Индексаторы и операции классов.

Лекция №5

Индексаторы.

Если в классе есть скрытое поле, представляющее собой набор элементов, например, массив, то в нем можно определить индексатор, обеспечивающий индексированный доступ к элементам поля с использованием имени объекта и номера элемента в квадратных скобках.

Индексаторы могут характеризоваться одной или несколькими размерностями.

Индексатор – это особый вид свойства.

Одномерные индексаторы.

```
спецификатор тип  this [тип_индекса индекс]
   {
    get  {код аксессора для получения данных}
    set {код аксессора для установки данных}
    }
```

Спецификаторы аналогичны спецификаторам свойств и методов. Нельзя использовать **static**. Чаще всего используют **public**.

тип — это тип элемента, к которому предоставляется доступ посредством индексатора.

Обычно он соответствует базовому типу элементов индексируемого поля, например, типу элементов массива.

Но индексаторы могут возвращать значение другого типа, отличающегося от типа данных в списке элементов.

тип_индекса не обязательно должен быть **int**, но поскольку индексаторы обычно используются для обеспечения индексации массивов, целочисленный тип — наиболее подходящий. Но в качестве индекса можно использовать даже строки.

При использовании индексатора аксессоры вызываются автоматически, и в качестве параметра оба аксессора принимают **индекс**. Если индексатор стоит слева от оператора присваивания, вызывается аксессор **set** и устанавливается элемент, заданный параметром **индекс**. В противном случае вызывается аксессор **get** и возвращается значение, соответствующее параметру **индекс**.

Например, пусть в классе **Massiv** определены закрытое поле

```
double[ ] x;
```

и индексатор

```
public double this [int i]
 {  get { return x[i];}
    set { x[i]=value;}
 }
```

И пусть **Y** – объект класса **Massiv.**

Тогда к элементам поля **x** объекта **Y** можно обращаться следующим образом:

```
Y[1] = 5.3;
Console.WriteLine(Y[i]);
```

Одно из достоинств индексатора состоит в том, что он позволяет точно управлять характером доступа к массиву, предотвращая попытки некорректного доступа.

Пример 1. В двух одномерных массивах заменить нулями все элементы , которые больше суммы одного элемента из первого массива и одного элемента из второго массива с заданными номерами.

```
class Massiv
  {
    double[ ] a;

    public Massiv(int n)
          {
             a = new double[n];
          }
```

```
public double this[int i]
 {
    get
              {
                 if (i >= 0 && i < a.Length) return a[i];
                 else throw new Exception("Плохой индекс " + i);
              }
  set
          {
             if (i >= 0 && i < a.Length) a[i] = value;
             else throw new Exception("Плохой индекс " + i);
          }
 }
```

```
public int Length
        { get { return a.Length; } }

public void vyvod(string zagolovok)
        {
            Console.WriteLine(zagolovok);
            foreach (double x in a)
                Console.WriteLine(x);
        }
```

```
public void vvod(string name)
        {
            Console.WriteLine("Введите элементы массива " + name);
            for (int i = 0; i < a.Length; i++)
            {
                a[i] = Convert.ToDouble(Console.ReadLine());
            }
        }

    }
```

```
class Program
    {
      static void Main(string[] args)
       {
      Console.WriteLine("ВВедите размер  массива A");
      Massiv A = new Massiv(Convert.ToInt32(Console.ReadLine()));
      A.vvod("A");

     Console.WriteLine("ВВедите размер  массива B");
      Massiv B = new Massiv(Convert.ToInt32(Console.ReadLine()));
      B.vvod("B");
```

```
A.vyvod("Массив A"); B.vyvod("Массив B");

Console.WriteLine("BВедите номер элемента  массива A");
 int n1 = Convert.ToInt32(Console.ReadLine());


Console.WriteLine("BВедите номер элемента  массива B");
 int n2 = Convert.ToInt32(Console.ReadLine());

  try
     {
          double    s = A[n1] + B[n2];
```

```
    for (int i = 0; i < A.Length; ++i)
                { if (A[i] > s) A[i] = 0; }
    for (int i = 0; i < B.Length; ++i)
                { if (B[i] > s) B[i] = 0; }
  }
catch (Exception e) { Console.WriteLine(e.Message); }
A.vyvod("Массив A"); B.vyvod("Массив B");
          Console.ReadKey();

      }
   }
```

