

Оглавление

СТРОКИ.....	1
1. Строки C++.....	1
2. Строки C#.....	2
2.1. Класс char.....	2
2.2. Класс char[] - массив символов.....	6
2.3. Класс String.....	8
2.3.1. Объявление строк. Конструкторы класса string.....	8
2.3.2. Операции над строками.....	8
2.3.3. Строковые константы.....	9
2.3.4. Неизменяемый класс string.....	10
2.3.5. Класс StringBuilder - построитель строк.....	13
3. Пространство имен RegularExpression и классы регулярных выражений.....	16
3.1. Синтаксис регулярных выражений.....	16
3.2. Классы пространства RegularExpressions.....	17
3.2.1. Класс Regex.....	17
3.2.2. Классы Capture и CaptureCollection.....	18
3.2.3. Класс RegexCompilationInfo.....	18
3.3. Примеры работы с регулярными выражениями.....	19

СТРОКИ.

Когда говорят о строковом типе, то обычно различают тип, представляющий:

- отдельные символы (тип **char**);
- строки постоянной длины (часто представляются массивом символов);
- строки переменной длины - тип **string**, соответствующий современному представлению о строковом типе.

Символьный тип **char**, представляющий частный случай строк длиной 1, полезен во многих задачах. Основные *операции* над строками - это разбор и *сборка*. При их выполнении приходится, чаще всего, доходить до каждого символа строки.

В языке Паскаль, где был введен тип **char**, сам *строковый тип* рассматривался, как **char[]** - массив символов. При таком подходе получение *i*-го символа строки становится такой же простой операцией, как и получение *i*-го элемента массива. Следовательно, эффективно реализуются обычные *операции* над строками - *определение* вхождения одной строки в другую, *выделение подстроки*, замена символов строки.

Однако представление строки массивом символов хорошо только для *строк постоянной длины*. Массив не приспособлен к изменению его размеров, вставки или удалению символов (подстрок).

Наиболее часто используемым строковым типом является тип **string**, который задает строки переменной длины. Над этим типом допускаются *операции* поиска вхождения одной строки в другую, *операции* вставки, замены и удаления подстрок.

1. Строки C++

В языке C++ есть все виды строк.

Символьный тип **char** используется для задания отдельных символов.

Для *строк постоянной длины* можно использовать массив символов - **char[]**. Особенностью, характерной для языка C++ (точнее для языка C), является завершение строки символом с нулевым кодом. Строки, завершаемые нулем, называются обычно *строками C*. Массив **char[]** задает *строку C* и потому должен иметь размер, по крайней мере, на единицу больше фактического размера строки.

Пример объявления подобных строк в C++:

```
//Массивы и строки
char strM1[] = "Hello, World!";
char strM2[20] = "Yes";
```

Массив `strM1` состоит из 14 символов, массив `strM2` - из 20, но его четвертый символ имеет код 0, сигнализирующий о фактическом конце строки.

Другой способ задания *строк* `C`, заканчивающихся нулем, состоит в использовании *типизированного указателя* - `char*`.

```
//Строки, заданные указателем char*
char* strPM1="Hello, World!";
char* strPM2;
```

Два типа, `char[]` и `char*`, допускают взаимные преобразования.

Рассмотрим процедуру копирования строк, соответствующую духу и стилю C++:

```
void mycopy(char* p, const char* q)
{
    while(*p++ = *q++);
}
```

Эта процедура копирует содержимое строки `q` в строку `p`. В этой короткой программе, в которой, кроме условия цикла `while`, ничего больше нет, фактически используются многие средства языка C++ - разыменование указателей, *адресная арифметика*, присваивание как операция, завершение строки нулем, логическая интерпретация значений. Работа программы: вначале указатель `q` задает адрес начала строки, поэтому разыменование `*q` задает первый символ копируемой строки. Это значение присваивается первому символу строки `p`. Постфиксные операции `p++` и `q++` увеличивают значение указателей на единицу, но поскольку используется *адресная арифметика*, то в результате вычисляется адрес, задающий следующий символ соответствующих строк, и процесс копирования продолжается. При достижении последнего символа строки `q` - символа с кодом нуль - он также будет скопирован в строку `p`. Но в этот момент выражение присваивание впервые вернет в качестве значения результат 0, который будет проинтерпретирован в условии цикла `while` как `false`, и цикл завершит свою работу. Строка будет скопирована.

Можно восхищаться этой короткой и эффективной программой, можно ругать ее за сложность восприятия. Трудно назвать ее интуитивно понятной. Но во многом все определяется вкусом и привычкой.

Тип `string` не является частью языка C++, но входит в библиотеку, определяемую стандартом языка. Стандартные библиотеки, по сути, являются продолжением языка. Тип (класс) `string` обеспечивает работу со строками переменной длины и поддерживает многие полезные операции над строками.

2. Строки C#

2.1. Класс char

В C# есть *символьный класс* `Char`, основанный на классе `System.Char` и использующий двухбайтную кодировку Unicode представления символов. Для этого типа в языке определены символьные константы - символьные литералы. Константу можно задавать:

- символом, заключенным в одинарные кавычки;
- escape-последовательностью, задающей код символа;
- Unicode-последовательностью, задающей Unicode-код символа.

Примеры объявления символьных переменных и работы с ними:

```
public void TestChar()
{
    char ch1='A', ch2='\x5A', ch3='\u0058';
    char ch = new Char();
    int code; string s;
    ch = ch1;
    //преобразование символьного типа в тип int
    code = ch; ch1=(char) (code +1);
    //преобразование символьного типа в строку s = ch;
    s = ch1.ToString()+ch2.ToString()+ch3.ToString();
    Console.WriteLine("s= {0}, ch= {1}, code = {2}", s, ch, code);
} //TestChar
```

Три символьные переменные инициализированы константами, значения которых заданы тремя разными способами. Переменная `ch` объявляется в объектном стиле, используя `new` и вызов конструктора

класса. Тип **char**, как и все типы C#, является классом. Этот класс наследует свойства и методы класса **Object** и имеет большое число собственных методов.

Существуют ли преобразования между классом *char* и другими классами? Явные или неявные преобразования между классами **char** и **string** отсутствуют, но, благодаря методу **ToString**, переменные типа **char** стандартным образом преобразуются в тип **string**.

Существуют *неявные преобразования типа char* в целочисленные типы, начиная с типа **ushort**. Обратные преобразования целочисленных типов в тип **char** также существуют, но они уже явные.

В результате работы процедуры **TestChar** строка **s**, полученная сцеплением трех символов, преобразованных в строки, имеет значение **BSX**, переменная **ch** равна **A**, а ее код - переменная **code** - **65**.

Семантика присваивания справедлива при вызове методов и замене формальных аргументов на фактические. Пример - две процедуры, выполняющие взаимно-обратные операции - получение по коду символа и получение символа по его коду:

```
public int SayCode(char sym)
{
    return (sym);
} // SayCode

public char SaySym(object code)
{
    return ((char)((int)code));
} // SaySym
```

В первой процедуре преобразование к целому типу выполняется неявно. Во второй - преобразование явное. Ради универсальности она слегка усложнена. Формальный параметр имеет тип **Object**, что позволяет передавать ей в качестве аргумента код, заданный любым целочисленным типом. Платой за это является необходимость выполнять два явных преобразования.

Класс *Char* (как и все классы в C#) наследует свойства и методы родительского класса **Object**, но у него есть собственные методы и свойства (табл. 1.1). Большинство статических методов перегружены. Они могут применяться как к отдельному символу, так и к строке, для которой указывается номер символа для применения метода. Основную группу составляют методы **Is**, крайне полезные при разборе строки.

Табл. 1.1. Статические методы и свойства класса Char

Метод	Описание
GetNumericValue	Возвращает численное значение символа, если он является цифрой, и (-1) в противном случае
GetUnicodeCategory	Все символы разделены на категории. Метод возвращает Unicode категорию символа. Ниже приведен пример
IsControl	Возвращает true, если символ является управляющим
IsDigit	Возвращает true, если символ является десятичной цифрой
IsLetter	Возвращает true, если символ является буквой
IsLetterOrDigit	Возвращает true, если символ является буквой или цифрой
IsLower	Возвращает true, если символ задан в нижнем регистре
IsNumber	Возвращает true, если символ является числом (десятичной или шестнадцатеричной цифрой)
IsPunctuation	Возвращает true, если символ является знаком препинания
IsSeparator	Возвращает true, если символ является разделителем
IsSurrogate	Некоторые символы Unicode с кодом в интервале [0x10000, 0x10FFF] представляются двумя 16-битными "суррогатными" символами. Метод возвращает true, если символ является суррогатным
IsUpper	Возвращает true, если символ задан в верхнем регистре
IsWhiteSpace	Возвращает true, если символ является "белым пробелом". К белым пробелам, помимо пробела, относятся и другие символы, например, символ конца строки и символ перевода каретки
Parse	Преобразует строку в символ. Естественно, строка должна состоять из одного символа, иначе возникнет ошибка
ToLower	Приводит символ к нижнему регистру
ToUpper	Приводит символ к верхнему регистру
MaxValue, MinValue	Свойства, возвращающие символы с максимальным и минимальным кодом. Возвращаемые символы не имеют видимого образа

