测量程序设计

C#–WPF

朱学军编著

xazhuxj@qq.com

西安科技大学

测绘科学与技术学院

二零一八年三月

本文档仅适用于西安科技大学测绘相关专业本科二年级以上学生教学使用。

也可用于其他专业人员参考。

教学计划及课程目标

教学计划

教学课时

该课程标准学时为 64 学时。本学期为 60 学时，上课 15 次 30 学时，上机 15 次 30 学时。
必修课，考试。笔试（60%）+ 上机实验与作业（40%）

学习内容

学习面向对象的程序设计与编写；
学习基本的测量算法程序编写，学习软件界面的编写。

编程预言的选择

注意：该课程不是再次学习某种编程语言，而是运用某种编程语言进行测量数据处理及测量算法的编写。

可能大家只学习了 C 语言，C 语言是面向过程的功能强大的语言，执行效率高，但界面程序编写较复杂。现代编程语言更多的是面向对象的编程语言，高效而且兼容 C 语言的是 C++，但界面的编写仍较复杂。因此我们选择更加现代化的编程语言 C#，它语法优美，界面编写方式有 WinForm 与 WPF 方式，虽然执行效率没有 C++ 高，但在 Windows7、Windows8/8.1、Windows10 中都几乎预装了运行库.Net Frame，并且得到了 Microsoft 的大力推广，Microsoft 的许多软件都在使用 C# 开发，学习与开发成本相比与 C++ 显著减少，效率显著提升。

C# 的语法与 JAVA 的许多语法是极其相似的，现代软件工程的许多方法也可以用 C# 去实现。因此，在本课程中我们将学习 C# 编程语言，并且会学习如何运用 C# 语言进行测量程序设计及测量数据处理。

现代软件基本上都具有简洁易用的界面，我们还将学习 WPF 的界面编写技术，为我们的程序设计处美观简洁的界面，这些都包含在 C# 之中。

编程环境与工具

本课程的基本编程工具为 Visual Studio，版本理论上为 2010 及更新版，推荐大家用 Visual Studio 2015 或 2017 版（目前的最新版）。

参考教材

C# 语言及 WPF 界面编写参考马骏主编，人民邮电出版社出版，《C# 程序设计及应用教程》第 3 版。

测量算法参考本教程（一直更新中.....）。

目录

教学计划及课程目标 i

目录 iii

第一章 从 C 走向 C# 1

1.1 Client/Server 程序设计模式 . . . 1

1.2 从面向过程走向面向对象的程序设计 1

1.2.1 面向过程的 C 代码示例 . 2

1.2.2 对 C 代码的解释 2

1.2.3 面向对象的 C# 代码 . . . 3

1.3 走向面向对象的 C# 代码 5

1.4 作业 9

第二章 C# 语言基础 11

2.1 C# 程序的组织结构 11

2.2 C# 中的数据类型 11

2.2.1 值类型 11

2.2.2 引用类型 11

2.2.3 自定义类型 12

2.2.4 装箱与拆箱 12

2.2.5 表达式 13

2.2.6 语句 13

第三章 C# 类与封装 17

3.1 类和对象 17

3.1.1 成员 17

3.1.2 可访问性 18

3.1.3 类型形参 18

3.1.4 字段 18

3.1.5 方法 19

3.1.6 静态方法和实例方法 . . . 21

3.1.7 构造函数 21

3.1.8 属性 22

第一章　从 C 走向 C#

基本上我们都学习过 C 语言，有过一定的软件设计基础。C 语言是功能强大的高效程序设计语言，但在现代的软件开发中特别是 Window 界面程序中开发的效率太低了。Microsoft 推出的.Net Framwork 在新版 Windows 中越来越多的进行了预安装，C# 语言面向对象、语法优美、功能强大、开发效率高，应用也越来越广泛。因此我们将从 C 语言程序设计转向 C# 程序设计，学习更多的现代程序设计方法，满足现代测绘大数据的处理与算法编写需求。

1.1　Client/Server 程序设计模式

现代软件往往是多人协作开发的成果，单个人进行大型软件的开发是比较少的。在软件开发中需要遵循软件工程的组织原则，寻求代码的可复用性、可测试性与可阅读性。在学习编程时我们首先应该改掉以下三个不良编码习惯：

改变 C 语言中将代码写入 main 函数中的习惯；

改变 C# 语言中将代码写入 Main 函数中的习惯；

改变在 UI(WinForm 或 WPF 或其它的界面) 中直接写入逻辑算法代码的习惯。

以上三种形式的代码书写处我们可以称为 Client,我们编写的逻辑算法代码可以称为 Server。试想如果我们去商场买东西，我们就是 Client，提供需求；商场就是 Server，提供服务。如果我们要买一只圆珠笔，也许我们只需简单的告诉商场人员，然后付钱拿笔走人就行了。但如果商场要让我们自己去仓库里找笔、查价钱、再在商场登记簿上登记库存等等一些动作，你想想会是怎么样的一种结果？买只笔都要把我们累死了。

