**Интерфейсы**

***Интерфейс (interface)*** представляет собой не более чем просто именованный набор абстрактных членов. **Абстрактные методы не имеют никакой стандартной реализации**. Интерфейс выражает поведение, которое данный класс или структура может избрать для поддержки. Более того, каждый класс (или структура) может поддерживать столько интерфейсов, сколько необходимо, и, следовательно, тем самым поддерживать множество поведений.

В интерфейсе ни у одного из методов не должно быть тела. Это означает, что в интерфейсе вообще не предоставляется никакой реализации. В нем указывается только, что именно следует делать, но не как это делать. Интерфейс может быть реализован в любом количестве классов. Кроме того, в одном классе может быть реализовано любое количество интерфейсов.

Для реализации интерфейса в классе должны быть определены тела (т.е. конкретные реализации) методов, описанных в этом интерфейсе. Каждому классу предоставляется полная свобода для определения деталей своей собственной реализации интерфейса.

Один и тот же интерфейс может быть реализован в двух классах по-разному. Тем не менее, в каждом из них должен поддерживаться один и тот же набор методов данного интерфейса. А в том коде, где известен такой интерфейс, могут использоваться объекты любого из этих двух классов, поскольку интерфейс для всех этих объектов остается одинаковым. Благодаря поддержке интерфейсов в C# может быть в полной мере реализован **главный принцип полиморфизма: *один интерфейс — множество методов***.

Интерфейсы объявляются с помощью ключевого слова **interface**.

Интерфейс содержит только сигнатуры методов, свойств, событий или индексаторов. Класс или структура, которые реализуют интерфейс, должны реализовать члены этого интерфейса, указанные в его определении. В следующем примере класс ImplementationClass должен реализовать метод с именем SampleMethod , который не имеет параметров и возвращает void.

В объявлении методов интерфейса используются только их возвращаемый\_тип и сигнатура. Они, по существу, являются абстрактными методами. В интерфейсе не может быть никакой реализации. Поэтому все методы интерфейса должны быть реализованы в каждом классе, включающем в себя этот интерфейс. **В самом же интерфейсе методы неявно считаются открытыми, поэтому доступ к ним не нужно указывать явно.**

interface ISampleInterface //*объявление интерфейса*

{

void SampleMethod();

}

class ImplementationClass : ISampleInterface //*синтаксис implementation.*

{

// *Явная реализация интерфейса:*

void ISampleInterface.SampleMethod()

{

// *Method implementation*.

}

static void Main()

{

// *Declare an interface instance*.

ISampleInterface obj = new ImplementationClass();

// *Call the member*.

obj.SampleMethod();

}

}

**Помимо методов, в интерфейсах можно также указывать свойства, индексаторы и события. Интерфейсы не могут содержать члены данных. В них нельзя также определить конструкторы, деструкторы или операторные методы. Кроме того, ни один из членов интерфейса не может быть объявлен как static.**

Как только интерфейс будет определен, он может быть реализован в одном или нескольких классах. Для реализации интерфейса достаточно указать его имя после имени класса, аналогично базовому классу. Ниже приведена общая форма реализации интерфейса в классе:

*class имя\_класса : имя\_интерфейса {*

*// тело класса*

*}*

где имя\_интерфейса — это конкретное имя реализуемого интерфейса. Если уж интерфейс реализуется в классе, то это должно быть сделано полностью. В частности, реализовать интерфейс выборочно и только по частям нельзя.

В классе допускается реализовывать несколько интерфейсов. В этом случае все реализуемые в классе интерфейсы указываются списком через запятую. В классе можно наследовать базовый класс и в тоже время реализовать один или более интерфейс. В таком случае имя базового класса должно быть указано перед списком интерфейсов, разделяемых запятой.

**Методы, реализующие интерфейс, должны быть объявлены как public.** Дело в том, что **в самом интерфейсе эти методы неявно подразумеваются как открытые, поэтому их реализация также должна быть открытой.** Кроме того, возвращаемый тип и сигнатура реализуемого метода должны точно соответствовать возвращаемому типу и сигнатуре, указанным в определении интерфейса.

Используя механизм наследования, мы можем дополнять и переопределять общий функционал базовых классах в классах-наследниках. Однако напрямую мы можем наследовать только от одного класса, в отличие, например, от языка С++, где имеется множественное наследование.

В языке C# подобную проблему позволяют решить интерфейсы. Они играют важную роль в системе ООП. Интерфейсы позволяют определить некоторый функционал, не имеющий конкретной реализации. Затем этот функционал реализуют классы, применяющие данные интерфейсы.

Для определения интерфейса используется ключевое слово interface. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования. Интерфейсы также, как и классы, могут содержать свойства, методы и события, только без конкретной реализации.

