

# 第3章 C#面向对象设计

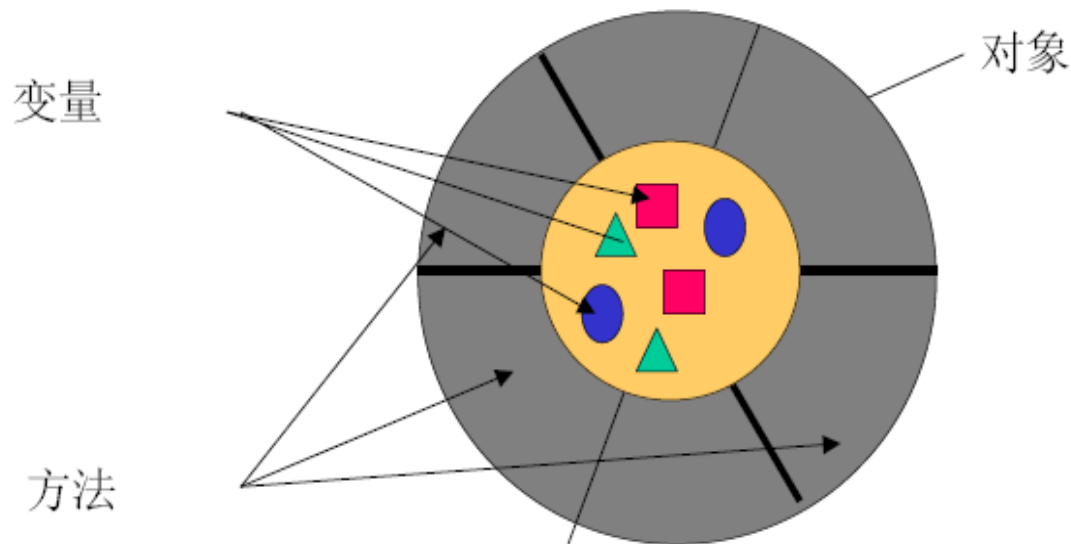
- 理解C#中的类和对象的概念；
- 掌握C#中的构造函数和析构函数的定义方法；
- 掌握函数定义及传参传址；
- 掌握C#中的访问修饰符的使用方法；
- 理解静态成员的特点和用法；
- 理解类的继承性和多态性；
- 了解抽象类和接口，结构体高级应用。

# 3.1 C#的面向对象特性

- 所有东西都是对象：变量和方法的集合。
- 初级特性：OO最基本的概念，即类和对象。
- 中级特性：OO最核心的概念，即封装、继承和多态。
- 高级特性：由初级特性和中级特性引出的一些问题，如构造函数的使用、覆盖的规则、静态变量和函数等。

# 初级特性

- 面向对象技术最基本的概念是类和对象：
  - 类是一个样板，以操作、表示和算法的形式完整地定义了一组对象的行为。它通常也是面向对象语言中的模块化、封装和数据抽象的基础。
  - 对象是类的一个具体实例，是一个软件单元，它由一组结构化的数据和在其上的一组操作构成。

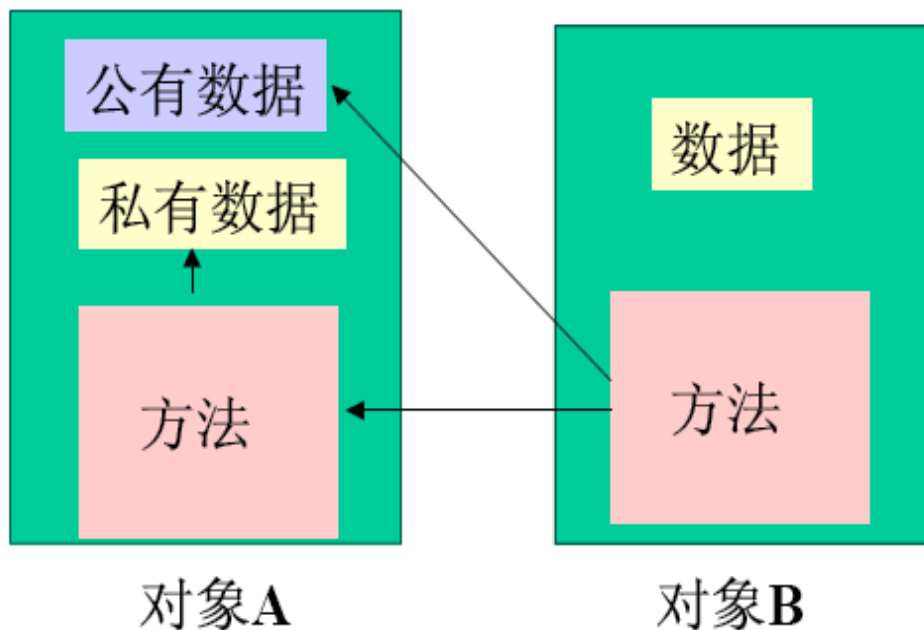


# 中级特性

- 面向对象技术的三个核心概念：
  - **封装**：将数据和操作组合到一起，并决定哪些数据和操作对外是可见的。
  - **继承**：父类中的变量和行为，子类可以同样使用。本质是代码重用。
  - **多态**：由继承引出的一种机制，父类型的引用变量可以指向子类型的对象。

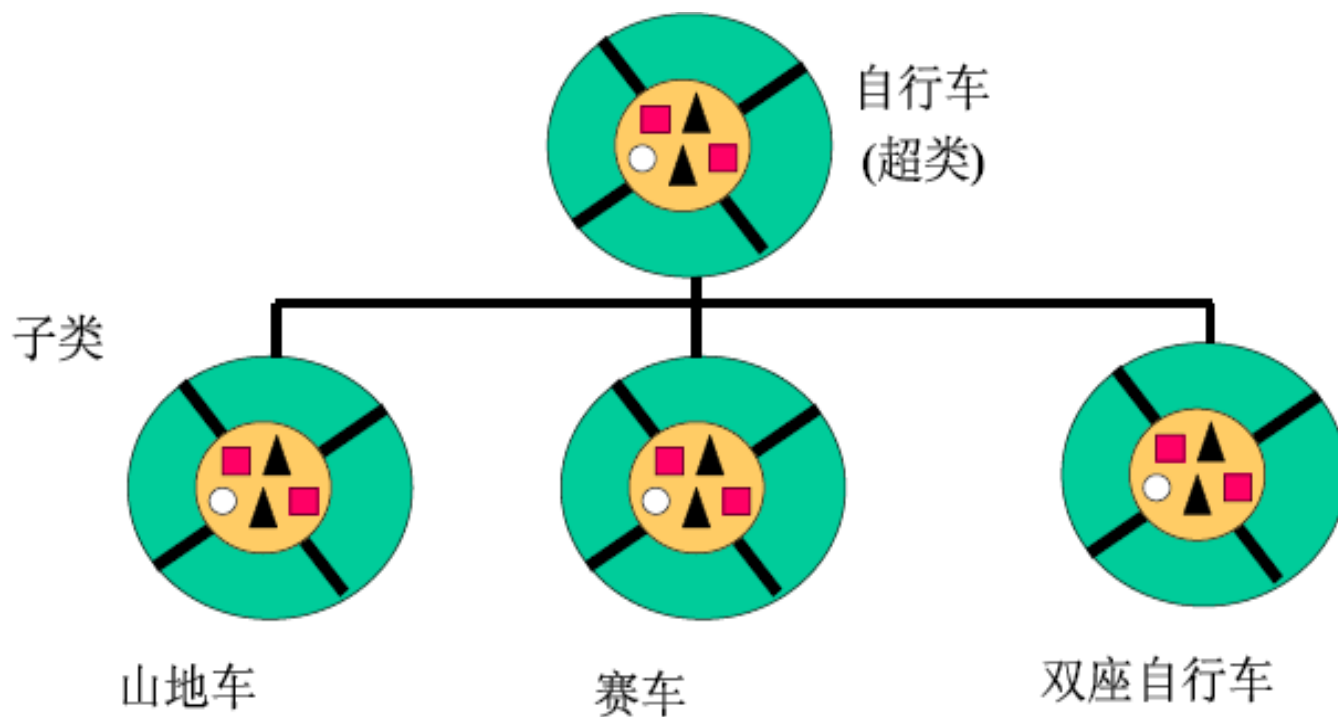
# 封装

- 封装把对象的所有组成部分组合在一起，有三个作用
  - 隐藏类的实现细节：使用方法将类的数据隐藏起来。
  - 要求用户使用一个接口去访问数据：定义程序如何引用对象的数据，控制用户对类的修改和访问数据的程度。
  - 使代码更好维护：类的内部实现改变，对外接口可以不变。



# 继承

- 继承提供了创建新类的一种方法，继承对开发者来说就是**代码共享**。
  - 通过继承创建的子类是作为另一个类的扩充或修正所定义的一个类。
  - 子类从父类中继承所有方法和变量。
  - 子类和父类之间是特化与范化的关系。



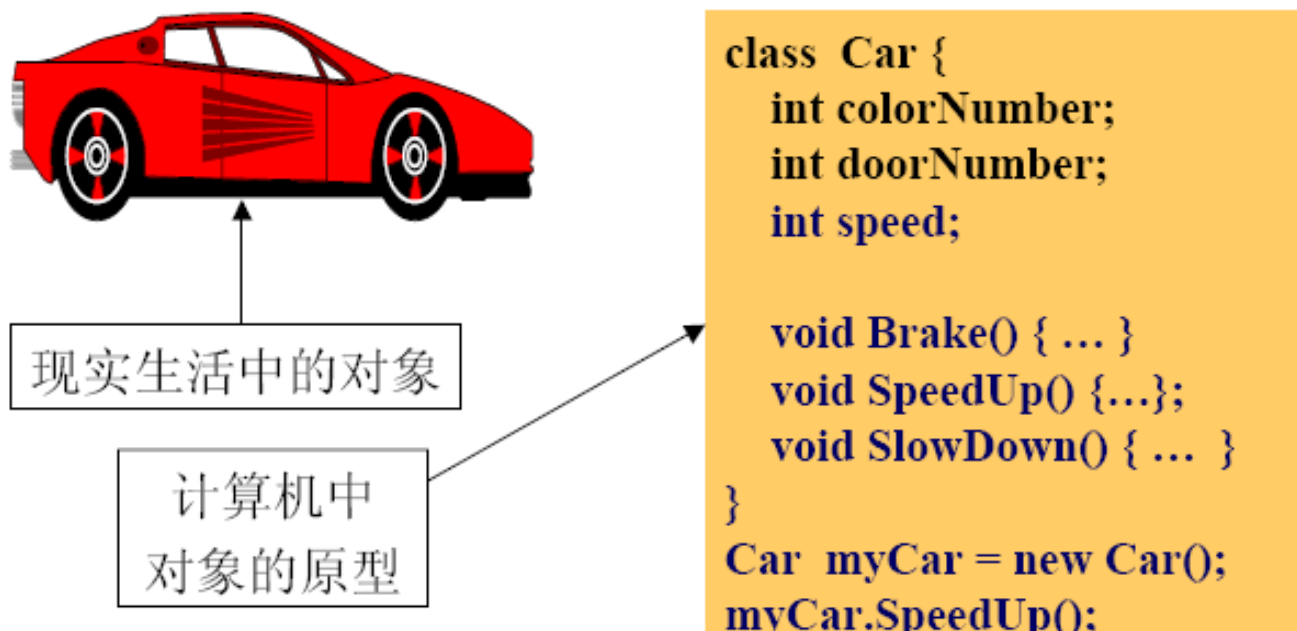
## 3.2 类与对象定义

- 类：C#所有的代码都是在某一个类中，因此不可能在类之外的全局区域有变量和方法。
- 对象：C#中的对象相当于一块内存区域，保存对象特有的类中所定义的数据。
- 引用：C#中对于对象的操作全部通过引用进行。



# 抽象数据类型

- 类实际上为实际的物体在计算机中定义了一种**抽象数据类型**。
- 抽象数据类型是仅由数据类型和可能在这个数据类型上进行的操作定义的。
- 使用者只能通过操作方法来访问其属性，不用知道这个数据类型内部各种操作是如何实现的。



