第3章

面向对象程序设计

前面章节介绍了C#程序的调试运行方法以及语言基础和语法规则，由此用户可建立自己的Windows窗体应用程序和控制台应用程序。但是，要深入理解Microsoft .NET Framework的强大功能，还需要掌握面向对象程序设计(Object Oriented Programming，OOP)技术。C#编程语言完全支持面向对象程序设计，本章主要介绍面向对象编程的基本原理和方法。

3.1 面向对象程序设计基础

本节通过与传统的结构化程序设计进行比较，说明什么是面向对象程序设计。同时，举出实例，列出到目前为止读者已经接触到的面向对象的概念。

3.1.1 什么是面向对象程序设计

前面章节介绍的编程方法称为结构化程序设计(Procedural Programming)，有时又称为面向过程的程序设计(Process Oriented Programming)。它以模块功能和处理过程作为程序设计的原则，采用自顶向下、逐步求精的程序设计方法，使用顺序、选择、循环3种基本语句结构。

但是，这种传统的编程方法往往会导致设计出来的应用程序过于单一，因为所有功能都被包含在几个甚至同一个代码块中，而这些代码块也只能服务于单一的程序。因此，为增加这些代码块的重用机会，以完成更多程序，就要使用面向对象程序设计方法，让每个代码块成为独立的模块，以提供特定的功能。

面向对象程序设计是一种创建应用程序的新方法，它使每一个计算机应用程序由一个个单一的、能起到子程序作用的对象(或称单元)组合而成。

3.1.2 类和对象的概念

面向对象程序设计是一种以对象为基础、以事件为驱动的编程技术。对象是程序的

基本元素，事件及其处理程序建立了对象之间的关联。

1. 类

日常生活中存在着无数的实体，如人、车、植物等，每个实体都有一系列的性质和行为。例如，一辆汽车可以定义其颜色、型号、品牌、载重量等性质，还能定义前进、制动、鸣笛等行为。在面向对象程序设计中，针对某个实体性质和行为的具体定义就称为类。

类用来表示在应用程序中处理的实际事物。例如，要建立一个汽车销售管理系统的应用程序，就需要定义一个用来表示各种汽车的类，在这个类中需要定义汽车的一系列性质和行为。

2. 对象

对象是面向对象编程技术的核心，是构成应用程序的基本元素。在类中，每个对象都是该类的一个实例。例如，人类是人这个实体的类，而每个不同的人就是人类的对象。每个人有不同的身高、体重等性质以及不同的起立、行走等行为。在面向对象程序设计中，对象的性质称为属性，在对象上发生的事情称为事件，而针对某个事件产生的行为称为方法。属性、方法和事件构成对象的三要素。

3. 预定义类

到目前为止，读者已经接触到 3 种面向对象的概念，它们是工具箱中的控件、用户自定义的窗体以及数据类型，这些类都是.NET Framework 提供的，被称为预定义类。

在建立 C# Windows 窗体应用程序时，工具箱上的某个控件就是一个标准控件类，如文本框类、标签类等。如果将它拖动到窗体上，就得到真正的控件类对象。

用户自定义的窗体也是一个类，当程序运行或窗体装入内存时就建立了一个窗体类对象。

第 2 章中介绍的数据类型同样是类的概念，当声明了某种数据类型的变量时，该变量就是此数据类型类的一个对象。例如，变量声明语句 int i；实际上是定义了一个整型类对象 i。

3.2 封装和隐藏

在面向对象程序设计中，封装是指将各种成员有机地组织在一个类中。在 C# 中，类的成员包括数据成员、属性、方法和事件。类是实现封装的工具，封装保证了类具有良好的独立性，防止外部程序破坏类内部的数据，同时便于对程序进行维护与修改。

在 C# 中，对于系统预定义的类，用户不能修改，只能用来创建其对象或派生出新的类。本节把理论知识付诸实践，主要介绍如何在 C# 中定义类，以实现对数据的封装和隐藏。

3.2.1 定义类

在C#中,类是通过class关键字来定义的。例如,在创建C# Windows窗体应用程序时所建立的窗体Form1就是一个用class关键字定义的类。

如图3-1所示的源代码是在新建C# Windows窗体应用程序时由系统自动生成的。这里,通过语句public partial class Form1: Form和其后紧跟的一对大括号{}对Form1类进行定义。public表明类是全局的,其意义与变量声明时使用的public关键字意义相同;partial表明在这里仅对Form1类作局部定义;class关键字后指定类名Form1。

用户自定义类的通用格式如下。

```
class 类名
{
    //定义数据成员
    //定义属性
    //定义方法
    //定义事件
}
```