Определим следующий интерфейс Interface1, который будет содержать методы и свойства для работы со счетом клиента. Для добавления интефрейса в проект можно нажать правой кнопкой мыши на проект и в появившемся контекстном меню выбрать Add-> New Item и в диалоговом окне добавления нового компонента выбрать Interface (в различных версиях последовательность действий может отличаться, например в 10-й можно : нажать на проект, выбрать добавить класс, затем выбрать интерфейс получим пустую заготовку:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Interfase\_01

{

interface Interface1

{

}

}

Рассмотрим пример:

Определим следующий интерфейс Interface1, который будет содержать методы и свойства для работы со счетом клиента. Добавим его след. образом:

/\*using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

\* \*/

namespace Interfase\_01

{

interface Interface1

{

// Текущая сумма на счету

int CurrentSum { get; }

// Положить деньги на счет

void Put(int sum);

// Взять со счета

void Withdraw(int sum);

// Процент начислений

int Percentage { get; }

}

}

Сущность данного интерфейса проста: он определяет два свойства для текущей суммы денег на счете и ставки процента по вкладам и два метода для добавления денег на счет и изъятия денег.

Еще один момент в объявлении интерфейса: все его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Применение интерфейса аналогично наследованию класса, поэтому создадим программный код:

Реализуем интерфейс в классе Client, так как клиент у нас обладает счетом:

class Client : Interfase1

{

    int \_sum; // Переменная для хранения суммы

    int \_percentage; // Переменная для хранения процента

    public string Name { get; set; }

    public Client(string name, int sum, int percentage)

    {

        Name = name;

        \_sum = sum;

        \_percentage = percentage;

    }

    public int CurrentSum

    {

        get { return \_sum; }

    }

 public void Put(int sum) //*положить сумму*

    {

        \_sum += sum;

    }

     public void Withdraw(int sum) *//снять сумму*

    {

        if (sum <= \_sum) //*большую сумму чем на счету не снимешь*

        {

            \_sum -= sum;

        }

    }

    public int Percentage

    {

        get { return \_percentage; }

    }

    public void Display()

    {

        Console.WriteLine("Клиент " + Name + " имеет счет на сумму " + \_sum);

    }

}

|  |  |
| --- | --- |
|  |  |

Как и в случае с абстрактными методами абстрактного классом класс Client реализует все методы интерфейса. При этом поскольку все методы и свойства интерфейса являются публичными, при реализации этих методов и свойств в классе к ним можно применять только модификатор public. Поэтому если класс должен иметь метод с каким-то другим модификатором, например, protected, то интерфейс не подходит для определения подобного метода.

Применение класса в программе ( *для ф-ии main()):*

Client client = new Client("Tom", 200, 10); //объект класса реализации

client.Put(30);

Console.WriteLine(client.CurrentSum); //230

client.Withdraw(100);

Console.WriteLine(client.CurrentSum); //130

|  |  |
| --- | --- |
|  |  |

При применении этого интерфейса **класс Client должен будет реализовать как методы и свойства интерфейса IDepositAccount, так и методы и свойства базового интерфейса** Interface1.

### Соберем код в пространстве имен Interfase\_01, получим:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Interfase\_01

{ interface Interface1

{

// Текущая сумма на счету

int CurrentSum { get; }

// Положить деньги на счет

void Put(int sum);

// Взять со счета

void Withdraw(int sum);

// Процент начислений

int Percentage { get; }

}

class Client : Interface1

{

int \_sum; // Переменная для хранения суммы

int \_percentage; // Переменная для хранения процента

public string Name { get; set; }

public Client(string name, int sum, int percentage)

{

Name = name;

\_sum = sum;

\_percentage = percentage;

}

public int CurrentSum

{

get { return \_sum; }

}

public void Put(int sum)

{

\_sum += sum;

}

public void Withdraw(int sum)

{

if (sum <= \_sum)

{

\_sum -= sum;

}

}

public int Percentage

{

get { return \_percentage; }

}

public void Display()

{

Console.WriteLine("Клиент " + Name + " имеет счет на сумму " + \_sum);

}

}

class Program

{

static void Main(string[] args)

{

Client client = new Client("Tom", 200, 10);

client.Put(30);

Console.WriteLine(client.CurrentSum); //230

client.Withdraw(100);

Console.WriteLine(client.CurrentSum); //130

//чтобы воспользоваться вызовом метода класса, через интерфейсную ссылку

//необходимо преобразовать ее к типу Client, подключить нижний заком. блок

/\* Interface1 client1 = new Client("Том", 200, 10);

client1.Put(200);

Console.WriteLine(client1.CurrentSum); // 400

// Интерфейс не имеет метода Display, необходимо явное приведение

((Client)client1).Display();

\*/

}

}

}

Выполним проект, получим:

### 

**Интерфейсы, как и классы, могут наследоваться**:

interface IDepositAccount : Interface1{

    void GetIncome(); // начисление процентов

}

|  |  |
| --- | --- |
|  |  |

### Интерфейсы в преобразованиях типов

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Поскольку класс Client реализует интерфейс Interface1, то **переменная типа** Interface1 **может хранить ссылку на объект типа Client**:

IAccount client1 = new Client("Том", 200, 10);

client1.Put(200);

Console.WriteLine(client1.CurrentSum); // 400

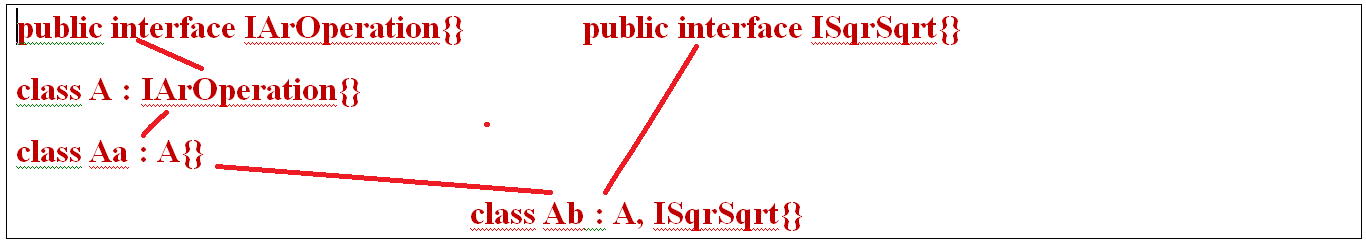
// Интерфейс не имеет метода Display, необходимо явное приведение

((Client)client1).Display();

|  |  |
| --- | --- |
|  |  |

И если мы хотим обратиться к методам класса Client, которые не определены в интерфейсе Interface1, а определены в самом классе Client или в его базовом классе (как, например, Display()), то нам **надо явным образом выполнить преобразование типов: ((Client)client1).Display();**

В следующем примере определим следующую структуру:



using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication6

{

// Создаем два интерфейса, описывающих абстрактные методы

// арифметических операций и операций Sqrt и Sqr

public interface IArOperation

{

// Определяем набор абстрактных методов

int Sum();

int Otr();

int Prz();

int Del();

}

public interface ISqrSqrt

{

int Sqr(int x);

int Sqrt(int x);

}

// Данный класс реализует интерфейс IArOperation

class A : IArOperation

{

int My\_x, My\_y;

public int x

{

set { My\_x = value; }

get { return My\_x; }

}

public int y

{

set { My\_y = value; }

get { return My\_y; }

}

public A() { } //конструктор по умолчанию

public A(int x, int y) //конструктор

{

this.x = x;

this.y = y;

}

// Реализуем методы интерфейса

public virtual int Sum()

{

return x + y;

}

public int Otr()

{

return x - y;

}

public int Prz()

{

return x \* y;

}

public int Del()

{

return x / y;

}

// В данном классе так же можно реализовать собственные методы

public virtual void rewrite()

{

Console.WriteLine("Переменная x: {0}\nПеременная y: {1}",x,y);

}

}

// Следующий класс унаследован от класса А, но при этом в нем не нужно

// заново реализовывать интерфейс, но при этом можно переопределить

// некоторые его методы

class Aa : A

{

public int z;

public Aa(int z, int x, int y)

: base(x, y)

{

this.z = z;

}

// Переопределим метод Sum

public override int Sum()

{

return base.x + base.y + z; //добавим в сумму знач. z

}

public override void rewrite()

{

base.rewrite(); //вызываем метод базового класса

Console.WriteLine("Переменная z: " + z); //и дополняем его

}

}

// Данный класс унаследован от класса А, и при этом

// реализует интерфейс ISqrSqrt

class Ab : A, ISqrSqrt

{

public int Sqr(int x)

{

return x \* x;

}

public int Sqrt(int x)

{

return (int)Math.Sqrt((double)(x));

}

}

class Program

{

static void Main()

{

A obj1 = new A(x: 10, y: 12);

Console.WriteLine("obj1: ");

obj1.rewrite();

Console.WriteLine("{0} + {1} = {2}",obj1.x,obj1.y,obj1.Sum());

Console.WriteLine("{0} \* {1} = {2}", obj1.x, obj1.y, obj1.Prz());

Aa obj2 = new Aa(z: -3, x: 10, y: 14);

Console.WriteLine("\nobj2: ");

obj2.rewrite();