Примеры, в которых используются многие из перечисленных методов:

```

public void TestCharMethods()
{
    Console.WriteLine("Статические методы класса char:");
    char ch='a', ch1='1', lim=';', chc='\xA';
    double d1, d2;
    d1=char.GetNumericValue(ch); d2=char.GetNumericValue(ch1);
    Console.WriteLine("Метод GetNumericValue:");
    Console.WriteLine("sym 'a' - value {0}", d1);
    Console.WriteLine("sym '1' - value {0}", d2);
    System.Globalization.UnicodeCategory cat1, cat2;
        cat1 =char.GetUnicodeCategory(ch1);
        cat2 =char.GetUnicodeCategory(lim);
    Console.WriteLine("Метод GetUnicodeCategory:");
    Console.WriteLine("sym '1' - category {0}", cat1);
    Console.WriteLine("sym ';' - category {0}", cat2);
    Console.WriteLine("Метод IsControl:");
    Console.WriteLine("sym '\xA' - IsControl - {0}", char.IsControl(chc));
    Console.WriteLine("sym ';' - IsControl - {0}", char.IsControl(lim));
    Console.WriteLine("Метод IsSeparator:");
    Console.WriteLine("sym ' ' - IsSeparator - {0}", char.IsSeparator(' '));
    Console.WriteLine("sym ';' - IsSeparator - {0}", char.IsSeparator(lim));
    Console.WriteLine("Метод IsSurrogate:");
    Console.WriteLine("sym '\u10FF' - IsSurrogate - {0}", char.IsSurrogate('\u10FF'));
    Console.WriteLine("sym '\\' - IsSurrogate - {0}", char.IsSurrogate("\\"));
    string str = "\U00010F00";
    //Символы Unicode в интервале [0x10000,0x10FFF]
    //представляются двумя 16-битными суррогатными символами
    Console.WriteLine("str = {0}, str[0] = {1}", str, str[0]);
    Console.WriteLine("str[0] IsSurrogate - {0}", char.IsSurrogate(str, 0));
    Console.WriteLine("Метод IsWhiteSpace:");
    str="пробелы, пробелы!" + "\xD" + "\xA" + "Всюду пробелы!";
    Console.WriteLine("sym '\xD' - IsWhiteSpace - {0}", char.IsWhiteSpace("\xD"));
    Console.WriteLine("str: {0}", str);
        Console.WriteLine("и ее пробелы - символ 8 {0}, символ 17 {1}",
            char.IsWhiteSpace(str,8), char.IsWhiteSpace(str,17));
    Console.WriteLine("Метод Parse:");
    str="A";
    ch = char.Parse(str);
    Console.WriteLine("str:{0} char: {1}",str, ch);
    Console.WriteLine("Минимальное и максимальное значение: {0}, {1}",
        char.MinValue.ToString(), char.MaxValue.ToString());
    Console.WriteLine("Их коды: {0}, {1}",
        SayCode(char.MinValue), SayCode(char.MaxValue));
}
//TestCharMethods

```

Результаты работы представлены на рис. 1.1.

```

E:\from_D\C#BookProjects\Strings\bin\Debug\Strings.exe
Статические методы класса char:
Метод GetNumericValue:
sum 'a' - value -1
sum '1' - value 1
Метод GetUnicodeCategory:
sum '1' - category DecimalDigitNumber
sum ';' - category OtherPunctuation
Метод IsControl:
sum '
' - IsControl - True
sum ';' - IsControl - False
Метод IsSeparator:
sum ' ' - IsSeparator - True
sum ';' - IsSeparator - False
Метод IsSurrogate:
sum '?' - IsSurrogate - False
sum '\' - IsSurrogate - False
str = ??, str[0] = ?
str[0] IsSurrogate - True
Метод IsWhiteSpace:
' ' - IsWhiteSpace - True
str: пробелы, пробелы!
Всюду пробелы!
и ее пробелы - символ 8 True, символ 17 True
Метод Parse:
str:A char: A
Минимальное и максимальное значение: a, ?
Их коды: 0, 65535
Press any key to continue

```

Рис. 1.1. Вызовы статических методов класса char

Кроме статических методов, у класса *Char* есть и динамические. Большинство из них - это методы родительского класса *Object*, унаследованные и переопределенные в классе *Char*. Из собственных динамических методов стоит отметить метод *CompareTo*, позволяющий проводить сравнение символов. Он отличается от метода *Equal* тем, что для несовпадающих символов выдает "расстояние" между символами в соответствии с их упорядоченностью в кодировке *Unicode*. Пример:

```

public void testCompareChars()
{
    char ch1, ch2;
    int dif;
    Console.WriteLine("Метод CompareTo");
    ch1='A'; ch2='Z';
    dif = ch1.CompareTo(ch2);
    Console.WriteLine("Расстояние между символами {0}, {1} = {2}", ch1, ch2, dif);
    ch1='a'; ch2='A';
    dif = ch1.CompareTo(ch2);
    Console.WriteLine("Расстояние между символами {0}, {1} = {2}", ch1, ch2, dif);
    ch1='Я'; ch2='A';
    dif = ch1.CompareTo(ch2);
    Console.WriteLine("Расстояние между символами {0}, {1} = {2}", ch1, ch2, dif);
    ch1='A'; ch2='A';
    dif = ch1.CompareTo(ch2);
    Console.WriteLine("Расстояние между символами {0}, {1} = {2}", ch1, ch2, dif);
    ch1='A'; ch2='A';
    dif = ch1.CompareTo(ch2);
    Console.WriteLine("Расстояние между символами {0}, {1} = {2}", ch1, ch2, dif);
    ch1='Ё'; ch2='A';
    dif = ch1.CompareTo(ch2);
    Console.WriteLine("Расстояние между символами {0},
        {1} = {2}", ch1, ch2, dif);
}
//TestCompareChars

```

Результаты сравнения изображены на рис. 1.2.

```

E:\from_D\C#BookProjects\Strings\bin\Debug\Strings.exe
Метод CompareTo
Расстояние между символами A, Z = -25
Расстояние между символами a, A = 32
Расстояние между символами Я, A = 31
Расстояние между символами A, A = 0
Расстояние между символами A, A = 975
Расстояние между символами Ё, A = -15
Press any key to continue

```

Рис. 1.2. Сравнение символов

Анализируя эти результаты, можно понять, что в кодировке **Unicode** как латиница, так и кириллица плотно упакованы. Исключение составляет буква **Ё** - заглавная и малая - они выпадают из плотной кодировки. Малые буквы в кодировке непосредственно следуют за заглавными буквами. Расстояние между алфавитами в кодировке довольно большое - русская буква **А** на 975 символов правее в кодировке, чем соответствующая буква в латинском алфавите.

2.2. Класс **char[]** - массив символов

В языке **C#** определен класс **Char[]**, его можно использовать для представления *строк постоянной длины*, как это делается в **C++**. Поскольку массивы в **C#** динамические, то расширяется класс задач, в которых можно использовать массивы символов для представления строк.

Задаёт ли массив символов **C# строку C**, заканчивающуюся нулем? Нет, не задаёт. Массив **char[]** - это обычный массив. Более того, его нельзя инициализировать строкой символов, как это разрешается в **C++**. Константа, задающая строку символов, принадлежит классу **String**, а в **C#** не определены взаимные преобразования между классами **String** и **Char[]**, даже явные. У класса **String** есть, правда, динамический метод **ToCharArray**, задающий подобное преобразование. Также возможно посимвольно передать содержимое переменной **string** в массив символов. Пример:

```
public void TestCharArAndString()
{
    //массивы символов
    //char[] strM1 = "Hello, World!";
    //ошибка: нет преобразования класса string в класс char[]
    string hello = "Здравствуй, Мир!";
    char[] strM1 = hello.ToCharArray();
    PrintCharAr("strM1",strM1);
    //копирование подстроки
    char[] World = new char[3];
    Array.Copy(strM1,12,World,0,3);
    PrintCharAr("World",World);
    Console.WriteLine(CharArrayToString(World));
} //TestCharArAndString
```

Закомментированные операторы в начале этой процедуры показывают, что прямое присваивание строки массиву символов недопустимо. Однако метод **ToCharArray**, которым обладают строки, позволяет легко преодолеть эту трудность. Ещё одну возможность преобразования строки в массив символов предоставляет статический метод **Copy** класса **Array**.

В примере часть строки **strM1** копируется в массив **World**. По ходу дела в методе вызывается процедура **PrintCharAr** класса **Testing**, печатающая массив символов как строку:

```
void PrintCharAr(string name,char[] ar)
{
    Console.WriteLine(name);
    for(int i=0; i < ar.Length; i++)
        Console.Write(ar[i]);
    Console.WriteLine();
} //PrintCharAr
```

Метод **ToCharArray** позволяет преобразовать строку в массив символов. Обратная операция не определена, поскольку метод **ToString**, которым обладают все объекты класса **Char[]**, печатает информацию о классе, а не содержимое массива. Ситуацию легко исправить, написав подходящую процедуру.

Листинг этой процедуры **CharArrayToString**, вызываемой в тестирующем примере:

```
string CharArrayToString(char[] ar)
{
    string result="";
    for(int i = 0; i < ar.Length; i++) result += ar[i];
    return(result);
} //CharArrayToString
```

Класс **Char[]**, как и всякий класс-массив в **C#**, является наследником не только класса **Object**, но и класса **Array**, и, следовательно, обладает всеми методами родительских классов.

А есть ли у него специфические методы, которые позволяют выполнять операции над строками, представленными массивами символов? Таких специальных операций нет. Но некоторые перегруженные методы класса **Array** можно рассматривать как операции над строками. Например, метод **Copy** даёт возможность выделять и заменять подстроку в теле строки. Методы **IndexOf**, **LastIndexOf** позволяют

определить индексы первого и последнего вхождения в строку некоторого символа. Их нельзя использовать для более интересной операции - нахождения индекса вхождения подстроки в строку. При необходимости такую процедуру можно написать самому. Пример:

```
int IndexOfStr( char[] s1, char[] s2)
{
    //возвращает индекс первого вхождения подстроки s2 в
    //строку s1
    int i =0, j=0, n=s1.Length-s2.Length; bool found = false;
    while( (i<=n) && !found)
    {
        j = Array.IndexOf(s1,s2[0],i);
        if (j <= n)
        {
            found=true; int k = 0;
            while ((k < s2.Length)&& found)
            {
                found =char.Equals(s1[k+j],s2[k]); k++;
            }
        }
        i=j+1;
    }
    if(found) return(j); else return(-1);
} //IndexOfStr
```

В реализации используется метод **IndexOf** класса **Array**, позволяющий найти начало совпадения строк, после чего проверяется совпадение остальных символов. Реализованный здесь алгоритм является самым очевидным, но далеко не самым эффективным.

