第5章 面向对象

本章学习目标

本章主要讲解 C#语言中面向对象程序设计的类定义、类的组成、对象创建、索引、静态成员、静态方法以及参数传递等基本技术。通过对本章的学习，读者应该掌握以下内容:

- 定义类并创建对象
- 引用类型和值类型
- 字段、属性
- 方法
- 索引
- 变量作用域
- 静态变量与静态方法
- 构造函数

5.1 面向对象程序设计概述

在掌握变量、数据类型、表达式及程序控制流程技术后，简单功能的应用程序已可以直接实现，但由于无法真实地反映现实世界而难以表达实际应完成的功能，根据人类认识现实世界的规律，C#语言设计成为面向对象的程序设计语言。

以学生信息管理系统为例，每位学生在入校后，在系统中必须记录其姓名、学号、电话号码、所住寝室、家庭住址、各学科成绩等，虽然可以分别把这些信息记录在数组中，并在程序中保证每一位学生的相关信息在各数组中的序号是相同的，那么需要访问某一位学生的相关信息时，先确定此学生信息在数组中的对应序号，然后再分别访问这些数组中对应序号的元素就可以获取此学生的所有信息，但是，程序设计太复杂，工作量大而且很容易出错。对应现实世界的情况，每位学生能记住自己的学号、姓名等信息，管理系统要求所有学生记录的信息种类和数量基本一致，只是各学生对应的各信息的值不同，这就和 int 这一整型数据类型一样，所有的整型变量都是只能记录整型数的变量也只能记录整型，需要记录整型数据时，只需要创建一个整型变量，然后设定变量的值。如果程序中有对应的“学生”这一数据类型，那么需要在系统中记录一位学生的信息时，只需要创建一个对应“学生”类型的变量，然后设置此学生的对应信息到变量中，同时能确保一位学生对应的姓名、学号、电话号码等信息都保存在此变量中，那么需要此学生对应的信息时，只需要找到此

变量，获取变量中的对应数据值，就能确保获取的信息是此学生的信息而不是其他学生的，程序的编写就可以简化。

但是，实际情况即由于现实世界的物体类型太多而且不固定，就学生信息而言，对于不同的学校，可能在学生信息管理系统中，对学生信息管理的要求也不一样，所以 C#语言无法给出一个对应的“学生”数据类型。为此，C#程序设计语言（包括其他所有的面向对象程序设计语言）给出一种由程序开发者自行设计数据类型的技术和方法，这种可以用于创建变量的复杂的数据类型有一个统一的称呼：类。C#语言中微软是提供了大量的类以方便完成程序开发。

面向对象程序设计中的核心概念包括：类和对象。

简单地说，类是一种复杂的数据类型，它是将不同类型的数据和与这些数据相关的操作封装在一起的集合体。对象则是以类为数据类型声明或创建的变量。

5.2　类的定义和对象的创建

1. 类的定义

类作为复杂的数据类型，内部可以包括：属性、字段、方法、事件、委托以及内部类。

类在使用前，必须先声明，声明类使用关键字 class，语法格式为：

```
属性　访问控制符　class　类名
{
    //类的属性、字段、方法、事件、委托、内部类
}
```

类以及类的成员（包括类的属性、字段、方法、事件、委托以及内部类）还可以通过访问控制符控制其可访问性，有关此部分内容，请参见后续有关面向对象高级应用部分内容，本章全用 public 访问控制符。根据代码规范化的要求及行业规范，类名一般用能代表类实际作用的英文单词组成，每个英文单词的首字母大写，同时，类名还必须符合标识符的命名规则，修饰类的属性一般并不是必须的。

以下示例声明了一个名为 Student 的类，此类被设计用来记录学生信息。

【例 5-1】声明一个用于记录学生信息的类，类名为 Student。

```
public class Student
{
}
```

创建类时，一般一个类的声明放在一个独立的源文件中，只有当两个类有非常密切的关系时才会把多个类的声明放在同一个源文件中。

在 Visual Studio 2008 IDE 中，创建一个类到项目的操作过程为：

（1）右击解决方案资源管理器中项目名称，单击“添加…”→“类…”命令，弹出“添加新项”对话框（如果找不到解决方案资源管理器，可以单击菜单“视图”→“解决方案资源管理器(P)”就可以打开解决方案资源管理器）；

（2）在“添加新项”对话框下方的“名称”文本框中，输入类名，本例为 Student；

（3）单击“添加”按钮，则解决方案资源管理器中新添加了一项，名为 Student.cs，同时，此

源文件也被打开，类声明的基本代码已自动完成，可以直接编写相关代码。

自动创建的 Student 类声明代码，在类的前面没有明确写明访问控制符，可以手动添加 public，也可以先不写，在需要时再添加相应需要的访问控制符。

为了说明类的作用与设计目的，在类的声明代码前，根据规范要求，一般需要添加类作用的注释内容。添加类的注释时，把光标移动到类的前一行，然后输入连续的三个“/”符号，IDE 自动为添加注释的相关格式控制代码，不移动光标，直接添加注释内容，完成的 Student 类代码如下所示。

【例 5-2】添加 Student 类的注释。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ClassDemo   //此命名空间根据项目名称的不同而不同
{
    /// <summary>
    /// Student 是用于管理学生相关信息的类
    /// </summary>
    public class Student
    {
    }
}
```

至止，Student 就成为了一种新的数据类型，可以和 int 等基本数据类型一样用于声明变量，并且所有 Student 类型的变量都有同样的特性。

2. 对象的声明及创建

设计类的目的就是为了根据需要创建一种新的数据类型，最终将用于声明和创建对应满足要求的变量，类创建的变量又称为对象。

和创建简单数据类型的变量一样，复杂数据类型的变量也需要先声明，如以下代码则声明了一个 Student 类型的对象 firstStudent。

```
//声明 Student 类型的对象
Student firstStudent;
```

在声明对象时，可以看到例 5-2 中对类进行的注释的内容，在输入代码过程中，可以看到如图 5-1 所示内容的提示信息。

声明了 firstStudent 对象后，与简单数据类型变量不同的是，类声明的对象还必须先创建，然后才能使用，因为此时 firstStudent 对象还不存在。

创建对象的关键字是 new，语法格式为：

```
new 类名();
```

示例代码如下所示 ：

```
//声明 Student 类型的对象
Student firstStudent;
//创建对象
firstStudent = new Student();
```

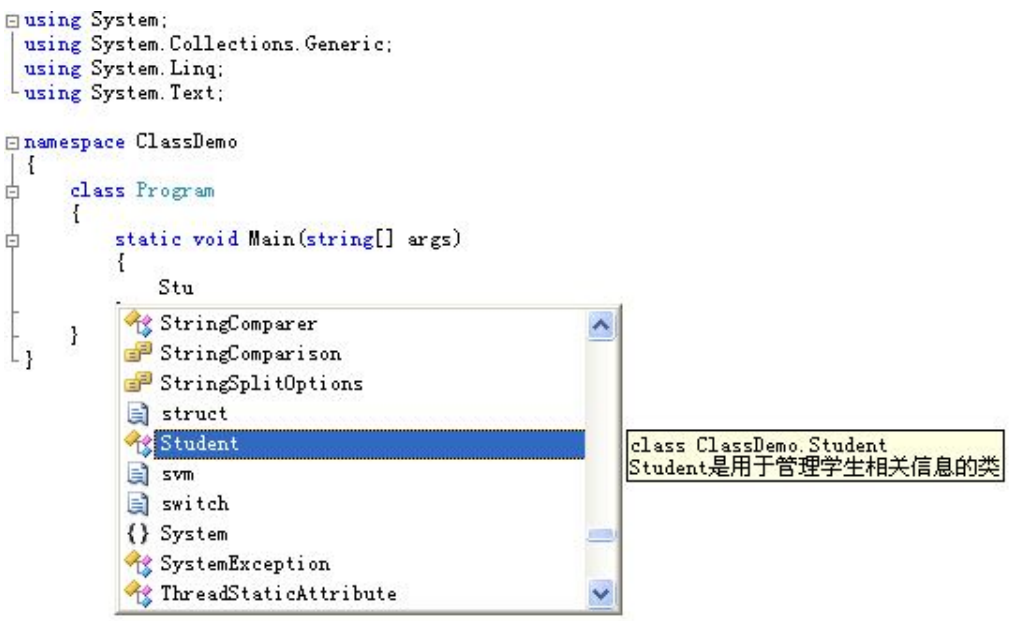

图 5-1　声明变量

5.3　类的字段和属性

在前一节已声明了用于管理学生信息的专用类 Student，但是实际上还无法管理学生的任何信息。为了使类能够管理需要管理的信息，在类的内部常常需要定义类的成员，类的成员包括字段、属性、常量、方法、事件、索引器、构造函数、析构函数以及内部类等。

1. 类的字段声明

字段是在类范围声明的变量。字段是其包含类型的成员，字段可以是内置数值类型或其他类的实例。例如，日历类可能具有一个包含当前日期的字段。

字段的声明语法格式为：

属性　访问控制符　数据类型　字段名

字段声明在类的内部进行。

在前例的 Student 类中，根据本章开始的功能要求，此类必须能记录学生的姓名、学号、电话号码等信息，因此，Student 类必须在内部声明对应的字段，每一信息对应地声明一个不同的字段。

以下示例在例 5-1 的基础上为 Student 类添加了部分字段，用以保存学生的基本信息。

【例 5-3】为 Student 类添加保存学生基本信息的字段。

```
/// <summary>
/// Student 是用于管理学生相关信息的类
/// </summary>
public class Student
{
    /// <summary>
```