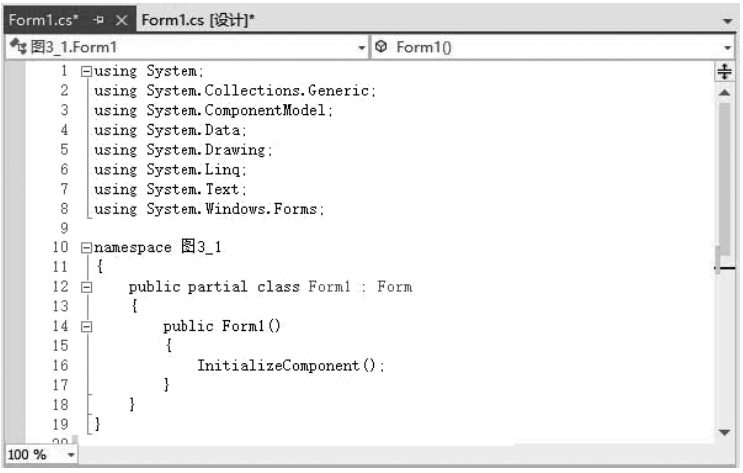

图3-1 Form1类

3.2.2 定义类成员

通过class关键字可以使用户定义自己的类。在类的定义中,也提供了类中所有成员的定义,包括:数据成员(有时也称为字段)、方法、属性和事件。类中所有成员都有自己的访问级别,可通过如表3-1所示的访问修饰符来定义。

表 3-1　常用的访问修饰符

修　饰　符	定　　义
public	类成员可以由任何代码访问
private	类成员只能由类中的代码访问，定义成员时，默认使用 private
protected	类成员只能由类或其派生类(即子类)中的代码访问
internal	类成员只能由包含它的程序集中的代码访问
protected internal	类成员只能由类或其派生类(即子类)以及包含它的程序集中的代码访问

访问修饰符用来指定类成员的作用域，这与第 2 章中介绍的变量的作用域意义相同。

1. 定义数据成员

类的数据成员通过标准的变量声明语句定义，并且结合访问修饰符来指定数据成员的访问级别。为起保护作用，数据成员一般以 private 或 protected 修饰符声明，例如：

```
class Vehicle
{
    private int wheels;                 //车轮数
    private float weight;               //车自重
}
```

以上代码定义了一个汽车类 Vehicle，包含两个 private 数据成员：车轮数 wheels 和车自重 weight。

2. 定义方法

类的方法通过标准的函数声明语句定义，并且结合访问修饰符来指定方法的访问级别。例如，为以上代码中的汽车类 Vehicle 添加两个方法。

```
class Vehicle
{
    ...                                 //定义数据成员 wheels 和 weight

    public void SetVehicle(int wheels, float weight)
                                        //定义方法 SetVehicle()设置车轮数和车自重
    {
        this.wheels = wheels;
        this.weight = weight;
    }

    public void GetVehicle()            //定义方法 GetVehicle()获得车轮数和车自重
    {
```

```
        MessageBox.Show("车轮数: " + this.wheels.ToString() + "\n 车自重:" +
    this.weight.ToString());
    }
}
```

以上代码为汽车类 Vehicle 添加了两个方法。其中,SetVehicle()方法给 Vehicle 的两个数据成员 wheels 和 weight 赋值,而 GetVehicle()方法实现从消息框输出数据成员 wheels 和 weight。

注意:在 SetVehicle()方法中,有两组 wheels 和 weight 变量。一组为 Vehicle 类数据成员,另一组为 SetVehicle()方法定义时的形参。当数据成员名和方法体中的参数(或局部变量)名重复时,可在数据成员名前使用 this 关键字加以区分。

例 3.1 定义一个汽车类 Vehicle,要求实现如下功能。

① 定义数据成员 wheels 和 weight 代表汽车的车轮数和车自重。

② 定义方法 SetVehicle()和 GetVehicle(),可以设置和获取车轮数和车自重。

③ 在窗体上单击“输入数据成员”按钮,实例化 Vehicle 类,利用 SetVehicle()方法和 GetVehicle()方法将文本框中输入的内容作为数据成员输入并输出。

程序运行界面如图 3-2 所示。

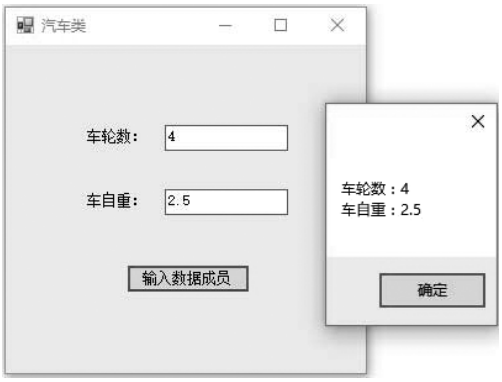

图 3-2 例 3.1 程序运行界面

首先,把前面定义的 Vehicle 类的源代码放在窗体的“代码”窗口中,与 Form1 类并列。

```
public partial class Form1 : Form
{
    ...
}

class Vehicle
{
    ...
}
```