Пример 2. Демонстрирует возможность использования в индексаторе в качестве индекса строки, а также перегрузку индексаторов.

```
class Sotrudnic
    {
        string fam;
        double[ ] zarplata = new double[6];
        string[ ] mes = new string[ ]
            { "январь", "февраль", "март", "апрель", "май", "июнь" };
    public Sotrudnic(string fam)
            { this.fam = fam; }
```

```
        public string Fam
                { get { return fam; } }
public double this[int i]
        {
           get
           {
              if (i >= 0 && i < 6) return zarplata[i];
              else throw new Exception("Неправильный номер
                                                    месяца!");
           }
```

```
set
        {
          if (i >= 0 && i < 6) zarplata[i]=value;
          else throw new Exception("Неправильный номер месяца!");
            }
        }

    public double this[string m]
            {
```

```
get
      {
    if (Array.IndexOf( mes, m) >= 0)
        return zarplata[Array.IndexOf(mes, m)];
     else throw new Exception("Неправильный  месяц " + m );
          }
set
          {
              if (Array.IndexOf(mes, m) >= 0)
                          zarplata[Array.IndexOf(mes, m)] = value;
             else throw new Exception("Неправильный месяц "+m);
          }
      } }
```

```
class Program
    {
        static void Main(string[] args)
        {
            string[ ] mesjac = new string[ ]
             { "январь", "февраль", "март", "апрель", "май", "июнь" };


          Sotrudnic rab1 = new Sotrudnic("Карлсон");

          rab1[0] = 300; rab1["февраль"] = 500;
          rab1[2] = 800; rab1["июнь"] = 1000;
```

```
Console.WriteLine("Зарплата сотрудника по фамилии "+rab1.Fam);
   foreach (string m in mesjac)
         Console.WriteLine(m+"   "+rab1[m]);

try
         {
           Console.WriteLine("За какой месяц хотите узнать  зарплату сотрудника " + rab1.Fam+"?");
            string   mmm = Console.ReadLine();
           Console.WriteLine(rab1[mmm]);
         }
```

```
catch (Exception e) { Console.WriteLine(e.Message); }

            Console.ReadKey();
        }
    }
```

Результат работы программы:

```
Зарплата сотрудника  по фамилии Карлсон
январь   300
февраль   500
март   800
апрель   0
май   0
июнь   1000
За какой месяц хотите узнать  зарплату сотрудника Карлсон?
сентябрь
Неправильный  месяц! сентябрь
```

Индексаторы можно использовать для доступа к различным полям класса, главное, чтобы они различались типом индекса.

Например, класс Sotrudnic можно дополнить индексатором:

```
public string this[char h]
    { get
      { switch (h)
        { case 'a': return mes[0];
          case 'b': return mes[1];
          case 'c': return mes[2];
          case 'd': return mes[3];
          case 'e': return mes[4];
          case 'f': return mes[5];
          default: return "0";
        }
      }
    }
```

Тогда результатом следующего фрагмента программы

```
for (char s = 'a'; s <= 'f'; ++s  )
        Console.WriteLine(rab1[s]);
```

где rab1 – объект класса Sotrudnik будет:

январь

февраль

март

апрель

май

июнь

Можно также создать индексатор класса для получения доступа к разным полям класса, которые не являются массивами.

