

Лекция 11 ИСПОЛЬЗОВАНИЕ ИНТЕРФЕЙСОВ

11.1 Понятие интерфейса

Слово "интерфейс" - многозначное, и в разных контекстах оно имеет различный смысл. Существует понятие программного или аппаратного интерфейса, но в большинстве случаев слово интерфейс ассоциируется с некоторой связью между объектами или процессами. В данной лекции речь идет о понятии интерфейса, стоящем за ключевым словом **interface**. В таком понимании интерфейс - это частный случай класса.

Интерфейс представляет собой полностью абстрактный класс, все методы которого абстрактны.

От абстрактного класса интерфейс отличается некоторыми деталями в синтаксисе и поведении.

Синтаксическое отличие состоит в том, что методы интерфейса объявляются без указания модификатора доступа.

Отличие в поведении заключается в более жестких требованиях к потомкам. Класс, наследующий интерфейс (интерфейсный класс), обязан полностью реализовать все методы интерфейса. В этом отличие от класса, наследующего абстрактный класс, где потомок может реализовать лишь некоторые методы родительского абстрактного класса, оставаясь абстрактным классом.

Важное отличие интерфейсного класса от обычного класса заключается в том, что он может наследовать несколько родительских интерфейсов. Таким образом, в C# разрешено множественное наследование, но только в интерфейсных классах.

Родительские интерфейсы перечисляются в списке за именем класса и двоеточием:

```
public interface INewClass: IInt1, IInt2, ..., IIntN  
{ ... }
```

Такого рода интерфейсные классы обязаны содержать реализации всех методов всех родительских интерфейсов.

Замечу, что интерфейсный класс может наследовать не только от интерфейсов, но и от одного (и только одного!) обычного класса, по отношению к которому он ведет себя как обычный наследник, то есть может переопределять его методы, добавлять поля и т. д.

Множественное наследие потенциально связано с возможностью конфликта имен и наличием общего родителя. Конфликт имен проявляется в том, что разные родительские интерфейсы могут содержать одноименные методы с одинаковым синтаксисом.

Поскольку интерфейсный класс обязан реализовывать все методы своих родительских интерфейсов, возникает коллизия, которую можно разрешить одним из следующих способов.

Склеивание методов. В этом случае интерфейсный класс полагает, что у всех одноименных методов должна быть одинаковая программная

реализация, и объявляет этот единственный метод для реализации всех одноименных методов своих родителей.

Переименование методов. Если реализация одноименных методов должна быть различной, методы переименовываются.

Отметим еще одно важное назначение интерфейсов, отличающее их от абстрактных классов. Абстрактный класс представляет собой начальный этап проектирования класса, который в будущем получит конкретную реализацию. Интерфейсы задают дополнительные свойства классу. Каждый интерфейс наделяет класс тем или иным новым свойством.

11.2 Синтаксис интерфейса

Общее описание интерфейса, включающее необязательные элементы (они выделены квадратными скобками), имеет следующий формат записи:

**[атрибуты] [спецификаторы] interface имя_класса [: родители]
{ тело_класса } где,**

атрибуты – задают дополнительную информацию о классе;

спецификаторы – обычно определяют условие доступа к составляющим класса;

родители – родительские интерфейсные классы, которые наследует наш класс:

тело класса – определяет состав интерфейсного класса.

Если внимательно посмотреть на формат записи интерфейса, то можно заметить, что его формат очень похож на формат записи обычного класса. Это объясняется тем, что интерфейс – частный случай класса.

В библиотеке платформы .NET имеется большое число интерфейсов, наследуя которые, классы получают дополнительные свойства.

Например, интерфейс `IComparable` задает метод сравнения объектов по принципу больше или меньше, что позволяет выполнять их сортировку.

Реализация интерфейсов `IEnumerable` и `IEnumerator` дает возможность просматривать (перебирать) содержимое объекта с помощью конструкции `foreach`, а реализация интерфейса `ICloneable` — клонировать объекты.

Каждый интерфейс наделяет класс теми или иными новыми возможностями. В этом смысле поле для разработки новых интерфейсов практически бесконечно.

