Регулярные выражения

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,

Немного теории

- Пусть T = {a1, a2, ...an} - алфавит символов. Словом в алфавите T называется последовательность записанных подряд символов, а длиной слова - число его символов.
- Пустое слово, не содержащее символов, обычно обозначается как e.
- Алфавит T можно рассматривать как множество всех слов длины 1.
- Рассмотрим операцию конкатенации над множествами, так, что конкатенация алфавита T с самим собой дает множество всех слов длины 2.

- Обозначается конкатенация TT как T2. Множество всех слов длины k обозначается - Tk, его можно рассматривать как k-кратную конкатенацию алфавита T.
- Множество всех непустых слов произвольной длины, полученное объединением всех множеств Tk, обозначается T+, а объединение этого множества с пустым словом называется итерацией языка и обозначается T*.
- Итерация описывает все возможные слова, которые можно построить в данном алфавите. Любое подмножество слов L(T), содержащееся в T*, называется языком в алфавите T.

- Определим класс языков, задаваемых регулярными множествами. Регулярное множество определяется рекурсивно следующими правилами:
- пустое множество, а также множество, содержащее пустое слово, и одноэлементные множества, содержащие символы алфавита, являются регулярными базисными множествами;
- если множества P и Q являются регулярными, то множества, построенные применением операций объединения, конкатенации и итерации – P+Q, PQ, P*, Q* - тоже являются регулярными

- Регулярные выражения представляют удобный способ задания регулярных множеств. Аналогично множествам, они определяются рекурсивно:
- регулярные базисные выражения задаются символами и определяют соответствующие регулярные базисные множества, например, выражение f задает одноэлементное множество {f} при условии, что f - символ алфавита T;
- если p и q - регулярные выражения, то операции объединения, конкатенации и итерации - p+q, pq, p*, q* - являются регулярными выражениями, определяющими соответствующие регулярные множества.

- По сути, регулярные выражения - это более простой и удобный способ записи регулярных множеств в виде обычной строки.
- Каждое регулярное множество, a, следовательно, и каждое регулярное выражение задает некоторый язык L(T) в алфавите T.
- Более важно, что для любого регулярного выражения можно построить конечный автомат, который распознает, принадлежит ли заданное слово языку, порожденному регулярным выражением.
- На этом основана практическая ценность регулярных выражений.

- С точки зрения практика 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, крайне полезны для практиков, избавляя программиста от ненужных сложностей.
- (В Net Framework эти усложнения чрезмерны. Выигрывая в мощности языка, проигрываем в простоте записи его выражений.)

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. Приведем таблицу, в которой 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. Она позволяет дать общее представление о синтаксисе, которое будет дополнено большим числом примеров. За деталями придется обращаться к справочной системе.

Символ	Интерпретация
	Категория: escape-последовательности
\b	При использовании его в квадратных скобках соответствует символу "обратная косая черта" с кодом - \u0008
\t	Соответствует символу табуляции \u0009
\r	Соответствует символу возврата каретки \u000D
\n	Соответствует символу новой строки \u000A
\e	Соответствует символу escape \u001B
\040	Соответствует символу ASCII, заданному кодом до трех цифр в восьмеричной системе
\x20	Соответствует символу ASCII, заданному кодом из двух цифр в шестнадцатиричной системе
\u0020	Соответствует символу Unicode, заданному кодом из четырех цифр в шестнадцатиричной системе

	Категория: подмножества (классы) символов
.	Соответствует любому символу, за исключением символа конца строки
[aeiou]	Соответствует любому символу из множества, заданного в квадратных скобках
[^aeiou]	Отрицание. Соответствует любому символу, за исключением символов, заданных в квадратных скобках
[0-9a-fA-F]	Задание диапазона символов, упорядоченных по коду. Так, 0-9 задает любую цифру
\p{name}	Соответствует любому символу, заданному множеству с именем name, например, имя Ll задает множество букв латиницы в нижнем регистре. Поскольку все символы разбиты на подмножества, задаваемые категорией Unicode, то в качестве имени можно задавать имя категории
\P{name}	Отрицание. Большая буква всегда задает отрицание множества, заданного малой буквой
\w	Множество символов, используемых при задании идентификаторов - большие и малые символы латиницы, цифры и знак подчеркивания
\d	Соответствует любому символу из множества цифр