```
    /// 姓名
    /// </summary>
    public string Name;
    /// <summary>
    /// 学号
    /// </summary>
    public string ID;
    /// <summary>
    /// 电话号码
    /// </summary>
    public string Phone;
    /// <summary>
    /// 出年日期
    /// </summary>
    public DateTime BirthDay;
}
```

字段名称必须符合标识符的相关规则，建议同时遵循相关的代码规范化要求。

字段声明后，类创建的对象自动都具备了同名的内部变量，但不同对象，其同名字段值可以不一样。

字段的数据类型如果是简单数据类型，则对象创建后，其字段值同时自动设置为对应数据类型的默认值，如果字段数据类型是复杂数据类型（如类），则对象创建后，其字段值自动设置为 null。

2. 类的字段访问

对象创建后，可以通过对象名和对应的字段名实现对相应对象的字段进行访问，以设置字段的值或读取字段的值。

访问对象的属性时，通过“.”运算符实现，语法格式为：

对象名.字段名

当其出现在赋值符号右侧时是读取对应字段值，在赋值符号左侧时是把赋值符号右侧的值保存到对象的字段中。

以下示例在例 5-3 的基础上创建了两个对象，分别设置字段值和读取字段值。

【例 5-4】在主方法中设置字段值和读取字段值。

```
static void Main(string[] args)
 {
    //声明 Student 类型的对象
    Student firstStudent;
    //创建对象
    firstStudent = new Student();
    //设置姓名字段值
    firstStudent.Name = "关羽";
    //设置出生日期
    firstStudent.BirthDay = new DateTime(1990, 1, 2);
    //设置学号
```

```
        firstStudent.ID = "0962101";

        //声明另一 Student 类型对象
        Student anotherStudent;
        //创建对象
        anotherStudent = new Student();
        //设置出生日期
        anotherStudent.BirthDay = new DateTime(1992, 3, 15);
        //设置姓名字段值
        anotherStudent.Name = "张飞";
        //设置学号
        anotherStudent.ID = "0962102";

        //显示 firstStudent 相关字段值
        Console.WriteLine("{0}号学生{1}出生日期是：{2}", firstStudent.ID,
       firstStudent.Name, firstStudent.BirthDay);
        //显示 anotherStudent 相关字段值
        Console.WriteLine("{0}号学生{1}出生日期是：{2}", anotherStudent.ID,
       anotherStudent.Name, anotherStudent.BirthDay);

        //等待用户输入，程序暂停，以查看输出结果
        Console.WriteLine("请按回车键结束程序");
        Console.ReadLine();
    }
```

程序执行结果为：

```
0962101 号学生关羽出生日期是：1990-1-2 0:00:00
0962102 号学生张飞出生日期是：1992-3-15 0:00:00
请按回车键结束程序
```

上例中，创建了两个同一类型的对象：firstStudent 和 anotherStudent，分别对这两个对象的两字段进行了赋值，最后读取并显示了相关的字段值。从此例中可以类比于现实世界的学生，不同的学生个体，其特征不同，比如姓名、出生日期等，但同一学生的信息有了关联，这些信息通过字段被保存在了同一对象内部。

在声明类的字段时，必须注意，只有同一类型的对象都具备的共同特征才声明为类的字段。

在声明类的字段时，请注意添加字段的注释，其方法与类的注释添加方法相同，字段注释添加后，在引用字段时，IDE 会自动显示对应的字段注释内容，方便编写代码，减少代码编写错误。

3. 类的属性声明

类的字段虽然可以实现对象信息存储和管理，但不能很好地满足程序设计的要求，同时不便于程序的修改。

以例 5-4 的 Student 类为例，由于开始设计的疏忽，没有设计学生的身份证号，现根据应用需要，添加学生的身份证号到类中，由于通用的习惯，身份证号一般命名为 ID，但此时，Student 类中已把 ID 这一标识符用于学号，而且在其他地方已使用了此字段，如果需要修改 Student 类中的

“学号”字段的名称，那么同时程序中原来使用过“学号”的所有代码都必须再次修改，实际开发中，由于程序规模已较大，修改很容易出错。此外，如果类中需要存储学生借书证的密码，则只需要在程序中找到对应学生对象，然后读取借书证密码对应字段的值就知道学生的借书证密码，这不安全，所以这些需要保密的字段不能设计访问控制符 public，而应保护起来。

为此，C#一般把字段设计为非 public 类型，并提供“属性”来实现字段的访问。

属性是这样的成员：它们提供灵活的机制来读取、编写或计算私有字段的值。可以像使用公共数据成员一样使用属性，但实际上它们是称作“访问器”的特殊方法。这使得可以轻松访问数据，此外还有助于提高方法的安全性和灵活性。

属性声明的基本语法格式为：

```
访问控制符   属性数据类型   属性名
{
        get { return 字段名; }
        set { 字段名 = value; }
}
```

注意，set 中的 value 变量名不能变。

其中的 get 又称为 get 属性访问器，set 又称为 set 属性访问器。

以例 5-4 中的学号为例，修改原学号字段为：

```
private string studentID;
```

声明对应的属性代码为：

```
/// <summary>
/// 学号属性
/// </summary>
public string StudentID
{
    //读取学号值
    get { return studentID; }
    //设置学号值
    set { studentID = value; }
}
```

属性名一般与对应字段的名称相同，只是字段由于不再是 public 访问控制符，其标识符的首字母一般为小写英文字母。属性的数据类型与对应的字段数据类型相同，然后跟随着一对大括号。属性中的代码分为两部分，其中的 get 访问器为读取属性值时执行的代码，而 set 访问器是设置属性值时将执行的代码。get 访问器，一般必须返回属性对应类型数据的值，而 set 访问器用于设置属性值，所以没有返回值，其中的 value 关键字代表设置所使用的数据值。

在 Visual Studio 2008 中，利用已声明的字段和 IDE 功能，可以通过快捷的方式把字段封装成属性，操作方法是：

（1）右击已声明的属性，弹出快捷菜单。

（2）单击移到快捷菜单的“重构…”项，弹出下一级菜单。

（3）单击“封装字段…”命令，弹出“封装字段”对话框，如图 5-2 所示。

图 5-2　封装字段

（4）在“封装字段”对话框中，系统自动把对应字段名称和建议的属性名填写好，默认的建议属性名为原字段名的首字母改写成大写的名称，单击“确定”按钮则自动完成属性的声明，属性代码包括 get 访问器和 set 访问器的代码，然后再根据需要修改属性中的代码。

在某些特殊情况下，属性中可能只有 get 访问器或 set 访问器两部分中的一部分，则属性不支持没有的那一部分所代表的功能。例如，对于学生的密码，为了保密，不提供程序读取学生密码值的功能，那么属性就只提供 set 访问器，而不提供 get 访问器，对于这种情况，一般仍把空的 get 访问器代码写入，但不返回正确的值，并把此部分代码注销，写明注销原因。其中，不实现 set 访问器的属性是只读的，只能读取不能修改。

密码字段及密码属性代码如下所示：

```
private string password;
public string Password
{
    //禁止获取密码
    //get { return null; }
    //设置密码值
    set { password = value; }
}
```

以下示例在例 5-4 的基础上，修改原有 Student 类，把原有的字段全部设计为非 public 字段，然后添加对应的属性以实现对象相关信息的访问。

【例 5-5】声明属性，以访问原有字段。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ClassDemo
```

```
{
    public class Student
    {
        private string name;
        public string Name
        {
            get { return name; }
            set { name = value; }
        }

        private string studentID;
        public string StudentID
        {
            get { return studentID; }
            set { studentID = value; }
        }

        private string id;
        public string ID
        {
            get { return id; }
            set { id = value; }
        }

        private string phone;
        public string Phone
        {
            get { return phone; }
            set { phone = value; }
        }

        private DateTime birthDay;
        public DateTime BirthDay
        {
            get { return birthDay; }
            set { birthDay = value; }
        }
    }
}
```

注：为节省空间，本章后续代码不再进行详细注释，但请读者在开发过程中务必写上必要的注释，注释是程序中非常重要的组成部分，也是合格的开发者必须完成的工作内容。同时，本章属性的访问控制符不一定是最合适的，但为简单，本章所有属性都使用 public 访问控制符，而字段都将使用 private 访问控制符，相关内容的进一步讨论请参见本书后续相关内容。

4. 类的属性访问

由于字段不再能直接进行访问，但程序仍需要读取和保存相关信息，此时属性的访问将替代原有字段的访问内容。

访问对象的属性方法和访问对象的字段方法一样，通过“.”运算符实现，语法格式为：

对象名.属性名

当对象的属性在赋值符号右侧时，是读取属性的值，即执行属性中的 get 访问器，得到 get 访问器的返回值。

当对象的属性在赋值符号左侧时，是设置属性的值，即执行属性中的 set 访问器，把赋值符右侧的值保存到属性对应的字段中，执行 set 访问器的代码时，value 变量自身则已保存了赋值符号右侧的值。

以下示例在例 5-5 的基础上，修改原有字段的访问代码为属性的访问，通过属性管理学生信息。

【例 5-6】修改主函数，通过属性管理学生信息。

```
static void Main(string[] args)
{
    //声明 Student 类型的对象
    Student firstStudent;
    firstStudent = new Student();
    //通过属性保存学生信息，通过 set 部分代码，学生信息实际保存在对应字段中
    firstStudent.Name = "关羽";
    firstStudent.BirthDay = new DateTime(1990, 1, 2);
    firstStudent.StudentID = "0962101";

    //声明另一 Student 类型对象
    Student anotherStudent;
    anotherStudent = new Student();
    anotherStudent.BirthDay = new DateTime(1992, 3, 15);
    anotherStudent.Name = "张飞";
    anotherStudent.StudentID = "0962102";

    //通过属性读取学生信息，通过 get 部分代码，实际读取对应字段的值
    Console.WriteLine("{0}号学生{1}出生日期是：{2}", firstStudent.StudentID,
                      firstStudent.Name, firstStudent.BirthDay);
    Console.WriteLine("{0}号学生{1}出生日期是：{2}", anotherStudent.StudentID,
                      anotherStudent.Name, anotherStudent.BirthDay);

    Console.WriteLine("请按回车键结束程序");
    Console.ReadLine();
}
```

程序执行结果为：

```
0962101 号学生关羽出生日期是：1990-1-2 0:00:00
0962102 号学生张飞出生日期是：1992-3-15 0:00:00
请按回车键结束程序
```

通过属性来访问对象的信息后，类的代码修改则相对要简单，例如例 5-6 中如果需要修改学生的 name 字段为 studentName，则直接修改 Student 类中的对应字段和 Name 属性中代码部分的对应字段名即可，而主函数中的代码则不需要进行修改。

