Работа с массивами в C#

Массивы в C#

- Массивы в C# с точки зрения синтаксиса практически не отличаются от массивов в C, C++ и Java.
- Однако внутренне массив в C# устроен как тип, производный от класса System.Array.
- Формально массив определяется как набор элементов, доступ к которым производится с помощью числового индекса.

Объявление массива

- Массивы объявляются путем помещения квадратных скобок ([ ]) после указания типа данных для элементов этого массива.

```
// Массив символьных строк с 10
//элементами {0, 1,…, 9}
string[] booksOnCOM;
booksOnCOM = new string[10];
// Массив символьных строк с 2
//элементами {0, 1}
string[] booksOnPLl = new string[2];
// Массив символьных строк из 100
//элементов {О, 1,…, 99}
string[] booksOnDotNet = new
string[100];
```

Такое объявление массива приведет к ошибке компилятора:

```
// При определении массива
//фиксированного размера мы обязаны
//использовать ключевое слово new
int[4] ages = {30, 54, 4, 10};  // Ошибка!
```

Размер массива задается при его создании, но не объявлении.

```
//Будет автоматически создан массив с
//4 элементами. Обратите внимание на
//отсутствие ключевого слова new
//и на пустые квадратные скобки
int[] ages = {20, 22, 23, 0};
```

Заполнение массива в C#

- Заполнить массив можно, перечисляя элементы последовательно в фигурных скобках.
- А можно использовать для этой цели числовые индексы.

```
// Используем последовательное
//перечисление элементов массива:
string[] firstNames = new string[5] {“Steve”,
“Gina”, “Swallow”, “Baldy”, “Gunner”};

// Используем числовые индексы:
string[] firstNames = new string[5];
firstNames[0] = “Steve”;
firstNames[1] = “Glna”;
firstNames[2] = “Swallow”;
firstNames[3] = “Baldy”;
firstNames[4] = “Gunner”;
```

Важное различие между массивами C++ и C#

- В C# элементам массива автоматически присваиваются значения по умолчанию в зависимости от используемого для них типа данных.

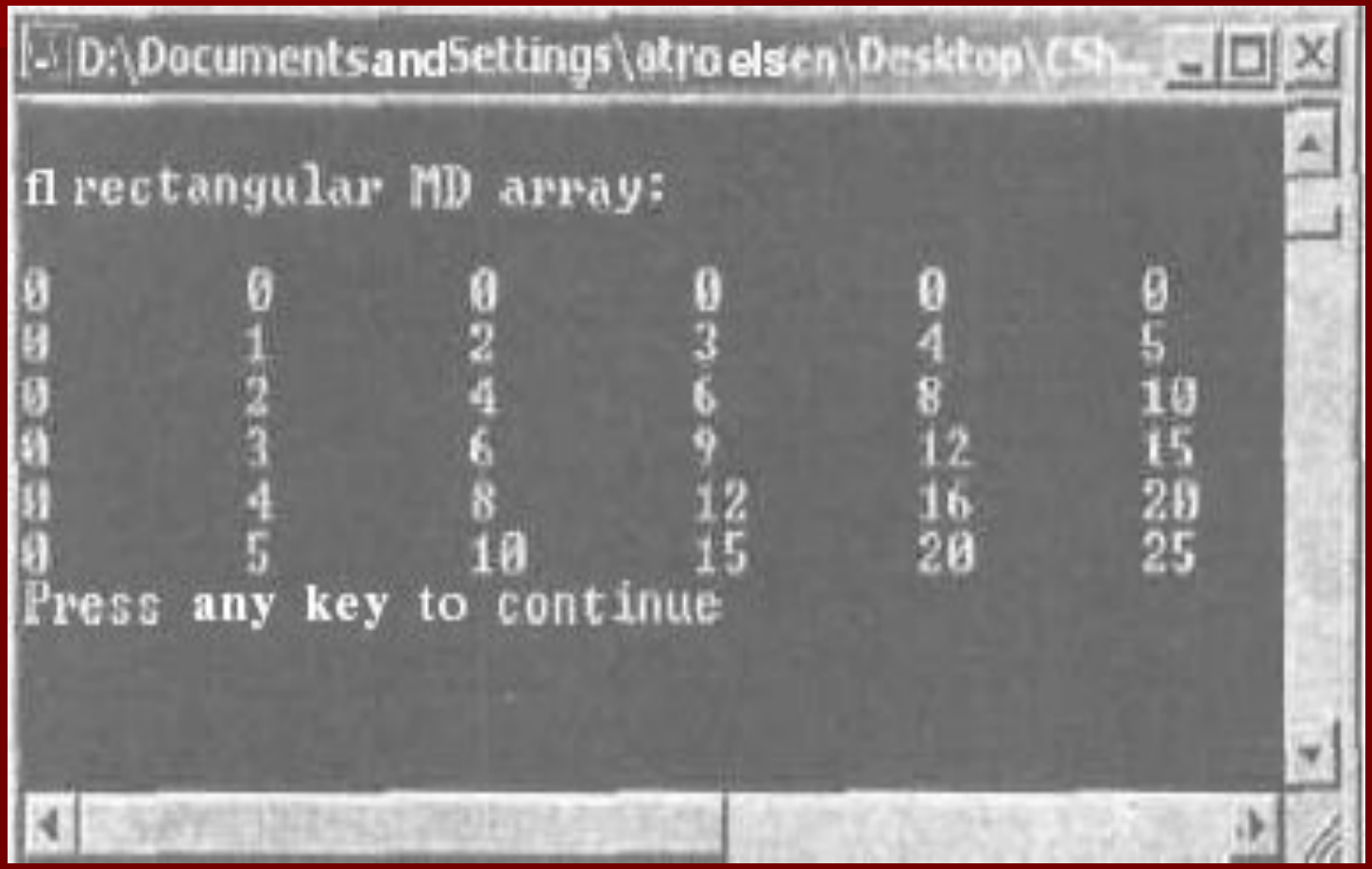
  Например, для массива целых чисел всем элементам будет изначально присвоено значение 0, для массива объектов — значение NULL и т. д.