- 3.2.1 类的定义

类使用class关键字声明。采用的形式为：

[类修饰符] class 类名称[:基类以及实现的接口列表]

{

    类体

}

# 类、对象和引用的声明

- 声明一个类：访问修饰字 `class` 类名{变量声明，方法声明}

```
class Student
```

```
{  
    long id; // 学号  
    char gender; //性别  
    int classID; // 班级号，注意不能用class作属性名  
    void ChangeClass(int aClassID) //更改班级{... }  
}
```

如何使用类：

- 声明一个对象引用：类名引用名 `Student student;`
- 创建一个对象：`new` 类构造函数

**`student = new Student();` //如果缺少这一步编译器会报错**

- 使用对象：引用名. 变量名/方法名(参数)

```
student.id = 94;
```

# 引用与对象举例

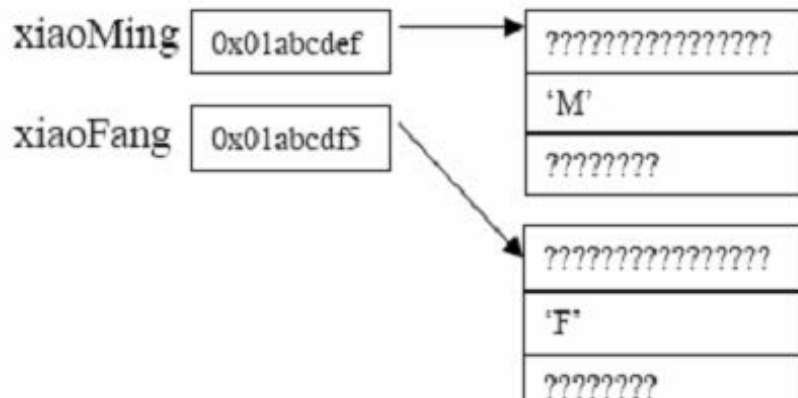
## 两个引用指向不同的对象

```
Student xiaoMing = new Student();  
Student xiaoFang = new Student();  
xiaoMing.gender = 'M';  
xiaoFang.gender = 'F';
```

说明:

结果xiaoMing.gender为'M', 因为引用xiaoMing和xiaoFang指向不同的对象。

( xiaoMing == xiaoFang)的结果为false。此时对象相等的判断应该用equals方法。



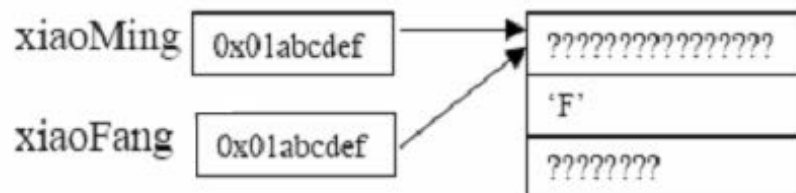
## 两个引用指向同一个对象

```
Student xiaoMing = new Student();  
Student xiaoFang = xiaoMing;  
xiaoMing.gender = 'M';  
xiaoFang.gender = 'F';
```

说明:

结果xiaoMing.gender为'F', 因为引用xiaoMing和xiaoFang指向同一个对象。

( xiaoMing == xiaoFang)的结果为true。xiaoMing.equals(xiaoFang)结果也为true。



## ■ 引用类似于C++中的对象指针。但又有区别：

- 1.在C#中” 引用“是**指向一个对象在内存中的位置**，在本质上是一种带有很强的完整性和安全性的限制的指针。
- 2.当声明某个类、接口或数组类型的一个变量时，变量的值总是某个对象的引用或者是**null**引用。
- 3.指针就是**简单的地址**而已，而引用除了表示地址而外，还是被引用的数据对象的缩影，可以提供其他信息。
- 4.指针可以有++、--运算，引用不可以运算。

- 3.2.2 类的成员

- 1. 类的成员分类

- 常量：表示与该类相关联的常量值。
- 字段：即该类的变量。
- 类型：用于表示一些类型，它们是该类的局部类型。
- 方法：用于实现可由该类执行的计算和操作。
- 属性：用于定义一些命名特性，通过它来读取和写入相关的特性。
- 事件：用于定义可由该类生成的通知。
- 索引器：使该类的实例可按与数组相同的（语法）方式进行索引。
- 运算符：用于定义表达式运算符，通过它对该类的实例进行运算。
- 实例构造函数：用于规定在初始化该类的实例时需要做些什么。
- 析构函数：用于规定在永久地放弃该类的一个实例之前需要做些什么。
- 静态构造函数：用于规定在初始化该类自身时需要做些什么。

类声明 ——— public class Furniture  
{

常量 ——— const double salesTax = .065;

字段 [——— private double purchPrice;  
——— private string vendor, inventoryID;

构造函数 [——— public Furniture(string vendor, string inventID, double  
——— purchPrice)  
——— {  
——— this.vendor = vendor;  
——— this.inventoryID = inventID;  
——— this.purchPrice = purchPrice;  
——— }

成员属性 [——— public string MyVendor  
——— { get { return vendor; } }

方法 [——— public double CalcSalesTax(double salePrice)  
——— { return salePrice \* salesTax; }  
——— }

- 2. 类成员的可访问性

表 2-2 类成员的可访问性

声明的可访问性	意义
public	访问不受限制。
protected	访问仅限于包含类或从包含类派生的类型。
internal	访问仅限于当前程序集。
protected internal	访问仅限于从包含类派生的当前程序集或类型。
private	访问仅限于包含类型。



- 类的默认访问类型是**internal**（当前程序集）。
- 类中所有的成员，默认均为**private**。

```
internal—— class Furniture
{
    const double salesTax = .065;
private—— private double purchPrice;
    private string vendor, inventoryID;

    public Furniture(string vendor, string inventID,
double purchPrice)
    {
        this.vendor = vendor;
        this.inventoryID = inventID;
        this.purchPrice = purchPrice;
    }
    public string MyVendor
    { get { return vendor; } }

    public double CalcSalesTax(double salePrice)
    { return salePrice * salesTax; }
}
```

```
class MyApp
{
    static void Main()
    {
        Furniture f = new Furniture("aaa", "001", 1.2);

        Console.WriteLine(f.salesTax);
        Console.WriteLine(Furniture.salesTax);

        f.purchasePrice = 10; //???

        string str = f.MyVendor;
    }
}
```

提问：是否可以访问？

## 3.2.3 构造函数

- 构造函数是一种用于对象初始化的特殊方法，有以下特点。
  - 构造函数只能在对象创建时调用，即和new运算符一起被调用。
  - 构造函数和类具有相同的名字。
  - 构造函数可以有0个、1个或多个参数。
  - 构造函数没有返回值。
  - 每个类至少有一个构造函数，一个类可以有多个构造函数。
  - 如果没有为类定义构造函数，系统会自动为其定义一个缺省的构造函数。缺省构造函数不带参数，作用是将实例变量都清零。
  - 一旦为类定义了构造函数，则系统不会再为其定义缺省构造函数。
- C#中构造函数有三种：实例构造，私有构造和静态构造

- 创建对象与构造函数

类声明后，可以创建类的实例，即对象。创建类的实例需要使用**new**关键字。类的实例相当于一个变量，创建类实例的格式如下：

类名 对象名=**new** 构造函数（参数类表）；

例如：

```
Point myPoint = new Point();
```

- 类的构造函数可通过初始值设定项来调用基类的构造函数，例如：

```
public Student(string no, string name, char sex, int  
    age) : base(name, sex, age)  
{ ... }
```

- 类的构造函数也可通过关键字this调用同一个类的另一个构造函数，例如：

```
public Point() : this(0, 20)  
{ ... }
```

# 构造函数举例

```
class Student
{
    long id;
    char gender;
    int classID;

    public Student(long aID, char aGender, int aClassID)
    {
        id = aID;
        gender = aGender;
        classID = aClassID;
    }
}
```

# 私有构造函数

- 在某些特殊的情况下，使用私有构造函数能够达到特殊的效果。比如，想建立这样一个类：**不允许被其他类实例化，但提供对外的静态接口成员。**在.NET框架类库中就存在这样的类，如System.Math类就不能够实例化，它的所有成员都是静态的。
- 用关键字private修饰的构造函数就是私有构造函数。

- 下面的代码建立一个无法实例化的类filling:

```
public class filling
{
    private filling() { } //私有构造
    public static void happy()
    {
        Console.WriteLine ("How happy!");
    }
    public static void sad()
    {
        Console.WriteLine ("So sad!!!!");
    }
}
public class MainClass
{
    public static void Main()
    {
        //filling f1=new filling(); // 不需要实例化，直接使用类名访问
        filling.happy();
        filling.sad();
    }
}
```



# 析构函数

- C#支持析构函数。虽然C#能够自动进行垃圾回收，但对于某些资源，.Net不知道如何回收，所以需要人工的内存回收。  
在.net 编程环境中，系统的资源分为托管资源和非托管资源。
  - 托管资源，如简单的**int,string,float,DateTime** 等等，是不需要人工干预回收的。
  - 非托管资源，例如**文件，窗口或网络连接**，对于这类资源虽然垃圾回收器可以跟踪封装非托管资源的对象的生存期，但它不了解具体如何清理这些资源。在使用完之后，必须显式的释放他们，否则会占用系统的内存和资源，而且可能会出现意想不到的错误。
- .net 中超过80%的资源都是托管资源。

# 非托管资源：Dispose方法

	析构函数	Dispose方法
意义	销毁对象	销毁对象
调用方式	不能被显示调用，在GC回收是被调用	需要显示调用 或者通过using语句
调用时机	不确定	确定，在显示调用或者离开using程序块

# 对象析构举例

- **Dispose** 方法是编程人员需要立即释放资源时调用，所以在 **Dispose** 方法中调用**Close**，并通知GC在回收垃圾时不需要再释放资源。
- 析构函数是编程人员在没有调用**Dispose**方法释放资源的情况下由GC在回收垃圾时调用。
  1. 一般不要提供析构函数，因为它不能及时地被执行；
  2. 实现**Dispose**方法的时候，一定要加上“**GC.SuppressFinalize( this )**”语句。（建议：网络再检索）

```
using System;
class MyFile
{
    public MyFile() { //...Open File}
    public void Close() { //...Close File}
    public void Dispose()
    {
        Close();
        GC.SuppressFinalize(this);
    }
    ~MyFile(){ Close(); }
}
class MyApp
{
    public static void Main()
    {
        MyFile file = new File();
        ...
        file.Dispose();
    }
}
```

## 3.3类中的静态问题

- 问题：
  - 用Student对象保存学生信息，希望每个对象有一个单独的编号。第一个创建的对象编号为0，第二个对象编号为1，以此类推。
  - 这就需要有一个所有Student对象都能访问的变量counter，而且变量counter在所有实例中共享。当一个对象创建时，构造函数增加counter值，下一个对象创建时使用增加过的值。
- 解决方法：
  - C#编程语言没有这样的全局变量，但类变量是可以从类的任何实例访问的单个变量。
  - 类变量在某种程度上与其它语言中的全局变量相似。但仅限于同类型的对象访问。

# 静态变量

- 类变量是在所有类的实例(对象)中共享的变量，在变量声明中用**static**关键字表示。
- 类变量可以被标记为**public**或**private**。如果被标记为**public**，不需要类的实例就可以访问。

```
public class Student
```

```
{  
    private int serialNumber;  
    private static int counter = 0;  
    public Student() {  
        serialNumber = counter;  
        counter++;  
    }  
}
```

```
class MyApp
```

```
{  
    static void Main()  
    {  
        Student stu = new Student();  
        Console.WriteLine(stu.serialNumber);  
        Student stu1 = new Student();  
        Console.WriteLine(stu1.serialNumber);  
    }  
}
```

