

**Наследование,
полиморфизм,
виртуальные функции.
Интерфейсы и
абстрактные классы**

Наследование – это механизм получения
нового класса на основе уже существующего

элементы
данных старого
класса



элементами
данных нового
класса

методы
старого класса

БАЗОВЫЙ
КЛАСС

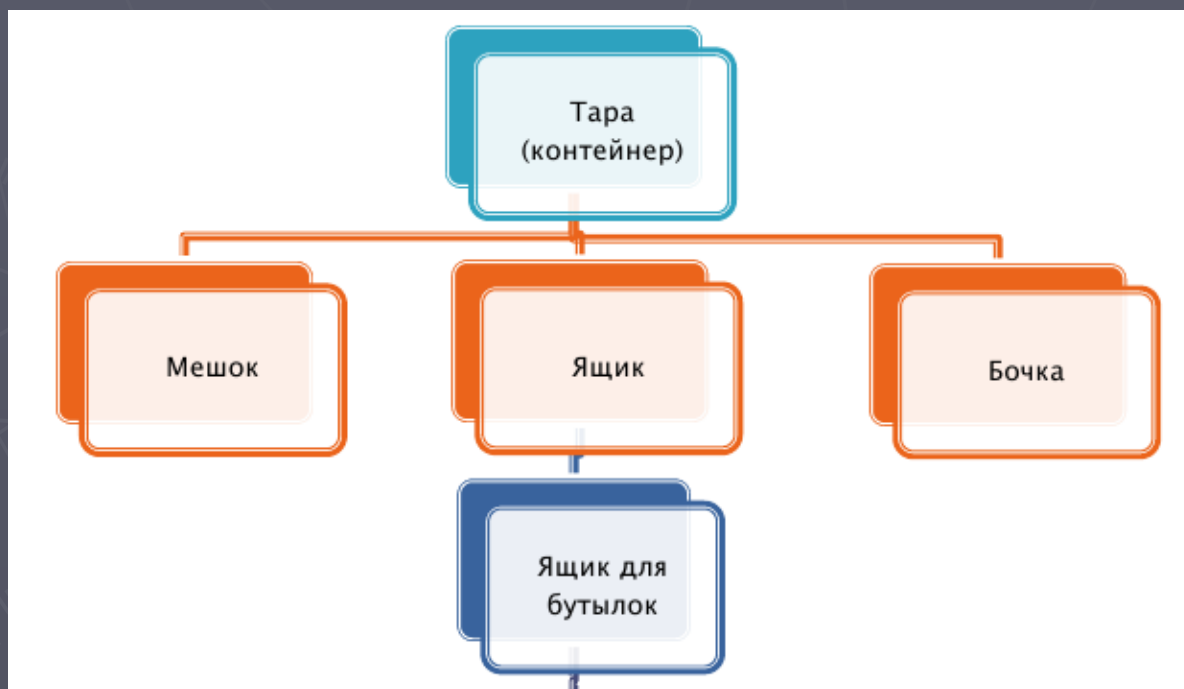


к старой
составляющей
объекта нового
класса

ПРОИЗВОД-
НЫЙ КЛАСС

Роль наследования

- ▶ формирует иерархию
- ▶ поощряет повторное использование кода



```
class Person
{
    private string _firstName;
    private string _lastName;

    public string FirstName
    {
        get { return _firstName; }
        set { _firstName = value; }
    }
    public string LastName
    {
        get { return _lastName; }
        set { _lastName = value; }
    }
}
```

```
class Student : Person
{
}
```

← базовый класс или суперкласс

Правила наследования:

- 1) В C# наследование всегда подразумевается открытым

```
class Student : Person
```

- 2) Запрещено множественное наследование классов (но не интерфейсов)
- 3) наследуются все свойства, методы, поля и т.д., которые есть в базовом классе
- 4) Производному классу доступны public, internal, protected и protected internal члены базового класса (private – недоступны)

5) не наследуются конструкторы базового класса

6) тип доступа к производному классу должен быть таким же, как и у базового класса или более строгим

```
internal class Машина { }
```

```
public class Грузовик :Машина{ }
```



```
class OOP_Lect.Грузовик:
```

Несогласованность по доступности: доступность базового класса "Машина" ниже доступности класса "Гр

- 7) Ссылке на объект базового класса можно присвоить объект производного класса (но вызываются для него только методы и свойства, определенные в базовом классе.)

```
class Person
{
    public void buy() { }
}
class Student : Person
{
    public void study() { }
}
```

```
Person anna = new Person();
Person uman = new Student();

anna.buy();
uman.buy();
uman.study();
```

Ссылки базового и производного классов

```
public class Point
{
    public int x = 10;
    public int y = 20;
    public int Sum() { return x + y; }
}

public class ColorPoint : Point
{
    public int color = 78;
    public int Mult() { return x * y * color; }
}

private static void Main(string[] args)
{
    Point a12 = new Point();
    ColorPoint ca100 = new ColorPoint();
    // ca100 = a12; // ошибка
    a12 = ca100;
    Console.WriteLine(a12.ToString());
    Console.WriteLine(a12.Sum());
    // Console.WriteLine(a12.Mult()); // ошибка нельзя
    // ВЫЗОВ МЕТОДОВ ИЗ ТИПА ССЫЛК
}
```