Например, можно разработать интерфейс для продажи – покупки валюты в соответствии с текущим курсом, интерфейсы для различных начислений коммунальных услуг с учетом льгот и т.д.

В качестве учебного примера опишем интерфейс, реализация методов которого позволит классу проводить некоторые преобразования над музыкальной записью – преобразуя 7 нот и паузу в цифры от 0 до 7 и выполнять обратные преобразования.

Как и любой учебный пример, он немного искусственный, поскольку наша главная задача сейчас состоит в том, чтобы рассмотреть технологию создания и использования интерфейсов.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        interface ITextNoti
        {
            string Codirovanie();
            string Decodirovanie();
        }

        class MyzikText: ITextNoti
        {
            string text;
            static string[] codeTable =
            {
                "до", "ре", "ми", "фа", "соль", "ля", "си", "пауза"
            };

            //Конструктор
            public MyzikText(string txt)
            {
                text = txt;
            }

            //Реализация интерфейсов
            public string Codirovanie()
            {
                Boolean ok;
                string rez = "";
                string[] noti;
                // преобразование к нижнему регистру
                string text1 = text.ToLower();
                //размерность массивов noti устанавливается
                // автоматически в соответствии с размерностью массива,
                //возвращаемого методом Split
                noti = text1.Split(' ');
                for (int i = 0; i < noti.Length; i++)
                {
                    ok = false;
                    for (int j = 0; j < 8; j++)
```

```

        if (noti[i] == codeTable[j])
        { ok = true; rez = rez + " " + j.ToString(); }
        if (ok == false) rez = rez + " ?";
    }
    return rez;
}
// дешифровка поля text
// с использованием таблицы нот
public string Decodirovanie()
{
    Boolean ok;
    string rez = "";
    string[] noti;
    // преобразование к нижнему регистру
    string text1 = text.ToLower();
    noti = text1.Split(' ');
    for (int i = 0; i < noti.Length; i++)
    {
        ok = false;
        for (int j = 0; j < 8; j++)
            if (Convert.ToInt32(noti[i]) == j)
                { ok = true; rez = rez + " " + codeTable[j]; }
            if (ok == false) rez = rez + " ?";
        }
    return rez;
}
}
public Form1()
{
    InitializeComponent();
}

private void button1_Click(object sender, EventArgs e)
{
    string a,b;
    a = textBox1.Text;
    MyzikText IcxodText = new MyzikText(a);
    b = IcxodText.Codirovanie();
    textBox3.AppendText(b + "\r\n");
}

private void button2_Click(object sender, EventArgs e)
{
    string a, b;
    a = textBox2.Text;
    MyzikText IcxodText1 = new MyzikText(a);
    b = IcxodText1.Decodirovanie();
    textBox3.AppendText(b + "\r\n");
}
}
}

```

Объявляем интерфейс `ItextNoti`, содержащий два метода – кодирование (словесный текст нот заменяется цифрами от 0 до 7) и декодирование (текст, представленный цифрами от 0 до 7, заменяется словесным текстом нот).