	Категория: Операции (модификаторы)
*	Итерация. Задает ноль или более соответствий; например, \w*
(abc)*.	Аналогично, {0,}
+	Положительная итерация. Задает одно или более соответствий; например, \w+ или (abc)+. Аналогично, {1,}
?	Задает ноль или одно соответствие; например, \w? или (abc)?. Аналогично, {0,1}
{n}	Задает в точности n соответствий; например, \w{2}
{n,}	Задает, по меньшей мере, n соответствий; например, (abc){2,}
{n,m}	Задает, по меньшей мере, n, но не более m соответствий; например, (abc){2,5}

	Категория: Группирование
(?<Name>)	При обнаружении соответствия выражению, заданному в круглых скобках, создается именованная группа, которой дается имя Name. Например, (?<tel> \d{7}). При обнаружении последовательности из семи цифр будет создана группа с именем tel
()	Круглые скобки разбивают регулярное выражение на группы. Для каждого подвыражения, заключенного в круглые скобки, создается группа, автоматически получающая номер. Номера следуют в обратном порядке, поэтому полному регулярному выражению соответствует группа с номером 0
(?imnsx)	Включает или выключает в группе любую из пяти возможных опций. Для выключения опции перед ней ставится знак минус. Например, (?i-s:) включает опцию i, задающую нечувствительность к регистру, и выключает опцию s - статус single-line

Классы пространства RegularExpressions

- В данном пространстве расположено семейство из одного перечисления и восьми связанных между собой классов.
- **Класс Regex**
- **Классы Match и MatchCollection**
- **Классы Group и GroupCollection**
- **Классы Capture и CaptureCollection**
- **Перечисление RegexOptions**
- **Класс RegexCompilationInfo**

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.

- Для описания регулярного выражения в классе определено несколько перегруженных конструкторов:
- **Regex()** – создает пустое выражение;
- **Regex(String)** – создает заданное выражение;
- **Regex(String, RegexOptions)** – создает заданное выражение и задает параметры для его обработки с помощью элементов перечисления RegexOptions (например, различать или нет прописные и строчные буквы).

Основные методы 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.

Следующий пример позволяет найти все номера телефонов в указанном фрагменте текста:

```
static void Main()
    {
        Regex r = new Regex(@"\d{2,3}(-\d\d){2}");
        string text = @"Контакты в Москве tel:123-45-67,
    123-34-56; fax:123-56-45 Контакты в Саратове
    tel:12-34-56; fax:12-56-45";
        Match tel = r.Match(text);
        while (tel.Success)
        {
            Console.WriteLine(tel);
            tel = tel.NextMatch();
        }
    }
```

Следующий пример позволяет подсчитать сумму целых чисел, встречающихся в тексте:

```
static void Main()
      {
         Regex r = new Regex(@"[-+]?\d+");
         string text = @"5*10=50 -80/40=-2";
         Match teg = r.Match(text);
         int sum = 0;
         while (teg.Success)
         {
            Console.WriteLine(teg);
            sum += int.Parse(teg.ToString());
            teg = teg.NextMatch();
         }
         Console.WriteLine("sum=" + sum);
    }
```

- Метод ***IsMatch*** возвращает true, если фрагмент, соответствующий выражению, в заданной строке найден, и false в противном случае.
- Метод ***Replace*** в указанной входной строке заменяет строки, соответствующие шаблону регулярного выражения, указанной строкой замены.