这说明了什么呢？提供服务功能的商场应该对有功能需求的客户简化和屏蔽各种复杂的中间环节。在软件开发时同样如此，我们的 main 或 Main 函数或 UI 处的代码应该简洁，基本上只是调用各个功能算法而已。

如果我们像以上这三种方式组织代码，将会带来一系列的问题，尤其在团队开发与多人协作时代码不能复用，不能进行 unittest(单元测试)、不能用 git 工具（著名的源代码管理工具）进行源代码的自动合并。软件系统稍微复杂一点，我们的开发就会面临失败的危险。

万丈高楼从地起，因此，在学习编程时首先我们要有 Client/Server 模式意识，要遵循界面与算法相分离的原则进行程序设计。

1.2　从面向过程走向面向对象的程序设计

良好的面向过程设计程序设计程序是可以很好的转向面向对象的程序设计的，我们将从一个简单的 C 程序开始设计结构较为良好的 C 代码，再将其用面向对象的 C# 进行实现。从中体会面向过程与面向对象的程序设计方法的不同。

1.2.1 面向过程的 C 代码示例

在测绘专业中我们经常与测量点打交道，因此我们定义一个点 Point（测点除了它的坐标 x,y 和高程 z 之外，还需至少有点名），另外我们定义一个简单的为大家所熟知的圆 Circle，来使用点 Point 定义它的圆心，并实现判断两圆是否相交，计算圆的面积和周长等功能。代码如下所示：

```
1  // Ch01Ex01.cpp : Defines the entry point for the console application.
2  //
3
4  #include "stdafx.h"
5
6  #define _USE_MATH_DEFINES
7  #include <math.h>
8  #include <string.h>
9  #define PI M_PI
10
11 typedef struct _point {
12     char name[11];
13     double x, y, z;
14 }Point;
15
16 typedef struct _circle {
17     Point center;
18     double r;
19     double area;
20     double length;
21 }Circle;
22
23 //计算圆的面积
24 void Area(Circle * c) {
25     c->area = PI * c->r * c->r;
26 }
27
28 //计算两点的距离
29 double Distance(const Point * p1, const Point * p2) {
30     double dx = p2->x - p1->x;
31     double dy = p2->y - p1->y;
32     return sqrt(dx*dx + dy * dy);
33 }
34
35 //判断两圆是否相交
36 bool IsIntersectWithCircle(const Circle * c1, const Circle * c2) {
37     double d = Distance(&c1->center, &c2->center);
38     return d <= (c1->r + c2->r);
39 }
```

1.2.2 对 C 代码的解释

代码中的 11 和 16 行定义结构体时使用了 typedef，这样在标准 C 中再定义 Point 类型或 Circle 类型时可以避免在其前面加关键词 struct。

第 24-26 行计算圆的面积，传入了圆的指针，这是 C 及 C++ 中传递自定义数据类型的常用的高效方式。由于计算的圆面积会保存在结构体 Circle 中的 area 中，所以函数不需要返回值，返回值定义为 void。

第 29-33 行为计算两点的距离，由于该函数不会修改 p1、p2 两个 Point 内的任何成员，故应该将这两个指针定义为 const 类型，防止函数内部无意或故意的修改。

同理第 36-39 行判断两圆是否相交，由于该函数不会修改 c1、c2 两个圆内的任何成员，也应将这两个指针定义为 const 类型。

第 37 行由于 Distance 函数的参数需要两个 Point 指针类型的参数，尽管 c1 与 c2 均为指针，但 c1->center 与 c2->center 不是指针，所以需要在其前用取地址符 & 。

第 38 行的关系运算符本身的运算结果就为 bool 型，故不需要进行 if、else 类型的判断。

相应的 main 函数测试代码如下：

```
1  int main()
2  {
3      Point pt1;
4      strcpy_s(pt1.name, 11, "pt1");
5      pt1.x = 100; pt1.y = 100; pt1.z = 425.324;
6
7      Point pt2;
8      strcpy_s(pt2.name, 11, "pt2");
9      pt2.x = 200; pt2.y = 200; pt2.z = 417.626;
10
11     Circle c1;
12     c1.center = pt1; c1.r = 80;
13
14     Circle c2;
15     c2.center = pt2; c2.r = 110;
16
17     Area(&c1) ; //计算圆c1的面积
18     printf("Circle1 的面积 = %lf \n", c1.area );
19
20     bool yes = IsIntersectWithCircle(&c1, &c2);
21     printf("Circle1 与 Circle2 是否相交 : %s\n", (yes ? "是" : "否") );
22
23     return 0;
24 }
```