5.4 索引器

1. 定义索引器

当对象中包括多个同类型的成员时，需要以快捷方便的方式访问指定序号对应的成员，此时可以使用索引器。索引器允许类的实例就像数组一样进行索引。索引器类似于属性，不同之处在于它们的访问器采用参数。

例如在实际情况中，一个班有多名学生，每名学生应对应一个独立的学生对象，而这些学生又应该在同一个集合中，以方便管理。学生所属班级还需要记录包括班级名称等其他信息，所以可以设计一个学生班级类（StudentClass），以管理一个班级内的所有学生对象和班级基本信息，应用索引则可以方便地访问指定序号的学生对象。

以下示例在例 5-6 的基础上，创建一个学生班级类，并在其中创建了索引器，以方便访问指定序号的学生对象。

【例 5-7】定义索引器。

```
/// <summary>
/// StudentClass 是用于管理多个学生的班级类
/// </summary>
public class StudentClass
{
    /// <summary>
    /// 班级内学生存储在 Student 类型的数组中，本例在一个班内最多只能有 35 位学生
    /// </summary>
    private Student[ ] students = new Student[35];

    /// <summary>
    /// 索引器，通过索引器可以方便地访问数组中的学生对象
    /// </summary>
    /// <param name="i">序号</param>
    /// <returns>如果序号有效，则返回对应序号的学生对象；否则返回 null</returns>
    public Student this[int i]
    {
        //get 访问器
        get
        {
            //如果指定序号有效，则返回对应序号的学生对象
            if ((i >= 0) && (i < students.Length))
            {
```

```
                return students[i];
            }
            else//否则返回 null
            {
                return null;
            }
        }
        set //set 访问器
        {
            //如果指定序号有效，则保存学生对象到数组中
            if ((i >= 0) && (i < students.Length))
            {
                students[i] = value;
            }
        }
    }
}
```

通过集合来管理多个同类对象的情况一般都可以设计索引器以访问集合中的对象，例 5-7 中的索引器就可以通过序号访问学生数组中对应序号的学生对象，有效序号的值从 0 到 34。在实际使用中，本例的数组可以用其他集合类代替。

索引器具有以下特点：

（1）使用索引器可以用类似于数组的方式为对象建立索引。

（2）get 访问器返回值。set 访问器分配值。

（3）this 关键字用于定义索引器。

（4）value 关键字用于定义由 set 索引器分配的值。

（5）索引器不必根据整数值进行索引，由用户决定如何定义特定的查找机制。

（6）索引器可被重载。

（7）索引器可以有多个形参，例如当访问二维数组时。

2. 应用索引器

索引器的应用与属性类似，通过 get 和 set 访问器来访问对应的对象，但写法有所不同。索引器的使用一般是通过包含有索引器的对象名称和索引对象的序号组成，如：对象名[索引序号]，此即是对应的对象，当它处于赋值符号左侧时，即调用 set 访问器，把赋值符号右侧的值保存到索引器中，而当它处于赋值符号右侧时，即调用 get 访问器，获得索引到的对象。

以下示例在例 5-7 的基础上，修改主函数代码，通过索引器来管理多个学生对象。

【例 5-8】应用索引器来管理多个学生对象。

```
static void Main(string[] args)
{
    //声明 Student 类型的对象
    Student firstStudent;
    firstStudent = new Student();
    firstStudent.Name = "关羽";
```

```
        firstStudent.BirthDay = new DateTime(1990, 1, 2);
        firstStudent.StudentID = "0962101";

        //声明另一 Student 类型对象
        Student anotherStudent;
        anotherStudent = new Student();
        anotherStudent.BirthDay = new DateTime(1992, 3, 15);
        anotherStudent.Name = "张飞";
        anotherStudent.StudentID = "0962102";

        //声明并创建学生班级对象
        StudentClass firstStudentClass = new StudentClass();
        //通过索引器，把 firstStudent 对象存储到 firstStudentClass 的 students 数组中，
        //其序号为 0
        firstStudentClass[0] = firstStudent;
        //通过索引器，把 anotherStudent 对象存储到 firstStudentClass 的 students 数组中，
        //其序号为 1
        firstStudentClass[1] = anotherStudent;

        Student getedStudent;
        //通过索引器，获取 firstStudentClass 中 students 数组的第 1 号元素，
        //即 anotherStudent
        getedStudent = firstStudentClass[1];
        //显示 getedStudent 相关信息，getedStudent 实际上就是 anotherStudent
        Console.WriteLine("{0}号学生{1}出生日期是：{2}", getedStudent.StudentID,
        getedStudent.Name, getedStudent.BirthDay);
        //通过索引器，获取 firstStudentClass 中 students 数组的第 0 号元素，即 firstStudent
        getedStudent = firstStudentClass[0];
        //显示 getedStudent 相关信息，getedStudent 实际上就是 firstStudent
        Console.WriteLine("{0}号学生{1}出生日期是：{2}", getedStudent.StudentID,
        getedStudent.Name, getedStudent.BirthDay);

        //等待用户输入，程序暂停，以查看输出结果
        Console.WriteLine("请按回车键结束程序");
        Console.ReadLine();
    }
```

程序执行结果为：

```
0962102 号学生张飞出生日期是：1992-3-15 0:00:00
0962101 号学生关羽出生日期是：1990-1-2 0:00:00
请按回车键结束程序
```

5.5　方法定义及调用

1. 定义方法

在实现世界中，每位学生都能有一定的行为能力。在程序中，对应定义的类，如果只能保存字段的数据，那么所有的数据处理都不能方便实现。为了简化数据处理的实现，C#语言中的类提供了处理数据的功能，类的数据处理通过类中的方法来实现。“方法”是包含一系列语句的代码块。作为数据处理的结果，类中的方法有时还可以有返回值，以传递数据处理的结果。

方法是通过指定访问级别（访问控制符）、返回值、方法名称和参数在类或结构中声明的。这些部分统称为方法的“签名”。方法参数括在括号中，每个参数需要指明参数对应的数据类型，并用逗号隔开，参数名必须符合标识符规定。方法参数列表中的参数数量不同、参数数据类型不同、参数的排序顺序不同都被认为是不同的参数列表，但只有参数名称不同而其他相同的参数列表被认为是相同的参数列表。空括号表示方法不需要参数。

类中定义方法的基本语法格式为：

```
属性　方法的访问控制符　返回值类型　方法名(参数列表)
{
//方法体，即方法中处理数据的实现过程；
//如果需要还可以有返回值；
}
```

方法名必须符合 C#语言中标识符的规定，一般由方法所完成功能的英文名称组成，各英文单词的首字母大写，推荐遵循代码规范化要求设计方法名。方法定义的位置与类中的字段、属性和索引器一样，在类的内部。

方法可以向调用方返回值。如果返回类型（方法名称前列出的类型）不是 void，则方法可以使用 return 关键字来返回值。如果语句中 return 关键字的后面是与返回类型匹配的值，则该语句将该值返回给方法调用方。return 关键字还会停止方法的执行。如果返回类型为 void，则可使用没有值的 return 语句来停止方法的执行。如果没有 return 关键字，方法执行到代码块末尾时即会停止。具有非 void 返回类型的方法才能使用 return 关键字返回值。若要使用从方法返回的值，调用方法可以在本来使用同一类型的值就已足够的任何位置使用方法调用本身，还可以将返回值赋给变量。

以下示例在例 5-8 的基础上，为 Student 类定义了 ShowMessage 方法以提供每个学生对象显示自身基本信息的行为能力。

【例 5-9】定义类中的方法，以实现学生对象显示自身基本信息。

```
public class Student
{
    #region 字段和属性
    //例 5-8 中原有字段和属性
    #endregion
```

```
    #region 方法
    /// <summary>
    /// 显示学生基本信息
    /// </summary>
    public void ShowMessage()
    {
        Console.WriteLine("{0}的基本信息：", name);
        Console.WriteLine("学号：{0}", studentID);
        Console.WriteLine("身份证号：{0}", id);
        Console.WriteLine("出生日期：{0}", birthDay);
        Console.WriteLine("电话号码：{0}", phone);
        Console.WriteLine("————————————————");
    }
    #endregion
}
```

其中 ShowMessage 就是显示学生基本信息的方法，此方法没有返回值，参数列表为空。

由于类中的代码数量较多，为了方便程序编码，可以使用 C#语言中的预处理器指令#region 和#endregion，#region 可以在使用 Visual Studio 代码编辑器的大纲显示功能时指定可展开或折叠的代码块，但是#region 和#endregion 需要成对使用。#region 后接英文空格和代码块的描述，单击#region 左侧的“-”标志可以把代码块折叠，并显示为灰色的代码块描述内容；再次单击折叠后的代码块左侧的“+”标志可以把代码块展开；鼠标移动到已折叠的代码块描述上方时，将自动显示代码块的前面部分。

在例 5-9 的代码中把字段和属性代码块折叠后如图 5-3 所示。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ClassDemo
{
    /// <summary>
    /// Student是用于管理学生相关信息的类
    /// </summary>
    public class Student
    {
        字段和属性

        #region 方法
        /// <summary>
        /// 显示学生基本信息
        /// </summary>
        public void ShowMessage()
        {
            Console.WriteLine("{0}的基本信息：", name);
            Console.WriteLine("学号：{0}", studentID);
            Console.WriteLine("身份证号：{0}", id);
            Console.WriteLine("出生日期：{0}", birthDay);
            Console.WriteLine("电话号码：{0}", phone);
            Console.WriteLine("————————————————");
        }
        #endregion
    }
}
```

图 5-3　#region 预处理指令

以下示例在例 5-9 的基础上在 StudentClass 类中添加了显示指定学号的学生基本信息的方法。

【例 5-10】定义 StudentClass 类中的方法，以显示指定学号的学生对象基本信息。

