

# Визуальное программи- рование

ЛЕКЦИЯ 3

# Содержание лекции

01 String

02 StringBuilder

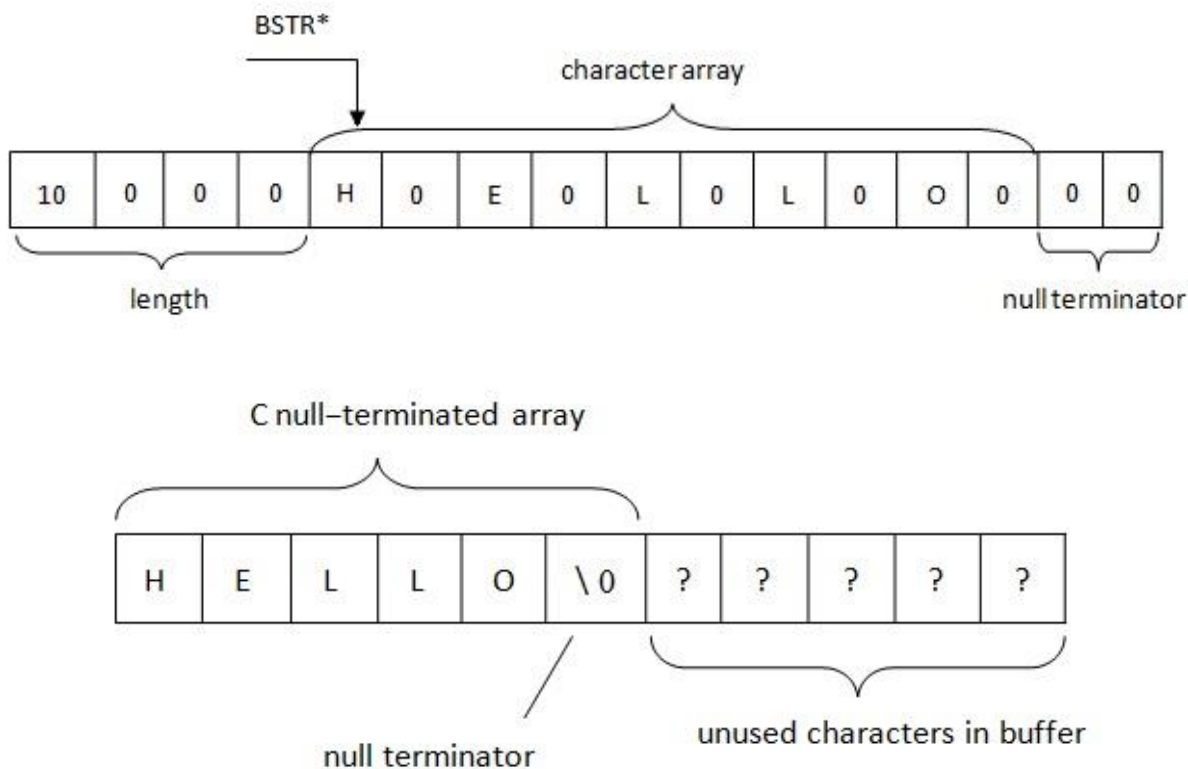
03 Регулярные  
выражения

# STRING

# Строки в .NET

В .NET строки располагаются согласно правилу **BSTR** (Basic string or binary string).

в C/C++ для представления строк используется **PWSZ** (Pointer to Wide-character String, Zero-terminated).



# Строки в .NET

- Длина строки ограничена неким числом, в отличие от PWSZ, где длина строки ограничена наличием свободной памяти.
- BSTR строка всегда указывает на первый символ в буфере. PWSZ может указывать на любой символ в буфере.
- У BSTR всегда в конце находится null символ, так же как и у PWSZ, но в отличие от последнего он является валидным символом и может встречаться в строке где угодно.
- За счет наличия null-символа в конце BSTR совместим с PWSZ, но не наоборот.

# Строки в .NET

Строки в .NET представляются в памяти согласно правилу BSTR. В буфере находится четырехбайтовая длина строки, за которой следуют двухбайтовые символы строки в формате UTF-16, за которыми следует два нулевых байта (\u0000).

Использование такой реализации имеет ряд преимуществ:

- длину строки не нужно пересчитывать, она хранится в заголовке;
- строка может содержать null-символы, где угодно;
- адрес строки(pinned) можно без проблем передавать в неуправляемой код там, где ожидается WCHAR\*.