程序的运行结果如下：

```
Circle1 的面积 = 20106.192983
Circle1 与 Circle2 是否相交 : 是
```

1.2.3 面向对象的 C# 代码

将以上代码相对应的 C 代码直接翻译为 C# 代码为：

```
1  using System;
2
3  namespace Ch01Ex02
4  {
5      class Point
6      {
7          public string name;
8          public double x;
9          public double y;
10         public double z;
11
12         public double Distance(Point p2)
13         {
14             double dx = x - p2.x;
```

```
            double dy = y - p2.y;
            return Math.Sqrt(dx * dx + dy * dy);
        }
    }

    class Circle
    {
        public Point center;
        public double r;
        public double area;
        public double length;

        public void Area()
        {
            area = Math.PI * r * r;
        }

        public bool IsIntersectWithCircle(Circle c2)
        {
            double d = this.center.Distance(c2.center);
            return d <= (r + c2.r);
        }
    }
}
```

由以上 C# 代码可以看出与 C 代码的不同。这是因为 C# 语言是纯面向对象语言的缘故。

程序或软件的基本概念是：程序 = 数据结构 + 算法。在以上 C 或 C# 代码中，Point 和 Circle 都可以看作是数据结构，计算两点距离和圆的面积或判断两圆是否相交都可以看作是算法。在 C 语言中，数据结构和算法是分离的，尽管函数的参数是数据结构，但函数并不属于数据结构。在 C# 中则不同，数据结构用 class 定义，则数据与算法都属于同一个 class。

计算两点距离的函数设计为 Point 类的一个方法，可以看作是计算当前点与另一个点的距离，因此函数的参数就只有一个 Point p2，另一个自然是 this（当前类本身）了。

同样的道理圆的面积计算也不需要参数了（可以想象为计算知道自己的半径，计算自己的面积并存入自己的成员变量中了）。判断两圆是否相交也可以看作是判断当前圆与另一个圆 Circle c2 是否相交了，股只需要一个参数。

相应 C# 的 Main 函数测试代码如下：

```
using System;

namespace Ch01Ex02
{
    class Program
    {
        static void Main(string[] args)
        {
            Point p1 = new Point();
            p1.name = "p1";
            p1.x = 100;
            p1.y = 100;
            p1.z = 425.324;


            Circle c1 = new Circle();
            c1.center = p1;
            c1.r = 80;
```

```
19
20              Point p2 = new Point();
21              p2.name = "p2";
22              p2.x = 200;
23              p2.y = 200;
24              p2.z = 417.626;
25
26              Circle c2 = new Circle();
27              c2.center = p2;
28              c2.r = 110;
29
30              c1.Area(); //计算圆c1的面积
31              Console.WriteLine("Circle1的面积={0}", c1.area );
32
33              c2.Area(); //计算圆c2的面积
34              Console.WriteLine("Circle2的面积={0}", c2.area);
35
36              bool yes = c1.IsIntersectWithCircle(c2);
37              Console.WriteLine("Circle1与Circle2是否相交:{0}",
38                  yes ? "是" : "否" );
39          }
40      }
41  }
```

程序的运行结果如下：

```
Circle1的面积=20106.1929829747
Circle2的面积=38013.2711084365
Circle1与Circle2是否相交:是
```

1.3 走向面向对象的 C# 代码

上面的 C# 不符合面向对象的软件设计原则（违背封装、继承与多态中的封装原则），且在 Main 函数中，如果由于疏漏忘记第 30 行或 33 行对圆的面积进行计算，圆 c1 或 c2 将不会有正确的面积。

因此，我们将其优化。首先对 Point 类进行封装，将其字段定义为 private，用属性方法将其向外暴露接口，相应的 C# 代码为：

```
1           private string name;
2           public string Name
3           {
4               get { return name; }
5               set { name = value; }
6           }
7
8           private double x;
9           public double X
10          {
11              get { return x; }
12              set { x = value; }
13          }
14          private double y;
15          public double Y
16          {
17              get { return y; }
```

```
18            set { y = value; }
19        }
20        private double z;
21        public double Z
22        {
23            get { return z; }
24            set { z = value; }
25        }
```

以上是用 C# 的属性对 private 字段的进行简单的封装，也可以写成如下形式：

```
1         public string Name { get ; set ; }
2         public double X { get ; set ;}
3         public double Y { get ; set ; }
4         public double Z {get ; set ;}
```

为了简化 Main 函数对 Point 类的调用及对其各字段值的初始化，我们定义构造函数如下：

```
1         public Point(double x, double y)
2         {
3             this.name = "";
4             this.x = x;
5             this.y = y;
6             this.z = 0;
7         }
8
9         public Point(string name, double x, double y, double z)
10        {
11            this.name = name;
12            this.x = x;
13            this.y = y;
14            this.z = z;
15        }
```

