Классы

Класс –

- это обобщенное понятие, определяющие характеристики и поведение некоторого множества объектов, называемых экземплярами класса.

- «Классический» класс содержит данные, определяющие свойства объектов класса, и методы, определяющие их поведение.

- Для Windows-приложений в класс добавляется– событии, на которые может реагировать объект класса.

Описание класса

[атрибуты] [спецификаторы] class имя_класса [: предки]

{тело_класса}

Простейший пример класса:

class Demo{ }

Спецификаторы определяют свойства класса, доступность класса для других элементов программы

Спецификатор	*Описание*
new	Задает новое описание класса взамен унаследованного от предка. Используется для вложения классов (в иерархии объектов)
public	Доступ к классу не ограничен
protected	Доступ только из данного или производного класса. Используется для вложенных классов
internal	Доступ только из данной программы (сборки)
protected internal	Доступ только из данного и производного класса и из данной программы (сборки)
private	Доступ только из элементов класса, внутри которых описан данный класс. Используется для вложенных классов
static	Статический класс. Позволяет обращаться к методам класса без создания экземпляра класса
sealed	Бесплодный класс. Запрещает наследование данного класса. Применяется в иерархии объектов
abstract	Абстрактный класс. Применяется в иерархии объектов

- Объекты создаются явным или неявным образом, то есть либо программистом, либо системой.

Программист создает экземпляр класса с помощью операции ***new***:

```
Demo a = new Demo ();  // Создается экземпляр класса Demo
```

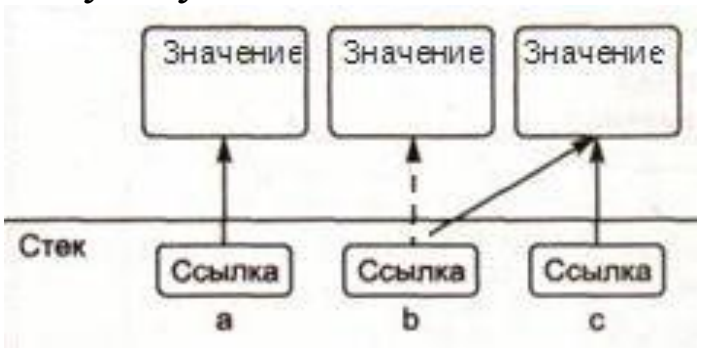
- Если достаточный для хранения объекта объем памяти выделить не удалось, то генерируется исключение ***OutOfMemoryException***.

В общем случае класс может содержать следующие функциональные элементы:

1. **Данные**: переменные или константы.
2. **Методы**, реализующие не только вычисления, но и другие действия, выполняемые классом или его экземпляром.
3. **Конструкторы** (реализуют действия по инициализации экземпляров или класса в целом).
4. **Свойства** (определяют характеристики класса в соответствии со способами их задания и получения).
5. **Деструкторы** (определяют действия, которые необходимо выполнить до того, как объект будет уничтожен).
6. **Индексаторы** (обеспечивают возможность доступа к элементам класса по их порядковому номеру).
7. **Операции** (задают действия с объектами с помощью знаков операций).
8. **События** (определяют уведомления, которые может генерировать класс).
9. **Типы** (типы данных, внутренние по отношению к классу).

Присваивание и сравнение объектов

- При присваивании значения копируется значение,
- При присваивании ссылки — ссылка,
- После присваивания одного объекта другому мы получим две ссылки, указывающие на одну и ту же область памяти:



- Созданы три объекта ***a, b и c***, выполнено присваивание ***b = c***. Ссылки ***b*** и ***c*** указывают на один и тот же объект, старое значение ***b*** становится *недоступным* и *очищается* сборщиком мусора.
- Аналогично с операцией проверки на равенство. Величины значимого типа равны, если равны их значения, величины ссылочного типа равны, если они ссылаются на одни и те же данные (объекты ***b*** и ***c*** равны, т.к. они ссылаются на одну и ту же область памяти, но ***a*** не равно ***b*** даже при равенстве их значений).

Данные: поля и константы

- При описании данных также можно указывать атрибуты и спецификаторы, задающие различные характеристики элементов.

[атрибуты] [спецификаторы] [const] тип имя [= начальное_значение]

Спецификаторы для данных

Спецификатор	*Описание*
new	Новое описание поля, скрывающее унаследованный элемент класса
public	Доступ к элементу не ограничен
protected	Доступ только из данного и производных классов
internal	Доступ только из данной сборки
protected internal	Доступ только из данного и производных классов и из данной сборки
private (по умолчанию)	Доступ только из данного класса
static	Одно поле для всех экземпляров класса
readonly	Поле доступно только для чтения (значение таких полей можно установить либо при описании, либо в конструкторе)
volatile	Поле может изменяться другим процессом или системой

Для констант можно использовать только спецификаторы 1-6.

- Поля, описанные со спецификатором static, константы существуют в единственном экземпляре для всех объектов класса, поэтому к ним обращаются не через имя экземпляра, а через имя класса.

