

Технология программирования

Строки

Работа с текстом

В языке C# имеется много различных инструментов для работы с текстом

- Встроенные типы данных – `char` и `string`
- Классы из стандартной библиотеки – `StringBuilder`
- Классы для работы с регулярными выражениями
- Средства LINQ для строк

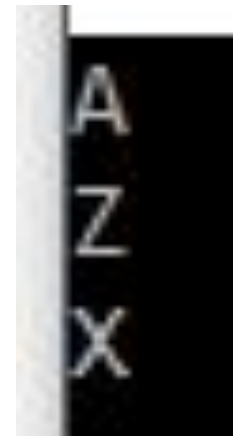
Отдельные символы

Фреймворк **.NET** содержит специальный класс для работы с отдельными символами – **System.Char**

- Использует **UTF-16** для хранения одной буквы, таким образом символ занимает **2 байта**
- В **C#** синонимом данного класса является встроенный тип **char**
- Также в языке определены **СИМВОЛЬНЫЕ КОНСТАНТЫ**, задающиеся
 - Символом, заключенным в одинарные кавычки
 - Управляющей последовательностью
 - Unicode-последовательностью

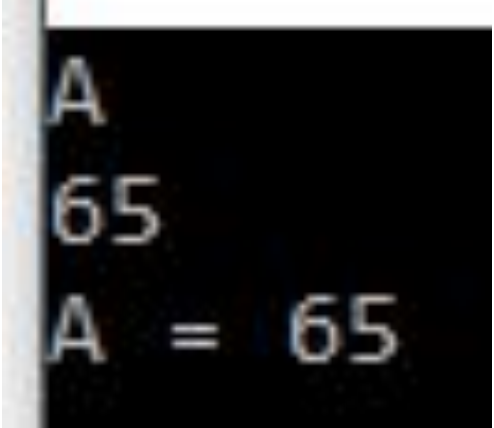
Отдельные символы

```
char ch = 'A';  
WriteLine(ch);  
ch = '\x5A';  
WriteLine(ch);  
ch = '\u0058';  
WriteLine(ch);
```



Отдельные символы

```
char ch = new Char();  
ch = (char)65;  
WriteLine(ch);  
int code = ch;  
WriteLine(code);  
string s = ch.ToString() + " = " + code;  
WriteLine(s);
```



A
65
A = 65

Отдельные символы

Цифры и буквы алфавитов обычно кодируются интервалами:

- **0 - 9** соответствует интервал **[65, 90]**
- **A - Z** соответствует интервал **[65, 90]**
- **a - z** соответствует интервал **[97, 122]**
- **А - Я** соответствует интервал **[1040, 1071]**
- **а - я** соответствует интервал **[1072, 1103]**
- Однако, буквам **Ё** и **ё** присвоены коды **1025** и **1105**

Отдельные символы

Класс `char` содержит достаточно много собственных методов. Большая часть из них используется для определения типа символа.

- **`IsDigit`** – проверка на десятичную цифру
- **`IsLetter`** – проверка на букву
- **`IsPunctuation`** – проверка на знак препинания
- **`IsWhiteSpace`** – проверка на пробелы, включая перевод строки и возврат каретки

Отдельные символы

В общем случае тип символа можно получить при помощи метода **GetUnicodeCategory**. Он возвращает одну из категорий, описанных в перечислении **UnicodeCategory** из пространства имен **System.Globalization**.

Отдельные символы

```
WriteLine("GetUnicodeCategory:");  
var c1 = char.GetUnicodeCategory('A');  
var c2 = char.GetUnicodeCategory(';');  
WriteLine("'A' - category {0}", c1);  
WriteLine("';' - category {0}", c2);
```

Отдельные символы

```
GetUnicodeCategory:  
'A' - category UppercaseLetter  
'; ' - category OtherPunctuation  
Метод IsLetter:  
'z' - IsLetter - True  
'я' - IsLetter - True  
Метод IsControl:  
'; ' - IsControl - False  
'\r' - IsControl - True  
Метод IsSeparator:  
' ' - IsSeparator - True  
'; ' - IsSeparator - False  
Метод IsWhiteSpace:  
' ' - IsWhiteSpace - True  
'\r' - IsWhiteSpace - True
```