然后,在 Form1 类中编写 button1_Click()事件处理程序。

```
private void button1_Click(object sender, EventArgs e)
{
    int wheels;
    float weight;
    wheels = int.Parse(textBox1.Text);          //从文本框输入车轮数
    weight = float.Parse(textBox2.Text);        //从文本框输入车自重
    Vehicle v = new Vehicle();                  //将 Vehicle 类实例化成对象 v
    v.SetVehicle(wheels, weight);               //调用 SetVehicle(),设置数据成员
    v.GetVehicle();                             //调用 GetVehicle(),输出数据成员
}
```

在以上 button1_Click()事件处理程序中,先从文本框输入车轮数和车自重;然后定义 Vehicle 类对象并赋予一个实例 v;接着调用 SetVehicle()方法设置两个数据成员的值;最后调用 GetVehicle()方法从消息框输出两个数据成员的值。有关对象及其成员的访问可参考 3.2.3 节。

3. 定义属性

在前面定义的 Vehicle 类中,数据成员 wheels 和 weight 是通过 private 修饰符声明的,这种以 private 声明的成员有时也被称为私有成员。私有成员由于受到保护,不能以"对象名.成员"形式赋值或访问。

在例 3.1 中,只能通过类中定义的公共方法(由 public 修饰符声明的方法称为公共方法)SetVehicle()和 GetVehicle()来实现数据成员 wheels 和 weight 的赋值和访问。这种用法会让经常开发 C# 窗体应用程序的程序员很不方便,因为他们已习惯于"对象名.属性名"工作形式。

幸好,C# 中能定义类的属性。类中对属性的定义包括两个类似于函数的代码块:一个用于设置属性值,用 set 关键字定义;另一个用于获取属性值,用 get 关键字定义。可以忽略其中的一个代码块来设置只读或只写属性,例如忽略 set 块来创建只读属性或者忽略 get 块来创建只写属性。属性定义的一般语法形式如下。

```
访问修饰符 数据类型 属性名
{
    set
    {
        ...                                     //属性的 set 代码块
    }
    get
    {
        ...                                     //属性的 get 代码块
    }
}
```

例 3.2 定义一个 Point 类，要求实现如下功能。

① 定义数据成员 x 和 y，分别代表点的横坐标和纵坐标。

② 定义属性 MyX 和 MyY，通过它们对数据成员 x 和 y 进行赋值和访问。

③ 定义只读属性 ReadXY，通过它获取数据成员 x 和 y 经赋值后构成的点坐标。

④ 在窗体上单击“输入数据成员”按钮，实例化 Point 类，对属性 MyX 和 MyY 进行赋值，访问 MyX 和 MyY 属性的结果显示在 label3 中，访问 ReadXY 的结果用消息框输出。

程序运行界面如图 3-3 所示。

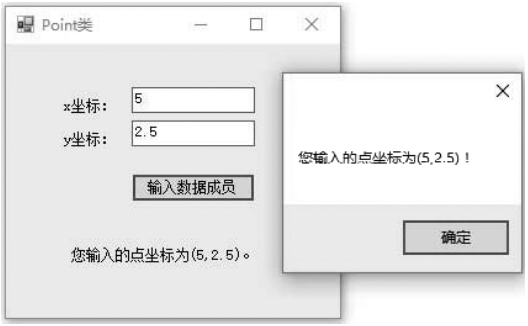

图 3-3 例 3.2 程序运行界面

首先，定义 Point 类，源代码如下。

```
class Point
{
    private float x;                    //x 坐标
    private float y;                    //y 坐标

    public float MyX                    //定义属性 MyX
    {
        set
        {
            x = value;                  //提供对数据成员 x 的赋值
        }
        get
        {
            return x;                   //提供对数据成员 x 的访问
        }
    }

    public float MyY                    //定义属性 MyY
    {
        set
```

```
        {
            y = value;                          //提供对数据成员 y 的赋值
        }
        get
        {
            return y;                           //提供对数据成员 y 的访问
        }
    }

    public string ReadXY                        //定义只读属性 ReadXY
    {
        get
        {
            return "(" + x + "," + y + ")";  //提供对数据成员 x 和 y 构成的点坐标的访问
        }
    }
}
```

然后,在 Form1 类中编写 button1_Click()事件处理程序。

```
private void button1_Click(object sender, EventArgs e)
{
    float x, y;
    string msg1, msg2;
    x = float.Parse(textBox1 .Text);                    //从文本框输入 x 坐标
    y = float.Parse(textBox2.Text);                     //从文本框输入 y 坐标
    Point p=new Point();                                //将 Point 类实例化成对象 p
    p.MyX = x;                                          //为属性 MyX 赋值
    p.MyY = y;                                          //为属性 MyY 赋值
    msg1 = "您输入的点坐标为(" + p.MyX.ToString() + "," + p.MyY.ToString() + ")。";
                                                        //访问属性 MyX 和 MyY
    label3.Text = msg1;
    msg2 = p.ReadXY;                                    //访问属性 ReadXY
    MessageBox.Show("您输入的点坐标为" + msg2 + "! ");
}
```

