第 5 章 面向对象程序设计基础

本章导引

在 21 点游戏中，包括如下的一些人和事物：参与者——玩家、庄家；游戏对象——扑克牌。在之前的程序中，我们更多地实现了游戏的逻辑，但没有在程序中编写体现这些概念的代码，这说明我们的程序与实际的情况是有区别的。

如何才能让程序与实际情况更一致呢？程序如何才能更好地反映现实世界呢？

在现实世界中，我们称这些参与游戏的人和事物为对象，在C＃语言中，也有对象的概念，描述对象的数据结构称为类。在这一章中，将对类的概念、相关的语法等进行详细地介绍。在本章的最后，我们将把面向对象的概念引入 21 点游戏中，带领读者编写面向对象的 21 点游戏程序。

5.1　面向对象的概念

在问到“这个世界是由什么组成的”这个问题时，化学家的答案是“这个世界是由分子、原子、离子等化学物质组成的”；画家会说“这个世界是由不同的颜色所组成的”；对于分类学家来说，“这个世界是由不同类型的物与事所构成的”。实际上，每个人都是一个分类学家，分类是人们认识世界的一种手段。

作为面向对象的程序员来说，我们要站在分类学家的角度去考虑问题。这个世界是由动物、植物等组成的。动物又分为单细胞动物、多细胞动物、哺乳动物等等，哺乳动物又分为人、大象、老虎……就这样，通过分类，我们了解了世界。

所谓对象，就是将组成这个世界的所有事物都看成对象。因此，对象的概念是具体的，它可以是一个人、一只狗、一只猫或者一辆汽车、一个皮包。那么在 21 点游戏中，游戏的参与者玩家是对象，游戏的道具扑克是对象，游戏本身也是对象。

5.1.1　类的概念

在现实世界中如何来对不同对象进行区分呢？例如，如何区分一只小猫和一条小狗？可以说，它们属于不同类型的事物。那么，如何区分小李和小张这两个人呢？他们属于同一类事物“人类”，却是两个独立的个体(对象)。我们说小李和小张长相不一样，他们能做的事情也不一样。也就是说，对于不同类型的事物，我们可以通过它们所属的类型来进行区分，而对于同一类事物，我们通过特征值和功能来进行区分。

通过上面的分析可以看出，对于同一类事物，可以用相同的特征和功能来描述，当这些功能具体的值或功能的执行结果不同时，我们就可以将同一类型的不同对象区分开来。在面向对象的程序设计语言中，描述同一类事物的数据结构称为“类”。类是对一类具有共同特征的事物的总体描述，将可以描述这类事物的特征、以及这类事物可以执行的功能封装起来，就得到了这类事物对应的类。

可以说，类是一个定义对象形式的模版，它指定了数据以及操作数据的代码。这里所说的数据就是前面提到的“特征”，这里所说的操作数据的代码就是前面提到的“功能”。

5.1.2　类的定义

在C＃中定义一个类的语法结构如下：

```
[修饰符] class 类名 [:基类][,要实现的基接口]
{
    类成员;
}
```

首先是类的修饰符，可以作为类的修饰符的有 public、internal 与 abstract、sealed 等的组

合。public 和 internal 是访问修饰符，决定类的访问限制级别及可访问性；abstract、sealed 是普通修饰符，决定类的种类和作用。类的访问修饰符默认值为 internal。abstract 修饰的类称为抽象类，sealed 修饰的类称为密封类。表 5-1 中给出了类的修饰符可用的组合以及其含义

表 5-1　　类的修饰符

修饰符	说明
public	可以在程序的任何位置访问该类。类可以实例化，也可以被继承
internal	只能在类所在项目中访问该类。类可以实例化，也可以被继承
public abstract	可以在程序的任何位置访问该类。类不能够实例化，只能被继承
public sealed	可以在程序的任何位置访问该类。类可以实例化，但不能被继承
internal abstract 或 abstract	只能在类所在项目修饰符及组合中访问该类。类不能够被实例化，只能被继承
internal sealed 或 sealed	只能在类所在项目中访问该类。类可以实例化，但不能被继承

接下来是 class 关键字，编译器识别这个关键字，就知道下面将要给出的是一个类的定义。class 关键字后面是类的名字，涉及到名称的时候，需要注意命名规范的问题。类的命名一般使用 PascalCase 的命名法，当然组成类的名称的单词或缩写必须是有实际意义的，类的名字一般都是以名词或名词词组命名。

冒号后面的是基类和基接口。当一个类的后面有基类和基接口时，就涉及到了继承的概念，我们称当前正在定义的类为派生类或子类。关于继承的概念，将在下一章详细阐述。需要记住的是，C＃中类不能实现多继承，也就是说，一个类最多只能继承自一个基类，但是一个类可以实现多个接口。

了解了定义类的语法，在【实验 5-1】中，我们将讨论如何在应用程序中添加和使用类。

【实验 5-1】创建和使用定义 21 点游戏中的类

内容：创建 21 点游戏项目，练习类的添加和使用。

设计：

(1)游戏元素：21 点游戏中包括游戏玩家、电脑玩家、游戏、扑克牌、一副牌。这些元素都需要在游戏中体现，使用类进行描述。

(2)类的访问修饰符：对于类的访问限制只在当前项目中即可，访问限制修饰符选择 internal。在游戏开始后，将有具体的游戏元素对象与之对应，因此，不使用 abstract 修饰。类的定义不是用作其他类的基类，不使用 sealed 修饰。