Ключевое слово base

Конструкторы не наследуются

Автоматически
вызывается
конструктор
базового
класса без
параметров

```
class X
{
    public X() { }
    public X(int key) { }
}
class Y : X
{
    public Y(int key) { }
    public Y() : base(125) { }
}
static void Main(string[] args)
//
{
    Y y0 = new Y();
    Y y1 = new Y(125);
}
```

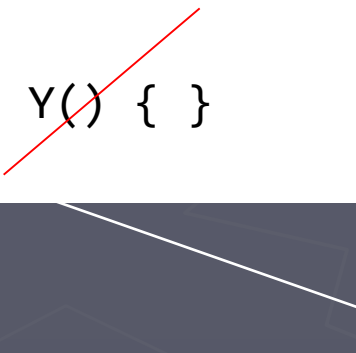
System.Object

Конструкторы базовых классов
вызываются, начиная с самого
верхнего уровня

Ошибка – невозможно найти
конструктор

```
class X
{
    public X(int _i) { }
}

class Y:X
{
    public Y() { }
}
```



```
class X
{
    public X(int _i) { }
}

class Y:X
{
    public Y():base (100) { }
}
```

```
class X
{
    public X() { }
    public X(int _i) { }
}

class Y:X
{
    public Y() { }
}
```

Если
конструктор
базового класса
требует
указания
параметров, он
должен
быть явным
образом вызван
в конструкторе
производного
класса в списке
инициализации
(base)

```
class X
{
    public X() { }
    public X(int key) { }
}
class Y : X
{
    public Y(int key) { }
    public Y() : base(125) { }
}
static void Main(string[] args)
//
{
    Y y0 = new Y();
    Y y1 = new Y(125);
}
```

System.Object

Стратегии наследования

- ▶ Обычное наследование всех членов базового класса в классе-наследнике
- ▶ Переопределение членов базового класса в классе-наследнике (полиморфизм)
- ▶ Соккрытие членов базового класса в классе-наследнике

Обычное наследование

```
public class Point
{
    public int x = 10;
    public int y = 20;
    public int Sum() { return x + y; }
}
public class ColorPoint : Point
{
    public int color = 78;
}
private static void Main(string[] args)
{
    ColorPoint ca100 = new ColorPoint();

    Console.WriteLine(ca100.Sum());//30
    // вызов методов по типу ссылки
}
```

Соккрытие имен при наследовании

- ▶ В производном классе можно определить члены с таким же именем, как и у члена его базового класса

```
public class Point
```

```
{
```

```
    public int x = 10;
```

```
    public int y = 20;
```

```
    public String ToString()
```

```
    { return "Point " + x + " " + y; }
```

```
}
```

```
public class ColorPoint : Point
```

```
{
```


```
    public int x = -78;
```

```
    new public String ToString()
```

```
    { return "ColorPoint " + x + base.ToString();
```

```
    }
```

```
}
```

▶  CS0108 "Iterarhi.ColorPoint.x" скрывает наследуемый член "Iterarhi.Point.x". Используйте ключевое слово new.

маскирует (или скрывает)

предупреждение можно заглушить, явно скрываем метод из базового класса

Обращение к скрытым членам

```
public class Point
{
    public int x = 10;
    public int y = 20;

    public int Sum()    { return x + y;    }

    public String ToString()
        { return "Point " + x + " " + y;  }
}

public class ColorPoint : Point
{
    public new int x = -78;

    new public String ToString()
        { return "ColorPoint " + x + base.ToString(); }

    new public int Sum()
        { return base.x + base.y + x;  }
}
```

Полиморфизм

- ▶ ключевой аспект объектно-ориентированного программирования
- ▶ способность к изменению функций, унаследованных от базового класса



Виртуальные: методы, свойства, индексаторы

полиморфный интерфейс в базовом классе - набор членов класса, которые могут быть переопределены в классе-наследнике

```
virtual public void A_method() { }
```

переопределение виртуального метода в производном классе:

```
override public void A_method() { }
```

без полиморфизма

```
public class Point
{
    public int x = 10;
    public int y = 20;
    public int Sum() { return x + y; }
}

public class ColorPoint : Point
{
    public int color = 78;
    public int Sum() { return x * y * color; }
}

private static void Main(string[] args)
{
    Point a12 = new Point();
    Console.WriteLine(a12.Sum()); //30

    ColorPoint ca100 = new ColorPoint();
    a12 = ca100;
    Console.WriteLine(a12.Sum()); //30
    //  ВЫЗОВ МЕТОДОВ ПО ТИПУ ССЫЛКИ
}
```

ПОЛИМОРФИЗМ

```
public class Point
{
    public int x = 10;
    public int y = 20;
    virtual public int Sum() { return x + y; }
}

public class ColorPoint : Point
{
    public int color = 78;
    override public int Sum() { return x * y * color; }
}

private static void Main(string[] args)
{
    Point a12 = new Point();
    Console.WriteLine(a12.Sum()); //30

    ColorPoint ca100 = new ColorPoint();
    a12 = ca100;
    Console.WriteLine(a12.Sum()); //15600
    // ВЫЗОВ МЕТОДОВ ПО ТИПУ ОБЪЕКТА
}
```

