ОБРАЗОВАТЕЛЬНЫЙ ЦЕНТР
ПРОГРАММИРОВАНИЯ И ВЫСОКИХ ТЕХНОЛОГИЙ

2020г.

Массивы и строки

AGENDA

- ✓ Одномерные массивы
- ✓ Многомерные массивы
- ✓ Ступенчатые массивы
- ✓ Строки и класс StringBuilder
- ✓ Сравнение строк

Начальные сведения о массивах

Массивы – упорядоченные коллекции элементов с одним и тем же типом данных. **CLR** поддерживает одномерные, многомерные и неравномерные массивы. Базовым для всех массивов является абстрактный класс **System.Array**, производный от **System.Object**.

Массивы относятся к ссылочному типу и размещаются в управляемой куче, а переменная в приложении содержит не элементы массива, а ссылку на массив.

Элементы массива всегда хранятся в непрерывном блоке памяти.

При создании массива всегда происходит инициализация его элементов стандартными значениями.

```
Int32[] myIntegers; // Объявление ссылки на массив
myIntegers = new Int32[100]; // Создание массива типа Int32

//Создание массива можно совместить с его объявлением:
int[] data = new int[10];
```

Начальные сведения о массивах

Свойство **Length** массива возвращает количество элементов в массиве. Изменить длину массива после его создания невозможно.

Для обращения к элементам массива используются индексы. Индекс представляет номер элемента в массиве, при этом нумерация начинается с нуля.

По умолчанию в качестве индекса массива используется значение типа **Int32** с начальным значением «0».

Для доступа к элементу массива указывается имя массива и индекс в квадратных скобках: **data[0] = 10;**

```
// Способы задания элементов массива при создании:
int[] data_1 = new int[4] { 1, 2, 3, 5 };
int[] data_2 = new int[] { 1, 2, 3, 5 };
int[] data_3 = new[] { 1, 2, 3, 5 };
int[] data_4 = { 1, 2, 3, 5 };
```

Класс «Array»

Все массивы в **C#** построены на основе класса **Array** из пространства имен **System**. Этот класс определяет ряд свойств и методов, которые мы можем использовать при работе с массивами.

Свойство **Length** – возвращает длину массива.
Свойство **Rank** – возвращает размерность массива.
Метод **BinarySearch()** – выполняет бинарный поиск.
Метод **Clear()** – очищает массив, устанавливая значение по умолчанию.
Метод **Exists()** – проверяет, содержит ли массив определенный элемент.
Метод **Find()** – находит элемент, который удовлеворяют условию.
Метод **FindAll()** – находит все элементы, которые удовлеворяют условию.
Метод **IndexOf()** – возвращает индекс элемента.
Метод **Resize()** – изменяет размер одномерного массива.
Метод **Sort()** – сортирует элементы одномерного массива.
Метод **GetUpperBound()** – получает индекс последнего элемента указанного измерения в массиве.

У каждого из этих методов существует множество перегруженных версий. Для многих из них имеются обобщенные перегруженные версии, обеспечивающие контроль типов во время компиляции и высокую производительность.

Одномерные массивы

Одномерный массив хранит фиксированное число элементов в линейном порядке, и для определения каждого элемента требуется лишь одно значение индекса.

```
// Инициализация в цикле

int[] TaxRates = new int[5];

for (int i=0; i<TaxRates.Length; i++)
{
    TaxRates[i] = 0;
}
```

Многомерные массивы

Массивы характеризуются таким понятием как **ранг** или количество измерений.

Соответственно могут быть массивы с различным количеством измерений. Но на практике обычно используются одномерные и двухмерные массивы.

Определенную сложность может представлять перебор многомерного массива. Надо учитывать, что длина такого массива - это совокупное количество элементов.

```
// Двумерный массив d:
int[,] d;
d = new int[10,2];

// Трехмерный массив Cube:
int[,,] Cube = new int[3,2,5];

// Объявим двумерный массив и инициализируем его:
int[,] c = new int[2,4] {
                          {1, 2, 3, 4},
                          {10, 20, 30, 40}};
```

Ступенчатые массивы

Ступенчатый массив (массив массивов, невыровненный массив) – это массив, элементы которого сами являются массивами.

Внутренние измерения в объявлении не указываются, т.к. в отличие от прямоугольного массива каждый внутренний массив может иметь произвольную длину.

Каждый внутренний массив неявно инициализируется **null**, а не пустым массивом.

Каждый внутренний массив должен быть создан вручную.

```
   int[][] myArray = new int[3][];     // будет 3 одномерных
массива

   myArray[0] = new int[] { 1, 3, 5, 7, 9 };
   myArray[1] = new int[] { 0, 2, 4, 6 };
   myArray[2] = new int[] { 11, 22 };
```

Строки и класс StringBuilder

Тип **string** (псевдоним класса **System.String**) в **C#** является неизменяемой последовательностью символов.

Поддерживает и определяет символьные строки.

Является примитивным типом.

Является ссылочным типом данных.

Пустая строка имеет нулевую длину. Чтобы создать пустую строку, можно использовать либо литерал, либо статическое поле **string.Empty**, для проверки, пуста ли строка, можно либо выполнить сравнение эквивалентности, либо просмотреть свойство **Length** строки.