Например, попытаемся определить, встречается ли в заданном тексте слово *собака*:

```
static void Main()
      {
         Regex r = new Regex("собака",RegexOptions.IgnoreCase);
         string text1 = "Кот в доме, собака в конуре.";
         string text2 = "Котик в доме, собачка в конуре.";
         Console.WriteLine(r.IsMatch(text1));
         Console.WriteLine(r.IsMatch(text2));
       }
```

Замечание. RegexOptions.IgnoreCase – означает, что регулярное выражение применяется без учеба регистра символов

Можно использовать конструкцию выбора из нескольких элементов. Варианты выбора перечисляются через вертикальную черту. Например, попытаемся определить, встречается ли в заданном тексте слов *собака* или *кот*:

```
static void Main(string[] args)
    {
        Regex r = new Regex("собака|кот",
        RegexOptions.IgnoreCase);
        string text1 = "Кот в доме, собака в конуре.";
        string text2 = "Котик в доме, собачка в конуре.";
        Console.WriteLine(r.IsMatch(text1));
        Console.WriteLine(r.IsMatch(text2));
    }
```

Попытаемся определить, есть ли в заданных строках номера телефона в формате xx-xx-xx или xxx-xx-xx:

```
static void Main()
        {
            Regex r = new Regex(@"\d{2,3}(-\d\d){2}");
            string text1 = "tel:123-45-67";
            string text2 = "tel:no";
            string text3 = "tel:12-34-56";
            Console.WriteLine(r.IsMatch(text1));
            Console.WriteLine(r.IsMatch(text2));
            Console.WriteLine(r.IsMatch(text3));
         }
```

Изменение номеров телефонов:

```
static void Main(string[] args)
    {
    string text = @"Контакты в Москве
tel:123-45-67, 123-34-56; fax:123-56-45.
Контакты в Саратове tel:12-34-56;
fax:11-56-45";
    Console.WriteLine("Старые данные\n"+text);
    string newText=Regex.Replace(text, "123-",
"890-");
    Console.WriteLine("Новые данные\n" +
newText);
    }
```

Удаление всех номеров телефонов из текста:

```
static void Main()
     {
     string text = @"Контакты в Москве tel:123-45-67,
    123-34-56; fax:123-56-45.
    Контакты в Саратове tel:12-34-56; fax:12-56-45";
     Console.WriteLine("Старые данные\n"+text);
     string newText=Regex.Replace(text,
    @"\d{2,3}(-\d\d){2}", "");
     Console.WriteLine("Новые данные\n" + newText);
     }
}
```

- **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.

Разбиение исходного текста на фрагменты:

```
static void Main()
{
     string text = @"Контакты в Москве tel:123-45-67,
    123-34-56; fax:123-56-45.
                  Контакты в Саратове tel:12-34-56;
    fax:12-56-45";
     string []newText=Regex.Split(text,"[ ,.:;]+");
     foreach( string a in newText)
          Console.WriteLine(a);
}
```

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:

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.

Рассмотрим 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

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.

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.

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.

Примеры работы с регулярными выражениями

функции FindMatch, которая производит поиск первого вхождения подстроки, соответствующей образцу:

```
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
```