Рассмотрим процедуру, в которой определяются индексы вхождения символов и подстрок в строку:

```
public void TestIndexSym()
{
    char[] str1, str2;
    str1 = "рококо".ToCharArray();
    //определение вхождения символа
    int find, lind;
    find= Array.IndexOf(str1,'o');
    lind = Array.LastIndexOf(str1,'o');
    Console.WriteLine("Индексы вхождения о в рококо:{0},{1};", find, lind);

    //определение вхождения подстроки
    str2 = "рок".ToCharArray();
    find = IndexOfStr(str1,str2);
    Console.WriteLine("Индекс первого вхождения рок в рококо:{0}", find);
    str2 = "око".ToCharArray();
    find = IndexOfStr(str1,str2);
    Console.WriteLine("Индекс первого вхождения око в рококо:{0}", find);
} //TestIndexSym
```

В этой процедуре вначале используются стандартные методы класса **Array** для определения индексов вхождения символа в строку, а затем созданный метод **IndexOfStr** для определения индекса первого вхождения подстроки. Корректность работы метода проверяется на разных строках. Результаты ее работы – на рис.1.3.

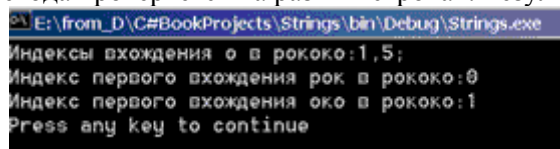


Рис. 1.3. Индексы вхождения подстроки в строку

Существует ли в C# тип **char***

В языке C# указатели допускаются в блоках, отмеченных как небезопасные. Теоретически в таких блоках можно объявить переменную *типa Char**, но все равно не удастся написать столь же короткую, как в C++, процедуру копирования строк. Правильно считать, что в C# строки *типa char** использовать не рекомендуется.

2.3. Класс String

Основным типом при работе со строками является тип *string*, задающий строки переменной длины.

Класс *String* в языке C# относится к ссылочным типам. Над строками - объектами этого класса - определен широкий набор операций, соответствующий современному представлению о том, как должен быть устроен *строковый тип*.

2.3.1. Объявление строк. Конструкторы класса string

Объекты класса *String* объявляются как все прочие объекты простых типов - с явной или отложенной инициализацией, с явным или неявным вызовом конструктора класса. Чаще всего, при объявлении строковой переменной конструктор явно не вызывается, а инициализация задается строковой константой. Но у класса **String** достаточно много конструкторов. Они позволяют сконструировать строку из:

- символа, повторенного заданное число раз;
- массива символов **char[]** ;
- части массива символов.

Некоторым конструкторам в качестве параметра инициализации можно передать строку, заданную типом **char***. Но все это небезопасно, и подобные примеры приводиться и обсуждаться не будут.

Примеры объявления строк с вызовом разных конструкторов:

```
public void TestDeclStrings()
{
    //конструкторы
    string world = "Мир";
    //string s1 = new string("s1");
    //string s2 = new string();
    string sssss = new string('s',5);
    char[] yes = "Yes".ToCharArray();
    string stryes = new string(yes);
    string strye = new string(yes,0,2);
    Console.WriteLine("world = {0}; sssss={1}; stryes={2};"+ " strye= {3}", world, sssss, stryes, strye);
}
```

Объект **world** создан без явного вызова конструктора, а объекты **sssss**, **stryes**, **strye** созданы разными конструкторами класса *String*.

Не допускается явный вызов конструктора по умолчанию - конструктора без параметров. Нет также конструктора, которому в качестве аргумента можно передать обычную строковую константу. Соответствующие операторы в тексте закомментированы.

2.3.2. Операции над строками

Над строками определены следующие операции:

- присваивание (=);
- две операции проверки эквивалентности (==) и (!=);
- конкатенация или сцепление строк (+);
- взятие индекса ([]).

Операция присваивания имеет важную особенность. Поскольку **string** - это ссылочный тип, то в результате присваивания создается ссылка на константную строку, хранимую в "куче". С одной и той же строковой константой в "куче" может быть связано несколько переменных строкового типа. Но эти переменные не являются псевдонимами - разными именами одного и того же объекта. Дело в том, что строковые константы в "куче" не изменяются, поэтому когда одна из переменных получает новое значение, она связывается с новым константным объектом в "куче". Остальные переменные сохраняют свои связи. Для программиста это означает, что семантика присваивания строк аналогична семантике значимого присваивания.

В отличие от других ссылочных типов, операции, проверяющие эквивалентность, сравнивают значения строк, а не ссылки. Эти операции выполняются как над значимыми типами.

Бинарная операция " + " сцепляет две строки, приписывая вторую строку к хвосту первой.

Возможность взятия индекса при работе со строками отражает тот приятный факт, что строку можно рассматривать как массив и получать без труда каждый ее символ. Каждый символ строки имеет тип **char**, доступный только для чтения, но не для записи.

Вот пример, в котором над строками выполняются данные операции:


```

public void TestOps()
{
    //операции над строками
    string s1="ABC", s2="CDE";
    string s3 = s1+s2;
    bool b1 = (s1==s2);
    char ch1 = s1[0], ch2=s2[0];
    Console.WriteLine("s1={0}, s2={1}, b1={2}," + "ch1={3}, ch2={4}", s1,s2,b1,ch1,ch2);
    s2 = s1;
    b1 = (s1!=s2);
    ch2 = s2[0];
    Console.WriteLine("s1={0}, s2={1}, b1={2}," + "ch1={3}, ch2={4}", s1,s2,b1,ch1,ch2);
    //Неизменяемые значения
    s1= "Zenon";
    //s1[0]='L';
}

```

2.3.3. Строковые константы

В C# существуют два вида строковых констант:

- обычные константы, которые представляют строку символов, заключенную в кавычки;
- *@-константы*, заданные обычной константой с предшествующим знаком **@**.

В обычных константах некоторые символы интерпретируются особым образом. Связано это с тем, что необходимо уметь задавать в строке непечатаемые символы, такие, как, например, символ табуляции. Возникает необходимость задавать символы их кодом - в виде escape-последовательностей.

Для всех этих целей используется комбинация символов, начинающаяся символом **"\"** - обратная косая черта. Так, пары символов: **"\n"**, **"\t"**, **"\\"**, **"\""** задают соответственно символ перехода на новую строку, символ табуляции, сам символ обратной косой черты, символ кавычки, вставляемый в строку, но не сигнализирующий о ее окончании.

Комбинация **"\xNNNN"** задает символ, определяемый шестнадцатеричным кодом **NNNN**. Хотя такое решение возникающих проблем совершенно естественно, иногда возникают неудобства: например, при задании констант, определяющих путь к файлу, приходится каждый раз удваивать символ обратной косой черты. Это одна из причин, по которой появились *@-константы*.

В *@-константах* все символы трактуются в полном соответствии с их изображением. Поэтому путь к файлу лучше задавать *@-константой*. Единственная проблема в таких случаях: как задать символ кавычки, чтобы он не воспринимался как конец самой константы. Решением является удвоение символа.

Примеры:

```

//Два вида констант
s1= "\x50";
s2=@"\x50"";
b1= (s1==s2);
Console.WriteLine("s1={0}, s2={1}, b1={2}", s1,s2,b1);
s1 = "c:\\c#book\\ch5\\chapter5.doc";
s2 = @"c:\c#book\ch5\chapter5.doc";
b1= (s1==s2);
Console.WriteLine("s1={0}, s2={1}, b1={2}", s1,s2,b1);
s1= "\"A\"";
s2=@""A"";
b1= (s1==s2);
Console.WriteLine("s1={0}, s2={1}, b1={2}", s1,s2,b1);

```

На рис. 1.4. приведены результаты работы приведенных фрагментов кода, полученные при вызове процедур **TestDeclStrings** и **TestOps**.

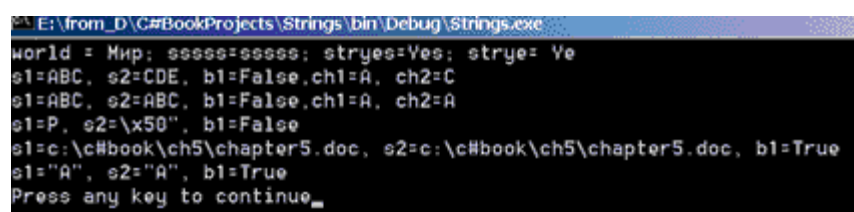


Рис. 1.4. Объявления, константы и операции над объектами string

2.3.4. Неизменяемый класс string

В языке C# существует понятие *неизменяемый (immutable) класс*. Для такого класса невозможно изменить значение объекта при вызове его методов. Динамические методы могут создавать новый объект, но не могут изменить значение существующего объекта.

К таким *неизменяемым классам* относится и класс *String*. Ни один из методов этого класса не меняет значения существующих объектов. Конечно, некоторые из методов создают новые значения и возвращают в качестве результата новые строки. Невозможность изменять значения строк касается не только методов. Аналогично, при работе со строкой как с массивом разрешено только чтение отдельных символов, но не их замена. Оператор присваивания из нашего последнего примера, в котором делается попытка изменить первый символ строки, недопустим, а потому закомментирован.

```
//Неизменяемые значения
s1= "Zenon"; ch1 = s1[0];

//s1[0]='L';
```

Статические свойства и методы класса String

Таблица 1.2. Статические методы и свойства класса String	
Метод	Описание
Empty	Возвращается пустая строка. Свойство со статусом read only
Compare	Сравнение двух строк. Метод перегружен. Реализации 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. Метод перегружен. Реализации метода позволяют сравнивать как строки, так и подстроки. Сравниваются коды символов
Concat	Конкатенация строк. Метод перегружен, допускает сцепление произвольного числа строк
Copy	Создается копия строки
Format	Выполняет форматирование в соответствии с заданными спецификациями <i>формата</i> . Ниже приведено более полное описание метода
Intern, IsIntern	Отыскивается и возвращается ссылка на строку, если таковая уже хранится во внутреннем пуле данных. Если же строки нет, то первый из методов добавляет строку во внутренний пул, второй - возвращает null . Методы применяются обычно тогда, когда строка создается с использованием построителя строк - класса <i>StringBuilder</i>
Join	Конкатенация массива строк в единую строку. При конкатенации между элементами массива вставляются разделители. Операция, заданная методом <i>Join</i> , является обратной к операции, заданной методом <i>Split</i> . Последний является динамическим методом и, используя разделители, осуществляет разделение строки на элементы

Метод Format

Метод *Format* в примерах встречался многократно. Всякий раз, когда выполнялся вывод результатов на консоль, неявно вызывался *иметод Format*. Рассмотрим оператор печати:

```
Console.WriteLine("s1={0}, s2={1}", s1,s2);
```

Здесь строка, задающая первый аргумент метода, помимо обычных символов, содержит *форматы*, заключенные в фигурные скобки. В данном примере используется простейший вид *формата* - он определяет объект, который должен быть подставлен в участок строки, занятый данным *форматом*. Помимо неявных вызовов, нередко возникает необходимость явного форматирования строки.

