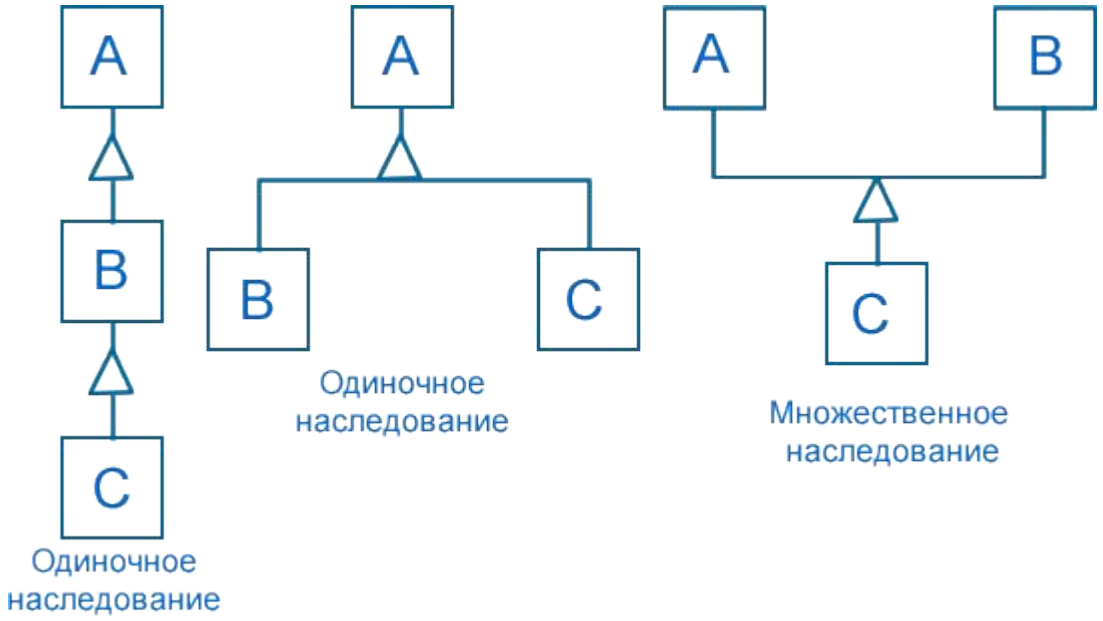
ИНТЕРФЕЙСЫ

Интерфейсы

В языке C# запрещено множественное наследование классов.



Интерфейсы

Тем не менее, в C# существует концепция, позволяющая имитировать множественное наследование.

Это концепция *интерфейсов*.

Интерфейсы

Интерфейс представляет собой набор объявлений свойств, индексаторов, методов и событий.

Класс или структура могут *реализовывать* определенный интерфейс.

В этом случае они берут на себя обязанность предоставить полную реализацию элементов интерфейса (хотя бы пустыми методами).

Интерфейсы

Интерфейс – это контракт, пункты которого суть свойства, индексаторы, методы и события.

Если пользовательский тип реализует интерфейс, он берет на себя обязательство выполнить этот контракт.

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, 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.

Частные случаи ☹

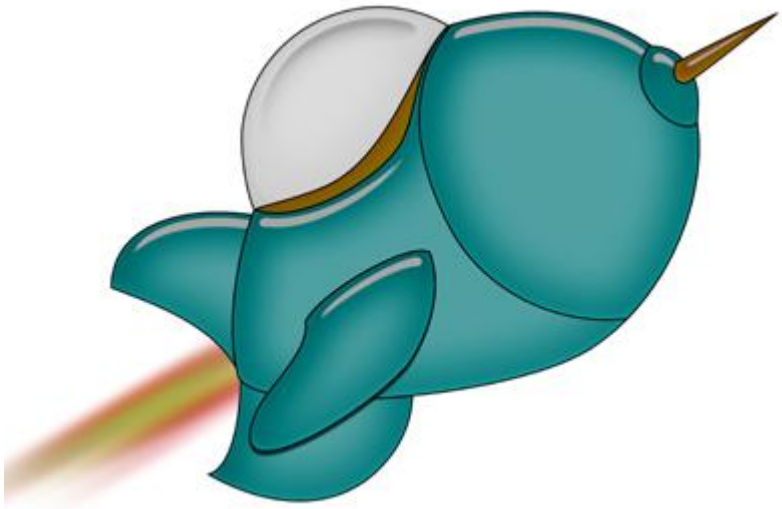
- Введение в язык частных случаев усложняет его и свидетельствует о некоторых изъянах, для преодоления которых и вводятся частные случаи. Например, введение структур в язык C# позволило определять классы как развернутые типы. Конечно, проще было бы ввести в объявление класса соответствующий модификатор, позволяющий любой класс объявлять развернутым. Но этого сделано не было, а, следуя традиции языка C++, были введены структуры как частный случай классов.

Частные случаи ☹

- Интерфейсы позволяют частично справиться с таким существенным недостатком языка, как отсутствие множественного наследования классов. Хотя реализация множественного наследования встречается с рядом проблем, его отсутствие существенно снижает выразительную мощь языка. В языке C# полного множественного наследования классов нет. Чтобы частично сгладить этот пробел, допускается множественное наследование интерфейсов. Обеспечить возможность классу иметь несколько родителей - один полноценный класс, а остальные в виде интерфейсов, - в этом и состоит основное назначение интерфейсов.

Интерфейсы

```
interface IFlying                          // полет
{
    void Fly();                            // метод
    double Speed { get; set; }  // Свойство
}
```



Интерфейсы

Обратите внимание – в определении элементов интерфейса отсутствуют модификаторы уровня доступа. Считается, что все элементы интерфейса имеют `public` уровень доступа.

Следующие модификаторы не могут использоваться при объявлении членов интерфейса: `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, `static`.

