Свойства. Индексаторы.

Свойства

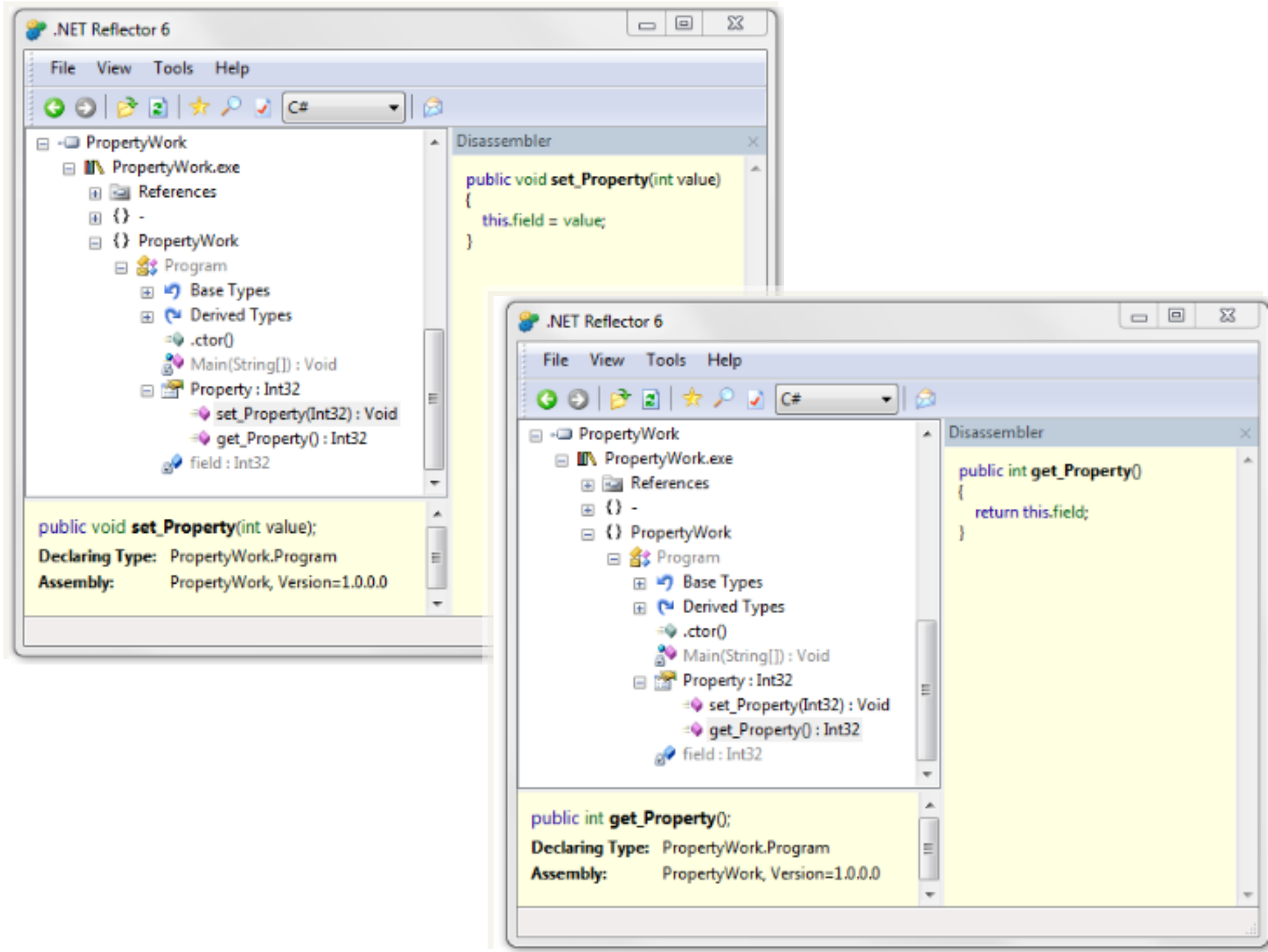
Свойство - член класса.
Свойство сочетает в себе поле с методами доступа к нему.
Свойство – «умное» поле т.е. поля с дополнительной логикой.

.NET поддерживает два вида свойств:

- без параметров - свойства,
- с параметрами – индексаторы.

```
тип имя
{
  get
   {
    // код аксессора для чтения из поля
   }
  set
   {
    // код аксессора для записи в поле
   }
}
```

тип - какой тип не может быть?