```
public class StudentClass
{
    //例 5-9 中原有代码

    /// <summary>
    /// 显示指定学号的学生基本信息，如果没有找到指定学生则什么也不做
    /// </summary>
    /// <param name="studentId">学号</param>
    public void ShowStudentMessageByStudentId(string studentId)
    {
        //遍历学生数组，查找指定学号对应的学生
        foreach (Student curStudent in students)
        {
            //如果当前学生对象不为空则判断当前对象是否为要找的对象
            if (curStudent != null)
            {
                //如果当前学生为需要找的学生
                if (curStudent.StudentID.Trim().CompareTo(studentId.Trim()) == 0)
                {
                    //显示学生基本信息
                    Console.WriteLine("{0}的基本信息：", curStudent.Name);
                    Console.WriteLine("学号：{0}", curStudent.StudentID);
                    Console.WriteLine("身份证号：{0}", curStudent.ID);
                    Console.WriteLine("出生日期：{0}", curStudent.BirthDay);
                    Console.WriteLine("电话号码：{0}", curStudent.Phone);
                    Console.WriteLine("————————————");
                    //结束遍历操作，跳出 foreach 代码块
                    break;
                }
            }
        }
    }
}
```

方法定义后需要添加相应注释，添加注释时，光标定位到方法声明的前一行，连续输入三个“/”字符，IDE 根据方法的声明内容，自动添加了相应的必须注释内容格式符，根据注释中的相应内容进行相关的说明性注释。在例 5-10 中，<summary>部分的内容是说明方法功能描述，“<param name="studentId">学号</param>”中的 param 是指方法的参数，其中 studentId 是指针对参数名为 studentId 的参数的说明。此外，还有返回值<return>等部分注释，将在以后的代码中举例说明。

以下示例在例 5-10 的基础上在 StudentClass 类中添加了查找指定学号的学生对象的方法。

【例 5-11】定义 StudentClass 类中的方法，以查找指定学号的学生对象。

```
public class StudentClass
{
//例 5-10 中原有代码

    /// <summary>
    /// 查找指定学号的学生
    /// </summary>
    /// <param name="studentId">学号</param>
    /// <returns>如果找到对应学号的学生，则返回找到对象；否则返回 null</returns>
    public Student FindStudentByStudentId(string studentId)
    {
        //遍历学生数组，查找指定学号对应学生
        foreach (Student curStudent in students)
        {
            //如果当前学生对象不为空则判断当前对象是否为要找的对象
            if (curStudent != null)
            {
                //如果当前学生为需要找的学生
                if (curStudent.StudentID.Trim().CompareTo(studentId.Trim()) == 0)
                {
                    //通过 return 语句结束方法执行
                    return curStudent;
                }
            }
        }

        //如果遍历完还没有找到对应的学号，则没有要找的学生对象，返回 null
        return null;
    }
}
```

例 5-11 中，方法指定返回值类型为 Student，则在方法返回时，必须指定返回值，返回值只能是两种类型，一种为 Student 类型对象，另一种则为 null。

2. 调用方法

在例 5-10 改成例 5-11 的过程中，可以发现方法 ShowStudentMessageByStudentId 中包括有 FindStudentByStudentId 方法的全部代码，为了提高程序的可维护性，相同的代码在程序中应该只出现一次，即在方法 ShowStudentMessageByStudentId 中调用 FindStudentByStudentId 以利用 FindStudentByStudentId 已实现的功能。

在 C#语言中，应用已完成的代码是通过调用对应对象的方法实现的，调用方法时，通过“.”运算符实现，其语法格式为：

对象.方法名(参数列表)

其中的参数列表根据方法定义时的要求，传递对应数量和数据类型的参数值，此参数称为“实参”，实参可以是数值，也可以是对象。

在调用方法时，实参与形参的结合不是按参数名称进行传递的，而是按参数列表从左到右一一结合，所有对应实参和形参必须是对应数据类型。

以下示例在例 5-11 的基础上在 StudentClass 类中调用 FindStudentByStudentId 方法，实现代码重购，修改主函数，以显示指定学号学生的基本信息。

【例 5-12】修改 StudentClass 类中的代码，重构代码，并显示指定学号学生信息。

在例 5-11 基础上修改 StudentClass 类中的 ShowStudentMessageByStudentId 方法体为如下所示，完成重构。

```
public void ShowStudentMessageByStudentId(string studentId)
{
    //查找指定学号的学生
    Student foundStudent = FindStudentByStudentId(studentId);

    //如果找到学生，则显示学生信息
    if (foundStudent != null)
    {
        //调用显示学生对象的基本信息方法来显示其信息
        foundStudent.ShowMessage();
    }
    else
    {
        Console.WriteLine("没有对应学号的学生");
    }
}
```

在例 5-11 基础上修改主函数代码，以要求显示指定学号的学生信息。

```
static void Main(string[] args)
{
    //声明 Student 类型的对象
    Student firstStudent;
    firstStudent = new Student();
    firstStudent.Name = "关羽";
    firstStudent.BirthDay = new DateTime(1990, 1, 2);
    firstStudent.StudentID = "0962101";

    //声明另一 Student 类型对象
    Student anotherStudent;
    anotherStudent = new Student();
    anotherStudent.BirthDay = new DateTime(1992, 3, 15);
    anotherStudent.Name = "张飞";
    anotherStudent.StudentID = "0962102";

    //声明并创建学生班级对象
```

```
        StudentClass firstStudentClass = new StudentClass();
        firstStudentClass[0] = firstStudent;
        firstStudentClass[1] = anotherStudent;

        //调用 firstStudentClass 对象的方法显示指定学号学生的信息
        //显示学号为 0962101 的学生的基本信息
        firstStudentClass.ShowStudentMessageByStudentId("0962101");
        //显示学号为 0962102 的学生的基本信息
        firstStudentClass.ShowStudentMessageByStudentId("0962102");

        //等待用户输入，程序暂停，以查看输出结果
        Console.WriteLine("请按回车键结束程序");
        Console.ReadLine();
    }
```

程序执行结果为：

```
关羽的基本信息：
学号：0962101
身份证号：
出生日期：1990-1-2 0:00:00
电话号码：
————————————————
张飞的基本信息：
学号：0962102
身份证号：
出生日期：1992-3-15 0:00:00
电话号码：
————————————————
请按回车键结束程序
```

通过方法定义和调用，方法所实现的功能可以在只编写一次代码的情况下，多次被调用，程序开发工作量更小，但却更好维护，因为在需要修改方法实现功能时，只需要修改方法体中的代码就可以。

5.6 值类型与引用类型

1. 值类型与引用类型

C#的数据类型，在进行赋值和传递时，可以分为两种：值类型和引用类型。

值类型变量直接包含其数据，值类型包括内置值类型、用户定义的值类型以及枚举类型，C#语言中的基本数据类型都是值类型。值类型在进行赋值时，仅仅把变量的值复制后设置到被赋值变量中，两变量之间不再有联系。

以下示例代码中，修改 y 变量的值将不会影响 x 变量的值。

```
int x;
int y = 10;
x = y;       //变量的值被赋值给了变量 x
Console.WriteLine("x 的值是：{0}，y 的值是：{1}", x, y);
y = 20;      //修改 y 变量的值，但不会影响 x 变量的值
Console.WriteLine("x 的值是：{0}，  y 的值是：{1}", x, y);
```

程序执行结果为：

```
x 的值是：10，y 的值是：10
x 的值是：10，y 的值是：20
```

引用类型变量包含对其数据的引用，引用类型变量赋值的是对象的引用，而不是复制对象的值，赋值后，两个变量对应的对象是同一个对象。引用类型包括接口、类以及数组。

以下示例代码是在例 5-12 的基础上编写的代码片段，修改 secondStudent 变量的 Name 属性后，firstStudent 变量的 Name 属性值也自动和 secondStudent 变量的 Name 属性的新值相等。

```
Student firstStudent = new Student();
firstStudent.Name = "张飞";
firstStudent.StudentID = "0962102";

//secondStudent 和 firstStudent 引用同一对象
Student secondStudent = firstStudent;
Console.WriteLine("修改 firstStudent 变量的 Name 属性前：");
Console.WriteLine("{0}号学生是：{1}",firstStudent.StudentID,firstStudent.Name);
Console.WriteLine("{0}号学生是：{1}", secondStudent.StudentID,
          secondStudent.Name);

//修改 firstStudent 的 Name 属性也同时修改了 secondStudent 的 Name 属性值
firstStudent.Name = "关羽";
Console.WriteLine("修改 firstStudent 变量的 Name 属性后：");
Console.WriteLine("{0}号学生是：{1}",firstStudent.StudentID,firstStudent.Name);
Console.WriteLine("{0}号学生是：{1}", secondStudent.StudentID,
          secondStudent.Name);
```

程序执行结果为：

```
修改 firstStudent 变量的 Name 属性前：
0962102 号学生是：张飞
0962102 号学生是：张飞
修改 firstStudent 变量的 Name 属性后：
0962102 号学生是：关羽
0962102 号学生是：关羽
```

但是，在修改变量所指向的对象后，变量之间可能不再有关系。

在上例的基础上，以下代码展示了修改 firstStudent 变量引用的对象后，再次修改 firstStudent 变量的属性值不再影响 secondStudent 变量的对应属性。

```
Student firstStudent = new Student();
firstStudent.Name = "张飞";
```

```
firstStudent.StudentID = "0962102";

//secondStudent 和 firstStudent 引用同一对象
Student secondStudent = firstStudent;
Console.WriteLine("修改 firstStudent 变量的 Name 属性前：");
Console.WriteLine("{0}号学生是：{1}", firstStudent.StudentID, firstStudent.Name);
Console.WriteLine("{0}号学生是：{1}", secondStudent.StudentID,
          secondStudent.Name);

//修改 firstStudent 的 Name 属性也同时修改了 secondStudent 的 Name 属性值
firstStudent.Name = "关羽";
Console.WriteLine("修改 firstStudent 变量的 Name 属性后：");
Console.WriteLine("{0}号学生是：{1}", firstStudent.StudentID, firstStudent.Name);
Console.WriteLine("{0}号学生是：{1}", secondStudent.StudentID,
          secondStudent.Name);

//修改 firstStudent 变量引用的对象
firstStudent = new Student();
firstStudent.Name = "赵子龙";
firstStudent.StudentID = "0962103";

Console.WriteLine("修改 firstStudent 变量引用的对象后：");
Console.WriteLine("{0}号学生是：{1}", firstStudent.StudentID, firstStudent.Name);
Console.WriteLine("{0}号学生是：{1}", secondStudent.StudentID,
          secondStudent.Name);
```

程序执行结果为：