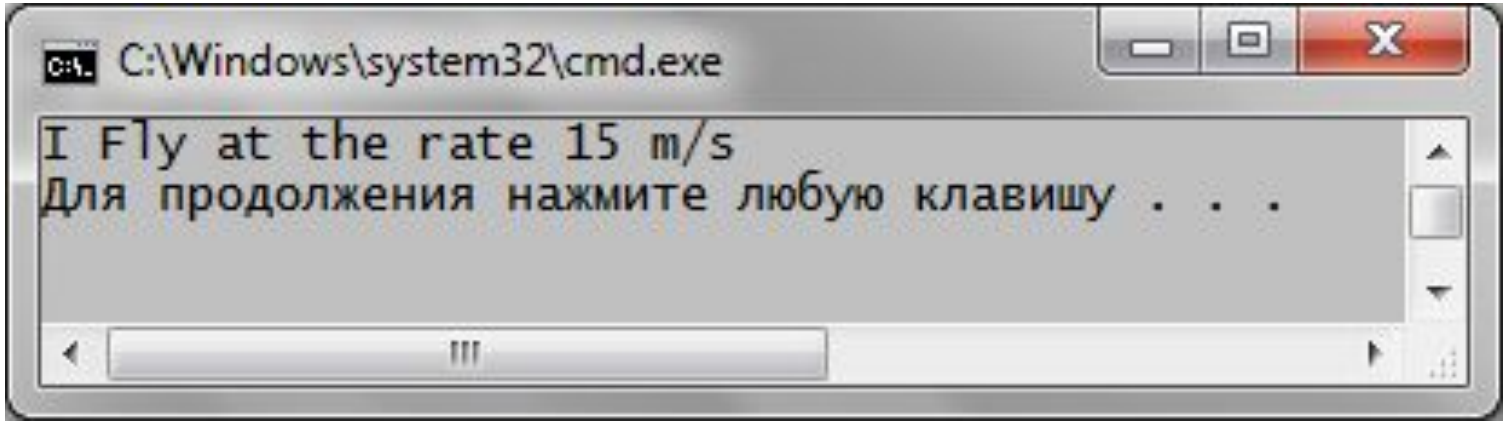
Для свойства, объявленного в интерфейсе, указываются только ключевые слова `get` и (или) `set`.

На основе интерфейса объявим класс:

```
class CFalcon : IFlying
{
    private double FS;          // скорость
    public void Fly()
    {
        Console.WriteLine("I Fly at the rate {0} m/s", FS);
    }
    public double Speed
    {
        get { return FS; }
        set { FS = value; }
    }
}
```

Как это работает (**способ 1**):

```
class Program
{
    static void Main(string[] args)
    {
        CFalcon f1 = new CFalcon();
        f1.Speed = 15;
        f1.Fly();
    }
}
```

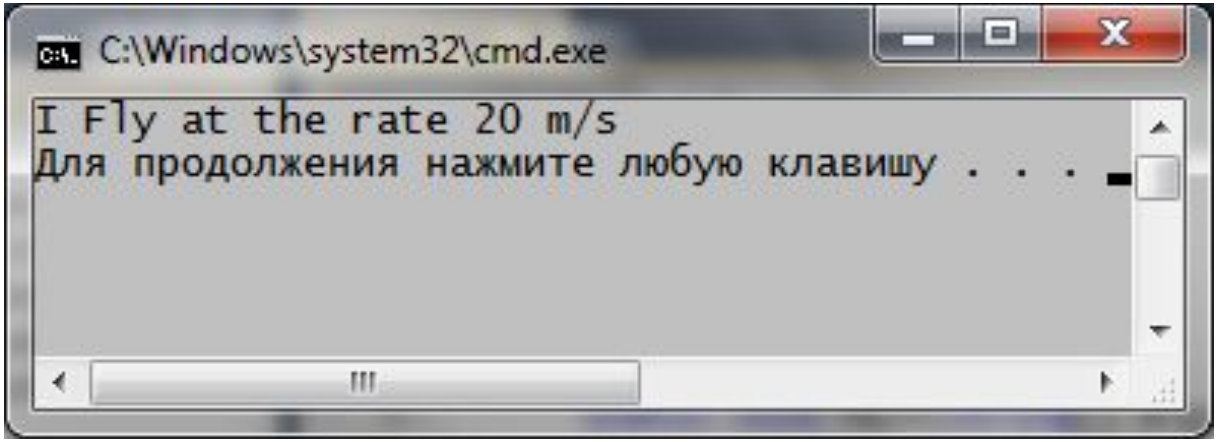


Можно сделать так (**способ 2**):

```
class Program
{
    static void Main(string[] args)
    {
        //CFalcon f1 = new CFalcon();
        //f1.Speed = 15;
        //f1.Fly();

        IFlying p1 = new CFalcon();
        p1.Speed = 20;
        p1.Fly();

    }
}
```



!!! Но нельзя: !!!

```
        //f1.Speed = 15;
        //f1.Fly();

        IFlying p1 = new CFalcon();
        p1.Speed = 20;
        p1.Fly();

        IFlying p1 = new IFlying();

}
```

интерфейс не имеет собственного конструктора!

Есть еще разница в способах 1 и 2:

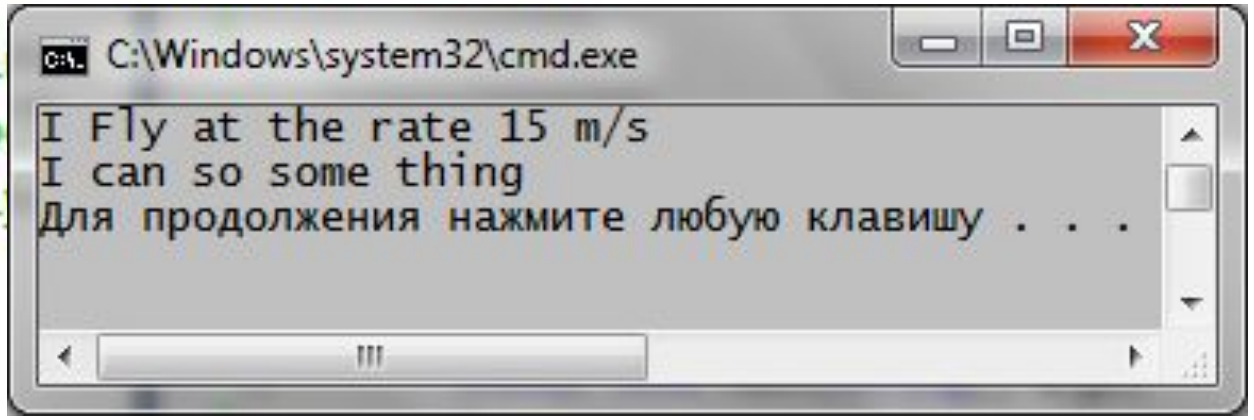
Добавим к классу `CFalcon` метод, не описанный в интерфейсе:

```
class CFalcon : IFlying
{
    private double FS;          // скорость
    public void Fly()
    {
        Console.WriteLine("I Fly at the rate {0} m/s", FS);
    }
    public double Speed
    {
        get { return FS; }
        set { FS = value; }
    }
    public void DoSomeThing()
    {
        Console.WriteLine("I can so some thing");
    }
```