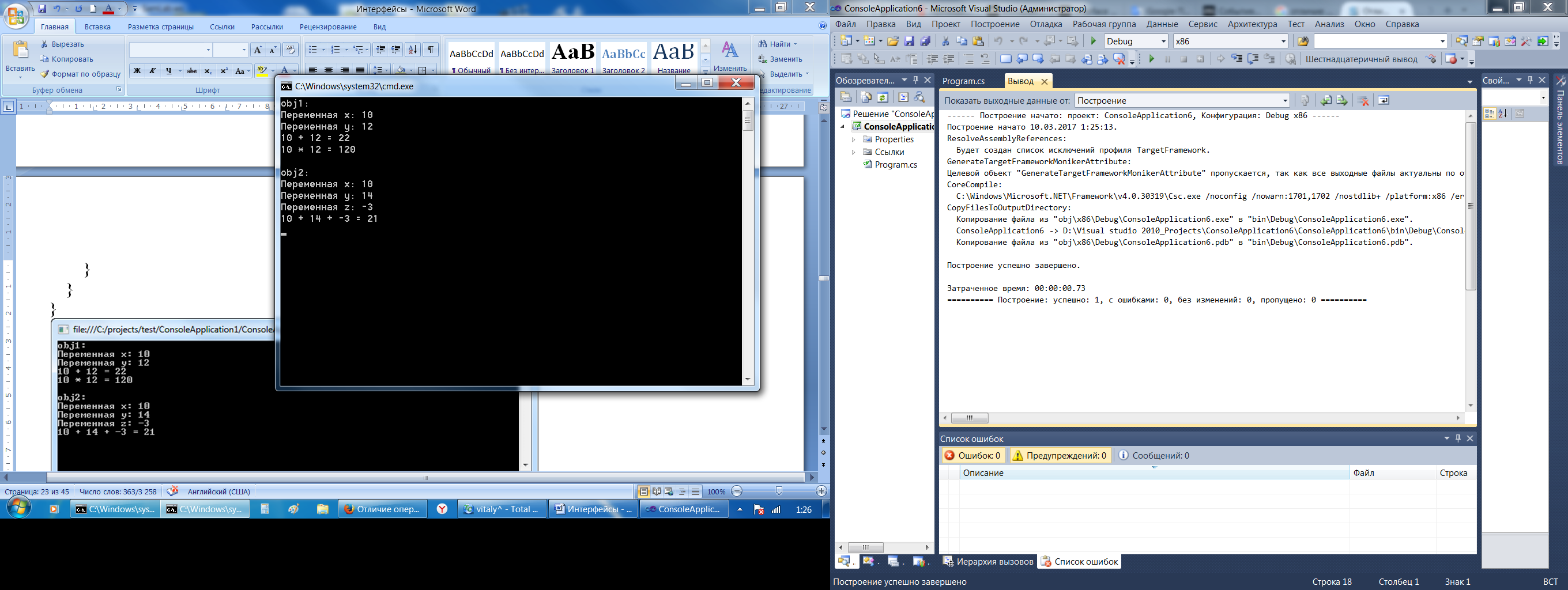
Console.WriteLine("{0} + {1} + {3} = {2}", obj2.x, obj2.y, obj2.Sum(), obj2.z);

Console.ReadLine();

}

}

}



# Интерфейсные ссылки

В C# допускается объявлять **переменные ссылочного интерфейсного типа**, т.е. переменные **ссылки на интерфейс**. **Такая переменная может ссылаться на любой объект, реализующий ее интерфейс**. При вызове метода для объекта посредством интерфейсной ссылки **выполняется его вариант, реализованный в классе данного объекта**. Этот процесс аналогичен применению ссылки на базовый класс для доступа к объекту производного класса.

**Переменной ссылки на интерфейс доступны только методы, объявленные в ее интерфейсе.** Поэтому интерфейсную ссылку нельзя использовать для доступа к любым другим переменным и методам, которые не поддерживаются объектом класса, реализующего данный интерфейс.

ing System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Interfase\_\_03

{

public interface IInfo

{

void uiName();

void uiFamily();

void uiAge();

}

class UI : IInfo

{

string Name, Family;

int Age;

public UI(string Name, string Family, int Age)

{

this.Name = Name;

this.Family = Family;

this.Age = Age;

}

// Реализуем интерфейс

public void uiName()

{

Console.WriteLine("Имя пользователя: " + Name);

}

public void uiFamily()

{

Console.WriteLine("Фамилия: " + Family);

}

public void uiAge()

{

Console.WriteLine("Возраст: " + Age);

}

// Собственный метод класса UI

public void allInfo()

{

Console.WriteLine("\n"+ Name + " " + Family + " " + Age);

}

}

class Program

{

static void Main()

{

UI ui1 = new UI(Name: "Иван", Family: "Иванов", Age: 30);

// Создадим ссылку на интерфейс

IInfo obj;

//Используем ссылку на объект ui1

obj = ui1;

obj.uiName();

obj.uiFamily();

obj.uiAge();

// Вызов собственного метода не разрешается:

// obj.allInfo();