```
修改 firstStudent 变量的 Name 属性前：
0962102 号学生是：张飞
0962102 号学生是：张飞
修改 firstStudent 变量的 Name 属性后：
0962102 号学生是：关羽
0962102 号学生是：关羽
修改 firstStudent 变量引用的对象后：
0962103 号学生是：赵子龙
0962102 号学生是：关羽
```

2. 装箱与拆箱

object 类型在 .NET Framework 中是 Object 的别名。在 C# 的统一类型系统中，所有类型（预定义类型、用户定义类型、引用类型和值类型）都是直接或间接从 Object 继承的。可以将任何类型的值赋给 object 类型的变量。在应用过程中，简单数据类型也可以转化为 object 类型，转化过程称为装箱；反之，从装箱以后的变量中提取出值类型的过程则称为拆箱。

以下代码把 valueType 变量装箱后放到 referenceType 变量中：

```
int valueType = 0;
```

```
object referenceType = valueType;   //装箱
```

以下代码则是把上例中已装箱的对象进行拆箱操作：

```
int unBoxing = (int)referenceType;  //拆箱，按 int 类型进行数据提取
```

需要特别指出的是，拆箱操作中，如果装箱过程的原数据类型与拆箱过程预期的数据类型不兼容，在编译过程没有语法错误，但运行时将抛出异常，如下例所示：

```
string valueType = "abc";
object referenceType = valueType;   //装箱，原始数据不能转化为 int 类型
int unBoxing = (int)referenceType;  //拆箱，按 int 类型进行数据提取，运行时将抛出异常
```

此外，由于装箱和拆箱比较耗费 CPU 资源，所以尽可能避免进行装箱与拆箱操作，在程序开发过程中，需要注意变量的数据类型。

5.6　参数的传递

方法在调用时，实参将把值赋给形参，这个过程称为实参与形参的结合。在赋值过程中，根据变量是值类型还是引用类型，分为按值传递和按引用传递。

1. 按值传递

向方法传递值类型变量意味着向方法传递变量的一个副本。在方法内发生的对参数的更改对该变量中存储的原始数据无任何影响。

以下示例通过值传递值类型参数。通过值传递将变量 n 传递给方法 SquareIt。方法内发生的任何更改对变量的原始值无任何影响。

【例 5-13】按值传递方式调用方法。

```
class PassingValByVal
{
     /// <summary>
     /// 按值传递方法方式传递值到参数 x，方法对 x 值的修改不影响实参的值
     /// </summary>
     /// <param name="x">按值传递的参数</param>
    static void SquareIt(int x)         //static 的用法请参见 5.9 节静态成员
    {
        x *= x;
        System.Console.WriteLine("方法内的 x 变量值为：{0}",  x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("调用方法前 n 的值为：{0}",  n);

        SquareIt(n);  //按值传递方式调用方法
        System.Console.WriteLine("调用方法后 n 的值为：{0}",  n);
    }
}
```

程序执行结果为：

```
调用方法前 n 的值为：5
方法内的 x 变量值为：25
调用方法后 n 的值为：5
```

2. 按引用传递

当通过值传递引用类型的参数时，有可能更改引用所指向的数据，如某类成员的值。但是无法更改引用本身的值。也就是说，不能使用相同的引用为新类分配内存并使之在块外保持。

以下示例通过传递引用类型参数。通过引用将数组 arr 传递给方法 Change。方法内对数组 pArray 中的元素进行的修改同样影响方法外的 arr 数组。

【例 5-14】按值传递方式调用方法。

```
class PassingValByRef
{
    /// <summary>
    /// 按引用方式传递参数，方法内对数组元素的修改将影响外部的原始数组
    /// </summary>
    /// <param name="pArray">数组为引用类型</param>
    static void Change(int[] pArray)
    {
        pArray[0] = 888;  //修改元素值将影响原始数据值
        Console.WriteLine("方法内 0 号元素值为：{0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = { 1, 4, 5 };
        Console.WriteLine("调用方法前数组 0 号元素的值为：{0}", arr[0]);
        Change(arr);
        Console.WriteLine("调用方法后数组 0 号元素的值为：{0}", arr[0]);
    }
}
```

程序执行结果为：

```
调用方法前数组 0 号元素的值为：1
方法内 0 号元素值为：888
调用方法后数组 0 号元素的值为：888
```

但对于方法内部，如果修改变量本身引用的对象，则方法内的引用变量的修改不会影响方法外的原始变量的所有数据。

以下示例通过传递引用类型参数。通过引用将数组 arr 传递给方法 ChangeArray。方法内对数组 pArray 自身进行修改，则 pArray 中元素的修改不会影响方法外的 arr 数组。

【例 5-15】按值传递方式调用方法，修改方法内的变量引用的对象。

```
class PassingValByRef
{
    /// <summary>
```

```
    /// 按引用方式传递参数，方法内对数组变量引用其他对象，则修改数组元素不会影响原始数组
    /// </summary>
    /// <param name="pArray"></param>
    static void ChangeArray(int[] pArray)
    {
        Console.WriteLine("修改 pArray 引用的对象前，pArray[0]的值为：{0}",
            pArray[0].ToString());
        pArray = new int[3];
        pArray[0] = 888;
        Console.WriteLine("修改 pArray 引用对象后，pArray[0]的值为：{0}",
        pArray[0].ToString());
    }

    static void Main()
    {
        int[] arr = { 1, 4, 5 };
        Console.WriteLine("调用方法前数组 0 号元素的值为：{0}", arr[0]);
        ChangeArray(arr);
        Console.WriteLine("调用方法后数组 0 号元素的值为：{0}", arr[0]);
    }
}
```

程序执行结果为：

```
调用方法前数组 0 号元素的值为：1
修改 pArray 引用的对象前，pArray[0]的值为：1
修改 pArray 引用对象后，pArray[0]的值为：888
调用方法后数组 0 号元素的值为：1
```

3. 使用 ref 传递参数

在调用方法时，如果需要实现在方法内修改值类型变量后能自动影响原始变量值，或者需要在方法内修改变量引用的对象后，仍使方法外的变量也自动引用方法体内的新对象，则可以明确地使用 ref 关键字声明方法。

使用 ref 关键字时，方法签名中的参数列表中，需要在使用 ref 方式传递的参数前，加上 ref 关键字；在调用方法时，对应实参前也加上 ref 关键字即可。

以下示例在例 5-13 的基础上，使用 ref 关键字，把值类型的参数传递方式改变成按引用方式传递参数。通过值将变量 n 传递给方法 SquareItRef。方法内发生的更改将对变量的原始值产生影响。

【例 5-16】按值传递方式调用方法。

```
class PassingValByVal
{
    /// <summary>
    /// 应用 ref 关键字，值类型传递的参数 x，方法内对 x 值的修改将影响实参的值
    /// </summary>
    /// <param name="x">按值传递的参数</param>
    static void SquareItRef(ref int x)
```

```
    {
        x *= x;
        System.Console.WriteLine("方法内的 x 变量值为：{0}", x);
    }
    static void Main()
    {
        int n = 5;
        Console.WriteLine("调用方法前 n 的值为：{0}", n);
        SquareItRef(ref n);  //按值传递方式调用方法
        Console.WriteLine("调用方法后 n 的值为：{0}", n);

        //等待用户输入，程序暂停，以查看输出结果
        Console.WriteLine("请按回车键结束程序");
        Console.ReadLine();
    }
}
```

程序执行结果为：

```
调用方法前 n 的值为：5
方法内的 x 变量值为：25
调用方法后 n 的值为：25
请按回车键结束程序
```

以下示例在例 5-15 的基础上，在引用类型参数前使用 ref 关键字，则方法内设计形参引用新的对象后，原始参数也将引用新的对象。

【例 5-17】引用类型参数使用 ref 关键字。

```
class PassingValByRef
{
    /// <summary>
    /// 对引用类型参数使用 ref 关键字，方法内对数组变量引用其他对象，则原始数组也引用新对象
    /// </summary>
    /// <param name="pArray"></param>
    static void ChangeArrayByRef(ref int[] pArray)
    {
        Console.WriteLine("修改 pArray 引用的对象前，pArray[0]的值为：{0}",
            pArray[0].ToString());
        pArray = new int[3];
        pArray[0] = 888;
        Console.WriteLine("修改 pArray 引用对象后，pArray[0]的值为：{0}",
            pArray[0].ToString());
    }

    static void Main()
    {
        int[] arr = { 1, 4, 5 };
```

```
        Console.WriteLine("调用方法前数组 0 号元素的值为：{0}", arr[0]);
        ChangeArrayByRef(ref arr);
        Console.WriteLine("调用方法后数组 0 号元素的值为：{0}", arr[0]);
    }
}
```

程序执行结果为：

```
调用方法前数组 0 号元素的值为：1
修改 pArray 引用的对象前，pArray[0]的值为：1
修改 pArray 引用对象后，pArray[0]的值为：888
调用方法后数组 0 号元素的值为：888
```

4. 使用 out 传递参数

在某些情况下，参数在调用方法前无法确定对象，而是在访问体中创建新的对象，此时可以使用 out 关键字。out 关键字的使用方法及应用效果与 ref 基本一致，但 ref 的参数在使用前需要初始化，而 out 关键字对应的参数可以不初始化。

以下示例在例 5-16 的基础上，在值类型参数前使用 out 关键字，则方法内对参数值的修改影响原始参数的值。

【例 5-18】值类型参数使用 out 关键字。

```
class PassingValByVal
{
    /// <summary>
    /// 使用 out 关键字，值类型传递的参数 x，方法内对 x 值的修改将影响实参的值
    /// </summary>
    /// <param name="x">out 类型的值参数</param>
    static void SquareIntOut(out int x)
    {
        x = 10;    //out 关键字使用的参数在函数内部必须赋值
        System.Console.WriteLine("方法内的 x 变量值为：{0}", x);
    }

    static void Main()
    {
        int n ;     //out 关键字使用的参数可以不初始化
       // Console.WriteLine("调用方法前 n 的值为：{0}", n);