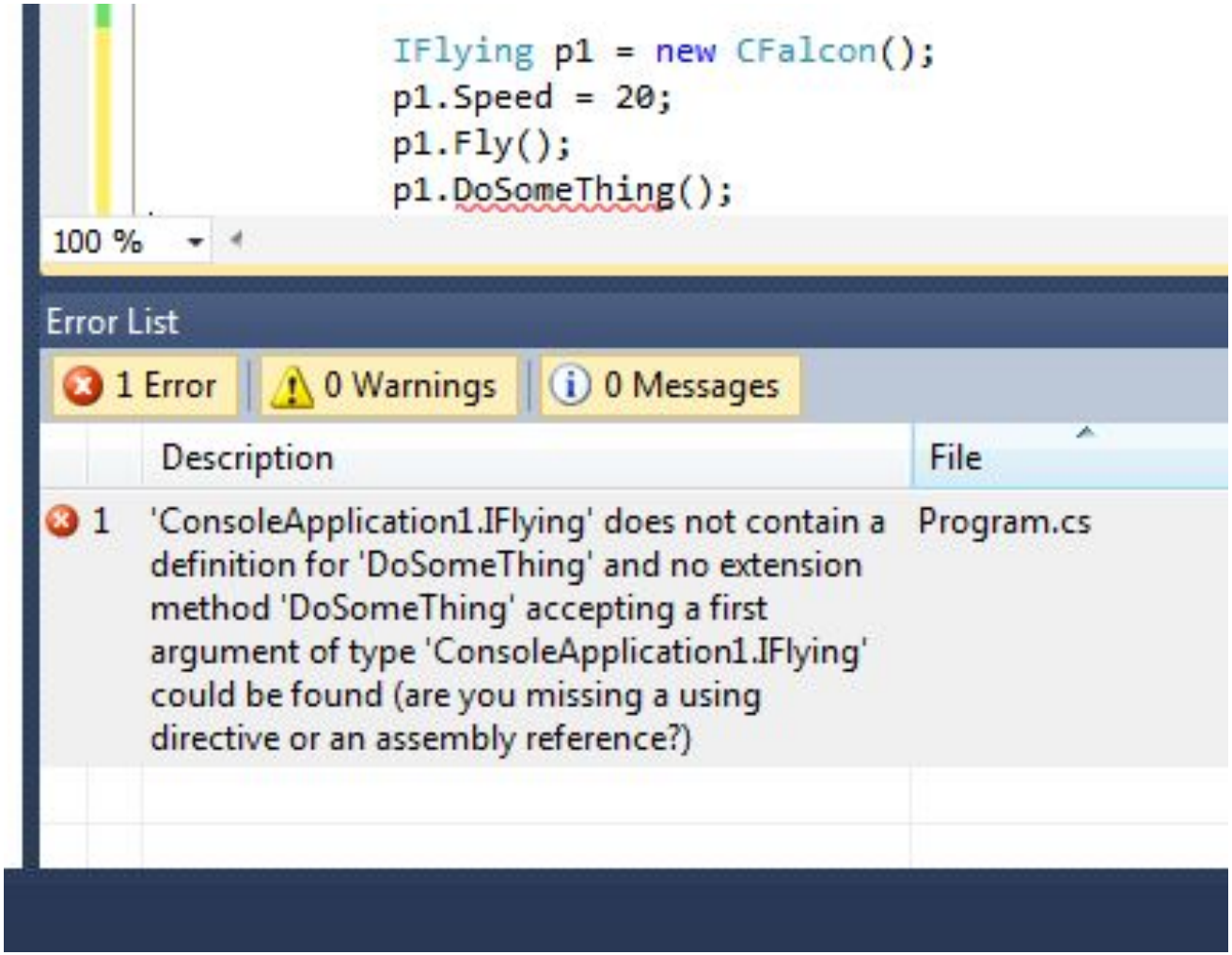
Интерфейсы

```
class Program
{
    static void Main(string[] args)
    {
        CFalcon f1 = new CFalcon();
        f1.Speed = 15;
        f1.Fly();
        f1.DoSomeThing();

        //IFlyi
        //p1.Sp
        //p1.Fl
```



Интерфейсы

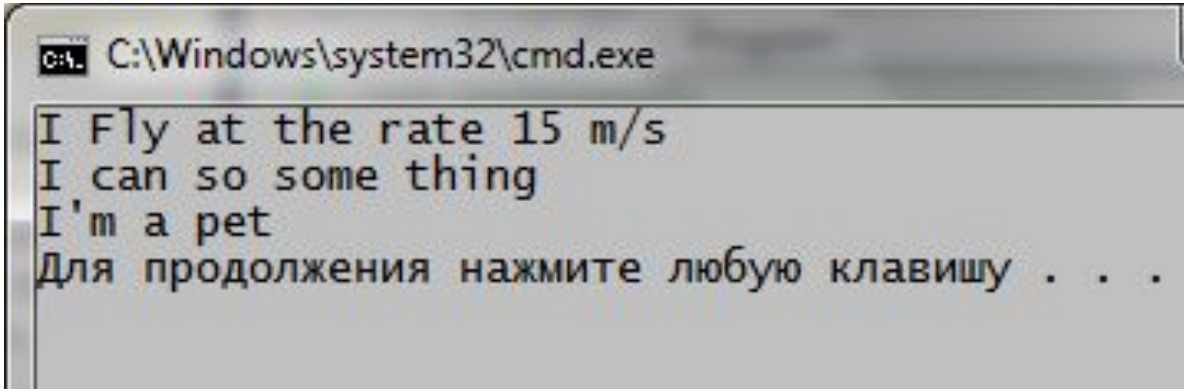

IFlying p1 = new CFalcon();
p1.Speed = 20;
p1.Fly();
p1.DoSomeThing();
100 %
Error List
1 Error
0 Warnings
0 Messages
Description
File
1
'ConsoleApplication1.IFlying' does not contain a definition for 'DoSomeThing' and no extension method 'DoSomeThing' accepting a first argument of type 'ConsoleApplication1.IFlying' could be found (are you missing a using directive or an assembly reference?)
Program.cs