Например, пусть класс Men описан след. образом:

```
class Men
  { double ves, rost, vozrast;
    public double this[int i]
      { get
        {   switch (i)
                  { case 0:return ves;
                  case 1: return rost;
                  case 2: return vozrast;
                  default: return 0; }  }
```

```
set
        {
            switch (i)
            {
                case 0: ves = value; break;
                case 1: rost = value; break;
                case 2: vozrast = value; break;
            }
        }
    }
}
```

Тогда в программе можно посредством индексатора получить доступ к полям класса Men:

```
Men petja = new Men( );
petja[0] = 89; petja[1] = 187.5; petja[2] = 25;
Console.WriteLine("Вес: " + petja[0] + "  Рост: " + petja[1] +
                  "  Возраст:  " + petja[2]);
```

Можно также создать индексатор для доступа к полям различного типа:

пусть поле **vozrast** имеет тип **int**.

Тогда индексатор мог бы иметь следующий вид:

```
public object this[int i]
   { get
      { switch (i)
         { case 0:return ves;
         case 1: return rost;
         case 2: return vozrast;
         default: return 0;
         }
      }
```

```
set
        {
            switch (i)
            {
                case 0: ves = (double) value; break;
                case 1: rost =(double) value; break;
                case 2: vozrast = (int)value; break;
            }
        }
    }
```

Использование в программе:

```
petja[0] = 89d;     petja[1] = 187.5d;     petja[2] = 25;
Console.WriteLine("Вес: " + petja[0] + "  Рост: " + petja[1] +
                  "  Возраст:  " + petja[2]);
```

Индексатор не обязательно возвращает значение базового поля.

```
class F_sin
   { double x;
      public F_sin(double xx)
      { x = xx; }

     public double this[double xx]
           { get { return Math.Sin(xx); }    }
```

```
public object this[int n]
    { get
        {
         double S=0;
         for (int j= 0; j <= n; ++j)
         {double f=1; for (int k=1; k <= (2*j+1);++k) f=f*k;
            S=S+Math.Pow(-1, j)*Math.Pow(x, 2*j+1)/f;}
         return S;
        }
    }
```

n-я частичная сумма ряда Тейлора

$$\sin x = \sum_{j=1}^{\infty} \frac{(-1)^j x^{2j+1}}{(2j+1)!}$$

```
class Program
    {
        static void Main(string[] args)
        {
            F_sin sin = new F_sin(2);


             Console.WriteLine("sin(2)= " + sin[2d]  + "  100-я частичная сумма: " + sin[100]);


            Console.ReadKey();
        }
    }
```

Индексатор может быть совсем не связан с каким либо полем класса:

```
class Pow2
   {
      public ulong this[int i]
      {
         get
         { if (i >= 0 && i <= 100) return (ulong) Math.Pow(2, i);
            else throw new Exception("Ошибка!!!");
         }
      }
   }
```

```
class Program
    {
        static void Main(string[] args)
        {
            Pow2 pow2 = new Pow2();
            for (int i = 5; i <= 10; ++i) Console.WriteLine(
                                        "2 в степени "+i+" = "+pow2[i]);
            Console.ReadKey();
        }
    }
```

Многомерные индексаторы.

спецификатор тип this [тип1 индекс1, тип2 индекс2, …,
типN индексN]

```
{
  get  {код аксессора для получения данных}
  set {код аксессора для установки данных}
 }
```

Например, если в классе объявлен массив

int[,] x;

то простейший индексатор, обеспечивающий доступ к массиву x может выглядеть так:

```
public int this[int i, int j]
  { get {return x[i ,j];}
    set  {x[i, j] = value;}
   }
```

Операции класса.

Стандартные операции определены для определенных типов данных. Язык C# позволяет определить значение операции для объектов созданного программистом класса. Такое создание новой операции со стандартным именем называется перегрузкой операций.

Например, стандартная операция сложения определена для типов int, uint, long, ulong, float, double, decimal. Можно создать еще одну операцию сложения, например, для объектов созданного класса Gruppa, в результате выполнения которой две группы сливаются в одну.

Например,

Gruppa g1, g2;

Использование такой операции будет выглядеть так: g1+g2

Перегружать операции можно только для **создаваемых** классов, для *стандартных* типов этого делать **нельзя**.

Унарные операции.

