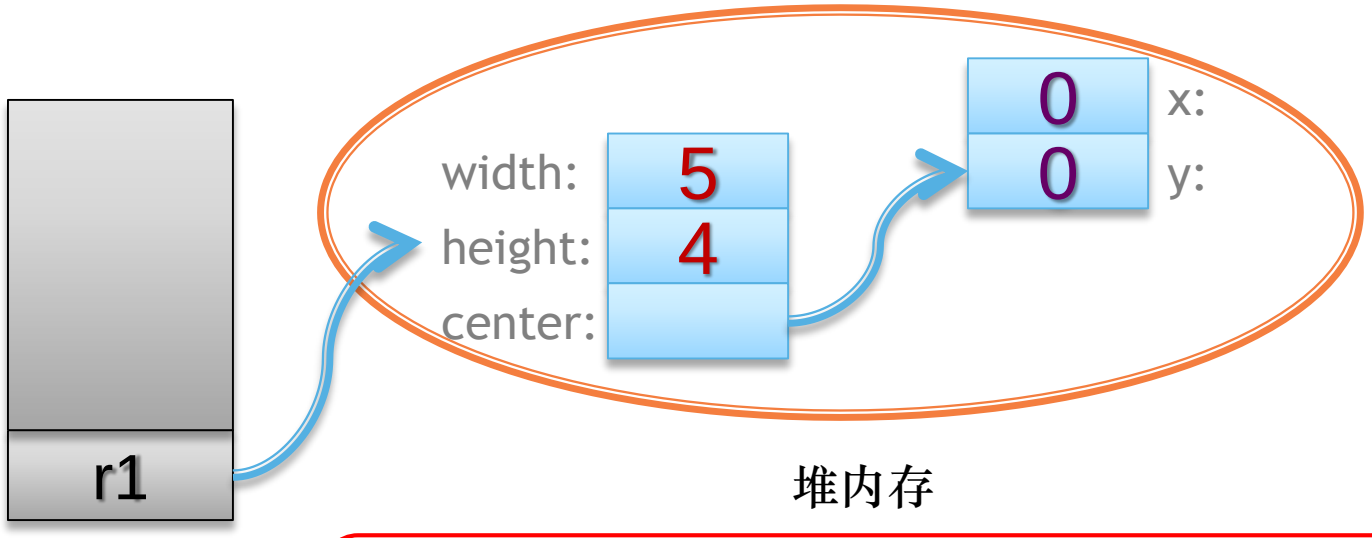
  - 上面的语句产生一个左上角在(0,0）位置，宽5高4的矩形实例，以及一个左上角在（1,1）位置，宽3高3的矩形实例。

- 创建实例以后可以使用对象中的成员变量/成员方法

- 上例中创建矩形实例r1、r2后，可以使用其成员变量或调用其成员方法；

```
System.out.println(r1.width);
System.out.println(r2.area());
```

- 实例在内存中



当一个实例被创建成功以后，将保存在堆内存中。Java程序不允许直接访问堆内存中的对象，只能通过该对象的引用操作该对象。

2.3 this关键字

- Java提供的this关键字总是指向调用该方法的对象
  - 构造方法中使用this，此时this指该构造方法正在初始化的对象
    - 对字段使用关键字this。这种情况的最常见原因是构造方法的参数遮蔽了对象的成员字段。

原构造方法示例

```
class Point{
        public int x=0;
        public int y=0;
        public Point(int a,int b){x=a;y=b;}
}
```

构造方法参数遮蔽对象的成员变量

```
class Point{
        public int x=0;
        public int y=0;
        public Point(int x,int y){
              this.x=x;
        }     this.y=y;
}
```

- 对构造方法本身使用this
  - **使用this关键字调用同一个类中的另一个构造方法，称为显式构造方法调用(explicit constructor invocation)。格式如下：**

```
this(参数列表);
```

  - **编译器根据参数的数量和类型判断应该调用哪个构造方法。**
  - **如果存在，对另一个构造方法的调用必须是构造方法中的第一行。**

显示构造方法调用

```
public class Rectangle{
        private x,y,width,height;
        public Rectangle(){
                this(0,0,0,0);
        }
        public Rectangle(int width,int height){
                this(0,0,width,height);
        }
        public Rectangle(int x,int y,int width,int height){
                this.x=x;
                this.y=y;
                this.width=width;
                this.height=height;
        }
        ……
}
```

用四个0值调用四个参数的构造方法

用两个0值和width、height值调用四个参数的构造方法

2.4 static修饰符和final修饰符

可用于修饰方法和成员变量。

static修饰的成员变量表明它属于这个类本身，而不属于经由该类创建的单个实例，称为**类变量**；而类变量为该类的所有对象所共享，它的值不因类的实例不同而不同。

不加static修饰的成员变量又叫**对象变量**，对象变量属于具体的实例，它的值因具体对象实例的不同而不同。

类名.成员变量名	对象名.成员变量名
类名.方法名	对象名.方法名

虽然两种方式都可以访问到/调用到成员变量和方法，但建议使用前者，来明确其属于类本身的特性

- final：修饰的成员变量叫最终成员变量，亦即常量。
  - 一开始创建该变量时将其设了一个初始值，在以后程序的运行过程当中，变量的值将一直保持这个值不变。
  - Java中的常量必须是类的成员。对于最终成员变量，任何赋值都将导致编译错误。因为常量在声明以后就不能改变其值，所以常量必须要初始化。无论是实例变量，还是类变量，都可以被说明成常量。final修饰符和static修饰符并不冲突。