.NET Reflector 6
File View Tools Help
C#
PropertyWork
PropertyWork.exe
References
{} -
{} PropertyWork
Program
Base Types
Derived Types
.ctor()
Main(String[]) : Void
Property : Int32
set_Property(Int32) : Void
get_Property() : Int32
field : Int32
Disassembler
public void set_Property(int value)
{
this.field = value;
}
public void set_Property(int value);
Declaring Type: PropertyWork.Program
Assembly: PropertyWork, Version=1.0.0.0
.NET Reflector 6
File View Tools Help
C#
PropertyWork
PropertyWork.exe
References
{} -
{} PropertyWork
Program
Base Types
Derived Types
.ctor()
Main(String[]) : Void
Property : Int32
set_Property(Int32) : Void
get_Property() : Int32
field : Int32
Disassembler
public int get_Property()
{
return this.field;
}
public int get_Property();
Declaring Type: PropertyWork.Program
Assembly: PropertyWork, Version=1.0.0.0

```
int id;
// свойство для доступа к закрытому
// полю класса - id
public int Id
{
   get { return id;                  }
   set { if (value > 0)id = value; }
}

string name;
// свойство для доступа к закрытому
// полю класса -  name
public string Name
{   get { return name; }
    set
    {  if (value.Length!=0) name = value;
       else name = "noName";
    }
}

// Автоматическое свойство
public int Pages { get; set; }
```

с версии C# 3.0, появилась возможность для реализации очень простых свойств, не прибегая к явному определению переменной, которой управляет свойство.

Базовую переменную для свойства автоматически предоставляет компилятор. Такое свойство называется ***автоматически реализуемым.***

```
тип имя { get; set; }
```

Возможные проблемы

- Синтаксис объявления поля не может включать инициализацию (решено в C#6.0)

```
public int X { get; } = 5;
```

- Механизм сериализации на этапе выполнения сохраняет имя поля в сериализованном потоке.
- Во время отладки нельзя указать точку останова

```
// Свойство Length только для чтения,
public int Length
{  get
   {
   return len;
   }
// ВАЖНО! Должен быть хотя бы один аксессор
}

 fs.Length = 10; // Ошибка, запись запрещена!
```

```
 public int Pages { get; set; }
// ВАЖНО! Должны быть обязательно два аксессора
```

Свойствам присущ ряд существенных ограничений.

1. Свойство не определяет место для хранения данных, и поэтому не может быть передано методу в качестве параметра **ref или out**.
2. Свойство **не подлежит перегрузке**. Наличие двух разных свойств с доступом к одной и той же переменной допускается, но это, скорее, исключение, чем правило.
3. Свойство **не должно изменять состояние базовой переменной при вызове аксессора get**. Действие аксессора get не должно носить характер вмешательства в функционирование переменной.

Применение модификаторов доступа в аксессорах

По умолчанию доступность аксессоров set и get оказывается такой же, как у свойства, частью которых они являются.

```
public int MyProp
{
  get
  {
   return prop;
  }
private set
  {  // закрытый аксессор
     prop = value;
  }
}
```

```
 public int Length { get; private set; }
```

Свойство Length может быть установлено только из кода в его классе, поскольку его аксессор set объявлен как private. А изменять свойство Length за пределами его класса не разрешается.

Применение модификаторов доступа в аксессорах

На применение модификаторов доступа в аксессорах накладываются следующие ограничения.

1. действию модификатора доступа подлежит **только один** аксессор: set или get, но не оба сразу.

2. модификатор должен обеспечивать **более ограниченный доступ к аксессору**, чем доступ на уровне свойства.

3. модификатор доступа нельзя использовать при объявлении аксессора в интерфейсе.

Известно! Индексирование массива осуществляется с помощью оператора [ ].

Индексатор позволяет индексировать объект, подобно массиву.

