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

*Наследование* – это механизм получения нового класса на основе уже существующего

элементы данных старого класса

методы старого класса

БАЗОВЫЙ КЛАСС

элементами данных нового класса

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

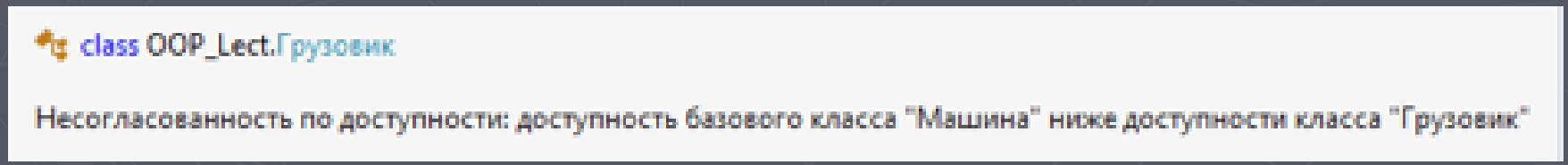
члены базового класса (private – недоступны)

5) не наследуются конструкторы базового класса (но можно вызвать)

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{ }
```



```
public class A
{
    private int _value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return _value;
        }
    }
}

public class C : A
{
    //    public int GetValue()
    //    {
    //        return _value;
    //    }
}

public class AccessExample
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}
// The example displays the following output:
//       10
```

возникает ошибка компилятора CS0122:
"A._value недоступен из-за уровня защиты"

- 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());//a12 имеет тип
ColorPoint
            Console.WriteLine(a12.Sum());
            Console.WriteLine(ca100.Sum());
            //Console.WriteLine(a12.Mult());//ошибка, нет метода
        }
```

# 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(ca100.Sum());
            Console.WriteLine((a12 as ColorPoint).Mult());
        }
   }
```

# Ключевое слово base

Конструкторы не наследуются

Автоматически вызывается конструктор базового класса без параметров

System.Object

```
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);

}
```

Конструкторы базовых классов вызываются, начиная с самого верхнего уровня

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

System.Object

```
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);


}
```

Ошибка – отсутствует аргумент базового конструктора

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

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

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

# Стратегии наследования

- Обычное наследование всех членов базового класса в классе-наследнике
- Переопределение членов базового класса в классе-наследнике (полиморфизм)
- Сокрытие 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
        }
```

# Сокрытие 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 "Ierarhi.ColorPoint.x" скрывает наследуемый член "Ierarhi.Po
используйте ключевое слово new.

```
ColorPoint -78Point 10  20
```

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;  } // -48
       }
```

# 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

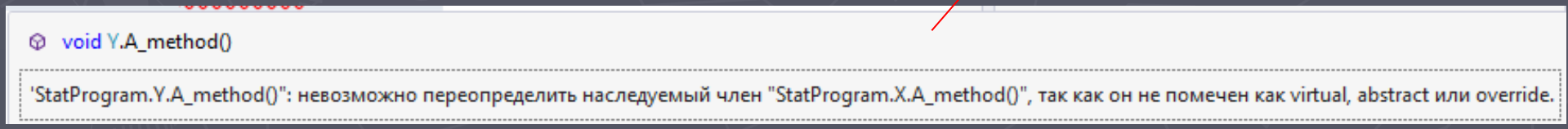
- 1) Переопределенный виртуальный метод должен обладать таким же набором параметров, как и одноименный метод базового класса.
- 2) не может быть static или abstract
- 3) вызывается ближайший вариант, обнаруживаемый вверх по иерархии (многоуровневая)

# Если не virtual переопределять нельзя

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

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

        }
```



- 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 (статический)

*позднее связывание* (только для методов классов) – связанное с формированием кода на этапе выполнения. Вызов метода происходит на основании типа объекта, а не типа ссылки на базовый класс (динамический)

Необходимые условия для реализации позднего связывания:

