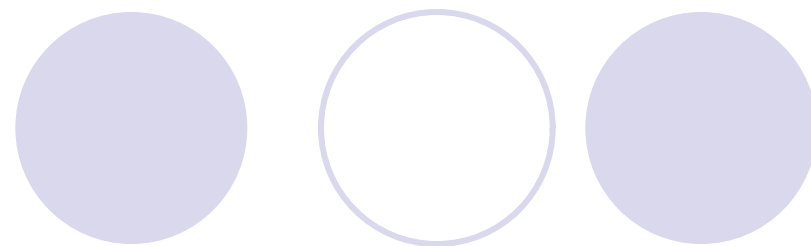


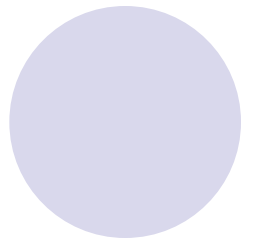
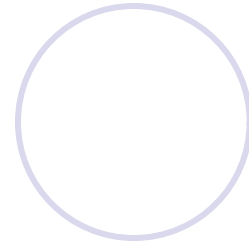
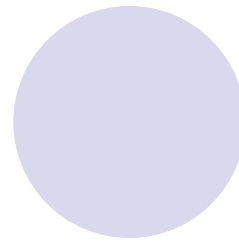
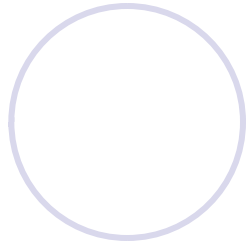
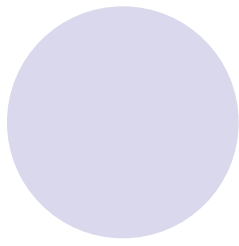
第5章 面向对象编程基础

- 传统的程序设计语言(如C语言)是面向过程的，程序主要由变量和过程组成，程序中任何问题的解决都是靠程序员一行一行地编写代码完成的。它的程序结构：程序=(算法)+(数据结构)，在这个结构中算法和数据结构都是相互独立的，而且以算法(函数或过程)为主。
- 面向对象编程(Object-Oriented Programming，简称OOP)是当今占主导地位的程序设计思想和技术。OOP技术允许用实体(entity)或对象(object)的思想方法来分析和设计应用程序，从而使软件开发过程更接近人的思维过程，并极大地提高了程序设计的效率。

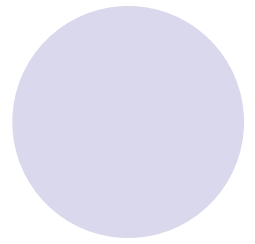
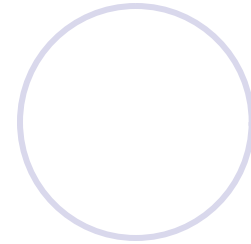
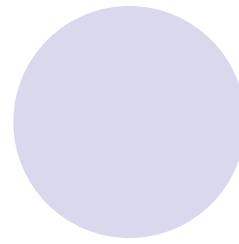
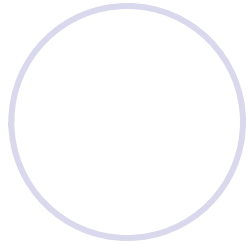
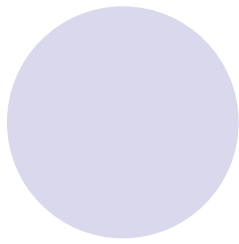
5.1 面向对象概念



- 面向对象程序的结构：程序由许多对象组成，对象是程序的实体，这个实体包含对该对象属性的描述(数据结构)和对该对象进行的操作(算法)。在面向对象的程序设计中通过类机制实现对对象的描述，从而使数据结构不再是一个独立静止的结构，它与相应的算法一起构成对问题对象的完整描述，并且可以嵌套重用，构成对更大问题对象的描述。



- 面向对象程序设计具有三个基本特征：封装、继承和多态，可以大大增加程序的可靠性、代码的可重用性及程序的可维护性，从而提高程序开发效率。
- 类是一组具有相同数据结构和相同操作的对象的集合，是对一系列具有相同性质的对象的抽象，是对对象共同特征的描述。比如每辆汽车是一个对象的话，所有的汽车可以作为一个模板，就定义汽车这个类。
- 比如说所有的大学生都有姓名、班级、学号、专业、性别等特性，还有上课、做作业等行为。这是大学生的共性，是所有大学生的模板，就可以定义大学生这个类。



- 对象是类的一个实例。
- 对象可以是现实生活中的客观物体，还可以是某一个概念。例如：一辆汽车、一个人、一本书，乃至一种语言、一个图形、一种管理方式，都可以作为一个对象。
- 某位大学生叫李红、100411班、03学号、电子信息专业、女生等，是具体的某位学生的特性，是个性，那么该学生就是一个对象。
- 类是在对象之上的抽象，它为属于该类的全部对象提供了统一的抽象描述。所以类是一种抽象的数据类型，是模板；对象则是类的具体化，是类的实例。
- 可以使用类的实例化定义对象，表示创建该类的一个实例。
- “类”和“对象”常常混淆，所以从一开始就正确区分它们非常重要。

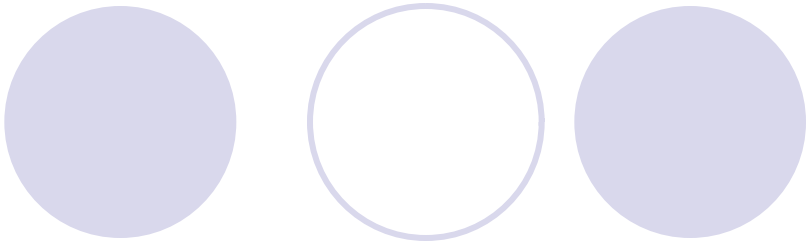
5.2 类

- 类(class)是C#语言的核心，所有的语句都位于类内。**.NET Framework**类库包含大量解决通用问题的类，一般可以通过创建自定义类和使用**.NET Framework**类库来解决实际问题。

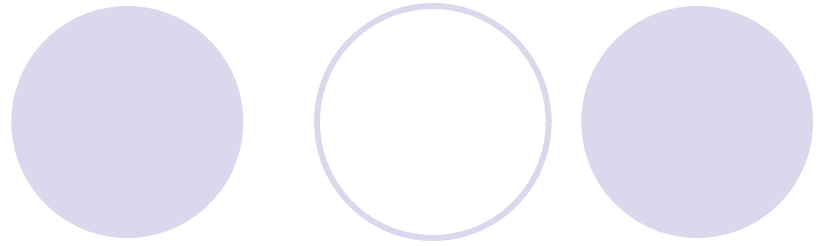
5.2.1 类的声明

- 使用类声明可以创建新的类。具体形式如下：
- **[特性][类修饰符][partial]class 类名[类型形参][:基类或接口[类型形参约束]]**
- **{**
- 类体;
- **}**
- **C#**类声明完整语法涉及内容比较复杂，下面只简单介绍各部分的意义，将在后续章节展开详细介绍。
- **[特性](可选)**：用于附加一些声明性信息。
- **[类修饰符](可选)**：用于定义类的可访问性等信息，可以是以表5-1所列的几种之一或者它们的有效组合，但在类声明中同一修饰符不允许出现多次。默认情况下，在命名空间中声明的任何类都是**internal**。
- **[partial] (可选)**：表示将类的定义拆分到两个或多个源文件中。
- **class**：表示定义类的关键字。
- **[类型形参] (可选)**：用于泛型类声明。
- **[： 基类或接口[类型形参约束]] (可选)**：用于声明要继承的类或接口。
- **类体**：用于定义该类成员，包含在一对花括号之间，类体可以为空

表5-1 类修饰符

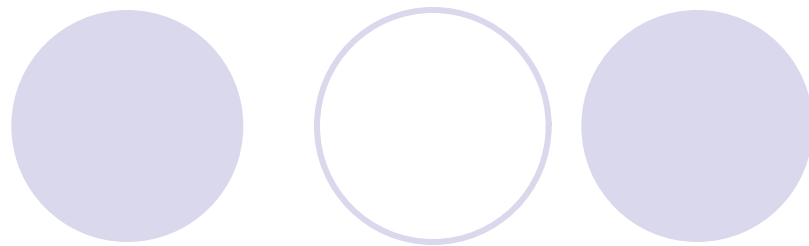


修 饰 符	描 述
new	只允许用在嵌套类中，表示所修饰的类会隐藏继承下来的同名成员
public	所属类的成员及非所属类的成员都可以访问
internal	当前程序集可以访问
private	只有所属类的成员才能访问
protected	所属类或派生自所属类的类型可以访问
abstract	表示是一个抽象类，该类含有抽象成员，因此不能被实例化，只能作为基类