同样道理，也应对 Circle 进行封装，将其各个字段定义为 private，并用属性方法将其向外简单的暴露接口，相应的 C# 代码为：

```
1         public Point Center { get; set; }
2
3         private double r;
4         public double R
5         {
6             get { return r; }
7             set
8             {
9                 if(value != r && value >= 0)
10                {
11                    r = value;
12                    CalArea(); //r的值发生改变，重新计算圆的面积
13                }
14            }
15        }
```

圆心 Center 只是简单的封装，圆的半径则不同。由圆的特性知，当圆的半径确定，其面积与周长也就确定了。为了计算的高效，封装时，当圆的半径值发生改变时，就需要重新计算圆的面积，确保面积的正确性。

由于圆的面积只与圆的半径有关，我们不需要在其他的情况对圆的面积进行修改，因此封装的 C# 代码为：

```
        private double area;
        public double Area
        {
            get { return area; }
        }
```

在此我们去掉了属性 Area 中的 set 语句，并将 area 成员定义为了 private，使外部无法修改 area 的值，来保证圆的面积 Area 的正确性。

还有一个地方可能会修改圆的半径，就是圆的初始化时，因此圆的构造函数应定义为：

```
        public Circle(double x, double y, double r)
        {
            Center = new Point(x, y);

            this.R = r;  //赋值给R而不是this.r，确保计算圆的面积
        }
```

为确保在半径 r 的值发生改变后能重新计算圆的面积和周长，此处赋值给半径属性 R。

完整的优化后的 C# 代码如下：

```
using System;

namespace Ch01Ex03
{
    class Point
    {
        private string name;
        public string Name
        {
            get { return name; }
            set { name = value; }
        }

        private double x;
        public double X
        {
            get { return x; }
            set { x = value; }
        }
        private double y;
        public double Y
        {
            get { return y; }
            set { y = value; }
        }
        private double z;
        public double Z
        {
            get { return z; }
            set { z = value; }
        }

        public Point(double x, double y)
        {
            this.name = "";
            this.x = x;
```

```
37              this.y = y;
38              this.z = 0;
39          }
40
41          public Point(string name, double x, double y, double z)
42          {
43              this.name = name;
44              this.x = x;
45              this.y = y;
46              this.z = z;
47          }
48
49          public double Distance(Point p2)
50          {
51              double dx = X - p2.X;
52              double dy = Y - p2.Y;
53              return Math.Sqrt(dx * dx + dy * dy);
54          }
55      }
56
57      class Circle
58      {
59          public Point Center { get; set; }
60
61          private double r;
62          public double R
63          {
64              get { return r; }
65              set
66              {
67                  if(value != r && value >= 0)
68                  {
69                      r = value;
70                      CalArea(); //r的值发生改变, 重新计算圆的面积
71                  }
72              }
73          }
74
75          private double area;
76          public double Area
77          {
78              get { return area; }
79          }
80
81          private double length;
82
83          public Circle(double x, double y, double r)
84          {
85              Center = new Point(x, y);
86
87              this.R = r;  //赋值给R而不是this.r, 确保计算圆的面积
88          }
89
90          //计算圆的面积
91          private void CalArea()
92          {
```

```
93              area = Math.PI * r * r;
94          }
95
96          //判断两圆是否相交
97          public bool IsIntersectWithCircle(Circle c2)
98          {
99              double d = this.Center.Distance(c2.Center);
100             return d <= (r + c2.r);
101         }
102     }
103 }
```

代码中由于计算圆的面积将会在属性 R 中和构造函数中调用，我们将其定义为函数 CalArea 供多次复用，并将可访问性定义为 private 供内部使用。

相应 C# 的 Main 函数测试代码如下：

```
1  using System;
2
3  namespace Ch01Ex03
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Circle c1 = new Circle(100, 100, 80);
10             Circle c2 = new Circle(200, 200, 110);
11
12             Console.WriteLine("Circle1的面积={0}", c1.Area );
13             Console.WriteLine("Circle2的面积={0}", c2.Area);
14
15             bool yes = c1.IsIntersectWithCircle(c2);
16             Console.WriteLine("Circle1与Circle2是否相交:{0}",
17                 yes ? "是" : "否" );
18         }
19     }
20 }
```

由优化后的 C# 代码可以看出 Main 十分简洁。

1.4 作业

1. 请完成示例中的圆的周长计算功能；
2. 请试着完成多段线 Polyline 的面积与长度计算功能（提示：多段线是点序的集合）；
3. 请试着完成多边形 Polygon 的面积与长度计算功能；