```
interface ITextNoti
{
    string Codirowanie();
    string Decodirowanie();
}
```

Далее объявляем интерфейсный класс, наследующий интерфейс и реализующий его методы. Выполняем общедоступную реализацию методов интерфейса. Алгоритм реализации интерфейсных методов прокомментирован в коде программы и не нуждается в дополнительных пояснениях.

Работа программы изображена на рисунке 11.1.

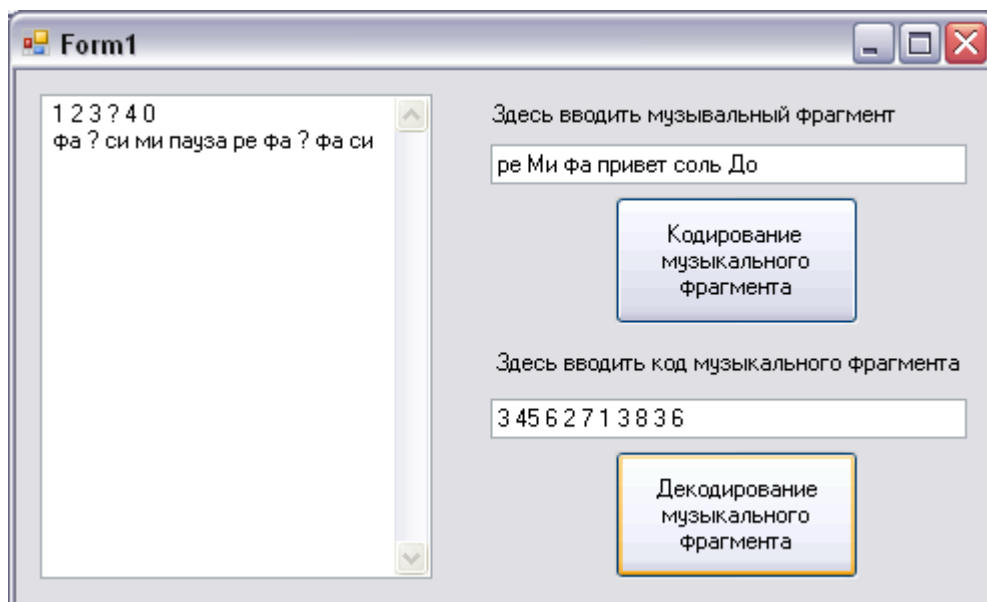


Рисунок 11.1 – Использование интерфейсного класса

В приведенном примере показана технология создания и использование интерфейса и интерфейсного класса.

11.3 Использование стандартного интерфейса `IEnumerable`

На первый взгляд преимуществ во введении интерфейсного класса нет – методы кодирования и декодирования можно разместить непосредственно в классе `MyzikText`.

Реально существующие в библиотеке платформы **.NET** классы включают большое число интерфейсных методов различных интерфейсов,

наследуя которые, классы получают дополнительные свойства – через название методов, а не через их реализацию. Реализацию интерфейсных методов каждый интерфейсный класс, как правило, должен выполнять самостоятельно. Например, если в нашем классе необходимо организовать просмотр с помощью цикла **foreach** некоторых перечисляемых объектов представленных массивом, то наш класс должен быть наследником интерфейса **IEnumerable** (перечислимый). У этого интерфейса всего один метод **GetEnumerator()**, возвращающий объект типа **Enumerator** (перечислитель). Формат записи метода **GetEnumerator()** имеет следующий вид:

```
IEnumerator GetEnumerator();
```

Таким образом, наш класс должен быть наследником интерфейсов **IEnumerable** и **IEnumerator**.

Интерфейса **IEnumerator** включает одно свойство `Object Current{get;}`, возвращающее очередной перечисляемый объект, и два метода – `bool MoveNext()`, передвигающий перечислитель на следующий перечисляемый объект, и метод `void Reset()`, устанавливающий перечислитель на первый перечисляемый объект.

В совокупности именно указанное свойство и эти два метода позволяют организовать процесс просмотра объектов массивов с помощью цикла **foreach**. Методы этих интерфейсов работают с виртуальной коллекцией (набором объектов определяемых в процессе работы программы), что и определяет их универсальность.

Если в классе необходимо выполнять сравнение объектов, например, при их сортировке, то такой класс следует объявить наследником интерфейса **IComparable**. Этот интерфейс имеет всего один метод `CompareTo(object obj)`, возвращающий целочисленное значение, положительное, отрицательное или равное нулю, в зависимости от выполнения отношения "больше", "меньше" или "равно".

Рассмотрим работу с интерфейсами **IEnumerable** и **IEnumerator** на учебном примере, в котором необходимо организовать просмотр товаров некоторого магазина. Для простоты считаем, что класс **Tovar** имеет два поля название товара и его цена.

