

# Строки, символы и регулярные выражения

Лекция №6

## Символы.

Тип `char` — это 16-разрядный тип данных без знака.

Для представления символов в C# используется *Unicode*, стандарт кодировки символов, позволяющий представлять алфавиты всех существующих в мире языков, а также любые знаки, используемые в мире.

Стандартный набор символов ASCII составляет подмножество Unicode.

Символьная константа может быть записана в одной из 4 форм:

- символ, имеющий графическое представление (кроме апострофа): ' **Е** ' ' **+** ' ' **ы** '
- управляющая последовательность: ' \n ' ' \' '
- символ, представленный шестнадцатеричным кодом: ' \x5B '
- символ в виде escape – последовательности Unicode: ' \u0021 '

В стандарте Unicode все символы разделены на категории:

DecimalNumber, LetterNumber и т.д.

Символьный тип соответствует стандартному классу Char библиотеки .NET из пространства имен System.

## Статические методы класса Char:

<b>GetNumericValue(ch)</b>	<p>Возвращает числовое значение символа <b>ch</b>, если он цифра, и -1 в противном случае.</p> <p>Возвращает результат типа <b>double</b>.</p> <p>Например: <b>Char.GetNumericValue('B');</b></p>
<b>GetUnicodeCategory(ch)</b>	<p>Возвращает <b>Unicode</b>- категорию символа <b>ch</b>.</p> <p>Например, <b>Char.GetUnicodeCategory('B')</b> возвращает <b>UppercaseLetter</b></p>
<b>Parse(s)</b>	<p>Преобразует строку <b>s</b>, состоящую из одного символа, в символ.</p> <p>Например: <b>Char.Parse("B")</b></p>

<b>IsControl(ch)</b>	Возвращает <b>true</b> , если символ <b>ch</b> является управляющим. Например: <b>Char.IsControl('\n')</b>
<b>IsDigit(ch)</b>	Возвращает <b>true</b> , если символ <b>ch</b> является десятичной цифрой. Например: <b>Char.IsDigit ('\n')</b>
<b>IsLetter(ch)</b>	Возвращает <b>true</b> , если символ <b>ch</b> является буквой. Например: <b>Char.IsLetter('F')</b>

<b>IsLetterOrDigit(ch)</b>	Возвращает <b>true</b> , если символ <b>ch</b> является буквой или цифрой.
<b>IsLower(ch)</b>	Возвращает <b>true</b> , если символ <b>ch</b> задан в нижнем регистре.
<b>IsNumber(ch)</b>	Возвращает <b>true</b> , если символ <b>ch</b> является числом (входит в одну из категорий: <b>DecimalDigitNumber</b> , <b>LetterNumber</b> , или <b>OtherNumber</b> ). Например, <code>Char.IsNumber("\u0660')</code>

<b>IsPunctuation(ch)</b>	Возвращает <b>true</b> , если символ <b>ch</b> является знаком препинания.
<b>IsSeparator(ch)</b>	Возвращает <b>true</b> , если символ <b>ch</b> является разделителем.
<b>IsUpper(ch)</b>	Возвращает <b>true</b> , если символ <b>ch</b> задан в верхнем регистре.
<b>IsWhiteSpace(ch)</b>	Возвращает <b>true</b> , если символ <b>ch</b> является пробелом, символом перевода строки, возврата каретки, табуляции и др.

<b>ToUpper(ch)</b>	Преобразует символ ch в верхний регистр.
<b>ToLower(ch)</b>	Преобразует символ ch в нижний регистр.
<b>ToString(ch)</b>	Преобразует символ ch к строковому типу.

Свойства:

<b>MaxValue, MinValue</b>	<b>Возвращает символы с максимальным и минимальным кодами (не имеют видимого представления).</b>
---------------------------	--



Для того чтобы узнать код символа можно использовать явное преобразование к целому типу:

**(int) a**

### Строки типа string.

Типу `string` соответствует базовый класс `System.String` библиотеки `.NET`, предназначенный для работы со строками символов в кодировке `Unicode`.

В классе `String` определено несколько конструкторов, которые позволяют создавать строки различными способами.