第二章　C# 语言基础

由于我们都学习过 C 语言，在此我们主要讲解 C# 中不同于 C 的一些基本语法。

2.1　C# 程序的组织结构

从以上 C# 示例代码可以看出，C# 中的组织结构的关键概念是程序 (program)、命名空间 (namespace)、类型 (type)、成员 (member) 和程序集 (assembly)。C# 程序由一个或多个源文件组成。程序中声明类型，类型包含成员，并且可按命名空间进行组织。类和接口就是类型的实例。字段 (field)、方法、属性和事件是成员的实例。在编译 C# 程序时，它们被物理地打包为程序集。程序集通常具有文件扩展名.exe 或.dll，具体取决于它们是实现应用程序 (application) 还是实现库 (library)。

2.2　C# 中的数据类型

C# 中的数据类型可分为两类：值类型 (value type) 和引用类型 (reference type)。值类型的变量直接包含它们的数据，而引用类型的变量存储对它们的数据的引用，后者称为对象。

2.2.1　值类型

C# 的值类型进一步划分为简单类型 (simple type)、枚举类型 (enum type)、结构类型 (struct type) 和可以为 null 的类型 (nullable type)。对于值类型，每个变量都有它们自己的数据副本（除 ref 和 out 参数变量外），因此对一个变量的操作不可能影响另一个变量。

简单类型	有符号整型：sbyte、short、int、long 无符号整型：byte、ushort、uint、ulong Unicode 字符：char IEEE 浮点：float、double 高精度小数型：decimal 布尔：bool
结构类型	struct S ... 形式的用户定义的类型
枚举类型	enum E ... 形式的用户定义的类型
可以为 null 的类型	其他所有具有 null 值的值类型的扩展

2.2.2　引用类型

C# 的引用类型进一步划分为类类型 (class type)、接口类型 (interface type)、数组类型 (array type) 和委托类型 (delegate type)。对于引用类型，两个变量可能引用同一个对象，因此对一个变量的操作可能影响另一个变量所引用的对象。

类类型	所有其他类型的最终基类：object Unicode 字符串：string class C ... 形式的用户定义的类型
接口类型	interface I ... 形式的用户定义的类型
数组类型	一维和多维数组，例如 int[] 和 int[,]
委托类型	delegate int D(...) 形式的用户定义的类型

C# 中的 string 不是值类型数据，而是引用类型数据。

2.2.3 自定义类型

C# 程序使用类型声明 (type declaration) 创建新类型。类型声明指定新类型的名称和成员。在 C# 类型分类中，有五类是用户可定义的：类类型 (class type)、结构类型 (struct type)、接口类型 (interface type)、枚举类型 (enum type) 和委托类型 (delegate type)。

类类型定义了一个包含数据成员（字段）和函数成员（方法、属性等）的数据结构。类类型支持单一继承和多态，这些是派生类可用来扩展和专用化基类的机制。

结构类型与类类型相似，表示一个带有数据成员和函数成员的结构。但是，与类不同，结构是一种值类型，并且不需要堆分配。结构类型不支持用户指定的继承，并且所有结构类型都隐式地从类型 object 继承。

接口类型定义了一个协定，作为一个公共函数成员的命名集。实现某个接口的类或结构必须提供该接口的函数成员的实现。一个接口可以从多个基接口继承，而一个类或结构可以实现多个接口。

委托类型表示对具有特定参数列表和返回类型的方法的引用。通过委托，我们能够将方法作为实体赋值给变量和作为参数传递。委托类似于在其他某些语言中的函数指针的概念，但是与函数指针不同，委托是面向对象的，并且是类型安全的。

类类型、结构类型、接口类型和委托类型都支持泛型，因此可以通过其他类型将其参数化。

枚举类型是具有命名常量的独特的类型。每种枚举类型都具有一个基础类型，该基础类型必须是八种整型之一。枚举类型的值集和它的基础类型的值集相同。

C# 支持由任何类型组成的一维和多维数组。与以上列出的类型不同，数组类型不必声明就可以使用。实际上，数组类型是通过在某个类型名后加一对方括号来构造的。例如，int[] 是一维 int 数组，int[,] 是二维 int 数组，int[][] 是一维 int 数组的一维数组。

可以为 null 的类型也不必声明就可以使用。对于每个不可以为 null 的值类型 T，都有一个相应的可以为 null 的类型 T?，该类型可以容纳附加值 null。例如，int? 类型可以容纳任何 32 位整数或 null 值。

2.2.4 装箱与拆箱

C# 的类型系统是统一的，因此任何类型的值都可以按对象处理。C# 中的每个类型直接或间接地从 object 类类型派生，而 object 是所有类型的最终基类。引用类型的值都被视为 object 类型，被简单地当作对象来处理。值类型的值则通过对其执行装箱和拆箱操作按对象处理。下面的示例将 int 值转换为 object，然后又转换回 int。

```
1 using System;
2 class Test
3 {
4     static void Main() {
5         int i = 123;
6         object o = i;       // Boxing
7         int j = (int)o;      // Unboxing
```

```
8     }
9 }
```

