第3章

chapter 3

对 象 和 类

到目前为止，已经介绍了组成C#语言的主要内容——变量的声明、数据类型和程序流语句，并简要介绍了一个只包含Main()方法的完整小例子。但还没有介绍如何把这些内容组合在一起，构成一个完整的程序，其关键就在于对类的处理。本章介绍了如何使用C#中的各个类，其重点是如何定义方法、构造函数、属性和单个类(或单个结构)中的其他成员。所有的类最终都派生于System.Object类。另外，本章还说明了如何创建继承类的层次结构，简要讨论了C#对继承的支持，然后详细论述了如何在C#中编码实现继承和接口继承。本章将重点阐述用于提供继承的语法和与继承相关的主题。

本章技能学习要点。

技能1：认识类
技能2：认识类成员
技能3：掌握方法成员
技能4：掌握属性成员
技能5：掌握构造函数
技能6：了解终结器
技能7：掌握索引器
技能8：掌握委托
技能9：掌握事件
技能10：掌握运算符重载
技能11：只读字段的使用
技能12：认识结构
技能13：认识部分类
技能14：静态类
技能15：Object类的使用
技能16：对象的相等比较
技能17：用户定义的数据类型转换
技能18：认识继承
技能19：领悟派生类的构造函数
技能20：掌握修饰符的作用
技能21：认识接口

项目一：MathTest 项目

下面的项目 MathTest 说明了类的定义和实例化、方法的定义和调用的语法。除了包含 Main()方法的类之外，它还定义了类 MathTest，该类包含两个方法和一个字段。

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

namespace ForTest
{
    class MainEntryPoint
    {
        static void Main()
        {
            //试图调用静态方法
            Console.WriteLine("Pi is" + MathTest.GetPi());
            int x = MathTest.GetSquareOf(5);
            Console.WriteLine("Square of 5 is" + x);
            //实例化一个对象
            MathTest Testmath = new MathTest();
            //这是 C# 的一种调用引用类型
            //调用非静态成员
            Testmath.value = 30;
            Console.WriteLine("Value field of math variable contains" + Testmath.value);
            Console.WriteLine("Square of 30 is" + Testmath.GetSquare());
            Console.ReadLine();
        }
    }
    //定义一个 MathTest 新类，用来调用方法
    class MathTest
    {
        public int value;
        public int GetSquare()
        {
            return value * value;
        }
        public static int GetSquareOf (int x)
        {
            return x * x;
        }
        public static double GetPi()
        {
                return 3.14159;
        }
    }
}
```

运行 MathTest 示例，会得到如图 3-1 所示结果。

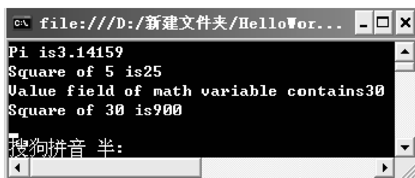

图 3-1

从代码中可以看出，MathTest 类包含一个字段和一个方法，该字段包含一个数字，该方法计算数字的平方。这个类还包含两个静态方法，一个返回 pi 的值，另一个计算传入参数的平方。

这个类有一些功能并不是 C＃程序设计的好例子。例如，GetPi()通常作为 const 字段来执行，而好的设计应使用目前还没有介绍的概念。

在面向对象程序设计中，类的设计非常重要，是软件开发中的关键一步，如果设计得当，既有利于程序进行扩充，又可以提高代码的重用性。

技能 1：认识类

从计算机语言的角度来说，类是一种数据类型，而对象是具有这种类型的变量。其实类和结构实际上都是创建对象的模板，每个对象都包含数据，并提供了处理和访问数据的方法。类定义了每个类对象(称为实例)可以被换成什么数据和功能。

1. 类的声明

类的声明语法如下：

```
[类的修饰符] class 类名 [:基类名]
{
    //类的成员;
}
```

其中，class 是声明类的关键字，“类名”必须是合法的 C＃标识符。“类的修饰符”有多个，如表 3-1 所示，有关内容后面再叙述。

表 3-1

类的修饰符	描 述
public	公有类。表示对该类的访问不受限制
protected	保护类。表示只能从所在类和所在类派生的子类进行访问
internal	内部类。只有其所在类才能访问
private	私有类。只有该类才能访问
abstract	抽象类。表示该类是一个不完整的类，不允许建立类的实例
sealed	密封类。不允许从该类派生新的类

例如，如果一本书一个顾客，可以定义字段 CustomerID、FristName、LastName 和 Address，以包含该顾客的信息。还可以定义处理存储在这些字段中的数据的功能。接着就可以实例化这个类的对象，以表示某个顾客，并为这个实例设置这些字段，使用其功能。