Строковый литерал создает строковый объект автоматически. Поэтому строковый объект часто инициализируется присваиванием ему строковой константы.

```
string <имя> = <строковая константа>;
```

Например:

```
string S = "Сегодня кина не будет!!!";
```

Еще один способ:

```
string <имя> = new string(<символ>,  
                           <количество повторений символа>);
```

Например, **string S1 = new string('ы', 10);**

Создать строку можно из символьного массива:

```
string <имя>=new string(<массив символов>);
```

Например,

```
char[ ] h = { 'B','a','a','y','!' };
```

```
string S2 = new string(h);
```

Еще один способ:

```
string <имя>=new string(<массив символов>,  
                        <нач. индекс>,<количество символов>);
```

Создаваемая таким образом строка будет состоять из *заданного количества* символов, взятых из массива, начиная с символа, индекс которого задан вторым параметром.

Например,

```
string S3 = new string(h, 1, 3);
```

Обратиться к символу строки можно по индексу, как к элементу массива, нумерация начинается с нуля:

S[2], S3[ j ].

Но так как в классе String индексатор определен только для чтения, изменять элементы строки используя обращение по индексу **нельзя**.

S3[0] = 'd'; - это **неправильно!!!**

Для определения количества символов в строке используется свойство **Length** класса **String**.

Например, `S3.Length` - длина строки `S3`.

### *Конкатенация строк*

Можно для этого использовать оператор "+". Например:

`S1+" Вася "`

или один из вариантов статического метода **Concat( )**.

Параметрами этого метода могут быть либо строки, либо данные типа `object`, причем метод принимает произвольное число аргументов. Если аргумент имеет тип `object`, то в конкатенации участвует текстовое представление объекта (результат метода `ToString( )`).

Например: `SS = string.Concat(10, 20.5, "Привет");`

## Сравнение строк

В классе String реализована перегрузка двух операторов:

`==` и `!=`.

При использовании этих операций учитывается регистр и пробелы в начале и в конце строки.

Например:

```
string ss1 = "вася"; string ss2 = "  вася";  
if (ss1 == ss2) Console.WriteLine(ss1+" равно "+ss2);  
else Console.WriteLine(ss1 + " не равно " + ss2);
```

Результат : вася не равно  вася

В классе String есть ряд методов сравнения строк.

Статический метод **Compare(s1, s2)** производит лингвистическое сравнение.

Возвращает положительное целое число, если строка s1 больше s2, отрицательное число, если s1 меньше s2, и нуль, если строки s1 и s2 равны. Тип результата **int**.

```
ss1 = "вася "; ss2 = "ВАСЯ";
```

```
Console.WriteLine(string.Compare(ss1, ss2));
```

Результат **1**

```
ss1 = "вася"; ss2 = "ВАСЯ";
```

```
Console.WriteLine(string.Compare(ss1, ss2));
```

Результат **-1**

## Compare(s1, s2, u)

Если логический параметр *u* равен значению **true**, при сравнении не учитываются различия между прописным и строчным вариантами букв. В противном случае эти различия учитываются. В остальном работает также как и предыдущий вариант.

```
ss1 = "вася"; ss2 = "ВАСЯ";
```

```
Console.WriteLine(string.Compare(ss1, ss2,true));
```

Результат **0**



## **Compare(s1, i1, s2, i2, n, u)**

Сравнивает части строк **s1** и **s2** из **n** символов начиная со строковых элементов **s1[i1]** и **s2[i2]**.

Метод возвращает положительное число, если часть строки **s1** больше части строки **s2**, отрицательное число, если часть строки **s1** меньше части строки **s2**, и нуль, если сравниваемые части строк равны.

Назначение параметра **u** такое же как в предыдущем варианте метода.

## Статический метод **CompareOrdinal (s1, s2)**

Сравнивает строку **s1** со строкой **s2**, независимо от языка, диалекта или территориального образования.

Возвращает положительное целое число, если строка **s1** больше **s2**, отрицательное число, если **s1** меньше **s2**, и нуль, если строки **s1** и **s2** равны. Тип результата **int**. Длина не учитывается.

Сравнение осуществляется по числовым кодам символов, входящих в каждую строку. Поэтому 'f' > 'F'.

## **CompareOrdinal (s1, i1, s2, i2, n)**

Сравнивает части строк **s1** и **s2** из **n** символов начиная со строковых элементов **s1[i1]** и **s2[i2]**.