- классы должны образовывать иерархию наследования;
- в классах должны быть методы с одинаковой сигнатурой. Элементы (методы) производных классов должны перекрывать (override) соответствующие элементы (методы) базовых классов;
- элементы (методы) класса должны быть виртуальными, то есть должны быть обозначены ключевыми словами virtual, override.

| 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 |

# C#9

- Начиная с версии C# 9.0, методы override поддерживают ковариантные типы возвращаемых значений. В частности, тип возвращаемого значения метода override может быть производным от типа возвращаемого значения соответствующего базового метода. В C# 8.0 и более ранних версий типы возвращаемых значений метода override и переопределенного базового метода должны быть одинаковыми

# C#9

```
public class EmployeeSkills
{
  public bool CanSendEmails { get; set; }
}

public class DeveloperSkills : EmployeeSkills
{
  public bool KnowsDotNet { get; set; }
}

public class Employee
{
  public string? FirstName { get; set; }
  public virtual EmployeeSkills GetSkills()
  {
    return new EmployeeSkills
    {
      CanSendEmails = true
    };
  }
}

public class Developer : Employee
{
  public override DeveloperSkills GetSkills()
  {
    return new DeveloperSkills
    {
      CanSendEmails = true,
      KnowsDotNet = true
    };
  }
}
```

# 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 можем переопределить метод Sum в классе, унаследованном от ColorPoint.

**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()
            {
                //...
            }
        }
```

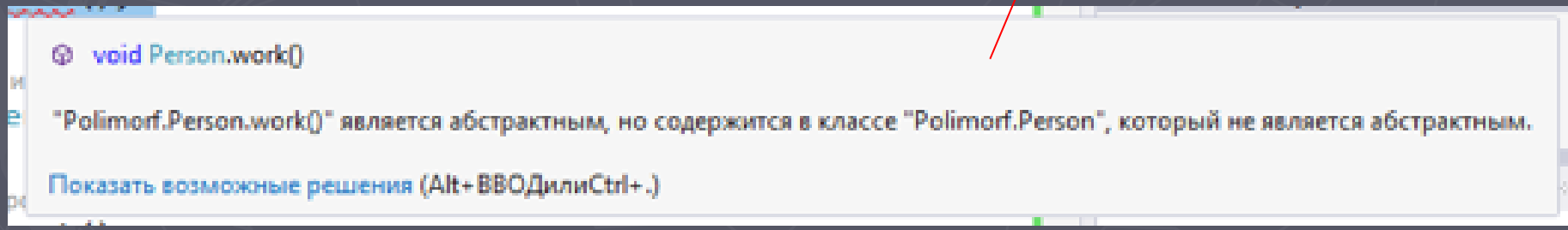
производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе

# Абстрактные классы

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

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

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



- Конструктор в абстрактном классе

```
Transport car = new Car("машина");
Transport ship = new Ship("корабль");
Transport aircraft = new Aircraft("самолет");

car.Move();          // машина движется
ship.Move();         // корабль движется
aircraft.Move();     // самолет движется
abstract class Transport
{
    public string Name { get; }
    // конструктор абстрактного класса Transport
    public Transport(string name)
    {
        Name = name;
    }
    public void Move() =>Console.WriteLine($"{Name} движется");
}
// класс корабля
class Ship : Transport
{
    // вызываем конструктор базового класса
    public Ship(string name) : base(name) { }
}
// класс самолета
class Aircraft : Transport
{
    public Aircraft(string name) : base(name) { }
}
// класс машины
class Car : Transport
{
    public Car(string name) : base(name) { }
}
```

производные классы должны в своих конструкторах вызвать этот конструктор.

абстрактными могут быть:

- Методы
- Свойства
- Индексаторы
- События

не должны иметь модификатор private

# Абстрактные свойства

```
abstract class Transport
{
    // абстрактное свойство для хранения скорости
    public abstract int Speed { get; set; }
}
// класс корабля
class Ship: Transport
{
    int speed;
    public override int Speed
    {
        get => speed;
        set => speed = value;
    }
}

class Aircraft : Transport
{
    public override int Speed { get; set; }
}
```

похоже на определение автосвойств
но это не автосвойство

можем переопределить это свойство, сделав его полноценным свойством (как в классе Ship), либо же сделав его автоматическим

# Отказ от реализации абстрактных членов