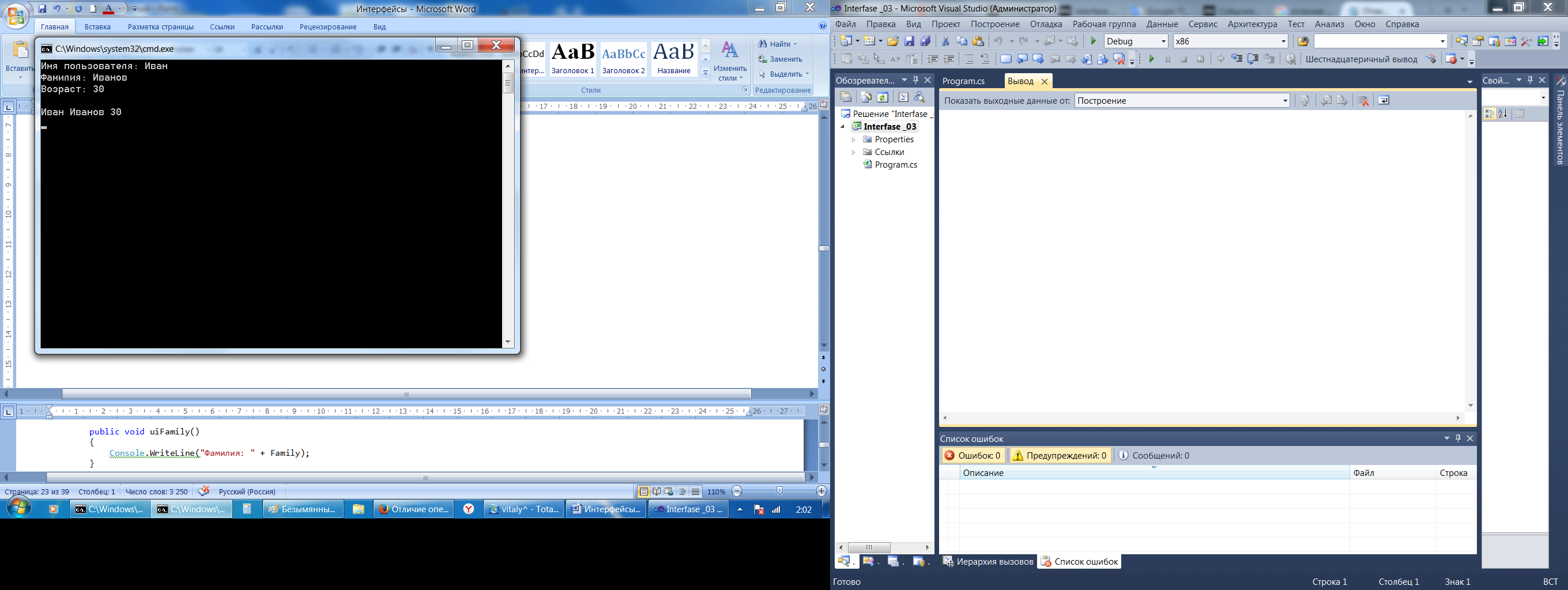
// Но можно явным приведением преобразовать ссылку и вызвать метод

((UI) obj).allInfo();

Console.ReadLine();

}

}



## Ключевые слово as и is.

Поскольку явное приведение может потерпеть неудачу, сгенерировав исключение, бывает так что вы хотите проверить тип переменной без выполнения приведения и наблюдения, получится оно или нет. Проверка такого рода утомительна и неэффективна, к тому же исключения дорого обходятся во время выполнения. По этой причине в С# предусмотрены операции, которые приходят на помощь в таких ситуациях:

* is - можете ли вы преобразовать данное выражение в указанный тип,
* as - возвращает ссылку на целевой тип.

## Ключевое слово as

Определить, поддерживает ли данный тип тот или иной интерфейс, можно с использованием **ключевого слова as**. Если объект удается интерпретировать как указанный интерфейс, то **возвращается ссылка на интересующий интерфейс**, а если нет, то ссылка **null**. Следовательно, перед продолжением в коде необходимо предусмотреть проверку на null.

В предыдущем примере, в методе Main() можно добавить следующую проверку:

IInfo obj = ui1 as IInfo;

if (obj != null)

Console.WriteLine("Тип UI поддерживает интерфейс IInfo");

else

Console.WriteLine(":(");

Обратите внимание, что в случае применения ключевого слова as использовать логику try/catch нет никакой необходимости, поскольку возврат ссылки, отличной от null, означает, что вызов осуществляется с использованием действительной ссылки на интерфейс.

## Ключевое слово is

Проверить, был ли реализован нужный интерфейс, можно также с помощью **ключевого слова is**. Если запрашиваемый объект не совместим с указанным интерфейсом, возвращается значение false, а если совместим, то можно спокойно вызывать члены этого интерфейса без применения логики try/catch.

if (ui1 is IInfo) //*можно ли привести к типу*

Console.WriteLine("Тип UI поддерживает интерфейс IInfo");

else

Console.WriteLine(":(");

# Интерфейсные свойства и индексаторы

## Интерфейсные свойства

Аналогично методам, свойства указываются в интерфейсе вообще без тела. Ниже приведена общая форма объявления интерфейсного свойства:

*// Интерфейсное свойство*

*тип имя{*

*get;*

*set;*

*}*

Очевидно, что в определении интерфейсных свойств, доступных только для чтения или только для записи, должен присутствовать единственный аксессор: get или set соответственно.

