
面向对象1

类与对象的关系

- 类(**class**)是对一类相似对象的描述，这些对象具有相同的属性和行为、相同的变量(数据结构)和方法实现。类定义就是对这些变量和方法实现进行描述。
- 类就像是对象的模板，实例化一个类，就得到该类的一个对象。有了类定义后，就可以生成该类的一个对象。例如：制造汽车的设计图纸 汽车
- 类 实例化 对象
- 类型 变量
- int i
- struct student zhangsan
- Rectangle object1, object2

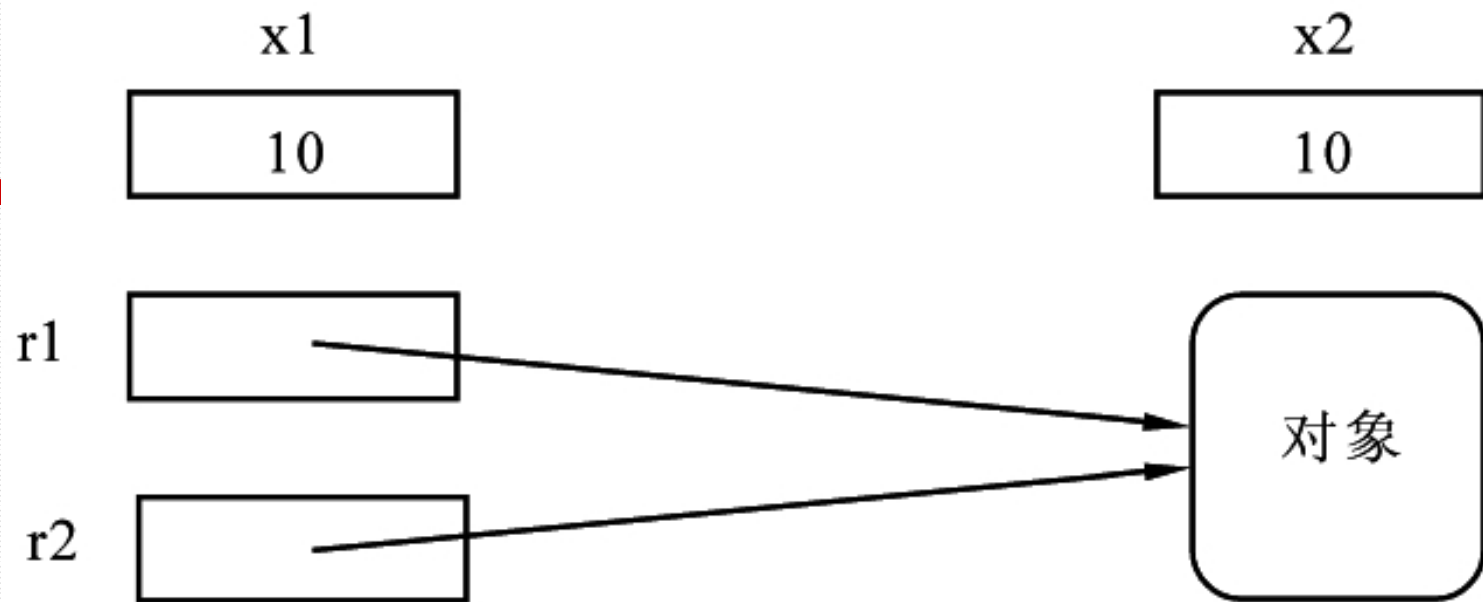
引用类型

Java数据类型分为基本类型和复合类型两大类。其中复合类型又称为引用类型。

把复合类型称为引用类型，是因为复合类型变量里存储的并不是复合类型数据(即对象)本身，而是指向复合类型数据的引用(或引用值)。

对一个对象的引用值，除了可以判断其类型和对其进行类型转换之外，并没有其他操作可言。但对对象的实例变量的访问以及对对象的实例方法的调用都需要通过该引用值进行。

如下代码：



- 1) `int x1, x2;`
- 2) `Rectangle r1, r2;`
- 3) `x1 = 10;`
- 4) `r1 = new Rectangle();`
- 5) `x2 = x1;`
- 6) `r2 = r1;`

Java程序结构:

§ **package**语句: 零个或一个, 必须放在文件开始

§ **import**语句: 零个或多个, 必须放在所有类定义之前

§ **InterfaceDefinition**: 零个或多个

§ **public ClassDefinition**: 零个或一个

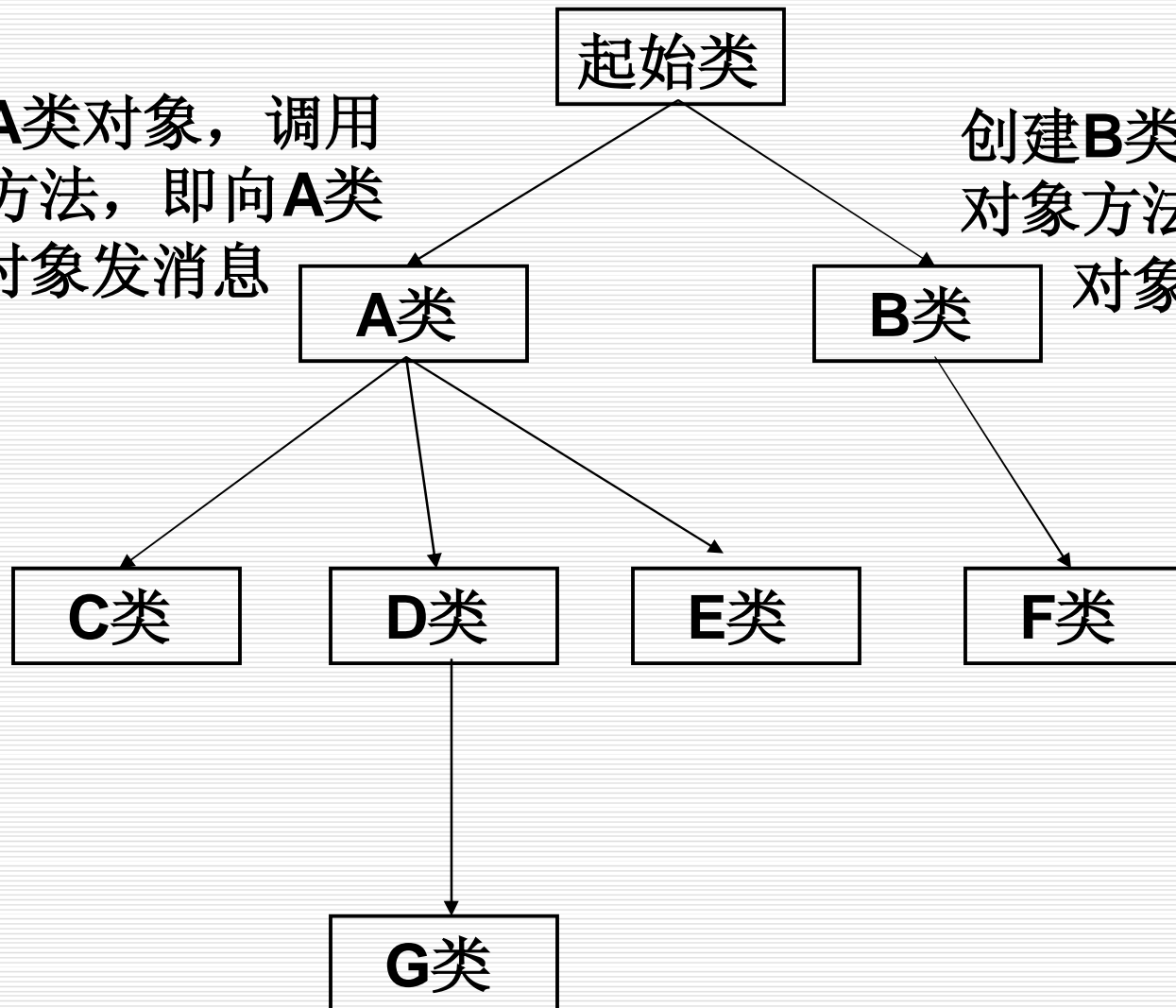
§ **ClassDefinition**: 零个或多个

类个数: 至少一个类, 最多只能有一个**public**类

源文件命名: 若有**public**类(主类), 源文件必须按该类命名

标识符: 区分大小写

创建**A**类对象，调用
对象方法，即向**A**类
对象发消息



类的定义

一个类包括两部分：

1.类声明

2.类体

例5.1 类的定义

class Rect

```
{  
    double length;  
    double width;  
}
```

在类的声明里，说明了类的名字及其它属性。

类体为该类的对象提供了在生存期内需要的所有代码。包括以下几部分：

1. 构造方法(**constructor**)
2. 成员变量(**member variable**)的声明
3. 方法(Method)的实现

例5.2 类的定义

```
class Rect{  
    double length;  
    double width;  
    double area()  
    { return length* width; }  
    void setDim(double w, double l){  
    { width=w;  
        length=l;  
    }  
}
```

Java中的类、方法和变量——类定义的语法规范

n 类的严格定义及修饰字

```
[类的修饰字] class 类名称 [extends 父类名称][implements 接口名称列表]
{
    变量定义及初始化;
    方法定义及方法体;
}
```

类体，其中定义了该类中所有的变量和该类所支持的方法，称为成员变量和成员方法。

类的修饰字: **[public] [abstract | final]**

§ 变量的定义及修饰字

[变量修饰字] 变量数据类型 变量名1,变量名2[=变量初值]...;




[public | protected | private] [static] [final] [transient][volatile]