```
class PhoneCustomer
{
      public const string DayOfSendingBill = "Monday";
      public int CustomerID;
      public string FristName;
      public string LastName;
}
```

又如定义一个 Person 类，如下代码所示。

```
public class Person
{
    public int pno;            //编号
    public string pname;       //姓名
    public void setdata(int no,string name)
    {
          pno = no; pname = name;
    }
    public void dispdata()
    {
          Console.WriteLine("{0} {1}", pno, pname);
    }
}
```

2. 定义类的对象

类和对象是不同的概念。类定义对象的类型，是对象的模板，但它不是对象的本身。对象是基于类的具体实体，只有定义类的对象才会给对象分配相应的空间。语法如下：

```
类名 对象名;
对象名 = new 类名();
```

或者可以合成一句：

```
类名 对象名 = new 类名();
```

例如：

```
PhoneCustomer myCustomer;
myCustomer = new PhoneCustomer();
```

或者合成一句：

```
PhoneCustomer myCustomer = new PhoneCustomer();
```

通常将对象引用和对象实例合用，但读者应了解它们的区别。语句中 PhoneCustomer() 部分是创建类的实例，然后传递回对该对象的引用并赋给 myCustomer，这样就可以通过对象引用 myCustomer 操作该对象了。两个对象引用可以指向同一个对象。例如：

```
Person p1 = new Person();
Person p2 = p1;
```

技能 2：认识类成员

类中的数据和函数称为类的成员。Microsoft 的正式术语对数据成员和函数成员进行

了区分。除了这些成员外，类还可以包含嵌套的类型(例如其他类)。类中的所有成员都可以声明为 public(此时可以在类的外部直接访问它们)或 private(此时，它们只能由类中的其他代码来访问)，protected(表示成员所在的类及其派生类访问)，后面将详细解释各种访问级别。

1. 数据成员

数据成员包含类的数据——字段、常量和事件。数据成员可以是静态数据(与整个类相关)或实例数据(类的每个实例都有它直接的数据副本)。通常，对于面向对象的语言，类成员总是实例成员，除非用 static 进行了显式的声明。

字段是与类相关的变量。在前面的例子中已经使用了 PhoneCustomer 类中的字段：一旦实例化 PhoneCustomer 对象，就可以使用语法 Object. FieldName 来访问这些字段如下所示。

```
PhoneCustomer Customer1 = new PhoneCustomer( );
Customer1. FristName = "Simon";
```

常量与类的关联方式同变量与类的关联方式一样。使用 const 关键字来声明常量。如果它们声明为 public，就可以在类的外部访问。

```
class PhoneCustomer
{
        public const string DayOfSendingBill = "Monday";
        public int CustomerID;
        public string FristName;
        public string LastName;
}
```

事件是类的成员，在发生某些行为(例如改变类的字段或属性，或者进行了某些形式的用户交互操作)时，它可以让对象通知调用程序。客户可以包含称为“事件处理程序”的代码来响应该事件。

2. 函数成员

函数成员提供了操作类中数据的某些功能，包括方法、属性、构造函数和终结器(finalizer)、运算符以及索引器。

方法是与某个类相关的函数，它们可以是实例方法，也可以是静态方法。实例方法处理类的某些实例，静态方法提供了更一般的功能，不需要实例化一个类(例如 Console. WriteLine()方法)。

属性是可以在客户机上访问的函数组，其访问方式与访问类的公共字段类似。C＃为读写类上的属性提供了专用语法，所以不必使用那些名称中嵌有 Get 或 Set 的偷工减料的方法。因为属性的这种语法不同于一般函数的语法，在客户代码中，虚拟的对象被当作实际的东西。

构造函数是在实例化对象时自动调用的函数。它们必须与所属类同名，且不能有返回类型。构造函数用于初始化字段的值。

终结器类似于构造函数，但是在 CLR 检测到不再需要某个对象时调用。它们的名称与类相同，但前面有一个～符号。

运算符执行的最简单的操作就是＋和－。在对两个整数进行相加操作时，严格地说，就是对整数使用＋运算符。C＃还允许指定把已有的运算符应用于自己的类(运算符重载)。

索引器允许对象以数组或集合的方法进行索引。

技能 3：掌握方法成员

在 C♯中，每个函数都必须与类或结构相关。正式的 C♯术语实际上并没有区分函数的方法。在这个术语中，“函数成员”不仅包含方法，而且也包含类或结构的一些非数据成员，包括索引器、运算符、构造函数和析构函数等，甚至还有属性，这些都不是数据成员。

1. 方法的声明

C♯中每个方法都单独声明为 public 或 private。另外，所有的 C♯方法都在类定义中声明和定义。方法的定义包括方法的修饰符（例如方法的可访问性）、返回值的类型，然后是方法名、输入参数的列表（用圆括号括起来）和方法体（用花括号括起来）。