# 静态方法

- 类静态方法是**不需要类的任何实例**就可以被调用的方法，在方法声明中用 **static** 关键字表示。
- 类方法只能访问静态变量，访问非静态变量的尝试会引起编译错误。
- 静态方法不能被覆盖成非静态的。
- **main** 是静态的，因为它必须在任何实例化发生前被访问，以便应用程序的运行。
- ```
public class GeneralFunction {  
    public static int AddUp(int x, int y)  
    { //静态方法  
        return x + y;  
    }  
}
```
- ```
public class UseGeneral {  
    public void method() {  
        int c = GeneralFunction.AddUp(9, 10); //调用静态方法  
        System.Console.WriteLine("addUp() gives " + c);  
    }  
}
```

# 静态构造函数

- 问题：在Student类的例子中，如果希望对象编号不是从1开始，而是从0到1000之间的随机的数开始。
- 构造函数中虽然可以执行代码，但每个对象创建时都执行，而本例中只希望第一个对象创建时执行。

解决：

**C#**中支持静态构造函数，静态构造函数在类中第一个对象初始化或引用任何静态成员之前执行。



using System;

```
public class Student
{
    public int serialNumber;
    private static int counter;
    static Student()
    {
        Random rand = new Random(0);
        counter = rand.Next(0, 1000);
    }
    public Student()
    {
        serialNumber = counter;
        counter++;
    }
}
```

```
class MyApp
{
    static void Main()
    {
        Student stu = new Student();
        Console.WriteLine(stu.serialNumber);
        Student stu1 = new Student();
        Console.WriteLine(stu1.serialNumber);
    }
}
```

- 静态构造函数用于初始化类。在创建第一个实例或引用任何静态成员之前，将自动调用静态构造函数来初始化类。

```
using System;
class BaseClass
{
    private static int count;
    static BaseClass()
    {
        count = 0;
        Console.WriteLine("aaaa");
    }

    public BaseClass()
    {
        Console.WriteLine("bbbb");
    }
}
class MyApp
{
    static void Main()
    {
        BaseClass a = new BaseClass();
        BaseClass a1 = new BaseClass();
    }
}
```

- 静态构造函数既没有访问修饰符，也没有参数。
- 无法直接调用静态构造函数。
- 在程序中，用户无法控制何时执行静态构造函数。
- 静态构造函数的典型用途是：当类使用日志文件时，将使用这种构造函数向日志文件中写入项。

## 案例：创建类（构造函数使用）

```
Class0918 ss = new Class0918(34);
```

```
//ss.a = 10000;
```

```
Class0918.a = 1000;
```

```
Class0918.fun1(1, 3);
```

```
class Class0918
{
    public static int a = 0;

    public Class0918(int i)
    {
        a = i;
    }

    public static void fun1(int s,int d)
    {
        Console.WriteLine("***** " + a);
        // Console.WriteLine("***** "+ (s+d));
    }
}
```

# 实验3类与对象

## 实验要求：

- 1、创建类（构造函数使用）；
- 2、public、private等使用；
- 3、新建变量和函数对象，并调用；
- 4、定义一个“学生类”，定义GetManNum()函数，计算男生数量。

## 3.4 函数

- C#编程语言支持的函数参数传递方式包括：
  1. **值传递**：方法中的变量是传入变量的一个拷贝，方法中对形参做的修改，不会影响方法外面的实参。
    - (1) 对于值类型数据，值传递就是传递了变量的值。
    - (2) 对于引用类型数据，值传递传递的是引用的值，即方法中的形参和方法外的实参将指向同一对象。因此，通过形参也能修改对象的实际内容。
  2. **地址传递**：方法中的变量是传入变量的一个引用，方法中对形参做的修改，也会影响方法外面的实参。
    - (1) **ref**：由调用方法初始化参数值。
    - (2) **out**：被调用方法初始化参数值，可以不用初始化就作为参数传递给方法。

# 值传递-值类型

```
class HelloWorld
{
    static void change100(int x)
    {
        System.Console.WriteLine("Inside method, x=" + x);
        x = 100;
        System.Console.WriteLine("Inside method, x=" + x);
    }
    static void Main(string[] args)
    {
        int y = 10;
        System.Console.WriteLine("Outside method, y=" + y);
        change100(y);
        System.Console.WriteLine("Outside method, y=" + y);
    }
}
```

形参

实参

Outside method, y=10  
Inside method, x=10  
Inside method, x=100  
Outside method, y=10

- 对方法中形参的操作不会改变方法外实参的值。

# 值传递-引用类型

```
class Student
{   public int No;}
class HelloWorld
{   static void change100(Student x)
    {       x.No = 100;
    }
    static void Main(string[] args)
    {
        Student s = new Student();
        s.No = 10;
        change100(s);
        System.Console.WriteLine("s.No = " + s.No);
    }
}
```

s.No = 100

对方法中引用的操作会改变方法外实际对象的内容。

## 引用参数-ref

- 很多情况下，我们要使用参数的引用传递。引用传递是传递变量的地址，使得形参和实参指向同一内存空间，方法中对于形参的修改，实际上就是对于实参的修改。
- 由调用方法初始化参数值。实参、形参中ref不能省。

```
class HelloWorld
```

```
{    static void change100(ref int x)
    {        System.Console.WriteLine("Inside method, x=" + x);
              x = 100;
              System.Console.WriteLine("Inside method, x=" + x);
    }
    static void Main(string[] args)
    {        int y = 10;
              System.Console.WriteLine("Outside method, y=" + y);
              change100(ref y);
              System.Console.WriteLine("Outside method, y=" + y);
    }
}
```

```
Outside method, y=10
Inside method, x=10
Inside method, x=100
Outside method, y=100
```

- 对方法中形参的操作会改变方法外实参的值。



```
public void CalcuArea(long L, ref long W)
{
    double Area;
    L+=10;
    W+=10;
    Area = (L * W);
    Console.WriteLine(Area);
}
```

调用之后CalcuArea()函数之后:

len仍为5

wid变为15

Area=225

假设过程调用如下:

len= 5;

wid= 5;

CalcuArea(len, ref wid);

## 输出参数-out

- 被调用方法初始化参数值，可以不用初始化就作为参数传递给方法。
- 实参、形参中out不能省。**

```
class HelloWorld
{
    static void change100(int x, out int y)
    {
        y = x;
    }
    static void Main(string[] args)
    {
        int x = 10, y;
        change100(x, out y);
        System.Console.WriteLine("y=" + y);
    }
}
```

y=10

- 使用out关键字，可以将未初始化的变量传递给方法，可以避免多余的初始化。

### 例3-6：输出参数的使用。

```
public class Test
```

```
{
```

```
    static void SplitPath(string path, out string dir, out string name)
```

```
    {
```

```
        int i = path.Length;
```

```
        while (i > 0)
```

```
        {
```

```
            char ch = path[i - 1];
```

```
            if (ch == '\\') break;
```

```
            i--;
```

```
        }
```

```
        dir = path.Substring(0, i);
```

```
        name = path.Substring(i);
```

```
    }
```

```
    static void Main()
```

```
    {
```

```
        string dir, name; //变量作为输出参数无须明确赋值
```

```
        SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
```

```
        Console.WriteLine(dir);
```

```
        Console.WriteLine(name);
```

```
    }
```

```
}
```

**c:\\Windows\\System\\  
hello.txt**

# 参数数组

- 在不能确定需要传递多少个参数的时候可以使用`params`关键字指明一个可变的参数数组。
- 数组参数的类型必须是一维数组，而且必须是形参表中的最后一个参数。
- 数组参数始终是通过值传递方式进行传递，不能将`params`同`ref`和`out`组合。

例如，下面定义一个具有params参数的求平均值函数：

```
public double AVG( params int[ ] Nums)
{
    int Sum=0; int Count=0;
    foreach(int n in Nums)
    {
        Sum+=n;
        Count+=1;
    }
    Nums[0] = 100;
    return (Sum/Count);
}
```

调用该函数时，可以采用下面的语句：

```
Class1 a = new Class1();
double d = a.AVG(13,27,33,25,78);
```

## 3.5 属性成员

- 属性主要用于描述和维护类对象的状态。从客户端看，对属性的访问就好像直接访问public字段成员，但在类内部是通过类方法访问的。
- 创建一个属性包括两步：
  1. 声明一个字段来存储属性值
  2. 编写一个属性声明，提供访问接口

属性的建立要使用属性声明，语法如下：

[访问修饰符] 类型名 属性名

```
{ get  
  { return 字段; }  
  set  
  { 私有字段 = value; }  
}
```

```
class Student{  
    public int No;  
}
```

```
class Student {  
    private int No;  
    public int GetNo(){  
        return No;  
    }  
    public void SetNo(int aNo){  
        if(aNo > 0)No = aNo;  
        else No = 0;  
    }  
}
```

```
class Student3 {  
    private int no;  
    public int No{  
        get{ return no; }  
        set{ if(value > 0) no = value;  
            else no = 0;  
            }  
    }  
}  
class HelloWorld  
{    static void Main(string[] args)  
    {        Student3 s = new Student3();  
            s.No = 1;  
    }  
}
```

```
public class checkval
{
    private string p_PropVal; //声明一个私有变量p_PropVal
    public string str1 //声明属性str1
    {
        get //返回存储在私有变量中的属性值
        {
            return p_PropVal;
        }
        set //存储属性值到私有变量
        {
            if (Convert.ToString(value).Length <= 10)
            { p_PropVal = value; }
            else
            { Console.WriteLine("too many words"); }
        }
    }
}
```



可以说，属性是一种特殊的方法，但属性和方法也有不同之处，主要有：

- 属性不必使用圆括号，但方法一定使用圆括号。
- 属性不能指定参数，方法可以指定参数。
- 属性不能使用void类型，方法则可以使用void类型。
- 属性使用方法与变量相同。

# 属性说明

- 可以创建只读或只写属性，即只有`get`或`set`方法。
- 可以创建静态属性，用`static`关键字。
- 静态属性不与特定实例有关联，因此在静态属性的`get`和`set`方法内引用`this`是错误的。
- 静态属性使用类名访问，并且，与静态属性相配合的私有字段也应该是静态的。

### 例3-7:

```
public class Checkval
{
    private static string p_time="00:00:00";//静态私有字段
    public static string mytime //静态属性
    {
        get
        { return p_time; }
    }
    public static string mytime1 //静态属性
    {
        set
        { p_time = value; }
    }

    static void Main()
    {
        Console.WriteLine(Checkval.mytime);
        Checkval.mytime1 = "10:2:22";
        Console.WriteLine("now is " + Checkval.mytime);
    }
}
```

输出结果为:

00:00:00

now is 10:2:22

- **this**关键字

- this引用的是当前实例。
- this关键字是一个隐含引用，它隐含于每个类的成员函数中。
- this关键字引用类的当前对象，成员通过this关键字可以知道自己属于哪一个实例。

```
public class Test1
{
    public string str;
    public void f(string str)
    {
        this.str = str;
    }

    public static void Main()
    {
        Test1 test = new Test1();
        test.f("aaa");
    }
}
```

- 静态函数没有this关键字。

## 3.6 继 承

- 语法: 子类声明:父类{子类体}
- 子类可以使用父类的protected和public可见的变量和方法, 就像这些变量和方法是自己定义的一样。
- C# 中, 如果类声明时没有声明父类, 那么缺省为Object 类的子类。C#中的所有类都是System.Object类的子类。
- C#中, 子类只能继承一个父类。(java, C++支持; c#使用接口实现该功能)

```
class Car
{
    int color;
    int door;
    int speed;
    void PushBreak() { }
    public void AddOil() { }
}
```

```
class TrashCar : Car
{ }

class MyApp
{
    static void Main()
    {
        TrashCar myCar = new TrashCar();
        myCar.AddOil();
        myCar.PushBreak();
    }
}
```

- 派生类的建立需要注意：

- (1).派生类会继承基类除了构造函数和析构函数的所有成员。
- (2).派生类调用构造函数时，会先调用基类的构造函数。默认只能调用没有参数的构造函数。
- (3).用**base**关键字显式调用基类构造函数。

```
class Car
{
    public Car(int i) { }
}
class TrashCar : Car
{
    public TrashCar(int i) { }
```