```
Transport tesla = new Auto();
tesla.Move();              // легковая машина едет
abstract class Transport
{
    public abstract void Move();
}
// класс машины
abstract class Car : Transport{}

class Auto: Car
{
    public override void Move()
    {
        Console.WriteLine("легковая машина едет");
    }
}
```

производный класс также должен быть определен как абстрактный:

# Свойства abstract методов

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

- 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 Iимя_интерфейса [ : предки]
          Тело интерфейса[; ]
```

| Свойства | Интерфейс |
| --- | --- |
| не может содержать | Не статические поля, операции, конструкторы, деструкторы, типы, |
| может содержать | абстрактные методы, обобщения свойства и индексаторы, а также события, делегаты<br>Default методы,<br>Статические поля, методы (с реализацией) и константы (с C# 8.0) |
| Доступность методов | **public по умолчанию (не указывается )**<br>(при переопределении тоже public) Могут быть private и protected |
| наследуются | C# поддерживается одиночное наследование для классов и множественное — для интерфейсов<br>(при реализации интерфейса нужно обеспечить точное совпадение)<br>Сначала всегда указывается имя базового класса, затем указывается интерфейс |
| Расширение интерфейса | Интерфейс наследуется интерфейсом |
| Имена | с прописной буквы I (не обязательно) |

- 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

# Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию.

```
IMovable tom = new Person();
Car tesla = new Car();
tom.Move();     // Walking
tesla.Move();   // Driving
interface IMovable
{
    void Move() => Console.WriteLine("Walking");
}
class Person : IMovable { }
class Car : IMovable
{
    public void Move() => Console.WriteLine("Driving");
}
```

Если класс не реализует метод, будет применяться реализация по умолчанию.

```
Person tom = new Person();
tom.Move();     // Ошибка - метод Move не определен в классе Person
```

# 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:
```

Поскольку класс Man реализует интерфейс IDo
то переменная типа IDo может хранить ссылку
на объект типа Man:

C# 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

```
public interface Imove
{
    Ссылок: 3
    void Move();
    Ссылок: 2
    public static void asd(Imove move) { move.Move(); }
}
Ссылок: 2
public class ColorPoint : Imove
{
    Ссылок: 2
    public void Move() => Console.WriteLine("CoPo");
}
Ссылок: 2
public class NotColorPoint : Imove
{
    Ссылок: 2
    public void Move() => Console.WriteLine("NoCoPo");
}
Ссылок: 0
private static void Main(string[] args)
{
    ColorPoint mo = new ColorPoint();
    Imove.asd(mo);//CoPo
    NotColorPoint nmo = new NotColorPoint();
    Imove.asd(nmo);//NoCoPo
}
```

```
public interface Imove
{
    Ссылок: 3
    void Move() => Console.WriteLine("Base");
    Ссылок: 2
    public static void asd(Imove move) { move.Move();
}
Ссылок: 1
public class ColorPoint : Imove
{
    Ссылок: 2
    public void Move() => Console.WriteLine("CoPo");
}
Ссылок: 1
public class NotColorPoint : Imove
{
    Ссылок: 2
    public void Move() => Console.WriteLine("NoCoPo")
}
Ссылок: 0
private static void Main(string[] args)
{
    Imove mo = new ColorPoint();
    Imove.asd(mo);//CoPo
    Imove nmo = new NotColorPoint();
    Imove.asd(nmo);//NoCoPo
}
```

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

```
public interface Imove
{
    Ссылок: 2
    void Move();
}

Ссылок: 3
public class NotColorPoint : Imove
{
    Ссылок: 2
    void Imove.Move() => Console.WriteLine("NoCoPo");
}
Ссылок: 0
private static void Main(string[] args)
{
    Imove imo = new NotColorPoint();
    imo.Move();//NoCoPo
    NotColorPoint nmo = new NotColorPoint();
    nmo.Move();//error
}
```

соответствующий элемент не входит в класс