## Нестатический метод **CompareTo(s)**

Возвращает положительное целое число, если вызывающая строка больше строки **s**, отрицательное число, если вызывающая строка меньше **s**, и нуль, если строки равны. Тип результата **int**.

```
ss1 = "  вася"; ss2 = "ВАСЯ";
```

```
    Console.WriteLine(ss1.CompareTo(ss2));
```

Результат **-1**

```
ss1 = "вася  "; ss2 = "ВАСЯ";
```

```
    Console.WriteLine(ss1.CompareTo(ss2));
```

Результат **1**

Нестатический метод **Equals (s)**

Возвращает true, если вызывающая строка равна s.

Например, **ss1.Equals(ss2)**

Статический метод **Equals (s1,s2)**

Возвращает true, если строка s1 равна s2.

Например, **string.Equals(ss1,ss2)**

### *Поиск в строке*

Нестатический метод **IndexOf(s)**

возвращает индекс первого вхождения символа или строки s в вызывающей строке. Если символа нет, возвращает -1.

Результат **ss1.IndexOf('B') 1**

Результат `ss1.IndexOf("CD")` 2

Нестатический метод `LastIndexOf(s)`

возвращает индекс последнего вхождения символа или строки `s` в вызывающей строке. Если символа нет, возвращает -1.

Результат `ss1.IndexOf('B')` 1

Нестатические методы

`IndexOfAny( a)`

`LastIndexOfAny( a)`

, где `a` – массив символов,

возвращают индекс первого(последнего) вхождения любого символа из массива `a`, который обнаружится в вызывающей строке.

## Нестатический метод **StartsWith (s)**

возвращает значение **true**, если вызывающая строка начинается с подстроки **s**, и значение **false** в противном случае.

Нестатический метод **EndsWith (s)** возвращает значение **true**, если вызывающая строка оканчивается подстрокой **s**.

Есть и другие варианты этих методов (см. в литературе).

## *Обработка строк.*

### Нестатический метод **Split(r)**

разбивает вызывающую строку на подстроки, которые возвращаются методом в виде строкового массива `string[ ]`.

**r** - это массив символов, содержащий разделители подстрок.

Если параметр *r* отсутствует, в качестве разделителя подстрок используется пробел.

### Нестатический метод **Split(r, n)**

разбивает вызывающую строку на подстроки, которые возвращаются методом в виде строкового массива из **n** подстрок.

**r** - это массив символов, содержащий разделители подстрок.

Пример из справочника Шилдта.

```
string str =
```

```
"Какое слово ты скажешь, такое в ответ и услышишь.";
```

```
char[ ] seps = {',', ' ', ' '};
```

```
string[ ] parts = str.Split(seps);
```

```
Console.WriteLine("Результат разбиения строки на части: " );
```

```
for(int i=0; i < parts.Length; i++) Console.WriteLine(parts[i]);
```

Можно `str.Split( );` или `str.Split(seps, 3);`



## Статический метод **Join(s, ss)**

возвращает строку, которая содержит объединенные строки, переданные в массиве строк **ss**, разделенные строкой **s**.

Например

```
string S = String.Join("ух", parts);
```

Результат:

Какое ух слово ух ты ух скажешь ух ух такое ух в ух ответ ух и ух услышишь ух

## Статический метод **Join(s, ss, i, n)**

возвращает строку, которая содержит **n** объединенных строк, переданных в массиве строк **ss**, начиная с **i**-й, разделенных строкой **s**.

## Нестатический метод **Trim( )**

возвращает строку с удаленными из вызывающей строки начальными и конечными пробелами.

Например, `ss1 = ss1.Trim( );`

## Нестатический метод **TrimEnd( )**

возвращает строку с удаленными из вызывающей строки конечными пробелами.

## Нестатический метод **TrimStart( )**

возвращает строку с удаленными из вызывающей строки начальными пробелами.

## Нестатический метод **PadLeft(n)**

возвращает строку с добавленными в вызывающую строку начальными пробелами в таком количестве, чтобы общая длина результирующей строки стала равной **n**.

Например, `ss1 = "ffff"; ss1 = ss1.PadLeft(8);`

Будут добавлены слева 3 пробела.

## Нестатический метод **PadRight(n)**

возвращает строку с добавленными справа в вызывающую строку пробелами в таком количестве, чтобы общая длина результирующей строки стала равной **n**.