Если класс является производным от некоторого базового класса


```
class CPet
{
    public int eyes = 2;     //глаза
    public void Speak()
    {
        Console.WriteLine("I'm a pet");
    }
}
class CFalcon : CPet, IFlying
{
    private double FS;          // скорость
    public void Fly()
    {
        Console.WriteLine("I Fly at the rate {0} m/s", FS);
    }
    public double Speed
```

теперь объект f1 умеет говорить

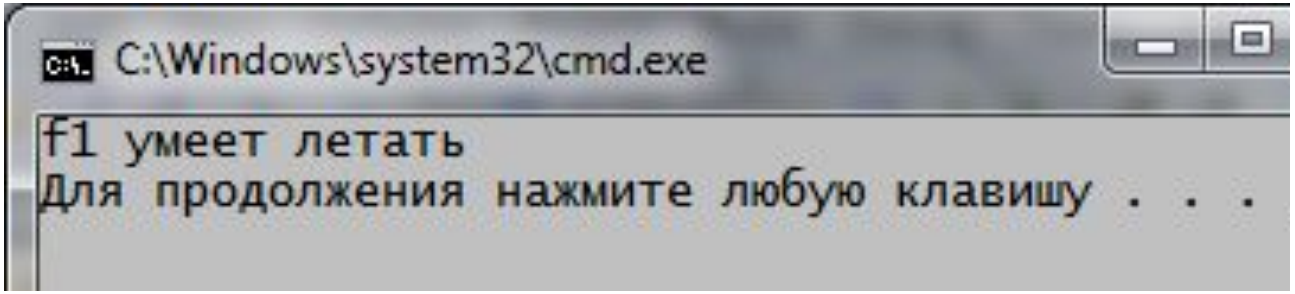
```
class Program
{
    static void Main(string[] args)
    {
        CFalcon f1 = new CFalcon();
        f1.Speed = 15;
        f1.Fly();
        f1.DoSomeThing();
        f1.Speak();
```



Если необходимо проверить, поддерживает ли объект некоего класса интерфейс

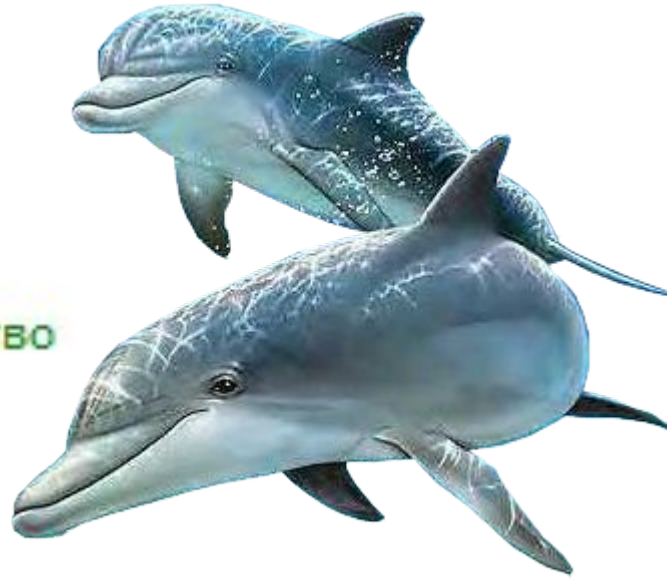
```
{
    CFalcon f1 = new CFalcon();
    f1.Speed = 15;
    //f1.Fly();
    //f1.DoSomeThing();
    //f1.Speak();

    if (f1 is IFlying) Console.WriteLine("f1 умеет летать");
```



Один класс может реализовывать несколько интерфейсов

```
interface ISwimming      // плавание
{
    void Swim();// Метод
    double Speed { get; set; }// Свойство
}
```

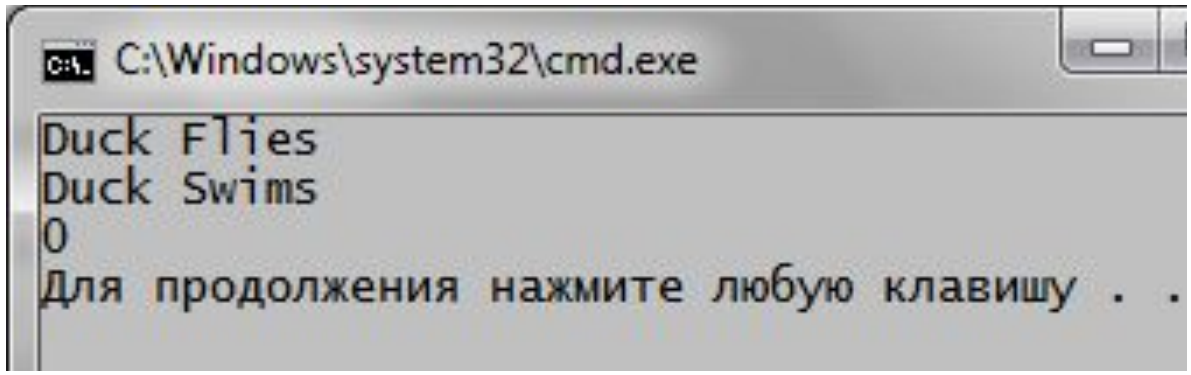


Один класс может реализовывать несколько интерфейсов

```
class CDuck : IFlying, ISwimming
{
    public void Fly()
    {
        Console.WriteLine("Duck Flies");
    }
    public void Swim()
    {
        Console.WriteLine("Duck Swims");
    }
    public double Speed
    {
        get { return 0.0; }
        set { }
    }
}
```

Один класс может реализовывать несколько интерфейсов

```
class Program
{
    static void Main(string[] args)
    {
        CDuck duck = new CDuck();
        duck.Fly();
        duck.Swim();
        duck.Speed = 2;
        Console.WriteLine(duck.Speed);
```



Две стратегии реализации интерфейса

Давайте опишем некоторый интерфейс, задающий дополнительные свойства объектов класса:

```
interface IProps
{
    void Prop1(string s);
    void Prop2 (string name, int val);
}
```

У этого интерфейса два метода, которые и должны будут реализовать все классы - наследники интерфейса. Заметьте, у методов нет модификаторов доступа.

