Частное учреждение образования

«Колледж бизнеса и права»

|  |
| --- |
| УТВЕРЖДАЮ  Заведующий  методическим кабинетом  \_\_\_\_\_\_\_\_\_\_\_\_\_Е.В.Фалей  « \_\_\_\_ » \_\_\_\_\_\_\_\_\_ 2016 |

|  |  |
| --- | --- |
| Специальность: 2-40 01 01 «Программное обеспечение информационных технологий» | Учебная практика по программированию |
| Составлена в соответствии с учебной программой, утвержденной директором Колледжа бизнеса и права 31.05.2016 | |

**Практическая работа №7**

**Инструкционно-технологическая карта**

Тема: Элементы языка регулярных выражений.

Цель: Научиться применять знания элементов языка регулярных выражений.

Время выполнения: 2 часа.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Ознакомиться с теоретическими сведениями.
2. Ознакомиться с примерами использования регулярных выражений, представленными в данной инструкционно-технологической карте.
3. Набрать, отладить, протестировать и выполнить фрагменты кода, представленные в данной инструкционно-технологической карте.
4. Выполнить задание, выданное преподавателем.
5. Оформить решение в отчет.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Регулярные выражения

Язык регулярных выражений распознает шаблоны символов. Типы .NET, подде­рживающие регулярные выражения, основаны на регулярных выражениях Perl 5 и обеспечивают функциональность поиска и поиска/ замены. Регулярные выражения используются для решения следующих задач:

* проверка текстового ввода, такого как пароли и телефонные номера (для этой цели в ASP.NET предоставляется элемент управления RegularExpressionValidator);
* преобразование текстовых данных в более структурированные формы (напри­мер, извлечение данных из HTML-страницы для сохранения в базе данных);
* замена образцов текста в документе (например, только целых слов).

Все типы для работы с регулярными выражениями определены в пространстве имен System.Text.RegularExpressions.

Рассмотрим элементы языка регулярных выражений.

Основы регулярных выражений

Одной из наиболее распространенных операций регулярных выражений является квантификатор. Операция ? — это квантификатор, который соответствует предшествующему элементу 0 или 1 раз. Другими словами, ? означает необязательный. Элемент — это либо одиночный символ, либо сложная структура символов в квадратных скобках. Например, регулярное выражение "colou?r" соответствует color и colour, но не colouur:

Console.WriteLine (Regex.Match ("color", @"colou?rM).Success); // True

Console.WriteLine (Regex.Match ("colour", @"colou?r").Success); // True

Console.WriteLine (Regex.Match ("colouur", @"colou?r").Success); //False

Метод Regex.Match выполняет поиск внутри большой строки. Возвращаемый им объект имеет свойства для позиции (Index) и длины (Length) соответствия, а также свойство для действительного значения (Value) соответствия:

Match m = Regex.Match ("any colour you like", @"colou?r");

Console.WriteLine(m.Success); //True

Console.WriteLine(m.Index); //4

Console.WriteLine(m.Length); //6

Console.WriteLine(m.Value); //colour

Console.WriteLine (m.ToString()); //colour

Метод Regex.Match можно воспринимать как более мощную версию метода IndexOf типа string. Разница в том, что он ищет совпадение с *шаблоном*, а не с лите­ральной строкой.

Метод IsMatch — это сокращения для вызова метода Match с последующей про­веркой свойства Success.

Механизм регулярных выражений по умолчанию работает слева направо, поэтому возвращается только самое левое соответствие. Для возврата дополнительных соответствий можно использовать метод NextMatch:

Match ml = Regex.Match ("One color? There are two colours in my head!",

colou?rs?");

Match m2 = ml. NextMatch();

Console.WriteLine(ml); // color

Console.WriteLine(m2); // colours

Метод Matches возвращает все соответствия в виде массива. Предыдущий пример можно переписать, как показано ниже:

foreach (Match m in Regex.Matches ("One color? There are two colours in my head!", @"colou?rs?"))