Есть еще варианты этих методов:

**Trim(ch)**

**PadLeft(n, ch)**

**PadRight(n, ch)**

## Нестатический метод **Insert( i, s)**

возвращает строку, которая является результатом вставки строки **s** в вызывающую строку, начиная с **i**-й позиции.

Например,

```
ss1 = "Иванов двоечник!"; ss1 = ss1.Insert(6, " не");
```

## Нестатический метод **Remove( i, n)**

возвращает строку, которая является результатом удаления из вызывающей строки **n** символов, начиная с **i**-го.

```
ss1 = ss1.Remove(6, 3);
```

## Нестатический метод **Replace( s1, s2)**

возвращает строку, равную вызывающей, в которой все символы или строки s1 заменены на символ или строку s2.

Например,      **ss1 = ss1.Replace('o', 'a');**  
                  **ss1 = ss1.Replace("o", "fff");**

## Нестатический метод **Substring(i, n)**

возвращает подстроку вызывающей строки из n символов, начиная с i-й позиции.

**ToLower( )** возвращает строку из строчных букв

**ToUpper( )** возвращает строку из прописных букв

## *Форматирование строк.*

Статический метод **Format(s,v1,v2,...,vn)**

Форматирует объекты **v1,v2,...,vn** согласно соответствующим командам форматирования, содержащимся в строке **s**.  
Возвращает копию строки **s**, в которой команды форматирования заменены форматированными данными

```
string ss3=
```

```
string.Format(" Стипендия: {0,3:d}$ Средний балл: {1,5:f1}", 200, 8.68);
```

Результат:

Стипендия: 200\$ Средний балл: 8,7

## Нестатический метод **ToString(fmt)**

возвращает строковое представление вызывающего объекта в соответствии с заданным спецификатором формата, переданным в параметре **fmt**.

Например, **3.99.ToString("f5")**                      **x.ToString("c")**

## Строки *thru* **StringBuilder**

Класс **StringBuilder** определен в пространстве имен **System.Text**.

Существуют различные способы создания строки:

```
StringBuilder <имя строки> = new StringBuilder( );
```

```
StringBuilder z = new StringBuilder( );
```

В этом случае создается пустая строка с объемом памяти 16 байт. Длина ее при этом равна нулю.

```
StringBuilder <имя строки> =  
new StringBuilder(<объем памяти> );
```

```
StringBuilder z = new StringBuilder(50);
```

```
StringBuilder <имя строки> =  
new StringBuilder(<инициализирующая строка> );
```

```
StringBuilder z = new StringBuilder("Привет");
```





Обратиться к символу строки можно по индексу: `z[6]`.

В классе **StringBuilder** можно изменять элементы строки используя обращение по индексу.

Например,

```
z[3] = 'D';
```

*Свойства:*

**Length** используется для определения количества символов в строке класса.

Например, `z.Length` - длина строки `z`.

**MaxCapacity** максимальный объем памяти. Только для чтения.

**Capacity** используется для получения и установки объема памяти, отводимой под строку.

При установке значения этого свойства меньше текущей длины строки или больше максимального значения генерируется исключение **ArgumentOutOfRangeException**.

### *Методы:*

Нестатический метод **Append** используется для добавления в конец строки: перегружен 18 раз.

**Append(s)** – добавление объекта *s* в конец вызывающей строки.

*s* – это может быть число, символ, строка, массив символов, **true** или **false**.

```
z.Append("ggg"); z.Append(2.5); z.Append('s');
```

```
char[ ] h = { 'B','a','a','y','!' }; z.Append(h);
```

**Append(s, n)** – добавление символа s в конец вызывающей строки n раз.

```
z.Append('f',3);
```

**Append(s, i, n)** – добавление в конец вызывающей строки части строки или массива символов s из n символов, начиная с i-го.

```
z.Append("fff",1,2);      z.Append(h,1,2);
```

Нестатический метод **Insert(i, s)**

вставляет объект s в вызывающую строку, начиная с i-й позиции.

s – это может быть число, символ, строка, массив символов, **true** или **false**.

Например, z.Insert(1,"hhh"); z.Insert(1,'m'); z.Insert(1,2.4);

Методы **Replace** и **Remove** аналогичны одноименным методам класса `String`, только они не создают новую строку, а изменяют вызывающую.

**AppendFormat(s,v1,v2,...,vn)** добавляет к вызывающей строке строку `s`, в которой команды форматирования заменены форматированными данными `v1,v2,...,vn`.