Несмотря на то что объявление свойства в интерфейсе очень похоже на объявление [автоматически реализуемого свойства](https://professorweb.ru/my/csharp/charp_theory/level5/5_14.php#autoGet) в классе, между ними все же имеется отличие. При объявлении в интерфейсе свойство не становится автоматически реализуемым. В этом случае указывается только имя и тип свойства, а его реализация предоставляется каждому реализующему классу. Кроме того, при объявлении свойства в интерфейсе не разрешается указывать модификаторы доступа для аксессоров. Например, аксессор set не может быть указан в интерфейсе как private.

Давайте рассмотрим пример:

using System;

namespace ConsoleApplication1

{

interface IUserInfo

{

string Name //*объявление свойства*

{

get;

set;

}

}

class UI : IUserInfo

{ //*реализация свойства*

string myName;

public string Name

{

set

{

myName = value;

}

get

{

return myName;

}

}

}

class Program

{

static void Main()

{

UI user1 = new UI();

user1.Name = "Alexandr";

Console.ReadLine();

}

}

}

В данном примере в классе UI реализуется свойство интерфейса IUserInfo.

## Интерфейсные индексаторы

В интерфейсе можно также указывать индексаторы. Ниже приведена общая форма объявления интерфейсного индексатора:

*// Интерфейсный индексатор*

*тип\_элемента this[int индекс]{*

*get;*

*set;*

*}*

Как и прежде, в объявлении интерфейсных индексаторов, доступных только для чтения или только для записи, должен присутствовать единственный аксессор: get или set соответственно. Пример реализации индексатора:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication7

{

// Indexer on an interface:

public interface ISomeInterface

{

// Indexer declaration:

int this[int index]

{

get;

set;

}

}

// Implementing the interface.

class IndexerClass : ISomeInterface

{

private int[] arr = new int[100];

public int this[int index] // indexer declaration

{

get

{

// The arr object will throw IndexOutOfRange exception.

return arr[index];

}

set

{

arr[index] = value;

}

}

}

class MainClass

{

static void Main()

{

IndexerClass test = new IndexerClass();

System.Random rand = new System.Random();

// Call the indexer to initialize its elements.

for (int i = 0; i < 10; i++)

{

test[i] = rand.Next();

}

for (int i = 0; i < 10; i++)

{

System.Console.WriteLine("Element #{0} = {1}", i, test[i]);

}

// Keep the console window open in debug mode.

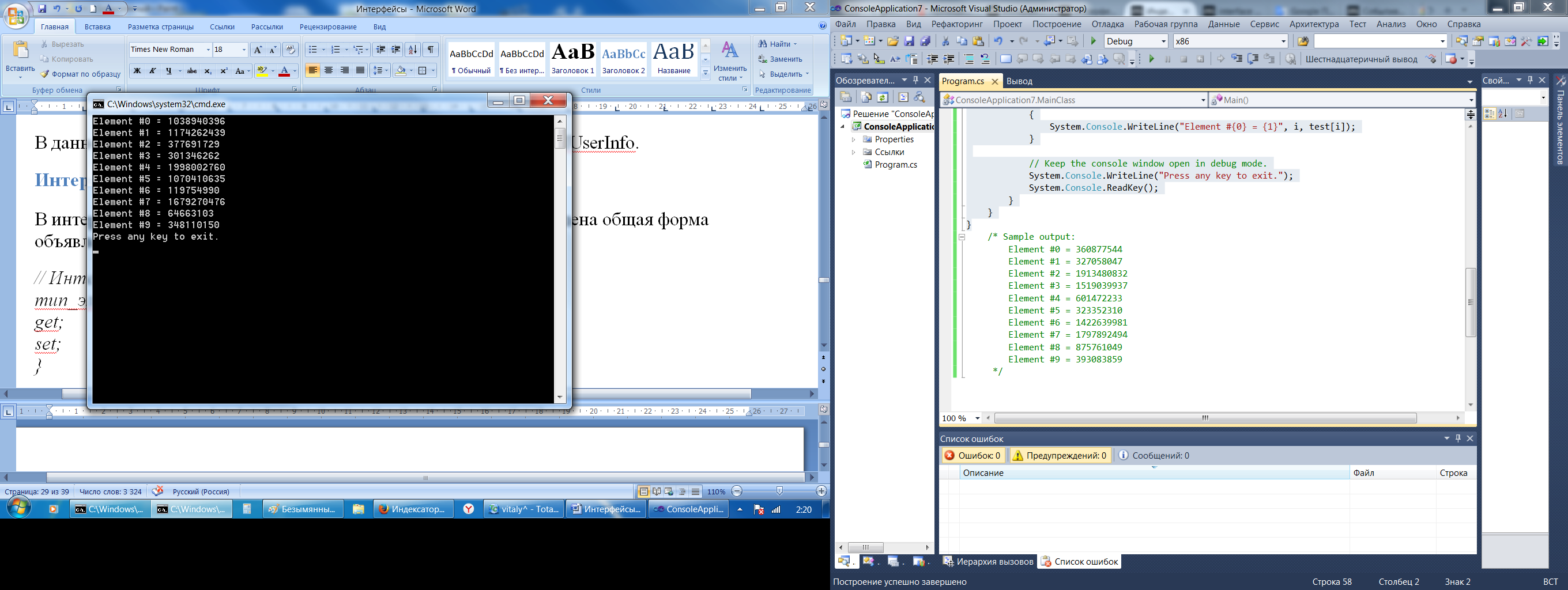
System.Console.WriteLine("Press any key to exit.");