```
[修饰符] 返回值类型 方法名([参数列表])
{
    //方法体
}
```

每个参数都包括参数的类型名及在方法体中的引用名称。但如果方法有返回值，return 语句必须与返回值一致，以指定出口点，例如：

```
public bool IsSquare (Rectangle rect)
{
    return (rect.Height == rect.Width);
}
```

这段代码使用一个表示矩形的.NET 基类 System.Drawing.Rectangle。

如果方法没有返回值，就把返回类型指定为 void，因为不能省略返回类型。如果方法不带参数，仍需要在方法名的后面写上一对空的圆括号( )(就像本章前面的 Main( )方法)。此时 return 语句就是可选的——当到达花括号时，方法会自动返回。注意方法可以包含任意多个 return 语句。

```
public bool IsSquare(int value)
{
    if (value < 0)
          return false;
        else
          return true;
}
```

2. 调用方法

访问方法与访问数据成员一样，都是通过“.”运算符，该运算符的功能是表示对象的成员。具体语法如下：

```
对象名.数据成员;
对象名.方法名(参数表);
```

调用数据成员，当数据成员访问修饰符为 public 时才可以在类外访问，否则只能在类内访问。方法成员一样。

示例：

这是一个控制台应用程序。

```
using System;
namespace Test
{   public class TPoint                        //声明类 TPoint
    {     int x,y;                             //类的私有字段
          public void setpoint(int x1,int y1)
          {   x = x1;y = y1;   }
          public void dispoint()
          {    Console.WriteLine("({0},{1})",x,y);   }
    }
    class Program
    {   static void Main(string[] args)
          {    TPoint p1  =  new TPoint();   //定义对象 p1
               p1.setpoint(2,6);
               Console.Write("第一个点 =>");
               p1.dispoint();
               TPoint p2  =  new TPoint();   //定义对象 p2
               p2.setpoint(8,3);
               Console.Write("第二个点 =>");
               p2.dispoint();
          }
    }
}
```

运行以上程序效果如图 3-2 所示。

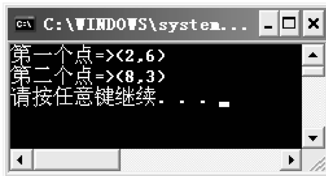

图　3-2

3. 给方法传递参数

参数可以通过引用传递给方法。在变量通过引用传递给方法时，被调用的方法得到的就是这个变量，所以在方法内部对变量进行的任何改变在方法退出后仍旧发挥作用。而如果变量是通过值传递给方法的，被调用的方法得到的是变量的一个副本，也就是说，在方法退出后，对变量进行的修改会丢失。对于复杂的数据类型，按引用传递的效率更高，因为在按值传递时，必须复制大量的数据。

所有的参数都是通过值传递的，除非特别说明。但是，在理解引用类型的传递过程时需要注意。因为引用类型的对象只包含对象的引用，它们只给方法传递这个引用，而不是对象本身，使用对底部对象的修改会保留下来。相反，值类型的对象包含的是实际数据，所以传递给方法的是数据本身的副本。例如，int 通过值传递给方法，方法对该 int 值所做的任何改变都没有改变原 int 对象的值。但如果数据或其他引用类型（如类）传递给方法后，方法会使用该引用改变这个数组中的值，而新值会反射到原来的数组对象上。

示例：

```
using System;
namespace parameterTestSample
{
    class ParameterTest
    {
```

```
        static void SomeFunction ( int[] ints, int i )
            {
                ints[0] = 100;
                i = 100;
            }
            public static void Main()
            {
                int i = 0;
                int[] ints = {0,1,2,4,8};
                //显示原始数据
                Console.WriteLine ("i = " + i);
                Console.WriteLine ("ints [0] = " + ints [0]);
                Console.WriteLine ("Calling SomeFunction");
                //函数返回值之后, ints 发生改变,但 i 没有改变
                SomeFunction (ints, i);
                Console.WriteLine ("i = " + i);
                Console.WriteLine ("ints [0] = " +  ints [0]);
                Console.ReadKey();
        }
    }
}
```

结果如图 3-3 所示。

图 3-3

注意：i 的值保持不变，而在 int 中改变的值在原来的数组中也改变了。字符串是不同的，因为字符串是不能改变的(如果改变字符串的值，就会创建一个全新的字符串)，所以字符串是无法显示一般引用类型的行为方式。在方法调用中，对字符串的任何改变都不会影响原来的字符串。

4. ref 参数

通过值传送变量是默认的，也可以迫使值参数通过引用传递给方法。为此，要使用 ref 关键字。如果把一个参数传递给方法，且这个方法的输入参数带有 ref 关键字，则该方法对变量所做的任何改变都会影响原来对象的值。