实现：

(1)创建控制台应用程序 Demo5-1，并保存到适当的位置。

(2)在解决方案资源管理器中右键单击项目名称，选择【添加】→【类】菜单项，如图 5-1 所示。

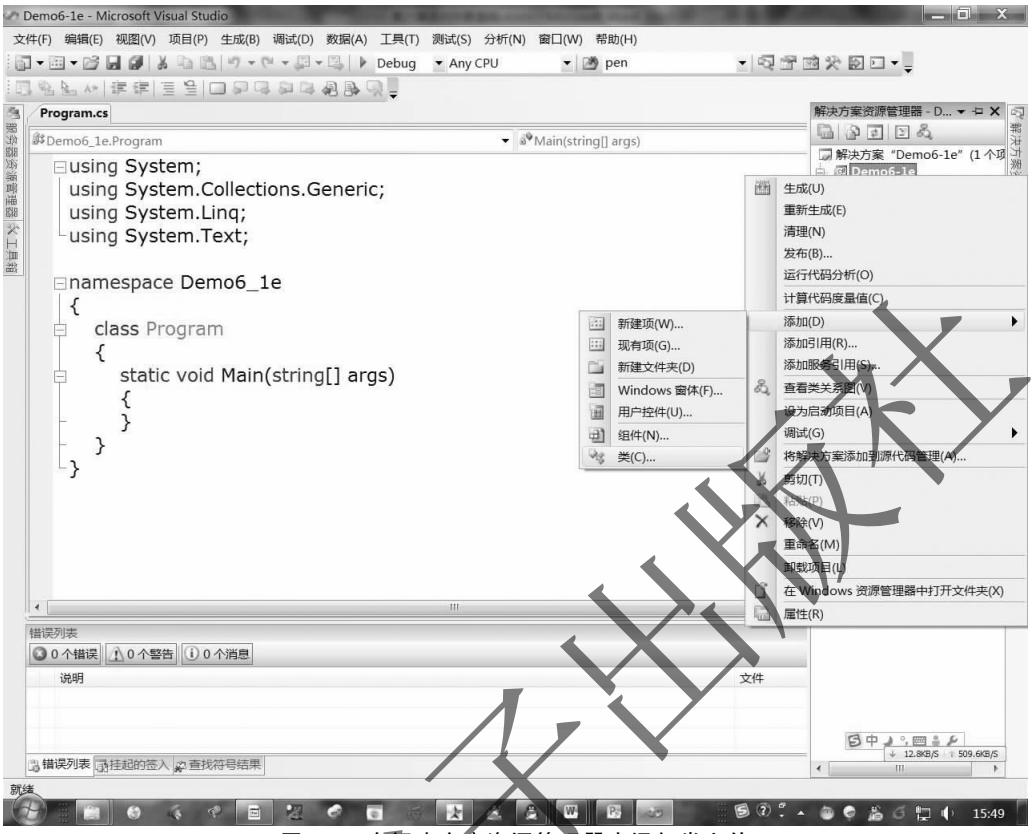

图 5-1　在解决方案资源管理器中添加类文件

(3)在弹出对话框中为该类命名为 Player,单击“添加”按钮,如图 5-2 所示。

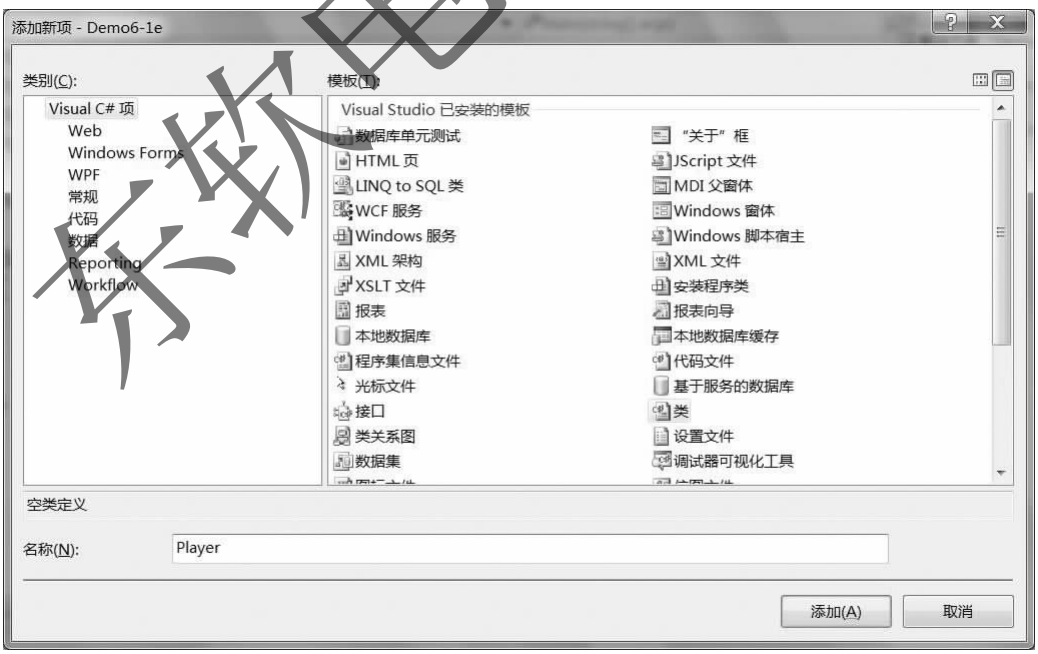

图 5-2　在项目中添加类

(4)同上添加类 ComputerPlayer、Game、Card 和 Deck,分别用于描述电脑玩家、游戏、扑克牌和一副牌。查看项目中的类文件,如图 5-3 所示。

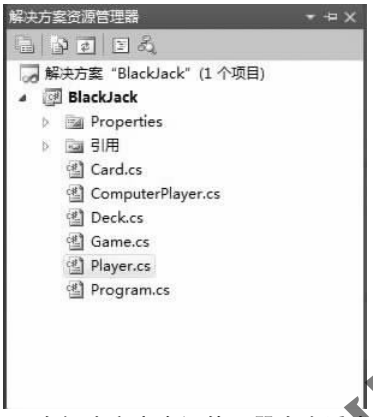

图 5-3 在解决方案资源管理器中查看类文件

(5)双击资源管理器中的类 Player,得到自动生成的类代码如图 5-4 所示。

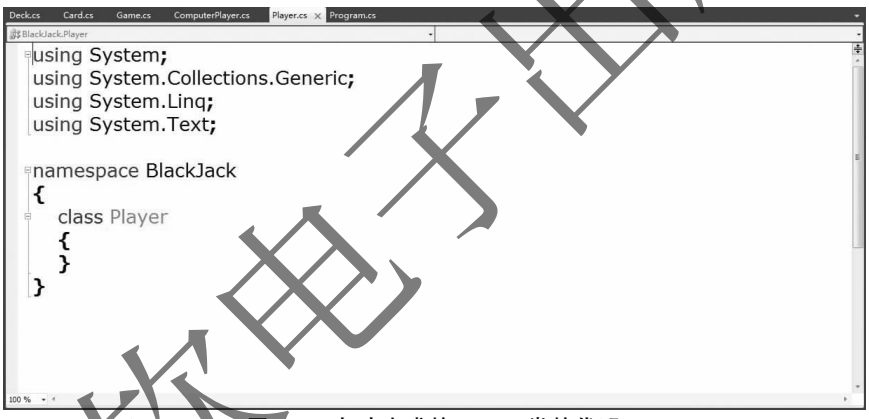

图 5-4 自动生成的 Player 类的代码

分析:

(1)在本例中添加了 5 个类,用于描述游戏中的元素。在应用程序中右键单击解决方案资源管理器中的"项目",可以添加类文件,或其他项目文件。此外,在菜单项中选择【项目】(如图 5-5),以及直接使用快捷键【Shift+Alt+C】,也可以实现添加类文件的功能。

(2)在.NET的项目中,一般将每个类保存在一个单独的文件中,也可以将多个类保存在同一个文件中,即上述的示例代码可以移至 Program.cs 文件的 class Program 定义下面,如图 5-6 所示。在.NET 2.0 以后的版本中,也允许将一个类的定义分解至多个文件中,但这种情况多适用于具有图形界面的类,在本书的 Windows 编程篇中将有所涉及。