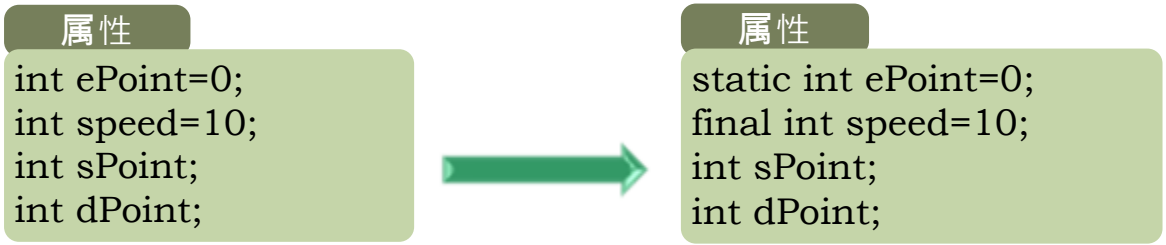
常量例程

```
class EFinal{
        public static void main(String args[]){
                System.out.println(Tom.MIN);
                Tom tom=new Tom();
                System.out.println(tom.MIN+tom.MAX);
        }
}
class Tom{
        final int MAX=70;
        static final int MIN=30;
}
```

例程输出结果

```
30
100
```

- static实例——电梯
  - 类WaitE的属性
    - **电梯的位置**
    - **电梯运行速度**
    - **人所在的位置**
    - **人要去的目的地**



- 构造方法

```
public WaitE(int x,int y){
        sPoint=x;
        dPoint=y;
}
```

- 成员方法
  - **获得电梯位置**
  - **获得电梯速度**

方法

```
public int getEPoint(){return EPoint;}
public int getSpeed(){return speed;}
```

  - **打印等待时间**
  - **打印乘坐时间**

方法

```
public void  WTime()
{
        if(sPoint=ePoint) System.out.println(“不需要等待”);
        else System.out.println(“需要等待”+abs(ePoint-sPoint)*speed+”秒”);
}
public void RTime()
{       if(dPoint=ePoint) System.out.println(“处于同一层!”);
        else System.out.println(“到达目的楼层需”+abs(ePoint-dPoint)*speed+”秒”);
}
```

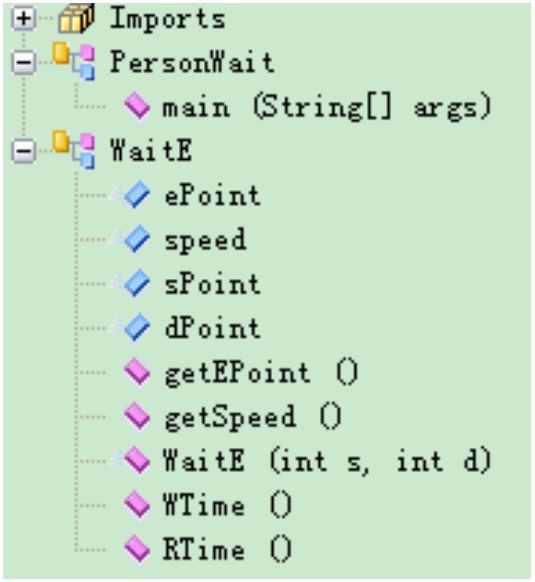
- 对于某个具体等待电梯的人的操作
  - **创建并初始化（即此人在哪一楼层，要去往哪一楼层）**
  - **打印等待时间**
  - **将电梯置位为该对象目前所在的楼层**
  - **打印乘坐时间**
  - **将电梯置位为该对象到达的楼层**

对象操作

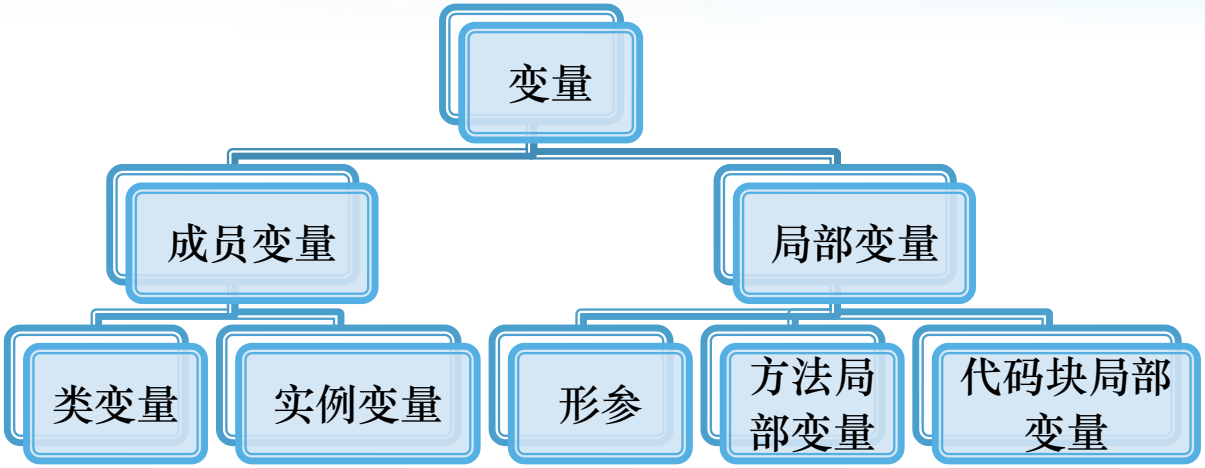
```
WaitE p1=new WaitE(x,y);
WTime();
WaitE.EPoint=x;
Rtime();
WaitE.EPoint=y;
```

例程输出结果

```
2 5
需要等待20秒
到达目的楼层需30秒
4 6
需要等待10秒
到达目的楼层需20秒
```



```
import java.util.*;
public class PersonWait{
    public static void main(String args[]){
        Scanner reader=new Scanner(System.in);
        int x,y;
        x=reader.nextInt();
        y=reader.nextInt();
        WaitE p1=new WaitE(x,y);
        p1.WTime();
        WaitE.ePoint=x;
        p1.RTime();
        WaitE.ePoint=y;
        x=reader.nextInt();
        y=reader.nextInt();
        WaitE p2=new WaitE(x,y);
        p2.WTime();
        WaitE.ePoint=x;
        p2.RTime();
        WaitE.ePoint=y;
    }
}
class WaitE {
    static int ePoint=0;
    final int speed=10;
    int sPoint;
    int dPoint;
    public int getEPoint(){ return ePoint;}
    public int getSpeed(){return speed;}
    WaitE(int s,int d){
        sPoint=s;
        dPoint=d;}
    public void  WTime(){
        if(sPoint==ePoint) System.out.println("不需要等待");
        else System.out.println("需要等待"+Math.abs(ePoint-sPoint)*speed+"秒");}
    public void RTime(){
        if(dPoint==ePoint) System.out.println("处于同一层!");
        else System.out.println("到达目的楼层需"+Math.abs(ePoint-dPoint)*speed+"秒");}
}
```