        SquareIntOut(out n);  //按值传递方式调用方法
        Console.WriteLine("调用方法后 n 的值为：{0}", n);
    }
}
```

程序执行结果为：

```
方法内的 x 变量值为：10
调用方法后 n 的值为：10
```

以下示例在例 5-17 的基础上，在引用类型参数前使用 out 关键字，则在调用方法前实参可以

不初始化，方法内设置形参引用新的对象后，原始参数也将引用新的对象。

【例 5-19】引用类型参数使用 ref 关键字。

```
class PassingValByRef
{
    /// <summary>
    /// 对引用类型参数使用 out 关键字，方法内对数组变量进行初始化，则原始数组也引用新对象
    /// </summary>
    /// <param name="pArray">未初始化的参数</param>
    static void ChangeArrayByOut(out int[] pArray)
    {
        pArray = new int[3];
        pArray[0] = 888;
        Console.WriteLine("pArray 初始化后，pArray[0]的值为：{0}",
            pArray[0].ToString());
    }

    static void Main()
    {
        int[] arr = null;
        ChangeArrayByOut(out arr);
        Console.WriteLine("调用方法后数组 0 号元素的值为：{0}", arr[0]);
    }
}
```

程序执行结果为：

```
pArray 初始化后，pArray[0]的值为：888
调用方法后数组 0 号元素的值为：888
```

out 关键字除了用于需要使方法体内对变量的修改影响原始变量的情况外，在被调用方法内有多个数据或结果需要传递回调用函数时，除了其中之一可以通过返回值的形式返回，还可以用 out 关键字来输出数据。

5.7 变量的作用域

在上一节中，需要设法把被调用方法内对形参值的修改结果传递回调用方法内，是由于在调用方法内不能直接访问到形参变量，因为变量的作用域有一定的范围。

程序元素的作用域是指可以引用该程序元素的代码区域，对于类、接口以及类内的方法、字段、属性等的作用域将在后续访问控制符的相关内容中进一步讲解，本节主要讲解变量的作用域。

C#中变量的作用域主要分：代码块变量、方法内部变量以及类的字段。

1. 代码块变量

代码块变量即是指变量只在一个代码块内有效。

C#语言的代码块一般是指以一对“{”和“}”为范围的代码集合，例如 for、foreach 等语句结

构后面的代码集合都可以称为代码块。

一个代码块中声明的变量只能在本代码块内部有效，代码块外部则不能访问代码块级变量。

以下代码中的 for 循环的循环变量 i，在循环结束的“}”后不再能被访问。

```
int sum = 0;
//此处以前不能访问变量 i，因为 i 还未声明
for (int i = 0; i < 5; i++)
{
    //代码块内 i 可以被访问
    sum += i;
}
//此后变量 i 不能访问，因为 i 是代码块级变量
```

以下代码中的两个变量 i 都可以正常被访问，并且不存在同名变量重复声明的错误，因为它们分别处于不同的代码段。

```
{
  int i = 10;
Console.WriteLine("第一个代码块内声明的变量，值为：{0}", i.ToString());
    {
        //以下变量 i 将引起语法错误，因为本代码块被包含在外侧的代码块中，
        //所以此处已有同名变量 i
        //int i = 30;
    }

}
{
    int i = 10;
    Console.WriteLine("第二个代码块内声明的变量，值为：{0}", i.ToString());
}
//此后两个变量 i 都不能被访问
```

2. 方法内变量

方法内变量就是在方法内部声明的变量，此类变量的代码域为从声明变量开始到方法返回时结束，最具代表性的就是方法的形参。就某种意义而言，方法也可以被看作为一个代码块。

方法内变量的示例可参见上一节中有关参数传递的示例。

3. 字段

类内声明的字段，所在类的范围内都可以访问字段，有时，还包括其他范围，相关技术将在介绍访问控制符的内容中进一步讨论。

类的字段名称可以和方法内变量或者代码块变量同名，但是，当在方法内变量或代码块中的变量与字段同名的范围内，直接引用变量名只能访问到方法内变量或代码块内变量。如果需要访问字段，则需要加上关键字“this”并使用“.”运算符，以形成以“this.字段名”为整体的标识，以特指当前对象自身的字段。

以下示例说明了方法内变量或代码块变量与字段同名时的运行情况。

【例 5-20】方法参数或代码块变量与字段同名。

```
class MathClass
{
    /// <summary>
    /// 字段变量
    /// </summary>
    private int i = 10;

    /// <summary>
    /// 方法参数与字段同名
    /// </summary>
    /// <param name="i">参数</param>
    public void DemoFunction(int i)
    {
        //当方法参数或者代码块变量与字段同名时，
        //方法参数或者代码块变量屏蔽了字段

        //显示两个同名变量的值
        Console.WriteLine("方法的参数 i 的值为：{0}", i.ToString());
        Console.WriteLine("字段 i 的值为：{0}", this.i.ToString());
        Console.WriteLine("————————————");

        //修改方法参数的值
        i = 20;
        Console.WriteLine("修改参数 i 的值为 20");
        Console.WriteLine("方法的参数 i 的值为：{0}", i.ToString());
        Console.WriteLine("字段 i 的值为：{0}", this.i.ToString());
        Console.WriteLine("————————————");

        //修改字段的值
        this.i = 30;
        Console.WriteLine("修改字段 i 的值为 30");
        Console.WriteLine("方法的参数 i 的值为：{0}", i.ToString());
        Console.WriteLine("字段 i 的值为：{0}", this.i.ToString());
        Console.WriteLine("————————————");
    }
}
```

在主函数中添加代码：

```
MathClass m = new MathClass();
m.DemoFunction(40);
```

程序执行结果为：

```
方法的参数 i 的值为：40
字段 i 的值为：10
————————————
```

```
修改参数 i 的值为 20
方法的参数 i 的值为：20
字段 i 的值为：10
————————————————————
修改字段 i 的值为 30
方法的参数 i 的值为：20
字段 i 的值为：30
————————————————————
```

5.8　构造函数

1. 构造函数

每个类都显示或隐式地包含一个构造函数。构造函数是一种特殊的方法，该方法用来实现对象的初始化。当使用 new 关键字创建对象时调用构造函数，构造函数通常用来为字段赋值。构造函数需要通过使用与其所属类相同的名称来定义。构造函数在类实例化时由 CLR 自动调用。构造函数没有返回值。

构造函数的一般语法格式为：

```
访问控制符　构造函数名(参数列表)
{
    //构造函数的方法体
}
```

以下示例是在例 5-9 的基础上，为 Student 类创建构造函数。

【例 5-21】创建 Student 类的构造函数。

```
public class Student
{
    #region 构造函数
    /// <summary>
    /// 无参构造函数
    /// </summary>
    public Student()
    {
        Console.WriteLine("创建一个学生对象");
    }
    #endregion
    //例 5-9 原有代码
}
```

修改原有主函数，删除原有代码，添加以下代码：

```
static void Main(string[] args)
{
    Student firstStudent = new Student();
```

```
        //等待用户输入，程序暂停，以查看输出结果
        Console.WriteLine("请按回车键结束程序");
        Console.ReadLine();
    }
```

程序执行结果为：

```
创建一个学生对象
请按回车键结束程序
```

程序在创建 firstStudent 对象时，调用了 Student 类的构造函数。

2. 构造函数重载

一个类的构造函数可以有多个，这是构造函数的重载。

当定义两种或多种具有相同名称的方法时，就称作重载。例如，在 System.Char 中，IsDigit 被重载。一种方法使用了一个 Char 并返回一个 Boolean。另一种方法使用了一个 String 和一个 Int32，并返回一个 Boolean。参数列表也可以通过 varargs 约束来限定，此约束指示方法支持一个变量参数列表。重载的进一步讨论在后续内容中专门进行。

由于在一些特殊情况下会自动调用类的默认构造函数，所以除非特殊情况要求，否则，在明确提供了一个非默认构造函数后，同时明确提供一个默认构造函数。如果一定需要禁止默认构造函数，则一般把默认构造函数编码后，再注释掉此构造函数，并通过注释说明注释掉此构造函数的原因。

以下示例在例 5-21 的基础上添加了重载 Student 类的构造函数。

【例 5-22】重载 Student 类的构造函数。

修改 Student 类代码为：

```
public class Student
{
    #region 构造函数
    /// <summary>
    /// 无参构造函数，又称为默认构造函数
    /// </summary>
    public Student()
    {
        Console.WriteLine("使用默认构造函数创建一个学生对象");
    }

    /// <summary>
    /// 带参数的构造函数
    /// </summary>
    /// <param name="studentId">学号</param>
    /// <param name="name">姓名</param>
    public Student(string studentId, string name)
    {
        this.studentID = studentId;
        this.name = name;
```

```
        Console.WriteLine("使用带两个参数的构造函数创建一个学生对象");
    }

    /// <summary>
    /// 带参数的构造函数
    /// </summary>
    /// <param name="studentId">学号</param>
    /// <param name="name">姓名</param>
    /// <param name="birthday">出生日期</param>
    public Student(string studentId, string name, DateTime birthday)
    {
        this.studentID = studentId;
        this.name = name;
        this.birthDay = birthday;
        Console.WriteLine("使用带三个参数的构造函数创建一个学生对象");
    }
    #endregion
}
```

修改主函数代码为：

```
static void Main(string[] args)
{
    //自动调用默认构造函数
    Student firstStudent = new Student();
    firstStudent.ShowMessage();
    //调用对应参数列表的构造函数
    Student secondStudent = new Student("0962101", "关羽",
                    new DateTime(1990, 1, 2));
    secondStudent.ShowMessage();

    //等待用户输入，程序暂停，以查看输出结果
    Console.WriteLine("请按回车键结束程序");
    Console.ReadLine();
}
```

程序执行结果为：

```
使用默认构造函数创建一个学生对象
的基本信息：
学号：
身份证号：
出生日期：0001-1-1 0:00:00
电话号码：
————————————————
使用带三个参数的构造函数创建一个学生对象
关羽的基本信息：
```