**z.AppendFormat(" добавили {0,5:f2} и еще {1}", 2.5, 'v');**

**AppendLine(s)** добавляет к вызывающей строке строку `s` и символ конца строки.

**z.AppendLine("fff"); Console.Write(z);**

Нестатический метод **ToString()** преобразует вызывающую строку к типу **string**.

```
string sss = z.ToString();
```

Нестатический метод **ToString(i,n)** преобразует подстроку вызывающей строки из n символов, начиная с i-го, к типу **string**.

```
string sss1 = z.ToString(2, 10);
```

## Регулярные выражения

**Регулярные выражения** - это особым образом отформатированные строки, используемые для поиска шаблонов в тексте, для подтверждения корректности формата данных, редактирования строк и т.д.

Для использования регулярных выражений в библиотеке .NET есть классы, которые объединены в пространство имен **System.Text.RegularExpressions**.

Это пространство имен не добавляется автоматически в список доступных пространств имен, поэтому в программе нужно вручную набрать оператор

```
using System.Text.RegularExpressions;
```

Регулярное выражение состоит из символов двух видов: *обычных символов*, представляющих в выражении сами себя, и *метасимволов*.

Метасимвол - это специальный символ, который служит обозначением класса символов (например, цифры или буквы), уточняет позицию поиска или задает количество повторений символов в выражении.

### **Классы символов:**

**. (точка)** – любой символ, кроме `\n`.

Например, выражение **"К.т"** соответствует фрагментам строк **Кот, Кит, К&т** и т.д.

Чтобы точка воспринималась непосредственно (не как метасимвол), нужно перед ней ставить `\`.



**[последовательность символов]**

**ИЛИ**

**[диапазон символов]** - любой одиночный символ из последовательности (или диапазона) внутри скобок.

Например, выражение **"К[иоэ-я]т"** соответствует фрагментам строк

**Кот, Кит, Кэт, Кят и Кют.**

**[^последовательность символов]**

**ИЛИ**

**[^диапазон символов]** - любой одиночный символ, не входящий в последовательность (или диапазон символов) внутри скобок.

Например, выражение **"К[^иоэ-я]т"** соответствует  
фрагментам строк

**Кft, К&т**

и не соответствует **Кот, Кит, Кэт, Кят и Кют.**

**\w** - любая буква или цифра.

Например, выражение **"К\\вт"** или **@"К\\вт"** соответствует  
фрагментам строк

**Кft, К7т, Кот, К\_т**

и не соответствует **К:т, К%т** и т.д.

**\W** - любой символ, не являющийся буквой или цифрой.

Например, выражение **"K\\Wт"** или **@ "K\\Wт"** соответствует фрагментам строк

**K:т, K%т, K т, K\nt**

и не соответствует **Kft, K7т, Kот, K\_т** и т.д.

**\s** - любой пробельный символ (пробел, \n, \t, \v, \f, \r).

Например, выражение **"K\\st"** или **@ "K\\st"** соответствует фрагментам строк

**K\тт, K\vт, K т, K\nt**

**\S** - любой не пробельный символ.

Например, выражение **"K\Sт"** или **@ "K\Sт"** соответствует фрагментам строк

**KRт, Kот, K7т, K\bт**

**\d** - любая десятичная цифра.

Например, выражение **"K\dт"** или **@ "K\dт"** соответствует фрагментам строк

**K0т, K3т.**

**\D** - любой символ, не являющийся цифрой.

**Уточняющие метасимволы (мнимые – им никакой символ в тексте не соответствует ):**

**^** - фрагмент, соответствующий регулярному выражению, должен располагаться только в начале строки.

Например, в строках " **Кот Кит** " и " **Кыт Кит** " нет фрагмента, соответствующего регулярному выражению "**^К[ои]т**"

**\$** - фрагмент, соответствующий регулярному выражению, должен располагаться только в конце строки.

Этот метасимвол нужно располагать в конце регулярного выражения: **@ "К[ои]т\$"**

**\b** - фрагмент, соответствующий регулярному выражению, должен располагаться на границе слова.

Например, в строке " **fКит Котd**" фрагментом, соответствующим регулярному выражению **@"\bК[ои]т"** будет **Кот**.

Выражению **@"К[ои]т\b"** будет соответствовать **Кит**.