Рассмотрим общий синтаксис *метода Format* и используемых в нем *форматов*.

Метод *Format*, как и большинство методов, является перегруженным и может вызываться с разным числом параметров. Первый необязательный параметр метода задает провайдера, определяющего национальные особенности, которые используются в процессе форматирования. В качестве такого параметра должен быть задан объект, реализующий интерфейс **System.IFormatProvider**. Если этот параметр не задан, то используется культура, заданная по умолчанию.

Примеры двух реализаций этого метода:

```
public static string Format(string, object);
public static string Format(IFormatProvider, string, params object[]);
```

Параметр типа *string* задает формируемую строку. Заданная строка содержит один или несколько *форматов*, они распознаются за счет окружающих *формат* фигурных скобок. Число *форматов*, вставленных в строку, определяет и число объектов, передаваемых при вызове *метода Format*. Каждый *формат* определяет форматирование объекта, на который он ссылается и который, после преобразования его в строку, будет подставлен в результирующую строку вместо *формата*. Метод *Format* в

качестве результата возвращает переданную ему строку, где все спецификации *формата* заменены строками, полученными в результате форматирования объектов.

Общий синтаксис, специфицирующий *формат*, таков:

```
{N [,M [:<коды_форматирования>]]}
```

Обязательный параметр **N** задает индекс объекта, заменяющего *формат*. Можно считать, что методу всегда передается массив объектов, даже если фактически передан один объект. Индексация объектов начинается с нуля, как это принято в массивах. Второй параметр **M**, если он задан, определяет минимальную ширину поля, которое отводится строке, вставляемой вместо *формата*. Третий необязательный параметр задает коды форматирования, указывающие, как следует форматировать объект. Например, код **C** (**Currency**) говорит о том, что параметр должен форматироваться как валюта с учетом национальных особенностей представления. Код **P** (**Percent**) задает форматирование в виде процентов с точностью до сотой доли.

Примеры:

```
public void TestFormat()
{
    //метод Format
    int x=77;
    string s= string.Format("x={0}",x);
    Console.WriteLine(s + "\tx={0}",x);
    s= string.Format("Итого:{0,10} рублей",x);
    Console.WriteLine(s);
    s= string.Format("Итого:{0,6:#####} рублей",x);
    Console.WriteLine(s);
    s= string.Format("Итого:{0:P} ",0.77);
    Console.WriteLine(s);
    s= string.Format("Итого:{0,4:C} ",77.77);
    Console.WriteLine(s);
    //Национальные особенности
    System.Globalization.CultureInfo ci = new System.Globalization.CultureInfo("en-US");
    s= string.Format(ci,"Итого:{0,4:C} ",77.77);
    Console.WriteLine(s);
} //TestFormat
```

Вначале демонстрируется, что и явный, и неявный вызовы *метода Format* дают один и тот же результат. В дальнейших примерах показано использование различных спецификаций *формата* с разным числом параметров и разными кодами форматирования. В частности, показан вывод процентов и валют. В последнем примере с валютами демонстрируется задание провайдером национальных особенностей. С этой целью создается объект класса **CultureInfo**, инициализированный так, чтобы он задавал особенности *форматирования*, принятые в США. Класс **CultureInfo** наследует интерфейс **IFormatProvider**. Российские национальные особенности форматирования установлены по умолчанию. При необходимости их можно установить таким же образом, как это сделано для США, задав соответственно константу "**ru-RU**". Результаты работы метода показаны на рис. 1.5.

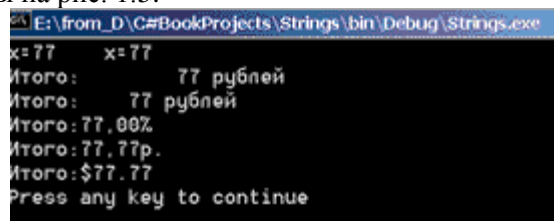


Рис. 1.5. Результаты работы метода Format

Методы Join и Split

Методы *Join* и *Split* выполняют над строкой текста взаимно обратные преобразования. *Динамический метод Split* позволяет осуществить разбор текста на элементы. *Статический метод Join* выполняет обратную операцию, собирая строку из элементов.

Заданный строкой текст зачастую представляет собой совокупность структурированных элементов - абзацев, предложений, слов, скобочных выражений и т.д. При работе с таким текстом необходимо разделить его на элементы, пользуясь специальными разделителями элементов, - это могут быть пробелы, скобки, знаки препинания. Практически подобные задачи возникают постоянно при работе со структурированными текстами. Методы *Split* и *Join* облегчают решение этих задач.

Динамический метод Split, как обычно, перегружен. Наиболее часто используемая реализация имеет следующий синтаксис:

```
public string[] Split(params char[])
```

На вход методу *Split* передается один или несколько символов, интерпретируемых как разделители. Объект **string**, вызвавший метод, разделяется на подстроки, ограниченные этими разделителями. Из этих подстрок создается массив, возвращаемый в качестве результата метода. Другая реализация позволяет ограничить число элементов возвращаемого массива.

Синтаксис *статического метода Join*:

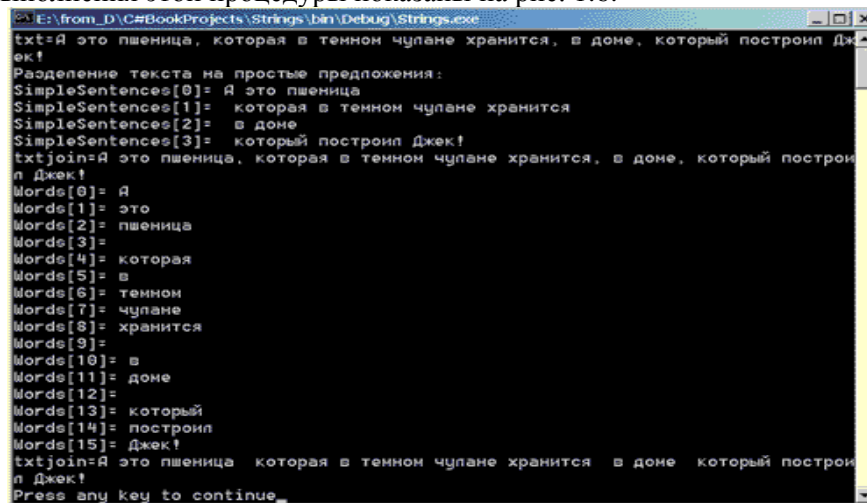
```
public static string Join(string delimiters, string[] items )
```

В качестве результата метод возвращает строку, полученную конкатенацией элементов массива **items**, между которыми вставляется строка разделителей **delimiters**. Как правило, строка **delimiters** состоит из одного символа, который и разделяет в результирующей строке элементы массива **items**; но в отдельных случаях ограничителем может быть строка из нескольких символов.

Примеры применения этих методов. В первом из них строка представляет сложноподчиненное предложение, которое разбивается на простые предложения. Во втором предложение разделяется на слова. Затем производится обратная сборка разобранного текста. Код соответствующей процедуры:

```
public void TestSplitAndJoin()
{
    string txt = "А это пшеница, которая в темном чулане хранится, " + "в доме, который построил Джек!";
    Console.WriteLine("txt={0}", txt);
    Console.WriteLine("Разделение текста на простые предложения:");
    string[] SimpleSentences, Words;
    //размерность массивов SimpleSentences и Words
    //устанавливается автоматически в соответствии с
    //размерностью массива, возвращаемого методом Split
    SimpleSentences = txt.Split(',');
    for(int i=0;i< SimpleSentences.Length; i++)
        Console.WriteLine("SimpleSentences[{0}]= {1}", i, SimpleSentences[i]);
    string txtjoin = string.Join(", ", SimpleSentences);
    Console.WriteLine("txtjoin={0}", txtjoin);
    Words = txt.Split(' ', ' ');
    for(int i=0;i< Words.Length; i++)
        Console.WriteLine("Words[{0}]= {1}", i, Words[i]);
    txtjoin = string.Join(" ", Words);
    Console.WriteLine("txtjoin={0}", txtjoin);
} //TestSplitAndJoin
```

Результаты выполнения этой процедуры показаны на рис. 1.6.



```
E:\from_D\CS\BookProjects\Strings\bin\Debug\Strings.exe
txt=А это пшеница, которая в темном чулане хранится, в доме, который построил Джек!
Разделение текста на простые предложения:
SimpleSentences[0]= А это пшеница
SimpleSentences[1]= которая в темном чулане хранится
SimpleSentences[2]= в доме
SimpleSentences[3]= который построил Джек!
txtjoin=А это пшеница, которая в темном чулане хранится, в доме, который построил Джек!
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!
txtjoin=А это пшеница которая в темном чулане хранится в доме который построил Джек!
Press any key to continue
```

Рис. 1.6. Разбор и сборка строки текста

Обратите внимание, что 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. Поэтому при сборке между элементами вставляется один разделитель, возможно, состоящий из нескольких символов;
- при разборе двух подряд идущих разделителей предполагается, что между ними находится пустое слово. Обратите внимание в тексте нашего примера, как и положено, после запятой следует пробел.

При разборе текста на слова в качестве разделителей указаны символы пробела и запятой. По этой причине в массиве слов, полученном в результате разбора, имеются пустые слова.