Отдельные символы

```
WriteLine("Метод IsLetter:");  
WriteLine("'z' - IsLetter - {0}",  
    char.IsLetter('z'));  
WriteLine("'Я' - IsLetter - {0}",  
    char.IsLetter('Я'));  
  
WriteLine("Метод IsControl:");  
WriteLine("';' - IsControl - {0}",  
    char.IsControl(';'));  
WriteLine(@"'\r' - IsControl - {0}",  
    char.IsControl('\r'));
```

Отдельные символы

```
WriteLine("Метод IsSeparator:");  
WriteLine("' ' - IsSeparator - {0}",  
    char.IsSeparator(' '));  
WriteLine("';' - IsSeparator - {0}",  
    char.IsSeparator(';'));  
  
WriteLine("Метод IsWhiteSpace:");  
WriteLine("' ' - IsWhiteSpace - {0}",  
    char.IsWhiteSpace(' '));  
WriteLine(@"'\r' - IsWhiteSpace - {0}",  
    char.IsWhiteSpace('\r'));
```

Отдельные символы

- **ToLower** – приводит символ к нижнему регистру
- **ToUpper** – приводит символ к верхнему регистру
- **CompareTo** – сравнивает два символа и возвращает разницу между их кодами

Массив символов

По аналогии с языками С и С++ мы можем представить строку в виде массива символов. Особого смысла это не имеет, т.к. у нас уже есть тип **string**, но может быть полезно для понимания внутреннего устройства различных классов по обработке текста. Над массивом символов сразу можно выполнять все операции с массивами: **Copy**, **IndexOf**, **LastIndexOf** и т.д.

Массив СИМВОЛОВ

```
char[] text = new [] {'H', 'e', 'l', 'l', 'o'};
for (int i = 0; i < text.Length; ++i) {
    if (char.IsLower(text[i])) {
        text[i] = char.ToUpper(text[i]);
    } else {
        text[i] = char.ToLower(text[i]);
    }
}
foreach (var ch in text) {
    Write(ch);
}
WriteLine();
```



hELLO

Массив символов

Стандартный класс `string` внутри представлен в виде массива символов. Поэтому у него есть стандартный метод для его получения – **`ToCharArray`**

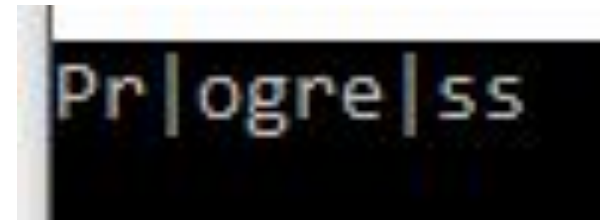
```
char[] text = "Hello".ToCharArray();
```


Массив символов

```
static int IndexOf(char[] text1,  
                  char[] text2) {  
    for (int i = 0; i < text1.Length; ++i) {  
        bool isFound = true;  
        for (int j = 0; j < text2.Length; ++j) {  
            if (i + j >= text1.Length ||  
                text1[i + j] != text2[j]) {  
                isFound = false;  
                break;  
            }  
        }  
        if (isFound) { return i; }  
    } return -1; }
```

Массив СИМВОЛОВ

```
char[] text1 = "Progress".ToCharArray();
char[] text2 = "ogre".ToCharArray();
int index = IndexOf(text1, text2);
if (index >= 0) {
    for (int i = 0; i < text1.Length; ++i) {
        if (i == index ||
            i == index + text2.Length) {
            Write("|");
        }
        Write(text1[i]);
    }
} else { WriteLine("Не найдено"); }
```



Строки string

Строки можно создавать при помощи строковых констант или конструктора `string`. Конструктор имеет много вариантов, но наиболее полезными являются:

- Создание строки из символа, повторенного заданное число раз
- Создание строки из массива символов `char[]`
- Создание строки из части массива СИМВОЛОВ

Строки string

```
string hello = "Hello";  
string separator = new string('-', 5);  
char[] array = hello.ToCharArray();  
string fromArray = new string(array);  
string strye = new string(array, 1, 3);
```

```
WriteLine(hello);  
WriteLine(separator);  
WriteLine(fromArray);  
WriteLine(strye);
```

```
Hello  
-----  
Hello  
ell
```

Строки string

В C# существуют два вида строковых констант:

- Обычные константы, которые представляют строку символов, заключенную в кавычки
- Константы с предшествующим знаком @.