从例 3.2 中可以看出,属性实际上提供了对类中私有数据成员的一种访问方式。当然,随着对面向对象程序设计的更深入研究,程序员会发现:属性的定义不仅解决了对数据成员的访问问题,更能提供强大的操作控制。

4. 定义事件

在 C# 中,也可以对类定义事件成员,定义事件时需要用到事件委托机制,比较复杂,本书作为一本 C# 语言的入门教程,不作详细介绍。需要进一步了解事件委托机制的读

者可以参考微软发布的 MSDN 官方帮助文档。

3.2.3 对象及其成员的访问

在第 2 章中已学过，变量在使用前必须声明。变量在声明时所指定的数据类型实际上就相当于“类”，而变量则相当于“对象”。在面向对象程序设计中，必须遵循“先定义、后使用”的规则，即任何预定义或自定义的“类”都必须实例化成“对象”后才能使用。

1. 对象的声明

在例 3.1 和例 3.2 中定义了 Vehicle 类和 Point 类，类只有在声明成对象后才能使用。对象声明的格式如下。

```
类名 对象名 = new 类名();
```

在例 3.1 的 Form1 类中编写 button1_Click()事件处理程序，通过语句 Vehicle v = new Vehicle()；实例化类对象 v，然后用对象 v 去访问各成员。同理，通过例 3.2 中的语句 Point p = new Point()；建立 Point 类对象 p。

对象建立后，才能访问其中的各种成员。

2. 成员的访问

类中定义的成员通常需要通过对象才能访问，对不同类型的数据成员，其访问形式也不同，一般格式如下。

- 数据成员：对象.数据成员。
- 属性：对象.属性。
- 方法：对象.方法(参数表)。
- 事件：较复杂，请参考微软 MSDN 官方帮助文档。

在例 3.1 和例 3.2 的 button1_Click()事件处理程序中，使用以上方法对类对象中的各成员进行访问。

例 3.3 建立一个 C＃控制台应用程序，根据已定义的学生类 Student，编写一个能应用该类的程序。要求：

① 类中的每个成员都能访问到。

② 通过 Console 类的 Read()/ReadLine()方法读入数据，通过 Write()/WriteLine()方法输出数据。

③ 程序以键盘输入任意键结束，可通过 Console.ReadKey()方法实现。

已知 Student 类中有 3 个数据成员 sId、sName 和 score，分别代表学生的学号、姓名和得分。其中，学号和姓名通过 SetInfo()方法输入，得分通过 MyScore 属性设置，OutPut()方法可实现对 3 个数据成员的格式化输出。

Student 类的程序源代码如下。

```
class Student
{
    private string sId;       //学号
    private string sName;     //姓名
    private float score;      //得分

    public void SetInfo(string sId,string sName)
                              //定义方法 SetInfo()设置学号和姓名
    {
        this.sId = sId;
        this.sName = sName;
    }

    public float MyScore      //定义属性 MyScore
    {
        set
        {
            score = value;    //提供对数据成员 score 的赋值
        }
        get
        {
            return score;     //提供对数据成员 score 的访问
        }
    }

    public string OutPut()    //定义方法 OutPut(),提供对所有数据成员的格式化输出
    {
        return "学号: " + sId + " 姓名: " + sName + " 得分: " + score;
    }
}
```

从 Student 类的定义中可看出：数据成员 sId、sName 和 score 由 private 关键字修饰，它们都是私有成员，因而不能以“对象.数据成员”形式从类的外部访问。但是，公共方法 SetInfo()提供了对成员 sId 和 sName 的赋值，而通过属性 MyScore 也可对成员 score 进行赋值。最后，公共方法 OutPut()又提供了对这 3 个私有数据成员的格式化输出。

C＃是一种面向对象的程序设计语言，所有代码都通过名称空间中的类来实现，在类中定义各类成员，包括类的数据成员、方法、属性和事件。C＃应用程序的入口规定为 Main()函数。在创建 C＃控制台应用程序时，Main()函数被定义在 Program 类中，该类在创建项目时自动产生。要实现对如上定义的 Student 类的应用，就需要在 Main()中先声明一个 Student 类对象，再以该对象的身份访问其不同类型的成员。