System.Console.ReadKey();

}

}

}



**Наследование интерфейсов**

Один интерфейс может наследовать другой. Синтаксис наследования интерфейсов такой же, как и у классов. Когда в классе реализуется один интерфейс, наследующий другой, в нем должны быть реализованы все члены, определенные в цепочке наследования интерфейсов.

Таким образом, интерфейсы могут быть организованы в иерархии. Как и в иерархии классов, в иерархии интерфейсов, когда какой-то интерфейс расширяет существующий, он наследует все абстрактные члены своего родителя (или родителей). Конечно, в отличие от классов, производные интерфейсы никогда не наследуют саму реализацию. Вместо этого они просто расширяют собственное определение за счет добавления дополнительных абстрактных членов.

Использовать иерархию интерфейсов может быть удобно, когда нужно расширить функциональность определенного интерфейса без нарушения уже существующих кодовых баз.

using System;

namespace ConsoleApplication1

{

public interface A

{

int Sum();

}

// Унаследованный интерфейс

public interface B : A

{

int Del();

}

class MyOperation : B

{

int x = 10, y = 5;

public int Sum()

{

return x + y;

}

public int Del()

{

return x / y;

}

}

class Program

{

static void Main()

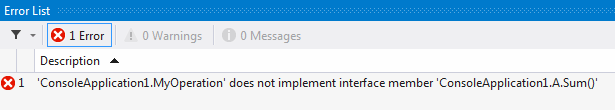
{

}

}

}

Обратите внимание, что класс MyOperation реализует методы обоих интерфейсов, иначе возникла бы ошибка при компиляции:



В отличие от классов, **один интерфейс может расширять сразу несколько базовых интерфейсов, что позволяет проектировать очень мощные и гибкие абстракции**.

**Явная реализация интерфейса**

Единственный класс или структура может реализовать любое количество интерфейсов. Из-за этого всегда существует вероятность реализации интерфейсов с членами, имеющими идентичные имена, и, следовательно, возникает необходимость в устранении конфликтов на уровне имен. **При реализации члена интерфейса имеется возможность указать его имя полностью вместе с именем самого интерфейса**. В этом случае получается явная реализация члена интерфейса, или просто **явная реализация**.

Когда один интерфейс наследует другой, то в производном интерфейсе может быть объявлен член, скрывающий член с аналогичным именем в базовом интерфейсе. Такое сокрытие имен происходит в том случае, если член в производном интерфейсе объявляется таким же образом, как и в базовом интерфейсе. Но если не указать в объявлении члена производного интерфейса ключевое слово **new**, то компилятор выдаст соответствующее предупреждающее сообщение.

Для явной реализации интерфейсного метода могут быть две причины. Во-первых, когда интерфейсный метод реализуется с указанием его полного имени, то такой метод оказывается доступным не посредством объектов класса, реализующего данный интерфейс, а по интерфейсной ссылке. Следовательно, явная реализация позволяет реализовать интерфейсный метод таким образом, чтобы он не стал открытым членом класса, предоставляющего его реализацию. И во-вторых, в одном классе могут быть реализованы два интерфейса с методами, объявленными с одинаковыми именами и сигнатурами. Но **неоднозначность в данном случае устраняется благодаря указанию в именах этих методов их соответствующих интерфейсов.** Рассмотрим Пример:

class Person : ISchool, IUniversity

{

    public void Study()

    {

        Console.WriteLine("Учеба в школе или в университете");

    }

}

interface ISchool

{

    void Study();

}

interface IUniversity

{

    void Study();

}

|  |  |
| --- | --- |
|  |  |

Класс Person определяет один метод Study(), создавая одну общую реализацию для обоих примененных интерфейсов. И вне зависимости от того, будем ли мы рассматривать объект Person как объект типа ISchool или IUniversity, результат метода будет один и тот же.