Строки string

- Обычные строковые константы могут содержать управляющие последовательности - `\n`, `\t`, `\r` и т.д.
- В `@`-константах все символы трактуются в полном соответствии с их изображением. Символ кавычки внутри строки задается при помощи удвоения символа

Строки string

```
WriteLine("\x50");  
WriteLine(@"\x50");  
WriteLine("c:\\folder\\folder\\file.txt");  
WriteLine(@"c:\folder\folder\file.txt");  
WriteLine("\"A\"");  
WriteLine(@""A""");
```

```
P  
\x50"  
c:\folder\folder\file.txt  
c:\folder\folder\file.txt  
"A"  
"A"
```

Строки string

Над строками определены следующие операции:

- Присваивание =
Строки являются ссылочным типом. Если есть две строки $s1$ и $s2$, то в результате выполнения выражения $s2 = s1$ произойдет копирование **ССЫЛОК** на текст, а не самого текста
- Проверка эквивалентности $==$ и $!=$
- Конкатенация или сцепление строк $+$
- Взятие индекса $[]$
Индексы доступны **только для чтения**

Строки string

```
string s1 = "ABC", s2 = "DEF";  
string s3 = s1 + s2;  
string s4 = "ABCDEF";  
WriteLine(s3);  
WriteLine("s3 == s4 ? {0}", s3 == s4);  
WriteLine("s1[2] = {0}", s1[2]);
```

```
ABCDEF  
s3 == s4 ? True  
s1[2] = C
```

Методы класса string

- **Compare** – сравнение двух строк. Различные варианты метода позволяют сравнивать как строки, так и подстроки. При этом можно учитывать или не учитывать регистр, особенности национального форматирования дат, чисел и т.д.
- **CompareOrdinal** – сравнение двух строк. Сравниваются коды символов

Методы класса string

- **Concat** – конкатенация строк. Допускает сцепление произвольного числа строк
- **Copy** – создается копия строки
- **Format** – выполняет форматирование в соответствии с заданными спецификациями формата (*используется в методе WriteLine, см. первую лекцию*)

Методы класса string

- **Join** – конкатенация массива строк в единую строку. При этом между элементами массива вставляются разделители
- **Split** – осуществляет разделение строки на элементы

Методы класса string

```
var txt = "А это пшеница, которая в темном"  
         "чулане хранится," +  
         " в доме, который построил
```

```
WriteLine(txt);
```

```
А это пшеница, которая в темном чулане хранится,
```

```
 в доме, который построил Джек!
```

Методы класса string

```
string[] sentences = txt.Split(',');  
for (int i = 0; i < sentences.Length; i++) {  
    WriteLine("sentences[{0}] = {1}",  
              i, sentences[i]);  
}  
WriteLine();
```

```
sentences[0] = А это пшеница  
sentences[1] = которая в темном чулане хранится  
sentences[2] = в доме  
sentences[3] = который построил Джек!
```

Методы класса string

```
string join = string.Join(",", sentences);  
WriteLine("join = {0}", join);  
WriteLine();
```

```
join = А это пшеница, которая в темном чулане хранится,  
в доме, который построил Джек!
```

Методы класса string

```
WriteLine();  
string[] words = txt.Split(' ');  
for (int i = 0; i < words.Length; i++) {  
    WriteLine("words[{0}] = {1}",  
             i, words[i]);  
}  
WriteLine();
```

```
words[0] = А  
words[1] = это  
words[2] = пшеница,  
words[3] = которая  
words[4] = в  
words[5] = темном  
words[6] = чулане  
words[7] = хранится,  
words[8] = в  
words[9] = доме,  
words[10] = который  
words[11] = построил  
words[12] = Джек!
```


Методы класса string

```
join = string.Join(" ", words);  
WriteLine("join = {0}", join);  
WriteLine();
```

```
join = А это пшеница, которая в темном чулане хранится,
```

```
в доме, который построил Джек!
```

Методы класса string

Методы **Split** и **Join** хорошо работают, когда при разборе используется только один разделитель. В этом случае сборка действительно является обратной операцией и позволяет восстановить исходную строку. Если же при разборе задается некоторое множество разделителей, то возникают две проблемы:

- Невозможно при сборке восстановить строку в прежнем виде, поскольку не сохраняется информация о том, какой из разделителей был использован при разборе строки
- Если при разборе предложения на слова использовать в качестве разделителей пробел и запятую, то запятая исчезнет как часть слова, но взамен появятся пустые слова

Методы класса string

- **Insert** – вставляет подстроку в заданную позицию
- **Remove** – удаляет подстроку в заданной позиции
- **Replace** – заменяет подстроку в заданной позиции на новую подстроку
- **Substring** – выделяет подстроку в заданной позиции