Если при разборе предложения на слова использовать в качестве разделителя только пробел, то пустые слова не появятся, но запятая будет являться частью некоторых слов.

Как всегда, есть несколько способов справиться с проблемой. Один из них состоит в том, чтобы написать собственную реализацию этих функций, другой - в корректировке полученных результатов, третий - в использовании более мощного аппарата регулярных выражений, и о нем мы поговорим чуть позже.

Динамические методы класса String

Операции, разрешенные над строками в C#, разнообразны. Методы этого класса позволяют выполнять вставку, удаление, замену, поиск вхождения подстроки в строку. Класс *String* наследует методы класса *Object*, частично их переопределяя. Класс *String* наследует и, следовательно, реализует методы четырех интерфейсов: *Comparable*, *Cloneable*, *Convertible*, *Enumerable*. Три из них уже рассматривались при описании классов-массивов.

Рассмотрим наиболее характерные методы при работе со строками.

В таблице 1.3 приведены методы класса, которые дают достаточно полную картину широких возможностей, имеющихся при работе со строками в C#.

Следует помнить, что класс *string* является неизменяемым. Поэтому *Replace*, *Insert* и другие методы представляют собой функции, возвращающие новую строку в качестве результата и не изменяющие строку, вызвавшую метод.

Таблица 1.3. Динамические методы и свойства класса String	
Метод	Описание
<i>Insert</i>	Вставляет подстроку в заданную позицию
<i>Remove</i>	Удаляет подстроку в заданной позиции
<i>Replace</i>	Заменяет подстроку в заданной позиции на новую подстроку
<i>Substring</i>	Выделяет подстроку в заданной позиции
<i>IndexOf</i> , <i>IndexOfAny</i> , <i>LastIndexOf</i> , <i>LastIndexOfAny</i>	Определяются индексы первого и последнего вхождения заданной подстроки или любого символа из заданного набора
<i>StartsWith</i> , <i>EndsWith</i>	Возвращается <i>true</i> или <i>false</i> , в зависимости от того, начинается или заканчивается строка заданной подстрокой
<i>PadLeft</i> , <i>PadRight</i>	Выполняет набивку нужным числом пробелов в начале и в конце строки
<i>Trim</i> , <i>TrimStart</i> , <i>TrimEnd</i>	Обратные операции к методам <i>Pad</i> . Удаляются пробелы в начале и в конце строки, или только с одного ее конца
<i>ToCharArray</i>	Преобразование строки в массив символов

2.3.5. Класс StringBuilder - построитель строк

Класс *string* не разрешает изменять существующие объекты.

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

Объявление строк. Конструкторы класса StringBuilder

Объекты этого класса объявляются с явным вызовом конструктора класса. Поскольку специальных констант этого типа не существует, то вызов конструктора для инициализации объекта просто необходим. Конструктор класса перегружен, и наряду с конструктором без параметров, создающим пустую строку, имеется набор конструкторов, которым можно передать две группы параметров. Первая группа позволяет задать строку или подстроку, значением которой будет инициализироваться создаваемый объект класса *StringBuilder*. Вторая группа параметров позволяет задать *емкость объекта* - объем памяти, отводимой данному экземпляру класса *StringBuilder*. Каждая из этих групп не является обязательной и может быть опущена. Примером может служить конструктор без параметров, который создает объект, инициализированный пустой строкой, и с некоторой *емкостью*, заданной по умолчанию, значение которой зависит от реализации.

Примеры синтаксис трех конструкторов:

- `public StringBuilder (string str, int cap)`. Параметр *str* задает строку инициализации, *cap* - *емкость объекта* ;
- `public StringBuilder (int curcap, int maxcap)`. Параметры *curcap* и *maxcap* задают начальную и максимальную *емкость объекта* ;
- `public StringBuilder (string str, int start, int len, int cap)`. Параметры *str*, *start*, *len* задают строку инициализации, *cap* - *емкость объекта*.

Операции над строками

Над строками этого класса определены практически те же операции с той же семантикой, что и над строками *класса String*:

- присваивание (=);
- две операции проверки эквивалентности (==) и (!=);
- взятие индекса ([]).

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

Со строкой этого класса можно работать как с массивом, но, в отличие от *класса String*, здесь уже все делается как надо: допускается не только чтение отдельного символа, но и его изменение.

Рассмотрим с небольшими модификациями старый пример:

```
public void TestStringBuilder()
{
    //Строки класса StringBuilder
    //операции над строками
    StringBuilder s1=new StringBuilder("ABC"), s2 =new StringBuilder("CDE");
    StringBuilder s3 = new StringBuilder();
    //s3= s1+s2;
    s3= s1.Append(s2);
    bool b1 = (s1==s3);
    char ch1 = s1[0], ch2=s2[0];
    Console.WriteLine("s1={0}, s2={1}, b1={2}," + "ch1={3}, ch2={4}", s1,s2,b1,ch1,ch2);
    s2 = s1;
    b1 = (s1!=s2);
    ch2 = s2[0];
    Console.WriteLine("s1={0}, s2={1}, b1={2}," + "ch1={3}, ch2={4}", s1,s2,b1,ch1,ch2);
    StringBuilder s = new StringBuilder("Zenon");
    s[0]='L';
    Console.WriteLine(s);
} //TestStringBuilder
```

Этот пример демонстрирует возможность выполнения над строками *класса StringBuilder* тех же операций, что и над строками *класса String*. В результате присваивания создается дополнительная ссылка на объект, операции проверки на эквивалентность работают со значениями строк, а не со ссылками на них. Конкатенацию можно заменить вызовом метода **Append**. Появляется новая возможность - изменять отдельные символы строки. (Для того чтобы имя *класса StringBuilder* стало доступным, в проект добавлено предложение **using System.Text**, ссылающееся на соответствующее пространство имен.)

Основные методы

У *класса StringBuilder* методов значительно меньше, чем у *класса String*, так как класс создавался с целью дать возможность изменять значение строки. По этой причине у *класса* есть основные методы, позволяющие выполнять такие операции над строкой как вставка, удаление и замена подстрок, но нет методов, подобных поиску вхождения, которые можно выполнять над обычными строками. Технология работы обычно такова: конструируется строка *класса StringBuilder*; выполняются операции, требующие изменение значения; полученная строка преобразуется в строку *класса String*; над этой строкой выполняются операции, не требующие изменения значения строки. Давайте чуть более подробно рассмотрим основные методы *класса StringBuilder*:

- **public StringBuilder Append (<объект>)**. К строке, вызвавшей метод, присоединяется строка, полученная из объекта, который передан методу в качестве параметра. Метод перегружен и может принимать на входе объекты всех простых типов, начиная от **char** и **bool** до **string** и **long**. Поскольку объекты всех этих типов имеют метод **ToString**, всегда есть возможность преобразовать объект в строку, которая и присоединяется к исходной строке. В качестве результата возвращается ссылка на объект, вызвавший метод. Поскольку возвращаемую ссылку ничему присваивать не нужно, то правильнее считать, что метод изменяет значение строки;
- **public StringBuilder Insert (int location,<объект>)**. Метод вставляет строку, полученную из объекта, в позицию, указанную параметром **location**. Метод **Append** является частным случаем метода **Insert** ;
- **public StringBuilder Remove (int start, int len)**. Метод удаляет подстроку длины **len**, начинающуюся с позиции **start** ;
- **public StringBuilder Replace (string str1,string str2)**. Все вхождения подстроки **str1** заменяются на строку **str2**;

- `public StringBuilder AppendFormat (<строка форматов>, <объекты>)`. Метод является комбинацией метода `Format` класса `String` и метода `Append`. Строка *форматов*, переданная методу, содержит только спецификации *форматов*. В соответствии с этими спецификациями находятся и форматируются объекты. Полученные в результате форматирования строки присоединяются в конец исходной строки.

За исключением метода `Remove`, все рассмотренные методы являются перегруженными. В их описании дана схема вызова метода, а не точный синтаксис перегруженных реализаций. Примеры, чтобы продемонстрировать, как вызываются и как работают эти методы:

```
//Методы Insert, Append, AppendFormat
StringBuilder strbuild = new StringBuilder();
string str = "это это не ";
strbuild.Append(str); strbuild.Append(true);
strbuild.Insert(4,false); strbuild.Insert(0,"2*2=5 - ");
Console.WriteLine(strbuild);
string txt = "А это пшеница, которая в темном чулане хранится, "+" в доме, который построил Джек!";
StringBuilder txtbuild = new StringBuilder();
int num =1;
foreach(string sub in txt.Split(','))
{
    txtbuild.AppendFormat(" {0}: {1} ", num++,sub);
}
str = txtbuild.ToString();
Console.WriteLine(str);
```

В этом фрагменте кода конструируются две строки. Первая из них создается из строк и булевых значений `true` и `false`. Для конструирования используются методы `Insert` и `Append`. Вторая строка конструируется в цикле с применением метода `AppendFormat`. Результатом этого конструирования является строка, в которой простые предложения исходного текста пронумерованы.

Сконструированная вторая строка передается в обычную строку класса `String`. Никаких проблем преобразования строк одного класса в другой класс не возникает, поскольку все объекты, в том числе, объекты класса `StringBuilder`, обладают по определению методом `ToString`.

Результаты работы – на рис.1.7.

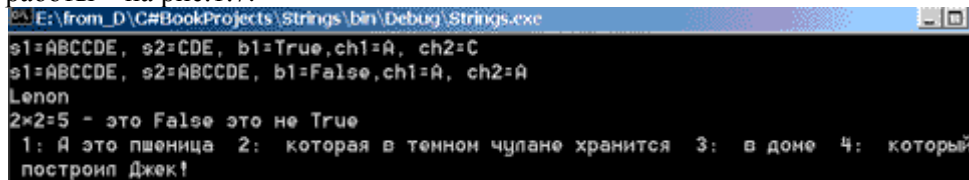


Рис. 1.7. Операции и методы класса `StringBuilder`

Емкость буфера

Каждый экземпляр строки класса `StringBuilder` имеет буфер, в котором хранится строка. Объем буфера - его емкость - может меняться в процессе работы со строкой. Объекты класса имеют две характеристики емкости - текущую и максимальную. В процессе работы текущая емкость изменяется, естественно, в пределах максимальной емкости, которая реально достаточно высока. Если размер строки увеличивается, то соответственно автоматически растет и текущая емкость. Если же размер строки уменьшается, то емкость буфера остается на том же уровне. По этой причине иногда разумно уменьшать емкость. Следует помнить, что попытка уменьшить емкость до величины, меньшей длины строки, приведет к ошибке.