(3)自动生成的类文件中只包含类的名称信息,且不会添加任何修饰符。因此,自动生成的类是 internal 访问限制级别的类。

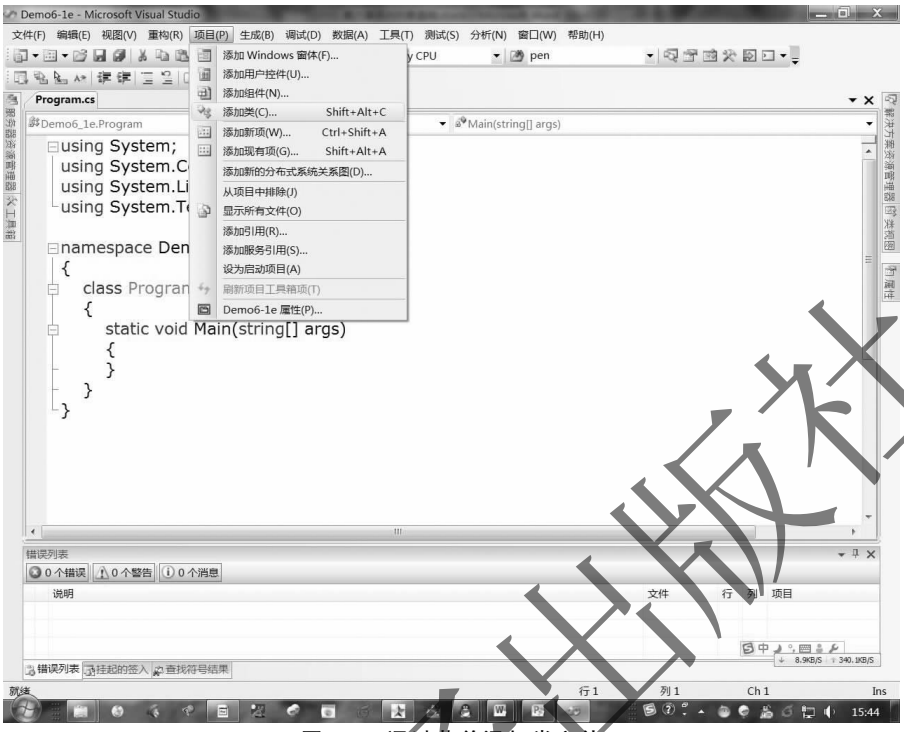

图 5-5　通过菜单添加类文件

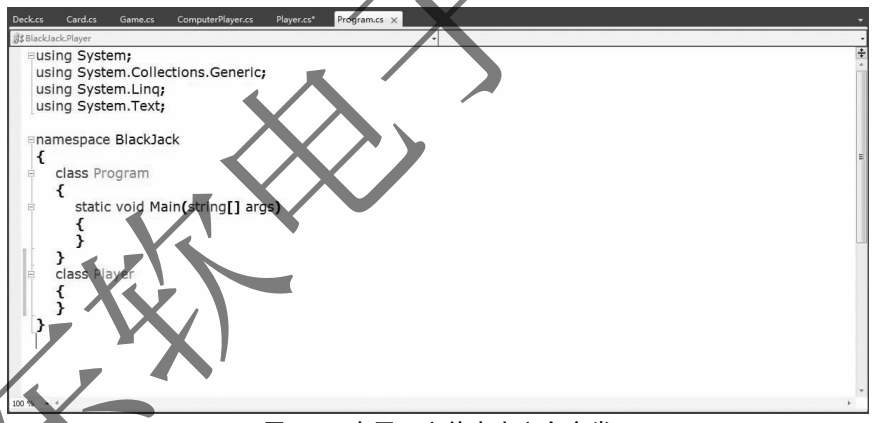

图 5-6　在同一文件中定义多个类

5.1.3　类对象、构造函数和析构函数

(1)类对象。

类类型的变量称为对象,对象是具体的,是类成员具体化之后的产物。声明类对象的语法为:

类名 对象名;

创建类对象要使用 new 方法,语法如下:

对象名 = new 类名([参数列表]);

可以在声明的同时创建类对象,即:

类名 对象名 = new 类名([参数列表]);

(2)构造函数。

在使用 new 方法创建类对象的时候,也就是将类成员赋予具体值的时候。C#提供构造函数初始化类的成员,因此使用 new 方法创建类对象也就是在调用构造函数。定义构造函数的语法如下:

```
[修饰符] 类名([参数列表])
{
    代码段
}
```

构造函数的修饰符一般使用 public。如果是 private 类型的构造函数,该类不能在类的外部实例化。构造函数是与类同名的,因此函数的名字也就是类的名字。与普通函数不同,构造函数一定没有返回值,因此不需要定义构造函数的返回值类型。

构造函数的修饰符还可以是 static,称为静态的构造函数。有 static 修饰的构造函数只在第一次创建该类对象或引用类的静态成员时会被调用,而且在给定的应用程序域中静态构造函数最多被执行一次。

【实验 5-2】声明和创建类 Card

内容:为项目 BlackJack 的类 Card 添加构造函数,并在主函数中创建类对象。

设计:

(1)扑克牌需要 2 个值来描述:牌的级别和牌的花色。使用字段 rank 描述牌的级别,使用字段 color 描述牌的花色,二者均定义为 int 类型。rank 的取值范围为 0～15,代表王,A,2,3……K,大王和小王;color 的取值为 0～4,代表♠、♥、◆、♣,以及没有花色(对应大小王)。

(2)构造函数用于给这两个字段赋值,第一次赋值称为初始化。

实现:

(1)创建控制台应用程序 Demo5-2,并保存到适当的位置。

(2)根据【实验 5-1】为项目添加类 Card。

(3)为类 Card 添加代码如下:

```
class Card
{
    public nt rank;// 用于描述牌的级别
    public nt color;// 用于描述牌的花色
    /// <summary>
    /// 无参的构造函数
    /// </summary>
    public Card()
        {
            rank = 0;
            color = 0;
        }
        /// <summary>
        ///有参的构造函数,具有指定的级别和花色
        /// </summary>
```

```
        /// <param name = "rank">级别</param>
        /// <param name = "color">花色</param>
        public Card(int rank,int color)
        {
            this.rank = rank;
            this.color = color;
        }
    }
```

(4)在 Main 函数中声明和实例化 Card 类,代码如下:

```
static void Main(string[] args)
{
    Card cardA = new Card();//调用无参的构造函数
    Card cardB = new Card(3,3); //调用有 2 个参数的构造函数
}
```

分析:

(1)在类 Card 中定义了两个构造函数,一个是无参的,另一个有两个参数。在使用 new 方法创建类对象时,会根据参数的类型选择构造函数。Main 函数中创建的 cardA 对象,在创建时就调用了无参的构造函数。

(2)Main 函数中创建 cardB 对象时,指定了两个参数,因此调用了具有两个参数的构造函数。在定义有参构造函数时,使用了名字与类成员名字一致的参数,则在构造函数内部,如果直接使用参数名字,根据临近原则代表的是函数的参数而不是类的同名成员;如果希望访问类的成员,需要使用“this.成员名”。

(3)析构函数。

创建类对象时,使用构造函数初始化类成员,释放该对象时则需要使用析构函数,释放该类对象所占用的资源。

析构函数的语法如下:

```
~类名()
{
代码体;
}
```

析构函数的名字也与类同名,但需要在前面添加“~”。析构函数不允许定义返回值,也没有参数,因此它不能被重载,一个类最多只有一个析构函数。类如果没有显式地声明析构函数,则编译器将自动产生一个默认的析构函数。

5.2　定义和访问类成员

在类的“{}”中封装了对类成员的定义,类的成员包括字段成员、属性成员和方法成员三种(构造函数和析构函数可以看作是特殊的方法成员),下面分别对这三种成员的定义和访问进行介绍。