错误

```
class Car
{
    public Car(int i) { }
}
class TrashCar : Car
{
    public TrashCar(int i):base(i) { }
```

错误

(4). 如果需要调用基类中的同名方法，应该使用” **base.方法名**” 来调用。

```
class Car
{
    public Car()
    {}
    protected void f() { Console.WriteLine("aaa"); }
}
class TrashCar : Car
{
    public TrashCar() { }
    void f()
    { Console.WriteLine("bbb"); }
    public void f1()
    {
        base.f(); //需调用基类同名的函数时
        f();
    }
}
class MyApp
{
    static void Main()
    {
        TrashCar myCar = new TrashCar();
        myCar.f1();
    }
}
```

结果：

aaa

bbb

### 例3-8 继承格式举例

public class parent //建立基类

```
{  
    public parent(string str) //基类带参数构造函数  
    {Console.WriteLine(str);}  
    public void showposition() //基类方法  
    {Console.WriteLine("基类的位置在 (0, 0) ");}  
}
```

public class child:parent //派生子类

```
{  
    public child():base("调用基类构造") //子类构造函数，显示调用基类构造函数  
    {Console.WriteLine("I am child");}  
    public void showposition()  
    {  
        base.showposition(); //子类调用基类方法  
        Console.WriteLine("派生类的位置在 (10,10) ");  
    }  
}
```

- 在Main()方法中执行下面的代码:

```
parent prt=new parent("I am a parent");  
child chd= new child();
```

```
prt.showposition();  
chd.showposition ();
```



- 输出结果为:

I am a parent

调用基类构造

I am child

基类的位置在 (0, 0)

基类的位置在 (0, 0)

派生类的位置在 (10,10)

- 例3-9: 编写一个程序, 计算球, 圆锥, 圆柱的表面积和体积

```
using System;
namespace ConApp1
{
    public class Circle
    {
        protected double radius;
        public Circle(double r) {radius=r;}
        public double GetArea(){return Math.PI*radius*radius;}
    }
    public class Sphere:Circle//球体类
    {
        public Sphere(double r):base(r){}
        public double GetArea(){return (4*base.GetArea());} //表面积
        public double GetVolumn()
        {return (4*Math.PI*Math.Pow(radius,3)/3);} //体积
    }
    public class Cylinder:Circle//圆柱类
    {
        private double height;//添加高度字段
        public Cylinder(double r,double h):base(r){height=h;}
        public double GetArea()
        {return (2*base.GetArea()+2*Math.PI*radius*height);}
        public double GetVolumn()
        {return(Math.PI*radius*radius*height);}
    }
}
```

```
public class Cone:Circle//圆锥类
{
    private double height;//添加高度字段
    public Cone(double r,double h):base(r){height=h;}
    public double GetArea()
    {return(Math.PI*radius*(radius+Math.Sqrt(height*height+radius*radius)));}
    public double GetVolumn()
    {return (Math.PI*radius*radius*height/3);}
}
public class Tester
{
    public static void Main()
    {
        Circle c1=new Circle(2);
        Sphere s1=new Sphere(2);
        Cylinder cd1=new Cylinder(2,10);
        Cone cn1=new Cone(2,10);
        Console.WriteLine("s1's serfacearea={0},
            volumn={1}",s1.GetArea(),s1.GetVolumn());
        Console.WriteLine("cd1's serfacearea={0},
            volumn={1}",cd1.GetArea(),cd1.GetVolumn());
        Console.WriteLine("cn1's serfacearea={0},
            volumn={1}",cn1.GetArea(),cn1.GetVolumn());
        Console.ReadLine();
    }
}
```

## 3.7 多 态

- 继承机制引出多态机制
- 某一类型的引用变量可以指向该类或者其子类的对象。
- 由于C#中System.Object类是所有类的祖先，所以可以用Object类型的引用指向所有类型的对象。

```
class Car  
{  
    ...  
}  
class TrashCar : Car  
{  
    ...  
}
```

```
Car car = new TrashCar();
```

- 多态性是指不同的对象收到相同的消息时，会产生不同动作。
- C#支持两种类型的多态性：
  - （1）编译时的多态性是通过重载方法实现的，系统在编译时，根据传递的参数个数、类型决定实现何种方法。
  - （2）运行时的多态性是指在运行时，根据实际情况决定实现何种操作。C#中运行时的多态性通过虚函成员覆盖实现。

# 编译时多态---重载

- 重载指在同一个类中至少有两个方法用同一个名字，但有不同的参数。
- 重载使得从外部来看，一个操作对于不同的对象有不同的处理方法。
- 调用时，根据参数的不同来区别调用哪个方法。
- 方法的返回类型可以相同或不同，但它不足以使返回类型变成唯一的差异。重载方法的参数表必须不同。

```
class Car
{
    int color;
    int door;
    int speed;
    public void PushBreak()
        { speed = 0; }
    public void PushBreak(int s)
        { speed -= s; }
}
```

```
Car car = new Car();
car.PushBreak();
car.PushBreak(2);
```

## 运行时多态---动态绑定(虚函数)

- 动态绑定就是根据对象的类型决定调用哪个方法，而不是引用的类型。
- 类的方法使用virtual关键字修饰后就成为虚方法，包括两个步骤：
  - (1).对于基类中要实现多态性的方法，用virtual关键字修饰。不允许再有static,abstract或override修饰符。
  - (2).对于派生类中的同名方法(覆盖) --相同的名称、返回类型和参数表，使用override关键字修饰。不能有new、static或virtual修饰符。

# 多态---覆盖

- C# 中声明覆盖时，父类方法前加**virtual**关键字，表示该方法可以被覆盖；子类方法前加**override**，表示将方法覆盖。
- 当用于子类的行为与父类的行为不同时，覆盖机制允许子类可以修改从父类继承来的行为。

```
class Car
{  int colorNumber;
   int doorNumber;
   protected int speed;

   public virtual void PushBreak()
   { speed=0; }
   void AddOil() { ... }
}
```

```
class TrashCar : Car
{  double amount;
   void fillTrash() { ... }
   public override void PushBreak()
   { speed=speed-10; }
}
```



例3-11下面的代码中，子类重写了父类的虚方法sleep()。

```
public class animal //基类
{
    public virtual void sleep() //虚方法
    {
        Console.WriteLine("animal all need sleep");
    }
}
public class fish:animal //派生类
{
    public override void sleep() //重写虚方法
    {
        Console.WriteLine("fish sleeping with eye_open");
    }
}
public class dog:animal //派生类
{
    public override void sleep() //重写虚方法
    {
        Console.WriteLine("dog sleeping with eye_closed");
    }
}
```

- Main()中的代码:

```
animal[ ] an=new animal[3];
```

```
an[0]=new animal();
```

```
an[1]=new fish();
```

```
an[2]=new dog();
```

```
an[0].sleep();
```

```
an[1].sleep();
```

```
an[2].sleep();
```

输出结果:

**animal all need sleep**

**fish sleeping with eye\_open**

**dog sleeping with eye\_closed**

实现多态性的核心和实质:

使用基类的引用指向派生类的对象，当程序运行时，编译器会自动确定基类对象的实际运行时类型，并根据实际类型调用正确的方法。

### 例3-12

```
class GeometricObject
{
    public virtual void draw(){Console.WriteLine("GeometricObject!");}
}
class Ellipse:GeometricObject
{
    public override void draw(){Console.WriteLine("Ellipse!");}
    public void getvector(){}
}
class Circle:Ellipse
{
    public override void draw(){Console.WriteLine("Circle!");}
    public double getArea(){return 1.0;}
}
```

main()函数中的代码

```
GeometricObject g=new Circle(); //父类型引用指向子类型对象 g.draw();
g.draw(); //draw调用的是哪个类的方法? 如果g换成Circle类引用呢?
//如果Circle类不覆盖draw方法, 调用的是哪个类的方法?
```

```
double d=g.getArea(); // 编译时能否通过。
```

Circle!

Ellipse!

不能通过

# 重载和覆盖的区别

## 相同点：

都涉及两个同名的方法。

## 不同点：

### 1. 类层次

- (1). 重载涉及的是同一个类的两个同名方法；
- (2). 覆盖涉及的是子类的一个方法和父类的一个方法，这两个方法同名。

### 2. 参数和返回值

- (1). 重载的两个方法具有不同的参数，可以有不同返回值类型；
- (2). 覆盖的两个方法具有相同的参数，返回值类型必需相同。

# 方法的隐藏

- 若覆盖时没有使用**virtual**和**override**关键字，则称子类的方法隐藏了父类的方法。此时编译器报警告。若要消除掉警告，可以使用**new**修饰符。
- C# 会根据引用的类型决定调用哪个类的方法。

例3-14对例3-12的方法隐藏

```
class GeometricObject
{
    public void draw(){Console.WriteLine("GeometricObject!");}
}
class Ellipse:GeometricObject
{
    public new void draw(){Console.WriteLine("Ellipse!");}
    public void getvector() {}
}
class Circle:Ellipse
{
    public double getArea(){return 1.0;}
}
```

main()函数中的代码

```
GeometricObject g=new Circle(); //父类型引用指向子类型对象g.draw();
g.draw(); //draw调用的是哪个类的方法? 如果g换成Circle类引用呢?
```

```
Circle c = new Circle();
c.draw();
```

//将类Ellipse中draw()的修饰符改为private后, c.draw()的结果?

运行结果:

**GeometricObject!**

**Ellipse!**

- 关键字new和override的区别:

(1).new修饰的方法表示显式隐藏基类继承的同名方法，不能够用基类的引用访问派生类的new方法。

(2).override表示重写基类的虚方法，可以用基类的引用指向派生类对象来访问派生类的重写方法。

# 虚属性

- 由于属性的本质就是类中的方法实现，所以，不但能够实现虚方法，还能够实现虚属性。



下面代码中实现一个虚属性round。

```
public class Square //基类
{
    public double x;
    public Square(double x) //构造方法
    {
        this.x = x;
    }
    public virtual double Area()
    //虚方法
    {
        return x*x;
    }
    public virtual double round
    //虚属性
    {
        get
        {
            return (4*x);
        }
    }
}
```

```
public class Cube: Square //派生类
{
    public Cube(double x): base(x) //构造方法
    {...}
    public override double Area() //重写方法
    {
        return (6*(base.Area()));
    }
    public override double round //重写属性
    {
        get
        {
            return (3*base.round);
        }
    }
}
```

# 密封类

C#提供一种不能被继承的类，称为密封类。密封类的声明方法是在类名前加上`sealed`修饰符。修饰符`abstract`和`sealed`不能同时使用。

- 密封类不允许派生子类。

下面的代码建立了一个密封类Runtime。

```
public sealed class Runtime
{
    private Runtime();// 私有构造不允许其他代码建立类实例
    public static string GetCommandLine()// 静态方法成员
    {
        ..... // 实现代码
    }
}
public class anotherclass:Runtime //错误，不能继承密封类
{
    ..... //实现代码
}
```

## 3.8 抽象类与接口

- 抽象类
- 类中的方法不提供具体实现，但该类的派生类必须实现这些方法，这些方法在C#中称为抽象方法。
- 抽象方法必须是一个没有被实现的空方法。包含抽象方法的类称为抽象类，抽象类中也可以包含非抽象方法。
- 因为抽象类是用来作为基类的，所以不能直接被外部程序实例化，而且也不能被密封。

# 抽象类

- 通过关键字**abstract**进行标记将类声明为抽象。
- 不能创建抽象类的对象，但可以创建抽象类的引用。
- 一个abstract类可以不包含抽象方法，可以包含非抽象方法和变量。
- 抽象方法是虚方法的特例。
- 构造函数和静态方法不能是抽象的。
- 一个非abstract类不能包含抽象方法。
- 子类若要覆盖抽象类的抽象方法时，要使用override关键字。

**abstract class WashingMachine**

```
{  
    public WashingMachine()//构造函数  
    {  
        Console.WriteLine("here is WashingMachine ");  
    }  
    abstract public void Wash(); //抽象方法  
    abstract public void Rinse(int loadSize); //抽象方法  
    abstract public long Spin(int speed); //抽象方法  
}
```