Console.WriteLine(m);

Еще одной распространенной операцией регулярных выражений является *переста­новка*, обозначаемая вертикальной чертой, |. Перестановка выражает альтернативы. Следующее выражение соответствует Jen, Jenny и Jennifer:

Console.WriteLine (Regex.IsMatch ("Jenny", "Jen(ny|nifer)?")); // True

Скобки вокруг перестановки отделяют альтернативы от остальной части выра­жения.

Начиная с версии .NET Framework 4.5, при поиске совпадений с регулярными выражениями можно указывать тайм-аут. Если операция поиска со­ответствия занимает больше времени, чем указанное в объекте TimeSpan, генерируется исключение RegexMatchTimeoutException. Это может быть полезно, если программа обрабатывает произвольные регулярные выражения (например, в диалоговом окне расширенного поиска), поскольку предотвращает бесконечное зацикливание неправильно сформи­рованных регулярных выражений.

**Скомпилированные регулярные выражения**

В некоторых приведенных ранее примерах вызывался статический метод Regex многократно с одним и тем же шаблоном. Альтернативным подходом в таких случаях является создание объекта Regex с этим шаблоном, а затем вызов методов экземпляра:

Regex r = new Regex(@"sausages?");

Console.WriteLine(r.Match ("sausage")); // sausage

Console.WriteLine(r.Match ("sausages")); // sausages

Это не только удобно с точки зрения синтаксиса: на самом деле экземпляр Regex использует облегченную генерацию кода (DynamicMethod в Reflection.Emit) для динамического построения и компиляции кода, настроенного на это конкретное вы­ражение. В результате обеспечивается более быстрый (до 10 раз) поиск соответствия за счет небольшого увеличения времени первоначальной компиляции (до нескольких десятков микросекунд).

Экземпляр Regex является неизменяемым.

Механизм регулярных выражений характеризуется высокой скоростью. Даже без компиляции нахождение простого совпадения требует менее одной микросекунды.

RegexOptions

Перечисление флагов RegexOptions позволяет настраивать поведение соответс­твия. Распространенным применением RegexOptions является выполнение поиска, нечувствительного к регистру символов:

Console.WriteLine(Regex.Match("a","A",RegexOptions.IgnoreCase)); // a

Это применяет правила для эквивалентности регистров символов текущей культу­ры. Флаг Culturelnvariant позволяет затребовать инвариантную культуру:

Console.WriteLine (Regex.Match ("a", **"A",** RegexOptions.IgnoreCase

| RegexOptions.Culturelnvariant));

Большинство флагов RegexOptions могут также активизироваться внутри самого регулярного выражения с использованием однобуквенного кода:

Console.WriteLine (Regex.Match ("a", @"(?i)A")); //a

Опции можно включать и отключать на протяжении всего выражения:

Console.WriteLine (Regex.Match ("AAAa", (?i)a(?-i)a")); // Aa

Другой полезной опцией является IgnorePatternWhitespace или (?х). Она поз­воляет вставлять пробельные символы, чтобы улучшить читабельность регулярного выражения, не трактуя эти символы литерально.

В таблице 1 перечислены все значения RegexOptions вместе с их однобуквенны­ми кодами.

Таблица 1 – Параметры регулярных выражений

|  |  |  |
| --- | --- | --- |
| Значение перечисления | Однобуквенный код | Описание |
| None |  |  |
| IgnoreCase | i | Игнорировать регистр символов (по умолча­нию регулярные выражения чувствительны к регистру символов) |
| Multiline | m | Изменить А и $ так, чтобы они соответство­вали началу/концу строки текста, а не началу/ концу всей строки регулярного выражения |
| ExplicitCapture | n | Захватывать только явно именованные или явно нумерованные группы (как описано в разделе "Группы" далее) |
| Compiled | c | Инициировать компиляцию регулярного вы­ражения в IL |
| Singleline | s | Сделать точку (.) соответствующей любому символу (вместо соответствия любому сим­волу кроме \n) |
| IgnorePatternWhitespace | x | Устранить из шаблона ненужные пробельные символы |
| RightToLeft | r | Выполнять поиск справа налево; не может быть указан посреди операции |
| ECMAScript |  | Обеспечить совместимость с ЕСМА (по умолчанию реализация не является совмес­тимой с ЕСМА) |
| CultureInvariant |  | Отключить поведение, специфичное для культуры, при сравнении строк |