- Обращение к полю класса выполняется с помощью операции доступа (точка). Справа от точки задается имя поля, слева — имя экземпляра для обычных полей или имя класса для статических.

// создание класса Demo и два способа обращения к его полям

```
pi=3,14
Окружность с центром в точке (0,0) и радиусом 3
Введите коэффициент= 3
 Новая окружность с центром в точке (-3,3) и радиусом 9
Для продолжения нажмите любую клавишу . . .
```

```
class Circle    {
        public int x=0;
        public int y=0;
        public int radius=3;
        public const double pi = 3.14;
        public static string name = "Окружность";
        double p;   double s;
}
    class Program    {
        static void Main()       {
            Circle cr = new Circle();          //создание экземпляра класса
            Console.WriteLine("pi=" + Circle.pi);        // обращение к константе
            Console.Write(Circle.name); // обращение к статическому полю
            //обращение к обычным полям
            Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", cr.x, cr.y, cr.radius);
            // Console.WriteLine(cr.p); - вызовет ошибку, т.к. поле p, с имеют тип private

            Console.Write("Введите коэффициент= ");
            int kof = int.Parse(Console.ReadLine());
            cr.x -= kof; cr.y += kof; cr.radius *= kof;
            Console.WriteLine(" Новая окружность с центром в точке ({0},{1}) и радиусом {2}",
                                    cr.x, cr.y, cr.radius);
           //cr.s = 2 * Circle.pi * cr.radius; - вызовет ошибку, т.к. поле s, с имеют тип private
        }
    }
```

Методы

- Методы находятся в памяти в единственном экземпляре и используются всеми объектами одного класса совместно, поэтому необходимо обеспечить работу методов нестатических экземпляров с полями именно того объекта, для которого они были вызваны.

- Для этого в любой нестатический метод автоматически передается скрытый параметр ***this***, в котором хранится ссылка на вызвавший функцию экземпляр.

- В явном виде параметр ***this*** применяется для того, чтобы возвратить из метода ссылку на вызвавший объект, а также для идентификации поля в случае, если его имя совпадает с именем параметра метода

```
class Circle  {
        public int x=0;
        public int y=0;
        public int radius=3;
        public const double pi = 3.14;
        public static string name = "Окружность";
        public Circle T()       //метод возвращает ссылку на экземпляр класса
{       return this;    }
public void Set(int x, int y, int r){
this.x = x;
this.y = y;
radius=r;     }
}

class Program{
static void Main(){
Circle cr = new Circle();           //создание экземпляра класса
Console.WriteLine("pi=" + Circle.pi);         // обращение к константе
Console.Write(Circle.name);   // обращение к статическому полю
//обращение к обычным полям
Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", cr.x, cr.y, cr.radius);
cr.Set(1, 1, 10);
Console.WriteLine("Новая окружность с центром в точке ({0},{1}) и радиусом {2}",
cr.x, cr.y, cr.radius);
Circle b=cr.T();       //получаем ссылку на объект cr, аналог b=c
Console.WriteLine("Новая ссылка на окружность с центром в точке ({0},{1})и радиусом {2}", b.x,
        b.y, b.radius);
}       }     }
```

```
pi=3,14
Окружность с центром в точке (0,0) и радиусом 3
Новая окружность с центром в точке (1,1) и радиусом 10
Новая ссылка на окружность с центром в точке (1,1)и радиусом 10
Для продолжения нажмите любую клавишу . . .
```

Конструкторы

- предназначен для инициализации объекта.

Конструкторы делятся на:

- ***конструкторы класса*** (для статических классов)
- ***конструкторы экземпляра класса*** (всех остальных классов).

Конструкторы экземпляра

- вызывается автоматически при создании объекта класса с помощью операции ***new***.

- Имя конструктора совпадает с именем класса.

Основные свойства конструкторов:

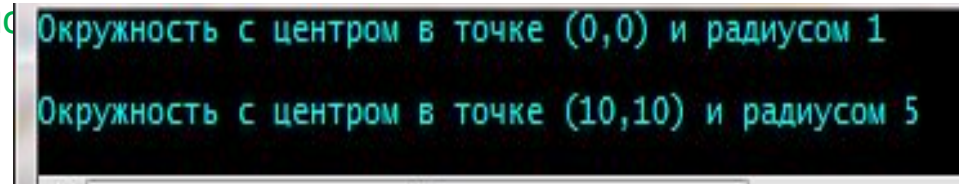
1. Конструктор не возвращает значение, даже типа void.

2. Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации.

3. Если программист не указал ни одного конструктора или какие-то поля не были инициализированы, полям значимых типов присваивается нуль, полям ссылочных типов — значение null.