В классе можно определять следующие операции:

+ - ! ~ ++ -- true false

public static тип_результата operator знак_операции

(тип_операнда операнд)

{тело_операции}

Тип операнда должен совпадать с классом, для которого определена операция.

```
class Tochka
    {
        double x, y;
        public Tochka(double xx, double yy) { x = xx; y = yy; }

        public double X
                { get { return x; } }

        public double Y
                { get { return y; } }
```

```
public static Tochka operator  - (Tochka tchk)
        { Tochka rez = new Tochka(0,0);
         rez.x = -tchk.x;     rez.y = -tchk.y;
         return rez;
        }

 public static Tochka operator   -- (Tochka tchk)
         {
            tchk.x = tchk.x - 1; tchk.y = tchk.y - 1;
             return tchk;
         }
}
```

Объект, на который ссылается операнд, модифицируется.

```
class Program
  {
    static void Main(string[ ] args)
    {
      Tochka t1 = new Tochka(2, 8);

      Console.WriteLine(-t1.X + ", " + -t1.Y);
      Console.WriteLine(t1.X + ", " + t1.Y);

     --t1;
     Console.WriteLine(t1.X + ", " + t1.Y);
     Console.ReadKey();
```

Префиксный и постфиксный инкременты и декременты не различаются.

Перегруженные версии ключевых слов **true** и **false** позволяют по новому определить понятия ИСТИНА и ЛОЖЬ в отношении создаваемых классов.

Например, можно определить, что для объектов класса Gruppa истина, если в группе больше 5 человек, т. е объект этого класса, в котором поле, содержащее количество студентов, меньше 6, является ложным (иначе, имеет значение false).

Операции **true** и **false** должны быть определены в паре. Нельзя перегружать только одну из них.

При этом тип результата должен быть **bool**.

Формат этих операций:

public static bool operator true (тип_операнда операнд)

{

тело операции с возвратом значения true *или* false.

}

public static bool operator false (тип_операнда операнд)

{

тело операции с возвратом значения true *или* false.

}

Например, определим класс Treugolnic, объект которого считается истинным, если он существует.

```
class Treugolnic
    {
       double a, b, c;
    public Treugolnic(double a1, double b1, double c1)
          { a = a1; b = b1; c = c1; }
public static bool operator true(Treugolnic p)
       {
          if (p.a + p.b > p.c && p.b + p.c > p.a && p.a + p.c > p.b)
                return true;
          else return false;
       }
```

```
public static bool operator false(Treugolnic p)
        {
            if (p.a + p.b > p.c && p.b + p.c > p.a && p.a + p.c > p.b)
                return false;
            else return true;
        }
    }
```

```
static void Main(string[] args)
    {
        Console.WriteLine("Введите стороны треугольника");
        double x = Double.Parse(Console.ReadLine());
        double y = Double.Parse(Console.ReadLine());
        double z = Double.Parse(Console.ReadLine());
       Treugolnic T1 = new Treugolnic(x, y, z);

    if (T1) Console.WriteLine(" Треугольник существует");
    else Console.WriteLine(" Треугольник не существует! ");

    Console.ReadKey();
            }
```

Перегруженная операция **!** также, как правило, возвращает результат типа bool.

Дополним класс Treugolnic операцией **!**

```
public static bool operator !(Treugolnic p)
        {
            if (p) return false;
            else return true;
        }
```

Тогда в методе Main ее можно использовать след. образом:

```
bool t;
        do
        {
            Console.WriteLine("Введите стороны треугольника");
                double x = Double.Parse(Console.ReadLine());
                double y = Double.Parse(Console.ReadLine());
                double z = Double.Parse(Console.ReadLine());

                Treugolnic T1 = new Treugolnic(x, y, z);

if  (T1)  Console.WriteLine(" Треугольник существует");
 else Console.WriteLine(
                    " Треугольник не существует! Повторите ввод.");
```

```
    t = !T1;

}
 while (t);
```

Бинарные операции.