```
interface IAction
{
    void Move();
}
class BaseAction : IAction
{
    void IAction.Move() => Console.WriteLine("Move in Base Class");
}
BaseAction baseAction1 = new BaseAction();

// baseAction1.Move();  // ! Ошибка - в BaseAction нет метода Move
// необходимо приведение к типу IAction
// небезопасное приведение
((IAction)baseAction1).Move();
// безопасное приведение
if (baseAction1 is IAction action) action.Move();
// или так
IAction baseAction2 = new BaseAction();
baseAction2.Move();
```

# Конфликт при множественном наследовании

```
interface INotDo
{
    void Sleep();

}

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

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

понадобиться явная реализация интерфейса

**Явная (explicit) реализация**

не можем использовать модификатор public

# Обращение к интерфейсу

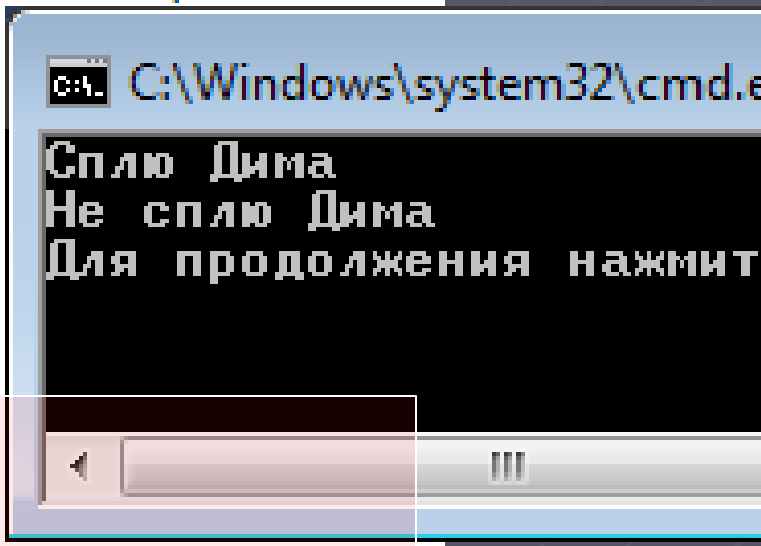
Все сказанное в отношении преобразования типов характерно и для интерфейсов

```
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();
```

# Default Interface Members

```
public interface IAuth
    {
        string Password { get; set; }

        public void ShowPassword()
        {
            Console.WriteLine($" {Password}");
        }
    }

public class Person : IAuth
    {
        public string Password { get; set; }
    }
class Program
    {
        static void Main(string[] args)
        {
            Person olga = new Person() { Password = "root" };
            olga.ShowPassword(); // ошибка, не удалось найти определения
            ((IAuth)olga).ShowPassword(); // ok
```

default методы доступны только через сам интерфейс

# Наследование интерфейсов с default реализацией

```
 public interface IAuth  {
        string Password { get; set; }
       public void ShowPassword() => Console.WriteLine($" {Password}");
        }
    }
public interface ISecurity : IAuth  {
       public new void ShowPassword() =>Console.WriteLine($"********");

    }
    public class Person : ISecurity{
        public string Password { get; set; }
    }
    class Program
    { static void Main(string[] args)
        {  Person olga = new Person() { Password = "root" };
         ((IAuth)olga).ShowPassword(); // form IAuth - root
         ((ISecurity)olga).ShowPassword(); // forme ISecurity -*******
```