当将值类型的值转换为类型 object 时，将分配一个对象实例（也称为“箱子”）以包含该值，并将值复制到该箱子中。反过来，当将一个 object 引用强制转换为值类型时，将检查所引用的对象是否含有正确的值类型，如果有，则将箱子中的值复制出来。

2.2.5 表达式

表达式由操作数 (operand) 和运算符 (operator) 构成。表达式的运算符指示对操作数适用什么样的运算。运算符的示例包括 +、-、*、/ 和 new。操作数的示例包括文本、字段、局部变量和表达式。

x.m	成员访问
x(...)	方法和委托调用
x[...]	数组和索引器访问
x++	后增量
x–	后减量
new T(...)	对象和委托创建
new T(...)...	使用初始值设定项创建对象
new ...	匿名对象初始值设定项
new T[...]	数组创建
typeof(T)	获取 T 的 System.Type 对象
checked(x)	在 checked 上下文中计算表达式
unchecked(x)	在 unchecked 上下文中计算表达式
default(T)	获取类型 T 的默认值
delegate ...	匿名函数（匿名方法）
(T)x	将 x 显式转换为类型 T
await x	异步等待 x 完成
x is T	如果 x 为 T，则返回 true，否则返回 false
x as T	返回转换为类型 T 的 x，如果 x 不是 T 则返回 null
(T x) => y	匿名函数（lambda 表达式）

其它的与 C 或 C++ 语言基本相同。

2.2.6 语句

程序的操作是使用语句 (statement) 来表示的。

声明语句 (declaration statement) 用于声明局部变量和常量。

表达式语句 (expression statement) 用于对表达式求值。可用作语句的表达式包括方法调用、使用 new 运算符的对象分配、使用 = 和复合赋值运算符的赋值、使用 ++ 和 – 运算符的增量和减量运算以及 await 表达式。

选择语句 (selection statement) 用于根据表达式的值从若干个给定的语句中选择一个来执行。这一组中有 if 和 switch 语句。

迭代语句 (iteration statement) 用于重复执行嵌入语句。这一组中有 while、do、for 和 foreach 语句。foreach 语句:

```
1 static void Main(string[] args) {
2     foreach (string s in args) {
3         Console.WriteLine(s);
```

```
    }
}
```

跳转语句 (jump statement) 用于转移控制。这一组中有 break、continue、goto、throw、return 和 yield 语句。yield 语句

```
static IEnumerable<int> Range(int from, int to) {
    for (int i = from; i < to; i++) {
        yield return i;
    }
    yield break;
}
static void Main() {
    foreach (int x in Range(-10,10)) {
        Console.WriteLine(x);
    }
}
```

try...catch 语句用于捕获在块的执行期间发生的异常，try...finally 语句用于指定终止代码，不管是否发生异常，该代码都始终要执行。throw 和 try 语句

```
 static double Divide(double x, double y) {
    if (y == 0) throw new DivideByZeroException();
    return x / y;
}
static void Main(string[] args) {
    try {
        if (args.Length != 2) {
            throw new Exception("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (Exception e) {
        Console.WriteLine(e.Message);
    }
    finally {
        Console.WriteLine( “Good bye!” );
    }
}
```

checked 语句和 unchecked 语句用于控制整型算术运算和转换的溢出检查上下文。

```
static void Main() {
    int i = int.MaxValue;
    checked {
        Console.WriteLine(i + 1);        // Exception
    }
    unchecked {
        Console.WriteLine(i + 1);        // Overflow
    }
}
```

lock 语句用于获取某个给定对象的互斥锁，执行一个语句，然后释放该锁。

```
class Account
{
    decimal balance;
```

```
    public void Withdraw(decimal amount) {
        lock (this) {
            if (amount > balance) {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}
```

using 语句用于获得一个资源，执行一个语句，然后释放该资源。

```
static void Main() {
    using (TextWriter w = File.CreateText("test.txt")) {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}
```

第三章　C# 类与封装

3.1　类和对象

类 (class) 是最基础的 C# 类型。类是一个数据结构，将状态（字段）和操作（方法和其他函数成员）组合在一个单元中。类为动态创建的类实例 (instance) 提供了定义，实例也称为对象 (object)。类支持继承 (inheritance) 和多态性 (polymorphism)，这是派生类 (derived class) 可用来扩展和专用化基类 (base class) 的机制。

使用类声明可以创建新的类。类声明以一个声明头开始，其组成方式如下：先指定类的特性和修饰符，后是类的名称，接着是基类（如有）以及该类实现的接口。声明头后面跟着类体，它由一组位于一对大括号 和 之间的成员声明组成。下面是一个名为 Point 的简单类的声明：

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

类的实例使用 new 运算符创建，该运算符为新的实例分配内存、调用构造函数初始化该实例，并返回对该实例的引用。下面的语句创建两个 Point 对象，并将对这两个对象的引用存储在两个变量中：