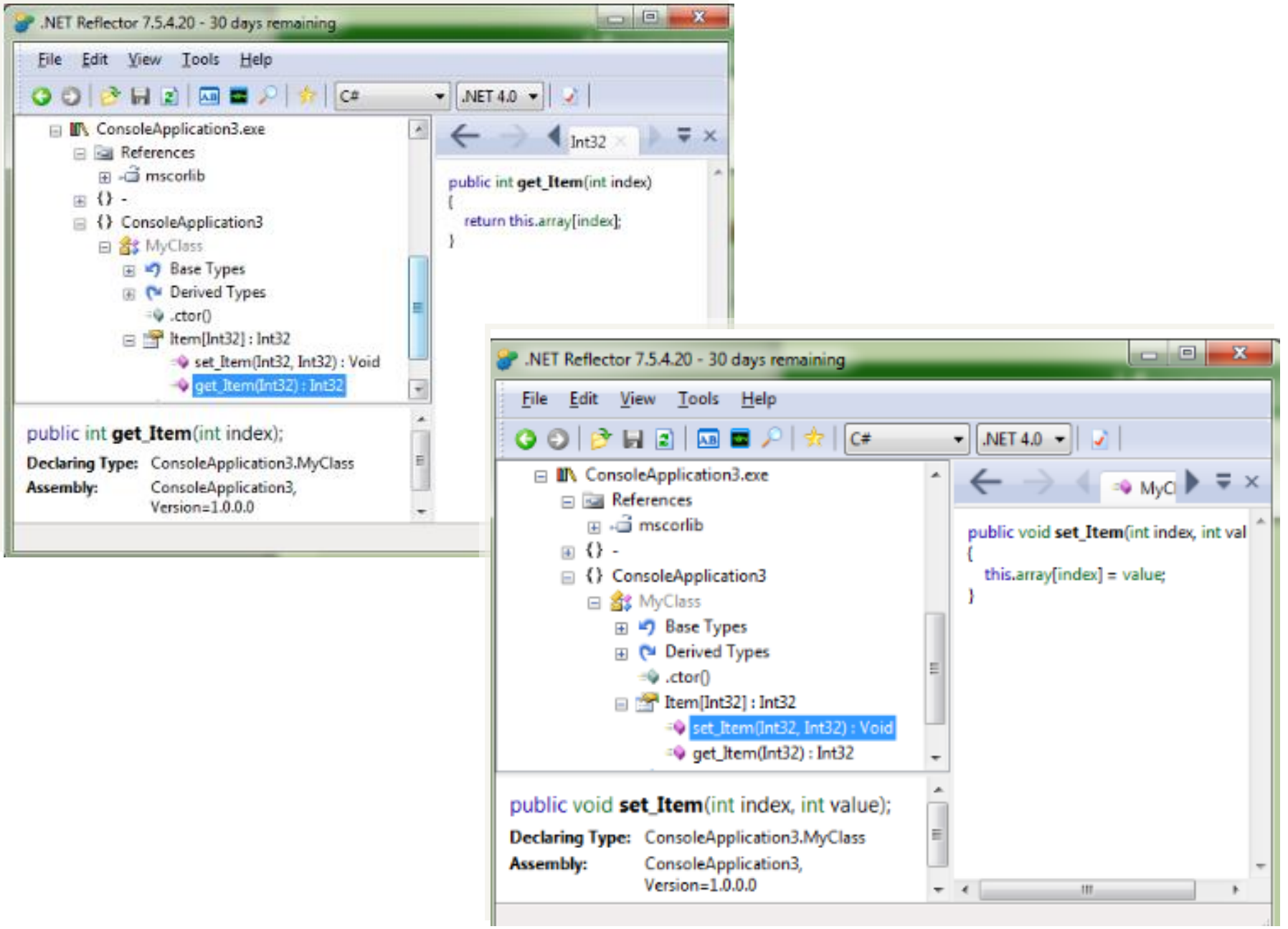
Применение: индексаторы применяются, в качестве средства, поддерживающего создание специализированных массивов, на которые накладывается одно или несколько ограничений. Индексаторы полезно использовать при создании специальных типов коллекций (обобщенных и необобщенных).

Индексаторы могут быть

- одномерными
- многомерными.

Одномерные индексаторы

```
модификатор_доступа тип_элемента this[int индекс]
{
    // Аксессор для получения данных,
    get
    {
        // Возврат значения, которое определяет индекс.
    }
    // Аксессор для установки данных,
    set
    {
        // Установка значения, которое определяет индекс.
    }
}
```


.NET Reflector 7.5.4.20 - 30 days remaining
File Edit View Tools Help
C#
.NET 4.0
ConsoleApplication3.exe
References
mscorlib
{} -
{} ConsoleApplication3
MyClass
Base Types
Derived Types
.ctor()
Item[Int32] : Int32
set_Item(Int32, Int32) : Void
get_Item(Int32) : Int32
Int32
public int get_Item(int index)
{
return this.array[index];
}
public int get_Item(int index);
Declaring Type: ConsoleApplication3.MyClass
Assembly: ConsoleApplication3, Version=1.0.0.0
.NET Reflector 7.5.4.20 - 30 days remaining
File Edit View Tools Help
C#
.NET 4.0
ConsoleApplication3.exe
References
mscorlib
{} -
{} ConsoleApplication3
MyClass
Base Types
Derived Types
.ctor()
Item[Int32] : Int32
set_Item(Int32, Int32) : Void
get_Item(Int32) : Int32
MyC
public void set_Item(int index, int val
{
this.array[index] = value;
}
public void set_Item(int index, int value);
Declaring Type: ConsoleApplication3.MyClass
Assembly: ConsoleApplication3, Version=1.0.0.0