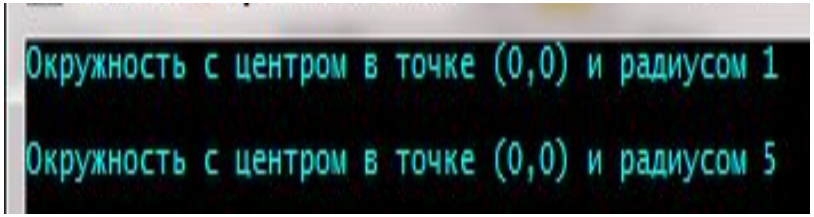
```
// Если при создании объектов требуется присваивать полю разные значения, это следует
      делать с помощью явного задания конструктора
class Circle  {
public int x;
public int y;
public int radius;
public const double pi = 3.14;
public static string name = "Окружность";
public Circle(int x, int y, int r)  //конструктор
{
this.x = x;
this.y = y;
radius = r;
}
public void Print(){
Console.Write(name);
Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", x, y, radius);
Console.WriteLine();
}     }

class Program{
static void Main(){
Circle a = new Circle(0, 0, 1);          //вызов конструктора
a.Print();
Circle b=new Circle(10, 10, 5);          //вызов конс
b.Print();
}     }    }
```

добавлен конструктор и метод Print для вывода информации об объекте



Удобно задать в классе несколько конструкторов, чтобы обеспечить возможность инициализации объектов разными способами (конструкторы должны иметь разные сигнатуры)

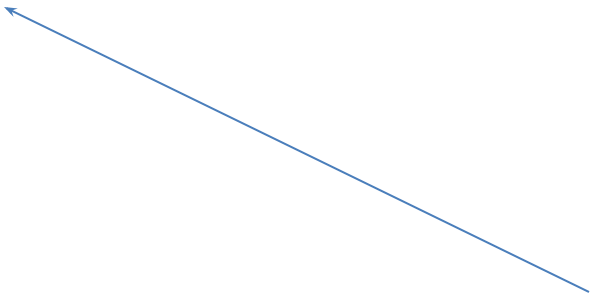
```
class Circle    {
      public int x;       public int y;       public int radius;
      public const double pi = 3.14;
      public static string name = "Окружность";
      public Circle(int x, int y, int r)    //конструктор 1
      {
         this.x = x;
         this.y = y;
         radius = r;
      }
      public Circle(int r)          //конструктор 2
      {          radius = r;       }
      public void Print()       {
         Console.Write(name);
         Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", x, y, radius);
         Console.WriteLine();       }
   }
   class Program    {
      static void Main()       {
         Circle a = new Circle(0, 0, 1);   //вызов конструктора 1
         a.Print();
         Circle b = new Circle(5);         //вызов конструктора 2
         b.Print();       }       }
```



в конструкторе 2 не были инициализированы поля ***x, y***, поэтому им присваивается значение ***0***.

Если один из конструкторов выполняет какие-либо действия, а другой должен делать то же самое плюс еще что-нибудь, то удобно вызвать первый конструктор из второго (используется ключевое слово this)

```
public Circle(int x, int y, int r):this(r)  //конструктор 1
        {
           this.x = x;
           this.y = y;
        }

public Circle(int r)    //конструктор 2
        {
           radius = r;
         }
```



инициализатор (код, который исполняется до начала выполнения тела конструктора.)
Конструктор 1 до выполнения своего кода вызывает конструктор 2

Конструкторы класса

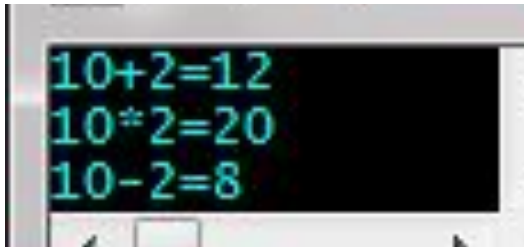
- Статические классы содержат только статические члены, в том числе и конструктор (храниться в памяти в единственном экземпляре, поэтому создавать экземпляры класса нет смысла)

- В первой версии C# для статических классов создавали два конструктора:

1. пустой закрытый (***private***) конструктор,
2. статический конструктор, не имеющий параметров

- Первый конструктор предотвращал попытки создания экземпляров класса,
- Второй конструктор автоматически вызывается системой до первого обращения к любому элементу статического класса, выполняя необходимые действия по инициализации

```
class Demo
   {
      static int a;
      static int b;
      private Demo() { }   //закрытый конструктор
      static Demo()          //статический конструктор
      {
         a = 10;
         b = 2;
      }
      public static void Print()
      {
         Console.WriteLine("{0}+{1}={2}", a, b, a + b);
         Console.WriteLine("{0}*{1}={2}", a, b, a * b);
         Console.WriteLine("{0}-{1}={2}", a, b, a - b);
      }
   }

   class Program
   {
      static void Main()
      {
         //Demo S=new Demo();    //ошибка содать экземпляр класса нельзя
         Demo.Print();
      }
   }
```



```
/*В версию 2.0 введена возможность описывать статический класс (класс с модификатором static).
      Экземпляры такого класса создавать запрещено, от него запрещено наследовать. Все элементы
      такого класса должны явным образом объявляться с модификатором static (константы и
      вложенные типы классифицируются как статические элементы автоматически). Конструктор
      экземпляра для статического класса задавать запрещается*/

static class Demo    {
       static int a=20;
       static int b=10;
       public static void Print ()
       {
          Console.WriteLine("{0}+{1}={2}",a,b,a+b);
          Console.WriteLine("{0}*{1}={2}",a,b,a*b);
          Console.WriteLine("{0}-{1}={2}",a,b,a-b);
       }
   }

   class Program    {
       static void Main()       {
          Demo.Print();
       }
   }
```

Свойства

Иногда требуется создать поле, которое с одной стороны, должно быть доступно для использования, с другой стороны, возможность что-то сделать с этим полем имеет ограничения. Например, полю нельзя присваивать произвольное значение, а только значения из какого-то диапазона. Свойство предлагает простой и удобный способ решения этой проблемы.