成员变量的类型可以是Java中任意的数据类型，包括简单类型，类，接口，数组。在一个类中的成员变量应该是唯一的。

变量在每次被线程访问时，该线程都必须从共享内存中重读该成员变量的值。而像串行的变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。

§方法的定义及修饰字



```
[方法修饰字] 返回类型 方法名称(参数1,参数2,...) [throws exceptionList]
{
    ...(statements;) //方法体: 方法的内容
}
```

[public | protected | private] [static] [final | abstract] [native] [synchronized]

返回类型可以是任意的Java数据类型，当一个方法不需要返回值时，返回类型为void。

参数的类型可以是简单数据类型，也可以是引用数据类型（数组、类或接口），参数传递方式是值传递。

方法体是对方法的实现。它包括局部变量的声明以及所有合法的Java指令。局部变量的作用域只在该方法内部。

对象

1. 创建对象
2. 对象的使用
3. 清除对象

1 创建对象

通过 **new** <类名> ([<实参表>]) 来创建一个对象, 也称实例化此类。

```
Rectangle rect = new Rectangle();
```

创建一个对象包括两部分:

1. 声名部分
2. 实例化部分

1 创建对象

○ 声明对象

声明对象的名字和类型，用类名来说明对象的类型。

格式：**type name;**

说明：

- (1) 声明对象的规则与变量声明规则相同，但对象变量是引用类型；
- (2) 在java里类和接口都可以作为数据类型来使用；
- (3) 对象声明通知编译器**name**用来引用**type**类型的变量
- (4) **对象声明并不创建新的对象。**

例：

Rectangle rect;

1 创建对象

○初始化对象

每个类都至少有一个构造方法，当创建对象时调用指定的构造方法来初始化对象,例:

```
Rectangle rect = new Rectangle();
```

```
Rectangle rect = new Rectangle(10, 20);
```

```
Rectangle rect = new Rectangle(new Point(44,78));
```

注：在定义类时如未定义构造方法系统，java会自动构造一个没有参数的构造方法(用**javap**说明)

○构造方法的定义：用于创建指定类的一个实例。其具体功能包括：

- (1) 为实例分配内存空间；
- (2) 初始化实例变量；
- (3) 调用构造方法，返回对该实例的一个引用。

特点：

- (1) 无返回类型
- (2) 方法名与类名相同
- (3) 创建对象时自动调用构造方法
- (4) 构造方法可被重载。

例如：

1 创建对象

构造函数的定义:

```
public class Thing
{
    private int x,y=0;
    public Thing()
    { x = 100; }
    public Thing(int newX)
    { x = newX; }
    public Thing(int newX,int newY)
    { x = newX; y=newY }
}
```

1 创建对象

构造函数的使用:

```
public class TestThing {  
    Thing th1=new Thing();  
    //th1.x ,th1.y?  
    Thing th2=new Thing(1);  
    //th2.x ,th2.y?  
    Thing th3=new Thing(10,10);  
    //th3.x ,th3.y?  
}
```

2 对象的使用

使用对象包括：

- 从对象中获得信息
- 改变对象的状态
- 使对象执行某些操作

实现途径：

- 引用对象的变量
- 调用对象的方法

引用对象的变量

引用对象变量的格式:

对象引用.对象的变量

例:

```
rect.origin = new Point(15, 37);  
area = rect.height * rect.width;  
height = new Rectangle().height;
```

说明: 可以像使用其它变量一样来使用对象的变量。

例如:

```
area = rect1.height * rect1.width
```

.调用对象的方法

格式: 对象引用.对象方法名();
或 对象引用.对象方法名(参数表);

例

```
rect.move(15, 37);  
new Rectangle(100, 50).area()
```

说明:

- .对于有返回值的方法,方法调用可以用于表达式中
- .调用一个对象的方法即是向该对象发送一个消息

3 清除对象

java运行时系统有一个垃圾回收进程负责清除不再使用的对象。

- 垃圾回收器

垃圾回收器定期扫描内存，对于被应用的对象加上标记，按可能的路径扫描结束后清除未加标记的对象。

- 被回收的对象是

不再被任何引用变量引用的对象

- 引用变量自动放弃

- 人为地将引用变量置为null



3 清除对象

- 垃圾收集器将作为优先级低的单独线程运行
- 可通过下列方式关闭应用程序中的垃圾收集

java -noasyncgc ...

