Регулярные выражения в .NET

Параграф 1. Введение в использование регулярных выражений в .NET

Регулярное выражение (regular expression, regexp) - это не новый язык, а стандарт для поиска и замены текста в строках. Существует два стандарта: основные регулярные выражения (BRE - basic regular expressions) и расширенные регулярные выражения (ERE - extended regular expressions). ERE включает все функциональные возможности BRE.

Наиболее развита поддержка регулярных выражений, до появления .Net, была в Perl (выполнялась на уровне интерпретатора). В настоящее время регулярные выражения поддерживаются практически всеми новыми языками программирования. Обработка регулярных выражений в .Net, хотя и не является встроенной, но сами регулярные выражения почти полностью аналогичны регулярным выражениям в Perl и имеют ряд дополнительных возможностей.

Существует два типа механизмов выполнения регулярных выражений: механизм детерминированных конечных автоматов и механизм не детерминированных конечных автоматов. Первый работает быстрее, но не поддерживает сохранение, позиционные проверки и минимальные квантификаторы. Net полностью поддерживает традиционный механизм недетерминированных конечных автоматов (не Posix совместимый). Его принцип действия заключается в последовательном сравнении каждого элемента регулярного выражения с входной строкой и запоминании найденных позиций совпадений. При неудачном дальнейшем поиске выполняется возврат к сохраненным позициям. В заключении, перебираются альтернативные варианты совпадений и выбираются ближайшие к левой границе точки начала поиска (в отличие от Posix совместимого, выбирающего максимально возможное).

Поддержка регулярных выражений в .Net выполняется классами пространства имен:

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

Основные классы:

- Regex - постоянное регулярное выражение.
- Match - предоставляет результаты очередного применения всего регулярного выражения к исходной строке.
- MatchCollection - предоставляет набор успешных сопоставлений, при итеративном применении образца регулярного выражения к строке.
- Capture - предоставляет результаты отдельного захвата подвыражения.

- Group - предоставляет результаты для одной регулярной группы.
- GroupCollection - предоставляет коллекцию найденных групп и возвращает набор групп как одно соответствие.
- CaptureCollection - предоставляет последовательность найденных подстрок и возвращает наборы соответствий отдельно для каждой группы.

Более подробно на использовании классов мы остановимся ниже, после рассмотрения синтаксиса регулярных выражений.

Параграф 2. Основные элементы синтаксиса регулярных выражений

Элементарные примеры в таблице даны для подстановки строки и регулярного выражения в две функции обработки нажатия кнопок 1 и 2, в которых одновременно демонстрируется и использование классов Match и MatchCollection. Для упрощения кода примеров, вывод результатов выполняется в заголовок формы:

Код 1. Функция для подстановки значений regex и строки s из таблицы 1:

```
private void button1_Click(object sender, EventArgs e)
{
 string s = Строка подстановки из таблицы 1 код 1;
 Regex regex = new Regex(Регулярное выражение из таблицы 1 код 1);
 Match match = regex.Match(s);
 Text = "";
 if (match.Success)
 {
  for (int i = 0; i < match.Groups.Count; i++)
  {
   Text += match.Groups[0].Value.ToString() + " ";
  }
 }
}
```

Код 2. Функция для подстановки значений regex и строки s из таблицы 1:

```
private void button2_Click(object sender, EventArgs e)
{
 string s = Строка подстановки из таблицы 1 код 2;
 Regex regex = new Regex(Регулярное выражение из таблицы 1 код 2);
 MatchCollection matchcollection = regex.Matches(s);
 Text = "";
 for (int i = 0; i < matchcollection.Count; i++)
 {
  Text += matchcollection[i].Value+ " ";
 }
}
```

Таблица 1.

Символ	Значение	Примеры применения
.	Любой одиночный символ, кроме символа новой строки \n.	Regex("a.b"); Код 1 Строка s = "a1ba2ba3ba4b"; Результат: a1b Код 2 Строка s = "a1ba2ba3ba4b"; Результат: a1b a2b a3b a4b
\	Определение метасимвола	Символ "." соответствует любому символу, а "\." Regex(@"a\.b"); в предыдущем примере ничего не найдет Код 1 Строка s = "a1ba2ba3ba.b"; Результат: a.b Код 2 Строка s = "a1ba.ba3ba.b"; Результат: a.b a.b
|	Разделение шаблона (или)	Regex("a|b"); Код 1 Строка s = "abcabcabcabc"; Результат: a Код 2 Строка s = "abcabcabcabc"; Результат: a b a b a b a b
[мн-во сим-в]	Соответствует любому символу из данного множества (одному и только одному)	Regex("[abc]"); Код 1 Строка s = "abcdabcdabcdabcd"; Результат: a Код 2 Строка s = "abcdabcdabcdabcd"; Результат: a b c a b c a b c a b c
[^мн-во сим-в]	Отрицание множества символов	Regex("[^abc]"); Код 1 Строка s = "abcdabcdabcdabcd"; Результат: d Код 2 Строка s = "abcdabcdabcdabcd"; Результат: d d d d
^	Соответствует началу строки Отрицание - если первый в квадратной скобке	Regex("^abc"); Код 1 Строка s = "abcdefabcdef"; Результат: abc Код 2 Строка s = "abcdefabcdef"; Результат: abc Regex("^bc"); в обоих примерах ничего не найдет
$	Соответствует концу строки. Это конец строки или позиция перед символом начала новой строки.	Regex("def$"); Код 1 Строка s = "abcdefabcdef"; Результат: def Код 2 Строка s = "abcdefabcdef"; Результат: def
(....)	Группировка элементов.	Regex("(ab)"); Код 1 Строка s = "abcdefabcdef"; Результат: ab ab Код 2 Строка s = "abcdefabcdef"; Результат: ab ab
?	Повторение 0 или 1 раз стоящего перед	Regex("ab?"); Код 1 Строка s = "aabbbcde"; Результат: a