Многомерные массивы

- Помимо массивов с одним измерением в C# поддерживаются также две основные разновидности многомерных массивов.
- Первую разновидность многомерных массивов иногда называют “прямоугольным массивом”. Такой тип массива образуется простым сложением нескольких измерений. При этом все строки и столбцы в данном массиве будут одинаковой длины.

```
// Прямоугольный многомерный массив
int[] myMatrix;
myMatrix = new int[6, 6];
// Заполняем массив 6 на 6
for (int i = 0; i < 6; i++)
 for (int j = 0; j < 6; j++)
      myMatrix[i, j] = i*j;
// Выводим элементы многомерного массива на
//системную консоль
for (int i = 0 ; i < 6; i++)
{
 for (int j = 0; j < 6; j++)
 {
      Console.Write(myMatrix[i, j] +"\t");
 }
 Console. WriteLine();
}
```

Результат работы программы

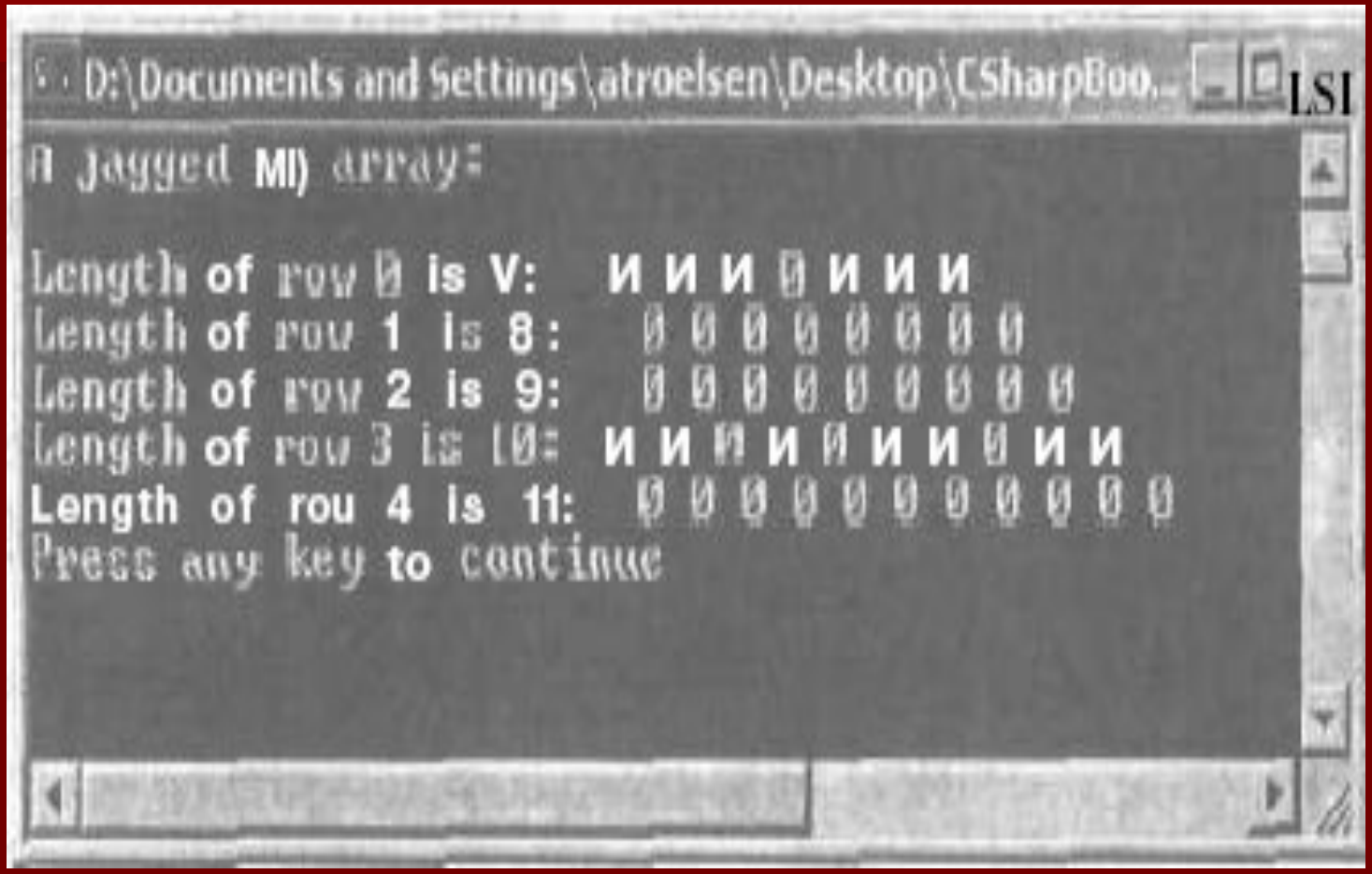


- Второй тип многомерного массива можно назвать “ломаным” (jagged). Такой массив содержит в качестве внутренних элементов некоторое количество внутренних массивов, каждый из которых может иметь свой внутренний уникальный размер.

```
// "Ломаный" многомерный массив (массив из
// массивов). В нашем случае - это массив
// из пяти внутренних массивов разного
// размера
int[][] myJagArray = new int[5][]:
// Создаем "ломаный" массив
for (int i = 0; i < myJagArray. Length; i++)
{
myJagArray[i] = new int[i + 7];
}
```

```
// Выводим каждую строку на системную консоль (как
//мы помним, каждому элементу
// присваивается значение по умолчанию - в нашем
//случае 0)
for (int i = 0; i < 5; i++)
{
  Console.Write("Length of row {0} is {1}:\t", i,
myJagArray[i].Length);
  for (int j = 0; j < myJagArray[i].Length; j++)
  {
      Console.Write(myJagArray[i][j] + " ");
  }
  Console.WriteLine();
}