Отмена символов

Регулярные выражения имеют следующие метасимволы, которые трактуются спе­циальным образом, отличным от их литерального смысла:

\\* + ?|{[()^$.#

Чтобы сослаться на метасимвол литерально, его потребуется предварить обратной косой чертой. В следующем примере мы отменяем символ ? для соответствия строке "what?":

Console.WriteLine (Regex.Match ("what?", @"what\?")); //what? (правильно)

Console.WriteLine (Regex.Match ("what?", @"what?")); //what (неправильно)

Если символ находится внутри набора (в квадратных скобках), это прави­ло не применяется, и метасимволы интерпретируются литеральным обра­зом.

Методы Escape и Unescape класса Regex преобразуют строку, содержащую ме­тасимволы регулярных выражений, путем замены их отмененными эквивалентами и наоборот.

Например:

Console.WriteLine (Regex.Escape (@"?")); // \?

Console.WriteLine (Regex.Unescape (@"\?")); // ?>

Все строки регулярных выражений в этой теме представлены с помощью литерала @ в С#. Это сделано для того, чтобы обойти механизм отмены самого языка С#, который также использует обратную косую черту. Без @ литеральная обратная косая черта потребовала бы четырех таких символов:

Console.WriteLine (Regex.Match ("\\", "\\\\")); // \

Если только не включена опция (?х), пробелы в регулярных выражениях тракту­ются литеральным образом:

Console.Write (Regex.IsMatch ("hello world", ©"hello world")); // True

Наборы символов

Наборы символов действуют в качестве групповых символов для отдельного набора символов.

Таблица 2 – Наборы символов

|  |  |  |
| --- | --- | --- |
| Выражение | Описание | Инверсия («не») |
| [abcdef] | Соответствие одиночному символу в списке | [^abcdef] |
| [a-f] | Соответствие одиночному символу в диапазоне | [^a-f] |
| \d | Соответствие десятичной цифре. То же самое, что и [0-9] | \D |
| \w | Соответствие символу, допустимому в словах (по умолчанию варьируется согласно Culturelnfo. CurrentCulture; например, в английском языке это то же самое, что и [a-zA-z\_0-9]) | \W |
| \s | Соответствие пробельному символу. То же самое, что и [\n\r\t\f] | \S |
| \p{категория} | Соответствие символу в указанной категории (см. приложение) | \P |
| . | (Стандартный режим.)  Соответствие любому символу кроме \n | \n |
| . | (Режим SingleLine.) Соответствие любому символу | \n |

Для соответствия в точности одному символу из набора поместите набор символов в квадратные скобки:

Console.Write (Regex.Matches ("That is that.", "[Tt]hat").Count); // 2

Для соответствия любому символу *за исключением* тех, что перечислены в наборе, поместите набор в квадратные скобки и перед первым символом укажите л:

Console.Write (Regex.Match ("quiz qwerty", "q[Aaeiou]").Index); // 5

С помощью дефиса можно задать диапазон символов. Следующее выражение соот­ветствует шахматному ходу:

Console.Write (Regex.Match ("Ы-с4", @"[a-h]\d-[a-h]\d").Success); //True

\d указывает цифровой символ, поэтому \d будет соответствовать любой цифре. \ D соответствует любому нецифровому символу.

\w указывает символ, допустимый в словах, что включает буквы, цифры и символ подчеркивания. \W соответствует любому символу, не допустимому в словах. Это рабо­тает ожидаемым образом и для неанглийских букв, таких как кириллица.

. соответствует любому символу кроме \n (но позволяет \r).