Однако **нередко бывает необходимо разграничить реализуемые интерфейсы**. В этом случае надо явным образом применить интерфейс:

class Person : ISchool, IUniversity

{

    void ISchool.Study()

    {

        Console.WriteLine("Учеба в школе");

    }

    void  IUniversity.Study()

    {

        Console.WriteLine("Учеба в университете");

    }

}

|  |  |
| --- | --- |
|  |  |

При явной реализации указывается название метода вместе с названием интерфейса, при этом мы не можем использовать модификатор public, то есть методы являются закрытыми. В этом случае при использовании метода Study в программе нам надо объект Person привести к типу соответствующего интерфейса:

static void Main(string[] args)

{

    Person p = new Person();

    ((ISchool)p).Study();

    ((IUniversity)p).Study();

    Console.Read();

}

|  |  |
| --- | --- |
|  |  |

Соберем проект:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication9

{

interface ISchool

{

void Study();

}

interface IUniversity

{

void Study();

}

class Person : ISchool, IUniversity

{

void ISchool.Study()

{

Console.WriteLine("Учеба в школе");

}

void IUniversity.Study()

{

Console.WriteLine("Учеба в университете");

}

}

/\* одна общая реализация для обоих примененных интерфейсов

\* class Person : ISchool, IUniversity

{

public void Study()

{

Console.WriteLine("Учеба в школе или в университете");

}

}

\*/

class Program

{

static void Main(string[] args)

{

Person p = new Person();

((ISchool)p).Study();

((IUniversity)p).Study();

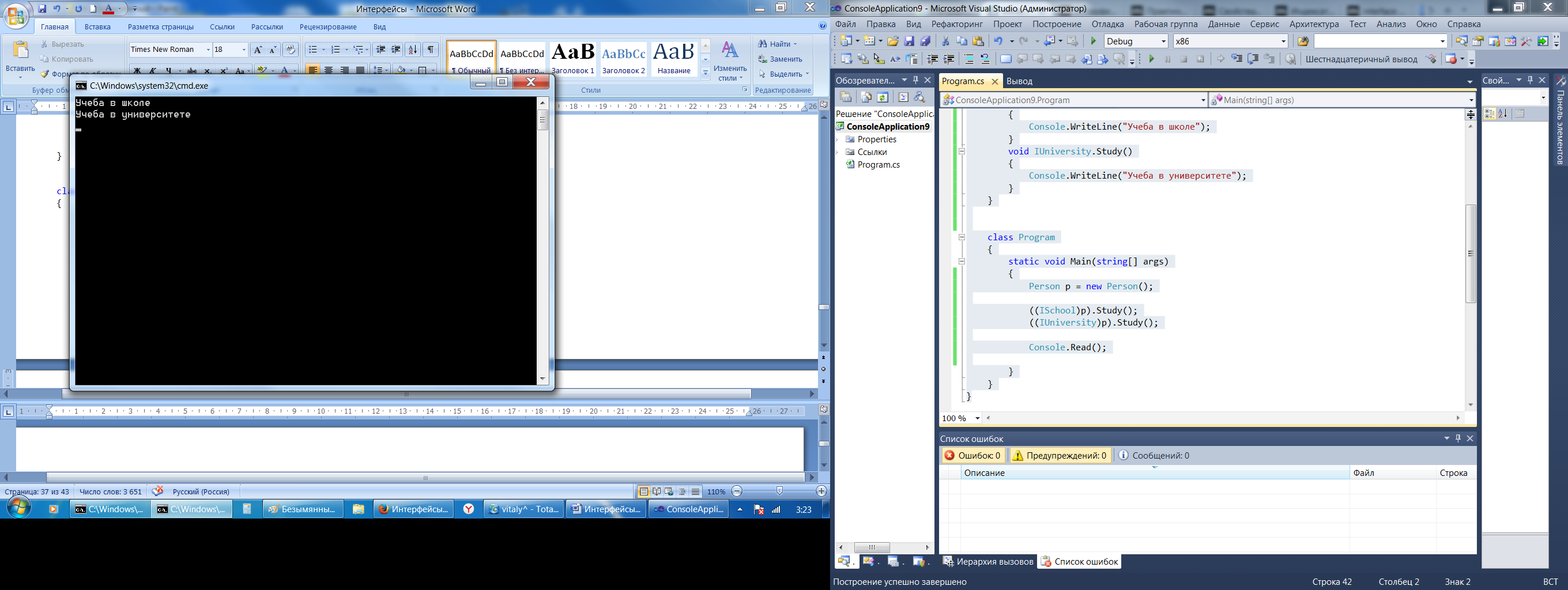
Console.Read();

}

}

}

Результат:



**Интерфейсы являются фундаментальным компонентом** .NET Framework. Какого бы типа приложение не разрабатывалось (веб-приложение, приложение с настольным графическим интерфейсом, библиотека доступа к данными и т.п.), ***работа с интерфейсами будет обязательной частью этого процесса***. Польза от их использования, в следующих случаях:

* При наличии единой иерархии, в которой только какой-то набор производных типов поддерживает общее поведение.
* При необходимости моделировать общее поведение, которое должно встречаться в нескольких иерархиях, не имеющих общего родительского класса помимо System.Object.