5.2.1 字段

(1)字段成员的定义。

类的字段成员主要用于描述类的特性,存放类的数据。定义字段成员的语法为:

[修饰符]数据类型字段名;

这里的数据类型可以是.NET Framework 已经定义的类型(预定义类型)如 int、bool 等,也可以是字段的命名规则符合C#中关键字的命名规则。一般使用 camelCase 命名法,有时也使用 m_camelCase 或 m_PascalCase 的命名方式。当字段取得不同值时,就得到了同一个类的不同对象。

(2)对字段成员的访问。

在类的内部对字段成员的访问可以直接进行,如【实验 5-2】中的无参构造函数中,对类字段的访问。

在类的外部对字段成员的访问一般通过“对象.字段名”的形式实现。

(3)修饰符。

字段的修饰符可以是普通修饰符或者访问限制修饰符。字段成员的访问限制修饰符和其他成员的访问限制修饰符是一致的,如表 5-2 所示。

表 5-2 类成员的访问限制修饰符

修饰符	说明
public	允许在类的内部及外部访问该成员
protected	仅允许在类及其派生类中访问该成员
private	仅允许在类的内部访问该成员

字段的其他修饰符主要包括 static、const 和 readonly。

①static。

使用关键字 static 修饰的成员称为静态成员。在访问级别允许的情况下,在类及其派生类中访问类的静态成员,可以直接访问;在类及其派生类以外,访问类的静态成员的语法为:“类名.成员名”。

需要注意的是,在静态的成员方法中,可以直接访问本类中的静态成员,但对于本类中的非静态成员,是不能够直接访问的,要参考在类及其派生类之外访问非静态成员的方法来访问。

②const。

const 修饰的字段中保存常量值,该字段必须在声明的同时初始化,const 字段默认就是 static 的,不需要再显式地将字段声明为 static。

③readonly。

readonly 修饰的字段可以在声明的同时初始化,也可以在构造函数中初始化,但是在代码的其他位置修改该字段的值就会引发编译错误。readonly 修饰的字段也可以作为静态成员来使用,但是它不是默认 static 类型的,需要显式地说明。

【实验 5-3】访问类 Card 的字段

内容:在主函数中实现对 Card 类字段的访问。

实现：

(1)创建控制台应用程序 Demo5-3，并保存到适当的位置。

(2)根据【实验 5-2】为项目添加类 Card，并定义 Card 类成员。

(3)在 Main 函数中再添加如下代码。

```
Console.WriteLine(cardA.rank);
Console.WriteLine(cardB.color);
```

(3)使用【Ctrl＋F5】执行程序，可以得到如图 5-7 所示的结果。

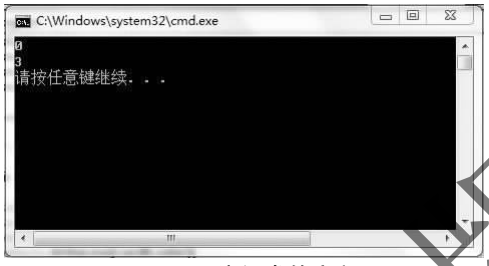

图 5-7　访问类的字段

(4)将项目中 Card 类的 rank 字段修饰符 public 删除，再次使用【Ctrl＋F5】执行程序，可以得到如图 5-8 和图 5-9 的结果。

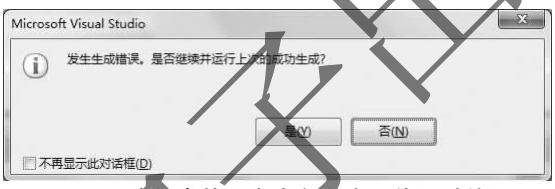

图 5-8　访问类的私有字段程序不能通过编译

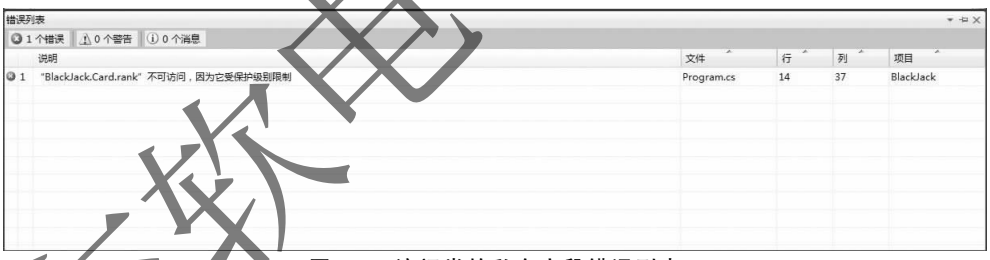

图 5-9　访问类的私有字段错误列表

分析：

(1)在实现的步骤(1)中通过类对象 cardA 访问了类的字段 rank 和 color，由于字段的访问限制级别是 public，程序没有语法错误，成功执行，窗口中显示了牌 cardA 的两个字段值。

(2)在实现的步骤(3)中，由于删除了 public，字段没有显示的访问限制符，则访问限制级别采用默认值 private。此时，在类的外部访问字段 rank，就出现了编译错误，并给出了“保护级别限制”的错误提示。这说明 private 字段是不允许在类的外部访问的，private 修饰的其他成员也是这样。

(3)一般地，类的字段成员应尽量地设置为 private，避免其直接暴露于类的外部，从而体现出类的封装性。这也与现实世界的情况一致，现实世界中事物所展现出来的特征与其内在的本质也不一定相同，有些内在的特征并不显露出来。这些字段如需在类的外部执行访问，将通过对应的属性实现，下一小节将介绍属性的定义和使用。

5.2.2 属性

1. 属性的定义

为实现类的封装性，类的字段成员一般为 private 访问限制级别，也就是说，类的字段成员在类的外部一般是不允许访问的。使用属性可以部分地放宽对类成员访问的限制，因此，属性也称为字段访问器。属性定义的语法为：

```
[修饰符]数据类型属性名
{
    [可访问修饰符]get
    {
    获取值的代码块(包含了 return 语句);
    }
    [可访问修饰符]set
    {
    修改字段值的代码块(包含了赋值语句);
    }
}
```

属性的命名一般采用 PascalCase 方式。属性中的 get 称为读访问器，set 称为写访问器；属性可以只包含读访问器 get，或者只包含了写访问器 set；在写访问器 set 中的赋值语句里使用 value 关键字来引用用户提供的属性值；在获取值和设置值之前还可以进行相应的判断；属性也可以使用 virtual、override 和 abstract 关键字修饰，分别用于基类的属性定义、派生类中重写属性以及抽象类中属性的定义。此外，访问器也可以有可访问性限制，如果在访问器之前没有添加可访问修饰符，则认为访问器的访问限制级别与整个属性是一致的，访问器的访问性限制不能高于其所属的属性。