		Код 2 Строка s = "aabbbcde"; Результат: a ab --- Regex("a?b?"); Код 1 Строка s = "aabbbcde"; Результат: a Код 2 Строка s = "aabbbcde"; Результат: a ab b b --- Regex("a?b"); Код 1 Строка s = "aabbbcde"; Результат: ab Код 2 Строка s = "aabbbcde"; Результат: ab b b
*****	Повторение 0 или более раз стоящего перед	Regex("ab*"); Код 1 Строка s = "aabbbcde"; Результат: a Код 2 Строка s = "aabbbcde"; Результат: a abbb --- Regex("a*b*"); Код 1 Строка s = "aabbbcde"; Результат: aabbb Код 2 Строка s = "aabbbcde"; Результат: aabbb --- Regex("a*b"); Код 1 Строка s = "aabbbcde"; Результат: aab Код 2 Строка s = "aabbbcde"; Результат: aab b b
+	Повторение 1 или более раз стоящего перед ним	Regex("ab+"); Код 1 Строка s = "aabbbcde"; Результат: abbb Код 2 Строка s = "aabbbcde"; Результат: abbb --- Regex("a+b+"); Код 1 Строка s = "aabbbcde"; Результат: aabbb Код 2 Строка s = "aabbbcde"; Результат: aabbb --- Regex("a+b"); Код 1 Строка s = "aabbbcde"; Результат: aab Код 2 Строка s = "aabbbcde"; Результат: aab
{n}	Повторение точно n раз	Regex("ab{2}"); Код 1 Строка s = "aabbbcde"; Результат: abb Код 2 Строка s = "aabbbcde"; Результат: abb

{n,m}	повторение от n до m раз	Regex("ab{2,4}"); Код 1 Строка s = "abbbcdeabbbbbcde"; Результат: abbb Код 2 Строка s = "abbbcdeabbbbbcde"; Результат: abbb abbbb
{n,}	повторение n и более раз	Regex("ab{2,}"); Код 1 Строка s = "abbbcdeabbbbbcde"; Результат: abb Код 2 Строка s = "abbbcdeabbbbbcde"; Результат: abbb abbbbb
***?**	повторение 0 или более раз, минимально возможное количество	Regex("ab*?c"); Код 1 Строка s = "abbbcdeabbbbbcde"; Результат: abbbc Код 2 Строка s = "abbbcdeabbbbbcde"; Результат: abbbc abbbbbc
+?	повторение 1 или более раз, минимально возможное количество	Regex("ab+?c"); Код 1 Строка s = "abbbcdeabbbbbcde"; Результат: abbbc Код 2 Строка s = "abbbcdeabbbbbcde"; Результат: abbbc abbbbbc
??	повторение 0 или 1 раз, минимально возможное количество	Regex("ab??c"); Код 1 Строка s = "acdeabcdeabbcde"; Результат: ac Код 2 Строка s = "acdeabcdeabbcde"; Результат: abc
\w	Слово (цифра или буква)	Regex(@"\w"); Код 1 Строка s = "abcd_~`!@#$%^&*()- =+\|:;\",.<>?/12345"; Результат: A Код 2 Строка s = "abcd_~`!@#$%^&*()- =+|:;\",.<>?/12345"; Результат: A Ф b c d _ 1 2 3 4 5
\W	Не слово (не цифра и не буква)	Regex(@"\W"); Код 1 Строка s = "abcd_~`!@#$%^&*()- =+\|:;\",.<>?/12345"; Результат: ~ Код 2 Строка s = "abcd_~`!@#$%^&*()- =+|:;\",.<>?/12345"; Результат: ~ ` ! @ # $ % ^ & * ( ) - = + | : ; " , . < > ? /
\d	Десятичная цифра	Regex(@"\d"); Код 1 Строка s = "abcd12345"; Результат: 1 Код 2 Строка s = "abcd12345"; Результат: 1 1 2 3 4 5
\D	Не десятичная цифра	Regex(@"\D"); Код 1 Строка s = "abcd12345"; Результат: a Код 2 Строка s = "abcd12345"; Результат: a b c d _ ~ ` ! @ # $ % ^ & * ( ) - = + | : ; " , . < > ? /