У класса `StringBuilder` имеется 2 свойства и один метод, позволяющие анализировать и управлять емкостными свойствами буфера. Напомню, что этими характеристиками можно управлять также еще на этапе создания объекта, - для этого имеется соответствующий конструктор. Рассмотрим свойства и метод класса, связанные с емкостью буфера:

- свойство `Capacity` - возвращает или устанавливает текущую емкость буфера;
- свойство `MaxCapacity` - возвращает максимальную емкость буфера. Результат один и тот же для всех экземпляров класса;
- метод `int EnsureCapacity (int capacity)` - позволяет уменьшить емкость буфера. Метод пытается вначале установить емкость, заданную параметром `capacity`; если это значение меньше размера хранимой строки, то емкость устанавливается такой, чтобы гарантировать размещение строки. Это число и возвращается в качестве результата работы метода.

3. Пространство имен `RegularExpression` и классы регулярных выражений

Стандартный класс `String` позволяет выполнять над строками различные операции, в том числе поиск, замену, вставку и удаление подстрок. Существуют специальные операции, такие как `Join`, `Split`, которые облегчают разбор строки на элементы. Тем не менее, есть классы задач по обработке символьной информации, где стандартных возможностей явно не хватает. Чтобы облегчить решение подобных задач, в `Net Framework` встроен более мощный аппарат работы со строками, основанный на регулярных выражениях. Специальное пространство имен `RegularExpression`, содержит набор классов, обеспечивающих работу с регулярными выражениями. Все классы этого пространства доступны для C# и всех языков, использующих каркас `Net Framework`.

В основе регулярных выражений лежит хорошая теория и хорошая практика их применения. Полное описание, как теоретических основ, так и практических особенностей применения этого аппарата в C#, требует отдельной книги. Ограничимся введением в эту область работы со строками, не рассматривая подробно все классы, входящие в пространство имен `RegularExpression`.

С точки зрения практики регулярное выражение задает образец поиска. После чего можно проверить, удовлетворяет ли заданная строка или ее подстрока данному образцу. В языках программирования синтаксис регулярного выражения существенно обогащается, что дает возможность более просто задавать сложные образцы поиска. Такие синтаксические надстройки, хотя и не меняют сути регулярных выражений, полезны для практиков, избавляя программиста от ненужных сложностей.

3.1. Синтаксис регулярных выражений

Регулярное выражение на C# задается строковой константой. Это может быть обычная или @-константа. Чаще всего, следует использовать именно @-константу. Дело в том, что символ "\" широко применяется в регулярных выражениях как для записи escape-последовательностей, так и в других ситуациях. Обычные константы в таких случаях будут выдавать синтаксическую ошибку, а @-константы не выдают ошибок и корректно интерпретируют запись регулярного выражения.

Синтаксис регулярного выражения простой формулой не описать, здесь используются набор разнообразных средств:

- символы и escape-последовательности;
- символы операций и символы, обозначающие специальные классы множеств;
- имена групп и обратные ссылки;
- символы утверждений и другие средства.

Конечно, регулярное выражение может быть совсем простым, например, строка "abc" задает образец поиска, так что при вызове соответствующего метода будут разыскиваться одно или все вхождения подстроки "abc" в искомую строку. Но могут существовать и очень сложно устроенные регулярные выражения.

В таблице (1.4) дается интерпретация символов в соответствии с их делением на группы. Таблица не полна, в ней отражаются не все группы, а описание группы не содержит всех символов. В ней не отражены такие категории, как подстановки, обратные ссылки, утверждения. Она позволяет дать общее представление о синтаксисе. За деталями придется обращаться к справочной системе.

Для приведенных категорий также не дан полный список возможных символов.

Таблица 1.4. Символы, используемые в регулярных выражениях

Символ	Интерпретация
Категория: escape-последовательности	
<code>\b</code>	При использовании его в квадратных скобках соответствует символу "обратная косая черта" с кодом - <code>\u0008</code>
<code>\t</code>	Соответствует символу табуляции <code>\u0009</code>
<code>\r</code>	Соответствует символу возврата каретки <code>\u000D</code>
<code>\n</code>	Соответствует символу новой строки <code>\u000A</code>
<code>\e</code>	Соответствует символу escape <code>\u001B</code>
<code>\040</code>	Соответствует символу ASCII, заданному кодом до трех цифр в восьмеричной системе
<code>\x20</code>	Соответствует символу ASCII, заданному кодом из двух цифр в шестнадцатеричной системе
<code>\u0020</code>	Соответствует символу Unicode, заданному кодом из четырех цифр в шестнадцатеричной системе
Категория: подмножества (классы) символов	
<code>.</code>	Соответствует любому символу, за исключением символа конца строки
<code>[aeiou]</code>	Соответствует любому символу из множества, заданного в квадратных скобках
<code>[^aeiou]</code>	Отрицание. Соответствует любому символу, за исключением символов, заданных в квадратных скобках
<code>[0-9a-fA-F]</code>	Задание диапазона символов, упорядоченных по коду. Так, 0-9 задает любую цифру

<code>\p{name}</code>	Соответствует любому символу, заданному множеству с именем name, например, имя LI задает множество букв латиницы в нижнем регистре. Поскольку все символы разбиты на подмножества, задаваемые категорией Unicode, то в качестве имени можно задавать имя категории
<code>\P{name}</code>	Отрицание. Большая буква всегда задает отрицание множества, заданного малой буквой
<code>\w</code>	Множество символов, используемых при задании идентификаторов - большие и малые символы латиницы, цифры и знак подчеркивания
<code>\s</code>	Соответствует символам белого пробела
<code>\d</code>	Соответствует любому символу из множества цифр
Категория: Операции (модификаторы)	
<code>*</code>	<i>Итерация</i> . Задает ноль или более соответствий; например, <code>\w*</code>
<code>(abc)*.</code>	Аналогично, <code>{0,}</code>
<code>+</code>	Положительная <i>итерация</i> . Задает одно или более соответствий; например, <code>\w+</code> или <code>(abc)+</code> . Аналогично, <code>{1,}</code>
<code>?</code>	Задает ноль или одно соответствие; например, <code>\w?</code> или <code>(abc)?</code> . Аналогично, <code>{0,1}</code>
<code>{n}</code>	Задает в точности n соответствий; например, <code>\w{2}</code>
<code>{n,}</code>	Задает, по меньшей мере, n соответствий; например, <code>(abc){2,}</code>
<code>{n,m}</code>	Задает, по меньшей мере, n, но не более m соответствий; например, <code>(abc){2,5}</code>
Категория: Группирование	
<code>(?<Name>)</code>	При обнаружении соответствия выражению, заданному в круглых скобках, создается именованная <i>группа</i> , которой дается имя Name . Например, <code>(?<tel> \d{7})</code> . При обнаружении последовательности из семи цифр будет создана <i>группа</i> с именем tel
<code>()</code>	Круглые скобки разбивают <i>регулярное выражение</i> на <i>группы</i> . Для каждого подвыражения, заключенного в круглые скобки, создается <i>группа</i> , автоматически получающая номер. Номера следуют в обратном порядке, поэтому полному <i>регулярному выражению</i> соответствует <i>группа</i> с номером 0
<code>(?imnsx)</code>	Включает или выключает в <i>группе</i> любую из пяти возможных опций. Для выключения опции перед ней ставится знак минус. Например, <code>(?i-s:)</code> включает опцию i , задающую нечувствительность к регистру, и выключает опцию s - статус single-line

3.2. Классы пространства RegularExpressions

В данном пространстве расположено семейство из одного перечисления и восьми связанных между собой классов.

3.2.1. Класс Regex

Это основной класс, всегда создаваемый при работе с *регулярными выражениями*. Объекты этого класса определяют *регулярные выражения*. Конструктор класса, как обычно, перегружен. В простейшем варианте ему передается в качестве параметра строка, задающая *регулярное выражение*. В других вариантах конструктора ему может быть передан объект, принадлежащий перечислению **RegexOptions** и задающий опции, которые действуют при работе с данным объектом. Среди опций отмечу одну: ту, что позволяет компилировать *регулярное выражение*. В этом случае создается программа, которая и будет выполняться при каждом поиске соответствия. При разборе больших текстов скорость работы в этом случае существенно повышается.

Рассмотрим четыре основных метода класса *Regex*.

Метод Match запускает поиск соответствия. В качестве параметра методу передается строка поиска, где разыскивается первая подстрока, которая удовлетворяет образцу, заданному *регулярным выражением*. В качестве результата метод возвращает объект класса *Match*, описывающий результат поиска. При успешном поиске свойства объекта будут содержать информацию о найденной подстроке.

Метод Matches позволяет разыскать все вхождения, то есть все подстроки, удовлетворяющие образцу. У алгоритма поиска есть важная особенность - разыскиваются непересекающиеся вхождения подстрок. Можно считать, что метод **Matches** многократно запускает метод *Match*, каждый раз начиная поиск с того места, на котором закончился предыдущий поиск. В качестве результата возвращается объект *MatchCollection*, представляющий коллекцию объектов *Match*.

Метод NextMatch запускает новый поиск, начиная с того места, на котором остановился предыдущий поиск.

Метод Split является обобщением метода **Split** класса **String**. Он позволяет, используя образец, разделить искомую строку на элементы. Поскольку образец может быть устроен сложнее, чем простое множество разделителей, то метод **Split** класса *Regex* эффективнее, чем его аналог класса **String**.

Классы Match и MatchCollection

Как уже говорилось, объекты этих классов создаются автоматически при вызове *методов Match и Matches*. Коллекция *MatchCollection*, как и все коллекции, позволяет получить доступ к каждому ее элементу - объекту *Match*. Можно, конечно, организовать цикл *foreach* для последовательного доступа ко всем элементам коллекции.