# Явная реализация default метода

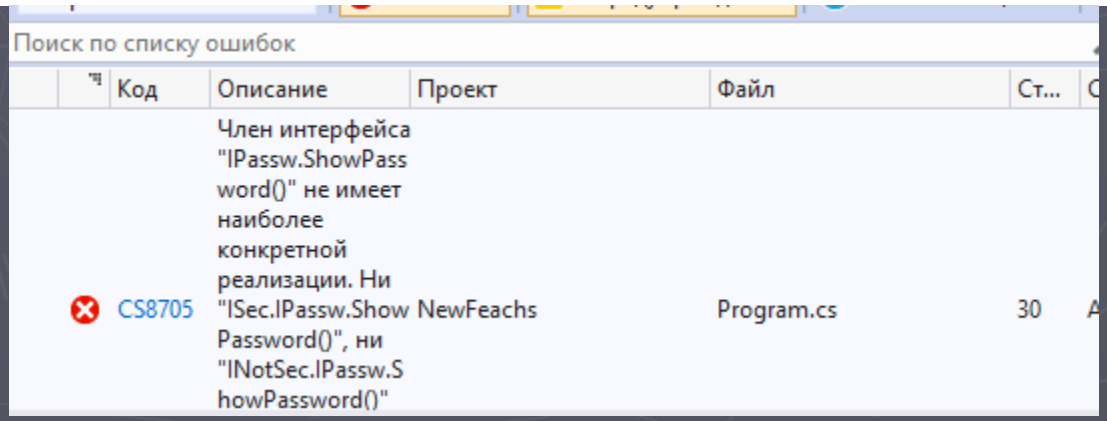
```
public interface IAuth
    {  string Password { get; set; }
       public void ShowPassword() =>Console.WriteLine($" {Password}");
    }
public interface ISecurity : IAuth
    {
        void IAuth.ShowPassword() =>Console.WriteLine($"********");
    }

    public class Person : ISecurity
    {  public string Password { get; set; }
    }
    class Program
    {  static void Main(string[] args)
        {
          Person olga = new Person() { Password = "root" };
          ((IAuth)olga).ShowPassword(); // form  ISecurity - ******
          ((ISecurity)olga).ShowPassword(); // forme ISecurity -*****
```

# Конфликт

```
public interface IPassw
    {
        void ShowPassword();
    }
    interface ISec : IPassw {
        void IPassw.ShowPassword() => Console.WriteLine("***");
    }
    interface INotSec : IPassw
    {
        void IPassw.ShowPassword() => Console.WriteLine("root");
    }

    class TwoInterface : ISec, INotSec { }
```



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

# 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

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 и

```
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.AO;
IBase id = d;
id.A(); // вызывается Base.AO;
```

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

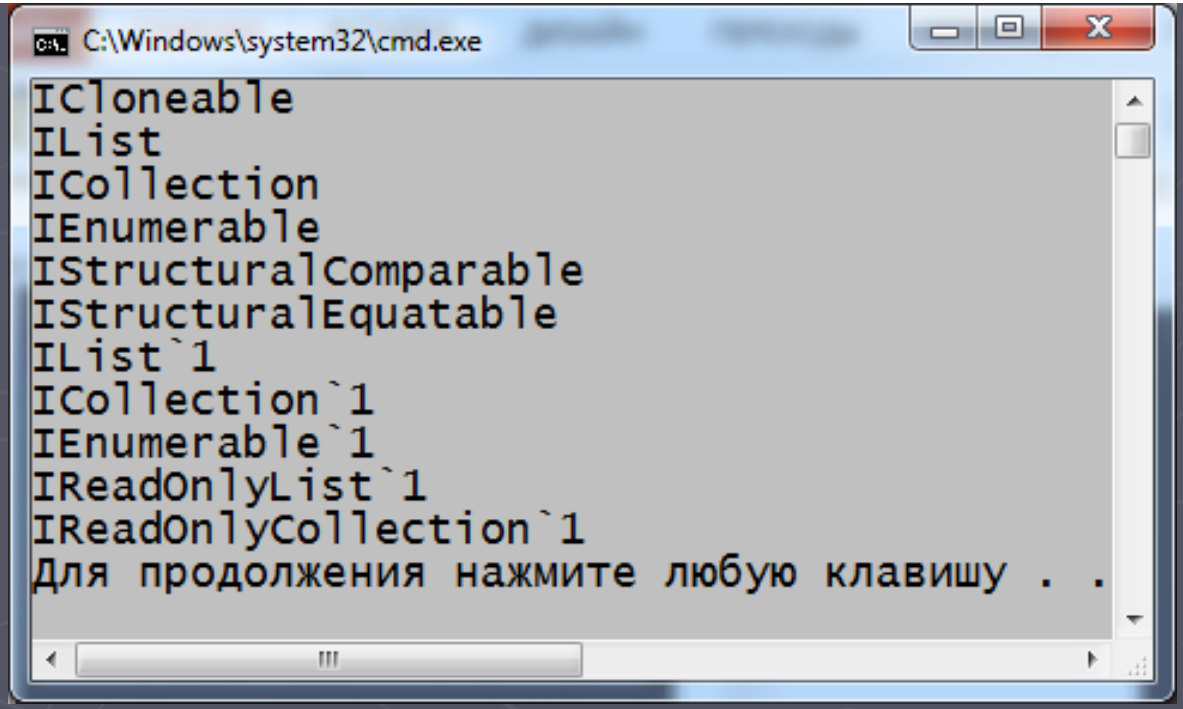
| | Абстрактные классы | Интерфейсы |
|---|---|---|
| | одна иерархия | несколько иерархий |
| модификаторы доступа | задаются | public не явно, могут быть другие модификаторы |
| Поля | есть | Могут быть |
| Наследник | может не определять – абстрактный | Должен реализовывать все элементы |
| В списке 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