\s	Пустое место (пробел, \f, \n, \r, \t, \v)	Regex(@"\s"); Код 1 Строка s = "ab cd"; Результат: b c Код 2 Строка s = "ab cd"; Результат: b c
\S	Не пустое место (не пробел, не \f, не \n, не \r, не \t, не \v)	Regex(@"\S"); Код 1 Строка s = "ab c\td"; Результат: a Код 2 Строка s = "ab c\td"; Результат: a b c d

\b	Граница слова (Символы для поиска пишутся:
	для начала слова - справа;
	для конца - слева.)
		Regex(@"\bqw"); //ничего не будет найдено при Regex(@"\bwe");
		Код 1
		Строка s = "abcde qwerty qwerty"; Результат: qw
		Код 2
		Строка s = "abcde"; Результат: qw qw
		--- 	
		Regex(@"ty\b"); //ничего не будет найдено при Regex(@"rt\b");
		Код 1
		Строка s = "abcde qwerty qwerty"; Результат: ty
		Код 2
		Строка s = "abcde qwerty qwerty"; Результат: ty ty

\B	Не граница слова (Символы для поиска пишутся:
	для начала слова - справа;
	для конца - слева.)
		Regex(@"\Bwe"); //ничего не будет найдено при Regex(@"\qw");
		Код 1
		Строка s = "abcde qwerty qwerty"; Результат: we
		Код 2
		Строка s = "abcde"; Результат: we we
		--- 	
		Regex(@"rt\B"); //ничего не будет найдено при Regex(@"ty\B");
		Код 1
		Строка s = "abcde qwerty qwerty"; Результат: rt
		Код 2
		Строка s = "abcde qwerty qwerty"; Результат: rt rt

\A	"Истинное" начало строки	Regex(@"ab\A"); //ничего не будет найдено при Regex(@"qw\A");
		Код 1
		Строка s = "abcde qwerty qwerty"; Результат: ab
		Код 2
		Строка s = "abcde qwerty qwerty"; Результат: ab

\Z	Конец строки, позиция перед символом начала новой строки.	Regex(@"ty\Z"); Код 1 Строка s = "abcde qwerty qwerty\n"; Результат: ty Код 2 Строка s = "abcde qwerty qwerty"; Результат: ty
\z	"Истинный" конец строки, позиция перед символом начала новой строки.	Regex(@"ty\n\z"); Код 1 Строка s = "abcde qwerty qwerty\n"; Результат: ty + нечитаемый символ Код 2

<table>
<tr><td></td><td></td><td>Строка s = "abcde qwerty qwerty"; Результат: ty + нечитаемый символ</td></tr>
<tr><td>\G</td><td>Непрерывное совпадение от конца предыдущего
Работает только с match.NextMatch() при использовании класса Match.</td><td>Regex(@"\G\d{3}");
Код 1 должен быть изменен
<pre>while(match.Success)
{
 for(int i = 0; i < match.Groups.Count; i++)
 {
  Text += match.Groups[0].Value.ToString() + " ";
 }
 match = match.NextMatch();
}</pre>
Строка s = "123456789a123456789"; Результат: 123 456 789
Код 2
Строка s = "123456789a123456789"; Результат: 123 456 789</td></tr>
<tr><td>(?=...)</td><td>Позитивная опережающая проверка
Здесь:
? определяет, что в скобках действие над группировками, а не группировка для поиска;
= позитивная опережающая проверка или то, что должно обязательно идти после подстроки, которую мы ищем.</td><td rowspan="3">Пример 1:
<pre>private void button3_Click(object sender, EventArgs e)
{
 StringBuilder mystringbuilder = new StringBuilder();
 mystringbuilder.Append("<html><head><title>");
 mystringbuilder.Append("Регулярные выражения ");
 mystringbuilder.Append("в .NET</TITLE></head><body>");
 mystringbuilder.Append(" Первый абзац текста\n ");
 mystringbuilder.Append(" Второй абзац текста\n");
 mystringbuilder.Append("его вторая половина ");
 mystringbuilder.Append(" Третий абзац текста ");
 mystringbuilder.Append("</body></html>");
 Regex regex = new Regex(@"(?<=<title>).*(?=</title>)",
 RegexOptions.IgnoreCase | RegexOptions.Singleline
  | RegexOptions.ExplicitCapture);
 try
 {
  MatchCollection matchcollection =
    regex.Matches(mystringbuilder.ToString());
  //matchcollection = regex.Matches(s);
  Text = ""; textBox1.Text ="";
  for (int i = 0; i < matchcollection.Count; i++)
  {
   Text += matchcollection[i].Value;
  }
 }
 catch (Exception ex)
 {
  Text=ex.Message;
 }
}</pre>
Результат:</td></tr>
<tr><td>(?!...)</td><td>Негативная опережающая проверка
Здесь:
? определяет, что в скобках действие над группировками, а не группировка для поиска;
= негативная опережающая проверка или то, что ни в коем случае не должно стоять после искомой строки.</td></tr>
<tr><td>(?<=...)</td><td>Позитивная ретроспективная проверка</td></tr>
</table>