потенциально виртуальный
virtual ставить нельзя

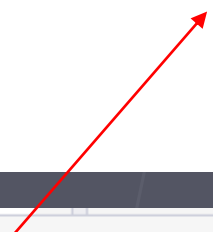
Правила переопределения

- ▶ 1) Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса.
- ▶ 2) не может быть `static` или `abstract`
- ▶ 3) вызывается ближайший вариант, обнаруживаемый вверх по иерархии (многоуровневая)

► Если не virtual переопределять нельзя

```
class X
{
    public void A_method() { }
}

class Y : X
{
    override public void A_method() { }
}
```



void Y.A_method()

'StatProgram.Y.A_method()': невозможно переопределить наследуемый член "StatProgram.X.A_method()", так как он не помечен как virtual, abstract или override.

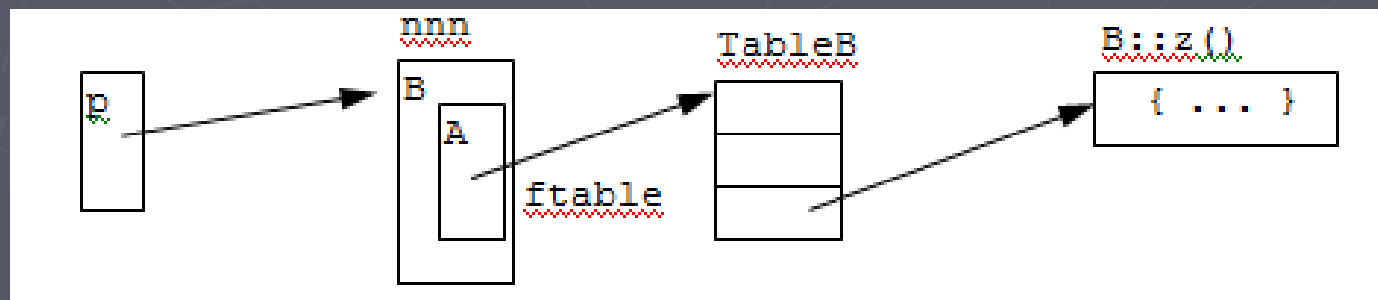
► Перегрузка методов обеспечивает статический полиморфизм, а виртуальный метод – динамический.

раннее связывание – адрес функции назначается во время компиляции, и именно этот адрес используется при вызове функции

позднее связывание (только для методов классов) – во время выполнения приложения определяется действительный класс объекта, адрес которого находится в указателе, и вызывается метод нужного класса.

Виртуальные функции предоставляют механизм **позднего (отложенного)** или **динамического связывания**.

*таблица
виртуальных
методов (vtbl),*



Тип связывания	Достоинства	Недостатки
Ранее	высокое быстродействие получаемых выполнимых программ	снижение гибкости программ
Позднее	высокая гибкость выполняемой программы, возможность реакции на события	относительно низкое быстродействие программы

Бесплодные (запечатанные) классы

- ▶ класс, от которого наследовать запрещается


```
abstract class AAA
{
    public abstract void A_method();
}

sealed class A : AAA
{
    override public void A_method() {}
}

class B : AAA
{
    override public void A_method() {}
}
```

Запрет переопределения методов

```
public class Point
{
    public int x = 10;
    public int y = 20;
    public virtual int Sum() { return x + y; }
}
public class ColorPoint : Point
{
    public int color = 78;
    public sealed override int Sum() { return x * y
    color; }
}
```



метод в незапечатанном классе
является запечатанным

не сможем переопределить метод Sum в
классе, унаследованном от ColorPoint.

Абстрактные классы

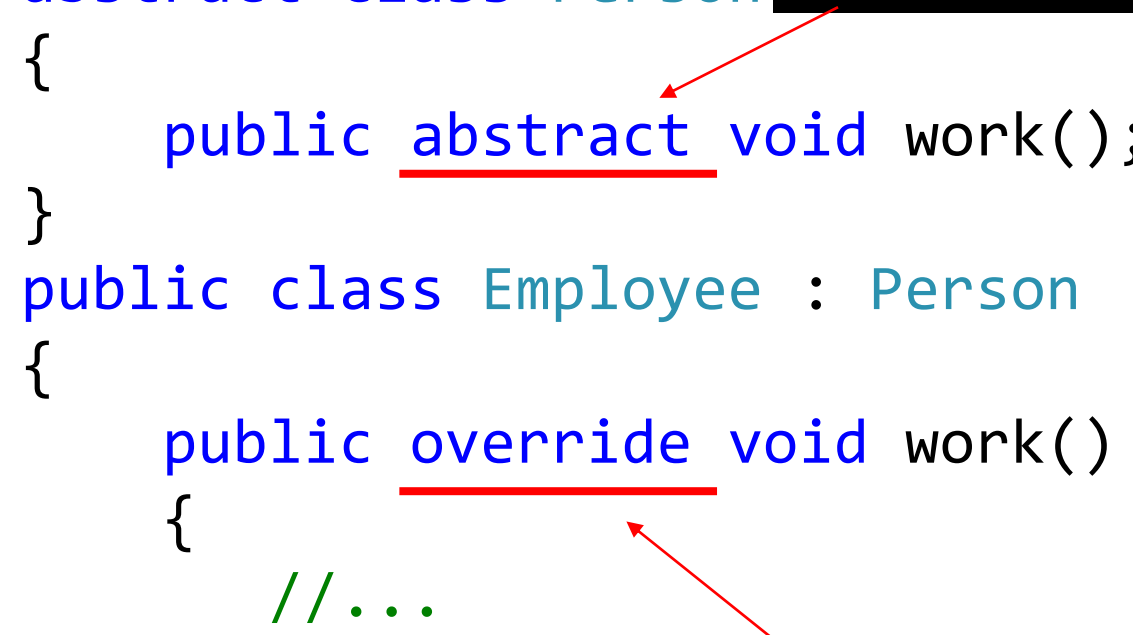
- ▶ Служит только для порождения потомков - предоставляют базовый функционал для классов-наследников.
- ▶ Задаёт интерфейс для всей иерархии
- ▶ Может содержать и полностью определённые методы, переменные, конструкторы, свойства
- ▶ Создавать объект абстрактного класса нельзя!!!!!! (ссылку можно)