```

Результат работы программы



Базовый класс System.Array

- Все наиболее важные различия между массивами в C++ и C# происходят оттого, что в C# все массивы являются производными от базового класса System. Array. За счет этого любой массив в C# наследует большое количество полезных методов и свойств, которые сильно упрощают работу программиста.

Таблица

Член класса	Назначение
BinarySearch()	Этот статический метод можно использовать только тогда, когда массив реали-зует интерфейс IComparer, в этом случае метод позволяет найти элемент массива.
Clear()	Этот статический метод позволяет очистить диапазон

	элементов (числовые элементы приобретут значения 0, а ссылки на на объекты – null)
CopyTo()	Используется для копирования элементов из исходного массива в массив назначения
GetEnumerato()	Возвращает интерфейс IEnumerator для указанного массива.

GetLengt().Lenght используется для определения количества элементов в указанном измерении массива. Length - это свойство только для чтения, с помощью которого можно получить количество элементов массива	Метод GetLength()
GetLowerBound() GetUpperBound()	Эти методы используются для определения верхней и нижней границы

	выбранного вами измерения массива.
GetValue() SetValue()	Возвращает или устанавливает значение указанного индекса для массива. Этот метод перегружен для нормальной работы как с одномерными, так и с многомерными массивами
Reverse()	Этот статический метод позволяет расставить

	элементы одномерного массива в обратном порядке.
Sort()	Сортирует одномерный массив встроенных типов данных. Если элементы массива поддерживают интерфейс IComparer, то с помощью этого метода вы сможете производить сортировку и ваших пользовательских типов данных.