# Тип string

В отличие от языков программирования, в которых строка представляет собой массив символов, в C# строки являются **объектами**. Следовательно, тип string относится к числу **ссылочных**.

В .NET представляют текст как последовательность символов **Unicode**.

Максимальный размер объекта string может составлять в памяти **2 ГБ**, или около **1 миллиарда** символов.

```
string str = "Строка";  
char[] chararray = { 's', 't', 'r' };  
string str2 = new string(chararray);
```

# Тип string

- Содержимое объекта типа string не подлежит изменению. Таким образом, однажды созданную последовательность символов изменить нельзя.
- Если требуется строка в качестве разновидности уже имеющейся строки, то для этой цели следует создать новую строку, содержащую все необходимые изменения.
- Переменные ссылки на строки (т.е. объекты типа string) подлежат изменению, а следовательно, они могут ссылаться на другой объект. Но содержимое самого объекта типа string не меняется после его создания.



# Класс String

В классе System.String предоставляется набор методов для определения длины символьных данных, поиска подстроки в текущей строке, преобразования символов из верхнего регистра в нижний и наоборот, и т.д.

```
...public sealed class String : IEnumerable<char>, IEnumerable, ICloneable, IComparable, IComparable<String?>, IConvertible, IEquatable<String>
{
    ...public static readonly String Empty;

    ...public String(char* value);
    ...public String(char[] value);
    ...public String(ReadOnlySpan<char> value);
    ...public String(sbyte* value);

    ...public String(char c, int count);
    ...public String(char* value, int startIndex, int length);
    ...public String(char[] value, int startIndex, int length);
    ...public String(sbyte* value, int startIndex, int length);
    ...public String(sbyte* value, int startIndex, int length, Encoding enc);

    ...public char this[int index] { ... }
    ...public int Length { get; }

    ...public static int Compare(String? strA, int indexA, String? strB, int indexB, int length, bool ignoreCase);
    ...public static int Compare(String? strA, int indexA, String? strB, int indexB, int length, bool ignoreCase, CultureInfo? culture);
    ...public static int Compare(String? strA, int indexA, String? strB, int indexB, int length, CultureInfo? culture, CompareOptions options);
    ...public static int Compare(String? strA, int indexA, String? strB, int indexB, int length, StringComparison comparisonType);
    ...public static int Compare(String? strA, String? strB);
    ...public static int Compare(String? strA, String? strB, bool ignoreCase);
    ...public static int Compare(String? strA, String? strB, bool ignoreCase, CultureInfo? culture);
    ...public static int Compare(String? strA, String? strB, CultureInfo? culture, CompareOptions options);
    ...public static int Compare(String? strA, String? strB, StringComparison comparisonType);
    ...public static int Compare(String? strA, int indexA, String? strB, int indexB, int length);
    ...public static int CompareOrdinal(String? strA, int indexA, String? strB, int indexB, int length);
}
```

# Класс String

Поле Empty обозначает **пустую строку**, т.е. такую строку, которая не содержит символы. Этим оно отличается от пустой ссылки типа String, которая просто делается на несуществующий объект.

```
public static readonly String Empty;
```

# Класс String

В классе определен **индексатор**, который позволяет получить символ по указанному индексу. Индексация строк, как и массивов, начинается с нуля.

```
public char this[int index] { get; }
```

# Класс String

Свойство `Length` возвращает количество символов в строке.

```
// Сводка:  
//     Gets the number of characters in the current System.String object.  
//  
// Возврат:  
//     The number of characters in the current string.  
public int Length { get; }
```

# Класс String

В классе есть несколько  
конструкторов.

```
public String(char* value);  
public String(char[] value);  
public String(ReadOnlySpan<char> value);  
public String(sbyte* value);  
public String(char c, int count);  
public String(char* value, int startIndex, int length);  
public String(char[] value, int startIndex, int length);  
public String(sbyte* value, int startIndex, int length);  
public String(sbyte* value, int startIndex, int length, Encoding enc);
```

# Класс String

Оператор `==` служит для проверки двух символьных строк на равенство.