```
class Tovar
{
    public string Naz; // Название и цена товара
    public int Cena;
    public Tovar(string n, int c) // Конструктор товара
    {
        Naz = n;
        Cena = c;
    }
}
```

Для хранения объектов типа **Tovar** используем класс **Cklad**, имеющий следующую структуру:

```
class Cklad
{
```

```

        public Tovar[] tovar;    // Массив товаров
        public Cklad()    // Конструктор склада
        {
            tovar = new Tovar[4];
        }
    }

```

Максимальное количество объектов, которое может храниться на складе, в учебных целях принято равным 4.

Рассмотрим работу программы без использования интерфейса **IEnumerable**.

Код программы:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace WindowsFormsApplication1
{
    public partial class Form1 : Form
    {
        public static string s;
        public static int kol;
        class Tovar
        {
            public string Naz; // Название и цена товара
            public int Cena;
            public Tovar(string n, int c) // Конструктор товара
            {
                Naz = n;
                Cena = c;
            }
        }
        class Cklad
        {
            public Tovar[] tovar;    // Массив товаров
            public Cklad()    // Конструктор склада
            {
                tovar = new Tovar[4];
            }
        }
        public Form1()
        {
            InitializeComponent();
            kol = 0;
            s = "";
        }
        Cklad ck1 = new Cklad();
    }
}

```

```

private void button2_Click(object sender, EventArgs e)
{
    if (kol < 4)
    {
        ckl.tovar[kol] = new Tovar(textBox1.Text,
            Convert.ToInt32(textBox2.Text));
        s = s + textBox1.Text + textBox2.Text + "\r\n";
    }
    else { s = s + "СКЛАД ПОЛНИЙ" + "\r\n"; kol--; }
    kol++;
    textBox3.Text = s;
}
private void button1_Click(object sender, EventArgs e)
{
    s = "";
    s = "Работает цикл foreach" + "\r\n";
    foreach (Tovar t in ckl.tovar)
    {
        s = s + t.Naz + "    " + t.Cena.ToString() + "\r\n";
    }
    s = s + "Работает цикл for" + "\r\n";
    for (int i = 0; i < kol; i++)
    {
        s = s + ckl.tovar[i].Naz + "    " +
            ckl.tovar[i].Cena.ToString() + "\r\n";
    }
    textBox3.Text = s;
}
}
}

```

Работа программы

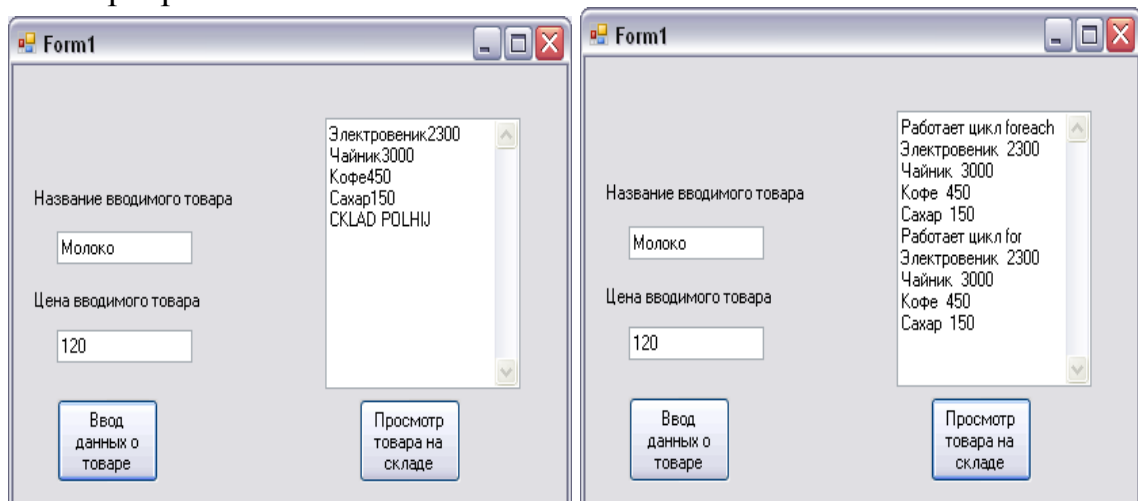


Рисунок 11.2 – Работа программы без интерфейсов

Необходимо отметить, что в программе цикл **foreach** используется только для переменной типа массив `ckl.tovar`, который уже имеет встроенный интерфейс – все классы массивы, независимо от типа элементов,

реализуют перечисление элементов массива, и для них определен метод **GetEnumerator**.