- 如果关闭了垃圾收集，程序极有可能会因为内存存在某个时刻耗尽而失败



3 清除对象

使用**finalize**方法:

- Java 提供了一种与 C++ 语言中的析构器相似的方式，可用于在控制返回操作系统前完成清除过程
- 如果存在 **finalize()**，它将在垃圾收集前被执行一次，而且每个对象仅执行一次
- protected void finalize() throws Throwable**
- 可以建议垃圾收集，但并不能保证它何时会发生



```
class A
{
    public void f()
    {
        System.out.println("A.f()");
    }
    protected void finalize()
    {
        System.out.println("-----");
    }
    public static void main(String[] args)
    {
        A a=new A();
        a.f();
        a=null;
        System.gc();
        System.out.println('aaaaaaaaaa');
    }
}
```

根据使用、设计Rec类

```
class TestRec
```

```
{ public static void main(String args[])
```

```
{ Rec rect1 = new Rec();
```

```
  Rec rect2 = new Rec();
```

```
  double area;
```

```
  rect1.setDim(10, 20); // 初始化每个长方形
```

```
  rect2.setDim(3, 6);
```

```
  area = rect1.getArea(); // 调用getArea方法
```

```
  System.out.println("第一个长方形的面积是: "+area);
```

```
  area = rect2.getArea(); // 调用getArea方法
```

```
  System.out.println("第二个长方形的面积是: "+area);
```

```
}
```

```
}
```

计算长、宽分别为10、20和3、6的两个长方形面积。本程序用构造方法来初始化长方形的大小

```
class Rect
{
    double length;
    double width;
    Rect(double l, double w)
    {   length = l;   width = w;   }
    double getArea()
    {   return length * width;   }
}
```

```
class RectDemo
```

```
{ public static void main(String args[])
```

```
{ Rect rect1 = new Rect(10,20);
```

```
Rect rect2 = new Rect(3,6);
```

```
double area;
```

```
area=rect1.getArea(); //调用area方法得到第一个长  
方形的面积
```

```
System.out.println("第一个长方形的面积是:"+area);
```

```
area=rect2.getArea(); //调用area方法得到第二个长  
方形的面积
```

```
System.out.println("第二个长方形的面积是:"+area);
```

```
}
```

```
}
```

构造方法的重载

```
class RectOverload
```

```
{ double length;
```

```
  double width;
```

```
  double area()
```

```
  { return length * width; }
```

```
RectOverload(double l, double w)
```

```
{ length = l; width = w; }
```

```
RectOverload(double s)
```

```
{ length = s; width = s; }
```

```
}
```

```
class RectDemo
```

```
{ public static void main(String args[])
```

```
{   RectOverload rect1 = new RectOverload(10,20);
```

```
    //初始化一个长方形
```

```
    RectOverload rect2 = new RectOverload(6);
```

```
    double ar;
```

```
    ar=rect1.area();//调用area得到一个长方形的面积
```

```
    System.out.println("长方形的面积是: " + ar);
```

```
    ar=rect2.area();//调用area得到一个正方形的面积
```

```
    System.out.println("正方形的面积是: " + ar);
```

```
    }
```

```
}
```

成员修饰符（访问权限/非访问权限）

访问权限修饰符：public, private, protected, friendly(缺省)

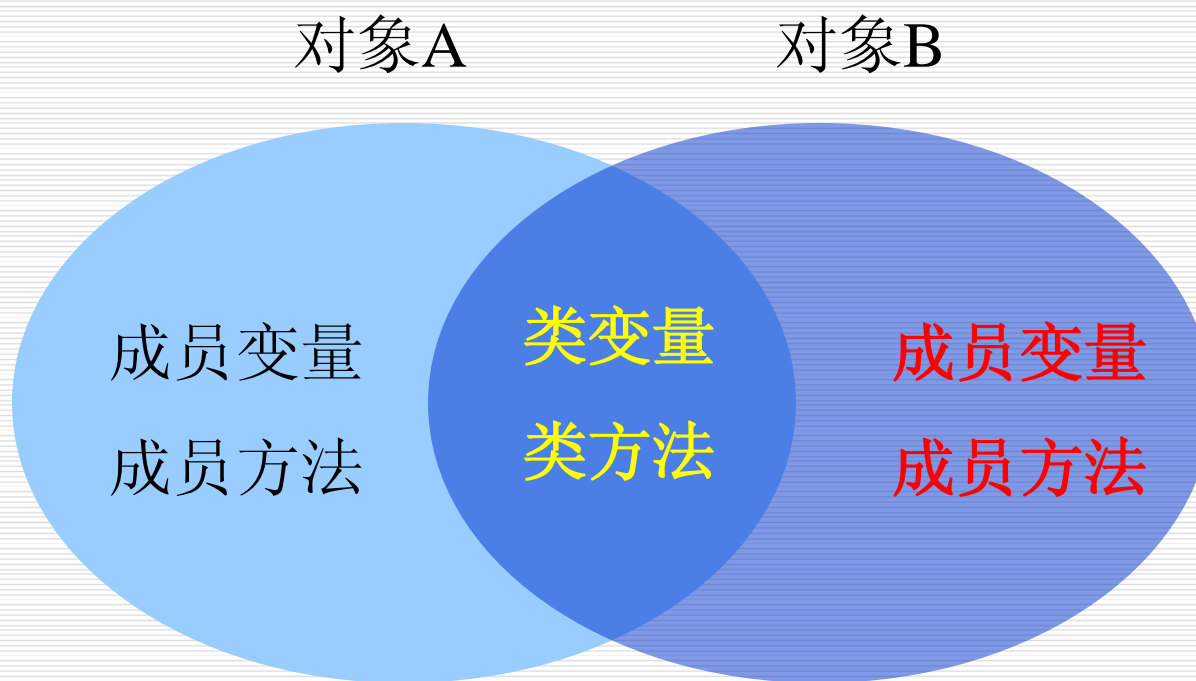
非访问权限修饰符：static, final, abstract, native等