	Здесь:
	? определяет, что в скобках действие над группировками, а не группировка для поиска;
	<= позитивная ретроспективная проверка или то, что должно обязательно идти перед подстрокой, которую мы ищем.
		Регулярные выражения в .NET
		
		Пример 2. Извлекаем все что в текстовой части в тэгах P:
		....код создание HTML из предыдущего примера .....
		Regex regex = new Regex
		(@"(?<=)(?<name1>[^( )]*)(?=)",
		RegexOptions.IgnoreCase | RegexOptions.ExplicitCapture
		| RegexOptions.Singleline);
		try
		{
		MatchCollection matchcollection =
		regex.Matches(mystringbuilder.ToString());
		for(int i = 0; i < matchcollection.Count; i++)
		{
		Text += matchcollection[i].Result("${name1}") + " ";
		}
		}
		catch (Exception ex)
		{
		textBox1.Text=ex.Message;
		}
		}
		Результат (если сохранить как текстовый файл):
		Первый абзац текста
		Второй абзац текста
		его вторая половина
		Третий абзац текста

(?<!...)	Негативная ретроспективная проверка Здесь: ? определяет, что в скобках действие над группировками, а не группировка для поиска; ! - негативная ретроспективная проверка или то, что ни в коем случае не должно идти перед подстрокой, которую мы ищем.	

В регулярных выражениях допускаются метасимволы: \t - горизонтальная табуляция, \r - возврат курсора, \n - перевод строки, \v - вертикальная табуляция, \e - Escape, \f - подача страница, \0AA - символ представленный двумя цифрами (AA) восьмеричного кода, \xAA - символ представленный двумя цифрами (AA) шестнадцатеричного кода, \xAAAA- символ представленный четырьмя цифрами (AAAA) шестнадцатеричного кода, \a - звуковой сигнал.

Параграф 3. Приоритет групповых регулярных выражений:

Наименование	**Обозначение**
Круглые скобки	( )(?:...)
Множители	?, +, *, {m,n}
Последовательность и фиксация	abc,\A, \Z
Дизъюнкция	|

Параграф 4. Примеры использования групповых регулярных выражений

В примерах используется код button2_Click, приведенный в начале параграфа 2.

Извлечь адрес электронной почты из строки

```
string s = "Почта: wladm@narod.ru Сайт: http://wladm.narod.ru ";
Regex regex = new Regex(@"\b\w+([\.\w]+)*\w@\w((\.\w)*\w+)*\.\w{2,3}\b");
```

В регулярном выражении \b определяет начало слова, далее \w+ слово (цифра или буква) повторенные 1 или более раз. Следующую часть строки ([\.\w]+)*\w@ для данного примера можно было заменить на @, но адрес может иметь еще слова через точку до собаки. Поэтому точка или слово (в квадратных скобках) повторенные 1 или более раз и все это может быть повторено 0 или более раз. Следующий символ \w@ тоже можно сократить \w, но символ @ обязателен. Он якорный для распознавания почтового адреса. Далее идет практически повторение того, что было до символа @ - повторение слов с точкой и в конце слово от двух до трех символов (ru, com....).

Извлечь адрес сайта из строки

Для примера взяты 2 адреса, короткий и длинный - реальный адрес MSDN с описанием класса Match. Первый адрес также показывает, что классы успешно работают с национальными кодировками.

```
string s = "http://msdn.microsoft.com/иванов.html Сайт: ";
s+="http://msdn.microsoft.com/library/rus/default.asp?url=";
s+="/library/RUS/cpref/html/frlrfSystemTextRegularExpressionsMatchClassTopic.asp";
Regex regex =
new Regex(@"(\b\w+:\/\/\w+((\.\w)*\w+)*\.\w{2,3}(\/\w*|\.\w*|\?\w*\=\w*)*)");
```

Пример похож на предыдущий, отличие в начале: \b\w+:\/\/\ выделяет http:// и в конце: (\/\w*|\.\w*|\?\w*\=\w*)*) - перечислено с какими знаками могут повторяться буквы ("/",".","?","=") и все они объединены операцией или "|".