Поскольку класс **String** неизменяемый, все методы, которые «манипулируют» строкой, возвращают новую строку, оставляя исходную незатронутой (то же самое происходит при повторном присваивании строковой переменной).

Строки и класс StringBuilder

Создание строк – создавать сроки можно, как используя переменную типа **string** и присваивая ей значение, так и применяя один из конструкторов класса **String**:

```
string s1 = "hello";
string s2 = null;
string s3 = new String('a', 6);
string s4 = new String(new char[]{'w', 'o', 'r', 'l', 'd'});
```

Строка как набор символов.

Так как строка хранит коллекцию символов, в ней определен индексатор для доступа к этим символам:

```
string s1 = "hello";
char ch1 = s1[1]; // символ 'e'
```

Строки и класс StringBuilder

Конкатенация. Конкатенация строк может производиться с помощью оператора **+**, и с помощью метода **Concat**:

```
string s1 = "hello";
string s2 = "world";
string s3 = s1 + " " + s2;
// результат: строка "hello world"
string s4 = String.Concat(s3, "!!!");
// результат: строка "hello world!!!"
```

Интерполяция строк. Начиная с версии языка C# 6.0 была добавлена такая функциональность, как интерполяция строк.

```
Console.WriteLine($"Имя: {person.Name} Возраст: {person.Age}");
```

Знак доллара перед строкой указывает, что будет осуществляться интерполяция строк. Внутри строки опять же используются плейсхолдеры {...}, только внутри фигурных скобок уже можно напрямую писать те выражения, которые мы хотим вывести.

Строки и класс StringBuilder

Форматирование строк. Вместо конкатенации мы можем применять форматирование:

```
Person person = new Person { Name = "Tom", Age = 23 };
Console.WriteLine("Имя: {0} Возраст: {1}",
   person.Name, person.Age);
```

То же самое мы можем сделать с помощью метода **String.Format**:

```
string output = String.Format("Имя: {0}  Возраст: {1}",
   person.Name, person.Age);
```

В методе **Format** могут использоваться различные спецификаторы и описатели, которые позволяют настроить вывод данных.

```
string result = String.Format("{0:C2}", 23.7);  // $23.70
```

Строки и класс StringBuilder

Класс **StringBuilder** применяется когда необходимо выполнить большое количесво операций над текстом большого объема.

Microsoft рекомендует использовать класс **String** в следующих случаях:

- При небольшом количестве операций и изменений над строками.
- При выполнении фиксированного количества операций объединения. В этом случае компилятор может объединить все операции объединения в одну.
- Когда надо выполнять масштабные операции поиска при построении строки, например **IndexOf** или **StartsWith**. Класс **StringBuilder** не имеет подобных методов.

Класс **StringBuilder** рекомендуется использовать в следующих случаях:

- При неизвестном количестве операций и изменений над строками во время выполнения программы.
- Когда предполагается, что приложению придется сделать множество подобных операций.

Сравнение строк

При сравнении двух значений в **.NET Framework** проводится различие между концепциями сравнения эквивалентности и сравнения порядка.

Сравнение эквивалентности проверяет, являются ли два экземпляра семантически одинаковыми; сравнение порядка выясняет, какой из двух экземпляров будет следовать первым в случае расположения их по возрастанию или убыванию.

Для сравнения эквивалентности строк можно использовать операцию **==** или один из методов **Equals** типа **string**. Последние являются более универсальными, потому что позволяют указывать такие опции, как нечувствительность к регистру символов.

Тип **string** не поддерживает операторы «**<**» и «**>**» для сравнений. Для сравнения порядка строк можно применять либо метод экземпляра **CompareTo**, либо статические методы **Compare** и **CornpareOrdinal**: они возвращают положительное или отрицательное число либо ноль - в зависимости от того, находится первое значение до, после или рядом со вторым.

Регулярные выражения

Используются при обработке больших текстов, позволяя существенно уменьшить объемы кода по сравнению с использованием стандартных операций со строками.

Синтаксис регулярных выражений:
^ – соответствие должно начинаться в начале строки.
$ – соответствие должно быть в конце строки.
. – знак точки определяет любой одиночный символ.
***** – предыдущий символ повторяется 0 и более раз.
+ – предыдущий символ повторяется 1 и более раз.
? – предыдущий символ повторяется 0 или 1 раз.
\s – соответствует любому пробельному символу.
\S – соответствует любому символу, не являющемуся пробелом.
\w – соответствует любому алфавитно-цифровому символу.
\W – соответствует любому не алфавитно-цифровому символу.
\d – соответствует любой десятичной цифре.
\D – соответствует любому символу, не являющемуся десятичной цифрой.

Регулярные выражения

```
Regex regex = new Regex(@"(\w*)рек(\w*)");
MatchCollection matches = regex.Matches(“река в реке”);
MatchCollection matches = regex.Matches(s);

if (matches.Count > 0)
{
    foreach (Match match in matches)
        Console.WriteLine(match.Value);
}
else
{
    Console.WriteLine("Совпадений не найдено");
}
```