用static修饰符修饰的成员变量和方法成员，成为静态成员(类成员)。静态成员存储于类的存储区，属于整个类，不属于任何一个类的具体对象，而是被所有该类对象共享。

特点：

- (1) 它被保存在类的内存区的公共存储单元中，而不是保存在某个对象的内存区中。因此，一个类的任何对象访问它时，存取到的都是相同的数值。
- (2) 可以通过类名加点操作符访问它。
- (3) static类数据成员仍属于类的作用域，还可以使用public static、private static等进行修饰。修饰符不同，可访问的层次也不同。

类成员

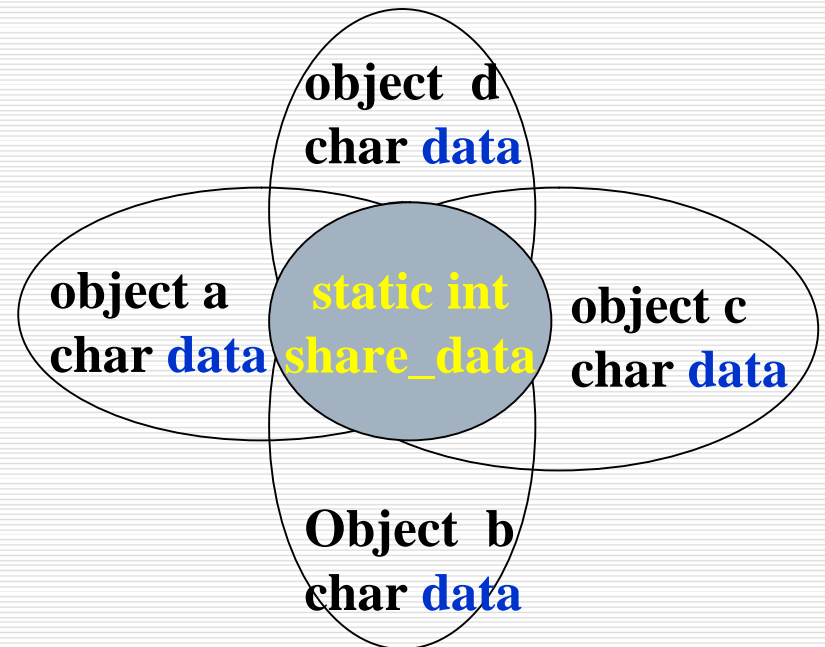


static: 类变量和类方法

static 在变量或方法之前，表明它们是属于类的，称为类方法（静态方法）或类变量（静态变量）。若无**static**修饰，则是实例方法和实例变量。

类变量在各实例间共享

```
class ABCD
{
    char data;
    static int share_data;
}
class Demo
{
    ABCD a,b,c,d;
}
```



```
class Test1
{
    static int i;
    int j;
    public static void method1()
    {
        i=100;
        j=200;
        System.out.println("11111");
    }
}
```

对静态成员的使用要注意

(1)静态方法不能访问属

量，而只能处理属于整个类的成员变量，
即**静态方法只能处理静态变量**。

(2)可以用两种方式调用静态成员，它们的作用相同。

变量：类名.变量、类对象.变量

方法：类名.方法名()、类对象.方法名()

```
class Example
{   static double ad=8;   }
public class staticDemo
{
    public static void main(String args[] )
    {
        Example m1=new Example();
        Example m2=new Example();
        m1.ad=0.1;
        System.out.println("m1.ad="+m1.ad);
        System.out.println("ad="+ Example.ad);
        System.out.println("m2.ad="+m2.ad);
    }
}
```

程序运行结果如下：

a=42

b=99


静态成员的使用

```
class StaticDemo{
    static int a=42;    //静态变量
    static int b=99;
    static void callme() { //静态方法
        System.out.println("a="+a);
    }
}

class StaticByName{
    public static void main(String args[]) {
        //不需创建对象，通过类名直接调用静态方法
        StaticDemo.callme();
        //通过类名直接调用静态变量
        System.out.println("b="+StaticDemo.b); }
}
```

静态方法，实例方法对静态变量的访问

```
class Test
{ static int i=8;
  int j;
  public void method()
  { i=100;
    j=200; }
  public static void method1()
  { i=200;
    j=200; }
}
```