Синтаксис свойства:

[атрибуты] [спецификаторы] тип имя_свойства
{
[get код_доступа]
[set код_доступа]
}

- Код доступа - блоки операторов, которые выполняются при получении (get) или установке (set) свойства.

- Может отсутствовать либо часть get, либо set, но не обе одновременно.
- Если отсутствует часть set - свойство доступно только для чтения.
- Если отсутствует часть get - свойство доступно только для записи.

```
Окружность с центром в точке (0,0) и радиусом 1

центр=(1,1) радиус=10 периметр=62,83 площадь=314,16
```

```
 class Circle   {
//закрытые поля
int x;int y;      int radius;
public static string name = "Окружность";
public Circle(int x, int y, int r):this(r)         //конструктор 1
{       this.x = x;          this.y = y;       }
public Circle(int r)     //конструктор 2
{       radius = r;         }
public void Print()     {
Console.Write(name);
Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", x, y, radius);    Console.WriteLine();}
public int X     //свойство для обращения к полю x
{       get         { return x;}     set {x = value;}          }
public int Y     //свойство для обращения к полю y
{       get         {return y;}      set {y = value;}          }
public int R      //свойство для обращения к полю radius
{       get         { return radius;}        set {radius = value;}  }
public double P          //свойство только для чтения
{       get { return 2* Math.PI *radius;}      }
public double S         //свойство только для чтения
{       get { return Math.PI *radius*radius;}          }         }
class Program{
static void Main(){
Circle a = new Circle(0, 0, 1);          //вызов конструктора
a.Print();
//установка новых значений
a.X=1; a.Y=1;  a.R=10;
//a.S=100; //ошибка – свойство доступно только для чтения
Console.WriteLine("центр=({0},{1}) радиус={2} периметр={3:f2} площадь={4:f2}", a.X, a.Y, a.R, a.P, a.S);     }         }
          }
```

Деструкторы

- Специальный вид метода, вызывается сборщиком мусора непосредственно перед удалением объекта из памяти.

- В деструкторе описываются действия, гарантирующие корректность последующего удаления объекта.

[атрибуты] [extern] ~имя_класса()
{тело_деструктора}

- не имеет параметров,
- не возвращает значения
- не требует указания спецификаторов доступа.

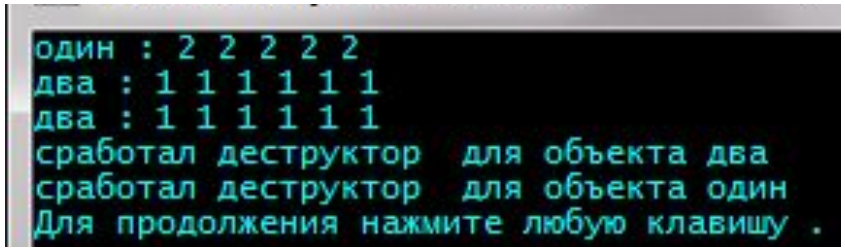
- Его имя совпадает с именем класса и предваряется тильдой (~),

- Тело деструктора представляет собой блок или просто точку с запятой.

- Если деструктор определен как внешний, то используется спецификатор extern.

```
class DemoArray{
int[] MyArray;    //закрытый массив
string name;      //закрытое поле
public DemoArray(int size,int x, string name)//конструктор
{MyArray = new int[size];    this.name = name;
for (int i=0;i<size; ++i) MyArray[i]=x;    }
public void Print ()      //метод
{Console.Write(name+ " : ");
foreach (int a in MyArray) Console.Write(a+" ");
Console.WriteLine();}
public int LengthN       //свойство
{    get { return MyArray.Length; }       }
~DemoArray()             //деструктор
{Console.WriteLine("сработал деструктор  для объекта "+this.name);     }
}
class Program    {
static void Main()       {
DemoArray a= new DemoArray(5,2, "один");
a.Print();
DemoArray b = new DemoArray(6,1, "два");
b.Print();
a = b;a.Print();    }          }
```

применение деструкторов замедляет процесс сборки мусора. Поэтому создавать деструкторы следует только тогда, когда необходимо освободить какие-то ресурсы перед удалением объекта