Пример

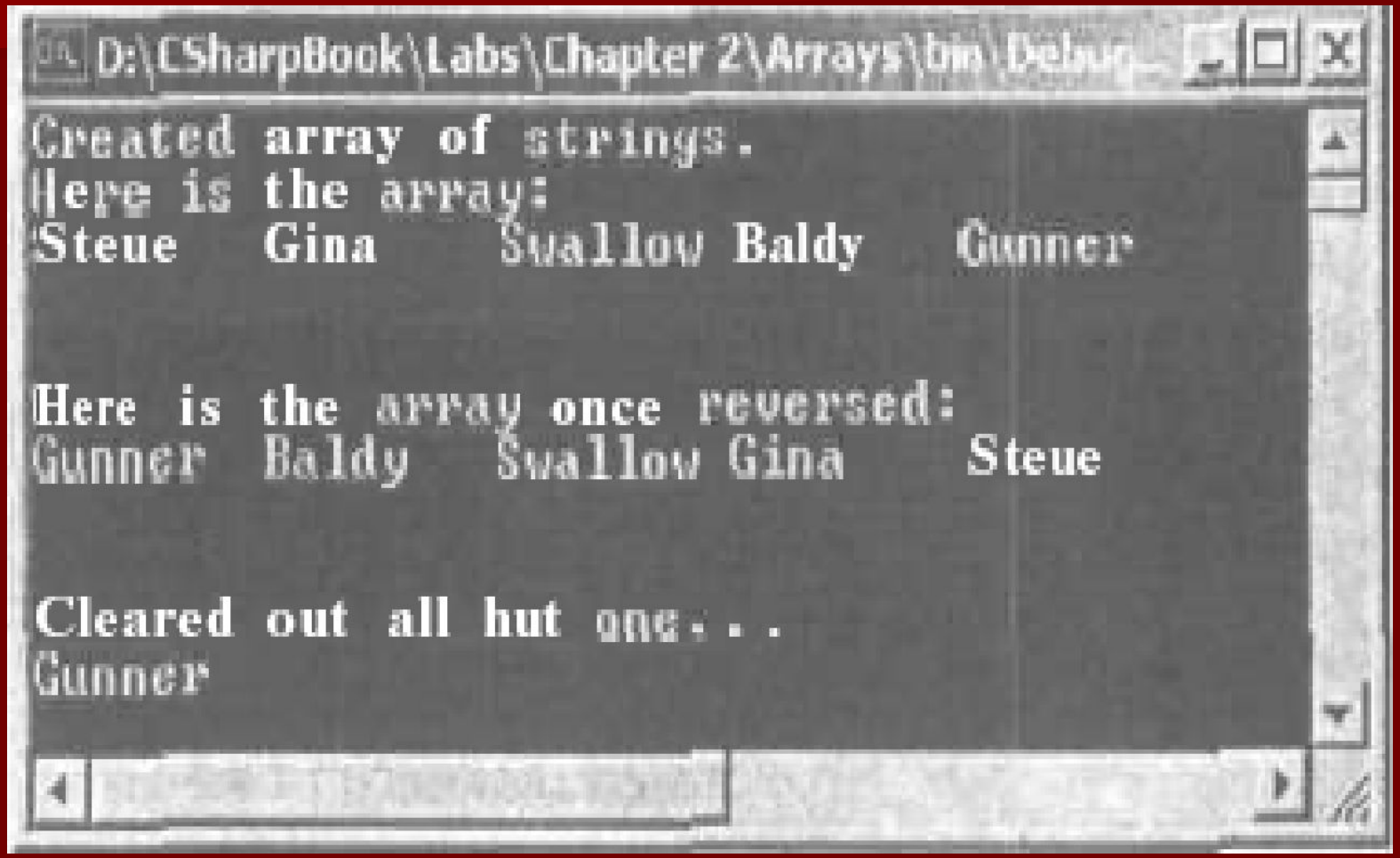
```
// Создаем несколько массивов символьных
// строк и экспериментируем с членами
// System.Array
class Arrays
{
public static int Main(string[] args)
   {
   // Массив символьных строк
   string[] firstNames - new string[5]
   {“Steve”, “Gina”, “Swallow”, “Baldy”, “Gunner”}
```

```
// Выводим имена в соответствии с порядком
// элементов в массиве
   Console. WriteLine(“Here is the array:”);

   for (int i = 0; i < firstNames. Length; i++)
    Console. Write(firstNames[i] + "\t");

// Расставляем элементы в обратном порядке
//при помощи статического
// метода Reverse()
```

```
    Array.Reverse(firstNames );
// ...и снова выводим имена
    Console. WriteLine("Here is the array once
    reversed:");
    for (int i = 0; i < firstNames.Length; i++)
     Console. Write(firstNames[i] + "\t");
// А теперь вычищаем всех, кроне юного
// Гуннара
    Console.WriteLine("Cleared out all but one...");
    Array. Clear(firstNames. 1, 4);
```

```
    for (int i = 0; i < firstNames. Length: i++)
    {
     Console. Write(firstNames[i] + "\t\n");
    }
    return 0;
    }
}
```

Результат программы



Работа со строками в C#

Строки в C#

- String (строки Unicode) — это встроенный тип данных C#.
- Все строки в мире C# и .NET происходят от единственного базового класса — System.String.
- System.String обеспечивает множество методов, которые призваны выполнить за вас всю черновую работу: возвратить количество символов в строке, найти подстроки, преобразовать все символы в строчные или прописные и т. д.

Таблица

Член класса	Назначение
Length	Это свойство возвращает длину указанной строки.
Concat()	Этот статический метод класса String возвращает новую строку, “склеенную” из двух исходных.
CompareTo()	Сравнивает две строки.

Copy()	Этот статический метод создает новую копию существующей строки
Format()	Используется для форматирования строки с использованием других примитивов (числовых данных, других строк) и подстановочных выражений вида {0}.
Insert()	Используется для вставки строки внутрь существующей.

PadLeft() PadRight()	Эти методы позволяют заполнить ("набить") строку указанными символами.
Remove() Replace()	Эти методы позволяют создать копию строки с внесенными изменениями (удаленными или замененными символами)
ToUpper() ToLower()	Эти методы используются для получения копии строки, в которой все символы станут строчными или прописными.

Обратить внимание

- Хотя *string* — это ссылочный тип данных, при использовании операторов равенства (== и !=) происходит сравнение значений строковых объектов, а не адресов этих объектов в оперативной памяти.
- Оператор сложения (+) в C# перегружен таким образом, что при применении к строковым объектам он вызывает метод Concat().