Когда оператор `==` применяется к ссылкам на объекты, он обычно проверяет, делаются ли обе ссылки на один и тот же объект. А когда оператор `==` применяется к ссылкам на объекты типа `String`, то на предмет равенства сравнивается содержимое самих строк.

```
public static bool operator ==(String? a, String? b);
```

# Класс String

Это же относится и к оператору `!=`.

Когда он применяется к ссылкам на объекты типа `String`, то на предмет неравенства сравнивается содержимое самих строк.

```
public static bool operator !=(String? a, String? b);
```

# Класс String

- Операторы `==` и `!=` просто сравнивают порядковые значения символов в строках, то есть без учета регистра и настроек культурной среды.
- Другие операторы отношения, в том числе `<` и `>=`, сравнивают ссылки на объекты типа `String` таким же образом, как и на объекты других типов. А для того чтобы проверить, является ли одна строка больше другой, следует вызвать метод `Compare()`, определенный в классе `String`.



# Сравнение строк

- Во-первых, сравнение может отражать обычаи и нормы отдельной культурной среды, которые зачастую представляют собой настройки культурной среды, вступающие в силу при выполнении программы. Это стандартное поведение некоторых методов сравнения.
- И во-вторых, сравнение может быть выполнено независимо от настроек культурной среды только по порядковым значениям символов, составляющих строку. При порядковом сравнении строки просто упорядочиваются на основании невидоизмененного значения каждого символа.

# Форматирование строк

<b>C / c</b>	Задаёт формат денежной единицы, указывает количество десятичных разрядов после запятой
<b>D / d</b>	Целочисленный формат, указывает минимальное количество цифр
<b>E / e</b>	Экспоненциальное представление числа, указывает количество десятичных разрядов после запятой
<b>F / f</b>	Формат дробных чисел с фиксированной точкой, указывает количество десятичных разрядов после запятой
<b>G / g</b>	Задаёт более короткий из двух форматов: F или E
<b>N / n</b>	Также задаёт формат дробных чисел с фиксированной точкой, определяет количество разрядов после запятой
<b>P / p</b>	Задаёт отображения знака процентов рядом с числом, указывает количество десятичных разрядов после запятой
<b>X / x</b>	Шестнадцатеричный формат числа

# Интернирование строк

- При использовании строк в C#, CLR выполняет **интернирование строк**. Это способ хранения одной копии любой строки.
- Среда CLR поддерживает таблицу - **пул интернирования**. Эта таблица содержит одну уникальную ссылку на каждую строку, которая либо объявлена, либо создана программно во время выполнения программы.
- .NET Framework предоставляет два метода для взаимодействия с пулом интернирования: `String.Intern()` и `String.IsInterned()`.

# STRINGBUILDER

# Соединение строк

Здесь мы определяем две строки, а затем объединяем их с помощью оператора `+` и присваиваем результат `stringA`.

Поскольку эти строки являются неизменяемыми, необходимо создать новую строку для хранения этой объединенной информации.

Оператор `+` вызывает статический метод `Concat` и выделяет для всего этого новую строку. Поэтому переопределяя `stringA`, мы просто обновляем ссылку, на которую указывает эта локальная переменная и получаем доступ к только что созданной, новой строке.

```
var stringA = "Hello";  
var stringB = " world!";  
stringA = stringA + stringB;
```

# Почему это плохо?

В масштабе это может легко привести к большому количеству пауз в работе сборщика мусора, а это не есть хорошо. На макроуровне это настоящая уязвимость, которая создает возможность как минимум для нестабильной (или замедленной) работы приложения, как максимум — для довольно болезненной DDoS-атаки.

# Класс StringBuilder

**StringBuilder** поможет избежать выделения строк, для этого используя свой собственный внутренний буфер символов, что обеспечит эффективное управление их последовательностями.