由此，在 Main()函数中可编写如下代码实现对 Student 类中每个成员的访问。

```
static void Main(string[] args)
{
    string id,name;
    float score;

    Console.Write("请输入学号: ");
    id = Console.ReadLine();                    //从键盘输入学号
    Console.Write("请输入姓名: ");
    name = Console.ReadLine();                  //从键盘输入姓名
    Console.Write("请输入得分: ");
    score = float.Parse(Console.ReadLine()); //从键盘输入得分

    Student s = new Student();                  //声明 Student 类对象 s
    s.SetInfo(id, name);                        //设置对象 s 的学号和姓名
    s.MyScore = score;                          //设置对象 s 的得分

    Console.WriteLine("\n========以下是输出========\n");
    Console.WriteLine(s.OutPut());              //输出结果
    Console.Write("\n 按任意键结束程序: ");
    Console.ReadKey();                          //等待用户从键盘输入任意键结束程序
}
```

注意：在以上 Console.WriteLine()方法中，输出字符串中的“\n”代表输出一个换行符。关于控制台输入/输出语句的用法可参见本书前两章的内容。程序运行结果如图 3-4 所示。

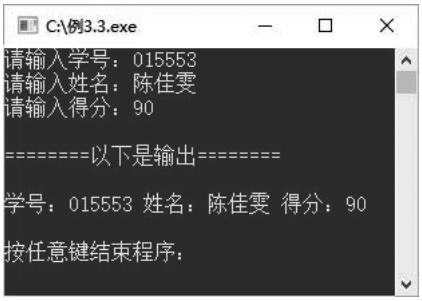

图 3-4　例 3.3 程序运行结果

3.2.4　构造函数和析构函数

在 C＃中定义类时，常常不需要定义构造函数和析构函数。因为如果在定义类时不添加它们，那么在程序编译时，编译器会自动添加。但是，程序员也可以自定义构造函数和析构函数，以实现类对象的初始化和释放操作。

1. 构造函数

构造函数是一种特殊的方法，通过该方法可以在声明对象的同时对数据成员赋初值。它在声明对象时与 new 关键字一起使用。

在类中定义构造函数的语句与定义方法的语句基本相同。但在定义构造函数时，无须提供函数的返回值类型。此外，构造函数的名称必须与其所属的类同名。

例 3.4 为例 3.3 中定义的 Student 类添加一个构造函数，实现在对象初始化时对其中的 3 个数据成员进行赋值。

首先，在 Student 类中定义如下构造函数。

```
public Student(string sId,string sName,float score)
{
    this.sId = sId;
    this.sName = sName;
    this.score = score;
}
```

然后，在 Main()函数中，使用新定义的构造函数实例化类对象。

```
...                                          //从键盘输入 id、name 和 score
Student s = new Student(id,name,score);      //实例化 Student 类对象 s
...                                          //格式化输出
```

由于使用了自定义的构造函数声明 Student 类对象 s，使得对象 s 中的 3 个数据成员 sId、sName 和 score 在声明对象的同时就得到了赋值。因此，在这里可以跳过例 3.3 中通过 SetInfo()方法和 MyScore 属性给数据成员赋值的语句，而直接访问 OutPut()方法进行格式化输出。

2. 构造函数的重载

在面向对象程序设计中，同一个类中也可以声明多个构造函数，可根据其中参数个数的不同(或参数的数据类型的不同)来区分它们，这被称为构造函数的重载。

对于同一个类中的多个构造函数，其函数名称相同，唯一不同的是它们的参数个数或参数的数据类型。在实例化类对象的过程中，编译器会根据构造函数中传递的参数来自动识别用哪一个构造函数创建类对象。

例 3.5 在例 3.4 的基础上为 Student 类添加第二个构造函数，仅对数据成员 sId 和 sName 实现赋值。

先在 Student 类中添加构造函数。

```
public Student(string sId,string sName)
{
    this.sId = sId;
```

```
        this.sName = sName;
    }
```

然后，将 Main()函数中实例化类对象的语句改为如下。

```
Student s = new Student(id,name);                    //实例化 Student 类对象 s
```

以上程序编译时，编译器会根据 new 关键字后构造函数中参数的个数，决定选用只有两个参数的构造函数(即例 3.5 中定义的构造函数)创建类对象。在这里，由于创建对象时只对数据成员 sId 和 sName 进行了赋值，并没有考虑成员 score。因此，为实现完整的程序输出，必须重新利用 MyScore 属性对成员 score 赋值。这就需要在创建类对象语句后添加如下语句。

```
s.MyScore = score;
```

由于构造函数和方法声明类似，因此程序员也可以在类中定义多个同名方法，通过参数个数或参数数据类型的不同加以区分，这被称为方法重载。在 3.4 节中描述类的重载与重写特性时，读者将见到方法重载的实例。

3. 析构函数

析构函数与构造函数的作用相反，当实例化后的类对象使用完毕时，系统会自动执行析构函数。析构函数中编写的代码通常用来做“清理善后”工作。

析构函数名也与类名相同，但为了和构造函数区分，必须在析构函数名前面加上～前缀。例如，为 Student 类定义析构函数。

```
public ~Student()
{
    //在这里定义析构函数,进行垃圾清理,释放资源
}
```