Две стратегии реализации интерфейса

Класс, наследующий интерфейс и реализующий его методы, может реализовать их явно, объявляя соответствующие методы класса открытыми.

```
public class Clain:IProps
{
    public Clain() {}
    public void Prop1(string s)
    {
        Console.WriteLine(s);
    }
    public void Prop2(string name, int val)
    {
        Console.WriteLine("name = {0}, val ={1}", name, val);
    }
}
```

Две стратегии реализации интерфейса

- Класс реализует методы интерфейса, делая их открытыми для клиентов класса и наследников.
- Другая стратегия реализации состоит в том, чтобы все или некоторые методы интерфейса сделать закрытыми. Для реализации этой стратегии класс, наследующий интерфейс, объявляет методы без модификатора доступа, что по умолчанию соответствует модификатору private, и уточняет имя метода именем интерфейса.

Две стратегии реализации интерфейса

```
public class ClainP:IProps
{
    public ClainP(){ }
    void IProps.Prop1(string s)
    {
        Console.WriteLine(s);
    }
    void IProps.Prop2(string name, int val)
    {
        Console.WriteLine("name = {0}, val ={1}", name, val);
    }
}
```

Две стратегии реализации интерфейса

- Класс ClainP реализовал методы интерфейса IProps, но сделал их закрытыми и недоступными для вызова клиентов и наследников класса. Как же получить доступ к закрытым методам?

Есть два способа решения этой проблемы

- **Обертывание**.

- **Кастинг**.

Обертывание

- Создается открытый метод, являющийся оберткой закрытого метода.

Кастинг

- Создается объект интерфейсного класса IProps, полученный преобразованием (кастингом) объекта исходного класса ClainP. Этому объекту доступны закрытые методы интерфейса.

Две стратегии реализации интерфейса

- В чем главное достоинство обертывания? Оно позволяет переименовывать методы интерфейса. Метод интерфейса со своим именем закрывается, а потом открывается под тем именем, которое класс выбрал для него.

Две стратегии реализации интерфейса

Вот пример обертывания закрытых методов в классе ClainP:

```
public void MyProp1(string s)
{
    ((IProps)this).Prop1(s);
}
public void MyProp2(string s, int x)
{
    ((IProps)this).Prop2(s, x);
}
```

Две стратегии реализации интерфейса

- Как видите, методы переименованы и получили другие имена, под которыми они и будут известны клиентам класса. В обертке для вызова закрытого метода пришлось использовать кастинг, приведя объект this к интерфейсному классу IProps.

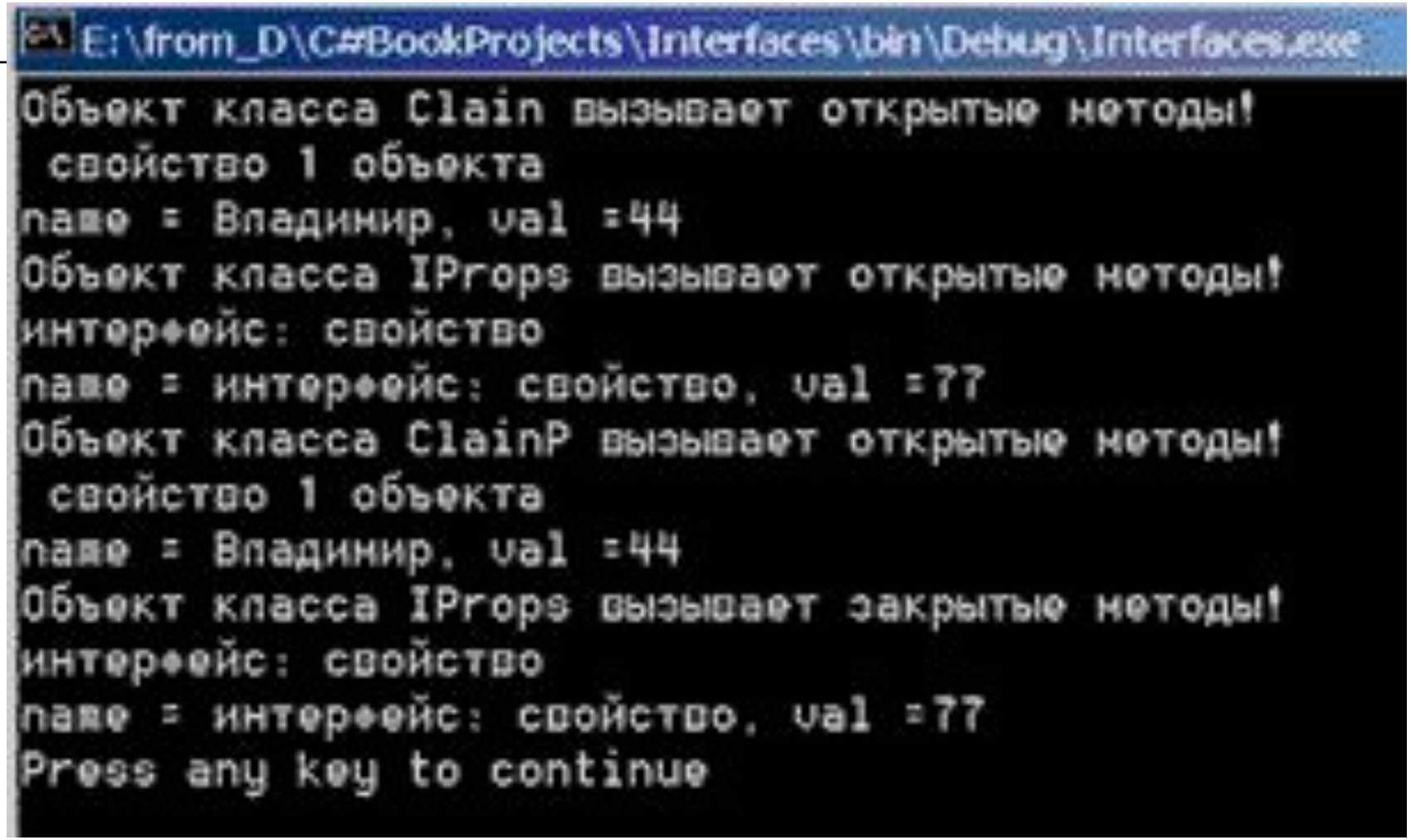
Преобразование к классу интерфейса

- Создать объект класса интерфейса обычным путем с использованием конструктора и операции new нельзя. Тем не менее, можно объявить объект интерфейсного класса и связать его с настоящим объектом путем приведения (кастинга) объекта наследника к классу интерфейса. Это преобразование задается явно. Имея объект, можно вызывать методы интерфейса - даже если они закрыты в классе, для интерфейсных объектов они являются открытыми.