静态数据成员的初始化

静态数据成员的初始化可以由用户在**定义时**进行，也可以由**静态初始化器**来完成。静态初始化器是由**关键字static引导的一对花括号括起的语句块**，其作用是在加载类时，初始化类的静态数据成员。静态初始化器与构造方法不同，它有下列特点：

- (1)静态初始化器用于对类的**静态数据成员**进行初始化。而构造方法用来对**新创建的对象**进行初始化。
- (2)静态初始化器**不是方法**，没有方法名、返回值和参数表。
- (3)静态初始化器是在它所属的类加载到内存时由**系统调用**执行的，而构造方法是在系统用new运算符产生新对象时自动执行的。

```

class CC
{
    static int n;
    int nn;
    static //静态初始化器
    { n=20;} //初始化类的静态数据成员n
    CC() //类CC的构造方法
    { nn=n++; }
}

public class StaticDemo
{
    public static void main(String args[ ])
    {
        CC m1=new CC();
        CC m2=new CC();
        CC m3=new CC();
        System.out.println("m1.nn="+m1.nn);
        System.out.println("m2.nn="+m2.nn);
        System.out.println("m3.nn="+m3.nn);    }
}

```

换成给nn赋值行不？

运行结果：

m.nn=20

m1.nn=21

m2.nn=22

static: 类变量和类方法

§ 同一个类的实例方法可以访问该类的类变量和类方法(即静态变量和静态方法);

n 而类方法只能访问该类的类变量和类方法, 不能直接访问实例的变量和方法。

static: 类变量和类方法

§ 不正确的引用

```
class StaticError
{
    String mystring="hello";
    public static void main(String args[])
    {
        System.out.println(mystring);
    }
}
```

编译时错误信息: nonstatic variable mystring cannot be referenced from a static context "System.out.println(mystring);"

为什么不正确: 只有对象的方法可以访问对象的变量。

static: 类变量和类方法

n 解决的办法

1. 将变量改成类变量

```
class NoStaticError
{
    static String mystring="hello";
    public static void main(String args[])
    {
        System.out.println(mystring);
    }
}
```

static: 类变量和类方法

n 解决的办法

2. 先创建一个类的实例

```
class NoStaticError
{ String mystring="hello";
  public static void main(String args[])
  {
    NoStaticError noError;
    noError = new NoStaticError();
    System.out.println(noError.mystring);
  }
}
```

final修饰符

如果一个类的数据成员用**final**修饰符修饰，则这个数据成员就被限定为最终数据成员。

最终数据成员可以在声明时进行初始化，也可以通过构造方法赋值，但不能在程序的其他部分赋值，它的值在程序的整个执行过程中是不能改变的。所以，也可以说用**final**修饰符修饰的数据成员是标识符常量，习惯上使用大写的标识符表示**final**变量。例如：

```
final double PI=3.1416;
```

```
final double G=9.18;
```

用**final**修饰符说明常量时，需要注意以下几点：

- (1)需要说明常量的数据类型并指出常量的具体值。
- (2)若一个类有多个对象，而某个数据成员是常量，最好将此常量声明为**static**，即用 **static final**两个修饰符修饰，这样做可节省空间。例如：

```
static final double PI=3.1416;
```

```
class CA
{ static int n=20;
  final int m; //声明m, 但没有赋初值
  final int k=40; //声明k并赋初值40
  CA()
  { m=++n; } //在构造方法中给nn赋值
}

public class StaticDemo
{ public static void main(String args[])
  { CA ca1=new CA();
    CA ca2=new CA();
    System.out.println("ca2.m="+ca2.m);
    System.out.println("ca2.k="+ca2.k);
    System.out.println("ca1.m="+ca1.m);
    System.out.println("ca1.k="+ca1.k);
  }
}
```

运行结果:

ca2.m=22

ca2.k=40

ca1.m=21

ca1.k=40

ca1.m=90;

×

局部变量

局部变量是定义在**块内、方法内的变量**。

使用局部变量，要注意以下几点：

- (1) 这种变量的作用域是以**块和方法为单位的**，仅在定义该变量的块或方法内有效，
- (2) 要**先定义赋值，然后再使用**，即不允许超前引用。
- (3) **局部变量**可以与**类变量、实例变量同名**。因为局部变量在查找时首先被查找，因此若某一局部变量与类的实例变量名或类变量名相同时，则该实例变量或类变量在方法体内被暂时“屏蔽”起来，只有退出这个方法后，实例变量或类变量才起作用。
- (4) 在局部变量的作用域中，当语句块有嵌套时，**内层语句块定义的变量不能与外层语句块的变量同名**。
- (5) **方法体内不能定义静态变量**。

实例变量和类变量

定义在类内、方法外的变量是实例变量，使用了修饰符static的变量是静态变量（或称类变量）。

实例变量和类变量的作用域是以类为单位的。因为实例变量、类变量与局部变量的作用域不同，故可以与局部变量同名。

【例】说明局部变量和实例变量、类变量。

说明实例变量和局部变量同名问题

```
class A
{
    int x = 8; //实例变量
    void f(){
        //局部变量与实例变量同名，屏蔽了实例变量
        int x = 6;
        System.out.println(" x = " + x);
    }
}
```