\р соответствует символу в указанной категории, такой как {Lu} для буквы верхне­го регистра или {Р} для знака пунктуации (список категорий будет приведен далее):

Console.Write (Regex.IsMatch ("Yes, please", @"\p{P}")); // True

Дополнительные случаи использования \d, \w и . касаются их комбинирования с квантификаторами.

Квантификаторы

Квантификаторы обеспечивают соответствие элементу указанное количество раз.

Таблица 3 - Квантификаторы

|  |  |
| --- | --- |
| j | Описание |
| \* | Ноль или большее число совпадений |
| + | Одно или большее число совпадений |
| ? | Ноль или одно совпадение |
| {n} | В точности n совпадений |
| {n,} | По меньшей мере, n совпадений |
| {n,m} | Совпадений между n и m |

Квантификатор \* обеспечивает соответствие предшествующего символа или группы ноль или более раз. Следующее выражение соответствует имени файла cv. doc, а также любых версий имени с числами (например, cv2.doc, cv15.doc):

Console.Write (Regex.Match ("cvl5.doc", @"cv\d\*\.doc").Success); // True

Обратите внимание на отмену символа точки в расширении файла с помощью об­ратной косой черты. Следующее выражение допускает наличие любых символов между *cv* и . *doc* и эквивалентно команде dir cv\* . doc:

Console.Write (Regex.Match ("cvjoint.doc", @"cv.\*\.doc").Success);// True

Квантификатор + обеспечивает соответствие предшествующего символа или группы один или более раз. Например:

Console.Write(Regex.Matches("slow! yeah slooow!","slo+w").Count); // 2

Квантификатор {} обеспечивает соответствие указанному количеству (или диапазону) повторений. Следующее выражение выводит показания артериального давления:

Regex bp = new Regex (@"\d{2,3}/\d{2,3}") ;

Console.WriteLine (bp.Match ("It used to be 160/110")); // 160/110

Console.WriteLine (bp.Match ("Now it's only 115/75")); // 115/75

Жадные и ленивые квантификаторы

По умолчанию квантификаторы являются жадными, как противоположность ленивым квантификаторам. Жадный квантификатор повторяется настолько много раз, сколько может, прежде чем продолжить. Ленивый квантификаторы повторяется настолько *мало* раз, сколько может, прежде чем продолжить. Чтобы сделать любой квантификатор ленивым, его необходимо снабдить суффиксом в виде символа ?. Для иллюстрации разницы рассмотрим следующий фрагмент HTML-разметки:

string html = "<i>By default</i> quantifiers are <i>greedy</i> creatures";

Предположим, что нужно извлечь две фразы, представленные курсивом. Если мы запустим следующий код:

foreach (Match m in Regex.Matches (html, @"<i>.\*</i>"))

Console.WriteLine (m) ;

то результатом будет не два, а *одно* совпадение:

<i>By default</i> quantifiers are <i>greedy</i>

Проблема в том, что квантификатор \* жадным образом повторяется настолько много раз, сколько он может, перед соответствием дескриптору </i>. Таким образом, он поглощает первый дескриптор </i>, останавливаясь только на финальном дескрипторе </i> *(последняя точка*, где пока еще обеспечивается совпадение).

Если сделать квантификатор ленивым:

foreach (Match m in Regex.Matches (html, @"<i>.\*?</i>"))

Console.WriteLine (m);

то он остановится в *первой* точке, где получено совпадение. Вот результат:

<i>By default</i>

<i>greedy</i>

**Утверждения нулевой ширины**

Язык регулярных выражений позволяет размещать условия, которые должны удов­летворяться *перед* или *после* совпадения, через *просмотр назад*, *просмотр вперед*, *привязки* и *границы слов.* Все вместе это называется утверждениями *нулевой ширины*, поскольку они не увеличивают ширину (или длину) самого совпадения.

**Просмотр вперед и просмотр назад**

Конструкция (?=ехрг) проверяет, соответствует ли следующей за ней текст выражению ехрг, не включая ехрг в результат. Это называется *положительным просмот­ром вперед.* В приведенном ниже примере ищется число, за которым следует слово miles:

Console.WriteLine (Regex.Match ("say 25 miles more", @"\d+\s(?=miles)")); ВЫВОД: 25

Обратите внимание, что слово miles не возвращается в результате, хотя оно и требуется для удовлетворения совпадению.