Преобразование к классу интерфейса

```
public void TestClainIProps()
{
        Console.WriteLine("Объект класса Clain вызывает открытые методы!");
        Clain clain = new Clain();
        clain.Prop1(" свойство 1 объекта");
        clain.Prop2("Владимир", 44);
        Console.WriteLine("Объект класса IProps вызывает открытые методы!");
        IProps ip = (IProps)clain;
        ip.Prop1("интерфейс: свойство");
        ip.Prop2 ("интерфейс: свойство",77);
        Console.WriteLine("Объект класса ClainP вызывает открытые методы!");
        ClainP clainp = new ClainP();       // это обертывание
        clainp.MyProp1(" свойство 1 объекта");
        clainp.MyProp2("Владимир", 44);
        Console.WriteLine("Объект класса IProps вызывает закрытые методы!");
        IProps ipp = (IProps)clainp;   // это кастинг
        ipp.Prop1("интерфейс: свойство");
        ipp.Prop2 ("интерфейс: свойство",77);
}
```

Преобразование к классу интерфейса



Проблемы множественного наследования

- При множественном наследовании классов возникает ряд проблем. Они остаются и при множественном наследовании интерфейсов, хотя становятся проще. Рассмотрим две основные проблемы - коллизию имен и наследование от общего предка.

Проблемы множественного наследования

- **Коллизия имен**

Проблема коллизии имен возникает, когда два или более интерфейса имеют методы с одинаковыми именами и сигнатурой. Сразу же заметим, что если имена методов совпадают, но сигнатуры разные, то это не приводит к конфликтам - при реализации у класса наследника просто появляются перегруженные методы. Но что следует делать классу-наследнику в тех случаях, когда сигнатуры методов совпадают? И здесь возможны две стратегии - склеивание методов и переименование.

Проблемы множественного наследования

- Стратегия склеивания применяется тогда, когда класс - наследник интерфейсов - полагает, что разные интерфейсы задают один и тот же метод, единая реализация которого и должна быть обеспечена наследником. В этом случае наследник строит единственную общедоступную реализацию, соответствующую методам всех интерфейсов, которые имеют единую сигнатуру.

Проблемы множественного наследования

```
interface IProps
{
    void Prop1(string s);
    void Prop2 (string name, int val);
    void Prop3();
}
interface IPropsOne
{
    void Prop1(string s);
    void Prop2 (int val);
    void Prop3();
}
```

Проблемы множественного наследования

```
public class ClainTwo:IProps,IPropsOne
{
    // склеивание методов двух интерфейсов
        public void Prop1 (string s)
    {
        Console.WriteLine(s);
    }
```

Проблемы множественного наследования

Другая стратегия исходит из того, что, методы разных интерфейсов должны быть реализованы по-разному. В этом случае необходимо переименовать конфликтующие методы.

Проблемы множественного наследования

Конечно, переименование можно сделать в самих интерфейсах, но это неправильный путь: наследники не должны требовать изменений своих родителей - они сами должны меняться.

Для переименования метода интерфейса в самом классе наследника достаточно реализовать методы разных интерфейсов как закрытые, а затем открыть их с переименованием.

Проблемы множественного наследования

```
public class ClainTwo:IProps,IPropsOne
 {/// перегрузка методов двух интерфейсов
        public void Prop2(string s, int x)
        {     Console.WriteLine(s + "; " + x);}
        public void Prop2 (int x)
        {     Console.WriteLine(x);       }
        // переименование методов двух интерфейсов
        void IProps.Prop3()
        {     Console.WriteLine("Метод 3 интерфейса 1");   }
        void IPropsOne.Prop3()
        {     Console.WriteLine("Метод 3 интерфейса 2");   }
        public void Prop3FromInterface1()
        {     ((IProps)this).Prop3();}
        public void Prop3FromInterface2()
        {     ((IPropsOne)this).Prop3(); }
```