【实验 5-4】定义和访问类 Card 的属性

内容：为 Card 类的字段定义属性，并访问属性。

设计：

(1)对于一张扑克牌，牌面上的花色和级别都是扑克牌生产出来时就确定的、无法改变的。因此，对应的两个字段应该设置为 readonly。

(2)扑克牌的花色和级别都是有限的、离散的数据，因此可以为它们定义特定的枚举类型。

(3)在类的外部，我们无法对 Card 的两个字段进行修改，只是获取它们的值。因此，只需要为它们提供只读属性。

(4)有时，我们需要获取牌面的数据，同时包括花色和级别。可以提供一个属性，同时获取这两个值，并且使用常规的表示方法。

实现：

(1)创建控制台应用程序 Demo5-4，并保存到适当的位置。

(2)按照【实验 5-3】添加和编写 Card 类代码。

(3)在项目中添加枚举类型 CardRank 和 CardColor，分别描述 Card 的花色和级别，代码如下：

```
public enum CardRank
{
    /// <summary>
    /// Ace
    /// </summary>
    A,
    /// <summary>
    /// 2
    /// </summary>
    Two,
    /// <summary>
    /// 3
    /// </summary>
    Three,
    /// <summary>
    /// 4
    /// </summary>
    Four,
    /// <summary>
    /// 5
    /// </summary>
    Five,
    /// <summary>
    /// 6
    /// </summary>
    Six,
    /// <summary>
    /// 7
    /// </summary>
    Seven,
    /// <summary>
    /// 8
    /// </summary>
    ight,
    /// <summary>
    /// 9
    /// </summary>
    Nine,
    /// <summary>
    /// 10
    /// </summary>
    Ten,
```

```
        /// <summary>
        /// J
        /// </summary>
        Jack,
        /// <summary>
        /// Q
        /// </summary>
        Queen,
        /// <summary>
        /// K
        /// </summary>
        King,
        /// <summary>
        ///小王
        /// </summary>
        SmallJoker,
        /// <summary>
        /// 大王
        // </summary>
        BigJoker
    }
    public enum CardColor
    {
        /// <summary>
        /// 红心
        /// </summary>
        Heart = 3,
        /// <summary>
        /// 方块
        /// </summary>
        Diamond,
        /// <summary>
        /// 草花
        /// </summary>
        Club,
        /// <summary>
        /// 黑桃
        /// </summary>
        Black = 3,
        /// <summary>
        /// 没有,指代大小王
        /// </summary>
```

```
    None
}
```

(4)修改 Card 类的字段 rank 和 color 的修饰符为 readonly,并修改其数据类型分别为 CardRank 和 CardColor。

(5)为 rank 和 color 字段添加对应的只读属性 Rank 和 Color,其中只有读访问器,并编辑注释。

(6)添加一个 string 类型的属性,用于获取级别和花色,并且以常规的形式显示,并编辑注释。类 Card 的代码如下。

```
class Card
{
    readonly nt rank;//用于描述牌的级别
    readonly nt color;//用于描述牌的花色
    /// <summary>
    /// 获取牌的级别
    /// </summary>
    public nt Rank
    {
        get { return rank; }
    }
    /// <summary>
    /// 获取牌的花色
    /// </summary>
    public nt Color
    {
        get { return color; }
    }
    /// <summary>
    /// 获取牌面信息
    /// </summary>
    public string CardStr
    {
        get
        {
            if (color ! = CardColor.None)
            {
                return ((char)((int)color)).ToString() + rank.ToString();
            }
            else
            {
                return rank.ToString();
            }
        }
```

```
        }
        /// <summary>
        /// 无参的构造函数
        /// </summary>
        public Card()
        {
            rank = CardRank.A;
            color = CardColor.Black;
        }
        /// <summary>
        ///有参的构造函数,具有指定的级别和花色
        /// </summary>
        /// <param name = "rank">级别</param>
        /// <param name = "color">花色</param>
        public Card(CardRank rank,CardColor  color)
        {
            this.rank = rank;
            this.color = color;
        }
    }
```

(7)修改 Main 函数中的代码,创建类对象,并访问相应的属性,代码如下。

```
Card cardA = new Card();
Card cardB = new Card(CardRank.Jack,CardColor.Black );
Console.WriteLine(cardA.Rank);
Console.WriteLine(cardA.Color);
Console.WriteLine(cardB.CardStr);
```

分析:

(1)属性有时不仅是针对一个字段。如本实验中的属性 CardStr,它根据两个字段的值取得了返回值。在属性的写访问器中,可以添加条件语句,确保字段的赋值的取值范围;在属性的读访问器中也可以对字段进行处理后返回,有的属性甚至可以与任何字段都没有关系。属性的使用帮助实现了类的封装性。

(2)为了在使用属性的时候能够看到对属性的用途等信息的描述,可以在类的定义中为属性添加注释,注释的标准写法为:

```
/// <summary>
        ///注释的内容
        /// </summary>
```

2. 属性的声明和定义

编写了标准注释的属性在访问时,可以查看其注释的内容,提高了程序的可读性并为程序的协作编写提供了方便。不仅是属性,其他的代码也应该及时地编写易于理解的注释,在商业应用软件中,有时代码的可读性比效率更重要。

在编写 CardStr 属性时,使用了♠、♥、◆、♣的 unicode 码,通过将枚举值转换成对应的整

数，再转换成以其为 unicode 码的字符，实现了花色字符的显示。

5.2.3　方法成员

1. 方法的定义

类的方法成员可以定义该类事物所能实现的功能。类的方法的定义要使用标准的函数格式，语法如下：

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

修饰符可以是访问限制修饰符与 virtual、abstract 和 override 的组合。用 virtual 修饰的方法可以重写；abstract 修饰的方法必须在抽象的派生类中重写(只用于抽象类中)；override 修饰的方法重写了一个基类的方法(如果方法被重写，就必须使用这个关键字)。在第 7 章中会对后面的这些修饰符进行应用。