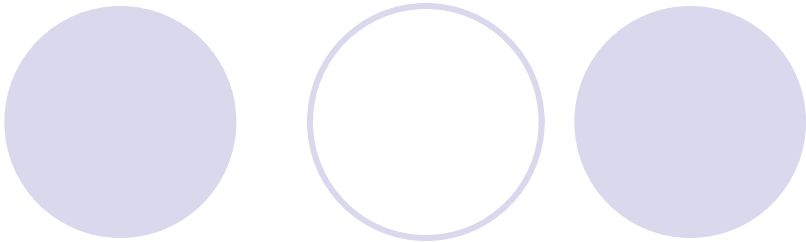
- ## //类体为空语句

5.2.2 类的成员

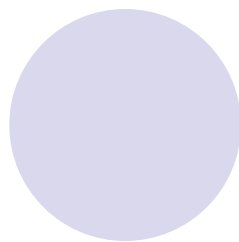
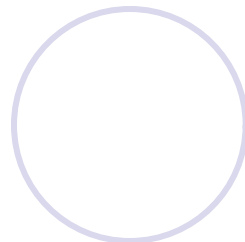
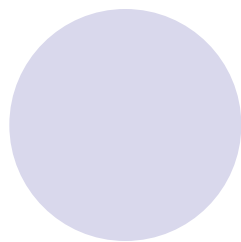
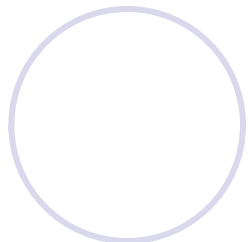
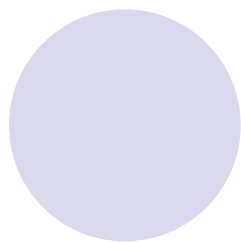


- 类的定义包括类头和类体两部分，类体用于定义该类的成员。
- 类成员由两部分组成，一部分是类体中以类成员声明形式引入的类成员，另一部分则是直接从它的基类继承而来的成员。类成员如表**5-2**所示。

表5-2 类成员



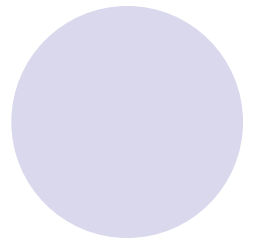
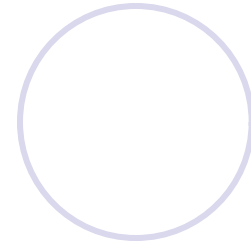
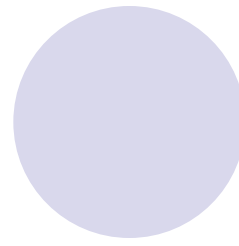
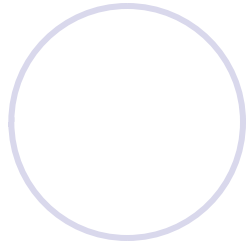
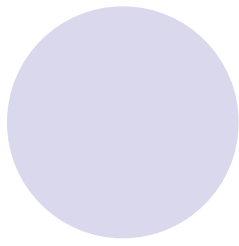
成 员	描 述
常数	与类相关联的常量值
字段	类的变量
方法	类可以执行的计算和操作
属性	定义一些命名特性以及与读取和写入这些特性相关的操作
索引器	与以数组相同的方式对类的实例进行索引相关联的操作
事件	可以由类产生的通知
运算符	类所支持的转换和表达式运算符
构造函数	初始化类的实例或类本身所需的操作
析构函数	在永久删除类的实例之前执行的操作
类型	类所声明的嵌套类型



- 当字段、方法、属性、事件、运算符或构造函数声明中含有**static**修饰符时，表明它们是静态成员，否则就是实例成员。访问或引用静态成员的格式：“类名.静态成员名”；访问或引用实例成员的格式：“实例名.实例成员名”。
- 类成员声明中可以使用如表5-3所示的5种访问修饰符中的一种：

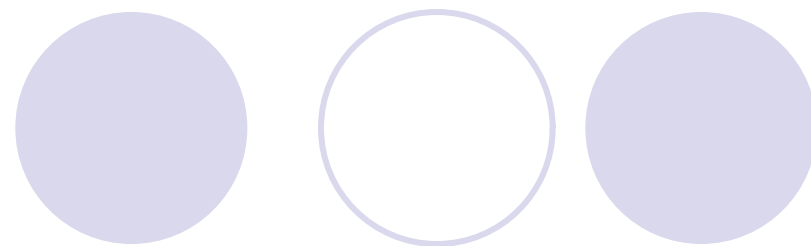
表5-3 访问修饰符

类成员的可访问性	描 述
public	访问不受限制
protected	访问仅限于此类或从此类派生的类
internal	访问仅限于此程序
protected internal	访问仅限于此程序或从此类派生的类
private	访问仅限于此类

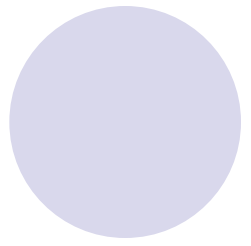
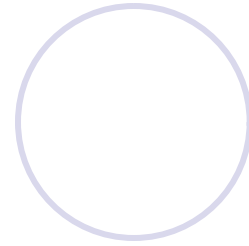
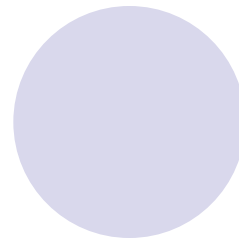
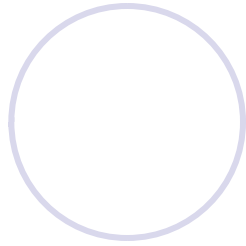
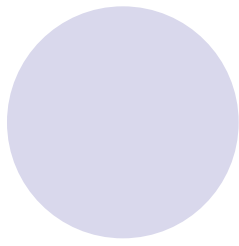


- 当类成员声明不包含访问修饰符时，默认约定访问修饰符为**private**。
- **【例5-2】** 声明一个类，类中包含两个字段成员，一个常量成员。
- **public class students**
- **{**
- **const int mm = 20;** //常量成员
- **public string name;** //字段成员，姓名(public)，访问不受限制
- **protected int age;** //字段成员，年龄(protected)，访问仅限于此类或从此类派生的类
- **}**

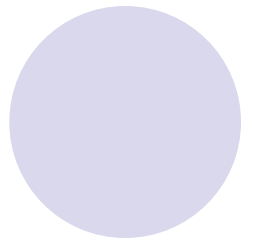
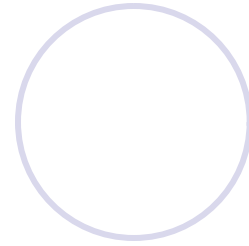
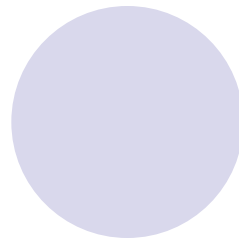
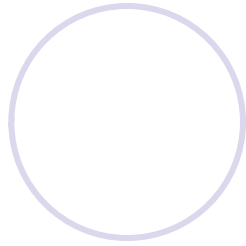
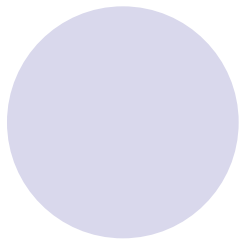
5.2.3 构造函数



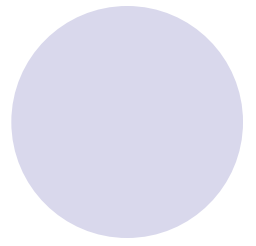
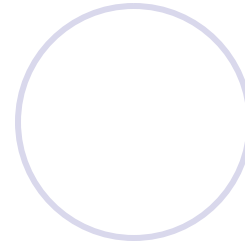
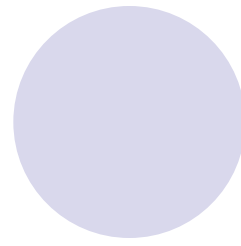
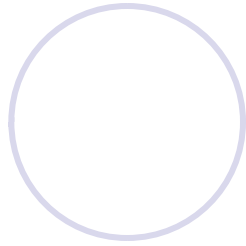
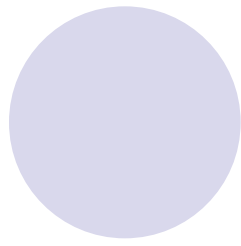
- 当定义了一个类之后，就可以通过**new**运算符将其实例化，产生一个对象。为了能规范、安全地使用这个对象，**C#**提供了对对象进行初始化的方法，这就是构造函数。
- 在**C#**中，类的成员字段可以分为实例字段和静态字段，与此相应的构造函数也分为实例构造函数和静态构造函数。



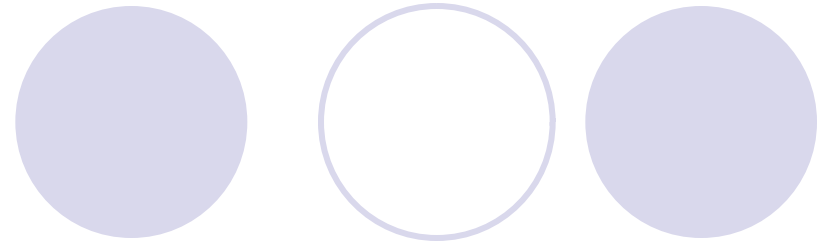
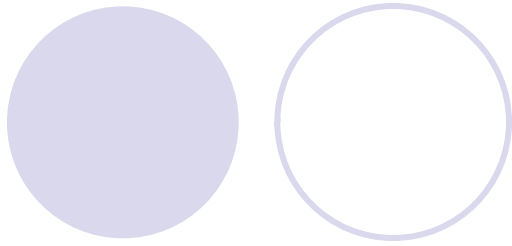
- 1. 实例构造函数声明
- 实例构造函数的声明语法为：
- [访问修饰符] 类名 ([参数列表]) [: **base** ([参数列表])] [: **this** ([参数列表])]
- {
- //构造函数的主体
- }
- 一般地，构造函数总是**public**类型的。如果构造函数是**private**类型的，表明类不能被外部类实例化。