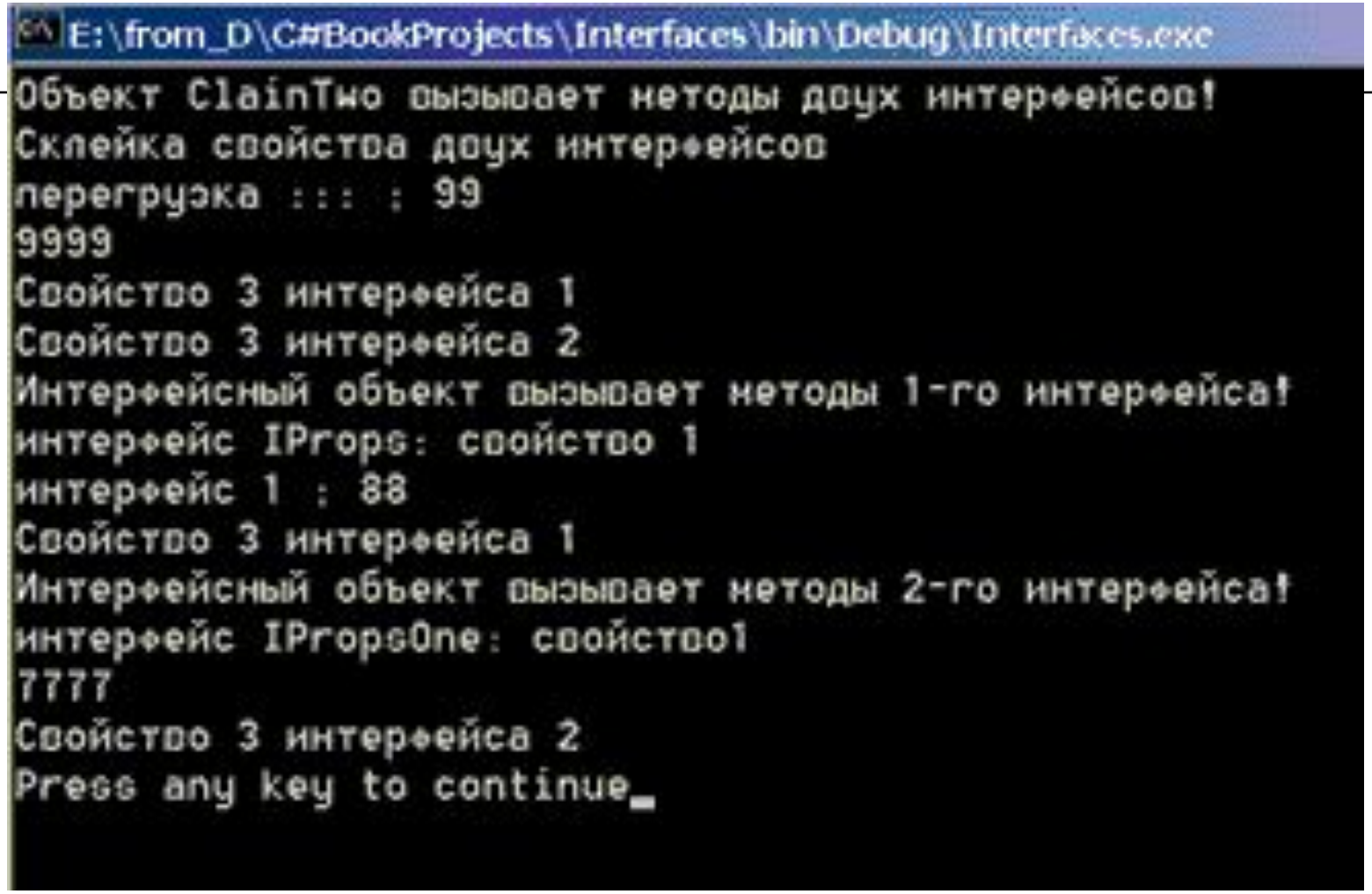
Проблемы множественного наследования

Для первого из методов с совпадающей сигнатурой выбрана стратегия склеивания, так что в классе есть только один метод, реализующий методы двух интерфейсов. Методы с разной сигнатурой реализованы двумя перегруженными методами класса. Для следующей пары методов с совпадающей сигнатурой выбрана стратегия переименования. Методы интерфейсов реализованы как закрытые методы, а затем в классе объявлены два новых метода с разными именами, являющиеся обертками закрытых методов класса.

Проблемы множественного наследования

```
public void TestCliTwoInterfaces()
{
        Console.WriteLine("Объект ClainTwo вызывает методы двух интерфейсов!");
        Cli.ClainTwo claintwo = new Cli.ClainTwo();
        claintwo.Prop1("Склейка свойства двух интерфейсов");
        claintwo.Prop2("перегрузка ::: ",99);
        claintwo.Prop2(9999);
        claintwo.Prop3FromInterface1();
        claintwo.Prop3FromInterface2();
        Console.WriteLine("Интерфейсный объект вызывает методы 1-го
              интерфейса!");
        Cli.IProps ip1 = (Cli.IProps)claintwo;
        ip1.Prop1("интерфейс IProps: свойство 1");
        ip1.Prop2("интерфейс 1 ", 88);
        ip1.Prop3();
        Console.WriteLine("Интерфейсный объект вызывает методы 2-го
              интерфейса!");
        Cli.IPropsOne ip2 = (Cli.IPropsOne)claintwo;
        ip2.Prop1("интерфейс IPropsOne: свойство1");
        ip2.Prop2(7777);
        ip2.Prop3();
}
```

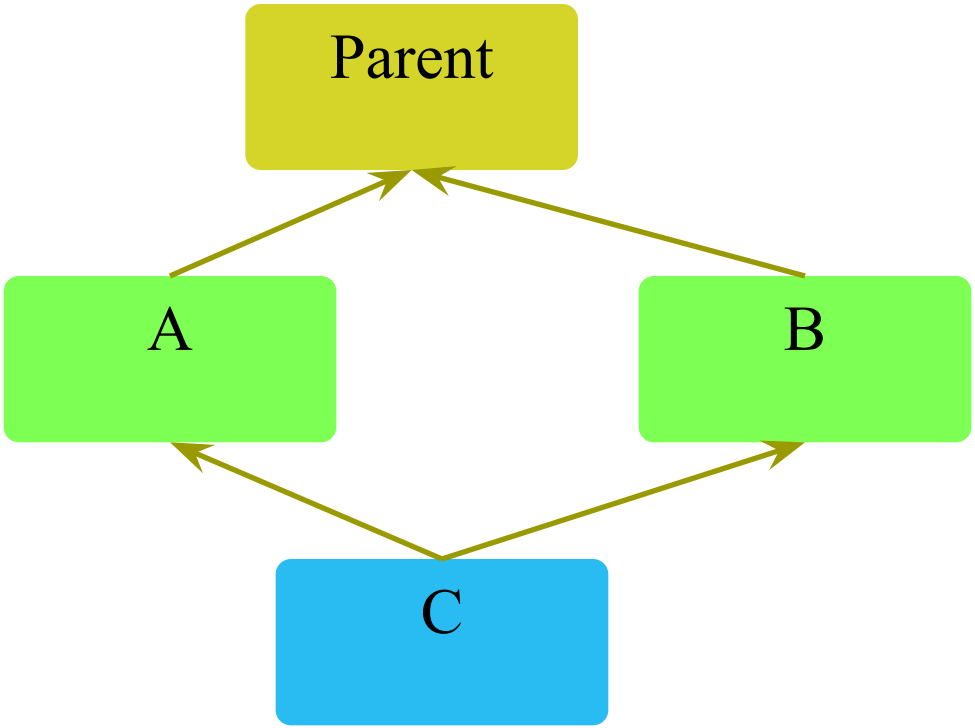
Проблемы множественного наследования



Наследование от общего предка

- Проблема наследования от общего предка характерна, в первую очередь, для множественного наследования классов. Если класс C является наследником классов A и B, а те, в свой черед, являются наследниками класса Parent, то класс наследует свойства и методы своего предка Parent дважды, один раз получая их от класса A, другой от - B. Это явление называется еще дублирующим наследованием. Для классов ситуация осложняется тем, что классы A и B могли по-разному переопределить методы родителя и для потомков предстоит сложный выбор реализации.

Наследование от общего предка



Наследование от общего предка

- Для интерфейсов сама ситуация дублирующего наследования маловероятна, но возможна, поскольку интерфейс, как и любой класс, может быть наследником другого интерфейса. Поскольку у интерфейсов наследуются только сигнатуры, а не реализации, как в случае классов, то проблема дублирующего наследования сводится к проблеме коллизии имен. По-видимому, естественным решением этой проблемы в данной ситуации является склеивание, когда методам, пришедшим разными путями от одного родителя, будет соответствовать единая реализация.