абстрактный метод

Определяет полиморфный интерфейс

не имеют никакой реализации

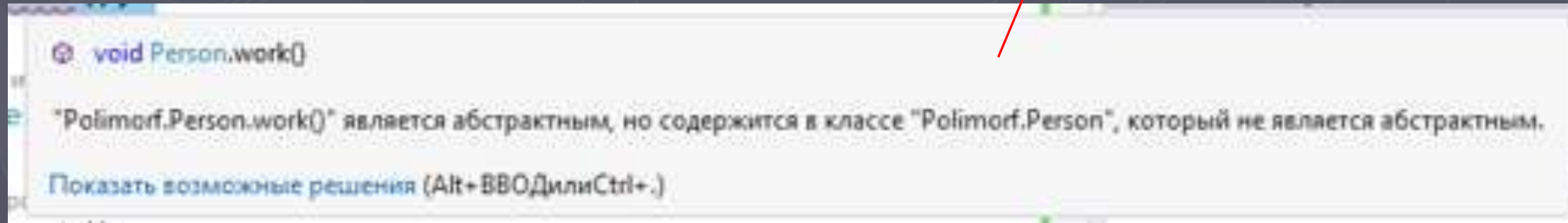
```
public abstract class Person
{
    public abstract void work();
}
public class Employee : Person
{
    public override void work()
    {
        //...
    }
}
```



производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе

- ▶ если класс имеет хотя бы одно абстрактное свойство или метод, то он должен быть определен как абстрактный.

```
public class Person
{
    public abstract void work();
}
```



Свойства abstract методов

- ▶ 1) абстрактные методы автоматически виртуальные (virtual не ставится)
- ▶ 2) абстрактные методы не используются со static

```
public abstract class Person
{
    public abstract void work();
}
private static void Main(string[] args)
{
    Person anna = new Person(); //ошибка
}
```

- 3) А.К. может быть параметром метода
- полиморфные методы

```
abstract class Car { }  
class Cargo : Car { }  
class BigCar : Car { }  
class Maneger  
{  
    public List<Car> createGarag (Car[] All);  
}
```

Интерфейсы

позволяют определить требования к реализации (контракт)

- Задается набор абстрактных методов, свойств, событий и индексаторов, которые должны быть реализованы в производных классах

[атрибуты] [спецификаторы]

interface `Имя_интерфейса` [: предки]

Тело интерфейса[;]

Свойства	Интерфейс
не может содержать	константы, поля, операции, конструкторы, деструкторы, типы, любые статические элементы
может содержать	абстрактные методы, обобщения свойства и индексаторы, а также события
Доступность методов	public по умолчанию (не указывается) (при переопределении тоже public)
наследуются	С# поддерживается одиночное наследование для классов и множественное — для интерфейсов (при реализации интерфейса нужно обеспечить точное совпадение) Сначала всегда указывается имя базового класса, затем указывается интерфейс
Расширение интерфейса	Интерфейс наследуется интерфейсом
Имена	с прописной буквы I (не обязательно)

МОГУТ : абстрактные методы, обобщения свойства и индексаторы, а также события

```
interface IDo
{
    void Go();
    int Jumn(int a);
    void Sleep();
    int Energy { get; } // шаблон свойства
}
```

По умолчанию public,
реализации нет

Назначение :

задания общих характеристик объектов
различных иерархий – навязывание
контракта

- Интерфейс или класс может наследовать свойства нескольких интерфейсов, в этом случае предки перечисляются через запятую

```
interface IDo
{
    void Go();
    int Jumn(int a);
    void Sleep();
    int Energy { get; } // шаблон свойства
}

interface IKnow
{
    void Count();
    int Math();
}

interface IPosebel : IDo, IKnow
{
}
```