Методы класса string

- **IndexOf, IndexOfAny, LastIndexOf, LastIndexOfAny** – определяют индексы первого и последнего вхождения заданной подстроки или любого символа из заданного набора
- **StartsWith, EndsWith** – возвращается true или false, в зависимости от того, начинается или заканчивается строка заданной подстрокой
- **PadLeft, PadRight** – вставляют нужное число пробелов в начале или в конце строки
- **Trim, TrimStart, TrimEnd** – удаляются пробелы в начале или в конце строки

Класс StringBuilder

В языке C# существует понятие **неизменяемый (immutable) класс**. Для такого класса невозможно изменить значение объекта. Методы могут создавать новый объект на основе существующего, но не могут изменить значение существующего объекта.

Класс StringBuilder

К таким **неизменяемым** классам относится и класс **string**. Ни один из методов этого класса не меняет значения существующих объектов. Когда метод изменяет строку, результатом является **новая строка** - новый объект в куче.

При работе со строкой как с массивом разрешено только **чтение** отдельных символов, но не их замена.

Класс `StringBuilder`

Класс `StringBuilder` позволяет компенсировать этот недостаток. Этот класс принадлежит к изменяемым классам, и его можно найти в пространстве имен `System.Text`

Класс StringBuilder

Специальных констант этого типа не существует, поэтому объекты этого класса объявляются с явным вызовом конструктора класса.

- `StringBuilder(string str, int cap)`
str - начальная строка
cap - **емкость** объекта
- `StringBuilder(int cur, int max)`
cur – начальная, **max** – максимальная емкость объекта

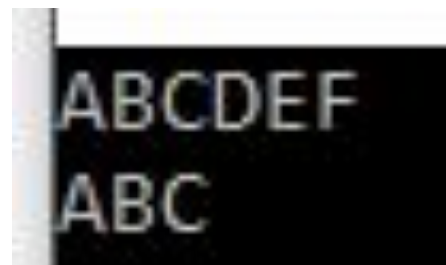
Класс StringBuilder

- `StringBuilder(string str, int start, int len, int cap)`

Параметры **str**, **start**, **len** задают строку инициализации, **cap** - емкость объекта

Класс StringBuilder

```
var s1 = new StringBuilder("ABC");  
var s2 = new StringBuilder("DEF");  
var s3 = s2.Insert(0, s1.ToString());  
WriteLine(s3);  
s3.Remove(3, 3);  
WriteLine(s3);
```



```
ABCDEF  
ABC
```

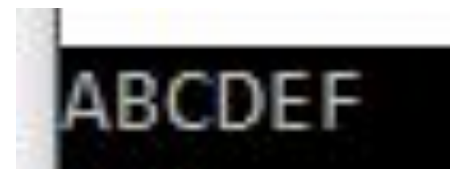
Класс StringBuilder

Операция конкатенации (+) не определена над строками класса `StringBuilder`, ее роль играет метод **Append**, дописывающий новую строку в хвост уже существующей

Класс StringBuilder

```
var s1 = new StringBuilder("ABC");  
var s2 = new StringBuilder("DEF");
```

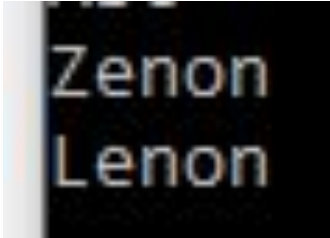
```
var s3 = s1;  
s3.Append(s2);  
WriteLine(s3);
```

A terminal window with a black background and white text. The text 'ABCDEF' is displayed in a monospaced font. The window has a light gray title bar at the top.

Выполнение этого кода изменит

Класс StringBuilder

```
var s4 = new StringBuilder("Zenon");  
WriteLine(s4);  
s4[0] = 'L';  
WriteLine(s4);
```



```
Zenon  
Lenon
```

Класс StringBuilder

- **Capacity** - возвращает или устанавливает текущую емкость буфера
- **MaxCapacity** - возвращает максимальную емкость буфера. Результат один и тот же для всех экземпляров класса
- **int EnsureCapacity(int capacity)** - если текущая емкость меньше, то она увеличивается до значения capacity, иначе не изменяется. Максимум текущей емкости и capacity возвращается в качестве результата работы метода.