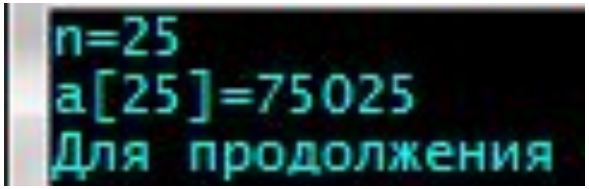


Индексаторы

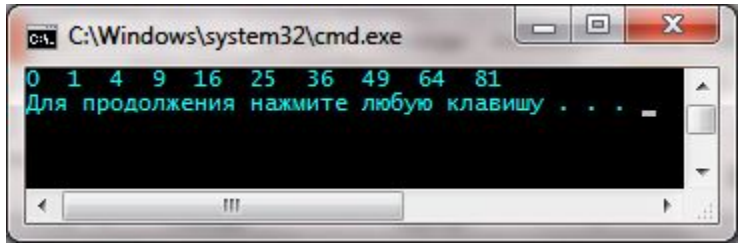
- Разновидность свойства
- Применяется для организации доступа к скрытым полям класса по индексу (к элементу массива)

  [атрибуты] [спецификаторы] тип this [список параметров]
  {
  [get код_доступа]
  [set код_доступа]
  }
- Спецификаторы аналогичны спецификаторам свойств и методов.
- Индексаторы чаще всего объявляются со спецификатором public, т.к. они входят в интерфейс объекта.
- Атрибуты и спецификаторы могут отсутствовать.
- Код доступа - блоки операторов, которые выполняются при получении (get) или установке (set) значения некоторого элемента класса.
- Может отсутствовать либо часть get, либо set, но не обе одновременно.
- Список параметров содержит одно или несколько описаний индексов, по которым выполняется доступ к элементу. Чаще всего используется один индекс целого типа.

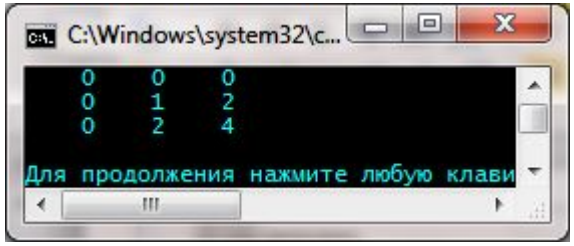
```
//индексатор, который позволяет получить n-член последовательности Фиббоначи:
 class DemoFib{
public int this[int i]        //индексатор, доступный только для чтения
{     get      {
if (i <=0) throw new Exception("недопустимое значение индекса");
else if (i==1 || i==2) return 1;
      else
{     int a=1, b=1, c;
      for (int j=3; j<=i; ++j)       {
      c=a+b;          a=b;   b=c;    }
      return b;       }
}     }
}
class Program      {
static void Main() {
Console.Write("n=");
int n=int.Parse(Console.ReadLine());
DemoFib a=new DemoFib();
try
{     Console.WriteLine("a[{0}]={1}",n,a[n]);       }
catch (Exception e)
{     Console.WriteLine(e.Message);}
}     }
```



```
/*класс-массив, значения элементов которого находятся в диапазоне [0, 100], при доступе к элементу
      проверяется, не вышел ли индекс за допустимые границы*/
   class DemoArray      {
      int[] MyArray;   //закрытый массив
      public DemoArray(int size)     //конструктор
      {            MyArray = new int[size];          }
      public int LengthArray  //свойство, возвращающее размерность
      {            get { return MyArray.Length; }          }
            public int this[int i]         //индексатор
      {            get               {
            if (i < 0 || i >= MyArray.Length) throw new Exception("выход за границы массива");
            else return MyArray[i];             }
         set               {
            if (i < 0 || i >= MyArray.Length) throw new Exception("выход за границы массива");
            else if (value >= 0 && value <= 100) MyArray[i] = value;
            else throw new Exception("присваивается недопустимое значение");                }
      }     }
   class Program          {
      static void Main()      {
         DemoArray a = new DemoArray(10);
         for (int i = 0; i < a.LengthArray; i++)
         {  a[i] = i * i;      // использование индексатора в режиме записи
            Console.Write(a[i] + "  "); // использование индексатора в режиме чтения
         }
         Console.WriteLine();
         try
         {          //a[10]=100;
         }
         catch (Exception e)
         {          Console.WriteLine(e.Message);              }
      }     }
```



```
//использование многомерных индексаторов, для работы с многомерными массивами
class DemoArray{
int[,] MyArray;       //закрытый массив
int n, m;             //закрытые поля: размерность массива
public DemoArray(int sizeN, int sizeM)        //конструктор
{     MyArray = new int[sizeN, sizeM]; this.n = sizeN;       this.m = sizeM;      }
public int LengthN //свойство, возвращающее количество строк
{     get { return n; }       }
public int LengthM //свойство, возвращающее количество строк
{     get { return m; }       }
public int this[int i, int j]  //индексатор
{     get       {       if (i < 0 || i >= n || j < 0 || j >= m) throw new Exception("выход за границы массива");
else return MyArray[i, j]; }
set   { if (i < 0 || i >= n || j < 0 || j >= m) throw new Exception("выход за границы массива");
else if (value >= 0 && value <= 100) MyArray[i, j] = value;
else throw new Exception("присваивается недопустимое значение");  }
}     }
class Program       {
static void Main()   {
DemoArray a = new DemoArray(3, 3);
for (int i = 0; i < a.LengthN; i++,Console.WriteLine())
{     for (int j = 0; j < a.LengthM; j++)   {
a[i, j] = i *j;      // использование индексатора в режиме записи
Console.Write("{0,5}", a[i, j]);     // использование индексатора в режиме чтения
}     } }       }
```