注意：定义析构函数时不指定任何参数，也无返回值类型。和构造函数不同的是，每个类中只能定义一个析构函数，不能重载。当然，如果程序员在定义类时没有编写析构函数，编译器会在对象使用完毕后调用一个默认的析构函数，以释放资源。

3.3 继承和派生

在面向对象程序设计中，继承是其中的一个重要特性。通过继承可以实现代码的重用，节省程序开发的时间和资源。

继承是一个形象的、易于理解的术语，如子承父业、继承遗产等。在面向对象程序设计中，继承则意味着将获得被继承方的所有相关属性及行为，它是一种连接类和类的层次

模型。通过继承，使现有类派生出新类。在这种继承关系中，现有类称为“基类”（或“父类”），而新类则称为“派生类”（或“子类”）。派生类会拥有其基类的一切特性，同时又添加了其自身的新特性。在设计程序时，只要在基类的基础上添加或修改程序代码就可完成对派生类的定义。这样进一步增强了代码的重用性，并且大大提高了软件开发的效率。

3.3.1 基类和派生类

所谓继承，就是在原有类的基础上构造派生类，派生类继承了基类中所有的数据成员、属性、方法和事件。从集合的角度讲，派生类是基类的子集。

例如，若在例 3.1 中 Vehicle 类的基础上定义一个表示小轿车的派生类 Car 和一个表示卡车的派生类 Truck，它们继承 Vehicle 类的一切特性，则 Car 类和 Truck 类都是 Vehicle 类的子集，如图 3-5 所示。

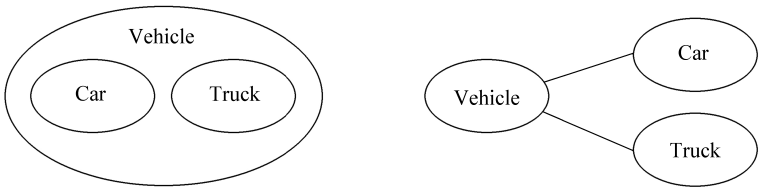

图 3-5 基类与派生类的关系

3.3.2 定义派生类

派生类除能继承基类的一切数据成员、属性、方法和事件外，通常还需要定义自己特有的成员。不仅如此，在实际使用中，派生类往往需要对继承过来的属性、方法等进行改写或扩充，这被称为重写。

定义派生类使用如下语句形式。

```
class 派生类名 : 基类名
{
    ...                                          //在这里定义派生类成员
}
```

例 3.6 在例 3.1 定义的 Vehicle 类的基础上定义派生类 Car，为其添加（或重写）数据成员和方法并应用该类。根据定义的车辆对象，输入当前的载客人数，判断汽车是否超载。程序运行界面如图 3-6 所示。

Car 类的定义应该满足以下要求。

① 定义数据成员 passenger 代表小轿车的载客量。

② 定义构造函数 Car()，可以初始化 Car 类的 3 个数据成员（包括从 Vehicle 类派生过来的 wheels 和 weight）。

(a) 汽车不超载情况　　(b) 汽车超载情况

图 3-6　例 3.6 程序运行界面

③ 定义方法 Overload()，可根据输入的乘客数判断汽车是否超载。

④ 重写基类中派生过来的 GetVehicle()方法，实现对 Car 类的 3 个数据成员的格式化输出。

首先，定义派生类 Car，该类继承自 Vehicle 类(对 Vehicle 类的定义这里不再说明，详见例 3.1)，程序源代码如下。

```
class Car : Vehicle
{
    private int passenger;          //载客量

    public Car(int wheels,float weight,int passenger)  //定义构造函数 Car
    {
        this.wheels = wheels;
        this.weight = weight;
        this.passenger = passenger;
    }

    public Boolean Overload(int p)  //定义方法 Overload,判断汽车是否超载
    {
        if (p > this.passenger)
            return true;
        else
            return false;
    }

    public new void GetVehicle()    //重写方法 GetVehicle(),格式化输出全部数据成员
```

```
    {
        MessageBox.Show( "车轮数: " + this.wheels.ToString() + "\n 车自重: " +
    this.weight.ToString()+"\n 载客量: "+this.passenger.ToString);
    }
}
```

注意：由于在 Car 类中新加入了数据成员 passenger，为获得对所有数据成员的输出，需要在 Car 类中重写从 Vehicle 类中继承过来的 GetVehicle()方法，这称为方法重写（又称方法覆盖）。通常，方法重写时要在新定义的方法返回值类型前加上 new 关键字，以此隐藏继承过来的同名方法。例如：

```
public new void GetVehicle()
```