Управляющие последоватльности

- В C#, как и в C, и в C++, и в Java строки могут содержать любое количество управляющих последовательностей (escape characters).

Пример

```
// Применение управляющих
//последовательностей - \t, \\, \n и прочих
string anotherString;
anotherString = “Every programming book need
   \“Hello World\””;
Console.WriteLine("\t" + anotherString);

anotherString =
   “c:\\CSharpProjects\\Strings\\string.cs”;
Console.WriteLine(“\t” + anotherString);
```

Таблица

Управляющие посл-ти	Назначение
\	Вставить одинарную кавычку в строку
\”	Вставить двойную кавычку в строку
\\	Вставить в строку обратный слэш. Особенно полезно при работе с путями в файловой системе.

\a	Запустить системное оповещение (Alert).
\b	Вернуться на одну позицию (Backspace).
\f	Начать следующую страницу (Form feed).
\n	Вставить новую строку (New line).
\r	Вставить возврат каретки (carnage Return).
\t	Вставить горизонтальный символ табуляции (horizontal Tab).

\u	Вставить символ Unicode.
\v	Вставить вертикальный символ табуляции (Vertical tab).
\0	Представляет пустой символ (NULL)

Вывод служебных символов

- Помимо управляющих последовательностей, в C# предусмотрен также специальный префикс @ для дословного вывода строк вне зависимости от наличия в них управляющих последовательностей. Строки с этим префиксом называются “дословными” (verbatim strings). Это — очень удобное средство во многих ситуациях:

```
// Префикс @ отключает обработку
// управляющих последовательностей
string finalString - @"\n\tString file:
'C:\CSnarpProjects\Strings\string.cs'";
Console.WriteLine(finalString);
```

Применение System.Text.StringBuilder

- При работе со строками в C# необходимо помнить очень важную вещь: значение строки не может быть изменено.
- Все методы, казалось бы, изменяющие строку, на самом деле лишь возвращают её измененную копию.

Пример

```
// Вносим изменения в строку? На самом деле
// нет...
System.String strFixed = "This is how I began life";
Console.WriteLine(strFixed);
string upperVersion = strFixed.ToUpper();
// Возвращает "прописную" копию strFixed
Console.WriteLine(strFixed);
Console.WriteLine(upperVersion);
```

- Работа с копиями копий может надоесть. Поэтому в C# существует класс, позволяющий изменять строки напрямую – это класс StringBuilder, определенный в пространстве имен System.Text. Он во многом напоминает CString в MFC.
- Все изменения, которые вы вносите в объект этого класса, немедленно в нем отражаются, что во многих ситуациях гораздо эффективнее, чем работать с множеством копий.

```
// Демонстрирует применение класса StringBullder
using System;
using System.Text; // Здесь живет StringBuilder!
class StringApp
{
    public static int Main(stnng[] args)
    {
     // Создаем объект StringBuilder и изменяем его
     //содержимое
     StringBuilder myBuffer = new StringBuilder("I am a
    buffer");
     myBuffer.Append(" that just got longer...");
     Console.WriteLine(myBuffer );
     return 0;
    }
}
```

- Помимо добавления класс StringBuilder допускает и другие операции, например удаление определенных символов или их замену.
- После того как вы добились нужного вам результата, часто бывает удобным вызвать метод ToString( ), чтобы перевести содержимое объекта StringBuilder в обычный тип данных String.

```
using System;
using System.Text;
class StringApp
{
public static int Main(string[] args)
{
    StringBuilder myBuffer = new StringBuilder("I am a
    buffer");
    myBuffer.Append(" that just got longer...");
    Console.WriteLine(myBuffer) ;
    myBuffer.Append("and even longer.");
    Console. WriteLine(myBuffer);
    // Делаем все буквы прописными
    string theReallyFinalString = myBuffer.ToString().
    ToUpper();
    Console.WriteLine(theReallyFinalString);
    return 0; } }
```

Перечисления в C#

- Часто бывает удобным создать набор значимых имен, которые будут представлять числовые значения.

```
// Создаем перечисление
enum EmpType
{
Manager,     // = 0
Grunt,       // = 1
Contractor,  // = 2
VP           // = 3
}
```

```
// Элементы перечисления могут иметь
// произвольные числовые значения
   enum EmpType
   {
   Manager = 10,
   Grunt = 1,
   Contractor = 100,
   VP = 99
   }
```

- При компиляции, компилятор попросту подставляет вместо элементов перечисления соответствующие числовые значения.
- По умолчанию для этих числовых значений компилятор использует тип данных Int. Однако ничто не мешает явным образом объявить компилятору, что следует использовать другой тип данных, например, byte.

```
// Вместо элементов перечисления будут
// подставляться числовые значения типа byte
enum EmpType : byte
{
   Manager = 10,
   Grunt = 1,
   Contractor = 100,
   VP = 9
}
```

- Точно таким же образом можно использовать любой из основных целочисленных типов данных C#(byte, sbyte, short, ushort, int, uint, long, ulong).