После успешного просмотра вперед поиск соответствия продолжается, как если бы предварительный просмотр никогда не осуществлялся. Таким образом, если добавить к выражению конструкцию . \*, как показано ниже:

Console.WriteLine (Regex.Match ("say 25 miles more", @"\d+\s(?=miles).\*"));

то результатом будет 25 miles more.

Просмотр вперед может быть полезен для навязывания правил выбора сильных паролей. Предположим, что пароль должен иметь длину не менее шести символов и содержать, по крайней мере, одну цифру.

С помощью просмотра этого можно достичь следующим образом:

string password = "...";

bool ok = Regex.IsMatch (password, @"(?=.\*\d).{6,}");

Здесь сначала выполняется *просмотр вперед*, чтобы удостовериться в наличии цифры где-нибудь в строке. Если это так, производится возврат к позиции перед началом предварительного просмотра и осуществляется проверка соответствия шести или более символам.

Противоположностью является конструкция *отрицательного просмотра вперед*, (?!*ехрr).* Она требует, чтобы не было соответствия следующему за ним *ехрr.* Приведенное ниже выражение соответствует good, если только позже в строке не встречается however или but:

string regex = "(?i)good(?!.\*(however|but))

Console.WriteLine (Regex.IsMatch ("Good work! But...", regex));// False

Console.WriteLine (Regex.IsMatch ("Good work! Thanks!", regex)); // True

Конструкция (?<=expr) обозначает *положительный просмотр назад* и требует, что­бы совпадению *предшествовало* указанное выражение. Противоположная конструкция, (?<!*ехрr*), обозначает *отрицательный просмотр назад* и требует, чтобы совпадению *не предшествовало указанное* выражение. Например, следующее выражение соответствует good, если только however не встречается *ранее* в строке:

string regex = " (?i) (?<!however.\*)good";

Console.WriteLine (Regex.IsMatch ("However good, we...", regex));// False

Console.WriteLine(Regex.IsMatch ("Very good, thanks!", regex)); // True

Приведенные примеры можно было бы усовершенствовать, добавляя *утверждения границ слову* которые вскоре будут описаны.

Привязки

Привязки ^ и $ соответствуют конкретной *позиции.* По умолчанию:

* ^ соответствует *началу* строки;
* $ соответствует *концу* строки.

Символ ^ имеет два значения, зависящие от контекста: *привязка* и *отри­цание класса символов.*

Символ $ также имеет два значения, зависящие от контекста: *привязка* и *маркер группы замены.*

Например:

Console.WriteLine (Regex.Match ("Not now", "^[Nn]о")); //No

Console.WriteLine (Regex.Match ("f = 0.2F", "[Ff]$")); // F

Если указать RegexOptions.Multiline или включить конструкцию (?m) в выра­жение, то:

^ соответствует началу всей строки или *строки текста* (прямо после \n);

$ соответствует концу всей строки или *строки текста* (прямо перед \n).

Существует одна особенность использования $ в многострочном (Multiline) ре­жиме: новая строка в Windows почти всегда обозначается с помощью \r\n, а не просто \n. Это значит, что для применения $ потребуется обычно искать совпадение также и с \r, применяя *положительный просмотр вперед:*

(?=\r?$)

Положительный просмотр вперед гарантирует, что \r не станет частью результата. Приведенный ниже код соответствует строкам, завершающимся на " .txt":

string fileNames = "a.txt" + "\r\n" + "b.doc" + "\r\n" + "c.txt"; string r = @".+\.txt(?=\r?$)";

foreach (Match m in Regex.Matches (fileNames, r, RegexOptions.Multiline)) Console.Write (m + " ");

ВЫВОД: a, txt c.txt

Следующий код соответствует всем пустым строкам внутри s:

MatchCollection emptyLines = Regex .Matches (s, "^(?=\r?$)",

RegexOptions.Multiline);