В классе можно определять следующие бинарные операции:

+	-	*	/	%
&	|	^	<<	
==	!=	>		

Если значения первого операнда достаточно, чтобы определить результат операции, второй операнд не вычисляется

При соблюдении определенных правил можно использовать операторы **&&** и **||** , действующие по сокращенной схеме.

public static тип_рез operator знак_операции

(тип_операнда1 операнд1, тип_операнда2 операнд2)

{тело_операции}

Тип хотя бы одного из операндов должен совпадать с классом, для которого определена операция.

Операции присваивания (включая составные, например "+=") перегружать **нельзя**.

Операции, перегрузка которых также запрещена:

[ ] ( ) new is sizeof typeof ?

Пример. Дополним класс Massiv из примера 1 операцией сложения массивов и операцией сложения массива с числом.

```
class Massiv
    {
        double[ ] a;

        public Massiv(int n)
                {
                    a = new double[n];
                }
```

```
public double this[int i]
 {
    get
              {
                 if (i >= 0 && i < a.Length) return a[i];
                 else throw new Exception("Плохой индекс " + i);
              }
  set
          {
             if (i >= 0 && i < a.Length) a[i] = value;
             else throw new Exception("Плохой индекс " + i);
          }
 }
```

```
public int Length
        { get { return a.Length; } }

public void vyvod(string zagolovok)
        {
            Console.WriteLine(zagolovok);
            foreach (double x in a)
                Console.WriteLine(x);
        }
```

```
public void vvod(string name)
    {
        Console.WriteLine("Введите элементы массива " + name);
        for (int i = 0; i < a.Length; i++)
        {
            a[i] = Convert.ToDouble(Console.ReadLine());
        }
    }
```

```
public static Massiv operator     +(Massiv a1, Massiv a2)
   {    int len1, len2, k;
   if (a1.Length > a2.Length)
                    { len1 = a1.Length; len2 = a2.Length; k = 1; }
    else { len1 = a2.Length; len2 = a1.Length; k = 2; }
     Massiv rez = new Massiv(len1);
      for (int i = 0; i < len1; ++i)
      {
                if   (i < len2)   rez[i] = a1[i] + a2[i];
                 else     rez[i] = (k == 1) ?   a1[i] :   a2[i];
        }
    return rez;}
```

```
public static Massiv   operator    +(Massiv a1, int a2)
        {
            Massiv rez = new Massiv(a1.Length);
            for    (int i = 0; i < a1.Length; ++i)    rez[i] = a1[i] + a2;
             return rez;
         }
 public static Massiv operator +( int a2,    Massiv a1)
        {
            Massiv rez = new Massiv(a1.Length);
            for (int i = 0; i < a1.Length; ++i) rez[i] = a1[i] + a2;
            return rez;
         }     }
```

```
class Program
    {
        static void Main(string[] args)
        { // Ввод и вывод исходных массивов
                    …
            Massiv C = A + B; C.vyvod("A+B");
            Massiv Z = A + 2; Z.vyvod("A+2");
            Massiv Z1 = 3 + B; Z1.vyvod("3+B");

            Console.ReadKey();

        }
```

Результаты работы:

Массив А
3
3
Массив В
1
1
1
А+В
4
4
1
А+2
5
5
3+В
4
4
4

Операцию сложения массивов можно было определить так:

```
public static Massiv operator +(Massiv a1, Massiv a2)
    {
       Massiv rez = new Massiv(a1.Length+a2.Length);
       for (int i = 0; i < a1.Length; ++i)    rez[i] = a1[i];
       for (int i = 0; i < a2.Length; ++i)    rez[a1.Length+i] = a2[i];
       return rez;
     }
```

Операции отношения следует перегружать парами. Например, определяя в классе операцию "<", нужно определить операцию ">", и наоборот.

Пары операций отношения:

==	!=
<	>
<=	>=

Перегружая операции == и ! = , следует перегрузить также методы Object.Equals( ) и Object.GetHashCode( ), которые будут рассмотрены в последующих лекциях.

Обычно операции отношения возвращают логическое значение.