Однако, если мы попытаемся использовать цикл **foreach** для объектов класса **Tovar** в объекте **ckl** типа **Cklad**, а не для массива **tovar** объекта класса **Cklad**, например,

```
foreach (Tovar t in ckl)
{
    s = s + t.Naz + "  " + t.Cena.ToString() + "\r\n";
}
```

то программа выдаст сообщение об ошибке:

«foreach statement cannot operate on variables of type 'WindowsFormsApplication1.Form1.Cklad' because 'Books' does not contain a public definition for 'GetEnumerator'»

(Оператор **foreach** не может применяться к переменным типа 'WindowsFormsApplication1.Form1.Cklad', так как переменные этого класса не содержат открытого определения метода '**GetEnumerator**').

Доопределим нашу программу необходимым интерфейсом, для этого в нашей программе необходимо добавить дополнительное пространство имен:

```
using System.Collections;
```

Класс **Cklad** должен наследовать интерфейс **IEnumerable**:

```
class Cklad : IEnumerable
```

В тело класса **Cklad** необходимо включить реализацию метода **GetEnumerator**:

```
public IEnumerator GetEnumerator()
{
    for (int i = 0; i < 4; i++) yield return tovar[i];
}
```

Необходимы некоторые комментарии, которые взяты из книги Т.А.Павловской (стр. 207.)

«Таким образом, если требуется, чтобы для перебора элементов класса мог применяться цикл **foreach**, необходимо реализовать четыре метода: **GetEnumerator**, **Current**, **MoveNext** и **Reset**. Например, если внутренние элементы класса организованы в массив, потребуется описать закрытое поле класса, хранящее текущий индекс в массиве, в методе **MoveNext** задавать изменение этого индекса на 1 с проверкой выхода за границу массива, в методе **Current** – возврат элемента массива по текущему индексу и т.д.

Это не интересная работа, а выполнять ее приходится часто, поэтому в версии 2.0 были введены средства, облегчающие выполнение перебора в объекте – итераторы.

Итератор представляет собой блок кода, задающий последовательность перебора элементов объекта. На каждом проходе цикла **foreach** выполняется один шаг итератора, заканчивающийся выдачей очередного значения. Выдача значения выполняется с помощью ключевого слова **yield**.

...

Все, что требуется сделать в версии 2.0 для поддержки перебора — указать, что класс реализует интерфейс **IEnumerable**, и описать итератор.

Доступ к нему может быть осуществлен через методы **MoveNext** и **Current** интерфейса **IEnumerator**. За кодом, приведенным в листинге итератора, стоит большая внутренняя работа компилятора.

На каждом шаге цикла **foreach** для итератора создается «оболочка» — служебный объект, который запоминает текущее состояние итератора и выполняет все необходимое для доступа к просматриваемым элементам объекта. Иными словами, код, составляющий итератор, не выполняется так, как он выглядит — в виде непрерывной последовательности, а разбит на отдельные итерации, между которыми состояние итератора сохраняется.».

Приведенные два небольших изменения в программе позволяют использовать цикл **foreach**, который ранее выдавал сообщение об ошибке.

Необходимо отметить особенность работы цикла **foreach**, которая может приводить к «зависанию» программ. Если в нашей программе, после ввода нескольких объектов товара, но не до полного заполнения массива, мы включим режим просмотра товаров на складе, то программа «повиснет» при попытке вывода несуществующих значений — необходимо контролировать «перебираемые» значения цикла **foreach**.

Цикл **for** не имеет этих недостатков, потому что его конечное значение определяется текущим значением глобальной переменной **kol**.

```
for (int i = 0; i < kol; i++)
{
    s = s + ckl.tovar[i].Naz + "  " +
        ckl.tovar[i].Cena.ToString() + "\r\n";
}
```

11.4 Некоторые интерфейсы объединения объектов

Рассмотренный интерфейс позволяет организовать перебор элементов в объектах некоторых классов, но сам процесс объединения этих элементов (добавления, удаления, сортировки и т.д.) обычно выполняется другими интерфейсами — интерфейсами коллекций и списков, которые будут рассмотрены в следующей лекции.