Класс *Match* является непосредственным наследником класса *Group*, который, в свою очередь, является наследником класса *Capture*. При работе с объектами класса *Match* наибольший интерес представляют не столько методы класса, сколько его свойства, большая часть которых унаследована от родительских классов. Рассмотрим основные свойства:

- свойства *Index*, *Length* и *Value* наследованы от прародителя *Capture*. Они описывают найденную подстроку - индекс начала подстроки в искомой строке, длину подстроки и ее значение;
- свойство *Groups* класса *Match* возвращает коллекцию *групп* - объект *GroupCollection*, который позволяет работать с *группами*, созданными в процессе поиска соответствия;
- свойство *Captures*, наследованное от объекта *Group*, возвращает коллекцию *CaptureCollection*.

Как видите, при работе с *регулярными выражениями* реально приходится создавать один объект класса *Regex*, объекты других классов автоматически появляются в процессе работы с объектами *Regex*.

Классы Group и GroupCollection

Коллекция *GroupCollection* возвращается при вызове свойства *Group* объекта *Match*. Имея эту коллекцию, можно добраться до каждого объекта *Group*, в нее входящего. Класс *Group* является наследником класса *Capture* и, одновременно, родителем класса *Match*. От своего родителя он наследует свойства *Index*, *Length* и *Value*, которые и передает своему потомку.

Давайте рассмотрим чуть более подробно, когда и как создаются *группы* в процессе поиска соответствия. Если внимательно проанализировать предыдущую таблицу, которая описывает символы, используемые в *регулярных выражениях*, в частности символы группирования, то можно понять несколько важных фактов, связанных с *группами*:

- при обнаружении одной подстроки, удовлетворяющей условию поиска, создается не одна *группа*, а коллекция *групп*;
- *группа* с индексом 0 содержит информацию о найденном соответствии;
- число *групп* в коллекции зависит от числа круглых скобок в записи *регулярного выражения*.

Каждая пара круглых скобок создает дополнительную *группу*, которая описывает ту часть подстроки, которая соответствует шаблону, заданному в круглых скобках;

- *группы* могут быть индексированы, но могут быть и именованными, поскольку в круглых скобках разрешается указывать имя *группы*.

Создание именованных *групп* крайне полезно при разборе строк, содержащих разнородную информацию.

3.2.2. Классы Capture и CaptureCollection

Коллекция *CaptureCollection* возвращается при вызове свойства *Captures* объектов класса *Group* и *Match*. Класс *Match* наследует это свойство у своего родителя - класса *Group*. Каждый объект *Capture*, входящий в коллекцию, характеризует соответствие, захваченное в процессе поиска, - соответствующую подстроку. Но поскольку свойства объекта *Capture* передаются по наследству его потомкам, то можно избежать непосредственной работы с объектами *Capture*. В примерах не встретится работа с этим объектом, хотя "за кулисами" он непременно присутствует.

Перечисление RegexOptions

Объекты этого перечисления описывают опции, влияющие на то, как устанавливается соответствие. Обычно такой объект создается первым и передается конструктору объекта класса *Regex*. В вышеприведенной таблице, в разделе, посвященном символам группирования, говорится о том, что опции можно включать и выключать, распространяя, тем самым, их действие на участок шаблона, заданный соответствующей *группой*. Об одной из этих опций - *Compiled*, влияющей на эффективность работы *регулярных выражений*, уже упоминалось. Об остальных говорить не буду - при необходимости можно посмотреть справку.

3.2.3. Класс RegexCompilationInfo

При работе со сложными и большими текстами полезно предварительно скомпилировать используемые в процессе поиска *регулярные выражения*. В этом случае необходимо будет создать объект класса *RegexCompilationInfo* и передать ему информацию о *регулярных выражениях*, подлежащих компиляции, и о том, куда поместить оттранслированную программу. Дополнительно в таких ситуациях следует включить опцию *Compiled*.

3.3. Примеры работы с регулярными выражениями

Примеры дополняют краткое описание возможностей *регулярных выражений* и позволят лучше понять, как с ними работать.

Функция **FindMatch** - производит поиск первого вхождения подстроки, соответствующей образцу:

```
static string FindMatch(string str, string strpat)
{
    Regex pat = new Regex(strpat);
    Match match = pat.Match(str);
    string found = "";
    if (match.Success)
    {
        found = match.Value;
        Console.WriteLine("Строка={0}\tОбразец={1}\tНайдено={2}", str, strpat, found);
    }
    return(found);
}
//FindMatch
```

В качестве входных аргументов функции передается строка **str**, в которой ищется вхождение, и строка **strpat**, задающая образец - *регулярное выражение*. Функция возвращает найденную в результате поиска подстроку. Если соответствия нет, то возвращается пустая строка. Функция начинает свою работу с создания объекта **pat** класса *Regex*, конструктору которого передается образец поиска. Затем вызывается метод *Match* этого объекта, создающий объект **match** класса *Match*. Далее анализируются свойства этого объекта. Если соответствие обнаружено, то найденная подстрока возвращается в качестве результата, а соответствующая информация выводится на печать. (Чтобы спокойно работать с классами *регулярных выражений*, следует добавить в начало проекта предложение: **using System.Text.RegularExpressions.**)

Поскольку запись *регулярных выражений* - вещь, привычная не для всех программистов, приведем несколько примеров:

```
public void TestSinglePat()
{
    //поиск по образцу первого вхождения
    string str, strpat, found;
    Console.WriteLine("Поиск по образцу");
    //образец задает подстроку, начинающуюся с символа a, далее идут буквы или цифры.
    str = "start"; strpat = @"a\w+";
    found = FindMatch(str, strpat);
    str = "fab77cd efg";
    found = FindMatch(str, strpat);
    //образец задает подстроку, начинающуюся с символа a,
    //заканчивающуюся f с возможными символами b и d в середине
    strpat = "a(b|d)*f"; str = "fabadddbdf";
    found = FindMatch(str, strpat);
    //диапазоны и escape-символы
    strpat = "[X-Z]+"; str = "aXYb";
    found = FindMatch(str, strpat);
    strpat = @"\"u0058Y\"x5A"; str = "aXYZb";
    found = FindMatch(str, strpat);
}
//TestSinglePat
```

Некоторые комментарии к этой процедуре.

Регулярные выражения задаются @ - константами. Здесь они как нельзя кстати.

В первом образце используется последовательность символов **\w+**, обозначающая, как следует из таблицы 1.4, непустую последовательность латиницы и цифр. В совокупности образец задает подстроку, начинающуюся символом **a**, за которым следуют буквы или цифры (хотя бы одна). Этот образец применяется к двум различным строкам.

В следующем образце используется символ ***** для обозначения *итерации*. В целом *регулярное выражение* задает строки, начинающиеся с символа **a** и заканчивающиеся символом **f**, между которыми находится возможно пустая последовательность символов из **b** и **d**.

Последующие два образца демонстрируют использование диапазонов и escape-последовательностей для представления символов, заданных кодами (в Unicode и шестнадцатеричной кодировке).

Результаты, полученные при работе этой процедуры – на рис 1.8.

```
E:\from_D\C#BookProjects\Strings\bin\Debug\Strings.exe
Поиск по образцу
Строка =start  Образец=a\w+  Найдено=art
Строка =fab77cd efg  Образец=a\w+  Найдено=ab77cd
Строка =fabadddbf  Образец=a(b|d)*f  Найдено=adddbf
Строка =aXyb  Образец=[X-Z]+  Найдено=XY
Строка =aXYZb  Образец=\u0058Y\x5A  Найдено=XYZ
Press any key to continue_
```

Рис. 1.8. Регулярные выражения. Поиск по образцу

Пример "око и рококо"

Следующий образец в примере позволяет прояснить некоторые особенности работы метода **Matches**. Сколько раз строка "око" входит в строку "рококо" - один или два? Все зависит от того, как считать. С точки зрения метода **Matches**, - один раз, поскольку он разыскивает непересекающиеся вхождения, начиная очередной поиск вхождения подстроки с того места, где закончилось предыдущее вхождение. Еще один пример на эту же тему работает с числовыми строками.

```
Console.WriteLine("око и рококо");
strpat="око"; str = "рококо";
FindMatches(str, strpat);
strpat="123";
str= "0123451236123781239";
FindMatches(str, strpat);
```

На рис. 1.9 показаны результаты поисков.

```
око и рококо
Строка =рококо Образец=око
Число совпадений =1
Index = 1 Value = око, Length =3
Строка =0123451236123781239 Образец=123
Число совпадений =4
Index = 1 Value = 123, Length =3
Index = 6 Value = 123, Length =3
Index = 10 Value = 123, Length =3
Index = 15 Value = 123, Length =3
```

Рис. 1.9. Регулярные выражения. Пример "око и рококо"

Пример "кок и кук"

Этот пример на поиск множественных соответствий. *Регулярное выражение* также не распознает эти слова. Обратите внимание на точку в *регулярном выражении*, которая соответствует любому символу, за исключением символа конца строки. Все слова в строке поиска - кок, кук, кот и другие - будут удовлетворять шаблону, так что в результате поиска найдется множество соответствий.

```
Console.WriteLine("кок и кук");
strpat="(т|к).(т|к)";
str="кок тот кук тут как кот";
FindMatches(str, strpat);
```

Результаты работы этого фрагмента кода – на рис.1.10.

```
кок и кук
Строка =кок тот кук тут как кот Образец=(т|к).(т|к)
Число совпадений =6
Index = 0 Value = кок, Length =3
Index = 4 Value = тот, Length =3
Index = 8 Value = кук, Length =3
Index = 12 Value = тут, Length =3
Index = 16 Value = как, Length =3
Index = 20 Value = кот, Length =3
```

Рис. 1.10. Регулярные выражения. Пример "кок и кук"

Пример "обратные ссылки"

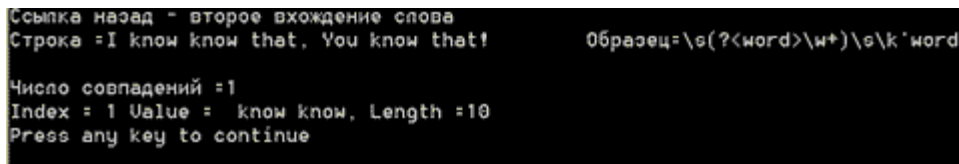
В этом примере рассматривается возможность задания в *регулярном выражении* обратных ссылок. Можно ли описать с помощью *регулярных выражений* язык, в котором встречаются две подряд идущие одинаковые подстроки? Ответ на этот вопрос отрицательный, поскольку грамматика такого языка должна быть контекстно-зависимой, и нужна память, чтобы хранить уже распознанные части строки. Аппарат *регулярных выражений*, предоставляемый классами *пространства RegularExpression*, тем не менее, позволяет решить эту задачу. Причина в том, что расширение стандартных *регулярных выражений* в Net Framework является не только синтаксическим. Содержательные расширения связаны с введением понятия *группы*, которой отводится память и дается имя. Это и дает возможность ссылаться на уже созданные *группы*, что и делает грамматику языка контекстно-зависимой. Ссылка на ранее полученную

группу называется *обратной ссылкой*. Признаком *обратной ссылки* является пара символов "`\k`", после которой идет имя *группы*.