Например, дополним класс массив операциями < и >.

```
public static bool operator <(Massiv a1, Massiv a2)
        {
            if (a1.Length < a2.Length) return true;
            else  return false;
        }

public static bool operator <(Massiv a1, Massiv a2)
        {
            if (a1.Length < a2.Length) return true;
            else  return false;
        }
```

Использовать можно так:

```
if (A > B) Console.WriteLine("A больше B");
```

Если при определении в классе логических операций не планируется использование логических операций, работающих по сокращенной схеме, можно перегружать операторы & и | по своему усмотрению.

Эти операции должны возвращать результат типа bool.

Например, опишем класс Student, каждый объект которого характеризуется фамилией и средним баллом.

И определим в этом классе операцию & , возвращающую значение true, если оба операнда нормально учатся.

```
class Student
{    string fam;
   double sr_ball;
   public Student(string f, double sb)
      { fam = f; sr_ball = sb; }
    public string Fam
      { get { return fam; }
        set { fam = value; }
      }
    public double Sb
      { get { return sr_ball; }
        set { sr_ball = value; }}
```

```
public static bool operator &(Student s1, Student s2)
    {
        if (s1.sr_ball >= 4    &&    s2.sr_ball >= 4) return    true;
        else return false;
    }

  class Program
     {
 static void Main(string[] args)
          {
              Student st1 = new Student("Петров", 2);
              Student st2 = new Student("Васечкин", 6.2);
```

```
if (st1 & st2) Console.WriteLine("Оба нормальные студенты");

else

    Console.WriteLine("Один из студентов может скоро уже  не быть студентом,\n а может быть и  оба...");

 st1.Sb = 4;

 Console.WriteLine(st1.Fam +" и "+st2.Fam+

                                  " исправились?  "+(st1 & st2));

 Console.ReadKey();

        }

    }
```

Аналогично можно определить операцию |

Чтобы иметь возможность использовать операции && и || , необходимо соблюдать следующие правила.

- В классе должны быть определены операции **true и false**.

- Класс должен перегружать операции & и |, которые должны возвращать **объект этого класса**.

- Каждый параметр должен представлять собой ссылку на объект класса, в котором определены операции.

Выполним все эти требования в классе Student.

```
public static bool operator true (Student s1)
    {
        if (s1.sr_ball >= 4) return true;
        else return false;
    }

  public static bool operator false(Student s1)
      {
          if (s1.sr_ball < 4) return true;
          else return false;
      }
```

```
public static Student operator  &(Student s1, Student s2)
    {
        if (s1.sr_ball >=4 && s2.sr_ball >=4) return new Student("",10);
        else return new Student("", 0); ;
    }
```

Тогда в вызывающем методе можно использовать эти операции так:

```
Student st1 = new Student("Петров", 8);
Student st2 = new Student("Васечкин", 6.2);
if (st1 & st2) Console.WriteLine("Оба нормальные студенты");
        else
Console.WriteLine("Один из студентов может скоро уже  не быть студентом,\n а может быть и  оба...");
    st1.Sb = 2;
Console.WriteLine(st1.Fam +" и "+st2.Fam+" оба хорошо учатся?  ");
```

```
if (st1   &&   st2) Console.WriteLine( " Да!!!  ");
        else Console.WriteLine(" Нет!!!  ");
```

Первый операнд st1 проверяется с помощью операции **operator false.** Если этот тест в состоянии определить результат всего выражения (т. е. st1 ложный), то оставшаяся &-операция уже не выполняется. В противном случае для определения результата используется соответствующая перегруженная операция "&".