Базовый класс System.Enum

- Все перечисления в C# происходят от единого базового класса System.Enum. Конечно же, в этом базовом классе предусмотрены методы, которые могут существенно облегчить работу с перечислениями.

GetUnderlyingType ( )

- Это статический метод, который позволяет получить информацию о том, какой тип данных используется для представления числовых значений элементов перечисления:

```
// Получаем тип числовых данных
// перечисления (в нашем примере это будет
// System. Byte)
Console.WriteLine(Enum.
GetUnderlyingType(typeof(EmpType)));
```

Enum.Format ( )

- Статический метод, который может получать значимые имена элементов перечисления по их числовым значениям.
- В нашем примере переменной типа EmpType соответствовало имя элемента перечисления Contractor (то есть эта переменная разрешалась в числовое значение 100).
- Для того чтобы узнать, какому элементу переменной соответствует это числовое значение, необходимо вызвать метод Enum.Format, указать тип перечисления

числовое значение (в нашем случае через переменную) и флаг форматирования (в нашем случае — G, что означает вывести как тип string)

```
// Этот код должен вывести на системную
// консоль строку "You are a Contractor
EmpType fred;
fred = EmpType.Contractor;
Console.WriteLine(“You are a {0}”, Enum.
   Format (typeof( EmpType), fred, "G"));
```

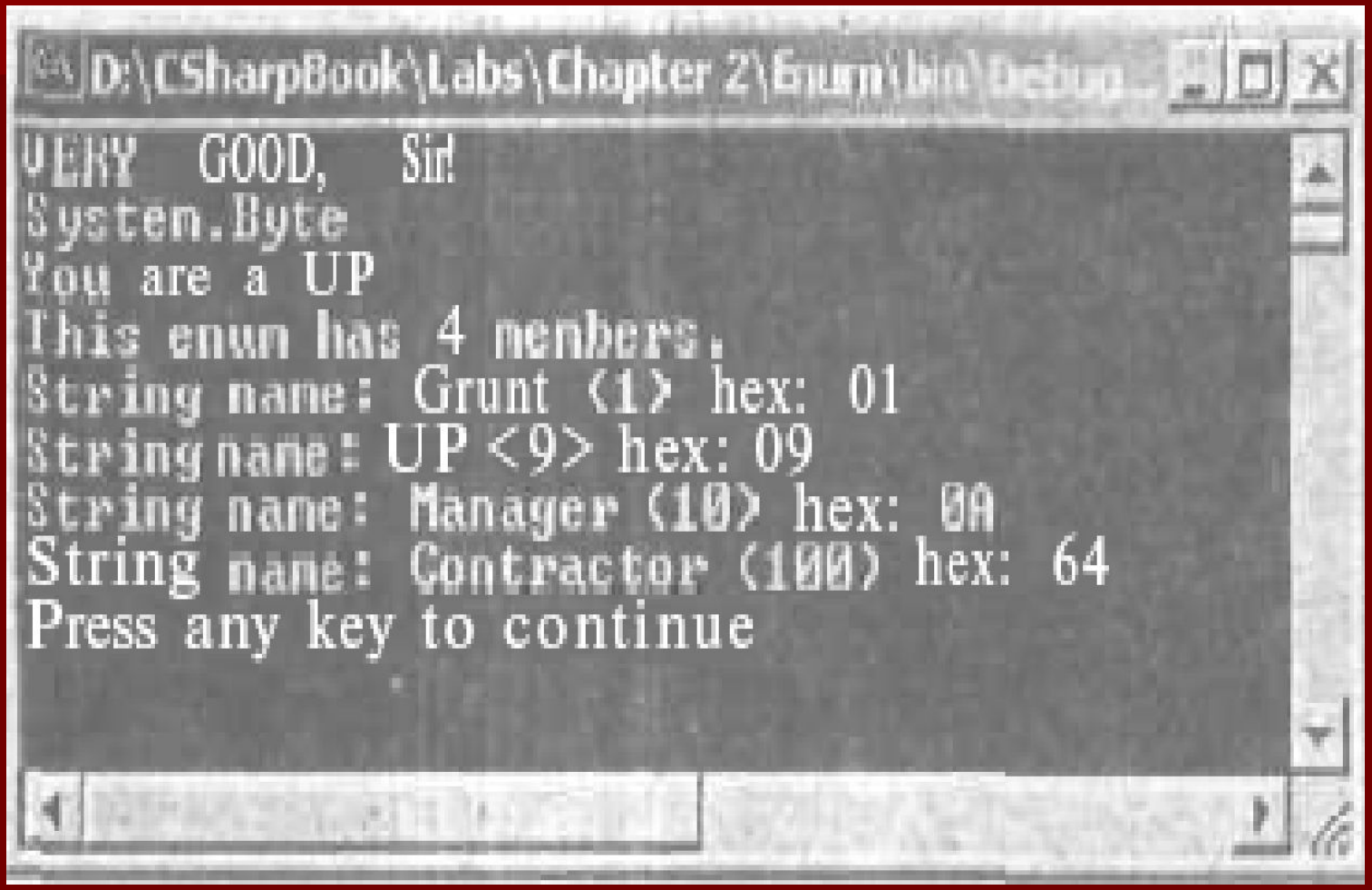
GetValues()