✓如果一个变量或方法依赖于类的具体实例，就应该定义为实例变量或实例方法；

✓如果一个变量或方法不依赖于类的具体实例，就应该定义为静态变量或静态方法。

✓程序中使用的局部变量应该尽可能的缩小作用范围，作用范围越小，在内存里停留的时间就越短，程序运行性能越好，代码内聚性越好。

2.5 方法

- 方法是类的主要组成部分，又称为成员方法，用来规定类属性上的操作，改变对象的属性与产生行为，实现类的内部功能的机制，接收来自其他对象的信息以及向其他对象发送消息。方法同时也是类与外界进行交互的重要窗口。
- 通常一个类由一个主方法和若干个子方法构成。主方法调用其他方法，其他方法间也可以互相调用，同一个方法可以被一个或多个方法调用。
- 方法实现子任务处理的原则和规律：
  - 算法中需要细化的步骤、重复的代码以及重载父类方法都可以定义成类的方法；
  - 方法应界面清晰、大小适中。
  - Java应用程序中，程序的执行从主方法即main方法开始，调用其他方法后返回到main方法，并在main方法中结束整个程序的运行。

方法调用例程

```
public class ScircleArea
{
        static double scircle(int r)
        {
                double ss;
                ss=3.14*r*r;
                return(ss);
        }
        static void Area(int a,int b)
        {
                int s;
                s=a*b;
                System.out.println(s);
        }
        public static void main(String args[])
        {
                int x=5,y=4;
                double result=scircle(x);
                System.out.println(result);
                Area(x,y);
                System.out.println(scircle(y));
        }
}
```

例程输出结果

```
78.5
20
50.24
```

方法表达式调用

方法语句调用

输出语句调用

无参数方法调用例程

```
class NoParameter
{
        static void NoParaSum()
        {
                int i,j,s;
                i=5;j=15;s=i+j;
                System.out.println(s);
        }
        public static void main(String[] args)
        {
                NoParaSum();
        }
}
```

例程输出结果

```
20
```

类名调用类方法例程

```
class Ntransfer
{
        public static void main(String args[])
        {
                double max=Com.max(10,25);
                System.out.println(max);
        }
}
class Com
{
        double x,y;
        static double max(double a,double b)
                {return a>b?(a*b):(a+b);}
}
```