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

当不再使用对象时，该对象占用的内存将自动收回。在 C# 中，没有必要也不可能显式释放分配给对象的内存。

3.1.1　成员

类的成员或者是静态成员 (static member)，或者是实例成员 (instance member)。静态成员属于类，实例成员属于对象（类的实例）。

成员	说明
常量	与类关联的常量值
字段	类的变量
方法	类可执行的计算和操作
属性	与读写类的命名属性相关联的操作
索引器	与以数组方式索引类的实例相关联的操作
事件	可由类生成的通知
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	在永久丢弃类的实例之前执行的操作
类型	类所声明的嵌套类型

3.1.2 可访问性

类的每个成员都有关联的可访问性，它控制能够访问该成员的程序文本区域。有五种可能的可访问性形式。下表概述了这些可访问性。

可访问性	含义
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

3.1.3 类型形参

类定义可以通过在类名后添加用尖括号括起来的类型参数名称列表来指定一组类型参数。类型参数可用于在类声明体中定义类的成员。在下面的示例中，Pair 的类型参数为 TFirst 和 TSecond：

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}

```

要声明为采用类型参数的类类型称为泛型类类型。结构类型、接口类型和委托类型也可以是泛型。当使用泛型类时，必须为每个类型参数提供类型实参：

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;     // TFirst is int
string s = pair.Second; // TSecond is string

```

提供了类型实参的泛型类型（例如上面的 Pair<int,string>）称为构造的类型。

3.1.4 字段

字段是与类或类的实例关联的变量。使用 static 修饰符声明的字段定义了一个静态字段 (static field)。一个静态字段只标识一个存储位置。无论对一个类创建多少个实例，它的静态字段永远都只有一个副本。不使用 static 修饰符声明的字段定义了一个实例字段 (instance field)。

类的每个实例都为该类的所有实例字段包含一个单独副本。在下面的示例中，Color 类的每个实例都有实例字段 r、g 和 b 的单独副本，但是 Black、White、Red、Green 和 Blue 静态字段只存在一个副本：

```
public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte r, g, b;
    public Color(byte r, byte g, byte b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}

```

如上面的示例所示，可以使用 readonly 修饰符声明只读字段 (read-only field)。给 readonly 字段的赋值只能作为字段声明的组成部分出现，或在同一个类中的构造函数中出现。

1

3.1.5 方法

方法 (method) 是一种成员，用于实现可由对象或类执行的计算或操作。静态方法 (static method) 通过类来访问。实例方法 (instance method) 通过类的实例来访问。方法具有一个参数 (parameter) 列表（可以为空），表示传递给该方法的值或变量引用；方法还具有一个返回类型 (return type)，指定该方法计算和返回的值的类型。如果方法不返回值，则其返回类型为 void。与类型一样，方法也可以有一组类型参数，当调用方法时必须为类型参数指定类型实参。与类型不同的是，类型实参经常可以从方法调用的实参推断出，而无需显式指定。方法的签名 (signature) 在声明该方法的类中必须唯一。方法的签名由方法的名称、类型参数的数目以及该方法的参数的数目、修饰符和类型组成。方法的签名不包含返回类型。

参数

参数用于向方法传递值或变量引用。方法的参数从调用该方法时指定的实参 (argument) 获取它们的实际值。有四类参数：值参数、引用参数、输出参数和参数数组。值参数 (value parameter) 用于传递输入参数。一个值参数相当于一个局部变量，只是它的初始值来自为该形参传递的实参。对值参数的修改不影响为该形参传递的实参。值参数可以是可选的，通过指定默认值可以省略对应的实参。引用参数 (reference parameter) 用于传递输入和输出参数。为引用参数传递的实参必须是变量，并且在方法执行期间，引用参数与实参变量表示同一存储位置。引用参数使用 ref 修饰符声明。下面的示例演示 ref 参数的用法。

```
using System;
class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }
```

```
    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j);          // Outputs "2 1"
    }
}

```

输出参数 (output parameter) 用于传递输出参数。对于输出参数来说，调用方提供的实参的初始值并不重要。除此之外，输出参数与引用参数类似。输出参数是用 out 修饰符声明的。下面的示例演示 out 参数的用法。

```
using System;
class Test
{
    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }
    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem); // Outputs "3 1"
    }
}

```

参数数组 (parameter array) 允许向方法传递可变数量的实参。参数数组使用 params 修饰符声明。只有方法的最后一个参数才可以是参数数组，并且参数数组的类型必须是一维数组类型。System.Console 类的 Write 和 WriteLine 方法就是参数数组用法的很好示例。它们的声明如下。

```
public class Console
{
    public static void Write(string fmt, params object[] args) {...}
    public static void WriteLine(string fmt, params object[] args) {...}
    ...
}