- Статический метод, который возвращает экземпляр System.Array, при этом каждому элементу массива будет соответствовать член указанного перечисления.

```
// Получаем информацию о количестве
// элементов в перечислении
Array obj = Enum.GetValues(typeof(EmpType));
Console. WriteLine("This enum has {0} members.", obj.Length);
```

```
// А теперь выводим имена элементов
// перечисления в формате string и
// соответствующие им числовые значения
foreach(EmpType e in obj)
{
   Console. Write("String name: {0}“, Enum.
   Format (typeof (EmpType), e, "G"));
   Console. Write(" ({0})“, Enum. Format(typeof(
   EmpType), e, "D"));
   Console. Write(" hex: {0}\n“, Enum. Format (
   typeof (EmpType), e, "X"));
}
```

Результат работы программы



IsDefined

- Свойство класса System.Enum, которое позволяет определить, является ли выбранная вами символьная строка элементом указанного перечисления.
- Например, предположим, что требуется узнать, является ли значение Salesperson элементом перечисления EmpType.

```
// Есть ли в EmpType элемент
// Salesperson?
if (Enum. IsDefined(typeof (EmpType).
   "Salesperson"))
   Console. WrlteLine(“Yes, we have sales
   people.”);
else
   Console. WriteLine(“No, we have no
   profits...”);
```

- Перечисления C# поддерживают работу с большим количеством перегруженных операторов, которые могут выполнять различные операции с числовыми значениями переменных.

```
// Какому из этих двух переменных-членов
// перечисления соответствует большее числовое
// значение?
EmpType Joe = EmpType. VP:
EmpType Fran = EmpType. Grunt;
if (Joe < Fran)
  Console. WrlteLine(“Joe's value is less than Fran's”);
else
  Console. WnteLine(“Fran's value is less than Joe's”);
```

Структуры в C#

Определение структур в C#

- Структуры C# можно рассматривать как некую особую разновидность классов.
- Для структур можно определять конструкторы (только принимающие параметры).
- Структуры могут реализовывать интерфейсы.

- Структуры могут содержать любое количество внутренних членов.
- Для структур C# не существует единого базового класса (тип System.Structure в C# не предусмотрен).
- Косвенно все структуры являются производными от типа ValueType.

Пример

```
// Вначале нам потребуется наше
// перечисление
enum EmpType : byte
{
   Manager = 10, Grunt = 1, Contractor = 100,
   VP = 9
}
struct EMPLOYEE
{
```

```
  public EmpType title; // Одно из полей структуры -     //
перечисление, определенное выше
  public string name;
  public short deptID;
}
class StructTester
{
  public static int Main(string[] args)
  {
        // Создаем и присваиваем значения Фреду
        EMPLOYEE fred;
        fred.deptID = 40;
        fred.name = "Fred";
        fred.title = EmpType.Grunt;
        return 0;
  }
}
```

- Вполне возможно, что в реальном приложении для более удобного присвоения значений членам структуры придется определить свой собственный конструктор или несколько конструкторов.
- Нужно помнить, что нельзя переопределить конструктор для структуры по умолчанию — тот конструктор, который не принимает параметров.

- Все ваши конструкторы обязательно должны принимать один или несколько параметров.

```
// Для структур можно определить
// конструкторы, но все они должны
// принимать параметры
struct EMPLOYEE
{
// Поля
public EmpType title;
```

```
 public string name;
 public short deptID;
 // Конструктор
 public EMPLOYEE (EmpType et, string n, short
d)
 {
     title = et;
     name = n;
     deptID = d;
 }
}
```

- При помощи такого определения структуры, в котором предусмотрен конструктор, вы можете создавать новых сотрудников следующим образом:

```
class StructTester
{
 // Создаем Мэри и присваиваем ей
 //значения при помощи конструктора
 public static int Main(string[] args)
```

```
{
    // Для вызова нашего
    // конструктора мы обязаны
    // использовать ключевое слово
    // new
    EMPLOYEE mary = new
EMPLOYEE(EmpType.VP, “Mary”, 10);
    return 0;
 }
}
```

- Структуры могут быть использованы в качестве принимаемых и возвращаемых методами параметров.

Упаковка и распаковка

- Упаковка и распаковка — это наиболее удобный способ преобразования структурного типа в ссылочный, и наоборот.
- Основное назначение структур — возможность получения некоторых преимуществ объектной ориентации, но при более высокой производительности за счет размещения в стеке.