Операции класса

```
newObject x, y, z;
…
z = x+y;      // используется операция сложения, переопределенная для класса newObject
```

- Определение собственных операций класса называют перегрузкой операций.
- Операции класса описываются с помощью методов специального вида:

[атрибуты] спецификаторы объявитель_операции
{тело}

В качестве спецификаторов одновременно используются ключевые слова ***public*** и ***static***.

- Операцию можно объявить как внешнюю - ***extern***.

Правила:

1) операция должна быть описана как открытый статический метод класса (***public static***);
2) параметры в операцию должны передаваться по значению (недопустимо использовать параметры ***ref*** и ***out***);
3) сигнатуры всех операций класса должны различаться;
4) типы, используемые в операции, должны иметь не меньшие права доступа, чем сама операция (то есть должны быть доступны при использовании операции).

Унарные операции

- В классе можно переопределять следующие унарные операции: + - ! ~ ++ --, константы ***true*** и ***false*** (если была перегружена константа ***true***, то должна быть перегружена и константа ***false***, и наоборот).

тип operator унарная_операция (параметр)

Примеры заголовков унарных операций:

public static int operator + (DemoArray m)
public static DemoArray operator --(DemoArray m)
public static bool operator true (DemoArray m)

Параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. Операция должна возвращать:

1) для операций +, -, !, ~ величину любого типа;
2) для операций ++, -- величину типа класса, для которого она определяется;
3) для операций ***true*** и ***false*** величину типа ***bool***.

- Операции не должны изменять значение передаваемого им операнда.
- Операция, возвращающая величину типа класса, для которого она определяется, должна создать новый объект этого класса, выполнить с ним необходимые действия и передать его в качестве результата.

- Создать класс, реализующий одномерный массив, в котором содержатся следующие функциональные элементы:

1) конструктор, позволяющий создать объект-массив заданной размерности;

2) конструктор, позволяющий инициализировать объект-массив обычным массивом;

3) свойство, возвращающее размерность массива;

4) индексатор, позволяющий просматривать и устанавливать значение по индексу в закрытом поле-массиве;

5) метод вывода закрытого поля-массива;

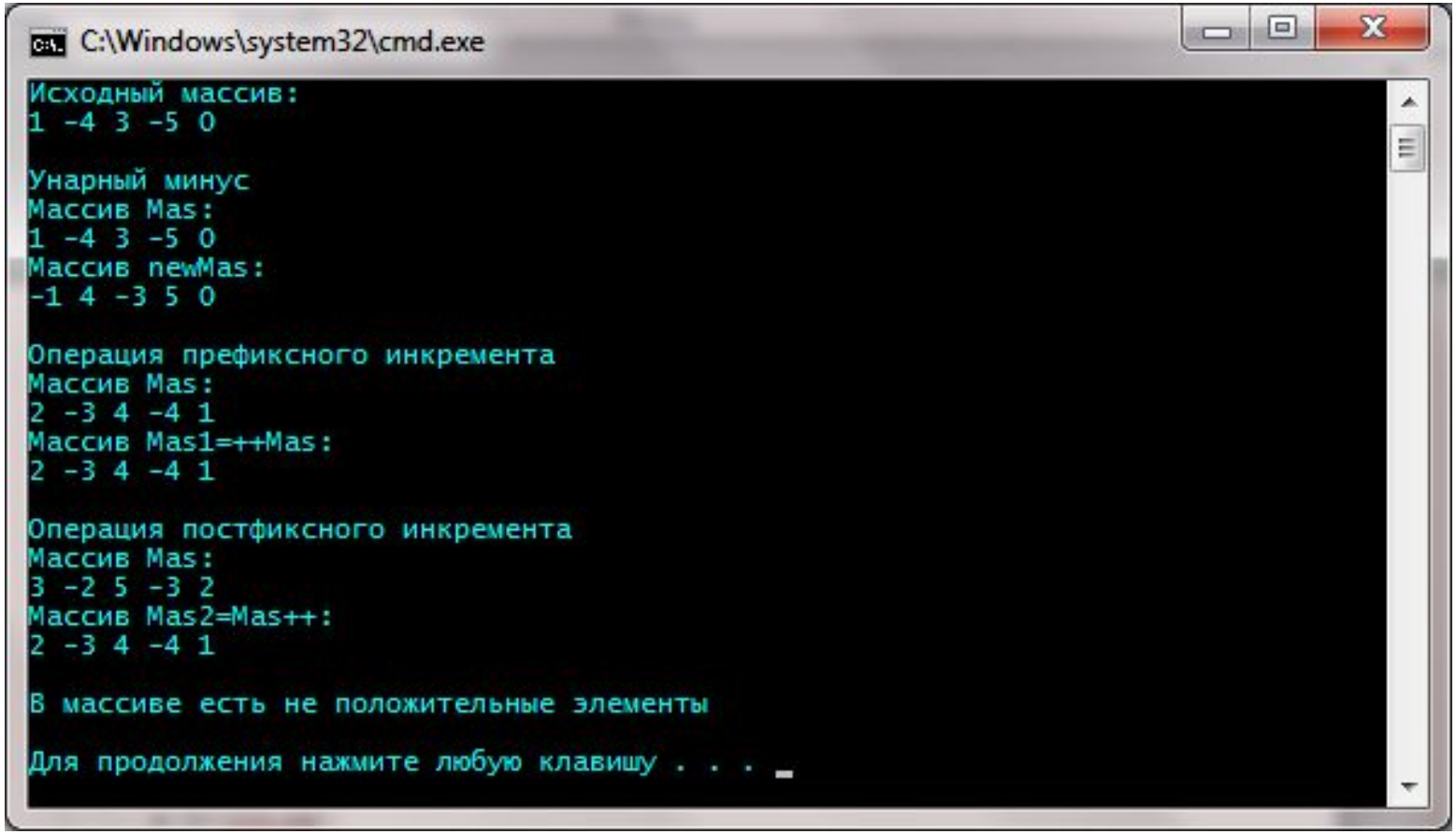
6) перегрузка операции унарный минус (все элементы массива меняют свое значение на противоположное);