抽象方法的定义

**WashingMachine m = new WashingMachine();**

- 对于上面的抽象类WashingMachine，派生类中的抽象方法可以如下实现：

```
class MyWashingMachine : WashingMachine
{
    public MyWashingMachine()
    { Console.WriteLine("here is MyWashingMachine ");}
    override public void Wash()
    { Console.WriteLine("Wash");}
    override public void Rinse(int loadSize)
    { Console.WriteLine("Rinse");}
    override public long Spin(int speed)
    { Console.WriteLine("Spin"); return (speed*1000);}
}
```

当一个类从抽象类派生时，该派生类必须实际提供**所有抽象成员**的实现，否则，该派生类仍然是抽象类。如下面的示例所示：

```
abstract public Class A
{ public abstract void F(); //抽象方法}
abstract public Class B:A
{ public void G(){}; //附加方法}
public Class C:B
{ public override void F(){.....} } //抽象方法实现
```

# 接口

接口的语法结构：

[访问修饰符] **interface** 接口标识符 [:基接口列表]

```
{  
    接口体;  
}
```

- 接口成员访问权限为**public**，但不能加访问修饰符
- 接口成员不能有定义
- 接口的成员必须是**方法，属性，事件或索引器**，不能包含常数、字段、运算符、实例构造函数、析构函数或类型。

例：

```
interface IA  
{  
    void f();  
}
```

```
interface IA  
{  
    public void f();  
}
```

错误

```
interface IA  
{  
    void f() {}  
}
```

错误

# 接口实现

- 实现：类要继承某个接口用 “:”，在类的定义中给出接口中所定义方法的实际实现。
- 除显示实现接口外，类中接口实现必须显示声明为public。

```
public interface Interface1
{
    new void fun1(int i); //隐藏基成员
    void M1(int y); //添加新成员M1
}
public class cls1:Interface1
{
    public void fun1(int i){.....}
    public void M1(int i){.....}

    //实现接口方法
    //不同实现
public class cls2:Interface1
{
    public void fun1(int i){.....}
    public void M1(int i){.....}
}
```



- public Interface IBase
- {
- void ClassPrind(string s);
- }
- 定义一个类继承于IBase接口，并且实现方法

- public class DogClass :IBase
- {
- public void ClassPrind(string s)
- {
- Console.WriteLine(s);
- }
- }

- 再定义一个类继承于IBase接口，并且实现方法

- public class CatClass :IBase
- {
- public void ClassPrind(string s)
- {
- Console.WriteLine(s+s);
- }
- }

```
IBase bas=new DogClass();  
bas.ClassPrind("小狗");
```

```
IBase bas=new CatClass();  
bas.ClassPrind("小猫");
```

只要改个类名、改个参数调用，结果就是别一个类中的方法

# 接口使用

- 接口由类实现后，接口成员可以通过对象实例访问，就好像是类的成员一样。

- **接口与对象：**

- (1).接口不是类，所以不能用接口创建对象，即不能用new运算符。

- `x = new Comparable();` //错误

- (2).可以声明接口类型的引用，该引用只能指向实现了该接口的对象。

- ```
class Student : Comparable {...}
```

- ```
Comparable x = new Student();
```

# is运算符

- is 运算符可以检查对象与类之间的关系，形式为：if ( obj is classname ).....
  - 当obj为classname类或其子类的对象时，运算返回true。
  - 如果引用没有指向对象，编译时报错。
- 可用于确定接口是否可用

```
Sphere obj = new Sphere(1);
```

```
//如果Sphere实现了ICalAreaAndVolumn接口
```

```
if(obj is ICalAreaAndVolumn) {...}
```

```
using System;
```

```
class Class1
```

```
{  
}
```

```
class Class2
```

```
{  
}
```

```
class IsTest
```

```
{
```

```
    static void Test(object o)
```

```
    {
```

```
        if (o is Class1)
```

```
        { Console.WriteLine("o is Class1"); }
```

```
        else if (o is Class2)
```

```
        { Console.WriteLine("o is Class2"); }
```

```
        else
```

```
        { Console.WriteLine("o is neither Class1 nor Class2."); }
```

```
    }
```

```
    static void Main()
```

```
    {
```

```
        Class1 c1 = new Class1();
```

```
        Class2 c2 = new Class2();
```

```
        Test(c1);
```

```
        Test(c2);
```

```
        Test("a string");
```

```
    }
```

```
}
```

**o is Class1**

**o is Class2**

**o is neither Class1 nor Class2.**

# as运算符

- as 运算符完成的功能等价于：先用is检查，再执行对象类型转换。
- 如果类型不兼容，as运算返回null。

```
Sphere obj = new Sphere(1);
```

```
ICalAreaAndVolumn myICal;
```

```
myICal = obj as ICalAreaAndVolumn;
```

```
//如果Sphere实现了ICalAreaAndVolumn接口
```

```
if(myICal != null) {...}
```

```
using System;
class Class1
{ }
```

```
class Class2
{ }
```

```
class MainClass
{
    static void Main()
    {
        object[ ] objArray = new object[6];
        objArray[0] = new Class1();
        objArray[1] = new Class2();
        objArray[2] = "hello";
        objArray[3] = 123;
        objArray[4] = 123.4;
        objArray[5] = null;

        for (int i = 0; i < objArray.Length; ++i)
        {
            string s = objArray[i] as string;
            Console.Write("{0}:", i);
            if (s != null)
            { Console.WriteLine(""" + s + """); }
            else
            { Console.WriteLine("not a string"); }
        }
    }
}
```

**0: not a string**  
**1: not a string**  
**2: 'hello'**  
**3: not a string**  
**4: not a string**  
**5: not a string**

# 接口和抽象类的对比

	抽象类	接口
不同点	用 abstract 定义	用 interface 定义
	只能继承一个父类	可以实现多个接口
	非抽象派生类必须实现抽象方法	实现接口的类必须实现所有成员
	需要override实现抽象方法	直接实现
相同点	不能实例化	
	包含未实现的方法	
	派生类必须实现未实现的方法	

# 结构体与类的区别

- 结构是一种用`struct`声明的自定义数据类型。它和类相似，可以包含构造函数，字段，属性，方法等。
- 一般情况下结构中只是一些数据，要是需要定义方法，一般将它定义为类。
- 结构不支持继承，但可继继承接口。

访问修饰字 `struct` 结构名[:接口]

{

结构体

}



下面的结构包含三个成员：

**struct SimpleStruct**

```
{  
    private int xval; //私有字段  
    public int X //属性  
    {  
        get  
        { return xval;}  
        set  
        { if (value < 100) xval = value;}  
    }  
    public void DisplayX() //方法  
    { Console.WriteLine("The stored value is: {0}", xval);}  
}
```

### 例3—18

```
public struct Point
```

```
{
```

```
    public int x, y;
```

```
    public Point(int x, int y) //构造函数必须带参数，且必须对字段进行赋值
```

```
    {
```

```
        this.x = x;
```

```
        this.y = y;
```

```
    }
```

```
    public void draw()
```

```
    {
```

```
        Console.WriteLine("结构对象创建之后才能调用");
```

```
    }
```

```
}
```

```
class MainClass
```

```
{ public static void Main()
```

```
{
```

```
    Point pt1; //可以不用new建立结构对象
```

```
    pt1.x = 10; //初始化结构对象，才能使用，否则出错
```

```
    pt1.y = 20;
```

```
    Point pt2=new Point(20,20); //用new建立结构对象
```

```
    Point pt3=new Point( ); //调用了系统提供的默认构造函数
```

```
    // 打印显示
```

```
    Console.WriteLine("pt1:");
```

```
    Console.WriteLine("x = {0}, y = {1}", pt1.x, pt1.y);
```

```
    Console.WriteLine("pt2:");
```

```
    Console.WriteLine("x = {0}, y = {1}", pt2.x, pt2.y);
```

```
}
```

```
}
```

对于结构的使用需要注意以下几点：

- 1.结构不能包含显式的无参数构造函数。
2. 显式定义的构造函数必须带参数。
- 3.与类不同，结构不能继承其他的结构或类，但可继承包接口。
- 4.在声明了结构类型后，可以使用new运算符创建结构对象。如果不使用 New 操作符，只有在所有的字段都被初始化之后，字段才被赋值，对象才被使用。
- 5.结构是值类型，类是引用型。

以下任何一条，应该使用类：

- 1.需要派生其他类型。
- 2.需要继承。
- 3.该类型作为方法参数传递。因为结构是值类型，每次调用都要创建结构的副本。但结构放在数组中例外。
4. 该类型用作方法的返回类型（地址）。