例程输出结果

```
35.0
```

方法不能独立定义，只能在类体里定义；

从逻辑意义上来看，方法要么属于该类本身，要么属于该类的一个对象；

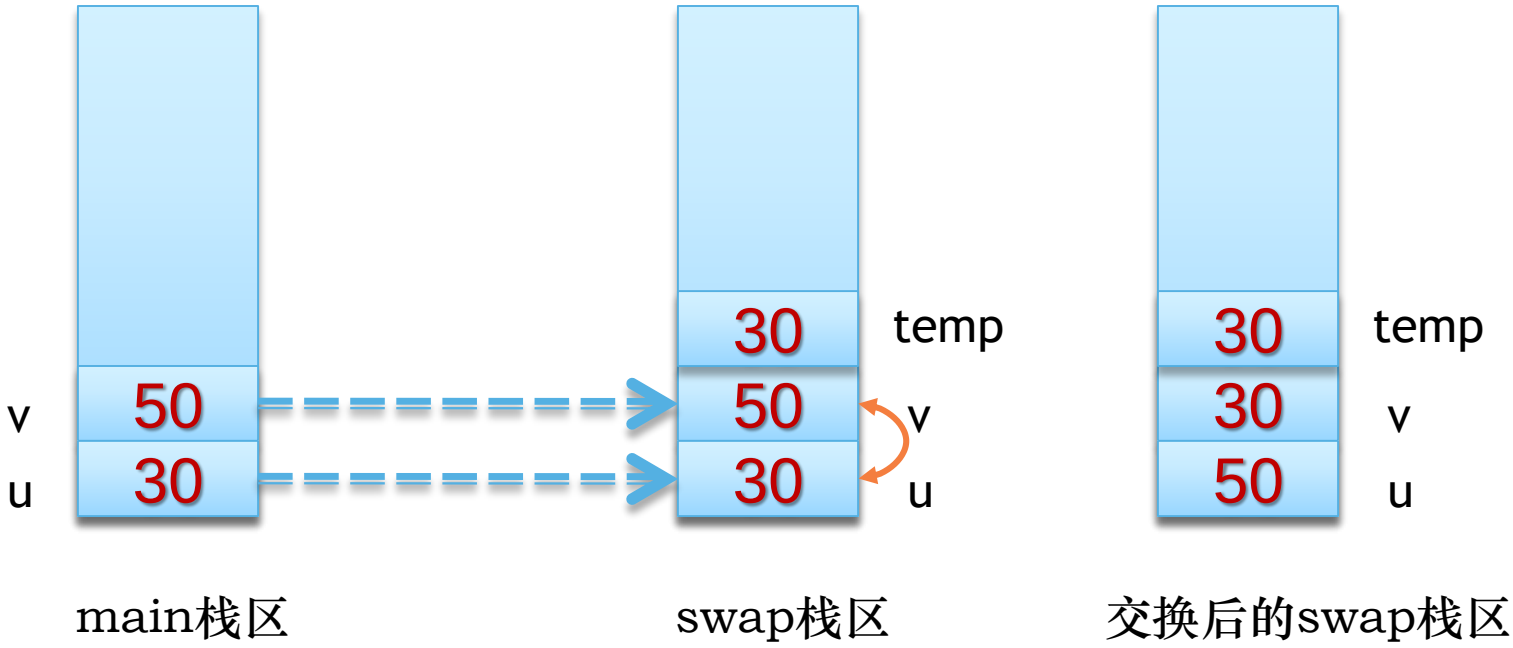
永远不能独立的执行方法体，必须使用类或者对象作为调用者。

- 方法的参数传递机制
  - 指在Java语言中调用一个带有形参的方法时，完成所提供的实参与形参的传输结合的过程。
    - **形参是方法声明中的变量清单；**
    - **实参(argument)是调用方法时传递给方法的实际值。**
  - 传递方式
    - **传值：若方法参数为简单数据类型，将实参的值传递给形参，方法接受实参的值，但是不能改变；**

```
public class SwapUV {
        static void swap(int u,int v){
                int temp;
                System.out.print("交换前:");
                System.out.println("u="+u+",v="+v);
                temp=u;u=v;v=temp;
                System.out.print("交换后");
                System.out.println("u="+u+",v="+v);
        }
        public static void main(String[ ] args){
                int u=30,v=50;
                System.out.print("调用前:");
                System.out.println("u="+u+",v="+v);
                swap(u,v);
                System.out.print("调用后");
                System.out.println("u="+u+",v="+v);
        }
}
```

例程输出结果

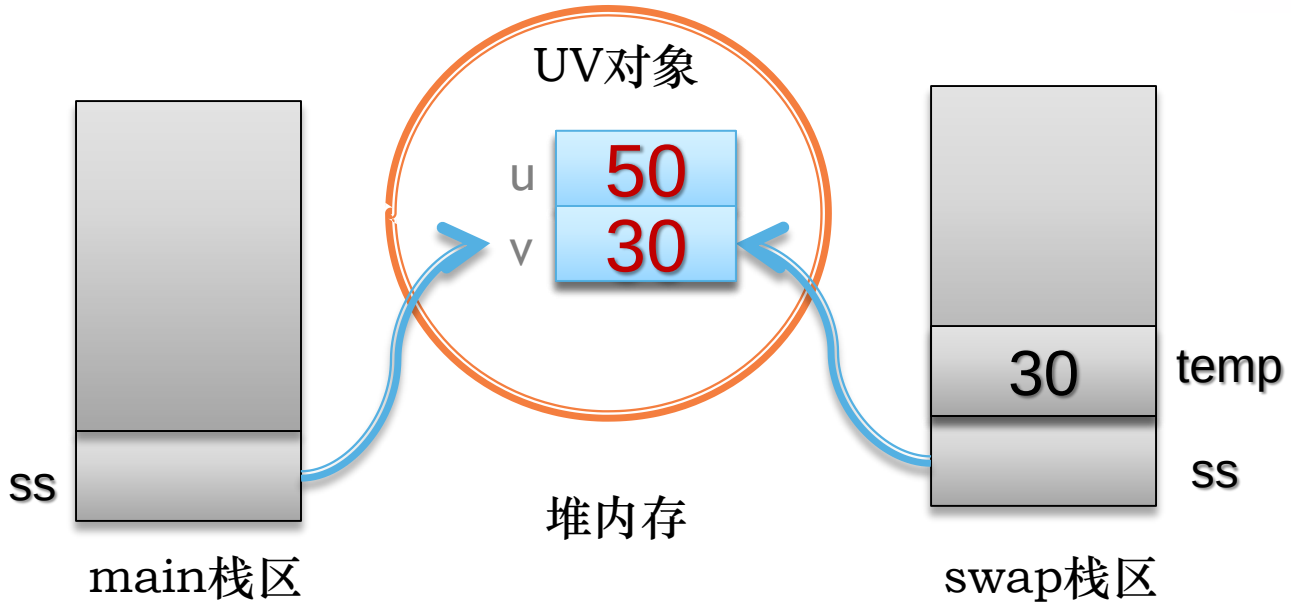
```
调用前:u=30,v=50
交换前:x=30,y=50
交换后x=50,y=30
调用后:u=30,v=50
```


v
u
50
30
30
50
30
temp
v
u
30
30
50
temp
v
u
main栈区
swap栈区
交换后的swap栈区

- 传引用：若方法参数是引用类型，则将对象的引用传递给方法

```
class UV{
    int u,v;
    public UV(int u,int v){       this.u=u;       this.v=v;    }
}
public class SwapRef {
    public static void swap(UV ss){
        int temp;
        System.out.println("交换前：u="+ss.u+",v="+ss.v);
        temp=ss.u;        ss.u=ss.v;        ss.v=temp;
        System.out.println("交换后：u="+ss.u+",v="+ss.v);    }
    public static void main(String[] args){
        UV ss=new UV(30,50);
        System.out.println("调用前：u="+ss.u+",v="+ss.v);
        swap(ss);
        System.out.println("调用后：u="+ss.u+",v="+ss.v);
    }
}
```

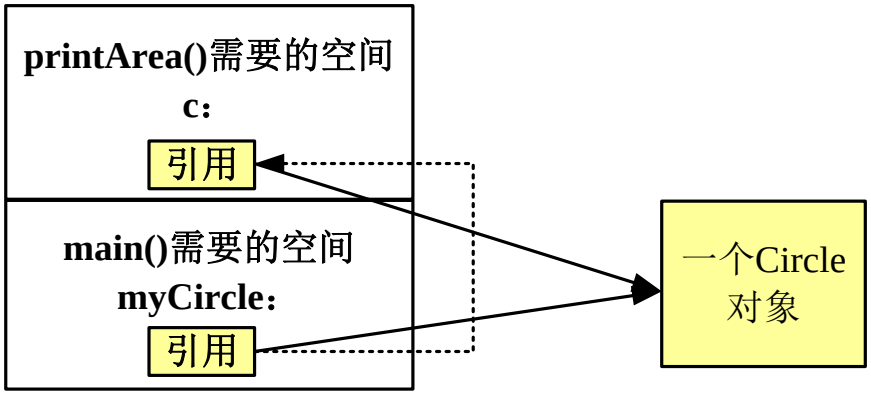
```
调用前：u=30, v=50
交换前：u=30, v=50
交换后：u=50, v=30
调用后：u=50, v=30
```



main栈区的引用ss与swap栈区的ss是两个不同的引用，但指向同一个对象。这点可以通过以下方法验证：
将swap栈区的ss赋值为null以后，仍然可以通过main栈区的ss访问到对象；