Что будет, если логически неверно определить операции отношения или false ?

```
class VVV
    {
       public double x;

public static bool operator    <(VVV    a, VVV    b)
       {
          if (a.x  <  b.x)     return    true;
          else    return    false;
       }
```

```
public static bool operator    >(VVV    a, VVV     b)
        {
           if (a.x > 2*b.x) return true;
           else return false;
        }

 static public bool operator    true(VVV    a)
         {
            if (a.x != 0) return true;
            else return false;
         }
```

```
static public bool operator false(VVV    a)
        {
            //  if (a.x == 0) return   true;
            //  else
            return false;
        }
```

```
static void Main(string[ ] args)
    {
      VVV    aa = new   VVV( ); VVV    bb = new VVV( );
       aa.x = 3;    bb.x = 2;
     Console.WriteLine("aa<bb="+(aa<bb)+"   aa>bb="+(aa>bb));

        aa.x=Double.Parse(Console.ReadLine());
        if (aa) Console.WriteLine("aa не ноль");
          else  Console.WriteLine("aa  ноль");

      Console.ReadKey();
    }
```

В этом случае нелогичность в определении false не повлияет на результат выполнения.

Операции преобразования типа.

Эти операции преобразуют объект некоторого класса в значение другого типа. Фактически, операция преобразования перегружает операцию приведения типов.

Существуют две формы операторов преобразования: явная и неявная.

Явная форма:

public static explicit operator тип_результата

(исходный_тип параметр)

{return значение;}

Операция выполняет преобразование из типа параметра в тип результата.

Одним из этих типов должен быть класс, для которого определяется операция.

Нельзя определять преобразование к типу object и наоборот.

Явное преобразование выполняется при использовании операции приведения типа.

Например, определим в классе Student операцию преобразования студента (т.е. объекта класса) к строковому типу.

```
public static explicit operator string(Student  s)
    { return s.fam; }
```

Применение :

```
string f = (string) st1; Console.WriteLine( f );
Console.WriteLine((string) st2);
```

Определим также в классе Student операцию преобразования вещественного числа в объекта класса с соответствующим средним баллом.

```
public static    explicit operator    Student(double sb)
    { return     new      Student("", sb); }
```

Применение :

```
Student st3 = (Student)    7.5;      st3.Fam = "Иванов";
```

Неявная форма:

public static implicit operator тип_результата

(исходный_тип параметр)

{return значение;}

Преобразование выполняется автоматически в следующих случаях:

- при присваивании объекта переменной, тип которой совпадает с типом результата;

- при использовании объекта в выражении , содержащем переменные, тип которых совпадает с типом результата;

- при передаче объекта в метод на место параметра с типом результата;

- при явном приведении типа.

Для одной и той же пары типов, участвующих в преобразовании, **нельзя** определить одновременно обе формы операции преобразования.

Например, дополним класс Student операцией преобразования строки в студента в неявной форме.

```
public static implicit operator Student(string ff)
  { return new Student(ff, 0); }
```

```
Student st4 = "Коньков";
Console.WriteLine("Средний балл студента с фамилией " +
                                        st4.Fam + "  :" + st4.Sb);
```

Методы с переменным количеством аргументов.

Чтобы метод мог принимать произвольное число аргументов, нужно в списке параметров использовать параметр-массив неопределенной длины, помеченный ключевым словом **params**.

Этот параметр может быть только **один** и должен размещаться в списке **последним**.

Например,

```
public int min(int x, int y, params int[ ] z) {  }
```

В этот метод могут быть переданы два и более аргументов.

Параметр с ключевым словом **params** может принять любое количество аргументов, **даже нулевое**.

Например, определим в классе Program метод, вычисляющий сумму нескольких чисел, первое из которых целое, а остальные вещественные.

```
class Program
    {public static double sum(int x, params double[] y)
        { double s = x;
            foreach (double yy in y) s = s + yy;
             return s;
        }
```

```
static void Main(string[] args)
        {
            Console.WriteLine("3+5=" + sum(3, 5));
            Console.WriteLine("3+5+2.5=" + sum(3, 5,2.5));
            double[] z= {2,4,6};
            Console.WriteLine("3+ сумма элементов z=" + sum(3,z));
            Console.ReadKey();
        }
```

В методе список параметров может состоять только из params-параметра. В этом случае нужно предусматривать возможность вызова метода без параметров, чтобы не возникали ошибки.