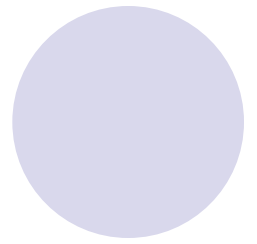
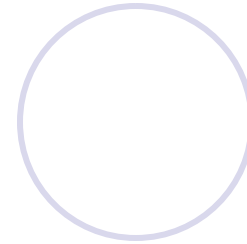
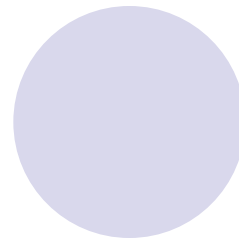
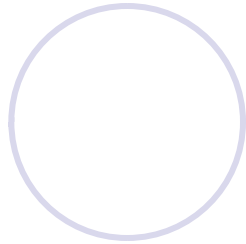
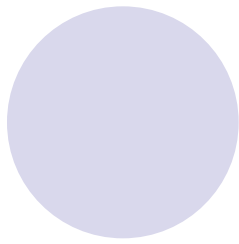
- 类名([参数列表]): 类名必须与这个类同名, 不声明返回类型, 并没有任何返回值。构造函数参数可以没有, 也可以有一个或多个。这表明构造函数在类的声明中可以有多个函数名相同, 但参数个数不同或者参数类型不同的多种形式, 这就是所谓的构造函数重载。



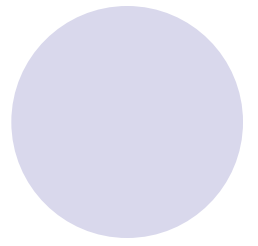
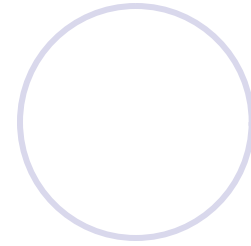
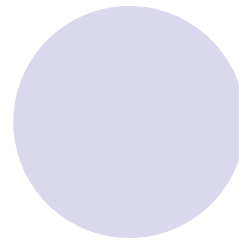
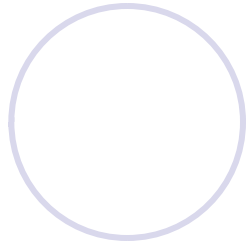
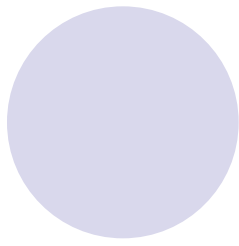
- 【例5-3】 创建一个学生类，类成员中包含一个字段成员**age**，一个静态成员**studcount**，一个构造函数，在构造函数中完成静态成员加1的功能来统计学生人数(新建一个对象，自动调用一次构造函数，里面的**studcount**就自动加1)。运行界面如图5.1所示，代码如下：



```
● public class students
● {
●     public string name;    //实例成员
●     public int age;
●     public static int studcount=0; //定义一个静态字段成员，用于统计学生的人数
●     public students()      //定义无参数的构造函数
●     {
●         studcount++;       //统计学生人数
●     }
●     //定义有一个参数的构造函数。构造函数重载，名字相同，但参数类型或参数格式
不同
●     public students(string s_name)
●     {
●         this.name = s_name;    //把构造函数的形参s_name传递给类成员name
●         studcount++;          //统计学生人数
●     }
● }
```

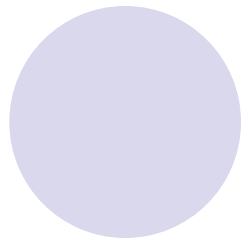
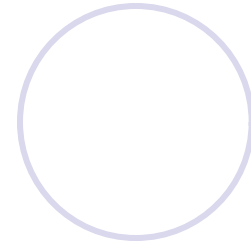
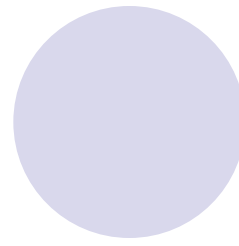
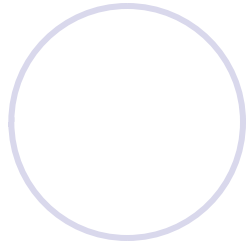
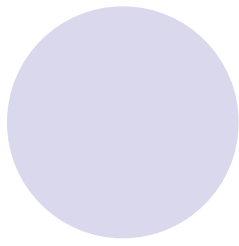


```
class Program
{
    static void Main(string[] args)
    {
        students student1 = new students();
        //用new创建类的一个实例student1
        //{0}为占位符，studcount是静态成员，前面应该是类名
        Console.WriteLine("学生人数={0}", students.studcount);
        students student2 = new students();
        //用new创建类的一个实例student2
        //{0}为占位符，studcount是静态成员，不能写成student2.studcount
        Console.WriteLine("学生人数={0}", students.studcount);
        student2.age = 21; //引用实例成员，student2.age
        Console.WriteLine("student2的年龄是：{0}", student2.age);
        students student3 = new students("lihong");
        //创建一个实例student3，实参是“lihong”
        Console.WriteLine("student3的姓名是:{0}", student3.name);
        Console.WriteLine("学生人数={0}", students.studcount);
        Console.ReadLine();
    }
}
```

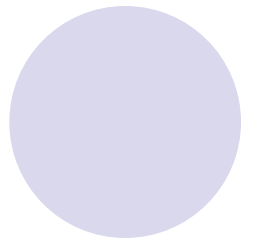
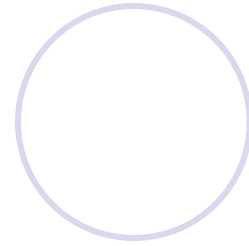
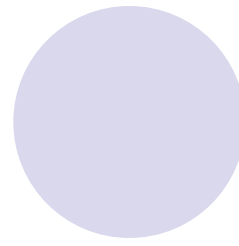
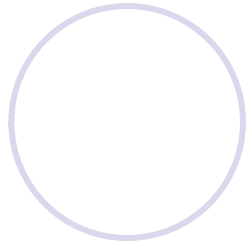
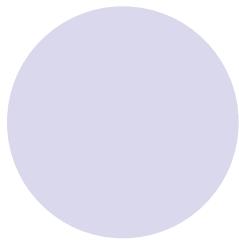


```
C:\ file:///E:/zm/c#exampl...
学生人数=1
学生人数=2
student2的年龄是: 21
student3的姓名是: lihong
学生人数=3
```

图5.1 例5-1运行界面



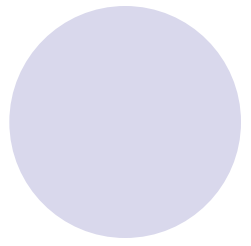
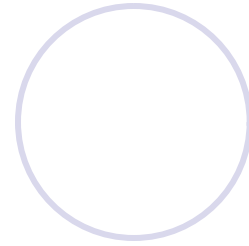
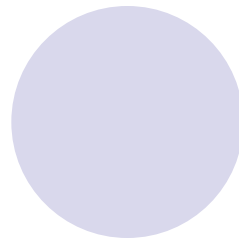
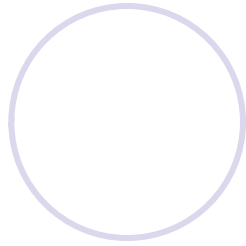
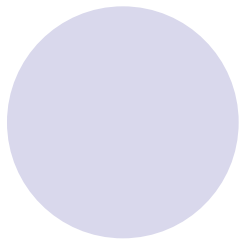
- 2. 静态构造函数声明
- 静态构造函数声明语法格式：
- **static**类名
- {
- //静态构造函数内容
- }
- 静态构造函数体：静态构造函数的目的是用于对静态字段进行初始化，它只能对静态数据成员进行初始化，而不能对非静态数据成员进行初始化。



- 静态构造函数是不可继承的，而且不能被直接调用。只有创建类的实例或者引用类的任何静态成员时，才能激活静态构造函数，所以在给定的应用程序域中静态构造函数至多被执行一次。如果类中没有声明静态构造函数，而又包含带有初始设定的静态字段，那么编译器会自动生成一个默认的静态构造函数。

【例5-4】 创建一个类，并使用静态构造函数。运行界面如图5.2所示，代码如下

```
● public class student
● {
●     public double s_allow;                //津贴
●     public string s_demand;               //培养目标
●     public string s_address;              //住址
●     public static readonly student Bachelor; //静态只读变量
●     public static readonly student Master;
●     public static readonly student Doctor;
●     public student(double allow, string demand, string address)
●     //带参数的构造函数
●     {
●         this.s_allow = allow;
●         this.s_demand = demand;
●         this.s_address = address;
●     }
● }
```



```
● static student() //静态构造函数，初始化静态只读变量
● {
●     Bachelor = new student(200, "studious", "1号楼");
●     //引用静态成员，激活静态构造函数
●     Master = new student(1000, "creative", "2号楼");
●     Doctor = new student(2000, "inventive", "3号楼");
● }
● }
● class Program
● {
●     static void Main(string[] args)
●     { //创建一个student本科生实例，同时初始化。Bachelor是静态的，前面是类名student
●         student student1 = student.Bachelor;
●         Console.WriteLine("该学生每月津贴为: {0}", student1.s_allow);
●         Console.WriteLine("对该学生的培养目标为: {0}", student1.s_demand);
●         Console.WriteLine("该学生住址为: {0}", student1.s_address);
●         Console.ReadLine();
●     }
● }
```