参数传递例程

```
class Circle{
        //类体略;
}
public class Example4_8{
        public static void main(String[] args){
                Circle myCircle = new Circle(1);
                printAreas(myCircle);
                System.out.println("Radius is "+myCircle.getRadius());
        }
        public static void printAreas(Circle c){
                System.out.println(c.getRadius()+" "+c.getArea());
                c.setRadius(c.getRadius()+1);
        }
}
```



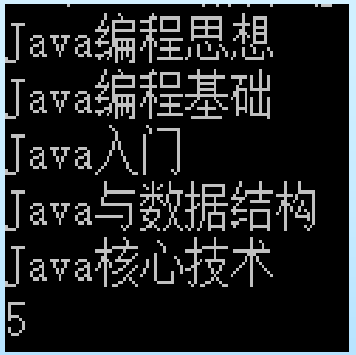
Java不允许将方法传递给方法，但是可以将对象传递给方法，然后调用这个对象的方法。

- 任意数量的实参(after jdk1.5)
  - Java允许定义形参个数可变的参数，从而允许为方法指定数量不确定的形参：

[修饰符] 返回值类型 方法名([形式参数列表,]最后一个参数的类型... 参数名称)

  - 可变形参只能处于形参列表的最后，一个方法中最多只能包括一个个数可变的形参。

```
public class Varargs{
    public static void test(int a,String... books){
        for(String tmp:books){
            System.out.println(tmp);
        }
        System.out.println(a);
    }
    public static void main(String[] args){
        test(5,"Java编程思想","Java编程基础","Java入门","Java与数据结构","Java核心技术");
    }
}
```



- 特殊的方法调用——递归
  - 递归是指用自身结构来描述自己、循环调用。例如：
    - 阶层 $\begin{cases} n! = n \times (n-1)! & n \geq 2 \\ n! = 1 & n = 1 \end{cases}$
    - Fibonacci数列 $\begin{cases} F(n) = F(n-1) + F(n-2) & n \geq 3 \\ F(1) = F(2) = 1 \end{cases}$
  - 递归方法有直接递归方法和间接递归方法。
    - 方法体中又调用自身称为直接递归方法；方法体中调用的虽然不是自身，但是它间接调用了自身，称为间接递归方法。

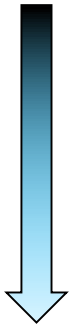
- 递归方法调用涉及两个概念：
  - 栈：数据依次入栈，后进先出。
  - 递归方法调用数据的保留：每调用一次便分配一次局部变量，待调用结束局部变量就释放。但从递归过程的执行可以看出，上一次调用的执行尚未结束，下一次调用就开始了。所以递归调用的数据保留采用栈来解决。

当一个方法递归调用自身时，必须在某个时刻方法的返回值是确定的，否则这种递归就变成了无穷递归，因此递归的重要准则就是：递归一定要向已知方向递归。

递归例程

```
class Factorial
{
        static long fac(int n)
        {
                if(n==1) return 1;
                else return n*fac(n-1);
        }
        public static void main(String args[])
        {
                int k=10;
                System.out.println(fac(k));
        }
}
```

5	fac(1)	1
4	fac(2)	?
3	fac(3)	?
2	fac(4)	?
1	fac(5)	?



由于n不等于1，所以连续执行4次fac(n-1)

4	fac(2)	2
3	fac(3)	?
2	fac(4)	?
1	fac(5)	?

第四次n=1，所以fac(1)=1，退栈一次，弹出栈顶数据。

3	fac(3)	6
2	fac(4)	?
1	fac(5)	?

再一次执行fac(n)=n*fac(n-1)，得到fac(n)=6,退栈。

1	fac(5)	120

以后的步骤类似，最后一次执行，n为5，fac(4)=24,fac(5)=120

- 对象的清除
  - 当不存在对一个对象的引用时，该对象成为一个无用对象。Java的运行环境周期性的检测某个实体是否已经不再被任何对象所引用，如果存在，则收集并释放它占有的内存空间。
- 并非实例、类变量和方法的所有组合都被允许使用
  - 实例方法可以直接访问实例变量和实例方法
  - 实例方法可以直接访问类变量和类方法
  - 类方法可以直接访问类变量和类方法
  - 类方法不能直接访问实例变量和实例方法，它们必须通过对象引用。

4.6 引用类型数组

- 引用类型数组的定义和简单类型结构定义类似，步骤如下：
  - 声明数组对象

```
类型[] 数组名;
```

  - 创建，并初始化数组的长度

```
类型[] 数组名=new 类名[数组的大小];
```

  - **初始化每个数组中的元素（即对象）**

```
数组名[下标]=new 构造方法([参数列表]);
```

- 经过以上步骤才为每个数组元素分配了存储空间，才可以使用引用类型数组中的元素。

- 实例：为PB1611班级创建一个学生类，并生成实例进行操作，判断是否能获得奖学金。
  - 类的声明

声明学生类

```
class Student{ }
```

  - 分析学生类应该具有的属性
    - 姓名
    - 班级
    - 平均分
    - 家庭年收入

学生属性

```
String name;
String classNumber;
double average;
int salary;
```

```
String name;
static final String classNumber="1711";
private  double average;
private int salary;
```

- 构造方法
  - 采用带有三个参数的构造方法，为学生的三个属性进行初始化

构造方法

```
public Student(String n,double a,int s){
        name=n;
        average=a;
        salary=s;
}
```