Результат:

```
http://msdn.microsoft.com/иванов.html
http://msdn.microsoft.com/library/rus/default.asp?url=
  /library/RUS/cpref/html/frlrfSystemTextRegularExpressionsMatchClassTopic.asp
```

Выделение цифр из строки

Результат:

```
string s = "Приход = +225 Расход: 211 Остаток: 335 Выручка 2222 Долги -357.8 ";
Regex regex = new Regex(@"[-+]?\d*\.?\d*");
```

Здесь [-+]? - знак перед цифрой повторенный 0 или 1 раз, \. точка повторенная 0 или 1 раз и цифры, которые могут быть повторены 0 и более раз.

Результат:

```
225        211        335        2222       -357.8
```

Удаление пробелов из текста

```
string
s = "   Это мой текст -  был с     пробелами: в начале, в конце и в середине?    ";
Regex regex = new Regex(@"\b(\w+)\,?\??\!?(\ \-\ )?\:?\ ?");
.......
//При формировании строки из слов, к каждому слову не надо
//добавлять пробел (его находит \ ?).
Text += matchcollection[i].Value;
```

Результат:

```
Это мой текст - был с пробелами: в начале, в конце и в середине?
```

```
string s = "Дядя: 812-555-55-55 Тетя: 555-55-55 Петя: 848-222-22-22";
Regex regex = new Regex
 (@"\b(\d{3}){0,1}(\-){0,1}(\d{3}\-)(\d{2}\-)(\d{2})\b");
```

В примере все знакомые конструкции. В начале слова код города с черточкой, которого может и не быть. Результат:

```
812-555-55-55  555-55-55  848-222-22-22
```

Более усложненный пример, понимающий код города в скобках и номер телефона с (и) без:

```
string s = "Дядя: (812)5555555 Тетя: 555-55-55 Петя: (848)222-22-22";
Regex regex = new Regex
 (@"((\(\b\d{3}\))|(\b(\d{3}){0,1}(\-){0,1}))(\d{7}|((\d{3}\-)(\d{2}\-)(\d{2})\b))");
```

Особенность примера в том, что начало слова не может быть отнесено к скобке и, поэтому, скобка фиксируется как отдельный элемент, а начало слова относится к тому, что в слове.

Результат:

```
(812)5555555  555-55-55  (848)222-22-22
```

Разбор сложного текстового файла содержащего символы табуляции

В данном примере используется реальный файл ведомости платежей, рассылаемый Мегафон в организации, сотрудники которой пользуются в служебных целях данной связью. Фрагмент данного файла содержит семизначный номер телефона, четыре колонки начислений за пользование различными видами связи и колонку суммарных значений платежа.

```
1111111412.86          288.67                 701.53
2222222626.63                                 626.63
3333333
4444444732.96          285.34   54.40         1072.70
5555555 288.23  135.00 156.74                 579.97
```

Пример кажется простым до того момента, пока мы не посмотрим его строковое

представление. Символы табуляции в файле обязательно идут только до номера телефона. Далее они "почти непредсказуемы" и их количество зависит от того, есть ли цифра в следующей колонке. При наличии цифры это два tab, при отсутствии один. Задача - превратить эту "мешанину" tab в нули там, где это необходимо.

\t1111111\t412.86\t\t288.67\t\t\t701.53

\t2222222\t626.63\t\t\t\t\t626.63

\t3333333\t\t\t\t\t\t

\t4444444\t732.96\t\t285.34\t54.40\t1072.70

\t5555555\t288.23\t135.00\t156.74\t\t\t579.97

Выполнить задачу можно только поэтапно:

- Этап 1 - извлекаем номер телефона:
- string sS=СОДЕРЖИМОЕ СТРОКИ С ЗАПИСЬЮ О СЧЕТЕ;
- Regex r = new Regex(@"\d{7}", RegexOptions.IgnoreCase);
- Match mymatch = r.Match(sS);
- if(mymatch.Success)
- {
- string sTel = mymatch.Groups[0].Value.ToString();
- ...
- Этап 2 - сокращаем строку, убирая из нее номер телефона:
- sS = sS.Substring(sS.IndexOf(sTel) + 8, sS.Length - sS.IndexOf(sTel) - 8);
- Этап 3 - заменяем в строке все \t перед которыми есть цифра пробелами:
- Regex r1 = new Regex(@"(?<=\d)\t",RegexOptions.IgnoreCase);
- sS = r1.Replace(sS," ");
- Этап 4 - заменяем оставшиеся \t нулями:
- Regex r2 = new Regex(@"\t",RegexOptions.IgnoreCase);
- sS = r2.Replace(sS1, " 00.00 ");
- Этап 5 - Извлекаем цифры - теперь они на своих местах:
- Regex r3 = new Regex(@"(\d+\.\d+)",RegexOptions.IgnoreCase);

Далее мы можем спокойно использовать извлеченные цифры:

MatchCollection matchcollection = r3.Matches(sS);

Параграф 5. Жадность квантификаторов

Квантификаторы или повторители шаблонов (+ и *) обладают т.н. жадностью - т.е.