Класс StringBuilder

```
var s1 = new StringBuilder(10, 100);  
WriteLine("s1: capacity = {0}, max = {1}",  
          s1.Capacity, s1.MaxCapacity);  
  
s1.Append("123");  
  
WriteLine("s1: capacity = {0}, max = {1}",  
          s1.Capacity, s1.MaxCapacity);
```

```
s1: capacity = 10, max = 100  
s1: capacity = 10, max = 100  
s2: capacity = 16, max = 2147483647  
s2: capacity = 20, max = 2147483647
```

Класс StringBuilder


```
var s2 = new StringBuilder("Hello");  
WriteLine("s2: capacity = {0}, max = {1}",  
          s2.Capacity, s2.MaxCapacity);  
  
s2.EnsureCapacity(20);  
  
WriteLine("s2: capacity = {0}, max = {1}",  
          s2.Capacity, s2.MaxCapacity);
```

```
s1: capacity = 10, max = 100  
s1: capacity = 10, max = 100  
s2: capacity = 16, max = 2147483647  
s2: capacity = 20, max = 2147483647
```


Класс StringBuilder

```
for (int i = 0; i < 1000; ++i) {  
    s1.Append("4");  
}
```

```
for (int i = 0; i < 1000; ++i) {  
    s1.Append("4");  
}
```

 **ArgumentOutOfRangeException was unhandled**

An unhandled exception of type 'System.ArgumentOutOfRangeException' occurred in mscorlib.dll

Additional information: емкость меньше текущего размера.

```
ReadKey();
```

LINQ и строки

Библиотека LINQ также может быть использована для обработки текста.

Это достигается за счет того, что строка текста может быть представлена в виде массива, что и позволяет использовать любые запросы по их обработке.

LINQ и строки


```
string aString = "ABCDE99F-J74-12-89A";
```

```
IEnumerable<char> stringQuery =  
    from ch in aString  
    where Char.IsDigit(ch)  
    select ch;
```

```
foreach (char c in stringQuery) {  
    Write(c + " ");  
}
```

LINQ и строки

```
int count = stringQuery.Count();  
WriteLine("Count = {0}", count);
```



```
9 9 7 4 1 2 8 9 Count = 8
```

LINQ и строки

```
string text = @"Historically, the world of data and the world of objects"  
+ @" have not been well integrated. Programmers work in C# or Visual Basic"  
+ @" and also in SQL or XQuery. On the one side are concepts such as classes,"  
+ @" objects, fields, inheritance, and .NET Framework APIs. On the other side"  
+ @" are tables, columns, rows, nodes, and separate languages for dealing with"  
+ @" them. Data types often require translation between the two worlds; there are"  
+ @" different standard functions. Because the object world has no notion of query, a"  
+ @" query can only be represented as a string without compile-time type checking or"  
+ @" IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to"  
+ @" objects in memory is often tedious and error-prone.";
```

LINQ и строки

Пример: подсчитать количество слов
“data” в тексте.

```
string searchTerm = "data";
```

```
string[] source = text.Split(  
    new char[] { '.', '?', '!', ' ', ';', ':', ',' },  
    StringSplitOptions.RemoveEmptyEntries);
```

LINQ и строки

```
var matchQuery =  
    from word in source  
    where word.ToLowerInvariant() ==  
           searchTerm.ToLowerInvariant()  
    select word;
```

LINQ и строки

```
int wordCount = matchQuery.Count();  
WriteLine("{0} occurrences(s) of the search" +  
    " term \"{1}\" were found.",  
    wordCount, searchTerm);
```

```
3 occurrences(s) of the search term "data" were found.
```


LINQ и строки

Пример: найти в тексте строку с заданными словами.

- **Distinct** – возвращает не повторяющиеся элементы последовательности
- **Intersect** – вычисляет пересечение двух массивов

LINQ и строки

```
string[] sentences = text.Split(  
    new char[] { '.', '?', '!' }  
);
```

```
string[] wordsToMatch = {  
    "Historically", "data", "integrated"  
};
```

LINQ и строки

```
var sentenceQuery =  
    from sentence in sentences  
    let w = sentence.Split(  
        new char[] {  
            '.', '?', '!', ' ', ';', ':', ','  
        },  
        StringSplitOptions.RemoveEmptyEntries)  
    where w.Distinct()  
        .Intersect(wordsToMatch)  
        .Count() == wordsToMatch.Count()  
    select sentence;
```

LINQ и строки

```
foreach (string str in sentenceQuery) {  
    WriteLine(str);  
}
```

Historically, the world of data and the world of objects have not been well integrated