```
static void SomeFunction(int[] ints, ref int i)
{
    ints [0] = 100;
    i =100; //i 的变化将会持续,直到 SomeFunction()退出
}
```

在调用该方法时，还需要添加 ref 关键字：

```
SomeFunction ( ints, ref i);
```

最后，C＃仍要求对传递给方法的参数进行初始化，理解这一点也是非常重要的。在传递给方法之前，无论是按值传递，还是按引用传递，都必须初始化变量。

5. out 关键字

在 C 风格的语言中，函数能从一个例程中输出多个值，这种情况很常见。这是使用输

出参数实现的——方法是把输出值赋给通过引用传递给方法的变量。通常,变量通过引用传送的初值是不重要的,这些值会被函数重写,函数甚至从来没有使用过它们。

如果可以在C♯中使用这种约定,就会非常方便。但C♯要求变量在被引用之前必须用一个初始值进行初始化。在把输入变量传递给函数前,可以用没有意义的值初始化它们,函数将使用真实、有意义的值初始化它们。但有一种方法能够简化C♯编译器所坚持的输入参数的初始化。

编译器使用out关键字来初始化。当在方法的输入参数前面加上out关键字时,传递给该方法的变量可以不初始化。该变量通过引用传递,所以在被调用的方法中返回时,方法对该变量进行的任何改变都会保留下来。在调用该方法时,还需要out关键字,这正如在定义该方法时一样。

```
static void SomeFunction (out int i)
{
     i = 100;
}
public static int Main ( )
{
    int i; //注意,无论 i 怎样声明但没有初始化
    SomeFunction (out int i);
    Console.WriteLine (i);
    return 0;
}
```

out关键字的引入使C♯更安全,更不容易出错。如果在函数中没有给out参数分配一个值,该方法就不能编译。

6. 方法的重载

C♯支持方法的重载——方法的几个有不同签名(名称、参数个数、参数类型)的版本。为了重载方法,只需声明同名但参数个数或类型不同的方法即可。

```
class ResultDisplayer
{
    void DisplayerResult ( string result )
    {
        //函数体
    }
        void DisplayerResult ( int result )
    {
        //函数体
    }
}
```

因为C♯不直接支持可选参数,所以需要所有方法重载来达到此目的。

```
class MyClass
{
    int DoSomething ( int x )              //第二个参数默认值为 10
    {
        DoSomething ( x, 10 );
```

```
    }
    int DoSomething ( int x, int y )
    {
        //声明部分
    }
}
```

在任何语言中，对于方法重载来说，如果调用了错误的重载方法，就有可能出现运行错误。

现在知道 C♯ 在重载方法的参数方面有一些小区别即可。

- 两个方法不能仅在返回类型上有区别。
- 两个方法不能根据参数是声明为 ref 还是 out 来区别。

技能 4：掌握属性成员

属性描述了对象的具体特性，它提供了对类或对象成员的访问。C♯ 中的属性更充分地体现了对象的封装性，属性不直接操作类的字段，而是通过访问器进行访问。

例如，Windows 窗体的 Height 属性。假定有下面的代码：

```
Windows. Height = 400;
```

执行这段代码，窗口的高度将设置为 400，因此窗口会在屏幕上重新设置大小。在语法上，上面的代码类似于设置一个字段，但实际上是调用了属性访问器，它包含的代码重新设置了窗体的大小。

属性在类模块里是采用下面的方式进行声明的，即指定变量的访问级别、属性的类型、属性的名称，然后是 get 访问器或者 set 访问器代码块。可以使用下面的语法：

```
修饰符 数据类型 属性名称
{
    get
    {
        return "属性值";
    }
    set
    {
        //为属性设值
    }
}
```

get 访问器不带参数，且必须返回属性声明的类型。也不应为 set 访问器指定任何显式参数，但编译器假定它带一个参数，其类型也与属性相同，并表示 value。例如，下面的代码包含一个属性 ForeName，它设置了一个字段 foreName，该字段有一个长度限制。

```
private string forename;
public string ForeName;
{
    get
    {
        return forename;
```