возвращают самый длинный фрагмент текста, соответствующий шаблону.

```
string s =
"Это мой текст был с пробелами в начале, в конце и в середине?";
Regex regex = new Regex(@"\bЭ(.*)о");
//или
..Regex regex = new Regex(@"\bЭ(.+)о");
```

Результат:

```
Это мой текст был с пробелами в начале, в ко
```

Параграф 6. Более подробно о Regex

6.1. Конструктор класса и его основные методы

Конструктор класса Regex имеет два перегружаемых метода, для второго из которых options - поразрядная комбинация OR значений перечисления RegexOption.

```
public Regex
(
 string pattern,
 RegexOptions options
);
```

Основные опции определяются, как логическое "И" RegexOptions опций.

- Compiled - регулярное выражение компилируется в сборку, что значительно увеличивает скорость их выполнения, но снижает скорость загрузки. Кроме того, скомпилированные регулярные выражения выгружаются только при завершении работы приложения, даже когда сам объект Regex освобождён и уничтожен сборщиком мусора.
- CultureInvariant - игнорировать культурные различия в языках.
- ECMAScript - включить поведение для выражения, соответствующее ECMAScript. Используется только в соединении с флагами IgnoreCase, Multiline и Compiled, с другими дает ошибку.
- ExplicitCapture - Указывает, что только верные явно названные или пронумерованные (с помощью конструкции (?)) группы сохраняются. Это позволяет избежать излишнего использования конструкции (?::).
- IgnoreCase - игнорировать регистр.
- IgnorePatternWhitespace - устраняет пробелы из шаблонов и позволяет использовать комментарии, отмеченные знаком "#".

- Multiline - значения ^ и $ обозначают начало и конец каждой строчки текста.
- RightToLeft - поиск справа налево.
- SingleLine - весь текст как одна строка. Символ точки в данном режиме включает и \n.

Основные методы класса - это:

- Match - поиск соответствия. Метод возвращает объект класса Match, содержащий результаты найденного соответствия.
- Matches - поиск всех непересекающихся соответствий и возвращение объекта MatchCollection.
- NextMatch - продолжение поиска соответствия для Match с позиции последнего найденного.

Эти методы мы уже многократно использовали при рассмотрении синтаксиса регулярных выражений. Далее остановимся на других методах класса.

6.2. Использование класса Regex для замены в строках по шаблонам

Для замещения вхождений образца символов, определенного регулярным выражением в указанной строке используется метод Replace. Метод имеет 10 перегружаемых реализаций, в которых используются следующие параметры:

- input - исходная строка.
- pattern - регулярное выражение для сопоставления.
- replacement -строка, на которую будут заменены найденные подстроки.
- evaluator - MatchEvaluator для оценки каждого этапа замещения.
- options - поразрядная комбинация OR значений перечисления RegexOption (см. выше).
- count - допустимое максимальное число замен.
- startat - исходная позиция, с которой будет начаты замены.

Метод возвращает либо строку с произведенными заменами, либо строку с результатами выполнения кода вызванного делегата и части строки, не подвергшейся обработке.

Примеры:

1. Заменяем в HTML документе содержимое тэга Title:

```
StringBuilder mystringbuilder = new StringBuilder();
mystringbuilder.Append("<html><head><title>");
mystringbuilder.Append("Регулярные выражения ");
mystringbuilder.Append("в .NET</TITLE></head>");
```

```
mystringbuilder.Append("<body><p>Абзац текста</p>");
mystringbuilder.Append("</body></html>");
Regex regex = new Regex(@"(?<=<title>).*(?=</title>)",
    RegexOptions.IgnoreCase | RegexOptions.ExplicitCapture
 | RegexOptions.Singleline);
 string s = regex.Replace(mystringbuilder.ToString(),
  "Регулярные выражения и их использование в Net");
textBox1.Text = s;
```

Результат

Регулярные выражения и их использование в Net

2. Вызов делегата при каждом совпадении:

```
private int viI=0;
private void button4_Click(object sender, EventArgs e)
{
 string s = "abcdeabrtfabqwe";
 Regex regex = new Regex("ab");
 //Вывод в заголовок и в TextBox
 Text = "";   textBox1.Text="";
 //Replace с вызовом делегата myMatchReplace
 string d = regex.Replace(s,myMatchReplace);
 textBox1.Text = d;
}
public string myMatchReplace(Match match)
{
 if(match.Success)
 {
  for (int i = 0; i < match.Groups.Count; i++)
  {
   Text += "Вызов: "+Convert.ToString(viI)+" "
    +match.Groups[0].Value.ToString() + " ";
  }
 }
 viI++;
 return "Вызов: "+Convert.ToString(viI-1)+" ";
}
```

В результате в Text формы отобразятся все результаты удачных сравнений, обработанные делегатом:

Вызов: 0 ab Вызов: 1 ab Вызов: 2 ab

В TextBox формы видим, то, что возвращает regex.Replace(s,MatchReplace). В этом случае метод возвращает как результаты удачных сравнений, обработанные делегатом, так и остаток строки до следующего совпадения:

Вызов: 0 cde Вызов: 1 rtf Вызов: 3 qwe

6.3. Проверка наличия совпадений

Для проверки наличия совпадений используется метод IsMatch: Метод имеет 2 перегружаемых реализации, в которых используются следующие параметры:

- input - исходная строка.
- startat - исходная позиция, с которой будет начаты замены.