# Стандартные интерфейсы .NET

| Интерфейс | методы | Назначение | прим |
|---|---|---|---|
| ICloneable | object Clone() | Клонирование объектов (поверхностное или глубокое) | |
| lEnumerable | (IEnumerator) GetEnumerator(); | перебор элементов необобщенной коллекции | основна для большинства коллекций |
| IEnumerator | Current<br>bool MoveNext()<br>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 больший приоритет |
| | | | |

| 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 | |
| IDisposable | void Dispose() | механизм для освобождения управляемых ресурсов | |
| | | | |

```
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;
         }
 }
```

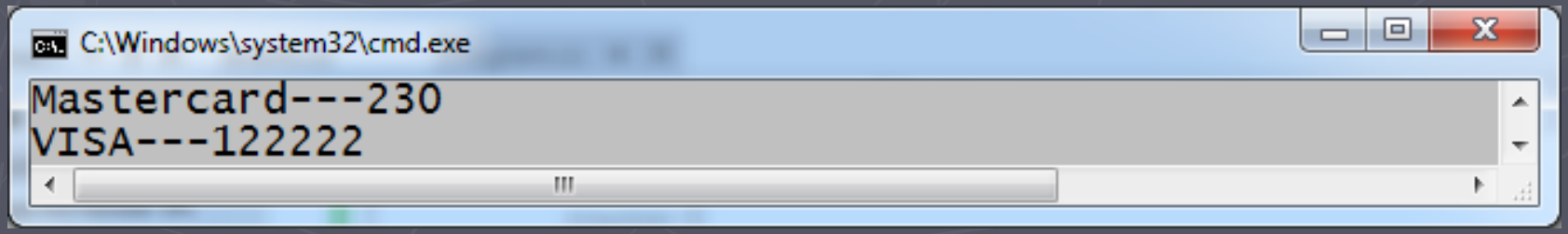
```
Card visa = new Card { Name = "VISA", Sum = 122222 };
Card mastercard = new Card { Name = "Mastercard", Sum = 230 };

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

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

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

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



```
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

имеет метод, возвращающий ссылку на другой интерфейс - перечислитель

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

```
public interface IEnumerator
{
    bool MoveNext(); // перемещение на одну позицию вперед в контейнере элементов
    object Current {get;}  // текущий элемент в контейнере
    void Reset(); // перемещение в начало контейнера
}
```

Метод MoveNext() перемещает указатель на текущий элемент на следующую позицию в последовательности. Если последовательность еще не закончилась, то возвращает true. Если же последовательность закончилась, то возвращается false.