用F11单步运行，看看怎样调用静态构造函数、有参数构造函数，并了解它们调用的顺序。

- 5.2.4 析构函数
- 构造函数名必须与类名相同，但为了区分构造函数，前面需加“~”表明它是析构函数。
- 析构函数不能写返回类型，也不能带参数，因此它不可能被重载，当然也不能被继承，所以一个类最多只能有一个析构函数。一个类如果没有显式地声明析构函数，则编译器将自动产生一个默认的析构函数。
- **【例5-5】** 析构函数实例。
- ```
public class Point
```
- ```
{
```
- ```
 private int x, y;
```
- ```
    ~Point()                //定义的析构函数，与类同名
```
- ```
 {
```
- ```
        Console.WriteLine ("Point's destructor ");
```
- ```
 }
```
- ```
}
```

5.3 方法

- 方法是包含一系列语句的代码块，通过这些代码块能够实现预先定义的计算或操作。如电话是一个对象，该对象有几个行为(接听、响铃、挂断)，那么就可以在电话类中定义这几个方法，分别完成接听、响铃、挂断的功能，如图5.3所示。

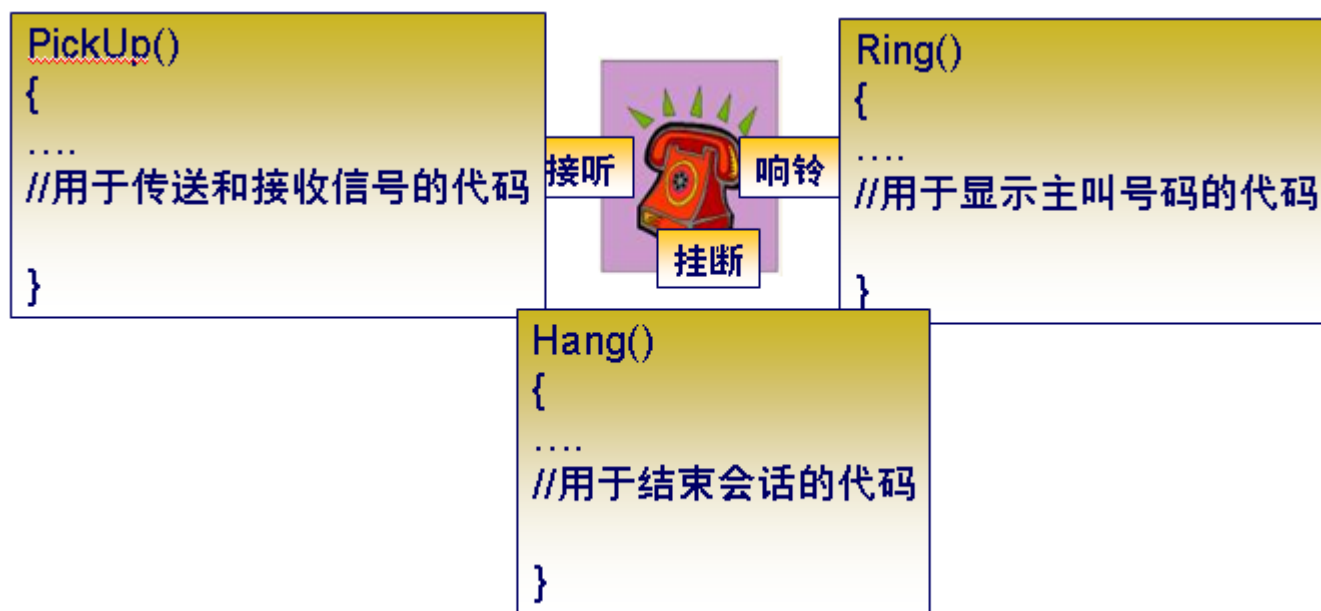
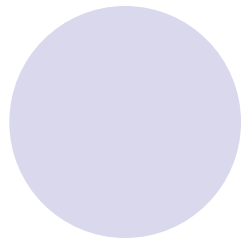
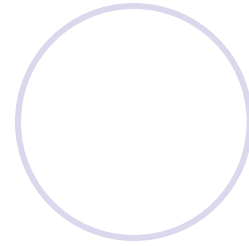
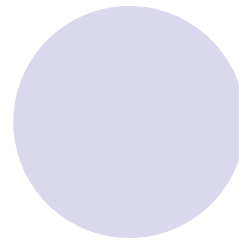
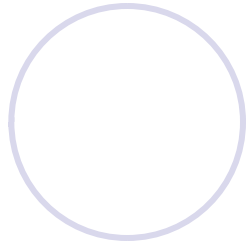
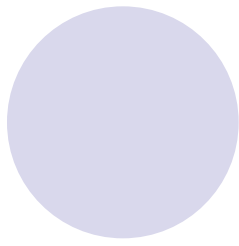


图5.3 电话对象的几个方法



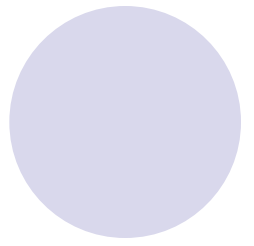
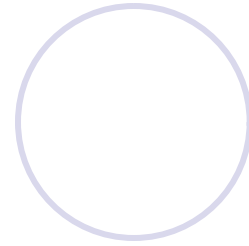
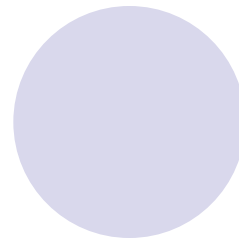
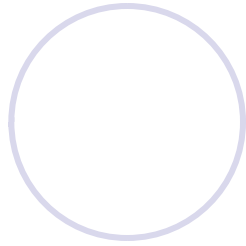
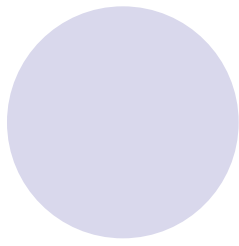
- 方法是与类相关的函数，在**C#**中，每个函数都必须与类或结构相关。**.NET Framework**框架中提供了很多类的方法。其中**Math**类方法可完成某些常见的数学计算。表**5-4**总结了一些**Math**类的方法。

5.3.1 方法的声明

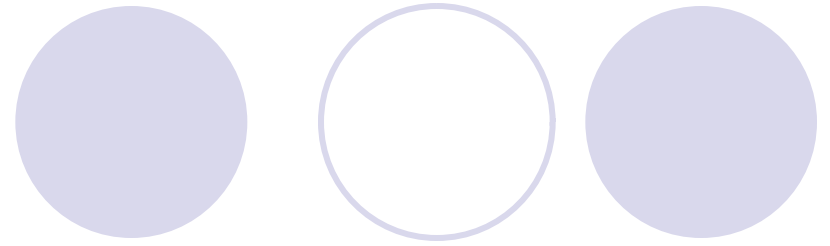
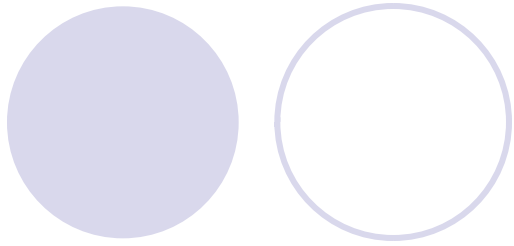
- 方法用于实现由类执行的计算和操作，是以函数的形式来定义的。
- [访问修饰符] 返回类型 方法名 ([参数列表])
- {
- 方法主体
- }
- 其中，访问修饰符(可选)指定方法的可访问性等，默认情况下为**private**。方法的修饰符如表5-5所示。

表5-5 方法的修饰符

修 饰 符	作 用 说 明
new	在一个继承结构中，用于隐藏基类同名的方法
public	该方法可以在任何地方被访问
protected	该方法可以在它的类体或派生类类体中被访问，但不能在类体外访问
private	该方法只能在该类体内被访问
internal	该方法可以被同处于一个工程的文件访问
static	静态方法，表示该方法属于类本身，而不属于某特定对象
virtual	该方法可在派生类中重写，来更改该方法的实现
abstract	该方法仅定义了方法名及执行方式，但没有给出具体实现，所以包含这种方法的类是抽象类，有待于派生类的实现
override	该方法是从基类继承的 virtual 方法的新实现
sealed	是一个密封方法，必须同时包含 override 修饰符，以防止它的派生类进一步重写方法
extern	该方法从外部实现

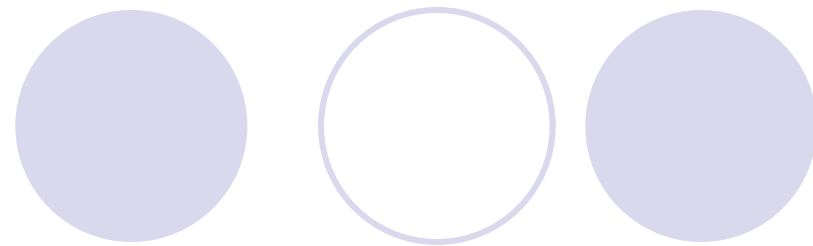


- 返回类型：指定方法计算后返回的值的类型，如果不需要返回任何值，则其返回值类型用**void**；参数列表(用圆括号括起来，并用逗号隔开，可能为空)表示传递给该方法的值或变量引用；方法名是一种标识符；方法体是方法执行的代码块。