## 实验4 函数

实验要求：

- 1、函数传参、传址；
- 2、继承：父类与子类；
- 3、多态（虚函数）；
- 4、结构体应用（声明，传值）。

# 第4章

## C#异常调试

# 调试

应用程序必须

无错误

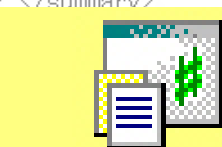
无故障

可靠

稳健

可以安装在

此方法的内容。

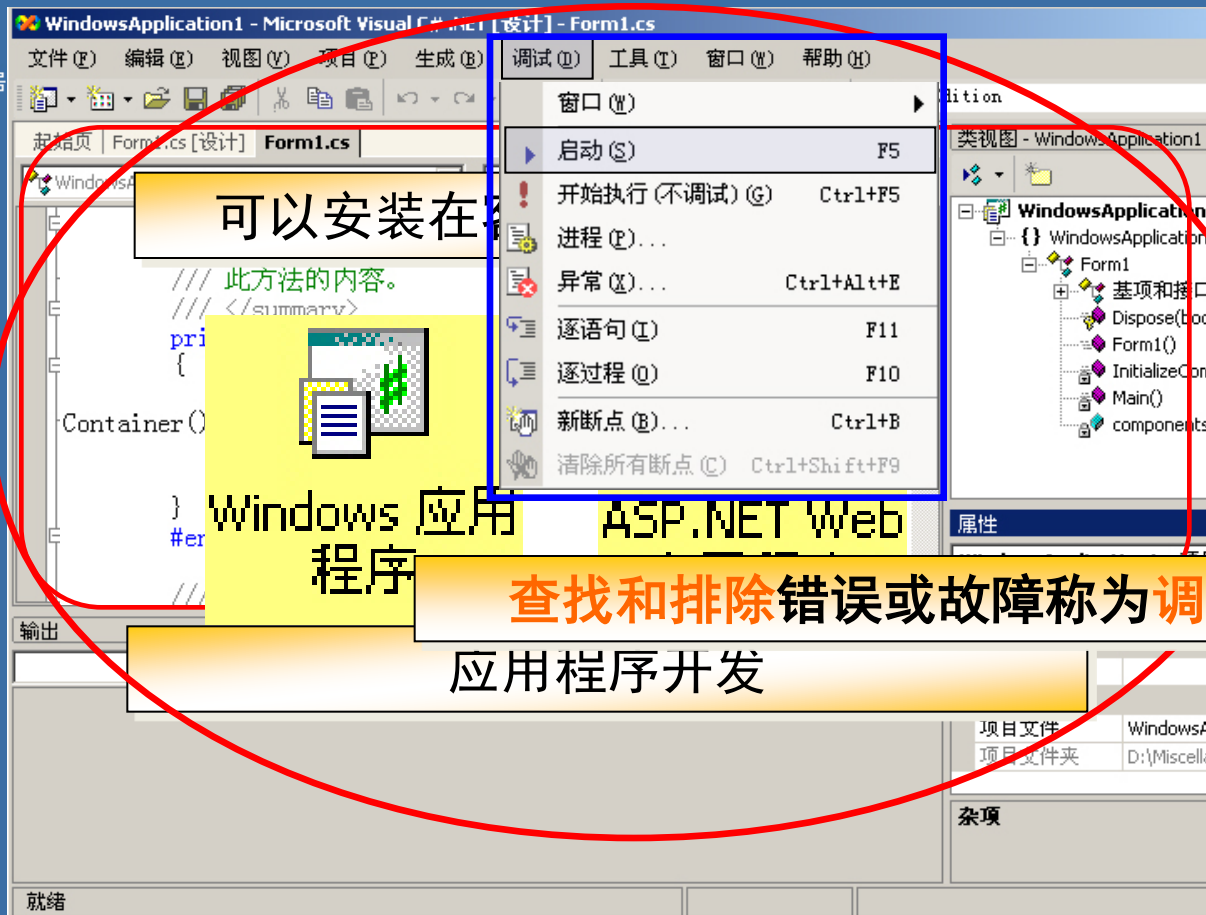


Windows 应用  
程序

ASP.NET Web

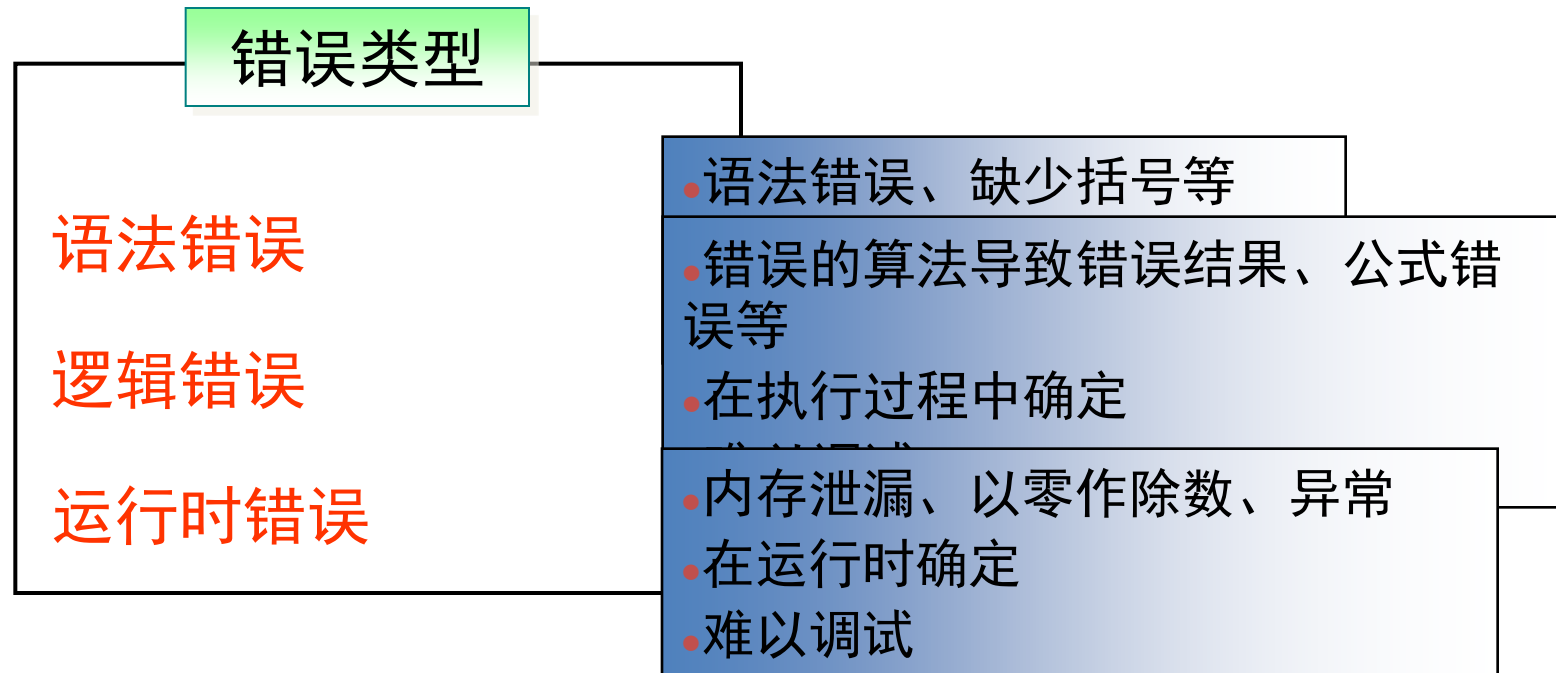
查找和排除错误或故障称为调试

应用程序开发





# 错误类型



# 调试过程

调试器

观察程序的运行时行为

跟踪变量的值

确定语义错误的位置

查看寄存器的内容

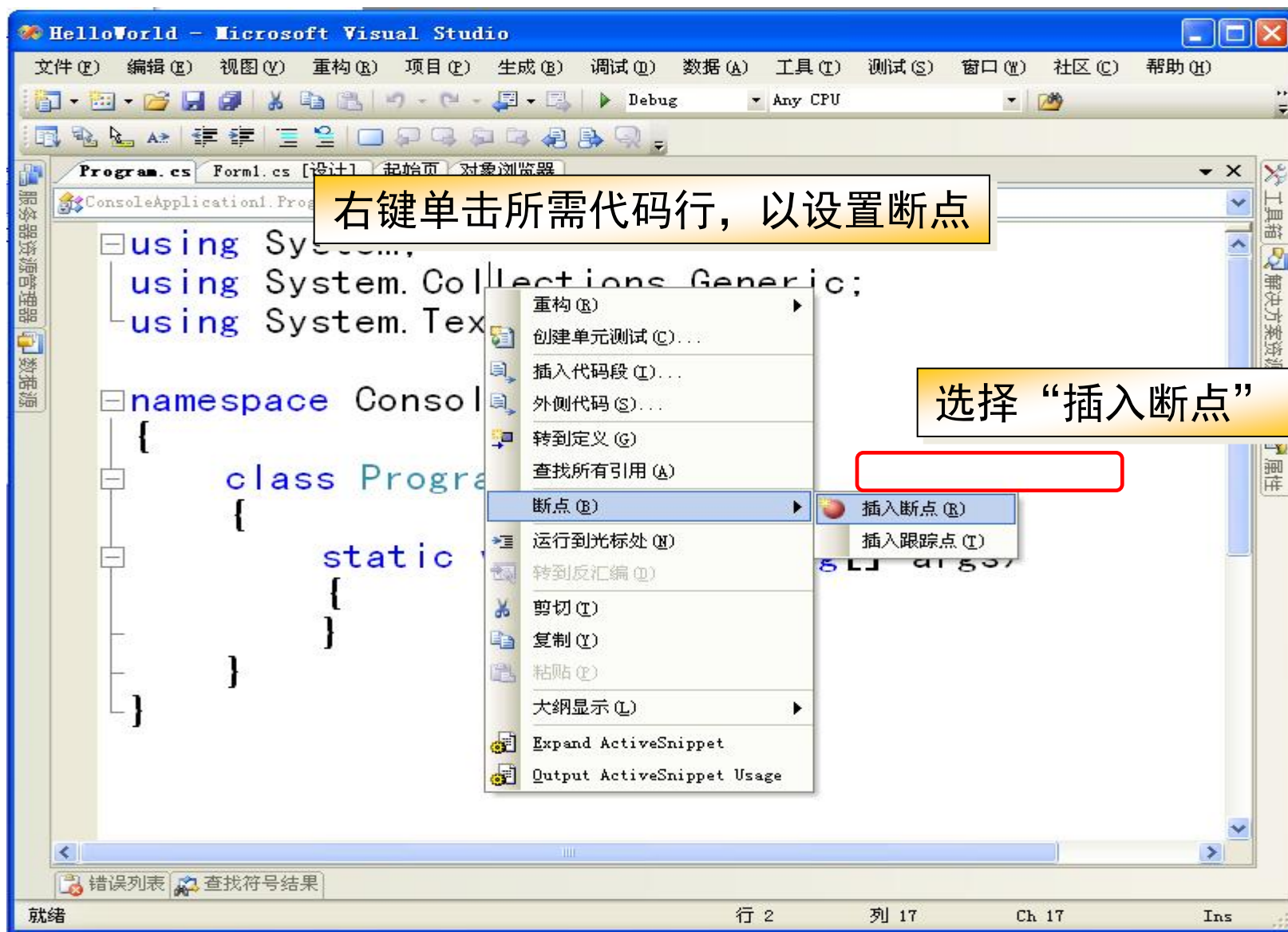
查看内存空间

# 调试过程

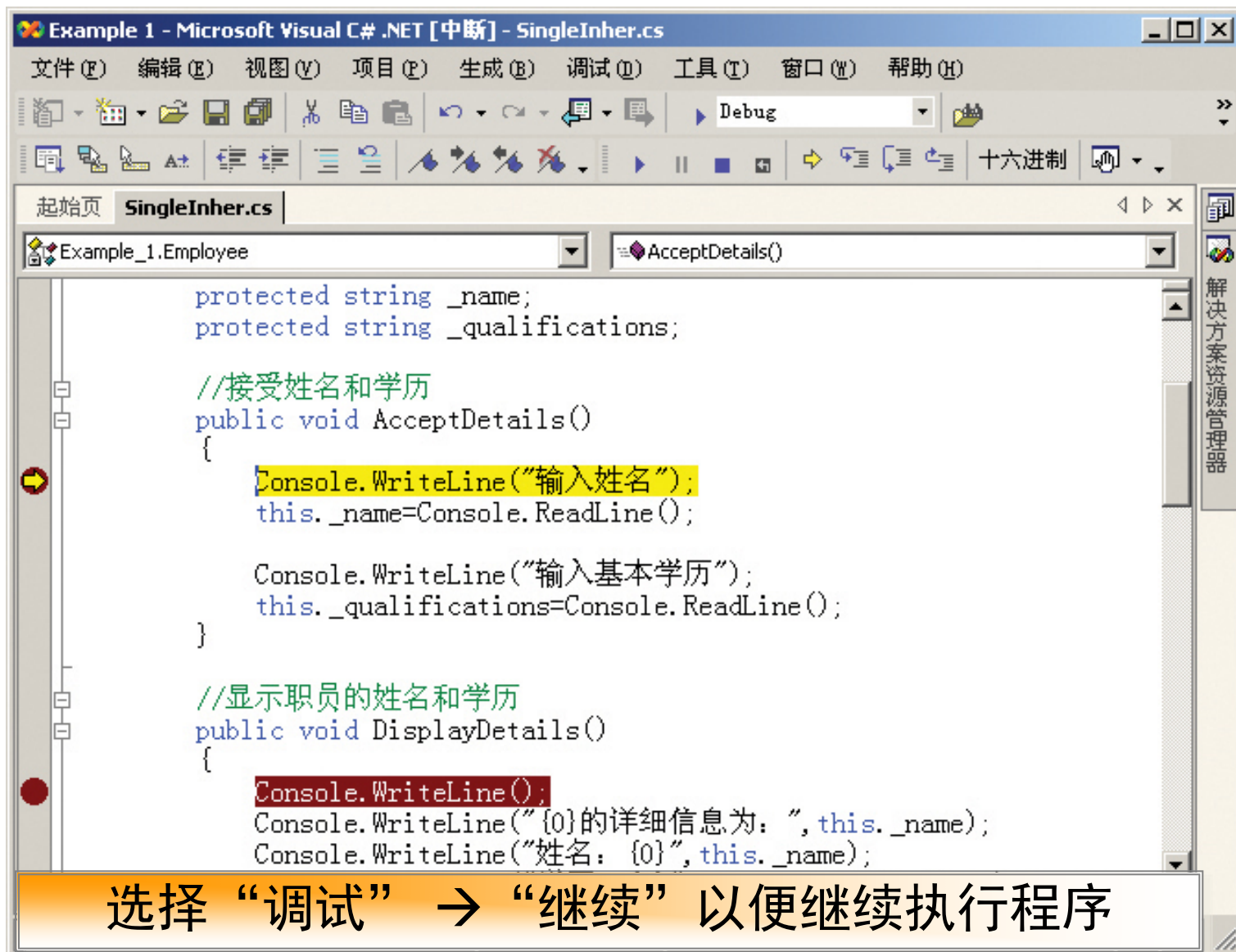


可在代码中插入“断点”，以便在特定行处暂停执行该代码

# 调试过程



# 调试过程

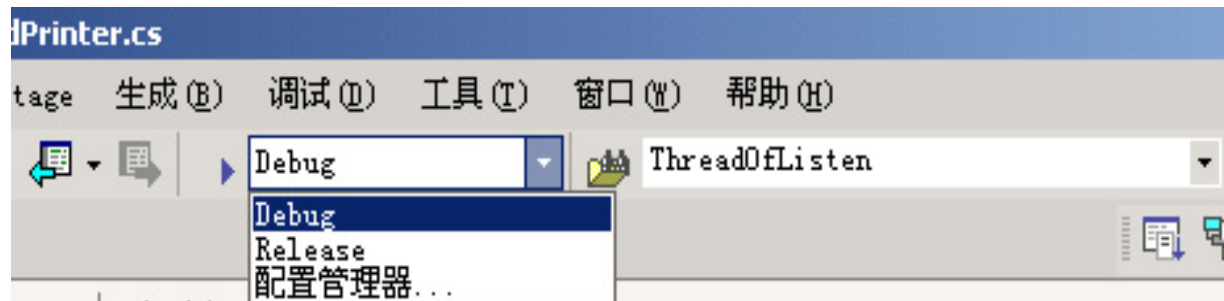