方法f输出的结果为：x = 6
(注：这个程序不能直接运行)

```
class MyObject
{ static short s = 400; //类(静态)变量
  int i = 200; // 实例变量
  void f()
  { System.out.println("s = " + s);
    System.out.println("i = " + i);
    short s = 300; // 局部变量
    int i = 100; // 局部变量, 与实例变量同名
    System.out.println("s = " + s);
    System.out.println("i = " + i); }
}

class TestMyObject
{ public static void main(String args[])
  { MyObject myObject = new MyObject();
    myObject.f(); }
}
```

实例变量与局部变量同名

```
1) class Example0405
2) {   int x;
3)     void method(int x)
4)     {   x = x + this.x;
5)         this.x = (int)(x>=0 ? Math.sqrt(x) : Math.abs(x));
6)         System.out.println(" x=" + this.x);
7)     }
8)     public static void main(String args[])
9)     {   Example0405 e=new Example0405();
10)        e.method(25);
11)        e.method(-1); }
12) }
```

结果是:

x=5

x=2

成员变量、局部变量的作用域

```
class c {  
    int i;    //成员变量  
    void method1(){  
        int i;    //局部变量  
        { int i;    } //非法  
    }  
    int method2(){  
        { int j=10;    }  
        System.out.print(j); //非法  
    }  
}
```

实例变量、类变量和局部变量的小结

- 1 局部变量与成员变量（实例变量、类变量）的变量名可以相同。
- 2 局部变量的语句块发生嵌套时，内层语句块定义的变量不能与外层语句块定义的变量同名。
- 3 成员变量名可与成员方法名相同。
- 4 方法的参数也属于局部变量。
- 5 局部变量必须显式初始化，成员变量不必显式初始化，在系统调用无参构造函数时有一个默认值。

private访问控制符

```
class P1
{
    private int n=9;    //私有数据成员n
    int k;
    P1()
    {    k=n++;    }
    void method()
    {    System.out.println("n="+n);    }
}
```

```
public class privateDemo
{
    public static void main(String args[])
    {
        P1 m1=new P1();
        System.out.println("m1.k="+m1.k);
        System.out.println("m1.n="+m1.n);
        m1.method();
    }
}
```

private成员
不能引用

用private修饰的数据成员或成员方法**只能被该类自身所访问和修改**，而不能被任何其他类(包括该类的子类)来访问和引用。它提供了最高的保护级别。当其他类希望获取或修改私有成员时，需要借助于类的方法来实现，即**只能通过取数和送数的方法来访问**。

这样的方法常命名为getxxx()和setxxx()等。

成员变量的隐藏

§可以用方法来实现
对成员变量的隐藏:

q 设置变量方法:

setVariable()

q 获取变量方法:

getVariable()

```
class Sample
{
    private int x;
    .....
    void setX(int var )
    {
        x = var;
    }
    int getX()
    {
        return x;
    }
    .....
}
```

```

class RectangleRC
{
    private double length;
    private double width;
    double getLength() //定义取长方形边长的方法
    {
        return length;
    }
    double getWidth()
    {
        return width;
    }
    RectangleRC(double len,double w)
    {
        length=len;
        width=w;
    }
}

```

程序运行结果如下：
长方形的面积是： 56.0

```

class RectDem05
{
    public static void main(String args[ ])
    {
        RectangleRC rect1=new RectangleRC(8,7);
        double ar=rect1.getLength()*rect1.getWidth();
        System.out.println("长方形的面积是： "+ar);
    }
}

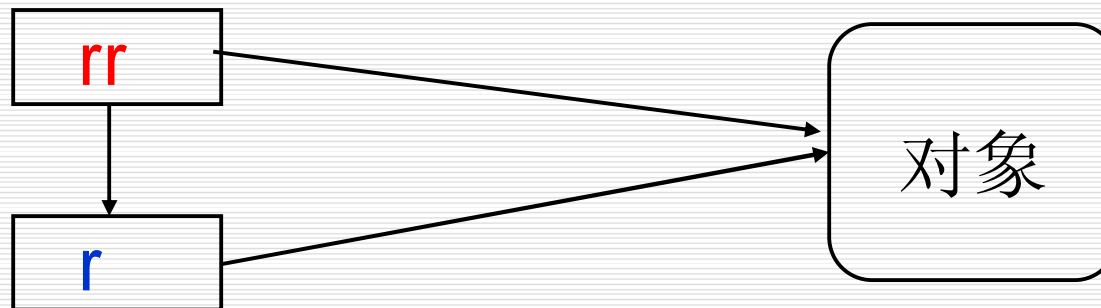
```