Реализация интерфейсов

```
interface IDo
{
    void Draw();
    void Jump( int a );
    //void Die();
    //int Energy {get; }
}
```

```
class Man : IDo
{
    public void Draw()
    {
        Console.WriteLine( "С уважением " + name );
    }
    public void Jump(int a)
    {
        if ( a > 0 ) Console.WriteLine( "Я прыгнул!" );
    }
    public Man(int a, int b, string m)
    {
        name = m;
        weigth = a;
        length = b;
    }

    string name;
    int weigth;
    int length;
}
```

МОЖНО
указать
доступ

► Обращение - через объект класса, через интерфейсную ссылку

```
static void Main()  
{
```

```
Man Вася = new Man( 50, 50, "Вася" ); // объект класса  
Вася.Draw(); // результат:
```

```
IDo Do = new Man( 10, 10, "Маша" ); // объект типа интерфейса  
Do.Draw(); // результат:
```

С# допускается объявлять переменные ссылочного интерфейсного типа, т.е. переменные ссылки на интерфейс. Переменная может ссылаться на любой объект, реализующий ее интерфейс.

- Присваивании ссылке на интерфейс объектов различных типов (классов), поддерживающих этот интерфейс

```
static void Act (IDo A )
{
    A.Draw();
}

static void Main()
{
    Man Вася = new Man( 50, 50, "Вася" ); // объект класса
    Вася.Draw(); // результат:

    IDo Do = new Man( 10, 10, "Маша" ); // объект типа интерфейса
    Do.Draw(); // результат:

    Act(Вася);
```

Второй способ реализации интерфейса в классе: явное указание имени интерфейса перед реализуемым элементом.

```
class Baby : IDo
{
    int IDo.Energy
    {
        get
        { return weight * length; }
    }
    void IDo.Draw()
    { Console.WriteLine("С уважением " + name ); }
    string name;
    int weight;
    int length;
}
```

соответствующий элемент
не входит в интерфейс
класса

```
static void Main()
{
    IDo Masha = new Baby();
    Masha.Draw (); // через объект типа интерфейса

    Baby Vasia = new Baby();
    Vasia.Draw(); // ошибка метод не входит в интерфейс класса
}
```

Конфликт при множественном наследовании

```
interface INotDo
{
    void Sleep();
}
```

Явная (explicit)
реализация

```
class A : IDo, INotDo
{
```

```
    void IDo.Sleep()
    { Console.WriteLine("Сплю " + name); }
```

```
    void INotDo.Sleep()
    {
        Console.WriteLine("Не сплю " + name);
    }
    string name;
}
```

не можем
использовать
модификатор
public

Обращение к интерфейсу

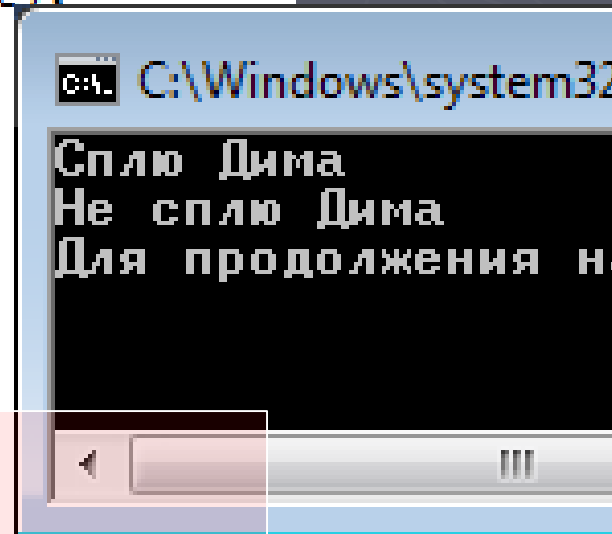
Все сказанное в отношении преобразования типов характерно и для интерфейсов

```
class A : IDo, INotDo
{
    void IDo.Sleep()
    { Console.WriteLine("Сплю " + name); }

    void INotDo.Sleep()
    {
        Console.WriteLine("Не сплю " + name);
    }
    public A(string m) { name = m; }
    string name;
}

static void Main()
{
```

```
    A Dima = new A("Дима" );
    ((IDo)Dima).Sleep(); // результат: Сплю Дима
    ((INotDo)Dima).Sleep(); //результат: Не сплю Дима
```



не требуется разное поведение

```
class Zombi : IDo, INotDo
{
    public void Sleep()
    { Console.WriteLine("Иду " + name); }

    public Zombi(string m) { name = m; }
    string name;
}
static void Main()
{
    Zombi Петя = new Zombi("Петя");
    Петя.Sleep();
}
```

Критерии	<u>Неявная (implicit) реализация</u>	<u>Явная (explicit) реализация</u>
Базовый синтаксис	<pre>interface IDo { void Sleep(); } public class ImplicitDo : IDo { public void Sleep() { } }</pre>	<pre>interface IDo { void Sleep(); } public class ExplicitDo : IDo { void IDo.Sleep() { } }</pre>
Видимость	<p>всегда является открытой (public)</p> <p>можно обращаться напрямую.</p> <pre>var imp = new ImplicitDo(); imp.Sleep();</pre>	<p>всегда закрыта (private)</p> <p>чтобы получить доступ необходимо приводить инстанцию класса к интерфейсу (upcast to interface).</p> <pre>var exp = new ExplicitDo(); ((IDo)exp).Sleep();</pre>
Полиморфизм	<p>может быть виртуальной (virtual), что позволяет переписывать эту реализацию в классах-потомках.</p>	<p>всегда статична</p> <p>не может быть переопределена (override) или перекрыта (new) в классах-потомках.</p>