Метод возвращает true, если в строке есть совпадения с шаблоном и false в противном случае.

Пример:

```
string s = "abcdeabrtfabqwe";
Regex regex = new Regex("ab");
bool f = regex.IsMatch(s);
if(f)
 Text = "Совпадение есть";
else
 Text = "Совпадений нет";
```

6.4. Разделение строки по шаблону

Для разделение строки по шаблону используется метод Split: Метод имеет 2 перегружаемых реализации, в которых используются следующие параметры:

- input - исходная строка.
- startat - исходная позиция, с которой будет начаты разбиения.
- count - допустимое максимальное число разбиений.

Метод разбивает строку, используя разделитель, определяемый регулярным выражением, а не набором знаков. Если указан count, строка разбивается максимум на count строк (последняя строка содержит остаток строки, от предыдущих разбиений). При Count=1 строка выводится полностью, при 2 - разбивается на 2 элементы - начало до совпадения и остаток строки и т.д.

Примеры:

```
string s = "cdabcdeabrtfabqwe";
Regex regex = new Regex("ab");
string[] s1 = regex.Split(s,2);
for (int i = 0; i < s1.Length; i++)
{
 textBox1.Text += s1[i] + " ";
}
```

Результат:

```
cd  cdeabrtfabqwe
```

Параграф 7. Некоторые особенности классов для работы с регулярными выражениями

Выше мы перечисляли основные классы и говорили об их предназначении. В этом

параграфе мы более подробно остановимся на некоторых особенностях использования их свойств и методов.

7.1 Match и Captures

Мы уже отмечали, что класс Match - предоставляет результаты одного сопоставления регулярного выражения. Класс Match имеет свойство Captures, унаследованное от класса родитетеля Group. Captures должно хранить массив результатов сопоставлений по группировкам. Но, поскольку, класс при очередном применении регулярного выражения к строке предоставляет только одно сопоставление, то содержание, хранящееся в CaptureCollection (коллекция объектов Capture) представляет собой один элемент Capture. Следующий пример наглядно демонстрирует данное утверждение (кстати, он наглядно демонстрирует и жадность квантификаторов - здесь {1,2}):

```
private void button4_Click(object sender, EventArgs e)
{
 string s = "1 слон 2 слон 3 слон 4 слон 5 слон ";
 string sregx = @"((\d+)\s+(слон)\s+){1,2}";
 Regex regex = new Regex(sregx, RegexOptions.IgnoreCase);
 Match match = regex.Match(s);
 int viI0 = 1;
 while (match.Success)
 {
  System.Console.WriteLine
   ("Применение регулярного выражения к строке " +
     Convert.ToString(viI0)+"й раз");
  viI0++;
  System.Console.WriteLine("Match содержит: " + match.ToString());
  CaptureCollection capturecollection = match.Captures;
  int viI = 1;
  foreach (Capture capture in capturecollection)
  {
   System.Console.WriteLine("Класс Capture номер " +
   Convert.ToString(viI) + " содержит: " + capture);
   viI++;
  }
  //Или так
  for (int i = 0; i < match.Captures.Count; i++)
  {
   System.Console.WriteLine("Класс Capture номер " +
   Convert.ToString(i + 1) + " содержит: " + match.Captures[i]);
  }
  match = match.NextMatch();
 }
}
```

Результат мы вывели в Output Form (Меню View\Output). Заметим, что

Captures.Count всегда равен 1.

```
Применение регулярного выражения к строке 1й раз
Match содержит: 1 слон 2 слон
Класс Capture номер 1 содержит: 1 слон 2 слон
Класс Capture номер 1 содержит: 1 слон 2 слон
Применение регулярного выражения к строке 2й раз
Match содержит: 3 слон 4 слон
Класс Capture номер 1 содержит: 3 слон 4 слон
Класс Capture номер 1 содержит: 3 слон 4 слон
Применение регулярного выражения к строке 3й раз
Match содержит: 5 слон
Класс Capture номер 1 содержит: 5 слон
Класс Capture номер 1 содержит: 5 слон
```