有关方法重写的详细内容，可参见 3.4 节中的介绍。

在定义 Vehicle 类时，曾将数据成员 wheels 和 weight 用 private 关键字修饰，这种以 private 修饰的类成员只能由类中的代码访问。现在，新定义的派生类 Car 必须继承其基类中的所有数据成员。为了能在派生类中访问基类中的数据成员 wheels 和 weight，就需要在 Vehicle 类中将这两个数据成员以 protected 关键字修饰。

因此，修改 Vehicle 类中定义数据成员的源代码。

```
class Vehicle
{
    protected int wheels;                              //定义受保护数据成员 wheels
    protected float weight;                            //定义受保护数据成员 weight

    ...                                                //定义 Vehicle 类的其他成员
}
```

以 protected 关键字修饰的成员可以被类或其派生类（即子类）中的代码访问。这样，既达到了对外保护类成员的目的，又做到了对内允许其派生类访问的效果。

最后，在 Form1 类中编写 button1_Click()事件处理程序。

```
private void button1_Click(object sender, EventArgs e)
{
    int wheels, passenger,p;
    float weight;

    wheels = int.Parse(textBox1.Text);             //从文本框输入车轮数
    weight = float.Parse(textBox2.Text);           //从文本框输入车自重
    passenger = int.Parse(textBox3.Text);          //从文本框输入载客量
    p = int.Parse(textBox4.Text);                  //从文本框输入当前载客人数
```

```
        Car c = new Car(wheels, weight, passenger);   //实例化 Car 类对象 c,同时对数据
                                                      //成员初始化
        if (c.Overload(p))                    //根据输入的当前载客人数,判断汽车是否超载
        {
            MessageBox.Show("请注意,车辆已超载,请减少乘客数! ");
            textBox4.Text = "";
            textBox4.Focus();
        }
        else
        {
            MessageBox.Show("车辆未超载,请安全驾驶! ");
            MessageBox.Show("本车正在行驶中…");
            c.GetVehicle();
        }
    }
```

由于在定义 Car 类时使用 new 关键字重写了从基类继承过来的 GetVehicle()方法，因此在这里编译器会自动识别语句“c.GetVehicle();”调用的是派生类中重写的 GetVehicle()方法。

3.4 重载和重写

类构成了实现 C# 面向对象程序设计的基础。类具有 3 个基本特性：封装性、继承性和多态性。本章前两节已介绍了类的封装性和继承性，本节主要讨论类的多态性特征。

在面向对象程序设计中，多态性是指同一个消息被不同的对象接收后导致完全不同的行为。例如，教室中有老师和学生两类对象，当上课铃响时，老师准备讲课，学生准备听课。也就是说，当上课铃响这一消息传来时，老师和学生的行为是完全不同的。

多态性允许对象以适合自身的方式去响应共同的消息，不必为相同功能的操作作用于不同对象而去刻意识别，这为软件开发和维护提供了极大的方便。在 C# 面向对象程序设计中，多态性通常体现在对函数或方法的重载与重写上。

3.4.1 重载

3.2.4 节中的例 3.4 和例 3.5 分别为 Student 类定义了两个构造函数。

```
public Student(string sId,string sName,float score)          ①
public Student(string sId,string sName)                      ②
```

这两个函数用来在创建 Student 类对象时完成对数据成员的初始化操作，它们的名称相同，但参数个数不同，这就是重载。所谓重载，就是几个不同的函数或方法共用一个

相同的名称，通过其中的参数个数或参数的数据类型加以区分。

在编译阶段，编译器可通过函数或方法调用时所传递参数的个数或数据类型来确定应该调用哪一个被重载的函数或方法。例如，语句 Student s = new Student(id,name,score);是对构造函数①的调用，因为它有 3 个实参；而语句 Student s = new Student(id,name);是对构造函数②的调用，因为它只有 2 个实参。

在 C# 中，重载要求函数或方法名称相同，但参数列表必须不同。也就是说，要么参数个数不同，要么参数的数据类型不同。重载可以在同一个类中实现，也可以在派生类和基类的关系中实现，请看下面两个实例。

1. 在同一个类中实现重载

例 3.7 在例 3.3 定义的 Student 类中重载 SetInfo()方法，使之能对所有的 3 个数据成员 sId、sName 和 score 进行初始化，并且实现对该方法的应用。

在例 3.3 定义的 Student 类中已经定义了一个包含两个参数的 SetInfo()方法，可以实现对学号 sId 及姓名 sName 成员赋初值的操作。

```
public void SetInfo(string sId, string sName)
{
    this.sId = sId;
    this.sName = sName;
}
```

现在添加新的 SetInfo()方法，实现方法重载。

```
public void SetInfo(string sId, string sName, float score)
{
    this.sId = sId;
    this.sName = sName;
    this.score=score;
}
```