Операции is

← Возвращает булевское значение, говорящее о том, можете ли вы преобразовать данное выражение в указанный тип

Оператор is никогда не генерирует исключение.

```
int j = 123;
object boxed = j;
object obj = new Object();
Boolean chekJ = boxed is int; //true
Boolean checkObj = obj is int; //false

Console.WriteLine("boxed {0} System.ValueType",
    boxed is ValueType ? "is" : "is not");
```

Операция as

позволяет преобразовывать тип в
определенный ссылочный тип с
применением следующего синтаксиса:

операнд as <тип>

Выполняется

- ▶ Если <операнд> имеет тип, заданный в <тип>.
- ▶ Если <операнд>, может быть неявно преобразован в <тип>.
- ▶ Если операнд <операнд>, может быть упакован в <тип>.

операнд as <тип>

```
class lection{
```

```
class CA {  
}
```

```
class CB : CA {  
}
```

```
public static void Main()  
{
```

```
    CA obj1 = new CA();
```

```
    CB obj2 = obj1 as CB;    // null
```

```
    CB obj3 = new CB();
```

```
    CA obj4 = obj3 as CA;    //OOP.lection.CB
```

```
}
```

as - никогда не
генерирует исключение


Если не сравнить с null и попытаться работать с
пустой ссылкой, возникнет исключение
System.NullReferenceException

as →

рассматривает только преобразования ссылок или преобразования с упаковкой

```
public class BaseType {}
public class DerivedType : BaseType {}
...
DerivedType derivedObj = new DerivedType();
BaseType baseObj1 = new BaseType();
BaseType baseObj2 = derivedObj;
DerivedType derivedObj2 = baseObj2 as DerivedType;
    if (derivedObj2 != null)
    {
        Console.WriteLine("Преобразование успешно");
    }
else
{
    Console.WriteLine("Преобразование не удалось")
}
```


Для null-ссылок оператор `is` всегда возвращает `false`, так как объекта, тип которого нужно проверить, не существует



```
string someStr = null;  
  
bool chekStr = someStr is string;
```

Контроль типов в CLR укрепляет безопасность, но при этом приходится жертвовать производительностью

```
if (someObj is Student)  
{  
    Student emma = (Student)someObj;  
}
```

```
Student ed = someObj as Student;
```

```
if (ed != null) { }
```

Работа с объектами через интерфейсы (преобразования)

- Проверка поддержки данного интерфейса

```
static void Act(object A)  
{  
    if (A is IDo)  
    {  
        IDo Actor = (IDo)A;  
        Actor.Sleep();  
    }  
}
```

```
static void Act(object A)
{
    if (A is IDo)
    {
        IDo Actor = (IDo)A;
        Actor.Sleep();
    }

    INotDo Actor2 = A as INotDo;
    if (Actor2 != null) Actor2.Sleep();
}
```

Переменной ссылки на интерфейс доступны только методы, объявленные в ее интерфейсе.

- Класс наследует все методы своего предка (интерфейсы). Он может переопределить (new), но обращаться к ним - через объект класса. Если использовать для обращения ссылку на интерфейс, вызывается не переопределенная версия

```
interface IBase
{
    void A() ;
}
class Base : IBase
{
    public void A() { }
}
class Derived: Base
{
    new public void A() { }
}

static void Main()
{
    Derived d = new Derived ();
    d.A(); // вызывается Derived.A();
    IBase id = d;
    id.A(); // вызывается Base.A();
}
```

если интерфейс реализуется с помощью виртуального метода класса, после его переопределения в потомке любой вариант обращения (через класс или через интерфейс) приведет к одному и тому же результату

```
interface IBase
{
    void A();
}
class Base : IBase
{
    public virtual void A() { }
}
class Derived: Base
{
    public override void A() { }
}

static void Main()
{
    Derived d = new Derived();
    d.A(); // вызывается Derived.AO;
    IBase id = d;
    id.A(); // вызывается Derived.AO;
```

- Метод интерфейса, реализованный явным указанием имени, объявлять виртуальным запрещается. При необходимости → переопределить в потомках его поведение:

```
interface IBase
{
    void A();
}
class Base : IBase
{
    void IBase.A() { A1(); }
    protected virtual void A1() { }
}
class Derived: Base
{
    protected override void A1() {}
}
```

```
interface IBase
{
    void A();
}
class Base : IBase
{
    void IBase.A() { } //не используется в Derived
}
class Derived : Base, IBase
{
    public void A() { }
}
```

- Существует возможность повторно реализовать интерфейс, указав его имя в списке предков класса наряду с классом-предком, уже реализовавшим этот интерфейс