- Чтобы преобразовать структуру в ссылку на объект, необходимо упаковать ее экземпляр:

```
// Создаем и упаковываем нового
// сотрудника
EMPLOYEE Stan = new
EMPLOYEE(EmpType.Grunt, “Stan”, 10);
object stanInBox = stan;
//stanInBox относится к ссылочным типам
// данных, но при этом сохраняет
//внутренние значения исходного типа
// данных EMPLOYEE
```

- Можно использовать stan во всех случаях, когда нужен объект, и при необходимости производить распаковку:

```
//Поскольку мы ранее произвели
//упаковку данных, мы можем
//распаковать их и производить операции
//с содержимым
public void UnboxThisEmployee(object o)
```

```
{
    // Производим распаковку в структуру
    // EMPLOYEE для получения доступа
    // ко всем полям
    EMPLOYEE temp = (EMPLOYEE)o;
    Console.WriteLine(temp.name +“is
    alive!”);
}
```

- Вызов этого метода может выглядеть следующим образом:

```
// Передаем упакованного сотрудника на
// обработку
t.UnboxThisEmployee(stanInBox);
```

- Компилятор C# при необходимости автоматически производит упаковку. Поэтому допускается передать объект stan (типа EMPLOYEE) напрямую:

```
// Stan будет упакован автоматически
t.UnboxThisEmployee(stan);
```

Пользовательские пространства имен в C#

Определение пользовательских пространств имен

- Часто бывает очень полезным сгруппировать используемые в приложении типы данных в специально созданные для этой цели пространства имен.
- В C# эта операция производится при помощи ключевого слова namespace.

Пример

```
// shapeslib.cs
namespace MyShapes
{
   using System;
   // Класс Circle
   public class Circle { // Интересные
   методы }
```

```
public class Hexagon // Класс Hexagon
{
  // Более интересные методы
}
public class Square // Класс Square
{
 // Еще более интересные методы
}
}
```

- Пространство имен MyShapes действует как контейнер для всех этих типов.
- Можно разбить единое пространство имен C# на несколько физических файлов. Для этого достаточно просто определить в разных файлах одно и то же пространство имен и поместить в него определения классов.

```
// circle.cs
namespace MyShapes
{
   using System;
   // Класс Circle
   class Circle { // Интересные методы }
}
```

Подобным образом определяются hexagon.cs и square.cs.

- Если потребуется использовать эти классы внутри другого приложения, удобнее всего это сделать при помощи ключевого слова using:

```
namespace MyApp
{
   using System;
   using MyShapes;
   class ShapeTester
   {
```

```
    public static void Main()
    {
     // Все эти объекты были определены
     // в пространстве имен MyShapes
     Hexagon h = new Hexagon();
     Circle c = new Circle();
     Square s = new Square();
    }
}
}
```

Применение пространств имен для разрешения конфликтов между именами классов

- Пространства имен могут быть использованы для разрешения конфликтов между именами объектов в тех ситуациях, когда в нашем приложении используются разные объекты с одинаковыми именами.

```
// Еще одно пространство имен для
// геометрических фигур
namespace My3DShapes
{
    using System;
    // Класс 3D Circle
    class Circle{}
    // Класс 3D Hexagon
    class Hexagon{}
    // Класс 30 Square
    class Square{}
}
```

```
// В коде есть двусмысленность!
namespace MyApp
{
    using System:
    using MyShapes;
    using My3DShapes:
    class ShapeTester
    {
    public static void Main()
    { // Неизвестно, к объектам какого пространства имен мы
      //обращаемся
      Hexagon h = new Hexagon();
      Circle c = new Circle();
      Square s = new Square();
    }
    }
}
```

- Проще всего избавиться от подобных конфликтов, указав для каждого класса его полное имя вместе с именем соответствующего пространства имен:

```
// Конфликтов больше нет
public static void Main()
{
   My3DShapes.Hexagon h = new
   My3DShapes.Hexagon();
   My3DShapes.Circle c = new
   My3DShapes.Circle();
   My3DShapes.Square s = new
   MyShapes.Square();
}
```

Использование псевдонимов для имен классов

- Еще одна возможность избавиться от конфликтов имен — использовать для имен классов псевдонимы.

```
namespace MyApp
{
   using System;
   using MyShapes:
   using My3DShapes;
```

```
// Создаем псевдоним для класса из
// другого пространства имен
using The3DHexagon = My3DShapes. Hexagon ;
class ShapeTester
{
   public static void Main()
   {
    Hexagon h = new Hexagon();
    Circle c = new Circle();
    Square s = new Square();
```

```
// Создаем объект при помощи
// псевдонима
    The3DHexagon h2 = new
   The3DHexagon();
        }
   }
}
```

Вложенные пространства имен

- Можно без каких-либо ограничений вкладывать одни пространства имен в другие.
- Такой подход очень часто используется в библиотеках базовых классов .NET для организации типов.

Пример

```
// Классы для геометрических фигур
// расположены в пространстве имен
// Chapter2Types.My3DShapes
namespace Chapter2Types
{
   namespace My3DShapes
   {
    using System;
```

```
    // Класс 3D Circle
    class Circle{}
    // Класс 3D Hexagon
    class Hexagon{}
    // Класс 30 Square
    class Squared
  }
}
```

Конец!!!