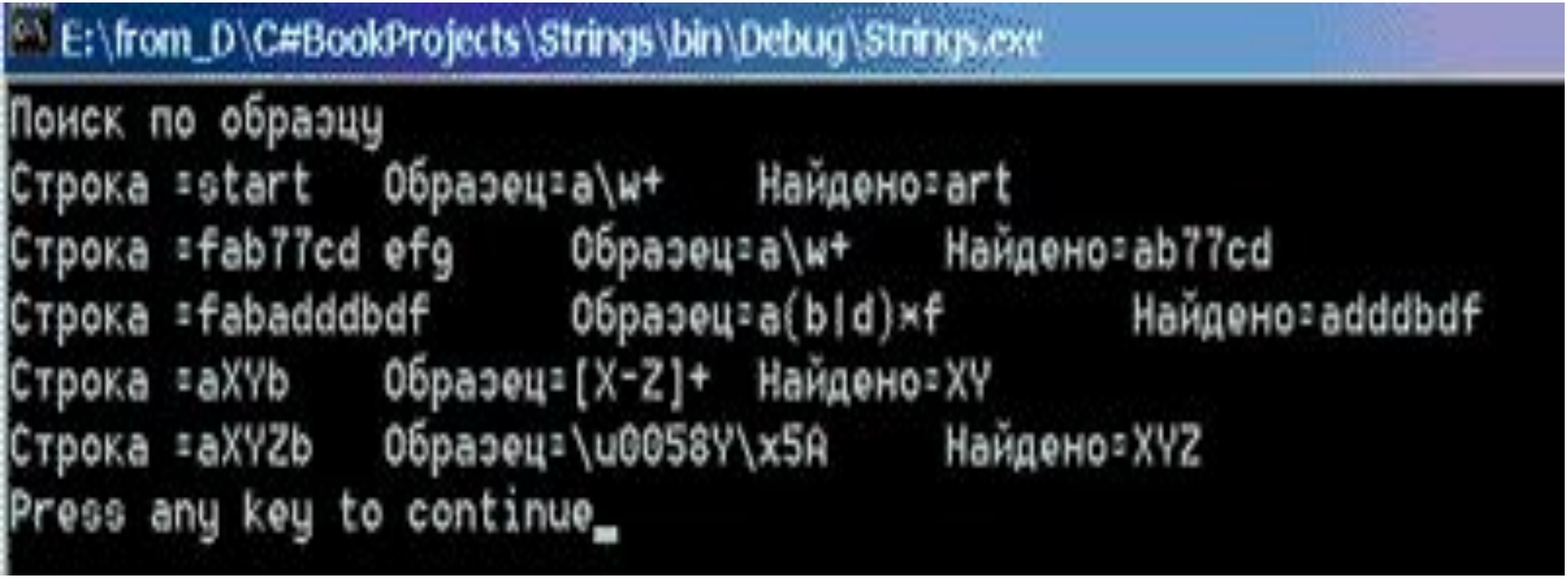
- В качестве входных аргументов функции передается строка str, в которой ищется вхождение, и строка strpat, задающая образец - регулярное выражение.
- Функция возвращает найденную в результате поиска подстроку. Если соответствия нет, то возвращается пустая строка. Функция начинает свою работу с создания 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, не забудьте добавить в начало проекта предложение: using System.Text.RegularExpressions.)

```
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
```

- Регулярные выражения задаются @-константами.
- В первом образце используется последовательность символов \w+, обозначающая, 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 шестнадцатиричной кодировке).

Результаты, полученные при работе этой процедуры.



Процедура FindMatches, позволяющую найти все вхождения образца в заданный текст:

```
void FindMatches(string str, string strpat)
{
 Regex pat = new Regex(strpat);
 MatchCollection matchcol =pat.Matches(str);
 Console.WriteLine("Строка
    ={0}\tОбразец={1}",str,strpat);
 Console.WriteLine("Число совпадений
    ={0}",matchcol.Count);
 foreach(Match match in matchcol)
 Console.WriteLine("Index = {0} Value = {1}, Length ={2}",
 match.Index, match.Value, match.Length);
}//FindMatches
```

- Входные аргументы у процедуры те же, что и у функции FindMatch, ищущей первое вхождение.
- Выполнение процедуры, так же, как и в FindMatch, начинается с создания объекта pat класса Regex, конструктору которого передается регулярное выражение.
- Класс Regex, так же, как и класс String, относится к неизменяемым (immutable) классам, поэтому для каждого нового образца нужно создавать новый объект pat.
- В отличие от FindMatch, объект pat вызывает метод Matches, который определяет все вхождения подстрок, удовлетворяющих образцу, в заданный текст.
- Результатом выполнения метода Matches является автоматически создаваемый объект класса MatchCollection, хранящий коллекцию объектов уже известного нам класса Match, каждый из которых задает очередное вхождение.

Пример "око и рококо"

- Следующий образец в нашем 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= "012345123612378123 9";
FindMatches(str, strpat);
```

Результаты поисков

```
око и рококо
Строка =рококо  Образец=око
Число совпадений =1
Index = 1 Value = око, Length =3
Строка =012345123612378123 9     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
```

Пример "кок и кук»

- 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

```
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
```