Интерфейс и абстрактный класс

	Абстрактные классы	Интерфейсы
	одна иерархия	несколько иерархий
спецификатор доступа	задаются	public не явно
Поля	есть	нет
Наследник	может не определять – абстрактный	определяет все его элементы
В списке предков может быть	один абстр. класс	несколько интерфейсов
Объекты создавать	нельзя	нельзя
набор действий	имеет смысл только для конкретной иерархии	к различным иерархиям


```
var v = new int[] { 1, 2, 3 };
```

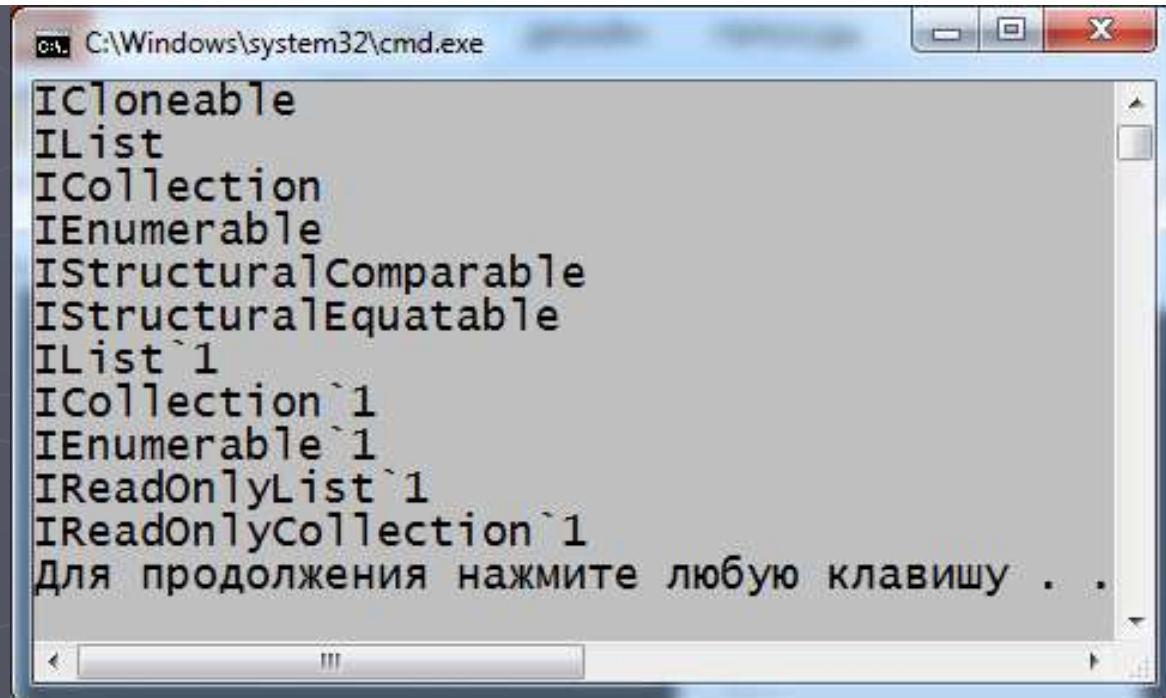
```
var t = v.GetType();
```

```
var i = t.GetInterfaces();
```

```
foreach (var tp in i)
```

```
    Console.WriteLine(tp.Name);
```

Интерфейсы, реализуются
целочисленным массивом



```
C:\Windows\system32\cmd.exe  
ICollectionable  
IList  
ICollection  
IEnumerable  
IStructuralComparable  
IStructuralEquatable  
IList`1  
ICollection`1  
IEnumerable`1  
IReadOnlyList`1  
IReadOnlyCollection`1  
Для продолжения нажмите любую клавишу . .
```

Стандартные интерфейсы .NET

Интерфейс	методы	Назначение	прим
ICloneable	object Clone()	Клонирование объектов (поверхностное или глубокое)	
IEnumerable	(IEnumerator) GetEnumerator();	перебор элементов необобщенной коллекции	основной для большинства коллекций
IEnumerator	Current bool MoveNext() void Reset()	перебор по необобщенной коллекции	можем перебирать объекты в цикле foreach
IComparable	int CompareTo(object obj)	Сравнение объектов Для выяснения порядка	
IComparer	int Compare(object o1, object o2);	Сравнение объектов	будут иметь большой приоритет
IDisposable	void Dispose()	механизм для освобождения	

Интерфейс	методы	Назначение	прим
IEquatable<T>	bool Equals(T other);	Сравнение объектов на равенство	
IStructuralEquatable	bool Equals(object, IEqualityComparer); int GetHashCode(...);	Проверка на равенство по значению	

```
class M : IComparable
{
    int n;

    public int CompareTo(object obj) // реализация интерфейса
    { return (int)obj - n; }

}
```

```
class Card : IComparable<Card>
{
    public string Name { get; set; }
    public int Sum { get; set; }
    public int CompareTo(Card p)
    {
        return this.Name.CompareTo(p.Name);
    }
}
```

компаратор объектов



```
class SumComparer : IComparer<Card>
{
    public int Compare(Card x, Card y)
    {
        if (x.Sum > y.Sum)
            return 1;
        else if (x.Sum < y.Sum)
            return -1;
        else
            return 0;
    }
}
```

```
Card visa = new Card { Name = "VISA", Sum = 122222 };
Card mastercard = new Card { Name = "Mastercard", Sum = 230 };

Card[] allMy = new Card[] { visa, mastercard};

Array.Sort(allMy, new SumComparer());

foreach (Card p in allMy)
{
    Console.WriteLine("{0}---{1}", p.Name, p.Sum);
}
```