7) перегрузка операции инкремента (все элементы массива увеличивают свое значение на 1);

8) перегруза констант true и false (при обращении к объекту будет возвращаться значение true, если все элементы массива положительные, в противном случае, будет возвращаться значение false).

```
class DemoArray   {
   int[] MyArray;   //закрытый массив
   public DemoArray(int size)     //конструктор 1
   {            MyArray = new int[size];           }
   public DemoArray(params int[] arr)   //конструктор 2
   {
      MyArray = new int[arr.Length];
      for (int i = 0; i < MyArray.Length; i++) MyArray[i] = arr[i];       }
   public int LengthArray  //свойство, возвращающее размерность
   {            get { return MyArray.Length; }             }
   public int this[int i]       //индексатор
   {          get                 {
         if (i < 0 || i >= MyArray.Length) throw new Exception("выход за границы массива");
         return MyArray[i];          }
      set               {
         if (i < 0 || i >= MyArray.Length) throw new Exception("выход за границы массива");
         else MyArray[i] = value;            }       }
   public static DemoArray operator -(DemoArray x)  //перегрузка операции унарный минус
   {   DemoArray temp = new DemoArray(x.LengthArray);
      for (int i = 0; i < x.LengthArray; ++i)
         temp[i] = -x[i];
      return temp;        }
   public static DemoArray operator ++(DemoArray x)          //перегрузка операции инкремента
   {  DemoArray temp = new DemoArray(x.LengthArray);
      for (int i = 0; i < x.LengthArray; ++i)
         temp[i] = x[i] + 1;  return temp;        }
   public static bool operator true(DemoArray a)       //перегрузка константы true
   {          foreach (int i in a.MyArray)            {
         if (i < 0)                  {                    return false;                  }              }
      return true;          }
```

```
        public static bool operator false(DemoArray a)      //перегрузка константы false
    {           foreach (int i in a.MyArray)
      {                 if (i > 0) {                    return true;            }           }
      return false;             }
    public void Print(string name)   //метод – выводит поле-массив на экран
    {
      Console.WriteLine(name + ": ");
      for (int i = 0; i < MyArray.Length; i++)          Console.Write(MyArray[i] + " ");
    Console.WriteLine();            }   }
class Program   {
    static void Main()      {
      try         {
        DemoArray Mas = new DemoArray(1, -4, 3, -5, 0);  //вызов конструктора 2
        Mas.Print("Исходный массив");
        Console.WriteLine("\nУнарный минус");
        DemoArray newMas = -Mas;              //применение операции унарного минуса
        Mas.Print("Массив Mas");       //создается новый объект и знаки меняются
        newMas.Print("Массив newMas");     //только у нового массива
        Console.WriteLine("\nОперация префиксного инкремента");
        DemoArray Mas1 = ++Mas;       Mas.Print("Массив Mas");    Mas1.Print("Массив
    Mas1=++Mas");
        Console.WriteLine("\nОперация постфиксного инкремента");
        DemoArray Mas2 = Mas++;       Mas.Print("Массив Mas");          Mas2.Print("Массив
    Mas2=Mas++");
        if (Mas)              Console.WriteLine("\nВ массиве все элементы положительные\n");
        else Console.WriteLine("\nВ массиве есть не положительные элементы\n");          }
      catch (Exception e)
      {
        Console.WriteLine(e.Message);          }          }   }
```


C:\Windows\system32\cmd.exe
Исходный массив:
1 -4 3 -5 0
Унарный минус
Массив Mas:
1 -4 3 -5 0
Массив newMas:
-1 4 -3 5 0
Операция префиксного инкремента
Массив Mas:
2 -3 4 -4 1
Массив Mas1=++Mas:
2 -3 4 -4 1
Операция постфиксного инкремента
Массив Mas:
3 -2 5 -3 2
Массив Mas2=Mas++:
2 -3 4 -4 1
В массиве есть не положительные элементы
Для продолжения нажмите любую клавишу . . .