方法参数是类的对象

在Java语言中，方法的参数类型除了可以是基本类型外，还可以是引用类型——类。在调用类参数时方法间传送的是引用。

```
class RefParameter{  
    double width,length,area;  
    RefParameter(double w,double l){  
        width = w; length = l;  
    }  
    void calArea(RefParameter r){  
        r.area = r.width * r.length;  
    }  
}
```

```
class PassObject
{ public static void main(String args[]){
    RefParameter rr = new RefParameter(10,20);
    rr.calArea(rr);
    System.out.println("长方形面积为:" + rr.area);
}
}
```



方法返回值为类的对象

```
class RetClass
{ double width,length,area;
  RetClass(double w,double l)
  { width = w;   length = l; }
  RetClass calArea(RetClass rc) //声明方法的返回值类型为引用类型
  { rc.area = rc.width * rc.length;
    return rc;    } // 返回值为引用类型的对象
}
```

```
class ReturnObject
```

```
{ public static void main(String args[])  
  { RetClass r1 = new RetClass(10,20);  
    RetClass r2;  
    r2= r1.calArea(r1);  
    System.out.println("长方形面积为: "+  
                        r2.area);  
  }  
}
```

类对象作为类的成员

若一个类的对象是一个类的成员时，要用 **new** 运算符为这个对象分配存储空间。

```
class RectC
{ double width,length;
  RectC(double w,double l)
  { width = w;
    length = l; }
}
```

```
class RectangleC           //具有两个成员的种类
{   RectC r = new RectC(10,20); //类成员要分配空间
    double area;           //基本类型成员
}

class ClassMember
{   public static void main(String args[])
    {   RectangleC rc = new RectangleC();
        rc.area = rc.r.width * rr.r.length;
        System.out.println("长方形面积为:" + rc.area);
    }
}
```


例 "has-a"关系举例

```
1) class A{
2)     private int x, y;
3)     private B o2; //实例变量为引用变量，指向B类的一个实例
4)     public A(int x, int y)
5)     {         this.x = x;
6)             this.y = y;
7)             o2 = new B(); //通过默认构造方法创建B类的实例
8)     }
9)     public B getOneB()
10)    {         return o2;         //返回实例变量的引用值
11)    }
12)    public void print()
13)    {         System.out.println("x="+x+" y="+y);
14)             o2.print();
15)    }
16) }
```

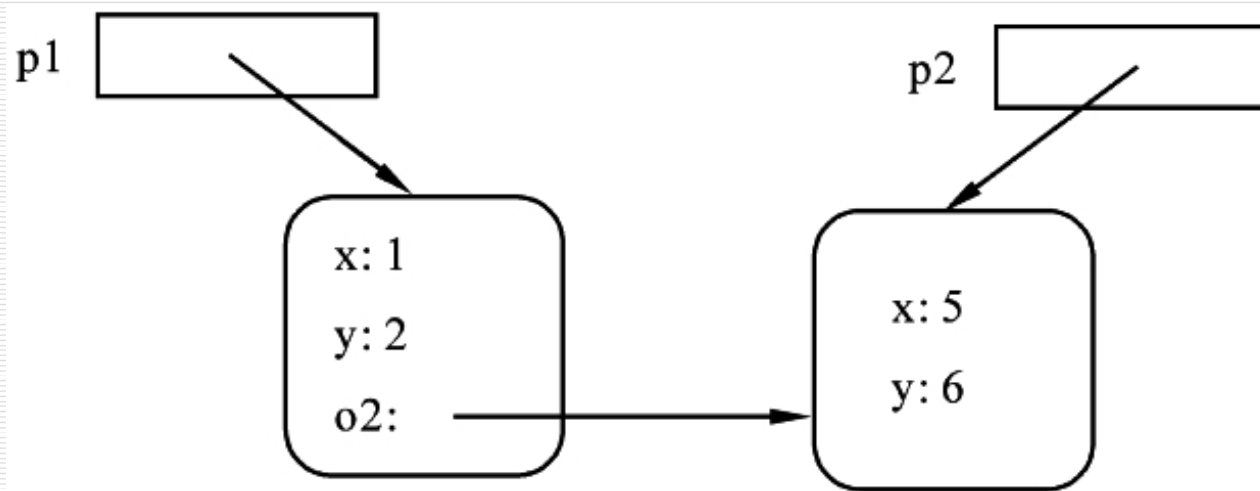
```
18) class B
19) { private int x=3, y=4;
20)     public void setB(int x, int y)
21)     {     this.x = x;
22)           this.y = y; }
23)     public void print()
24)     {     System.out.println("x =" +x+" y=" +y); }
25) }
26) public class ADemo
27) { public static void main(String args[])
28)     { A p1 = new A(1, 2);
29)       B p2 =p1.getOneB();
30)       p2.setB(5, 6);
31)       p1.print();
32)       p2.print(); }
33) }
```

程序运行结果:

x = 1 y = 2

x = 5 y = 6

x = 5 y = 6



下课! 😊

Thank you!