7.2 Match и Group

В регулярное выражение, как правило, входят группировки и кванторы. Как результат, регулярное выражение может иметь несколько отдельных захватов каждым подвыражением регулярного выражения, из которых, как отмечали выше, в качестве значения Match выбирается наибольшее. Результаты промежуточных захватов хранятся в свойстве Groups как классы Group.

```
private void button1_Click(object sender, EventArgs e)
{
 string s = "1 слон 2 слон 3 слон 4 слон 5 слон ";
 string sregx = @"((\d+)\s+(слон)\s+){1,2}";
 Regex regex = new Regex(sregx, RegexOptions.IgnoreCase);
 Match match = regex.Match(s);
 int viI0 = 1;
 while(match.Success)
 {
  System.Console.WriteLine
    ("Применение регулярного выражения к строке " +
        Convert.ToString(viI0) + "й раз");
  viI0++;
  System.Console.WriteLine("Match содержит: " + match.ToString());
  for(int i = 0; i < match.Groups.Count; i++)
  {
   Group group = match.Groups[i];
   System.Console.WriteLine("Group № "+Convert.ToString(i)+
                    " Значение: " + group);
  }
  match = match.NextMatch();
 }
}
```

Результат мы видим в Output Form (Меню View\Output):

```
Применение регулярного выражения к строке 1й раз
Match содержит: 1 слон 2 слон
Group № 0 Значение: 1 слон 2 слон
```

```
Group № 1 Значение: 2 слон
Group № 2 Значение: 2
Group № 3 Значение: слон
Применение регулярного выражения к строке 2й раз
Match содержит: 3 слон 4 слон
Group № 0 Значение: 3 слон 4 слон
Group № 1 Значение: 4 слон
Group № 2 Значение: 4
Group № 3 Значение: слон
Применение регулярного выражения к строке 3й раз
Match содержит: 5 слон
Group № 0 Значение: 5 слон
Group № 1 Значение: 5 слон
Group № 2 Значение: 5
Group № 3 Значение: слон
```

7.3. Match, Group, Capture

Группа захвата может захватить ноль, одну или более строк в отдельном сопоставлении из-за кванторов, поэтому Group их хранит в коллекции объектов Capture.

```
private void button1_Click(object sender, EventArgs e)
{
 string s = "1 слон 2 слон 3 слон 4 слон 5 слон ";
 string sregx = @"((\d+)\s+(слон)\s+){1,2}";
 Regex regex = new Regex(sregx, RegexOptions.IgnoreCase);
 Match match = regex.Match(s);
 int viI0 = 1;
 while(match.Success)
 {
  System.Console.WriteLine
    ("Применение регулярного выражения к строке " +
        Convert.ToString(viI0) + "й раз");
  viI0++;
  System.Console.WriteLine("Match содержит: " + match.ToString());
  for(int i = 0; i < match.Groups.Count; i++)
  {
   Group group = match.Groups[i];
   System.Console.WriteLine("Group № "+Convert.ToString(i)+
                    " Значение: " + group);
    int viI = 0;
    foreach (Capture capture in group.Captures)
    {
     System.Console.WriteLine("Capture № " + Convert.ToString(viI)
       +" Значение: "+capture);
     viI++;
    }
  }
  match = match.NextMatch();
 }
}
```

Результат мы видим в Output Form (Меню View\Output):

```
Применение регулярного выражения к строке 1й раз
Match содержит: 1 слон 2 слон
Group № 0 Значение: 1 слон 2 слон
Capture № 0 Значение: 1 слон 2 слон
Group № 1 Значение: 2 слон
Capture № 0 Значение: 1 слон
Capture № 1 Значение: 2 слон
Group № 2 Значение: 2
Capture № 0 Значение: 1
Capture № 1 Значение: 2
Group № 3 Значение: слон
Capture № 0 Значение: слон
Capture № 1 Значение: слон
Применение регулярного выражения к строке 2й раз
Match содержит: 3 слон 4 слон
Group № 0 Значение: 3 слон 4 слон
Capture № 0 Значение: 3 слон 4 слон
Group № 1 Значение: 4 слон
Capture № 0 Значение: 3 слон
Capture № 1 Значение: 4 слон
Group № 2 Значение: 4
Capture № 0 Значение: 3
Capture № 1 Значение: 4
Group № 3 Значение: слон
Capture № 0 Значение: слон
Capture № 1 Значение: слон
Применение регулярного выражения к строке 3й раз
Match содержит: 5 слон
Group № 0 Значение: 5 слон
Capture № 0 Значение: 5 слон
Group № 1 Значение: 5 слон
Capture № 0 Значение: 5 слон
Group № 2 Значение: 5
Capture № 0 Значение: 5
Group № 3 Значение: слон
Capture № 0 Значение: слон
```

И последнее замечание: Match в свойстве Groups хранит объекты класса GroupCollection и, поэтому, можно пользоваться следующим кодом для доступа к значениям Group:

```
GroupCollection groupcollection = match.Groups;

int viI = 0;
foreach (Group group in groupcollection)
{
 System.Console.WriteLine("Group № " + Convert.ToString(viI) +
                    " Значение: " + group);
 viI++;}
```