```

在使用参数数组的方法中，参数数组的行为完全就像常规的数组类型参数。但是，在具有参数数组的方法的调用中，既可以传递参数数组类型的单个实参，也可以传递参数数组的元素类型的任意数目的实参。在后一种情况下，将自动创建一个数组实例，并使用给定的实参对它进行初始化。示例：

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);

```

等价于以下语句：

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);

```

3.1.6 静态方法和实例方法

使用 static 修饰符声明的方法为静态方法 (static method)。静态方法不对特定实例进行操作，并且只能直接访问静态成员。不使用 static 修饰符声明的方法为实例方法 (instance method)。实例方法对特定实例进行操作，并且能够访问静态成员和实例成员。在调用实例方法的实例上，可以通过 this 显式地访问该实例。而在静态方法中引用 this 是错误的。下面的 Entity 类具有静态成员和实例成员。

```
class Entity
{
    static int nextSerialNo;
    int serialNo;
    public Entity() {
        serialNo = nextSerialNo++;
    }
    public int GetSerialNo() {
        return serialNo;
    }
    public static int GetNextSerialNo() {
        return nextSerialNo;
    }
    public static void SetNextSerialNo(int value) {
        nextSerialNo = value;
    }
}

```

每个 Entity 实例都包含一个序号（我们假定这里省略了一些其他信息）。Entity 构造函数（类似于实例方法）使用下一个可用的序号来初始化新的实例。由于该构造函数是一个实例成员，它既可以访问 serialNo 实例字段，也可以访问 nextSerialNo 静态字段。

GetNextSerialNo 和 SetNextSerialNo 静态方法可以访问 nextSerialNo 静态字段，但是如果直接访问 serialNo 实例字段就会产生错误。

下面的示例演示 Entity 类的使用。

```
using System;
class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());                    // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());                    // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());            // Outputs "1002"
    }
}

```

注意：SetNextSerialNo 和 GetNextSerialNo 静态方法是在类上调用的，而 GetSerialNo 实例方法是在该类的实例上调用的。

3.1.7 构造函数

C# 支持两种构造函数：实例构造函数和静态构造函数。实例构造函数 (instance constructor) 是实现初始化类实例所需操作的成员。静态构造函数 (static constructor) 是一种用于在第一次

加载类本身时实现其初始化所需操作的成员。

构造函数的声明如同方法一样，不过它没有返回类型，并且它的名称与其所属的类的名称相同。如果构造函数声明包含 static 修饰符，则它声明了一个静态构造函数。否则，它声明的是一个实例构造函数。实例构造函数可以被重载。例如，List<T> 类声明了两个实例构造函数，一个无参数，另一个接受一个 int 参数。实例构造函数使用 new 运算符进行调用。下面的语句分别使用 List<string> 类的每个构造函数分配两个 List 实例。

```
List<string> list1 = new List<string>();
List<string> list2 = new List<string>(10);

```

实例构造函数不同于其他成员，它是不能被继承的。一个类除了其中实际声明的实例构造函数外，没有其他的实例构造函数。如果没有为某个类提供任何实例构造函数，则将自动提供一个不带参数的空的实例构造函数。

3.1.8 属性

属性 (property) 是字段的自然扩展。属性和字段都是命名的成员，都具有相关的类型，且用于访问字段和属性的语法也相同。然而，与字段不同，属性不表示存储位置。相反，属性有访问器 (accessor)，这些访问器指定在它们的值被读取或写入时需执行的语句。

属性的声明与字段类似，不同的是属性声明以位于定界符 和 之间的一个 get 访问器和/或一个 set 访问器结束，而不是以分号结束。同时具有 get 访问器和 set 访问器的属性是读写属性 (read-write property)，只有 get 访问器的属性是只读属性 (read-only property)，只有 set 访问器的属性是只写属性 (write-only property)。get 访问器相当于一个具有属性类型返回值的无形参方法。除了作为赋值的目标，当在表达式中引用属性时，将调用该属性的 get 访问器以计算该属性的值。set 访问器相当于具有一个名为 value 的参数并且没有返回类型的方法。当某个属性作为赋值的目标被引用，或者作为 ++ 或 – 的操作数被引用时，将调用 set 访问器，并传入提供新值的实参。List<T> 类声明了两个属性 Count 和 Capacity，它们分别是只读属性和读写属性。下面是这些属性的使用示例。

```
List<string> names = new List<string>();
names.Capacity = 100;          // Invokes set accessor
int i = names.Count;               // Invokes get accessor
int j = names.Capacity;        // Invokes get accessor

```

与字段和方法相似，C# 同时支持实例属性和静态属性。静态属性使用 static 修饰符声明，而实例属性的声明不带该修饰符。属性的访问器可以是虚的。当属性声明包括 virtual、abstract 或 override 修饰符时，修饰符应用于该属性的访问器。