Он делает модификацию и конкатенацию гораздо более оптимальным образом, позволяя манипулировать символами с меньшим объемом памяти.

```
public sealed class StringBuilder : ISerializable
{
    ...public StringBuilder();
    ...public StringBuilder(int capacity);
    ...public StringBuilder(string? value);
    ...public StringBuilder(int capacity, int maxCapacity);
    ...public StringBuilder(string? value, int capacity);
    ...public StringBuilder(string? value, int startIndex, int length, int capacity);

    ...public char this[int index] { get; set; }
    ...public int Capacity { get; set; }
    ...public int Length { get; set; }
    ...public int MaxCapacity { get; }

    ...public StringBuilder Append(float value);
    ...public StringBuilder Append(string? value);
    ...public StringBuilder Append(string? value, int startIndex, int count);
    ...public StringBuilder Append(ulong value);
    ...public StringBuilder Append(StringBuilder? value, int startIndex, int count);
    ...public StringBuilder Append(ushort value);
    ...public StringBuilder Append(uint value);
    ...public StringBuilder Append(StringBuilder? value);
    ...public StringBuilder Append(ReadOnlySpan<char> value);
    ...public StringBuilder Append(sbyte value);
    ...public StringBuilder Append(object? value);
    ...public StringBuilder Append(long value);
    ...public StringBuilder Append(int value);
    ...public StringBuilder Append(short value);
    ...public StringBuilder Append(double value);
    ...public StringBuilder Append(decimal value);
    ...public StringBuilder Append(char[]? value, int startIndex, int charCount);
}
```

# Класс StringBuilder

```
const string testString = "test string";
var output = string.Empty;
var iterations = int.Parse(Console.ReadLine() ?? "0");
for (var i = 0; i < iterations; i++)
{
    output += testString;
}
```

```
const string testString = "test string";
var iterations = int.Parse(Console.ReadLine() ?? "0");

var str = new StringBuilder();
for (var i = 0; i < iterations; i++)
{
    str.Append(testString);
}
var output = str.ToString();
```



# Класс StringBuilder

- Такой подход не приводит к выделению новой строки на каждой итерации. Вместо этого внутренний буфер «расширяется» по мере добавления новых символов.
- Это существенная экономия для тривиального изменения кода. Кроме экономии, также повышается стабильность кода. В этом и заключается смысл StringBuilder.

# Когда использовать

Класс `StringBuilder` рекомендуют использовать в двух случаях:

- Когда нам неизвестно количество операций и изменений над строками, которые появятся по ходу выполнения программы.
- Когда мы заведомо знаем, что приложению придется сделать множество подобных операций.

# Класс StringBuilder

- **Length** — возвращает количество символов в строке.
- **Capacity** — извлекает или задает количество символов, которое StringBuilder может хранить в буфере.
- **MaxCapacity** — возвращает максимальную емкость.
- **[index]** — индексатор, возвращающий символ в указанной позиции.

```
public char this[int index] { get; }  
public int Capacity { get; set; }  
public int Length { get; set; }  
public int MaxCapacity { get; }
```

# Класс StringBuilder

В классе есть несколько конструкторов, которые позволяют указать емкость (начальную и максимальную) или передать строку.

```
public StringBuilder();  
public StringBuilder(int capacity);  
public StringBuilder(string? value);  
public StringBuilder(int capacity, int maxCapacity);  
public StringBuilder(string? value, int capacity);  
public StringBuilder(string? value, int startIndex, int length, int capacity);
```

# Методы StringBuilder

Append() — добавляет строку после последнего символа текущей строки.

```
public StringBuilder Append(float value);
public StringBuilder Append(string? value);
public StringBuilder Append(string? value, int startIndex, int count);
public StringBuilder Append(ulong value);
public StringBuilder Append(StringBuilder? value, int startIndex, int count);
public StringBuilder Append(ushort value);
public StringBuilder Append(uint value);
public StringBuilder Append(StringBuilder? value);
public StringBuilder Append(ReadOnlySpan<char> value);
public StringBuilder Append(sbyte value);
public StringBuilder Append(object? value);
public StringBuilder Append(long value);
public StringBuilder Append(int value);
public StringBuilder Append(short value);
public StringBuilder Append(double value);
public StringBuilder Append(decimal value);
public StringBuilder Append(char[]? value, int startIndex, int charCount);
public StringBuilder Append(char[]? value);
public StringBuilder Append(char value, int repeatCount);
public StringBuilder Append(char* value, int valueCount);
public StringBuilder Append(bool value);
```