【实验 5-5】Deck 类及其方法

内容：为 BlackJack 项目添加 Deck 类，用于描述一副牌，并设计 Deck 类的部分字段、方法和属性。

设计：

(1)Deck 类用于描述一副牌，因此可使用 Card[]类型的字段 allCards 表示所有的牌。

(2)Deck 类应提供洗牌功能，功能由 Shuffle()方法实现。

(3)在 Deck 类的构造函数中，应将字段初始化，生成完整的一副牌(21 点游戏中没有大小王，只有 52 张牌)。

(4)Deck 类应提供下一张牌的属性。

(5)Derk 类应提供一个获取当前剩余牌数的属性。

(5)为了便于 Deck 类对象的输出，编写 Show()方法，用于将每张牌输出。

(6)基于上面的设计思路，设计 Deck 类的结构如图 5-10 所示。

图 5-10　Derk 类结构

实现：

(1)创建控制台应用程序 Demo5-5，并保存到适当的位置。

(2)根据【实验 5-4】为项目添加和编写类 Card 的代码。

(3)为项目添加类 Derk,并修改类代码如下:

```
public class Derk
{
    #region 字段
    Card[] allCards;//所有的牌
    int count;
    #endregion
    #region 属性
    /// <summary>
    /// 剩余牌的数量
    /// </summary>
    public nt CardCount
    {
        get { return count; }
    }
    /// <summary>
    /// 下一张牌,用于发牌,没有下一张了,返回空
    /// </summary>
    public Card NextCard
    {
        get
        {
            if (CardCount > 0)
            {
                Card nextCard = allCards[52 - CardCount];
                count--;
                return nextCard;
            }
            else
            {
                return null;
            }
        }
    }
    #endregion

    #region 方法
    /// <summary>
    /// 创建一副牌,52 张牌
    /// </summary>
    public Derk()
    {
```

```
        allCards = new Card[52];//
        this.count = 52;
        for (int = 0; i<52; i++)
        {
            allCards[i] = new Card((CardRank)(i % 13) + 1,(CardColor)(i % 4));
        }
    }
    /// <summary>
    /// 洗牌,将本副牌乱序
    /// </summary>
    public void Shuffle()
    {
        Random rand = new Random();
        Card[] newCardList = new Card[52];
        for (int = 0; i<52; i++)
        {
            int randIndex = rand.Next(52 - );//从当前牌中随机抽一张
            //计算该牌的正确索引
            for (int j = 0; j<randIndex; j++)
            {
                if (allCards[j] = = null)
                {
                    randIndex++;
                }
            }
            newCardList[i] = allCards[randIndex]; // 将选中的牌加入到新列表中
        }
        count = 52;
        allCards = newCardList;//旧列表 = 新列表
    }
    /// <summary>
    /// 显示所有的牌
    /// </summary>
    public void  Show()
    {
        foreach (Card card n allCards)
        {
            card.Show();
            Console.WriteLine();
        }
    }
    #endregion
```

```
}
```

(4)在 Main 函数中创建一副牌的实例,并将洗牌后的结果输出,代码如下。

```
static void Main(string[] args)
{
    Derk derkA = new Derk();
    derkA.Shuffle();
    derkA.Show();
}
```

使用【Ctrl+F5】,执行程序得到的结果如图 5-11 所示。

图 5-11 Deck 类对象的创建和洗牌的结果

分析:

(1)在 NextCard 属性中,如果没有添加 if 语句,直接返回索引对应的 Card 对象,当索引值达到 52 时,就会发生索引越界的错误。为了便于捕获这一错误,这里在属性中没有直接返回值,而是增加了判断,并对越界的情况抛出了异常,使用用户可以理解的字符串说明异常发生的原因。

(2)观察类 Deck 的结构,首先是类的字段,接着是类的属性成员,最后是类的方法成员。书写类的时候应该规范,不能随意放置类的成员,将不同的成员如字段和属性混合放置,会降低应用程序的可读性。

(3)在主函数的代码中没有显式地调用方法 ToString(),却得到了该方法的结果。这是因为 WriteLine()方法默认地会调用输出对象的 ToString()方法。如果没有为 Deck 类重新定义 ToString()方法,请大家思考本实验的输出结果会是什么。

2. 方法的重载

在同一个类中,允许多个同名方法同时存在的现象,只要这些方法的参数不同。我们称这种方法名相同但参数不同的情况为方法的重载。

这里提到的参数的不同包括参数的个数、顺序,以及类型不同。需要注意的是,如果两个方法的参数相同,但返回值类型不同是不能列为重载的,编译器将判定这种情况为重复定义的语法错误。

较为常见的重载的例子是构造函数的重载。在【实验 5-5】中为 Card 类定义了重载的构造

函数。当实例化 Card 对象时，通过指定不同的参数会执行不同的构造函数。

5.3　综合实验

在这一节当中，将主要运用本章所学的知识点，完成基于面向对象的 21 点游戏。本实验将在【实验 5-5】的基础上继续进行。请读者在理解代码功能的同时，体会面向对象的思想。