- 成员方法
  - 获取姓名
  - 获取班级
  - 获取平均分
  - 获取家庭年收入

获取姓名

```
public String getName(){return name;}
```

获取班级名

```
public String getClassNumber(){return classNumber;}
```

获取平均分

```
public double getAverage(){return average;}
```

获取家庭年收入

```
public int getSalary(){return salary;}
```

- 学生数组
  - 声明

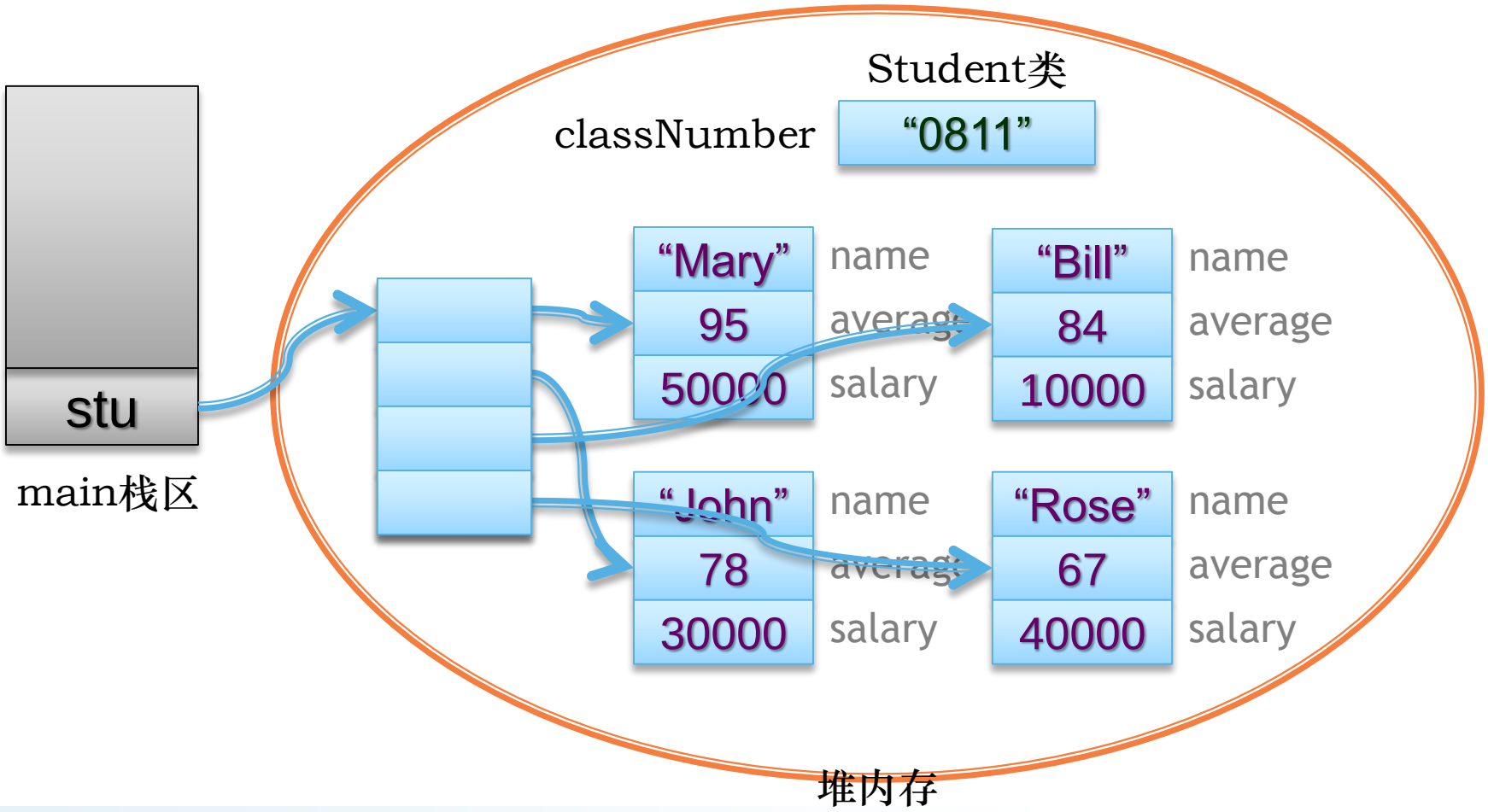
```
Student[ ] stu;
```

  - 数组的创建

```
stu= new Student[4];
```

  - 每个数组元素的初始化

```
stu[0]=new Student(“Mary”,95,50000);
stu[1]=new Student(“John”,78,30000);
stu[2]=new Student(“Bill”,84,10000);
stu[3]=new Student(“Rose”,67,40000);
```


Student类
classNumber
"0811"
stu
main栈区
"Mary"
name
95
average
50000
salary
"Bill"
name
84
average
10000
salary
"John"
name
78
average
30000
salary
"Rose"
name
67
average
40000
salary
堆内存

- 打印学生信息

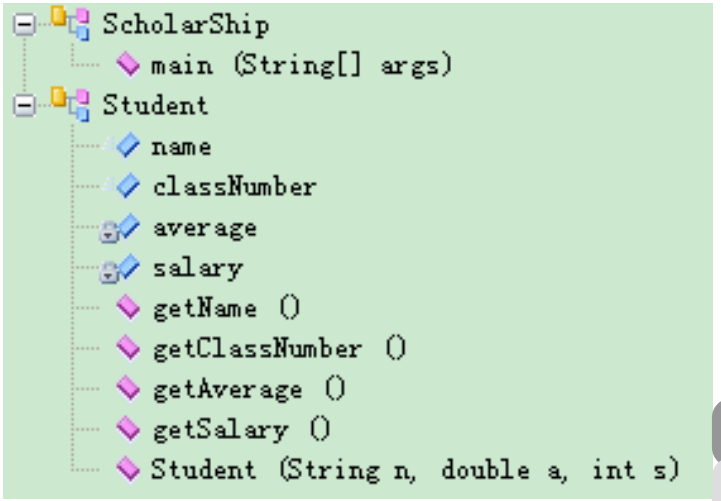
打印学生信息

```
System.out.println("Class Number:"+Student.classNumber);
for(int i=0;i<stu.length;i++){
        Student s=stu[i];
        System.out.println("Name:"+s.getName()+
                                "|Average:"+s.getAverage()+
                                "|salary:"+s.getSalary());
}
```