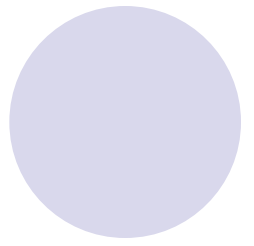
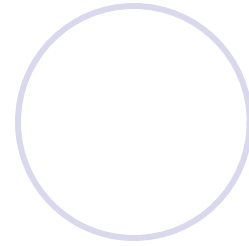
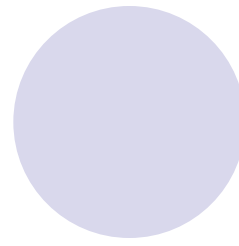
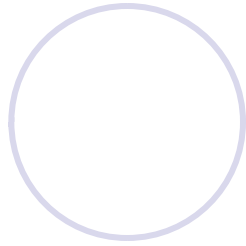
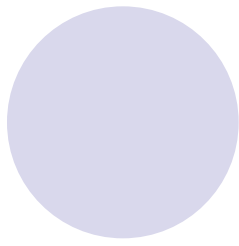


- **【例5-6】** 在类中创建一个方法。
- `public class A` //创建一个类
- `{`
- `public int F(int a, int b)`
- `//在类中定义一个方法，方法名为F，返回类型为int,参数为a,b`
- `{ return a + b; }` //方法体，return为返回，此句是返回a、b的和
- `}`

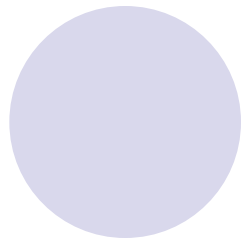
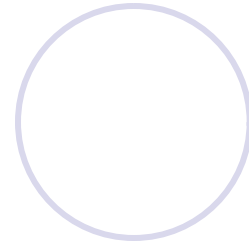
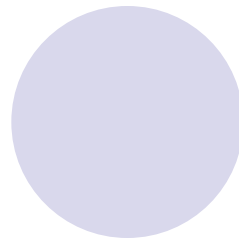
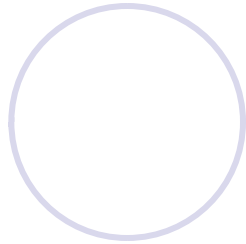
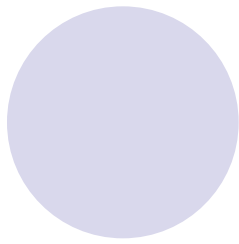
5.3.2 方法的参数



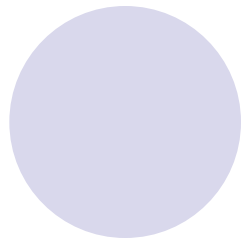
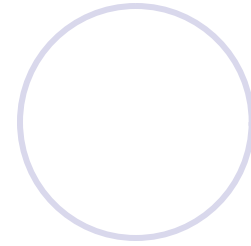
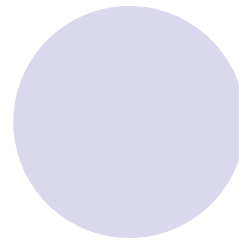
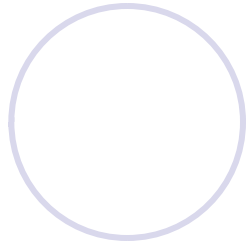
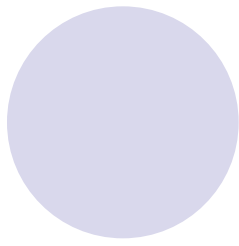
- 方法可以包含一个或多个参数，如果存在多个参数，则参数之间使用逗号分隔。方法的参数有4种类型：
- 值参数：声明参数时不含任何修饰符，如`int i`。
- 引用型参数：声明参数时带有`ref` 修饰符，如`ref int i`。
- 输出参数：声明参数时带有`out` 修饰符，如`out int i`。
- 数组型参数：声明参数时带有`params`修饰符，如`params int[] a`。



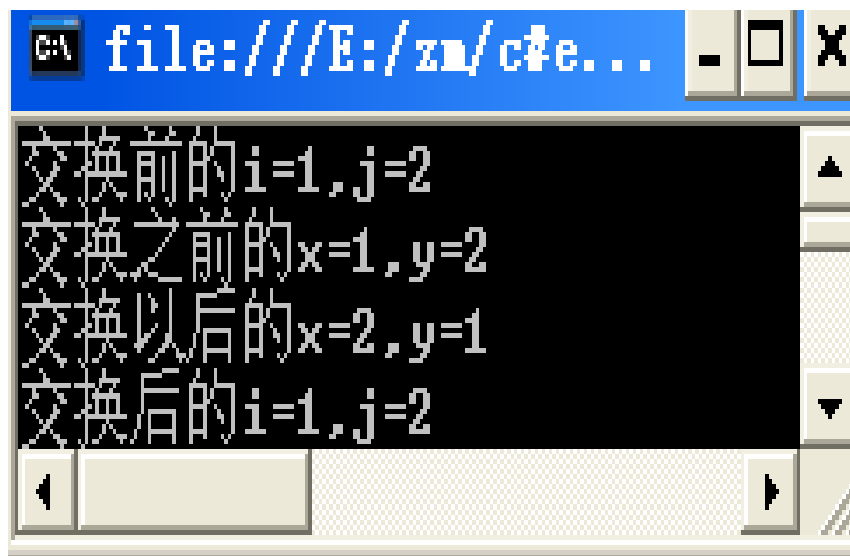

- 1. 值参数
- 在方法声明时不加修饰符的形参就是值参数，当使用值参数调用方法时，编译器将实参的值做一份副本，并将此副本传递给该方法的相应形参。被调用的方法不会修改内存中实参的值，所以使用值参数时，可以保证实参值是安全的。



- **【例5-7】** 值参示例，利用值参进行两个数的交换 ,运行界面如图5.4所示
- `class Program`
- `{`
- `public void Swap(int x, int y) //定义值形参的方法，交换两个形参`
的值
- `{`
- `Console.WriteLine("交换之前的x={0},y={1}", x, y);`
- `//输出两个形参交换之前的值`
- `int temp = x; //交换两个形参`
- `x = y;`
- `y = temp;`
- `Console.WriteLine("交换以后的x={0},y={1}",x,y);`
- `//输出交换以后的形参的值`
- `}`



- `static void Main(string[] args)`
- `{`
- `int i = 1, j = 2; //在Main函数中定义的两个变量`
- `Program tt = new Program(); //创建一个Program类的实例`
- `Console.WriteLine("交换前的i={0},j={1}", i, j);`
- `//输出两个实参交换之前的值`
- `tt.Swap(i, j); //注意怎样调用实例方法`
- `Console.WriteLine("交换后的i={0},j={1}", i, j);`
- `//输出两个实参交换后的值，说明对形参的修改不影响实参`
- `Console.ReadLine();`
- `}`
- `}`

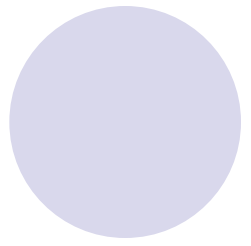
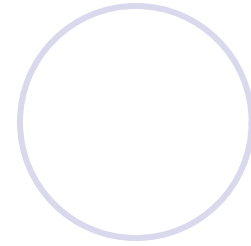
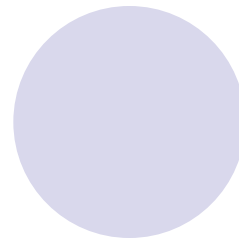
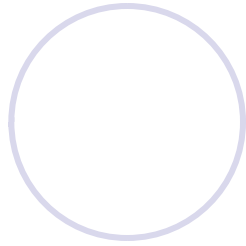
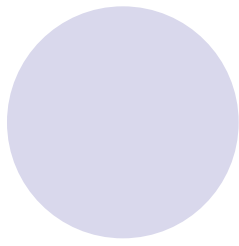


```
file:///E:/zm/c#e...  
交换前的i=1,j=2  
交换之前的x=1,y=2  
交换以后的x=2,y=1  
交换后的i=1,j=2
```

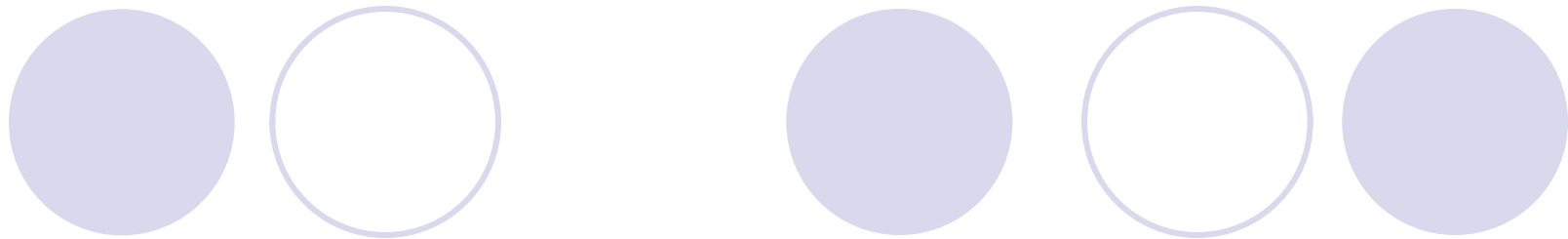
图5.4 例5-7运行结果

【例5-8】 当方法传递的是一个引用对象(如数组)时，对形参的修改会影响到实参(实质上属于引用参数，但参数前没有修饰符)。运行结果如图5.5所示。

- `class MyClass`
- `{` //创建一个方法。形参前没有任何修饰符(是值参数)，但形参本身是数组，是引用类型
- `public void change (int [] a)` //与实参指向同一内存地址
- `{`
- //将a的第一个元素赋值为67，改变形参的第一个元素，调用方法后比较实参的值是否改变
- `a[0] = 67;` //实参、形参仍指向同一内存地址
- //在方法内使用new运算符来分配新的内存地址，将a引用新的数组。此时不与实参指向同
- //一内存地址，因此，在该方法内在此之后的任何更改都不会影响原始数组arr
- `a = new int[5] { -1,-3,0,6,9};`
- `Console.WriteLine("方法内的数组第一个元素是: {0}",a[0]);` //输出形参的第一个元素
- `}`