```
学号：0962101
身份证号：
出生日期：1990-1-2 0:00:00
电话号码：
————————————————

请按回车键结束程序
```

3. 指定初始值设定项

当具有多个构造函数时，这些构造函数初始化的方式常常很相似，如果在每个构造函数中都编写这些相似的代码，则违背了相同代码只编写一次的原则。因此可以将共同的代码集中放在一个构造函数中，然后在其他的构造函数中调用此构造函数。在例 5-22 中，对于调用默认构造函数创建的 Student 对象，由于没有进行明确的字段初始化，显示的信息为空，调用带三个参数的构造函数创建的 Student 对象，其身份证号也为空，因为对象创建时，会自动为其简单数据类型的字段设置为对应类型的默认值，而复杂数据类型的字段则自动设置为 null。为此，可以应用这一原则，为创建的对象设置重要的字段提供信息，但在多个构造函数中都需要这些初始化代码。

通过使用关键字 this，可以调用类中定义的一个特定构造函数。使用方法是把 this 关键字添加到构造函数声明中，将调用对应参数列表匹配的构造函数。

以下示例在例 5-22 的基础上添加实现 Student 类的构造函数调用其他构造函数。

【例 5-23】Student 类的构造函数调用本类的其他构造函数。

在例 5-22 基础上修改 Student 类代码为如下所示：

```
Public class Student
{
    // 例 5-22 中原有代码

    #region 构造函数
    /// <summary>
    /// 无参构造函数，又称为默认构造函数
    /// </summary>
    Public Student()
    {
        Console.writeline("使用默认构造函数创建一个学生对象");
        This.studentid = "未指定学号";
        This.name = "未指定姓名";
        This.phone = "未指定电话号码";
        This.id = "未指定身份证信息";
        This.birthday = new datetime(1990, 1, 1);
    }

    /// <summary>
    /// 带两个参数的构造函数
    /// </summary>
    /// <param name="studentid">学号</param>
    /// <param name="name">姓名</param>
```

```
    Public Student(string studentid, string name) : this()
    {
        This.studentid = studentid;
        This.name = name;
        Console.writeline("使用带两个参数的构造函数创建一个学生对象");
    }

    /// <summary>
    /// 带三个参数的构造函数
    /// </summary>
    /// <param name="studentid">学号</param>
    /// <param name="name">姓名</param>
    /// <param name="birthday">出生日期</param>
    Public Student(string studentid, string name, datetime birthday) :
                    this(studentid, name)
    {
        This.birthday = birthday;
        Console.writeline("使用带三个参数的构造函数创建一个学生对象");
    }
    #endregion
}
```

主函数保持不变。

程序执行结果为：

```
使用默认构造函数创建一个学生对象
未指定姓名的基本信息：
学号：未指定学号
身份证号：未指定身份证信息
出生日期：1990-1-1 0:00:00
电话号码：未指定电话号码
————————————————
使用默认构造函数创建一个学生对象
使用带两个参数的构造函数创建一个学生对象
使用带三个参数的构造函数创建一个学生对象
关羽的基本信息：
学号：0962101
身份证号：未指定身份证信息
出生日期：1990-1-2 0:00:00
电话号码：未指定电话号码
————————————————
```

程序执行结果的显示信息展示了构造函数的调用过程及执行顺序。

4. readonly 修饰符

在有些特殊情况下，对象中字段的值被赋值后就不允许再被修改，对应的字段需要使用修饰符

readonly，具体赋值过程可以在声明中赋值，也可以在构造函数中进行赋值。

此类字段之所以不用常量，是因为这些字段的值在编写代码和编译程序时不能确定其值，而是在类或对象初始化时才能确定。

以下示例在例 5-23 的基础上，修改 Student 类中的身份证号字段和属性，使其值只能在对象被创建时才能进行赋值，其他地方不能进行赋值。

【例 5-24】修改 Student 类身份证号字段和属生，使用 readonly 修饰符。

```
public class Student
{
    // 例 5-22 中原有代码
    /// <summary>
    /// 身份证号
    /// </summary>
    private readonly string id;
    /// <summary>
    /// 身份证号
    /// </summary>
    public string ID
    {
        get { return id; }
        //set { id = value; }
    }
}
```

代码中，由于 id 字段使用了 readonly 修饰符，所以只能在构造函数中进行赋值。其属性 ID 中也不能进行赋值操作，所以必须把 set 访问器删除。

5.9 静态成员

在本章前面的所有技术及示例中，字段、属性、索引器及方法的调用都必须通过“对象名”来实现，这样的类成员又被称为实例成员，即它们都属于某个具体的对象（即类的实例）。在某些情况下，同一类的所有对象需要共享数据，此时实例成员不能满足要求，为此设计了类的静态成员。

静态成员属于类，而不属于实例，静态构造函数还可以初始化类。静态成员是与类相联系的概念，当需要初始化或提供由类的所有实例共享的数值时，使用静态成员很有用。

由于静态成员属于类而不属于实例，所以它们都是通过类而不是通过类的实例（对象）来访问的。

1. 静态字段

静态字段的声明格式与一般字段声明的语法格式一样，只是其修饰符为 static。

静态字段一般通过类来访问，不可以通过实例来访问。

以下示例在例 5-23 的基础上，添加了一个静态字段 CollegeName，用于记录全校学生共享的校名。

【例 5-25】修改 Student 类，添加静态字段。

```
public class Student
{
    //例 5-23 原有代码

    /// <summary>
    /// 学校名称
    /// </summary>
    static public string CollegeName;

    /// <summary>
    /// 显示学生基本信息
    /// </summary>
    public void ShowMessage()
    {
        Console.WriteLine("{0}的基本信息：", name);
        //通过类的实例访问类的静态字段
        //相关于 this.CollegeName
        Console.WriteLine("就读于：{0}", CollegeName);
        Console.WriteLine("学号：{0}", studentID);
        Console.WriteLine("身份证号：{0}", id);
        Console.WriteLine("出生日期：{0}", birthDay);
        Console.WriteLine("电话号码：{0}", phone);
        Console.WriteLine("————————————");
    }
}
```

主函数修改为：

```
static void Main(string[] args)
{
    //在创建任何 Student 类型的实例前，可直接通过类设置其静态字段值
    Student.CollegeName = "卡耐基梅隆大学";

    //自动调用默认构造函数
    Student firstStudent = new Student();
    firstStudent.ShowMessage();

    //调用对应参数列表的构造函数
    Student secondStudent = new Student("0962101", "关羽",
                            new DateTime(1990, 1, 2));
    secondStudent.ShowMessage();

    //等待用户输入，程序暂停，以查看输出结果
    Console.WriteLine("请按回车键结束程序");
```

```
    Console.ReadLine();
}
```

程序执行结果为：

```
未指定姓名的基本信息：
就读于：卡耐基梅隆大学
学号：未指定学号
身份证号：未指定身份证信息
出生日期：1990-1-1 0:00:00
电话号码：未指定电话号码
————————————
关羽的基本信息：
就读于：卡耐基梅隆大学
学号：0962101
身份证号：未指定身份证信息
出生日期：1990-1-2 0:00:00
电话号码：未指定电话号码
————————————
请按回车键结束程序
```

2. 静态属性

正如实例字段可以对应设计属性一样，静态字段也可以对应设计静态属性，但静态属性必须是访问的静态字段的值。

静态属性一般通过类来访问，不可以通过实例来访问。

以下示例在例 5-25 的基础上，修改原有静态字段 CollegeName，并添加对象的静态属性。

【例 5-26】修改 Student 类，添加静态属性。

```
public class Student
{
    //例 5-23 原有代码

    /// <summary>
    /// 学校名称
    /// </summary>
    static private string collegeName;
    /// <summary>
    /// 学校名称
    /// </summary>
    static public string CollegeName
    {
        get { return collegeName; }
        set { collegeName = value; }
    }
}
```

代码其余部分不变。

3. 静态方法

类中的方法也可以是静态的，静态方法只能通过类来调用，而不能通过实例来调用。

静态方法的声明和定义与一般方法声明和定义的语法格式一样，只是其修饰符为 static。使用静态修饰符的方法一般是全局的，当成员引用或操作的信息是关于类的而不是类的实例时，这个成员就应该设置成静态成员。

静态方法中，只能访问类中的静态字段或静态属性等类共享的信息，而不能访问实例数据。

以下示例在例 5-26 的基础上，添加静态方法用以获取全校学生数量。

【例 5-27】修改 Student 类，添加静态方法用以获取全校学生数量。

原有 Student 类代码修改为如下所示：

```
public class Student
{
    //例 5-26 中原有代码
    public Student()
    {
        this.studentID = "未指定学号";
        this.name = "未指定姓名";
        this.phone = "未指定电话号码";
        this.id = "未指定身份证信息";
        this.birthDay = new DateTime(1990, 1, 1);

        //每创建一个学生对象时，学生数量加 1
        //由于 Student 所有的构造函数都会调用默认构造函数，此处能统计出正确数量
        studentCount++;
    }
    /// <summary>
    /// 全校学生数量
    /// </summary>
    static private int studentCount = 0;
    /// <summary>
    /// 获取全校学生数量
    /// </summary>
    /// <returns>全校学生数量</returns>
    static public int GetStudentCount ()
    {
        return studentCount;
    }
}
```

主函数修改为：

```
static void Main(string[] args)
{
    //在创建任何 Student 类型的实例前，可直接通过类设置其静态字段值
    Student.CollegeName = "卡耐基梅隆大学";
```

```
        //自动调用默认构造函数
        Student firstStudent = new Student();
        firstStudent.ShowMessage();

        //调用对应参数列表的构造函数
        Student secondStudent = new Student("0962101", "关羽",
                                    new DateTime(1990, 1, 2));
        secondStudent.ShowMessage();

        Console.WriteLine("全校共有学生：{0}", Student.GetStudentCount().ToString());

        //等待用户输入，程序暂停，以查看输出结果
        Console.WriteLine("请按回车键结束程序");
        Console.ReadLine();
    }
```

程序执行结果为：

```
未指定姓名的基本信息：
就读于：卡耐基梅隆大学
学号：未指定学号
身份证号：未指定身份证信息
出生日期：1990-1-1 0:00:00
电话号码：未指定电话号码
————————————————

关羽的基本信息：
就读于：卡耐基梅隆大学
学号：0962101
身份证号：未指定身份证信息
出生日期：1990-1-2 0:00:00
电话号码：未指定电话号码
————————————————