- 判断是否有学生获得奖学金

判断奖学金

```
boolean b=false;
for(int i=0;i<stu.length;i++)
{
        Student s=stu[i];
        if(s.getAverage()>=95||s.getSalary()<=12000)
        System.out.println(s.getName()+"获得奖学金");
        b=true;
}
if(b!=true)
        System.out.println("无人获得奖学金！");
```



例程输出结果

```
Class Number:0811
Name:Mary|Average:95.0|salary:50000
Name:John|Average:78.0|salary:30000
Name:Bill|Average:84.0|salary:10000
Name:Rose|Average:67.0|salary:40000
Mary获得奖学金
Bill获得奖学金
```

- 数组浅复制例程：

```
class Point{
        int x,y;
        public Point(){x=0;y=0;}
        public Point(int x,int y){        this.x=x;this.y=y;    }
        void setXY(int x,int y){          this.x=x;this.y=y;    }
        public String toString(){return("["+x+","+y+"]");}
}
public class ArrayCopyEx{
        public static void main(String args[]){
                Point[] pArr=new Point[10];
                for(int i=0;i<pArr.length;i++){pArr[i]=new Point(i,i);}
                Point[] pArr2=new Point[10];
                System.arraycopy(pArr,0,pArr2,0,10);
                for(int i=0;i<pArr.length;i++){
                        pArr[i].setXY(i+1,i+1);
                        System.out.println(pArr2[i].toString());
                }
        }
}
```

当需要通过类A的一个对象a来复制一个完全相同的新对象b、并且此后对b任何改动都不会影响到a中的值时，使类A实现clone()方法是其中最简单，也是最高效的手段。

在java.lang.Object类中提供clone()方法，将返回Object对象的一个拷贝。

```
class A implements Cloneable{
    public int aInt;
    public Object clone(){
        A b = null;
        try{
            b = (A)super.clone();
        }catch(CloneNotSupportedException e){
            e.printStackTrace();
        }
        return b;
    }
}
```

上述类A实现了clone()方法，当A的某个对象a调用该方法时，即可得到一个复制的新对象。

- 包含引用类型成员变量的对象的复制

```
class CloneSimp{
    int data;
    public CloneSimp(int data){this.data=data;}
    void setDataDouble(){data=2*data;}
    public String toString(){return ""+data; }
}

class CloneNoSimp implements Cloneable{
    int data;
    CloneSimp c=new CloneSimp(5);
    public CloneNoSimp(int data){this.data=data;}
    public Object clone(){
        CloneNoSimp cns=null;
        try{
                cns=(CloneNoSimp)super.clone();
        }catch(CloneNotSupportedException e){
                e.printStackTrace();
        }
        return cns;
    }
}
```

```
public class CloneEx {
    public static void main(String args[]){
            CloneNoSimp source=new CloneNoSimp(10);
            System.out.println("before clone:source:data="+source.data+";c="+source.c);

            CloneNoSimp dest=(CloneNoSimp)source.clone();
            System.out.println("after clone:dest:data="+dest.data+";c="+dest.c);

            dest.data=20;
            dest.c.setDataDouble();

            System.out.println("after clone&double:source:data="+source.data+";c="+source.c);
            System.out.println("after clone&double:dest:data="+dest.data+";c="+dest.c);
    }
}
```

```
before clone:source:data=10;c=5
after clone:dest:data=10;c=5
after clone&double:source:data=10;c=10
after clone&double:dest:data=20;c=10
```

从上述运行结果中可以看出，其中对象的简单数据类型的成员变量得到了复制，更改**dest**的**data**值时，并未将**source**的**data**值同时做改变，可见这是一个复制的全新对象；

但也可看出，其中的复合数据类型成员变量**c**，改变**dest**的**c**也就同时改变了**source**的**c**，也就是说**c**并未得到克隆，他们仍然是简单的引用复制，指向同一个内存空间。

若要实现深度的复制，则需要让**CloneSimp**类也实现**Cloneable**接口以及**clone()**方法，并在**CloneNoSimp**的**clone()**方法中手动添加对该复合数据类型成员变量调用**clone()**方法的语句。具体实现请自行完成。

4.7 多例设计模式

- 在设计中出现需要一个类只能产生固定几种对象的情况

构造方法私有化是多例设计的核心。

```
class Color{
    private String title;
    private static final Color RED=new Color("red");
    private static final Color GREEN=new Color("green");
    private static final Color BLUE=new Color("blue");
    private Color(String title){
        this.title=title;
    }
    public static Color getInstance(int ch){
        switch(ch){
            case 1:return RED;
            case 2:return GREEN;
            case 3:return BLUE;
            default:return null;
        }
    }
    public String toString(){return this.title;}
}
public class MultiP{
    public static void main(String[] args){
        Color c1=Color.getInstance(1);
        System.out.println(c1);
    }
}
```

- 枚举是简化的多例设计模式
  - 关键字enum用来定义枚举类型。上例可以改写为：

```
enum Color{
    RED,GREEN,BLUE;
}
public class EnumDemo1{
    public static void main(String[] args){
        Color c1=Color.RED;
        System.out.println(c1);
    }
}
```

  - 使用enum定义的类型相当于默认继承自Enum类，可以使用Enum类提供的方法

```
for(Color c:Color.values()){
    System.out.println(c.ordinal()+"-"+c.name());
}
```

values():将枚举类中的全部对象以对象数组的形式返回；

ordinal():取得当前枚举对象的序号；

name():取得当前枚举对象的名字

- 枚举也是类，因此可以包括属性、方法、构造方法，并实现接口等。

```
enum Sex{
    MALE("male"),FEMALE("female");
    private String title;
    private Sex(String title){
        this.title=title;
    }
    public String toString(){
        return this.title;
    }
}
class Member{
    private String name;
    private int age;
    private Sex sex;
    public Member(String name,int age,Sex sex){
        this.name=name;
        this.age=age;
        this.sex=sex;
    }
    public String toString(){
        return "name:"+this.name+",age:"+this.age+",sex:"+this.sex;
    }
}
public class EnumDemo2{
    public static void main(String[] args){
        System.out.println(new Member("John",36,Sex.MALE));
    }
}
```