【实验 5-6】面向对象的 21 点游戏

内容：同【实验 3-7】。

设计：

(1)每个游戏元素都是一个类，根据游戏的需要设计类的结构和关系如图 5-12 所示。

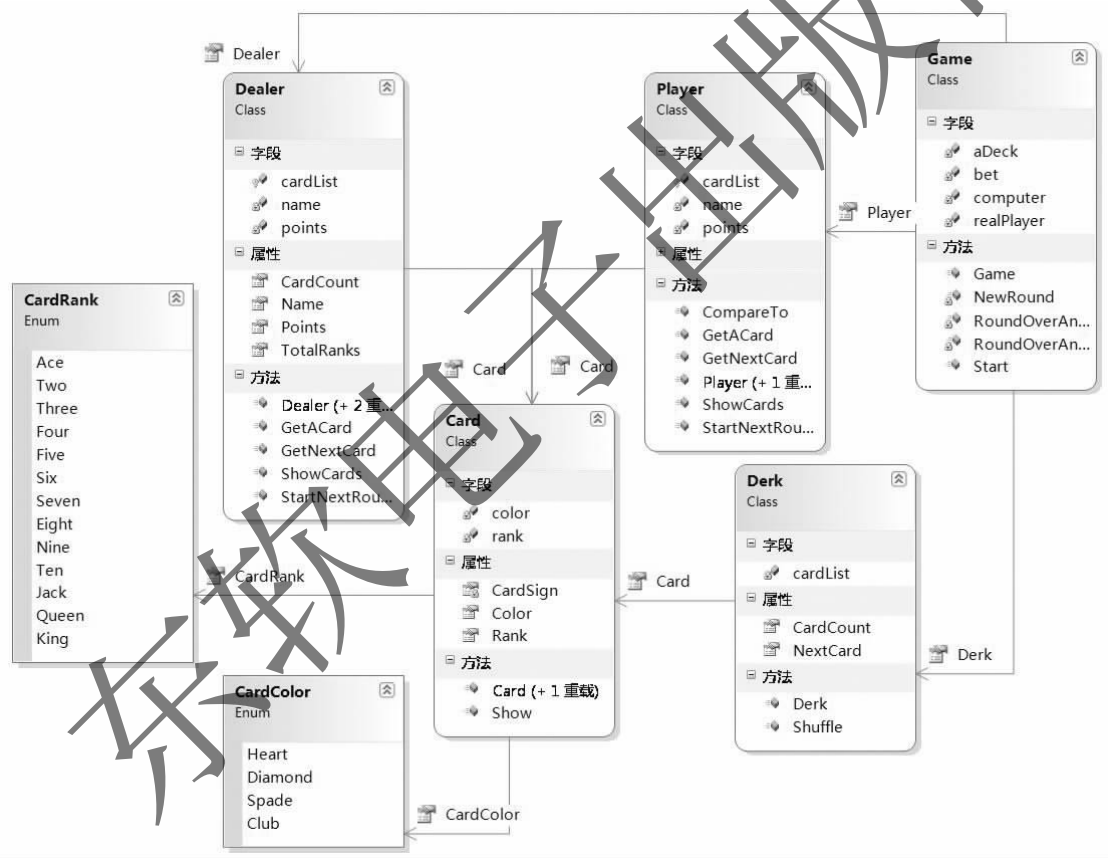

图 5-12　【实验 5-6】类关系图

(2)Card 类和 Derk 类的定义与本章前面的实验相同。

(3)这些游戏元素都作为游戏类 Game 的元素，与 Game 具有关联关系。

实现：

(1)创建控制台应用程序 Demo5-6，并保存到适当的位置。

(2)根据【实验 5-5】为应用程序添加图 5-11 中的类和枚举。

(3)根据【实验 5-5】修改 CardRank 和 CardColor 枚举的定义、类 Card 的定义，以及类 Derk

的定义。

(4)添加类 Dealer 的代码如下：

```
/// <summary>
///描述庄家的类
/// </summary>
public class Dealer
{
    #region 字段
    private string name;//玩家姓名
    protected Card[] cardList;//玩家手中的牌
    int points;//积分
    int count;//玩家手中的牌数
    #endregion

    #region 属性
    /// <summary>
    /// 玩家名字
    /// </summary>
    public string Name
    {
        get { return name; }
        protected set
        {
            name = value;


        }
        /// <summary>
        /// 获取牌的数量
        /// </summary>
        public nt CardCount
        {
            get { return count; }
        }
        /// <summary>
        /// 获取和设置积分
        /// </summary>
        public nt Points
        {
            get { return points; }
            set { points = value; }
        }
```

```
/// <summary>
/// 获庄家的点数
/// </summary>
public   int TotalRanks
{
    get
    {
        int ranks = 0;
        for (int  = 0; i<count; i + + )
        {
            Card card = cardList[i];
            if ((int)(card.Rank) < = 10)
            {
                ranks + = (int)(card.Rank);
            }
            else
            {//10 以上的牌取 10,JKQ
                ranks + = 10;
            }
        }
        for (int  = 0; i<count; i + + )
        {
            Card card = cardList[i];
            if (ranks < = 11)
            {
                if (card.Rank = = CardRank.Ace)
                {
                    ranks + = 10;
                }
            }
        }
        return ranks;
    }
}
#endregion

#region 构造函数
public Dealer()
{
    this.name = "牛牛";
    cardList = new Card[52];
    Points = 0;
```

```
}
public Dealer(string name)
{
    this.name = name;
    cardList = new Card[52];
    Points = 0;
}
public Dealer(string name,int points)
{
    this.name = name;
    this.points = points;
    cardList = new Card[52];
}
#endregion

#region 方法
/// <summary>
/// 叫牌,同时判断收到该牌后的输赢
/// </summary>
/// <param name = "nextCard">发给的下 1 张牌</param>
public nt GetACard(Card nextCard)
{
    cardList[count] = nextCard;
    count + + ;
    if (CardCount = = 2 && TotalRanks = = 21)
    {//BlackJack
    return 1;
    }
    if (TotalRanks > 21)//爆牌
    {
        return - 1;
    }
    if (CardCount = = 5 && TotalRanks < = 21)//5 张还没爆牌,赢了
    {
        return 1;
    }
    return 0;//上面情况都没发生,没有输赢
}

/// <summary>
/// 下一局游戏,牌的列表清空
/// </summary>
```

```
        public void StartNextRound()
        {
            this.cardList = new Card[5];
            count = 0;
        }

        public  bool GetNextCard()
        {
            if (TotalRanks<16)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        public void ShowCards(bool showAll)
        {
            Console.Write(this.Name + ":");
            if (showAll)
            {
                for (int = 0; i<count; i + + )
                {
                    Card card = cardList[i];
                    card.Show();//显示牌
                    Console.Write("\t");
                }
                Console.WriteLine();
            }
            else
            {
                cardList[0].Show();
                Console.WriteLine();
            }
        }
        #endregion
}
```

(5)添加类 Player 的代码如下：

```
/// <summary>
/// 描述玩家的类
/// </summary>
```

```
public class Player
{
    #region 字段
    private string name;//姓名
    protected Card[] cardList;//手中的牌
    int points;//积分
    int count;
    #endregion

    #region 属性
    /// <summary>
    /// 获取和设置玩家姓名
    /// </summary>
    public string Name
    {
        get { return name; }
        protected set
        {
            name = value;
        }
    }
    /// <summary>
    /// 获取和设置手中牌的数量
    /// </summary>
    public int CardCount
    {
        get { return count; }
        protected set
        {
            count = value;
        }
    }
    /// <summary>
    /// 获取和设置积分
    /// </summary>
    public int Points
    {
        get { return points; }
        set { points = value; }
    }
    /// <summary>
    /// 获取牌的分值
```

```
/// </summary>
protected int TotalRanks
{
    get
    {
        int ranks = 0;
        //遍历所有的牌,A 按大的不爆牌取值
        for(int i = 0;i<count;i++)
        {
            Card card = cardList[i];
            if ((int)(card.Rank )<= 10)
            {
                ranks += (int)(card.Rank);
            }
            else
            {//10 及以上取 10,JKQ
                ranks += 10;
            }
        }
        for (int i = 0; i < count; i++)
        {
            Card card = cardList[i];
            if (ranks <= 11)
            {
                if (card.Rank == CardRank.Ace)
                {
                    ranks += 10;
                }
            }
        }
        return ranks;
    }
}
#endregion


#region 构造函数
public Player()
{
    this.cardList = new Card[52];
    Points = 0;
}
```

```
public Player(string name, int points)
{
    cardList = new Card[52];
    Name = name;
    Points = points;
}
#endregion

#region 方法
/// <summary>
/// 收到下 1 张牌,判断输赢
/// </summary>
/// <param name = "nextCard">发给下 1 张牌</param>
public int GetACard(Card nextCard)
{
    cardList[count] = nextCard;
    count + + ;
    if (CardCount = = 2 && TotalRanks = = 21)
    {//BlackJack
        return 1;
    }
    if (TotalRanks>21)//爆牌
    {
        return -1;
    }
    if(CardCount = = 5&&TotalRanks< = 21)//5 张没爆,赢了
    {
        return 1;
    }
    return 0;//以上情况都没发生,没有输赢
}
/// <summary>
/// 下一局
/// </summary>
public void StartNextRound()
{
    this.cardList = new Card[5];
    count = 0;
}

/// <summary>
/// 比较玩家与庄家的点数大小
```

```
    /// </summary>
    /// <param name = "obj">庄家</param>
    /// <returns>点数比较结果</returns>
    public int CompareTo(object obj)
    {
        Dealer otherPlayer = obj as Dealer;//obj 转换成庄家
        return this.TotalRanks.CompareTo(otherPlayer.TotalRanks);
    }

    /// <summary>
    /// 询问玩家是否叫牌
    /// </summary>
    /// <returns>根据玩家选择返回是否叫牌</returns>
    public bool GetNextCard()
    {
        Console.WriteLine("是否继续叫牌? Y 继续,其他不叫");
        return (Console.ReadLine() = = "Y");
    }

    /// <summary>
    /// 显示手中每张牌
    /// </summary>
    public void ShowCards()
    {
        Console.Write(this.Name + ":");
        for (int i = 0; i < count; i + + )
        {
            Card card = cardList[i];
            card.Show();//显示牌
            Console.Write("\t");
        }
        Console.WriteLine();
    }
    #endregion
}
```