Пример:

```
Console.WriteLine("Ссылка назад - второе вхождение слова");
strpat = @"\s(?:<word>\w+)\s\k'word'";
str = "I know know that, You know that!";
FindMatches(str, strpat);
```

Рассмотрим более подробно *регулярное выражение*, заданное строкой `strpat`. В *группе*, заданной скобочным выражением, после знака вопроса идет имя *группы* "`word`", взятое в угловые скобки. После имени *группы* идет шаблон, описывающий данную *группу*, в нашем примере шаблон задается произвольным идентификатором "`\w+`". В дальнейшем описании шаблона задается ссылка на *группу* с именем "`word`". Здесь имя *группы* заключено в одинарные кавычки. Поиск успешно справился с поставленной задачей (рис.1.11).



```
Ссылка назад - второе вхождение слова
Строка = I know know that, You know that!      Образец=\s(?:<word>\w+)\s\k'word'

Число совпадений = 1
Index = 1 Value = know know, Length = 10
Press any key to continue
```

Рис. 1.11. Регулярные выражения. Пример "обратные ссылки"

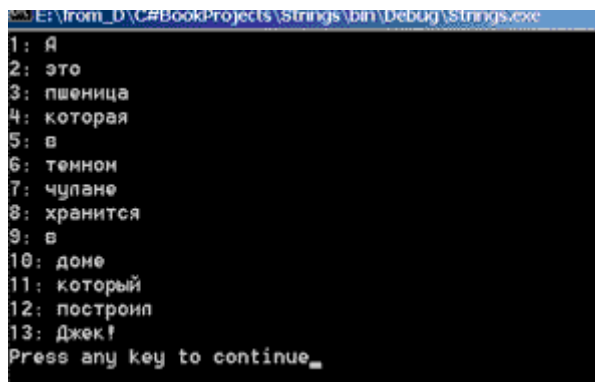
Пример "Дом Джека"

Вернемся к задаче разбора предложения на элементы. В классе `string` для этого имеется метод `Split`, который и решает поставленную задачу. Однако у этого метода есть существенный недостаток, - он не справляется с идущими подряд разделителями и создает для таких пар пустые слова. Метод `Split` класса `Regex` лишен этих недостатков, в качестве разделителей можно задавать любую пару символов, произвольное число пробелов и другие комбинации символов.

Повторим прежний пример:

```
public void TestParsing()
{
    string str, strpat;
    //разбор предложения - создание массива слов
    str = "А это пшеница, которая в темном чулане хранится, "+" в доме, который построил Джек!";
    strpat = " +|, +";
    Regex pat = new Regex(strpat);
    string[] words;
    words = pat.Split(str);
    int i=1;
    foreach(string word in words)
        Console.WriteLine("{0}: {1}",i++,word);
} //TestParsing
```

Регулярное выражение, заданное строкой `strpat`, определяет множество разделителей. В качестве разделителя задан пробел, повторенный сколь угодно много раз, либо пара символов - запятая и пробел. Разделители задаются *регулярными выражениями*. Метод `Split` применяется к объекту `pat` класса `Regex`. В качестве аргумента методу передается текст, подлежащий расщеплению. Вот как выглядит массив слов после применения метода `Split` (рис.1.12).



```
1: А
2: это
3: пшеница
4: которая
5: в
6: темном
7: чулане
8: хранится
9: в
10: доме
11: который
12: построил
13: Джек!
Press any key to continue_
```

Рис. 1.12. Регулярные выражения. Пример "Дом Джека"

Пример "Атрибуты"

Регулярные выражения особенно хороши при разборе сложных текстов. Примерами таковых могут быть различные справочники, различные текстовые базы данных, весьма популярные теперь XML-документы, разбором которых приходится заниматься.

В качестве примера рассмотрим структурированный документ, строки которого содержат некоторые атрибуты, например, телефон, адрес и e-mail. Структуру документа можно задавать по-разному; будем предполагать, что каждый атрибут задается парой "имя: Значение". Задача состоит в том, чтобы выделить из строки соответствующие атрибуты. В таких ситуациях *регулярное выражение* удобно задавать в виде *групп*, где каждая *группа* соответствует одному атрибуту.

Начальный фрагмент кода процедуры, в котором описываются строки текста и образцы поиска:

```
public void TestAttributes()
{
    string s1 = "tel: (831-2) 94-20-55 ";
    string s2 = "Адрес: 117926, Москва, 5-й Донской проезд, стр.10, кв.7 ";
    string s3 = "e-mail: Valentin.Berestov@tverorg.ru ";
    string s4 = s1 + s2 + s3;
    string s5 = s2 + s1 + s3;
    string pat1 = @"tel:\s(?:<tel>\((\d|-)*\)\s(\d|-)+\s";
    string pat2 = @"Адрес:\s(?:<addr>[0-9А-Яа-я \-\.]+)\s";
    string pat3 = @"e-mail:\s(?:<em>[a-zA-Z\.\@ ]+)\s";
    string compat = pat1 + pat2 + pat3;
    string tel = "", addr = "", em = "";
```

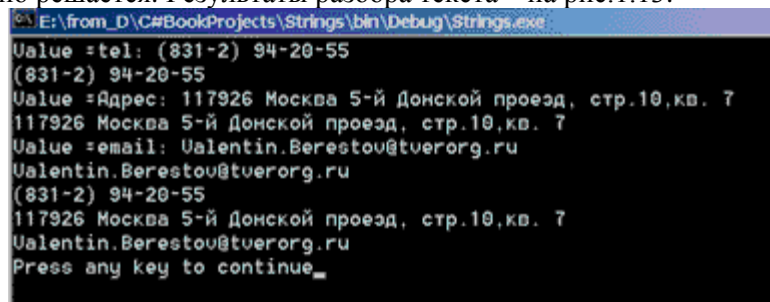
Строки *s4* и *s5* представляют строку разбираемого документа. Их две, для того чтобы можно было проводить эксперименты, когда атрибуты в документе представлены в произвольном порядке. Каждая из строк *pat1*, *pat2*, *pat3* задает одну именованную *группу* в *регулярном выражении*, имена *групп* - *tel*, *Адрес*, *e-mail* - даются в соответствии со смыслом атрибутов. Сами шаблоны подробно не описаны, сделаем лишь одно замечание. Например, шаблон телефона исходит из того, что номеру предшествует код, заключенный в круглые скобки. Поскольку сами скобки играют особую роль, то для задания скобки как символа используется пара - "\("". Это же касается и многих других символов, используемых в шаблонах, - точки, дефиса и т.п. Строка *compat* представляет составное *регулярное выражение*, содержащее все три *группы*. Строки *tel*, *addr* и *em* нам понадобятся для размещения в них результатов разбора. Применим вначале к строкам *s4* и *s5* каждый из шаблонов *pat1*, *pat2*, *pat3* в отдельности и выделим соответствующий атрибут из строки. Код, выполняющий эти операции:

```
Regex reg1 = new Regex(pat1);
Match match1 = reg1.Match(s4);
Console.WriteLine("Value =" + match1.Value);
tel = match1.Groups["tel"].Value;
Console.WriteLine(tel);
Regex reg2 = new Regex(pat2);
Match match2 = reg2.Match(s5);
Console.WriteLine("Value =" + match2.Value);
addr = match2.Groups["addr"].Value;
Console.WriteLine(addr);
Regex reg3 = new Regex(pat3);
Match match3 = reg3.Match(s5);
Console.WriteLine("Value =" + match3.Value);
em = match3.Groups["em"].Value;
Console.WriteLine(em);
```

Все выполняется нужным образом - создаются именованные *группы*, к ним можно получить доступ и извлечь найденный значения атрибутов. Решим ту же задачу одним махом, используя составной шаблон *compat*:

```
Regex comreg = new Regex(compat);
Match commatch = comreg.Match(s4);
tel = commatch.Groups["tel"].Value;
Console.WriteLine(tel);
addr = commatch.Groups["addr"].Value;
Console.WriteLine(addr);
em = commatch.Groups["em"].Value;
Console.WriteLine(em);
} // TestAttributes
```

И эта задача успешно решается. Результаты разбора текста – на рис.1.13.



```
E:\from_D\CABookProjects\Strings\bin\Debug\Strings.exe
Value =tel: (831-2) 94-20-55
(831-2) 94-20-55
Value =Адрес: 117926 Москва 5-й Донской проезд, стр.10, кв. 7
117926 Москва 5-й Донской проезд, стр.10, кв. 7
Value =email: Valentin.Berestov@tverorg.ru
Valentin.Berestov@tverorg.ru
(831-2) 94-20-55
117926 Москва 5-й Донской проезд, стр.10, кв. 7
Valentin.Berestov@tverorg.ru
Press any key to continue_
```

Рис. 1.13. Регулярные выражения. Пример "Атрибуты"