修改器：

用于修改对象内部状态的实例方法；

一般说来，修改器都会修改对象中某个字段的值，作为一种约定，修改器方法的名称往往以“**set**”开头，加上需要修改的字段名称。

通常修改器方法的返回值为**void**类型，而它的参数是对象的新状态或对现有状态的改变量。

访问器：

只访问对象的状态信息，但不修改它们的实例方法；

访问器会返回对象中某个字段的值，作为一种约定，访问器方法的名称往往以“**get**”或“**is**”开头，加上需要访问的字段名称。

通常访问器方法没有参数，但必须有返回值。

课堂测试

- 设计二维坐标系下的直线类Line
  - 属性
    - 直线的端点
  - 方法
    - 构造方法
    - 成员方法
      - 求直线的长度getLength
      - 求直线的斜率

扩展了解

- 面向对象软件的开发过程

面向对象的程序设计和问题求解力求符合人们日常自然的思维习惯，降低、分解问题的难度和复杂性，提高整个求解过程的可控制性、可监测性和可维护性，从而达到以较小的代价和较高的效率获得较满意效果的目的。

  - 面向对象的分析（OOA：Object Oriented Analysis）
    - 明确用户需求，并用标准化的面向对象的模型规范地表述这一需求，最后将形成面向对象的分析模型，即OOA模型。
    - 这一阶段应由用户和开发人员共同完成。
  - 面向对象的设计（OOD：Object Oriented Design）
    - 将对OOA模型加以扩展，引入界面管理、任务管理和数据管理三部分内容，得到OOD模型，并对最初的OOD模型做进一步的细化分析、设计和验证。在明确各对象类的功能和组成时，充分利用已存在的、可获得的对象类或组件。

- 在较大型的开发项目中，通常设置专人专门负责管理所有的可重用资源，将这些资源组织成类库或其他的可重用结构。

- 面向对象的实现（OOP：Object Oriented Programming)

  将程序看做由一系列对象组成，而不是由一系列动作组成。
  我们需要做的是将整个程序拆分成若干个不同的对象，由每个对象来完成特定的功能，而不再是将整个任务划分为子任务。

  - 就是具体的程序编码阶段，主要任务如下：
    - **选择一种合适的面向对象的编程语言；**
    - **用选定的语言编码实现详细设计过程；**
    - **用相关的公式、图表、说明和规则等对软件系统各对象与类进行详尽的描述；**
    - **将编写好的各个类代码块根据类的相互关系集成；**
    - **利用开发人员提供的测试样例和用户提供的测试样例分别检验代码完成的各个模块和整个软件系统。**

- 在面向对象的开发过程中，测试工作可以随着整个实现阶段编码工作的深入而同步完成。
- 编程阶段完成后进入运行阶段。

- 面向对象的特性
  - 抽象性（Abstraction）
    - 是具体事物一般化的过程，即对具有特定属性的对象进行概括，从中归纳出该类对象的共性，并从通用性的角度描述共有的属性和行为特征。
    - 抽象使得系统能够抓住事物的实质特征，因此具有普遍性，可以使用在不同的问题中。
    - 抽象包括以下两方面内容：
      - **数据抽象：描述某类对象的共同属性**
      - **方法抽象：描述某类对象的行为特征**

- Java中，通过定义属性和行为来表述对象的特征和功能，通过定义接口来描述它们的地位以及与其他对象的关系，最终形成一个广泛联系的、可理解、可扩充、可维护、更接近于问题本来面目的动态对象模型系统。

- 封装性（Encapsulation）
  - 利用抽象数据类型将数据和基于数据的操作封装在一起，即把对象的属性和行为结合成一个独立的相同单位，并尽可能隐蔽对象的内部细节。
  - 封装的特点是使某类能建立起严格的内部结构，保护好内部数据，减少外界的干扰，以保证类自身的独立性，可工作在不同的环境中。
  - 对象以外的部分不能随意存取对象的内部数据（属性），从而有效的避免了外部错误对它的错误影响，使软件错误局部化，进而大大减少查错和排错的难度。

- 继承性（Inheritance）
  - 是面向对象程序设计中最具魅力的特色，是软件复用的一种形式，对降低软件的复杂性行之有效。使得程序结构清晰、降低编码和维护的工作量，提高了系统效率。
  - Java中的所有类都是通过直接或间接的继承java.lang.Object类得到。
  - 采用对象但没有继承性的语言是基于对象的语言，不是面向对象的语言。
  - 新类由已经存在的类（父类）生成，通过保留它们的属性和行为，并且根据新类自身的特色和要求加以修改，增加新的属性和行为。通过继承得到的类称为子类，被继承的类称为父类。
  - 类继承就是子类继承父类的成员变量和方法，作为自己的成员变量和方法，就好像它们是在子类中直接声明的一样。当然，子类能否继承父类的变量和方法还有一定的限制。

- 子类从父类继承主要包括以下两方面：
  - **属性**
  - **方法**
- 继承的分类
  - **单继承**
  - **多继承：子类从一个以上的父类继承，Java不支持多继承**

- 多态性（Polymorphism）
  - 指程序中同名的不同方法共存的情况下，Java根据调用方法时传送参数的多少及传送参数的类型来调用具体不同的方法，即可采用同样的方法获得不同的行为特征。在运行时自动选择正确的方法进行调用称为动态绑定（Dynamic Binding）。
  - 多态性可提高程序的抽象程度，使得一个类在使用其他类的功能、操作时，不必了解该类的内部细节，只需明确它提供的外部接口。

- 多态的两种情况
  - **覆盖（Override）：子类对继承自父类的方法重新定义**
  - **重载（Overload）：在同一个类中定义多个同名的不同方法**
- 面向对象程序设计方法的优点
  - 可重用性
  - 可扩展性
  - 可管理性
  - 可自律性
  - 可分离性
  - 接口和消息机制