(6)添加类 Game 的代码如下：

```
/// <summary>
/// 游戏类,包括游戏过程的控制等
/// </summary>
public class Game
{
    int bet;//当前的的赌注
```

```
Player realPlayer;
Dealer computer;
Derk aDeck;
/// <summary>
/// 创建一个游戏,玩家、庄家的名字和分数初始化、一副新的牌;游戏结束标识清空
/// </summary>
public Game()
{
    computer = new Dealer("NiuNiu", 1000);
    Console.WriteLine("请输入玩家的名字");
    realPlayer = new Player(Console.ReadLine(), 1000);
    aDeck = new Derk();
    this.bet = 0;
}

public Player Player
{
    get
    {
        throw new System.NotImplementedException();
    }
    set
    {
    }
}

public Dealer Dealer
{
    get
    {
        throw new System.NotImplementedException();
    }
    set
    {
    }
}

public Derk Derk
{
    get
    {
        throw new System.NotImplementedException();
```

```
        }
        set
        {
        }
    }
    /// <summary>
    /// 开始游戏
    /// </summary>
    public void Start()
    {
        do
        {
            //游戏开始,先下注
            Console.WriteLine("请输入 Y 下注,开始游戏,每局 50 分,玩家拥有积分" + realPlayer.
Points.ToString()
                + ",庄家拥有积分" + computer.Points.ToString());
            if (Console.ReadLine() != "Y")
            {
                break;
            }
            else
            {//玩家要继续玩
                NewRound();//开局
                //玩家和庄家的积分都减掉 50
                computer.Points -= 50;
                realPlayer.Points -= 50;
                //记录当前赌注为 50
                this.bet = 50;
                //分给玩家 1 张牌
                realPlayer.GetACard(aDeck.NextCard);
                //分给庄家 1 张牌
                computer.GetACard(aDeck.NextCard);
                //再分给玩家 1 张牌
                int realPlayerWin = realPlayer.GetACard(aDeck.NextCard);
                //再分给庄家 1 张牌
                int computerResult = computer.GetACard(aDeck.NextCard);
                //庄家和玩家都显示手中的牌
                computer.ShowCards(false);
                realPlayer.ShowCards();
                while (realPlayerWin != 1 && realPlayerWin != -1)
                {//玩家没有赢,没有爆牌
                    if (realPlayer.GetNextCard())
```

```
                {  //玩家还要叫牌//再分给玩家一张牌
                    realPlayerWin  =  realPlayer.GetACard(aDeck.NextCard);
                    realPlayer.ShowCards();
                }
                else
                {
                    break;//从玩家叫牌的循环中退出,轮到庄家
                }
            }//循环结束表示玩家赢了,或者不叫牌了,或者爆牌了
            if (realPlayerWin  = =  1)
            {//玩家赢了
                if(realPlayer.CardCount = =5)//5 张牌
                {//返还赌注,并加上赢来的点数
                RoundOverAndWin(true,realPlayer.Name + "5 张牌,胜利!");
                }
                else
                {
                    RoundOverAndWin(true,realPlayer.Name + "Black Jack,胜利!");
                }
                continue;//继续询问是否继续游戏,再次下注
            }
            else if (realPlayerWin  = =  -1)//玩家爆牌了
            {
                RoundOverAndWin(false, realPlayer.Name
+ "爆牌," + computer.Name + "胜利!");//庄家返还赌注 2 倍
                continue;//继续询问是否继续游戏,再次下注
            }
            else
            {//玩家没有赢也没有输,显示庄家的牌
                computer.ShowCards(true);//显示庄家的牌
                while (computerResult ! = 1  && computerResult ! =  -1)
                {//庄家没有赢,没有爆牌
                    if (computer.GetNextCard())
                    {//庄家要叫牌//分给庄家下 1 张牌
                        computerResult  =  computer.GetACard(aDeck.NextCard);
                        computer.ShowCards(true);//显示全部牌
                    }
                    else
                    {
                        break;//从庄家叫牌的循环中退出
                    }
                }
```

```
                    if (computerResult == 1)
                    {//庄家赢了
                        if (computer.CardCount == 5)
                        {
                            RoundOverAndWin(false, computer.Name + "5 张牌,胜利!");
                        }
                        else
                        {
                        RoundOverAndWin(false, computer.Name + "拿到黑杰克,胜利!");
                        }
                        continue;
                    }
                    else if (computerResult == -1)
                    {//庄家输了
                        RoundOverAndWin(true,computer.Name
+ "爆牌," + realPlayer.Name + "胜利!");
                        continue;
                    }
                    else
                    {//都没有爆牌,也没有赢
                        if (realPlayer.CompareTo(computer) < 0)
                        {//玩家输了
                            RoundOverAndWin(false,realPlayer.Name
+ "点数小," + computer.Name + "胜利!");
                            continue;
                        }
                        else if (realPlayer.CompareTo(computer) > 0)
                        {//玩家赢了
                            RoundOverAndWin(true, realPlayer.Name + "点数大,胜利!");
                            continue;
                        }
                        else
                        {//没有输赢
                            RoundOverAndLicense();
                            continue;
                        }
                    }
                }
            }
        } while (computer.Points > 50 && realPlayer.Points > 50);
        Console.WriteLine("游戏结束");
    }
```

```
        /// <summary>
        /// 一局结束,返还赢的一方双倍赌注
        /// </summary>
        /// <param name = "player">赢家是玩家</param>
        private void RoundOverAndWin(bool playerWon,string msg)
        {
            if (playerWon)
            {
                this.realPlayer.Points += bet * 2;
            }
            else
            {
                this.computer.Points += bet * 2;
            }
            Console.WriteLine(msg);
            realPlayer.ShowCards();
            computer.ShowCards(true);
            Console.WriteLine(realPlayer.Name + "积分:" + realPlayer.Points.ToString());
            Console.WriteLine(computer.Name + "积分:" + computer.Points.ToString());
            NewRound();
        }
        /// <summary>
        /// 一局结束,和牌;返还双方赌注
        /// </summary>
        private void RoundOverAndLicense()
        {
            realPlayer.Points += bet;
            computer.Points += bet;
        }
        /// <summary>
        /// 为下一局准备
        /// </summary>
        private void NewRound()
        {
            realPlayer.StartNextRound();//下一局
            computer.StartNextRound();//下一局
            aDeck = new Derk();//准备牌
            aDeck.Shuffle();//洗牌
            bet = 0;//赌注清空
        }
    }
```

(7)在 Main()函数中添加如下代码:

```
static void Main(string[] args)
{
    Game blackJack = new Game();
    blackJack.Start();
}
```

(8)使用【Ctrl＋F5】执行程序,可以得到如图 5-13 所示的结果。

图 5-13　【实验 5-6】的一次执行结果

分析:

(1)在第 3 章的代码中,我们使用面向过程的思想模拟了 21 点游戏,但在该游戏中只有点数,没有牌的花色、级别等信息。当应用程序描述的信息量较小、过程控制简单时,使用面向过程的程序设计方法比较适合,第 3 章的实验代码主要集中在一个函数 Main()的内部来实现。

(2)在本章中,我们使用类描述所有的游戏元素,使游戏的代码实现过程与实际情况非常一致。实际游戏中的牌、玩家、庄家等在本实验中均使用相应的类进行了描述;游戏元素的特征,使用字段和属性来描述,游戏元素的功能使用方法来描述。

(3)这样,我们看到 Main()函数中的代码非常简单,只是创建了一个游戏对象,并启动游戏,这与一个真正的游戏是一样的。

(4)实际上在类的内部,当编写实现功能的方法时,我们采用的仍然是面向过程的思想。因此,可以认为面向对象的程序设计是面向过程程序设计的一个封装,将变量和方法封装到一起来模拟现实世界中的对象。