Свойство Current возвращает объект в последовательности, на который указывает указатель.

Метод Reset() сбрасывает указатель позиции в начальное положение.

```
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) **до C# 10** 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) или** default
- **5)** размещение в стеке

```
Person tom = new Person();  // вызов конструктора
// или так
// Person tom = new();

tom.name = "Tom";    // изменяем значение по умолчанию в поле name

tom.Print();     // Имя: Tom  Возраст: 0

struct Person
{
    public string name;
    public int age;

    public void Print()
    {
        Console.WriteLine($"Имя: {name}  Возраст: {age}");
    }
}
```

- 6) **До C# 10** Нельзя инициализировать поля структуры при объявлении
- 7) **До C# 10** 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)

с версии C# 10, мы можем напрямую инициализировать поля структуры при их определении

```
Person tom = new Person();
tom.Print();    // Имя:Tom  Возраст: 1

struct Person
{
    // инициализация полей значениями по умолчанию - доступна с C#10
    public string name = "Tom";
    public int age = 1;
    public Person() { }
    public void Print() =>Console.WriteLine($"Имя: {name}  Возраст: {age}");
}
```

9) Могут реализовывать интерфейсы

10 ) нельзя объявить деструктор (метод завершения) в типе структуры

## readonly struct

тип структуры является неизменяемым. Все члены данных структуры readonly должны быть доступны только для чтения следующим образом:

Любое объявление поля должно иметь readonly модификатор

Любое свойство, включая автоматически реализуемое, должно быть доступно только для чтения. В C# 9.0 и более поздних версиях свойство может иметь init метод доступа .

код определяет readonly структуру с установщиками свойств только для инициализации, доступными в C# 9.0 и более поздних версиях:

```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}
```

```
Person tom = new();
Person bob = new("Bob");
Person sam = new("Sam", 25);

tom.Print();    // !!!! Имя:   Возраст: 0
bob.Print();    // Имя: Bob  Возраст: 1
sam.Print();    // Имя: Sam  Возраст: 25

struct Person
{
    public string name;
    public int age;

    public Person(string name = "Tom", int age = 1)
    {
        this.name = name;
        this.age = age;
    }
    public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");
}
```

```
Person tom = new(); // по прежнему используется конструктор без параметров по умолчанию
tom.Print();    // !!!! Имя:   Возраст: 0
```

# с версии C# 10 мы можем определить свой конструктор без параметров

```
Person tom = new();

tom.Print();    // Имя: Tom  Возраст: 37

struct Person
{
    public string name;
    public int age;

    public Person()
    {
        name = "Tom";
        age = 37;
    }
    public void Print() => Console.WriteLine($"Имя: {name}  Возраст: {age}");
}
```

до версии C# 11 при определении конструктора структуру в нем необходимо было инициализировать все поля структуры, начиная с версии C# 11 это делать необязательно

```
struct Person
{
public string name;
public int age;
public static int a;

public void Print()
{
    Console.WriteLine($"Имя: {name}  Возраст: {age}");
}
 static Person()
 {    a =18;
      Console.WriteLine("статический конструктор");
 } //статический  конструктор
 public static void P()
{
    Console.WriteLine($"Возраст: {a}");
}
}
static void Main() {
   Person tom = new Person();  // вызов конструктора
   tom.name = "Tom";   // изменяем значение по умолчанию в поле name
   tom.Print();     // Имя: Tom  Возраст: 0
   Console.WriteLine(Person.a);
   Person.P();
}
```

```
статический конструктор
Имя: Tom  Возраст: 0
18
Возраст: 18
```

# 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 | Да | Да | Да | Да | Да |
| protected | Да | Да | Да | Да | Нет |
| public | Да | Да | Да | Да | Да |
| sealed | Нет | Да | Да | Да | Нет |
| virtual | Нет | Да | Да | Нет | Нет |