# 调试过程

## .NET 集成开发环境

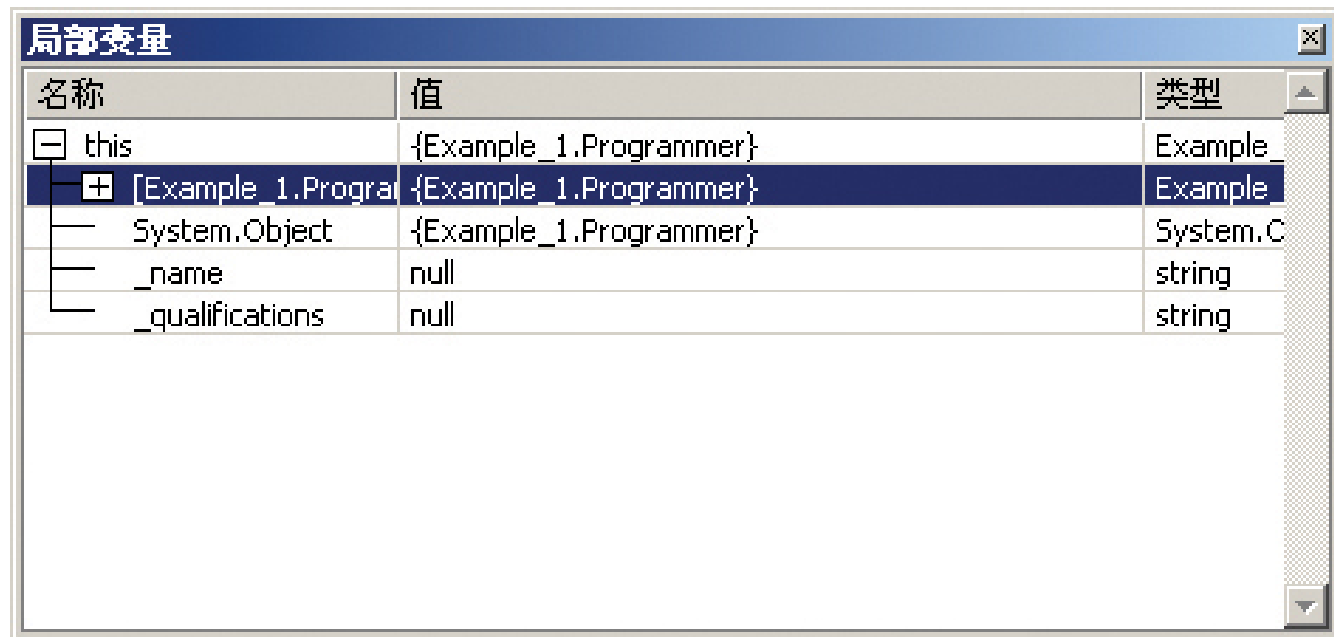
Debug模式

Release模式



# VS.NET 中的调试工具

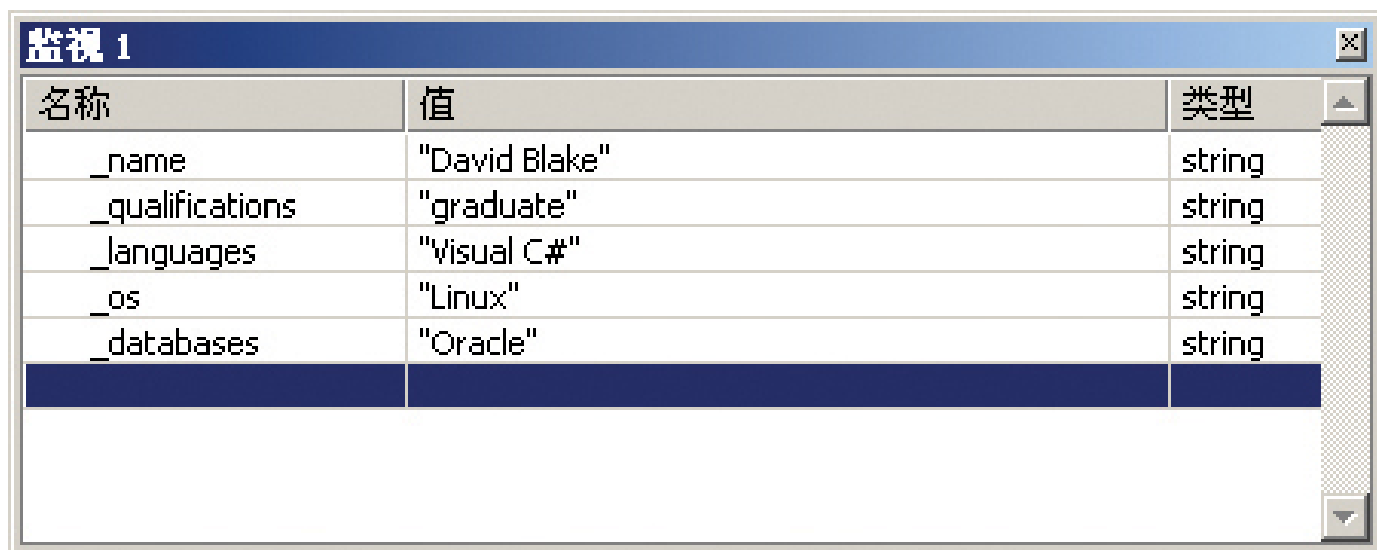
## “局部变量” 窗口



名称	值	类型
this	{Example_1.Programmer}	Example
[Example_1.Progra	{Example_1.Programmer}	Example
System.Object	{Example_1.Programmer}	System.C
_name	null	string
_qualifications	null	string

# VS.NET 中的调试工具

## “监视”窗口



名称	值	类型
_name	"David Blake"	string
_qualifications	"graduate"	string
_languages	"Visual C#"	string
_os	"Linux"	string
_databases	"Oracle"	string