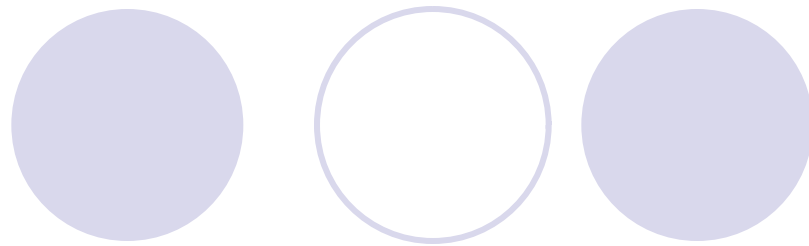
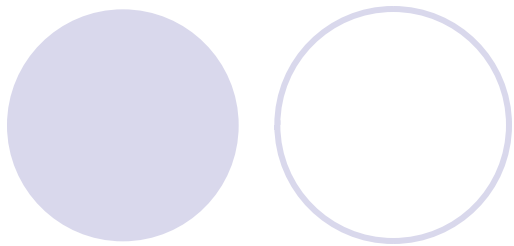


- `static void Main(string[] args)`
- `{`
- `int[] arr = new int[3] { 1,2,3};`
- `Myclass tt = new Myclass();` //创建一个实例
- `//输出调用方法前实参的值`
- `Console.WriteLine("调用方法前的数组的第一个元素是:`
- `{0}", arr[0]);`
- `tt.change(arr);` //调用方法，形参实参指向同一内存地址
- `//调用方法后实参的值已经被改变`
- `Console.WriteLine("调用方法后的数组的第一个元素是:`
- `{0}", arr[0]);`
- `Console.Read();`
- `}`
- `}`

A screenshot of a Windows console window. The title bar is blue and contains the text "file:///E:/xm/c#example/ConsoleA...". The console area is black with white text. It displays three lines of output: "调用方法前的数组的第一个元素是: 1", "方法内的数组第一个元素是: -1", and "调用方法后的数组的第一个元素是: 67". The window has standard Windows controls (minimize, maximize, close) in the top right corner and a scrollbar on the right side.

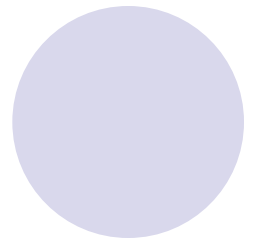
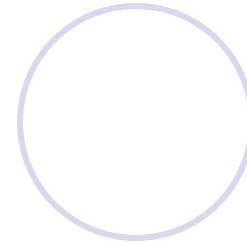
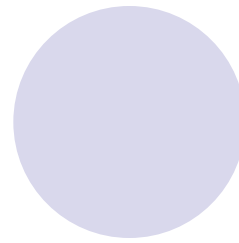
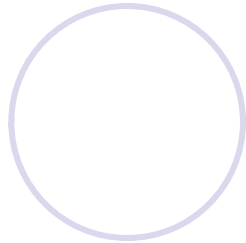
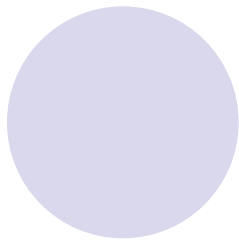
```
file:///E:/xm/c#example/ConsoleA...  
调用方法前的数组的第一个元素是: 1  
方法内的数组第一个元素是: -1  
调用方法后的数组的第一个元素是: 67
```

图5.5 例5-8运行结果



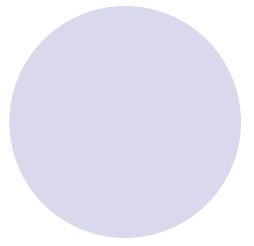
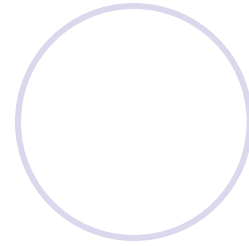
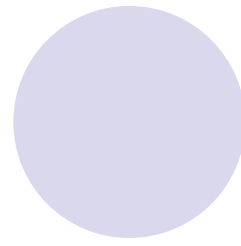
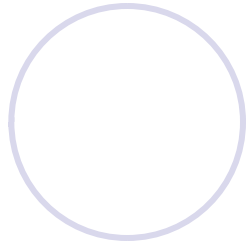
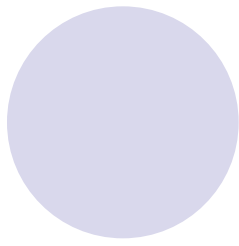
- 2. 引用参数
- 值类型参数传递的是实参值的副本，而引用型参数向方法传递的是实参的地址。
- 引用型参数一般在两种情况下存在：方法的形参本身就是引用类型(值参数的第二种情况，没有**ref**修饰符)；通过使用**ref**关键字使参数按引用传递。
- 【例5-9】 通过**ref**关键字使参数按引用传递。运行结果如图5.6所示，

```
class MyClass
{ //形参x的值通过关键字ref来修饰，改变x的值，将改变输入的实参的值
  public void squareit(ref int x)
  {
    x *= x;
    Console.WriteLine("squareit方法内的 x的值={0}",x);
    //输出方法内形参的值
  }
  static void Main(string[] args)
  {
    int n = 6;
    MyClass tt = new MyClass();
    Console.WriteLine("调用方法前的n值是: {0}", n); //输出没有调用方法前
n的值
    tt.squareit(ref n); //调用方法，参数是引用类型
    //调用方法后，实参的值也随着形参的改变而改变
    Console.WriteLine("调用方法后的n值是: {0}", n);
    Console.Read();
  }
}
```



```
C:\ file:///E:/zm/c#exampl...  
调用方法前的n值是: 6  
squareit方法内的 x的值=36  
调用方法后的n值是: 36
```

图5.6 例5-9运行结果



- 3. 输出参数
- 由引用参数得知，可以使用引用型参数的传递方式来完成从方法返回多个变化的值。**C#**还提供了一种特殊的参数传递方式，专门用于从方法返回数据，完成这种数据传递方式是输出型参数，用关键字**out**表示。
- 与引用型参数类似，输出型参数也不开辟新的内存区域。
- 与引用型参数的差别在于，调用带有**out**关键字参数的方法之前，无须对传递给形参的实参值进行初始化。但是，调用完实参作为输出型参数的方法后，该实参变量将会被方法中的形参赋值，并将数据从方法中传递至调用处。

【例5-10】 使用输出型参数实例。运行结果如图5.7所示，代码如下：

```
class Outclass
{
    public string Outtest(out string y)    //定义带有输出型参数的方法
    {
        y="I am a doctor";    //形参的值发生改变，实参的值也将发生改变
        string t="You are a good doctor"; //方法的返回值
        return t;
    }
    static void Main(string[] args)
    {
        string quzhi;    //变量quzhi用来保存执行方法Outtest后的返回值
        string x;        //未对要做实参的变量x进行初始化
        Outclass tt = new Outclass(); //创建一个实例tt,并调用默认的构造函数
        //调用带输出参数的方法，x是实参，quzhi保存的是方法中return后面表达式的值
        quzhi = tt.Outtest(out x);
        Console.WriteLine("方法的返回值是：{0}",quzhi); //输出方法的返回值
        Console.WriteLine("实参的值是：{0}",x); //输出实参的值，其值已经发生
        变化
        Console.Read();
    }
}
```

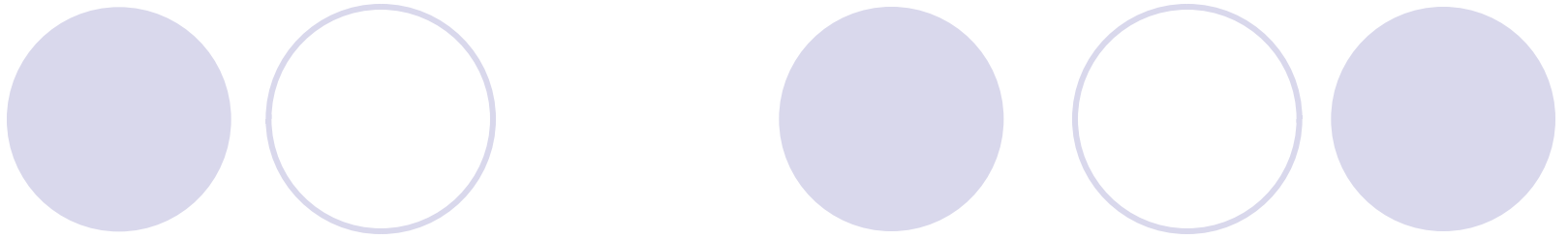
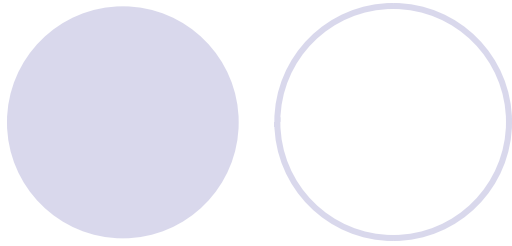
A screenshot of a Windows command prompt window. The title bar shows the file path "file:///E:/zm/c#example/Co...". The window contains two lines of text: "方法的返回值是: You are a good doctor" and "实参的值是: I am a doctor". The text is displayed in a black font on a white background. The window has standard Windows window controls (minimize, maximize, close) in the top right corner and a scroll bar on the right side.