Одномерные индексаторы

```
class MyArray
{
int [] Mass;  // объявление массива
public MyArray(int N)
   {  Mass=new int[N];  }
// создание индексатора
public int this[int index]
{ get
   { // если индекс в пределах индексации массива Mass
     // то возвращаем значение элемента массива
    if(0 <= index && index < Mass.Length)
          { return Mass[index];  }
    else  { return -1;
   }
}
  set
   {  // если индекс в пределах индексации массива Mass
      // то устанавливаем значение элемента миссива Mass
      if(0 <= index && index < Mass.Length)
        { Mass[index]=value;  }
    }
}
```

```
public static void Main(string[] args)
{
  // Создание объекта класс MyArray
  MyArray arr=new MyArray(5); // размер массива 5
  Random rnd=new Random();
  Console.WriteLine(" в этом цикле выхода за границы массива нет");
  for(int i=0; i<5;i++)
    {// работа с объектам класса через индексатор
      arr[i]=rnd.Next(0,50);
      Console.WriteLine("Значение arr[{0,2}]={1,2}",i, arr[i]);
     }

  Console.WriteLine(" в цикле имеется выход за границы массива");
  for(int i=-2; i<8;i++)
   { // работа с объектам класса
     arr[i]=rnd.Next(0,50);
     Console.WriteLine("Значение
   }
}
```

```
Индексаторы
 в этом цикле выхода за границы массива нет
Значение arr[ 0]=40
Значение arr[ 1]= 8
Значение arr[ 2]=43
Значение arr[ 3]=48
Значение arr[ 4]=28
 в этом цикле имеется выход за границы массива
Значение arr[-2]=-1
Значение arr[-1]=-1
Значение arr[ 0]=12
Значение arr[ 1]=27
Значение arr[ 2]=30
Значение arr[ 3]=22
Значение arr[ 4]=17
Значение arr[ 5]=-1
Значение arr[ 6]=-1
Значение arr[ 7]=-1
Press any key to continue . . .
```

Одномерные индексаторы

- Индексаторы не обязаны использовать в качестве индекса целочисленное значение, конкретный механизм поиска определяет разработчик.
- Индексаторы можно перегружать.
- Индексаторы могут иметь более одного формального параметра, например, при доступе к двух мерному массиву.

Перегрузка индексаторов

```
// Перегрузка индексатора
public int this[double index]
{
    get
       {  // реализация
       }
    set
       {// реализация
       }
}
```

```
Console.WriteLine(" Индексация дробными числами!");
for(double i=-2.2; i<6.7; i=i+0.3)
{
    // работа с объектам класса через индексатор
    Console.WriteLine("Значение arr[{0,3}]={1,3}",i, arr[i]);
}
```

Применение индексатора без массива

```
class MyFactorial
{
// создание индексатора (только для чтения)
 public double this[int Nmax]
 {
  get
  {   double M = 1;
      for (int i = 1; i <= Nmax; i++)
      {  M = M * i;  }
      return M;
  }
  //  set или get  - может отсутствовать!
 }
}
```

```
public static void Main(string[] args)
{
   Console.WriteLine("Пример индексатора без массива!");
   MyFactorial fact = new MyFactorial();
   Console.WriteLine("Fact[0]={0:N}", fact[0]);
   Console.WriteLine("Fact[5]={0:N}",fact[5]);
   Console.WriteLine("Fact[12]={0:N}", fact[12]);
}
```

Многомерные индексаторы

```
class Book
    {   private int id;
        private string name;
        private int pages;
        public Book(int id, string name, int pages)
        {  this.id = id; this.name = name; this.pages = pages;
        }
        // создание многомерного индексатора
        // первым параметром переменная целого типа
        // вторым параметром переменная строкового типа
        public int this [int Id, string Name]
        {  get
            {  if (id == Id && Name == name)
                {  return pages; }
                else return 0;
    }}}
```

```
static void Main(string[] args)
{  Console.WriteLine("Многомерный индексатор");
   Book book1 = new Book(1, "Война и мир", 350);
   Console.WriteLine(book1[1, "Война и мир"]);
}
```