Бинарные операции

При разработке класса можно перегрузить следующие бинарные операции:

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

(операций присваивания нет!)

Синтаксис объявителя бинарной операции:

тип operator бинарная_операция (параметр1, параметр 2)

Примеры заголовков бинарных операций:

public static DemoArray operator + (DemoArray a, DemoArray b)
public static bool operator == (DemoArray a, DemoArray b)

При переопределении бинарных операций нужно учитывать следующие правила:

- Хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется.
- Операция может возвращать величину любого типа.
- Операции отношений определяются только парами и обычно возвращают логическое значение. (Чаще всего переопределяются операции сравнения на равенство и неравенство для того, чтобы обеспечить сравнение значения некоторых полей объектов, а не ссылок на объект).

```
class DemoArray
    { …
    public static DemoArray operator +(DemoArray x, int a) //добавляет к каждому элементу массива заданное
    число
      {     DemoArray temp = new DemoArray(x.LengthArray);
            for (int i = 0; i < x.LengthArray; ++i)
                  temp[i]=x[i]+a;
            return temp;                  }
      public static DemoArray operator +(DemoArray x, DemoArray y) //поэлементно складывает два массива
      {     if (x.LengthArray == y.LengthArray)
            {     DemoArray temp = new DemoArray(x.LengthArray);
                  for (int i = 0; i < x.LengthArray; ++i)
                        temp[i] = x[i] + y[i];
                  return temp;        }
            else throw new Exception("несоответствие размерностей");                  }       }
    class Program {
      static void Main()          {
            try{    DemoArray a = new DemoArray(1, -4, 3, -5, 0);
                  a.Print("Массива a");
                  DemoArray b=a+10;
                  b.Print("\nМассива b");
                  DemoArray c = a+b;
                  c.Print("\nМассива c");          }
            catch (Exception e)
            {     Console.WriteLine(e.Message); }
      }     }
```

Операции преобразования типов

explicit operator целевой_тип (параметр) ***//явное преобразование***
implicit operator целевой_тип (параметр) ***//неявное преобразование***

- преобразование из типа параметра в тип, указанный в заголовке операции
- одним из этих типов должен быть класс, для которого выполняется преобразование.

Неявное преобразование выполняется автоматически в следующих ситуациях:

- при присваивании объекта переменной целевого типа;
- при использовании объекта в выражении, содержащем переменные целевого типа;
- при передаче объекта в метод параметра целевого типа;
- при явном приведении типа.

Явное преобразование выполняется при использовании операции приведения типа.

При определении операции преобразования типа следует учитывать следующие особенности:

- тип возвращаемого значения (целевой_тип) включается в сигнатуру объявителя операции;
- ключевые слова ***explicit*** и ***implicit*** не включаются в сигнатуру объявителя операции.

Для одного и того класса нельзя определить одновременно и явную, и неявную версию. Однако, т.к. неявное преобразование автоматически выполнятся при явном использовании операции приведения типа, то достаточно разработать только неявную версию операции преобразования типа.

```
//одномерный массив, в нём неявная версиюя переопределения типа DemoArray в тип одномерный массив и наоборот:
class DemoArray {
    …
    public static implicit operator DemoArray (int []a) //неявное преобразование типа int [] в DemoArray
    { return new DemoArray(a); }
    public static implicit operator int [](DemoArray a) //неявное преобразование типа DemoArray в int []
    { int []temp=new int[a.LengthArray];
      for (int i = 0; i < a.LengthArray; ++i) temp[i] = a[i];
      return temp; } }
class Program {
    static void arrayPrint(string name, int[]a) //метод, который позволяет вывести на экран одномерный массив
    { Console.WriteLine(name + ": ");
      for (int i = 0; i < a.Length; i++) Console.Write(a[i] + " "); Console.WriteLine();}
    static void Main() {
      try {
    DemoArray a = new DemoArray(1, -4, 3, -5, 0);
    int []mas1=a; //неявное преобразование типа DemoArray в int []
    int []mas2=(int []) a; //явное преобразование типа DemoArray в int []
    DemoArray b1 =mas1; //неявное преобразование типа int [] в DemoArray
  DemoArray b2 =(DemoArray)mas2; //явное преобразование типа int [] в DemoArray
                    //изменение значений
        mas1[0]=0; mas2[0]=-1; b1[0]=100; b2[0]=-100;
                    //вывод на экран
    a.Print("Массива a"); arrayPrint("Массив mas1", mas1);
                    arrayPrint("Массив mas2", mas2); b1.Print("Массива b1");
b2.Print("Массива b2"); }
    catch (Exception e) { Console.WriteLine(e.Message); } } }
```