# Методы StringBuilder

Insert() — вставляет подстроку в заданной позиции в текущую строку.

```
public StringBuilder Insert(int index, sbyte value);
public StringBuilder Insert(int index, ulong value);
public StringBuilder Insert(int index, uint value);
public StringBuilder Insert(int index, ushort value);
public StringBuilder Insert(int index, string? value, int count);
public StringBuilder Insert(int index, string? value);
public StringBuilder Insert(int index, float value);
public StringBuilder Insert(int index, object? value);
public StringBuilder Insert(int index, long value);
public StringBuilder Insert(int index, int value);
public StringBuilder Insert(int index, short value);
public StringBuilder Insert(int index, double value);
public StringBuilder Insert(int index, decimal value);
public StringBuilder Insert(int index, char[]? value, int startIndex, int charCount);
public StringBuilder Insert(int index, char[]? value);
public StringBuilder Insert(int index, bool value);
public StringBuilder Insert(int index, ReadOnlySpan<char> value);
public StringBuilder Insert(int index, byte value);
public StringBuilder Insert(int index, char value);
```

# Методы StringBuilder

Replace() — заменяет все вхождения заданной подстроки или символа новой подстрокой или символом в текущей строке.

```
public StringBuilder Replace(char oldChar, char newChar, int startIndex, int count);  
public StringBuilder Replace(string oldValue, string? newValue, int startIndex, int count);  
public StringBuilder Replace(string oldValue, string? newValue);  
public StringBuilder Replace(char oldChar, char newChar);
```

# Методы StringBuilder

Remove() — удаляет указанные символы из текущей строки.

```
public StringBuilder Remove(int startIndex, int length);
```



# Методы StringBuilder

Clear() — удаляет все символы из текущей строки.

```
public StringBuilder Clear();
```

ToString() — преобразует значение объекта в обычную строку.

```
public string ToString(int startIndex, int length);  
public override string ToString();
```

# Изменение емкости

```
StringBuilder strbuild = new StringBuilder("123456", 5);  
Console.Write(strbuild.Length); // 6  
Console.Write(strbuild.Capacity); // емкость равна длине: 6  
strbuild.Append("78901234567");  
Console.WriteLine(strbuild.Length); // 17  
Console.WriteLine(strbuild.Capacity); // емкость равна длине: 17  
Console.WriteLine(strbuild.ToString());
```

```
StringBuilder strbuild = new StringBuilder(5);  
Console.Write(strbuild.Capacity); // начальная емкость: 5  
strbuild.Append("123456");  
Console.Write(strbuild.Length); // 6  
Console.Write(strbuild.Capacity); // вместимость удвоена: 10  
Console.Write(strbuild.ToString());
```

# РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ



# Регулярные выражения

Язык регулярных выражений предназначен специально для обработки строк. Он включает два средства:

- набор управляющих кодов для идентификации специфических типов символов;
- система для группирования частей подстрок и промежуточных результатов таких действий.

# Для чего они нужны

- поиск определенных шаблонов символов;
- проверка текста на соответствие predetermined шаблону (например, адресу электронной почты);
- извлечение, изменение, замена или удаление текстовых подстрок;
- добавление извлеченных строк в коллекцию для создания отчета.

Метасимвол	Значение
[...]	Любой из символов, указанных в скобках
[^...]	Любой из символов, не указанных в скобках
.	Любой символ, кроме перевода строки или другого разделителя Unicode-строки
\w	Любой текстовый символ, не являющийся пробелом, символом табуляции и т.п.
\W	Любой символ, не являющийся текстовым символом
\s	Любой пробельный символ из набора Unicode
\S	Любой непробельный символ из набора Unicode. Обратите внимание, что символы \w и \S - это не одно и то же
\d	Любые ASCII-цифры. Эквивалентно [0-9]
\D	Любой символ, отличный от ASCII-цифр. Эквивалентно [^0-9]
{n,m}	Соответствует предшествующему шаблону, повторенному не менее n и не более m раз
{n,}	Соответствует предшествующему шаблону, повторенному n или более раз
{n}	Соответствует в точности n экземплярам предшествующего шаблона
?	Соответствует нулю или одному экземпляру предшествующего шаблона; предшествующий шаблон является необязательным
+	Соответствует одному или более экземплярам предшествующего шаблона
*	Соответствует нулю или более экземплярам предшествующего шаблона