全校共有学生：2
请按回车键结束程序
```

4. 静态构造函数

通过构造函数可以初始化类的对象，也可以设计初始化类本身的构造函数，此类构造函数称为静态构造函数，静态构造函数使用 static 修饰符，不能有访问控制符。

静态构造函数不对类的特定实例进行操作，也称为全局构造函数。

静态构造函数不能直接调用，静态构造函数会在第一个实例创建和静态方法被调用之前自动执行，并且只执行一次，因此静态构造函数适合于对类的所有实例都用到的数据进行初始化。

静态构造函数与静态方法一样，只能访问静态成员。

以下示例在例 5-27 的基础上，添加静态构造函数。

【例 5-28】修改 Student 类，添加静态构造函数。

```
public class Student
{
    //例 5-27 原有代码
    /// <summary>
    /// 静态构造函数，不能有访问控制符
    /// </summary>
    static Student()
    {
        studentCount = 0;
        Personality = "训练有素";
        Console.WriteLine("静态构造函数被调用");
    }
    public Student()
    {
        this.studentID = "未指定学号";
        this.name = "未指定姓名";
        this.phone = "未指定电话号码";
        this.id = "未指定身份证信息";
        this.birthDay = new DateTime(1990, 1, 1);
        studentCount++;
        Console.WriteLine("默认构造函数被调用");
    }
    public Student(string studentId, string name) : this()
    {
        this.studentID = studentId;
        this.name = name;
        Console.WriteLine("带两个参数的构造函数被调用");
    }
    public Student(string studentId, string name, DateTime birthday) :
this(studentId, name)
    {
        this.birthDay = birthday;
        Console.WriteLine("带三个参数的构造函数被调用");
    }
    static private int studentCount;
    /// <summary>
    /// 全校学生共有特质
    /// </summary>
    static public string Personality;
}
```

程序执行结果为：

```
静态构造函数被调用
默认构造函数被调用
```

```
未指定姓名的基本信息：
就读于：卡耐基梅隆大学
学号：未指定学号
身份证号：未指定身份证信息
出生日期：1990-1-1 0:00:00
电话号码：未指定电话号码
————————————
默认构造函数被调用
带两个参数的构造函数被调用
带三个参数的构造函数被调用
关羽的基本信息：
就读于：卡耐基梅隆大学
学号：0962101
身份证号：未指定身份证信息
出生日期：1990-1-2 0:00:00
电话号码：未指定电话号码
————————————
全校共有学生：2
```

注意静态构造函数的自动调用时机。

5.10 内部类和匿名类

1. 内部类

在某些情况下，少数类只在一些类的内部需要，为了简化程序的管理，降低程序复杂性，这些类被声明于类的内部，称为内部类。内部类的声明语法格式与一般类相同。

以下示例声明了一个内部类。

【例 5-29】声明内部类。

```
/// <summary>
/// 外部类
/// </summary>
public class OuterClass
{
    /// <summary>
    /// 内部类声明的字段
    /// </summary>
    InnerClass innerField;
    public OuterClass()
    {
        //初始化
        innerField = new InnerClass();
    }
    public void ShowMessage()
    {
```

```
        Console.WriteLine("这是在外部类中显示提示信息");

        innerField.ShowMessage();
    }
    /// <summary>
    /// 内部类
    /// </summary>
    class InnerClass
    {
        public void ShowMessage()
        {
            Console.WriteLine("这是在内部类中显示提示信息");
        }
    }
}
```

2. 匿名类

匿名类提供了一种方便的方法，可用来将一组只读属性封装到单个对象中，而无须首先显式定义一个类型。类型名由编译器生成，并且不能在源代码级使用。这些属性的类型由编译器推断。

以下示例展示一个用两个分别名为 Amount 和 Message 的属性初始化的匿名类。

【例 5-30】使用匿名类。

```
var v = new { Amount = 108, Message = "Hello" };
Console.WriteLine("v.Amount = {0}, v.Message = {1}", v.Amount, v.Message);
```

程序执行结果为：

```
v.Amount = 108, v.Message = Hello
```

由于内部类和匿名类的应用并不常见，有关内部类和匿名类的详细使用请读者参见相关资料。

本章小结

本章讲解了面向对象（Object-Oriented，OO）的概念，类、对象以及类和对象的使用，字段、属性、索引器、构造函数等面向对象的基本概念，为后续章节的学习打下坚实的基础。

面向对象是一种有效的软件开发方法，它将数据和对数据的操作作为一个互相依赖、不可分割的整体，它符合人们的思维习惯，同时有助于控制软件的复杂性，提高软件的开发效率。因此，面向对象得到了广泛的应用，已成为目前最流行的一种软件开发方法。现在的主流开发语言都会涉及面向对象的概念，类和对象的概念适用于大部分主流开发语言，值得认真掌握。

习题

1．填空题

（1）静态构造函数在_____前被自动调用，一共执行_____次。

（2）readonly 修饰符修饰的字段只能在_____或_____时进行赋值。

（3）静态成员的修饰符是_____。

2．程序设计题

（1）卡耐基梅隆大学对学生的训练异常严格，课业繁重，在普林斯顿评论每年“学生累得像狗的大学排名”中，一直高居前几位。根据这一特点，在例 5-28 的基础上，为学生编写一描述性成员，用于说明所有学生的共同特点：累得像狗；并在显示学生基本信息时，显示此特点。

（2）设计一个简单的银行客户账号管理系统。首先创建一个控制台应用程序，项目名称为 BankAccountPrj，然后设计银行客户账号类（BankAccount），账号类需要保存账号的类型、账号号码、客户姓名、客户身份证号、客户余额等信息。客户账号的类型只能是活期账户、定期账户、支票账户三种中的某一种（使用枚举类型），用账号可以进行检查账户余额、往账户中加钱、从账户中取钱（如果账户资金充足）三种操作。

案例完善——客户信息管理系统中面向对象的客户信息管理

【实训任务和目标】

本章实训在上一章完成多客户信息管理的基础上，应用面向对象技术来管理多客户信息，实现功能包括添加客户账户、查找客户账户、编辑客户账户、删除客户账户以及显示所有客户账户信息功能。通过本实训，主要掌握类的定义、对象创建及使用等技能。

【案例分析】

在上一章完成的银行客户信息管理系统中，虽然能够管理多名客户账户的信息，但是由于同一客户账户的信息被分散到多个数组中，每次操作一位客户的相关信息时，都必须先确定客户信息所在数组中的序号，然后再分别操作多个数组对应序号元素的值，实现方式烦琐。同时，所有的代码都写在了一个方法内，此方法实现的功能太多，不可能完成真实的客户信息管理系统中复杂要求，所以本章的解决方案更新为使用面向对象技术，把整个系统的代码分散到不同的类和方法中，把大问题分解成多个小问题，以降低问题的难度。

由于仍需要管理多位客户的账户信息，所以本章仍需要使用数组技术，但数组中的元素是对应的客户类类型，本章仍设定最多能管理 50 位客户的账户信息，在后续技术讲解应用过程中修改本章项目解决方案，以管理任意数量客户的账户信息。

系统采用 C#语言来编写，客户账户信息与前面章节对应的信息采用相同的数据类型和相关要求。

【案例实现】

启动 Visual Studio 2008，打开本章的起始项目解决方案 CMS.sln。案例实现的步骤及要求如下：

（1）打开 Program.cs 文件，注释掉原有的 Main 方法中的所有代码。

（2）在解决方案管理器中，添加客户账户对应的类 Customer，并且在类中添加用于管理客户

账号的字段 accountNumber，添加用于管理客户账户类型的字段 accountCategory，此字段的数据类型为前面章节中定义的 CUSTOMERCATEGORY 枚举类型，添加管理客户姓名的字段 name，添加管理客户账户余额的字段 balance，添加管理客户地址的字段 address，添加管理客户电话的字段 phone。balance 字段访问控制符设定为 protected，其余所有的字段都设定访问控制符为 private。

（3）在类 Customer 中，为前一步骤添加的字段除 balance 外，全部添加对应的属性。

（4）在类 Customer 中，添加静态字段 currentAccountNumber，用于管理新创建的客户账号应该设置的账号值，在每一位客户账户创建后，此值需要加 1。

（5）添加类 Customer 的静态构造函数，在静态构造函数中，设置 currentAccountNumber 的值为 500000，以控制所有客户的账号大于等于 500000。

（6）添加类 Customer 的默认构造函数，在默认构造函数中，设置当前对象的账号为当前客户应有的账号值，并使 currentAccountNumber 值加 1，得到下一客户账号值。

（7）在类 Customer 中添加所有客户信息的输入功能方法，对用户输入数据进行必要的有效性检验，发现是无效数据后，需要用户再次输入。

（8）在解决方案管理器中，添加类 Manager，用此类实现原项目中 Main 方法的流程控制。

（9）在类 Manager 中添加 Customer 类型的数组 Customers，用于管理所有客户对象。

（10）在类 Manager 中添加 int 类型的常量 MAXCUSTOMERCOUNT，用于控制程序最多能管理客户的数量，本项目设置其值为 50。

（11）在类 Manager 中添加 int 类型的字段 currentCustomerIndex，用于记录下一个新创建的客户对象保存到数组中的元素序号，此字段的值初始化为 0。

（12）在类 Manager 中添加 void 类型方法 MainFunction，此方法实现原 Main 方法中的控制流程，把原 Main 方法中的控制流程代码复制过来，去掉原有的添加、查找、删除、显示客户信息的具体代码，仅保留流程控制代码。

（13）在类 Manager 中分别添加根据客户账号、姓名、电话号码等信息在数组 Customers 中查找对应客户账户对象的方法，所有方法返回值类型都为 Customer。

（14）在 Main 方法中，创建一个 Manager 类型的对象，再调用此对象的 MainFunction，完成客户账户信息的管理。

（15）在 Manager 类中，如果删除一个客户对象，在数组 Customers 中找到对应元素后，设置此元素的值为 null，以确保此对象不再存在系统中。

【参考代码】

请参考“源代码\实训源代码\CMS_Project05\5_Solution”中源代码。