**\B** - фрагмент, соответствующий регулярному выражению, не должен располагаться на границе слова.

Какая именно граница имеется ввиду, зависит от расположения этого метасимвола в регулярном выражении.

## *Повторители:*

\* - ноль или больше повторений предыдущего элемента.

Например, фрагментом, соответствующим регулярному выражению "Ки\*т" будет **Кт, Кит, Киит** ит.д.

Выражению "К[ои]\*т" будут соответствовать

**Кт, Кит, Кот, Киоит, Кооит** и т.д.

+ - одно или больше повторений предыдущего элемента.

Выражению "К[ои]+т" будут соответствовать

**Кит, Кот, Киоит, Кооит** и т.д. , но не будет соответствовать **Кт,**

? – ни одного или одно повторение предыдущего элемента.

Выражению "К[ои]?т" будут соответствовать

**Кит, Кот, Кт.**

{n} – n повторений предыдущего элемента.

Выражению "Ки{3}т" будет соответствовать **Кииит.**

{n, } – n или больше повторений предыдущего элемента.

{n, m} – от n до m повторений предыдущего элемента.

Выражению "Ки{2,3}т" будут соответствовать **Киит и Кииит.**



В качестве повторяющегося элемента может использоваться группа символов, заключенная в скобки.

Например, "Тра(-ля){2}", "Тра(-л[яю]){3}"

В регулярных выражениях можно использовать конструкцию **выбора** из нескольких элементов. Варианты выбора перечисляются через вертикальную черту.

Например,

**"Василий|Петя|Ваня"**

В строке " Ваня Петя Василий" этому выражению будет соответствовать **Ваня**,

в строке "Василий Ваня Петя " - **Василий** .

Фрагмент текста, соответствующий фрагменту регулярного выражения, можно запомнить в некоторой переменной:

**(?<имя переменной>фрагмент выражения)**

**и использовать эту переменную внутри выражения.**

Например, `@"(?<s>\w\w).{2,}\k<s>"`

Чтобы использовать в программе регулярное выражение, нужно создать объект класса **Regex**:

**Regex <имя> = new Regex(<строка, задающая вид РВ>)**

Например,

```
string rv = @"(?<s>\w\w).{2,}\k<s>",
```

```
Regex R = new Regex(rv);
```

## Методы класса *Regex*

Нестатический **IsMatch(s)** возвращает true, если фрагмент, соответствующий вызывающему регулярному выражению в строке s найден, и false в противном случае.

```
string rv = "(Ва\\wя){3}", s1 = "ВасяВаняВаля";  
Regex R = new Regex(rv);
```

**R.IsMatch(s1)** будет true.

Нестатический **Match(s)** возвращает первый фрагмент, соответствующий вызывающему регулярному выражению в строке s .

Для примера выше **R.Match(s1)** вернет **Вася**.

Нестатический **Matches(s)** возвращает коллекцию фрагментов, соответствующих вызывающему регулярному выражению в строке *s* (объект класса *MatchCollection*).

Обратиться к каждому элементу коллекции можно по его номеру (нумерация с нуля), указав его в квадратных скобках.

Например: **R.Matches(s1)[0]**

Узнать количество найденных фрагментов можно с помощью свойства **Count** класса *MatchCollection*.

Например: **R.Matches(s1).Count**

Преобразовать элемент коллекции в строку можно с помощью метода **ToString**.

Например: **string s = R.Matches(s1)[0].ToString();**

Нестатический метод **Split(s)** разбивает строку s в соответствии с разделителями, заданными с помощью регулярного выражения и возвращает эти фрагменты в виде массива строк.

```
string rv = "[- ,.::;]+",
```

```
s1 = "Вася, Ваня::,Валя; Петя,- Муся.;; Буся.";
```

```
Regex R = new Regex(rv);
```

```
string[ ] s = R.Split(s1);
```

```
foreach (string ss in s) Console.WriteLine(ss);
```

Статический метод **Replace(s, srv, s1)** возвращает строку, которая является копией строки s, в которой все фрагменты, соответствующие регулярному выражению srv заменены на строку s1.

```
string stip = «Янчев - 200 тыс. руб. Кухаренко - 500000руб.»;  
string stip1 =
```

```
Regex.Replace(stip, "( тыс. руб.)|(000руб.)", "$");
```

**Янчев - 200\$ Кухаренко - 500\$**