然后，在 Main()函数中，调用新的 SetInfo()方法。

```
s.SetInfo(id, name,score);                    //设置对象 s 的学号、姓名和得分
```

在例 3.3 程序的基础上，需要将 Main()函数中为 MyScore 属性赋值的语句去除。删除以下语句。

```
s.MyScore = score;                            //设置对象 s 的得分
```

在这里，由于调用了带 3 个参数的 SetInfo()方法，它能同时为学号、姓名和得分赋值。因此，在本例中不再需要通过 MyScore 属性为数据成员 score 赋值。

2. 在派生类继承基类时实现重载

例 3.8 在例 3.7 中 Student 类的基础上定义派生类 StudentLeader，即学生干部类。程序期望运行界面如图 3-7 所示。要求：

① 新增数据成员 duty 代表学生干部的职责，只继承 Student 类中的 sId 和 sName 数据成员。

② 重载 SetInfo()方法，能实现对派生类的所有数据成员赋初值。

首先，为了让派生类能访问基类中的数据成员，需要将 Student 类中的数据成员 sId 和 sName 以 protected 关键字修饰，代码如下。

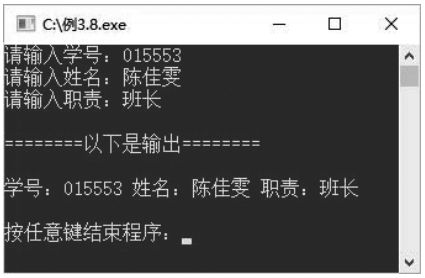

图 3-7 例 3.8 程序期望运行界面

```
class Student
{
    protected string sId;                    //定义受保护数据成员 sId
    protected string sName;                  //定义受保护数据成员 sName
    private float score;                     //定义私有数据成员 score,使之不被继承

    ...                                      //定义 Student 类的其他成员
}
```

在 StudentLeader 类中，我们只关心学生干部的学号、姓名和职责，表示得分的 score 数据成员不被继承，因此仍然保留 private 修饰符。

然后，定义派生类 StudentLeader，继承自 Student 类，添加数据成员 duty，重载 SetInfo()方法，代码如下。

```
class StudentLeader : Student
{
    private string duty;                     //职责

    public void SetInfo(string sId, string sName, string duty)
                                             //重载方法 SetInfo()设置学号、姓名和
                                             //职责
    {
        this.sId = sId;
        this.sName = sName;
        this.duty = duty;
    }
}
```

最后，在 Main()函数中编写如下代码创建并应用 StudentLeader 类对象。

```
static void Main(string[] args)
{
    string id, name,duty;

    Console.Write("请输入学号: ");
    id = Console.ReadLine();                    //从键盘输入学号
    Console.Write("请输入姓名: ");
    name = Console.ReadLine();                  //从键盘输入姓名
    Console.Write("请输入职责: ");
    duty = Console.ReadLine();                  //从键盘输入职责

    StudentLeader sl = new StudentLeader();     //声明 StudentLeader 类对象 sl
    sl.SetInfo(id, name,duty);                  //设置对象 sl 的学号、姓名和职责

    Console.WriteLine("\n========以下是输出========\n");
    Console.WriteLine(sl.OutPut());             //输出结果
    Console.Write("\n 按任意键结束程序: ");
    Console.ReadKey();                          //等待用户从键盘输入任意键结束程序
}
```

在以上代码中，为显示如图 3-7 所示的输出结果，调用了 sl 对象的 OutPut()方法。但是，由于 OutPut()方法继承自 Student 类，该方法被定义在基类中，用来输出学号、姓名和得分。因此，若这样直接调用基类中的 OutPut()方法会产生错误的输出结果，如图 3-8 所示。

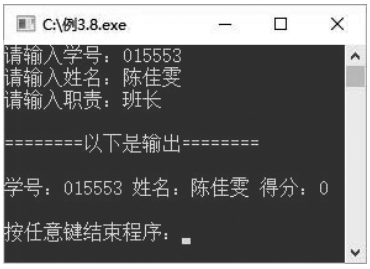

图 3-8　例 3.8 程序实际运行界面

实际上，为获得如图 3-7 所示的正确输出结果，可以在派生类 StudentLeader 中重载 OutPut()方法，输出 sl 对象的 sId、sName 和 duty 数据成员。但是，实现方法重载的前提条件必须是同名方法的参数列表有所不同，即要么参数个数不同，要么参数的数据类型不同。

在基类 Student 中定义的 OutPut()方法的源代码如下。

```
public string OutPut()
{
    return "学号: " + sId + " 姓名: " + sName + " 得分: " + score;
}
```

可以看出，该方法不带任何参数。这就意味着，如果要在 StudentLeader 类中重载 OutPut()方法，必须在该方法名后至少添加一个参数。即：

```
public string OutPut(参数列表)
{
```