图5.7 例5-10运行结果



● 4. 数组型参数

- 一般而言，调用方法时实参必须与该方法声明的形参在类型和数量上相匹配，但有时候希望更灵活一些，能够给方法传递任意个数的参数。**C#**提供了传递可变长度的参数表的机制——数组型参数，即使用**params**关键字来指定一个参数为可变长的参数表。
- 带有**params**关键字的数组型参数必须是方法的参数列表的最后一个参数，否则会出现编译错误。
- **【例5-11】** 编写并调用具有数组型参数的方法。运行结果如图5.8所示，代码如下：



```
2      3      4      0      -1      -2
2      3      4
x=1      y=2      z=3
```

```
class Paramsclass
```

```
{
    public void test(params int[] list)
```

```
{
```

```
    for (int i = 0; i < list.Length; i++)
```

```
    { list[i]++; }
```

```
    foreach (int i in list )
```

```
    //注意foreach的用法，i是临时变量，作用域在该语句内
```

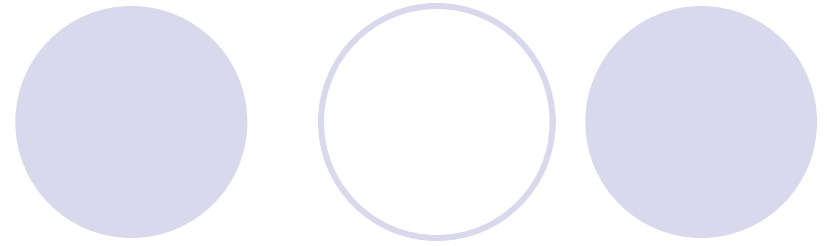
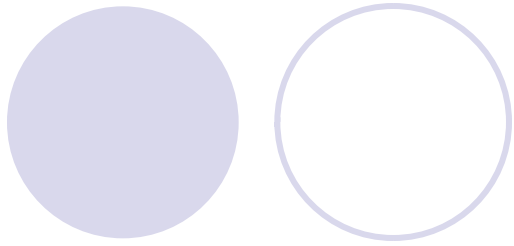
```
    { Console.Write("{0}\t", i); }
```

```
    //“\t”为转义字符，水平制表(跳到下一个Tab位置)
```

```
    Console.WriteLine("\n"); //输出索引的数组元素后换行
```

```
}
```

图5.8 例5-11运行结果



- `static void Main(string[] args)`
- `{`
- `int x = 1, y = 2, z = 3;`
- `Paramsclass tt = new Paramsclass ();`
- `//创建一个实例tt, 并调用默认的构造函数`
- `tt.test(1,2,3,-1,-2,-3);` `//调用test方法, 传递6个参数`
- `tt.test(x, y, z);` `//调用test方法, 传递3`
- `个参数`
- `Console.Write("\nx={0}\t",x);` `//查看调用完方法后, 实参的值`
- `是否改变`
- `Console.Write("y={0}\t",y);`
- `Console.Write("z={0}",z);`
- `Console.Read();`
- `}`
- `}`

5.3.3 方法的重载

- 重载是面向对象程序设计的一个重要特征，通过重载可以使多个具有相同功能但参数不同的方法共享同一个方法名。这样将使程序员不再为每个功能近似的多个方法去记不同的方法名而烦恼。
- 一个方法的名字和形式参数的个数、修饰符及类型共同构成这个方法的签名，同一个类中不能有相同签名的方法。如果一个类中有两个或两个以上方法的名字相同，而它们的形参个数或形参类型有所不同是允许的，它们属于不同的方法签名。实际上这些同名方法都在某些方面具有唯一性，编译器能够正确区别它们。不过需要注意的是，方法的返回值类型不是方法签名的组成部分，也就是说，仅返回值的类型不同的同名方法，编译器是不能识别的。

【例5-12】 方法的重载示例。运行结果如图5.9所示.

```
class Myclass
{
    public int add (int x, int y)
    {
        Console.WriteLine("调用了两个整数相加的方法");
        return x+y ;
    }

    public double add (double x, double y)
    {
        Console.WriteLine("调用了两个实数相加的方法");
        return x+y ;
    }

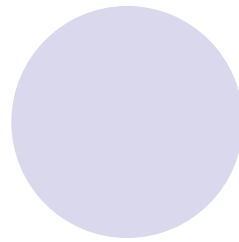
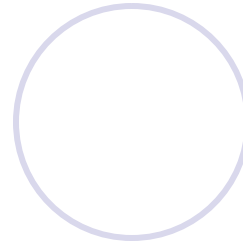
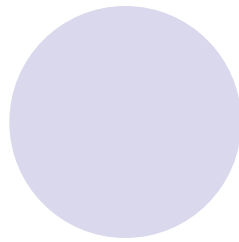
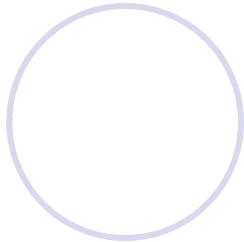
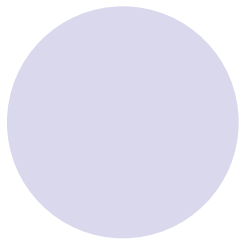
    public int add(int x, int y, int z)
    {
        Console.WriteLine("调用了三个整数相加的方法");
        return x+y+z ;
    }
}
```

- public double add(double x, double y, double z)
- {
- Console.WriteLine("调用了三个实数相加的方法");
- return x + y + z;
- }
-

- static void Main(string[] args)
- {
- int a = 1, b = 2, c = 3;
- double e = 2.3, f = 3.5, g = 6.9;
- Myclass tt = new Myclass(); //创建一个实例tt，并调用默认
的构造函数
- tt.add(a,b,c); //调用三个整数相加的方法
- tt.add(a, b); //调用两个整数相加的方法
- tt.add(e, f); //调用两个实数相加的方法
- tt.add(e, f, g); //调用三个实数相加的方法
- Console.Read();
- }
- }

5.3.4 静态方法和实例方法

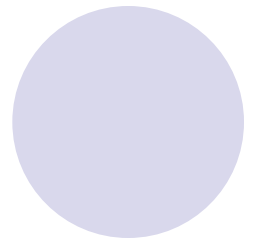
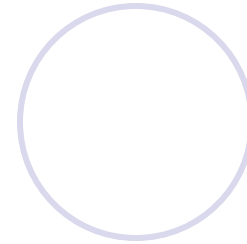
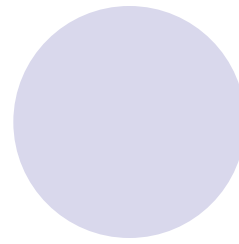
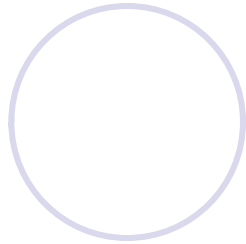
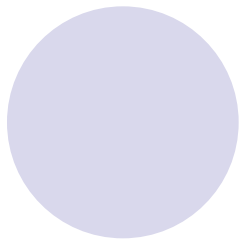
- 将类中的某个成员声明为**static**，该成员称为静态成员。类中的成员要么是静态，要么是非静态的。静态成员是属于类所有的，非静态成员则属于类的实例(对象)所有的。
- 类中的字段分静态字段和实例字段。构造函数分为静态构造函数和实例构造函数。
- 方法分为静态方法和实例方法。



- 通常若一个方法声明中含有**static**修饰符，则表明这个方法是静态方法，同时说明它只对这个类中的静态成员操作，不可直接访问实例字段。
- 若一个方法声明中不包含**static**修饰符，则该方法是一个实例方法。一个实例方法的执行与特定对象相关联，所以需要有一个对象存在。实例方法可以直接访问静态字段和实例字段。