^	Соответствует началу строкового выражения или началу строки при многострочном поиске.	^Hello	"Hello, world", но не "Ok, Hello world" т.к. в этой строке слово "Hello" находится не в начале
\$	Соответствует концу строкового выражения или концу строки при многострочном поиске.	Hello\$	"World, Hello"
\b	Соответствует границе слова, т.е. соответствует позиции между символом \w и символом \W или между символом \w и началом или концом строки.	\b(my)\b	В строке "Hello my world" выберет слово "my"
\B	Соответствует позиции, не являющейся границей слов.	\B(ld)\b	Соответствие найдется в слове "World", но не в слове "ld"
	Соответствует либо подвыражению слева, либо подвыражению справа (аналог логической операции ИЛИ).		
(...)	Группировка. Группирует элементы в единое целое, которое может использоваться с символами *, +, ?,   и т.п. Также запоминает символы, соответствующие этой группе для использования в последующих ссылках.		
(?:...)	Только группировка. Группирует элементы в единое целое, но не запоминает символы, соответствующие этой группе.		



# Класс Regex

```
.public static string Escape(string str);
.public static bool IsMatch(string input, string pattern);
.public static bool IsMatch(string input, string pattern, RegexOptions options);
.public static bool IsMatch(string input, string pattern, RegexOptions options, TimeSpan matchTimeout);
.public static Match Match(string input, string pattern);
.public static Match Match(string input, string pattern, RegexOptions options);
.public static Match Match(string input, string pattern, RegexOptions options, TimeSpan matchTimeout);
.public static MatchCollection Matches(string input, string pattern, RegexOptions options, TimeSpan matchTimeout);
.public static MatchCollection Matches(string input, string pattern, RegexOptions options);
.public static MatchCollection Matches(string input, string pattern);
.public static string Replace(string input, string pattern, string replacement, RegexOptions options, TimeSpan matchTimeout);
.public static string Replace(string input, string pattern, MatchEvaluator evaluator, RegexOptions options, TimeSpan matchTimeout);
.public static string Replace(string input, string pattern, string replacement, RegexOptions options);
.public static string Replace(string input, string pattern, MatchEvaluator evaluator, RegexOptions options);
.public static string Replace(string input, string pattern, string replacement);
.public static string Replace(string input, string pattern, MatchEvaluator evaluator);
.public static string[] Split(string input, string pattern, RegexOptions options, TimeSpan matchTimeout);
.public static string[] Split(string input, string pattern, RegexOptions options);
.public static string[] Split(string input, string pattern);
.public static string Unescape(string str);
.protected internal static void ValidateMatchTimeout(TimeSpan matchTimeout);
.public string[] GetGroupNames();
.public int[] GetGroupNumbers();
.public string GroupNameFromNumber(int i);
.public int GroupNumberFromName(string name);
.public bool IsMatch(string input);
```

```
// Создаем шаблон для поиска групп с номером, начинающимся с 9
string pattern = @"^b[9]\d+";
// Создаем экземпляр Regex
Regex rg = new Regex(pattern);
// Передаем исходную строку
string groups = "В конкурсе приняли участие студенты групп 940, 040, 945, 946 и 0714";
// Получаем все совпадения
MatchCollection matchedGroups = rg.Matches(groups);
/// Выводим все номера групп
for (int count = 0; count < matchedGroups.Count; count++)
    Console.WriteLine(matchedGroups[count].Value);
```

```
940
945
946
```

```
string[] source = {  
    "Алясов и Дербуков пошли в буфет", "алясов съел сосиску в тесте", "дЕрБуКоВ допил чай аляСоВа"};  
Regex regex = new Regex("Дербуков");  
Console.WriteLine("Регистрозависимый поиск: ");  
foreach (string str in source)  
{  
    if (regex.IsMatch(str))  
        Console.WriteLine($"В исходной строке: \"{str}\" есть совпадения!");  
}  
Console.WriteLine();  
  
regex = new Regex("алясов", RegexOptions.IgnoreCase);  
Console.WriteLine("Регистронезависимый поиск: ");  
foreach (string str in source)  
{  
    if (regex.IsMatch(str))  
        Console.WriteLine($"В исходной строке: \"{str}\" есть совпадения!");  
}
```

Регистрозависимый поиск:

В исходной строке: "Алясов и Дербуков пошли в буфет" есть совпадения!

Регистронезависимый поиск:

В исходной строке: "Алясов и Дербуков пошли в буфет" есть совпадения!

В исходной строке: "алясов съел сосиску в тесте" есть совпадения!

В исходной строке: "дЕрБуКоВ допил чай аляСоВа" есть совпадения!

# ВОПРОСЫ ПО ЛЕКЦИИ