Например, определим метод для нахождения максимального из заданных вещественных чисел.

```
public static double    max(params double[ ]   y)
    {
       if (y.Length = = 0)
            { Console.WriteLine("Нет аргументов "); return 0; }
       else
       {
          double s = y[0];
          foreach (double    yy   in    y) if (yy  >=  s)   s = yy;
          return s;
       }
       }
```

```
static void Main(string[] args)
        {
            Console.WriteLine("max(2,8)= " + max(2, 8));
            Console.WriteLine(" max(3,5,2.5)= " + max(3, 5, 2.5));
            double[ ] z= {2,4,6,10};
            Console.WriteLine("максимум в z= " + max(z));
            Console.WriteLine("max( )= " + max( ));
            Console.ReadKey();
        }
```

Перегрузка методов.

Определение нескольких методов с одним и тем же именем, но различными параметрами называется **перегрузкой методов**, а сами методы— ***перегруженными*** (overloaded).

Все перегруженные методы должны иметь списки параметров, которые отличаются по типу и / или количеству, а также по способу передачи.

Перегруженные методы **могут** отличаться и типами возвращаемых значений.

Компилятор определяет, какой метод требуется вызвать по типу и количеству фактических параметров, т.е. осуществляет **разрешение перегрузки.**

При разрешении не учитываются тип возвращаемого значения и параметр с модификатором params.

При вызове перегруженного метода компилятор выбирает вариант метода, соответствующий типу и количеству передаваемых в метод аргументов.

Если **точное соответствие не найдено**, выполняется неявное преобразование типов в соответствии с общими правилами. Если преобразование невозможно, возникает ошибка.

Если существует несколько подходящих вариантов метода, выбирается лучший из них: содержащий меньшее количество и длину преобразований.

Если выбрать лучший не удается, выдается сообщение об ошибке.

Например:

```
class Student
    { int[] ocenki ;
     string fam;
      public double Sr_b
            { get {
                double s = 0;
                foreach (int x in ocenki) s = s + x;
                return s / ocenki.Length;
                }
            }
```

```
public void vvod()
        {Console.WriteLine("Фамилия ?");
         fam = Console.ReadLine();
        }

public void vvod(int n)
        {
           Console.WriteLine("введите "+n+" оценок");
           ocenki = new int[n];
           for (int i = 0; i < n; i++)
                ocenki[i]=Convert.ToInt32(Console.ReadLine());
        }
```

```
public void rez()
    {
        Console.WriteLine("Успеваемость студента " + fam);
        foreach(int x in ocenki) Console.Write(x+" ");
        Console.WriteLine();
    }
```

```
public void rez(int k)
        {
Console.Write("Количество оценок " +k+" студента "+ fam+ ": ");
            int m = 0;
            foreach (int x in ocenki)   if (x = = k) m = m + 1;
            Console.WriteLine(m);
        }
```

```
public void rez(int k,out int m)
        {
            Console.Write("Количество оценок ниже " + k + " студента " +
fam + ": ");
             m = 0;
            foreach (int x in ocenki) if (x <k) m = m + 1;
            Console.WriteLine(m);
        }
```

```
public void rez(Student st)
        {
            if (Sr_b>st.Sr_b)
            Console.WriteLine(fam+" учится лучше чем "+st.fam);
                else Console.WriteLine(st.fam + " учится лучше чем " +
fam);
        }
    }
```

В методе Main :

```
Student st1 = new Student();
st1.vvod( ); st1.vvod(5);
st1.rez( );
st1.rez(9);
int mm;  st1.rez(4, out mm);
if (mm > 0) Console.WriteLine("Этот студент - двоечник" );
Student st2 = new Student( ); st2.vvod( ); st2.vvod(4);
st1.rez(st2);
```

Фамилия ?
Тяпкин
введите 5 оценок
9
3
4
6
9
Успеваемость студента Тяпкин
9 3 4 6 9
Количество оценок 9 студента Тяпкин: 2
Количество оценок ниже 4 студента Тяпкин: 1
Этот студент - двоечник
Фамилия ?
Ляпкин
введите 6 оценок
2
3
5
5
4
3
Тяпкин учится лучше чем Ляпкин