【例5-13】 静态字段、静态方法、实例字段、实例方法实例。
注意下列程序代码哪些是错误的。

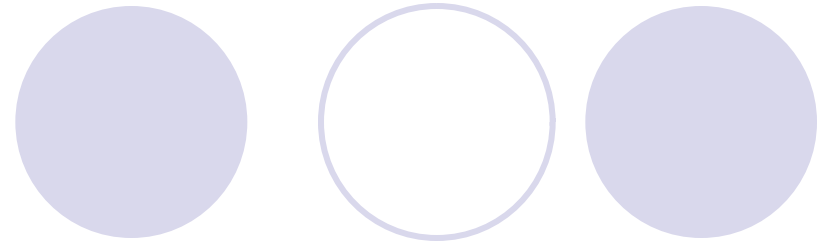
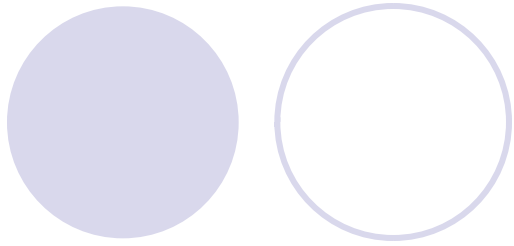
```
class StaticSample
{
    int x;
    static int y;
    void F() //非静态方法，可以访问静态字段、非静态字段
    {
        x = 1; //正确，等价于this.x = 1
        y = 1; //正确，等价于StaticSample.y = 1
    }
    static void G() //静态方法，静态方法只能访问静态成员，不能访问实例字段
    {
        x = 1; //错误，不能访问 this.x
        y = 1; //正确，等价于StaticSample.y = 1
    }
}
```



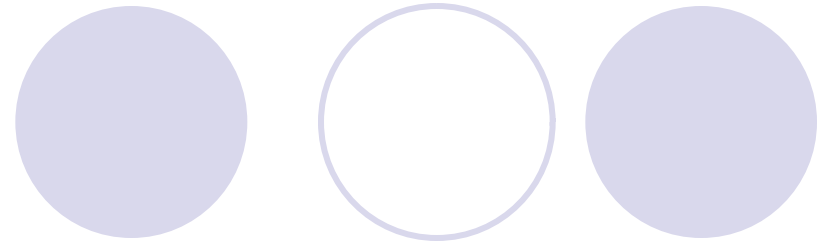
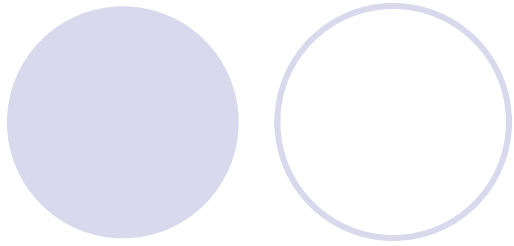
- `static void Main(string[] args)`//主程序例外，可以访问静态字段、非静态字段
- {
- `StaticSample t = new StaticSample();`
- `t.x = 1; //正确`
- `t.y = 1; //错误，不能在类的实例中访问静态成员`
- `StaticSample.x = 1; //错误，不能按类访问非静态成员`
- `StaticSample.y = 1; //正确`
- }
- }

5.4 属性

- 在C#中，数据成员的访问方式一般设为私有(**private**)或保护(**protected**)时，定义相应属性来访问数据成员。
- 属性是一种用于访问对象或类的特性的成员。属性的示例包括字符串长度、字体样式、字体大小、标题等。属性和字段都是具有相关类型的命名成员，访问属性和字段的语法相同。属性又与字段不同，属性不表示存储位置；对编译器来说，属性值的读/写是通过类中封装的特别方法**get**访问器和**set**访问器实现的。属性是字段的自然扩展，属性也可作为特殊的方法使用，并不要求它和字段一一对应，所以属性还可以用于各种控制和计算。



- 属性声明的基本形式：
- [属性修饰符] 类型 属性名
- {
- [get{get访问器}]
- [set{set访问器}]
- };
- 访问属性：对象.属性名图5.10 例5-14 运行结果
- 本质上，属性是方法。**C#**中的属性是通过**get**和**set**访问器来对属性的值进行读/写的。**get**访问器的返回值类型与属性的类型相同，所以语句块中的**return**语句必须有一个可隐式转换为属性类型的表达式。
- **set**访问器没有返回值，但它有一个隐式的值参数，其名称为**value**，**value**的类型与属性的类型相同。
- 同时包含**get**访问器和**set**访问器的属性具有读/写功能，只包含**get**访问器的属性具有只读功能，只包含**set**访问器的属性具有只写功能。



- 【例5-14】 在类中对属性通过get、set访问器设定读/写功能。运行结果如图5.10所示，代码如下：

```
● class TextBox                                //定义一个类
● {
●     private string text;                    //私有字段，标题
●     private string fontname;                //私有字段，字体
●     private int fontsize;                   //私有字段，字体大小
●     public TextBox()                        //无参数的构造函数，为一些字
    段赋初值
●     {
●         text = "text1";
●         fontname = "宋体";
●         fontsize = 12;
●     }
```

public string Text //属性，可读可写，对text私有字段进行读/写，注意大小写

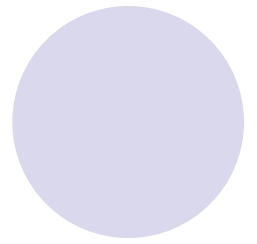
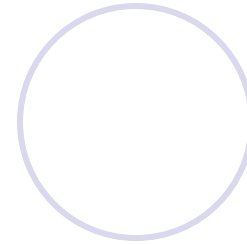
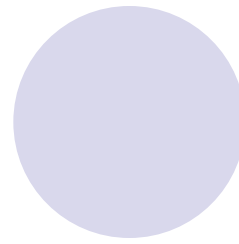
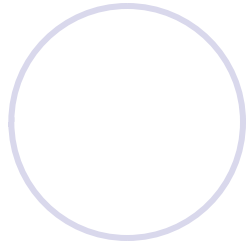
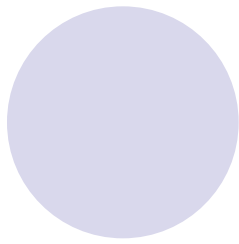
```
{  
    get //读  
    { return text; }  
    set //写  
    { text = value; } //隐式参数value  
}
```

public string FontName
//FontName属性，只读属性，对fontname字段进行读，注意大小写

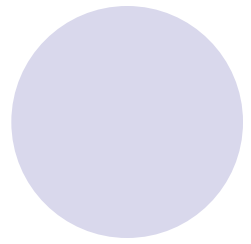
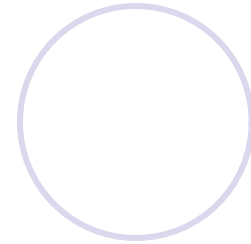
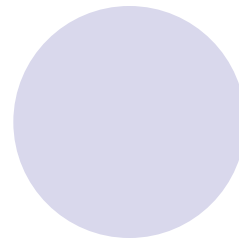
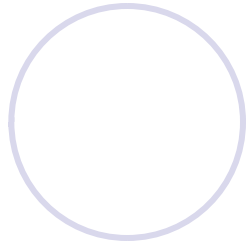
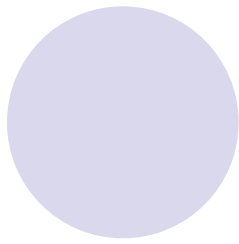
```
{  
    get  
    { return fontname; }  
}
```

public int FontSize
//FontSize属性，可读可写，对fontsize字段进行读/写，注意大小写

```
{  
    get  
    { return fontsize; }  
    set  
    { fontsize = value; }  
}
```



```
• class Point                                //定义一个类
• {
•     int x, y;          //私有字段，坐标
•     public int X        //属性，只读，返回私有字段x的值，属性名为X，注意大小写
•     {
•         get
•         { return x; }
•     }
•     public int Y        //属性，只读，返回私有字段y的值，属性名为Y，注意大小写
•     {
•         get
•         { return y; }
•     }
•     public Point() //无参数的构造函数，为一些字段赋初值
•     { x = y = 0; }
•     public Point(int x, int y) //有参数的构造函数，构造函数的重载，为一些字段赋初值
•     {
•         this.x = x;
•         this.y = y;
•     }
• }
```



- class Label
- {
- Point p1 = new Point();
- //Label类中的字段p1是Point类型，调用Point无参构造函数
- Point p2 = new Point(5, 10);
- //Label类中的字段p2是Point类型，调用Point有参构造函数
- public int Width //只读属性，计算两点之间的宽度
- {
- get
- { return p2.X - p1.X; }
- }
- public int Height //只读属性，计算两点之间的高度
- { get
- { return p2.Y-p1.Y; }
- }
- }

```
class Test
```

```
{
```

```
    static void Main(string[] args)
```

```
    {
```

```
        TextBox Text1 = new TextBox(); //创建一个对象，调用无参的构造函数
```

```
        Console.WriteLine("Text1.Text= {0} ", Text1.Text);
```

```
        Text1.Text = "这是文本框";
```

```
        //对Text1对象的Text属性进行赋值，value="这是文本框"
```

```
        Console.WriteLine("Text1.Text= {0} ", Text1.Text);
```

```
        Console.WriteLine ("Text1.Fontname= {0} ", Text1.FontName);
```

```
        Text1.FontSize=36;
```

```
        Console.WriteLine ("Text1.FontSize= {0} ", Text1.FontSize);
```

```
        Label Label1 = new Label(); //创建一个对象，调用默认的构造函数
```

```
        Console.WriteLine("Label1.Width= {0} ", Label1.Width);
```

```
        Console.WriteLine("Label1.Height= {0} ", Label1.Height);
```

```
        Console.Read ();
```

```
    }
```

```
}
```

//定义一个类



```
C:\ file:///E:/zm/c#e...
Text1.Text= text1
Text1.Text= 这是文本框
Text1.Fontname= 宋体
Text1.FontSize= 36
Label1.Width= 5
Label1.Height= 10
```

图5.10 例5-14 运行结果