Наследование от общего предка

```
public interface IParent
{
    void ParentMethod();
}
public interface ISon1:IParent
{
    void Son1Method();
}
public interface ISon2:IParent
{
    void Son2Method();
}
```

Наследование от общего предка

класс, наследующий оба интерфейса:
public class Pars:ISon1, ISon2

```
{
    public void ParentMethod()
    {
        Console.WriteLine("Это метод родителя!");
    }
    public void Son1Method()
     {
         Console.WriteLine("Это метод старшего сына!");
     }
    public void Son2Method()
    {
        Console.WriteLine("Это метод младшего сына!");
    }
}
```

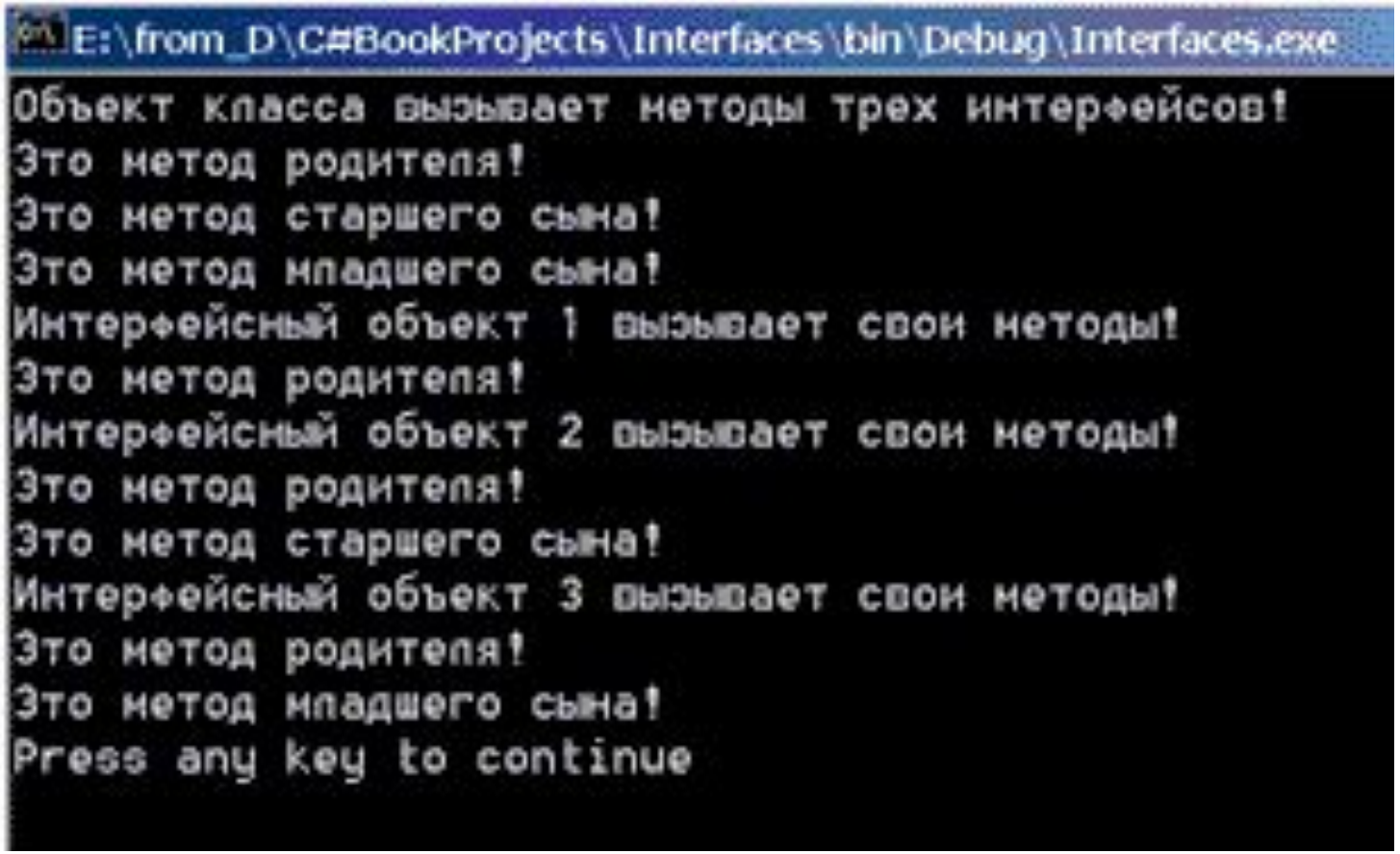
Наследование от общего предка

- Класс обязан реализовать метод ParentMethod, приходящий от обоих интерфейсов. Понимая, что речь идет о дублировании метода общего родителя - интерфейса IParent, лучшей стратегией реализации является склеивание методов в одной реализации, что и было сделано.

Наследование от общего предка

```
public void TestIParsons()
{       Console.WriteLine("Объект класса вызывает методы трех интерфейсов!");
        Cli.Pars ct = new Cli.Pars();
        ct.ParentMethod();
        ct.Son1Method();
        ct.Son2Method();
        Console.WriteLine("Интерфейсный объект 1 вызывает свои методы!");
        Cli.IParent ip = (IParent)ct;
        ip.ParentMethod();
        Console.WriteLine("Интерфейсный объект 2 вызывает свои методы!");
        Cli.ISon1 ip1 = (ISon1)ct;
        ip1.ParentMethod();
        ip1.Son1Method();
        Console.WriteLine("Интерфейсный объект 3 вызывает свои методы!");
        Cli.ISon2 ip2 = (ISon2)ct;
        ip2.ParentMethod();
        ip2.Son2Method();
}
```

Наследование от общего предка



Наследование интерфейсов

Подобно классам, интерфейсы могут наследоваться от других интерфейсов.

При этом, в отличие от классов, наследование интерфейсов может быть множественным.

Отличие интерфейса от абстрактного класса

- Абстрактный класс, помимо абстрактных методов и свойств, может содержать конструкторы, поля и реализованные методы
- Все, что определено в абстрактных классах, доступно только наследникам
- Наследники абстрактного класса должны обязательно иметь реализацию абстрактных методов

Отличие интерфейса от абстрактного класса

- Интерфейс выражает поведение, которое данный класс или структура может избрать для поддержки. Каждый класс (или структура) может поддерживать столько интерфейсов, сколько необходимо, и, следовательно, тем самым поддерживать множество поведений.