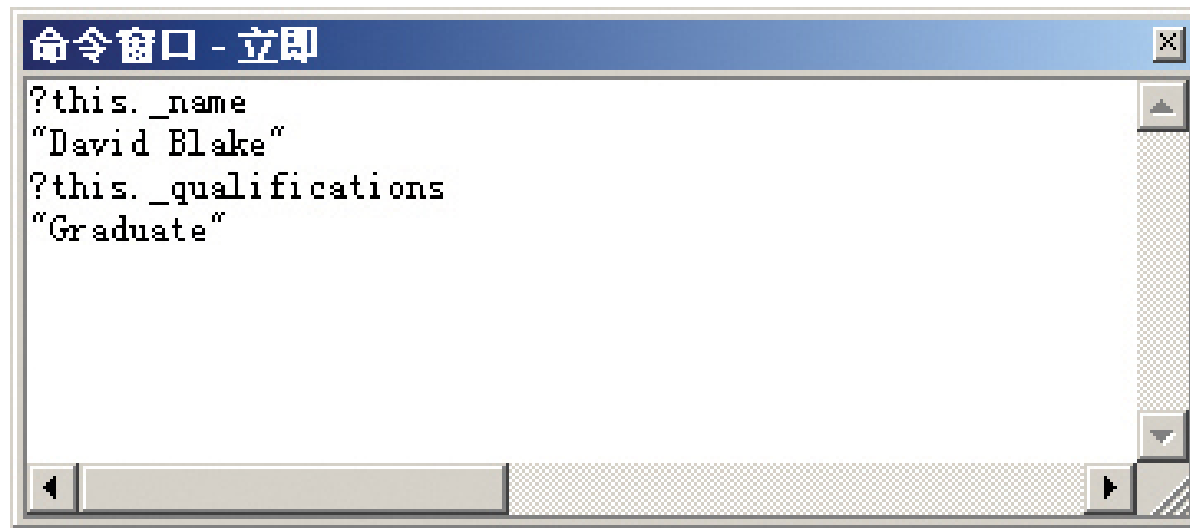
# VS.NET 的调试工具

## “快速监视”对话框



# VS.NET 中的调试工具

## “即时” 窗口

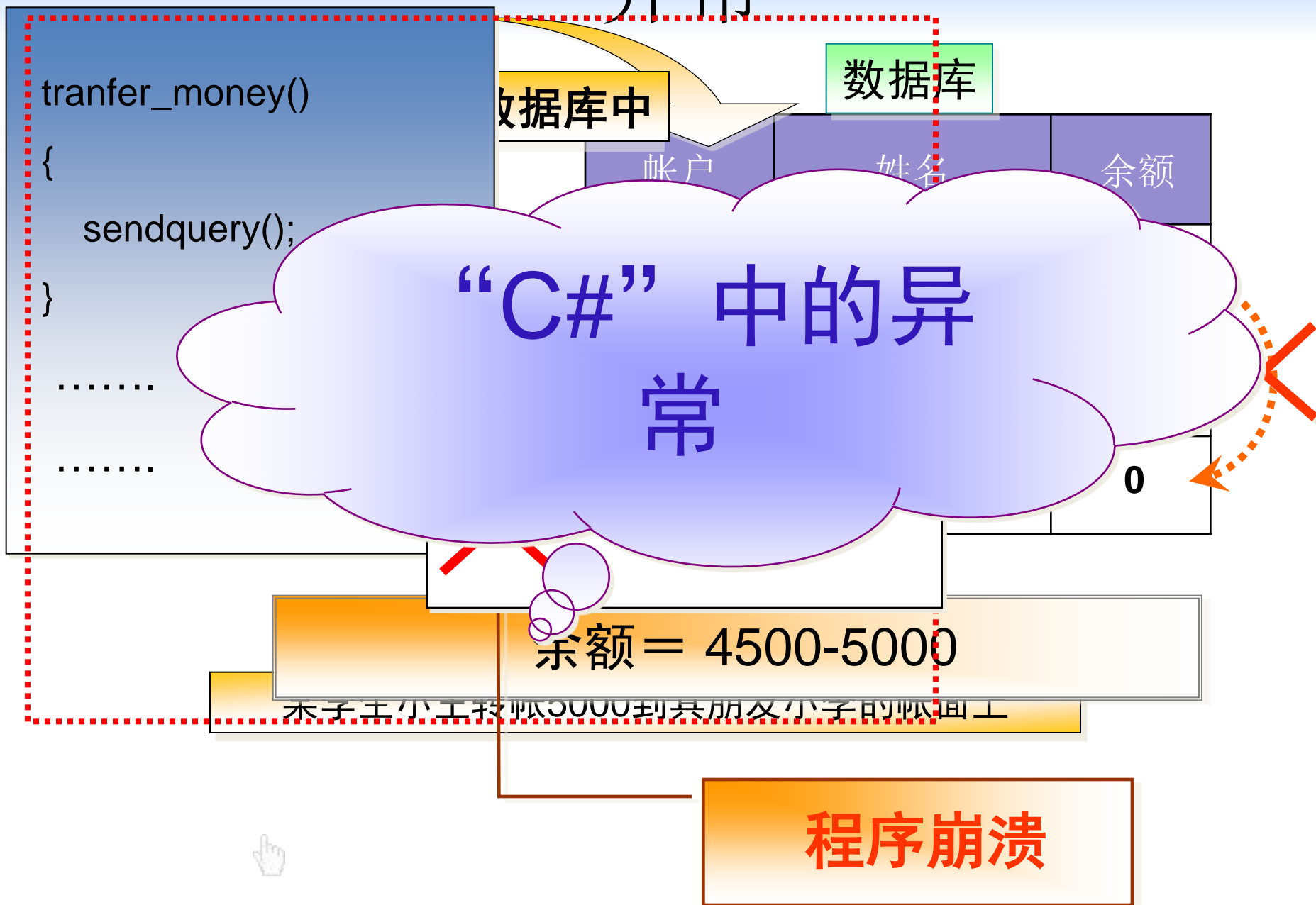


# VS.NET 中的调试工具

## Visual Studio .NET 调试器的功能

- ◆ 跨语言调试
- ◆ 调试使用 .NET 框架编写的应用程序以及 Win32 本机应用程序
- ◆ 加入正在运行的程序
- ◆ 调试多个程序

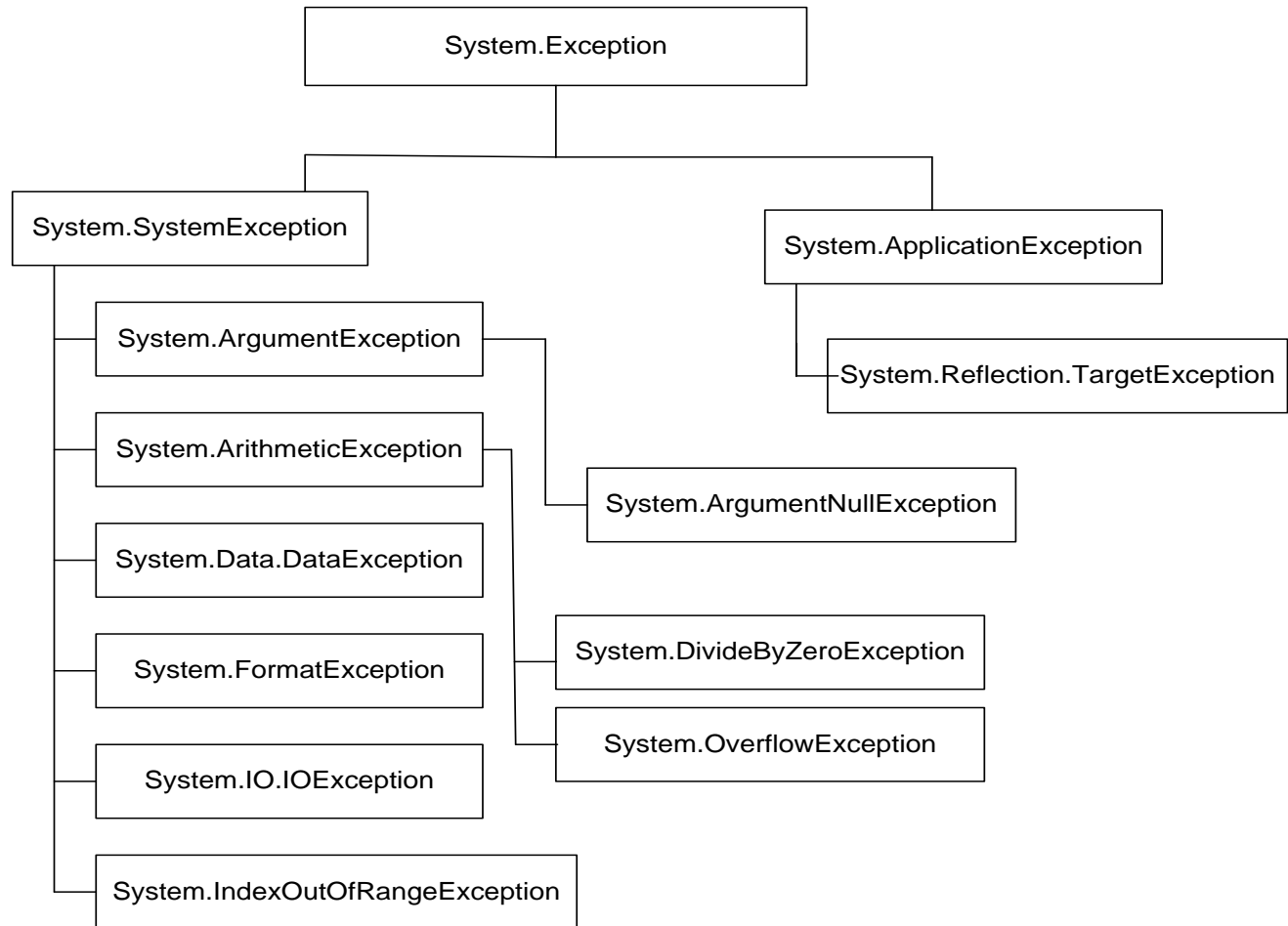
# 异常



# 错误与异常

- **错误：** 可预见，如信用卡号格式不对或口令不对。可由程序代码进行排除。
- **异常：** 与程序无关的外部原因造成。如数据表不可用或硬件故障等。

# System.Exception



# System.Exception

## 属性

**Message** 获取描述当前异常的消息。

**Source** 获取或设置导致错误的应用程序或对象的名称(程序集的名称)。

**TargetSite** 获取引发当前异常的方法。

**StackTrace** 获取当前异常发生时调用堆栈上的帧的字符串表示形式。

# System.Exception

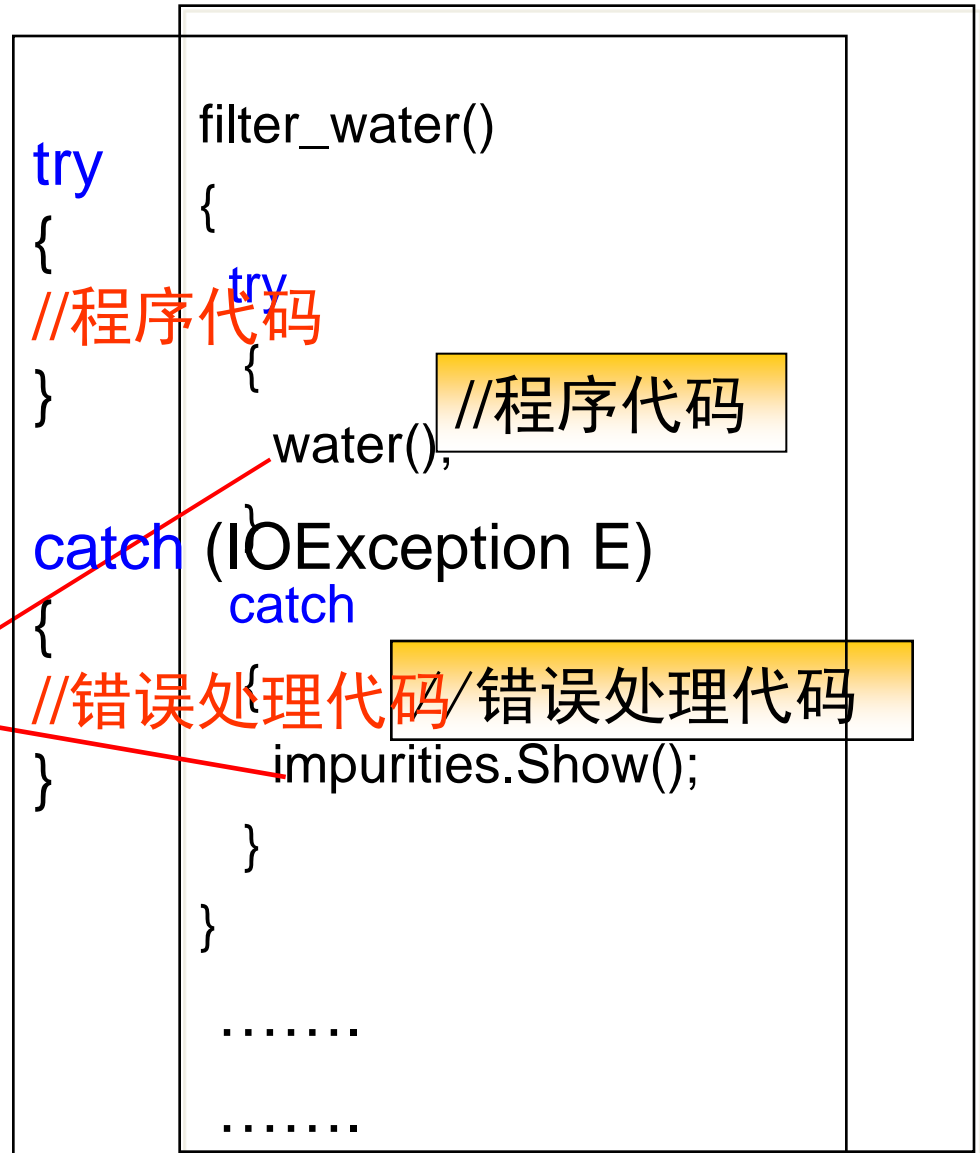
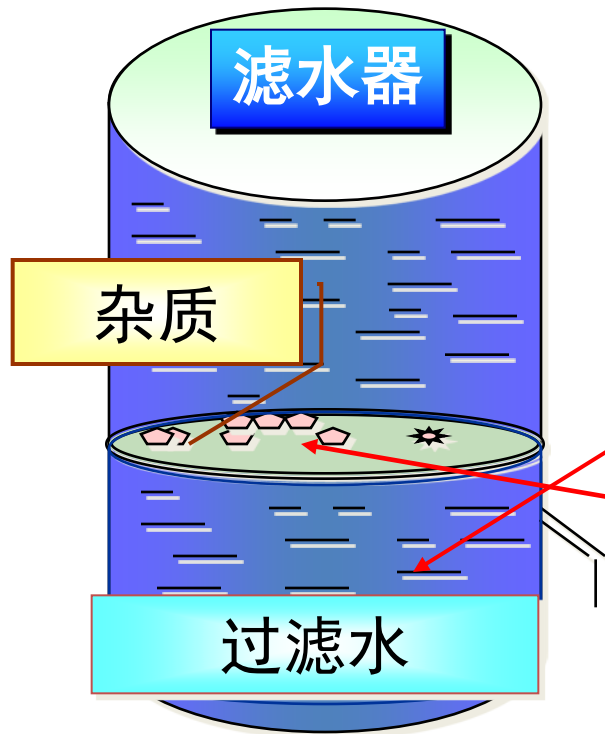
在 C# 程序中，引发异常共有以下两种方式

- 使用显式 throw 语句来引发异常。在此情况下，控制权将无条件转到处理异常的部分代码
- 使用语句或表达式在执行过程中激发了某个异常的条件,使得操作无法正常结束,从而引发异常

**Try...Catch...Finally**



# try 和 catch 块



# try 和 catch 块

```
try
{
    //程序代码
}

catch (IOException E)
{
    //错误处理代码
}
```

引发I/O 设备异常

# try 和 catch 块

```
try
{
//程序代码
}

catch( System.Exception E)
{
//错误处理代码
}
```

可处理系统中的任何一种异常

# try 和 catch 块

```
if (grade < 0 && grade > 150)
{
    throw new InvalidNumberInput
    (grade+ “不是合法的成绩” );
}
```

**throw** 可用来引发自定义异常 “InvalidNumberInput”

# 使用 finally

```
try
{
//程序代码
}

catch
{
//错误处理代码
}

finally
{
//finally 代码
}
```

无论有否异常该代码都会执行

# 多重 catch 块

```
try
{
    //程序代码
}

catch (IOException E)
{
}

catch (OutOfMemoryException E)
{
    //错误处理代码
}
```

用于捕捉两种异常的“**catch**”块