не важно, реализует ли класс интерфейс IComparable или нет правила сортировки, установленные компаратором, будут иметь больший приоритет. В начале будут идти объекты, у которых сумма меньше



A screenshot of a Windows command prompt window. The title bar shows the path "C:\Windows\system32\cmd.exe". The command prompt displays the output of the program: "Mastercard---230" on the first line and "VISA---122222" on the second line. The text is in a monospaced font, and the window has a standard Windows XP-style title bar with minimize, maximize, and close buttons.

```
class Person : ICloneable
{
    public string Name { get; set; }
    public int Age { get; set; }
    public object Clone()
    {
        return new Person { Name = this.Name,
                             Age = this.Age };
    }
}
```

```
Person p1 = new Person { Name = "Tom", Age = 23 };
Person p2 = (Person)p1.Clone();
```

- Работа с перечислителями
- Пусть есть магазин с товарам

```
public class Shop
{
    private string[] _items = new string[0];

    public int ItemsCount {
        get {
            return _items.Length;
        }
    }

    public void AddItem(string item) {
        Array.Resize(ref _items, ItemsCount + 1);
        _items[ItemsCount - 1] = item;
    }

    public string GetItem(int index) {
        return _items[index]; }
}
```


► Сделаем магазин перечисляемым

```
public interface IEnumerable {  
    IEnumerator GetEnumerator();  
}
```

```
public interface IEnumerator {  
    object Current { get; }  
    bool MoveNext();  
    void Reset();  
}
```

```
public class Shop : IEnumerable
{
    // опущены элементы
    private string[] _items = new string[0];
    private class ShopEnumerator : IEnumerator
    {
        private readonly string[] _data; // локальная копия данных
        private int _position = -1;      // текущая позиция в наборе

        public ShopEnumerator(string[] values) {
            _data = new string[values.Length];
            Array.Copy(values, _data, values.Length);
        }

        public object Current { get { return _data[_position]; } }
        public bool MoveNext() {
            if (_position < _data.Length - 1)
            {
                _position++; return true;
            } return false;
        }
        public void Reset() { _position = -1; }
    }
    public IEnumerator GetEnumerator() {
        return new ShopEnumerator(_items);    } }
}
```

```
var shop = new Shop();  
shop.AddItem("computer");  
shop.AddItem("monitor");  
  
foreach (string s in shop)  
    Console.WriteLine(s);
```

интерфейс IDisposable

```
public class ClassWithDispose : IDisposable
{
    public void DoSomething() {
        Console.WriteLine("I am working...");
    }

    public void Dispose() {
        // здесь должен быть код освобождения управляемых ресурсов
        Console.WriteLine("Bye!");
    }
}
```

using (получение-ресурса) вложенный-оператор

```
using (ClassWithDispose x = new ClassWithDispose())
{
    x.DoSomething();
    // компилятор C# поместит сюда вызов x.Dispose()
}
```

ИТОГО

кл. слово	назначение
interface	вводит в обращение имя метода
virtual	первая реализация метода
override	еще одна реализация
sealed	последняя реализация

Структуры

- ▶ 1) struct
- ▶ 2) Может иметь конструктор с парам.
- ▶ 3) нельзя определить конструктор, используемый по умолчанию (конструктор без параметров). Он определяется для всех структур автоматически и не подлежит изменению
- ▶ 4) Объект структуры может быть создан с помощью **оператора new (или нет)**
- ▶ 5) размещение в стеке

- ▶ 6) Нельзя инициализировать поля структуры при объявлении
- ▶ 7) нет автоматической инициализации полей компилятором

```
struct Time
{
    private int hours = 0; // ошибка в ходе компиляции
    private int minutes;
    private int seconds;
}
```

- ▶ 8) **структуры** не поддерживают **наследование**
- ▶ Назначение : повышении эффективности и производительности программ (тип значения)

Перечисления

- ▶ набор логически связанных констант
- 1) тип перечисления - целочисленный тип (byte, int, short, long)
- 2) По умолчанию используется тип int

```
enum MathOperation  
{  
    Add ,  
    Subtract,  
    Multiply,  
    Divide  
}
```


3) каждому элементу перечисления присваивается целочисленное значение, 0, 1 т.д.

4) Можно определять явным образом

```
enum MathOperation :short
{
    Add = 4,
    Subtract,      //5
    Multiply =9 ,  //9
    Divide         //10
}
```

```
class Math
{
    MathOperation operation = MathOperation.Multiply;
}
```

Ключевое слово	interface	abstract class	class	sealed	struct
abstract	Нет	Да	Нет	Нет	Нет
new	Да	Да	Да	Да	Нет
override	Нет	Да	Да	Да	Нет
private	Нет	Да	Да	Да	Да
protected	Нет	Да	Да	Да	Нет
public	Нет	Да	Да	Да	Да
sealed	Нет	Да	Да	Да	Нет
virtual	Нет	Да	Да	Нет	Нет