Показанный далее код соответствует всем строкам, которые либо пусты, либо содержат только пробельные символы:

MatchCollection blankLines = Regex .Matches (s,”^[ \t]\*(?=\r?$)",

RegexOptions.Multiline);

Поскольку привязка соответствует позиции, а не символу, указание одной лишь привязки соответствует пустой строке:

Console.WriteLine (Regex.Match ("x", "$").Length); // 0

**Границы слов**

Утверждение границы слова \b дает соответствие, когда символы, допустимые в словах (\w), соседствуют с одним из перечисленного ниже:

* символы, не допустимые в словах (\W);
* начало/конец строки (^ и $).

\b часто используется для соответствия целым словам. Например:

foreach (Match m in Regex.Matches ("Wedding in Sarajevo", 0"\b\w+\b")) Console.WriteLine (m) ;

Wedding

In

Sarajevo

Следующие операторы подчеркивают эффект от границы слова:

int one = Regex.Matches ("Wedding in Sarajevo", @"\bin\b").Count;// 1

int two = Regex.Matches ("Wedding in Sarajevo", @"in").Count; // 2

Приведенное ниже выражение применяет положительный просмотр вперед для возврата слов, за которыми следует (sic):

string text = "Don’t loose (sic) your cool";

Console.Write (Regex.Match (text, @"\b\w+\b\s(?=\(sic\))")); // loose

**Группы**

Временами регулярное выражение удобно разделять на последовательности под­выражений, или *группы.* Например, рассмотрим следующее регулярное выражение, которое представляет телефонные номера в США, такие как 206-465-1918:

\d{3}-\d{3}-\d{4}

Предположим, что нужно разделить его на две группы: код зоны и локальный номер. Этого можно достигнуть, используя круглые скобки для *захвата* каждой группы:

(\d{3})-(\d{3}-\d{4})

Затем группы можно извлечь программно:

Match m = Regex.Match ("206-465-1918", @"(\d{3})-(\d{3}-\d{4})");

Console.WriteLine (m.Groups[1]); // 206

Console.WriteLine (m.Groups[2]); // 465-1918

Нулевая группа представляет полное совпадение. Другими словами, она имеет то же самое значение, что и свойство Value совпадения:

Console.WriteLine (m.Groups[0]); // 206-465-1918

Console.WriteLine (m); // 206-465-1918

Группы являются частью самого языка регулярных выражений. Это означает, что можно ссылаться на группу внутри регулярного выражения. Синтаксис \n позволяет индексировать группу по ее номеру n в рамках выражения. Например, выраже­ние (\w)ее\1 дает совпадения для deed и peep. В следующем примере ищутся в строке все слова, начинающиеся и заканчивающиеся на ту же самую букву:

foreach (Match m in Regex.Matches ("pop pope peep", @"\b(\w)\w+\l\b"))

Console.Write (m + " ") ; // pop peep

Скобки вокруг \w указывают механизму регулярных выражений на необходимость сохранения подсовпадений в группе (в этом случае одиночной буквы), поэтому их можно будет использовать позже. В дальнейшем на эту группу можно ссылаться с при­менением \1, что означает первую группу в выражении.

Именованные группы

В длинном или сложном выражении работать с группами удобнее по именам, а не по индексам. Ниже приведен предыдущий пример, переписанный с использованием группы по имени ' letter':

string regEx =

@"\b" + // граница слова

@"(?'letter'\w)"+ // соответствует первой букве; назовем группу 'letter'

@"\w+" + // соответствует промежуточным буквам

@"\k'letter'" + // соответствует последней букве, отмеченной 'letter'

@"\b"; // граница слова

foreach (Match m in Regex.Matches ("bob pope peep", regEx))

Console .Write (m + " "); // bob peep

Вот как именовать захваченную группу:

(?'имя-группы'выражение-группы) или (?<имя-группы>выражение-группы)

Вот как ссылаться на группу:

\к'имя-группы' или \к<имя-группы>

В следующем примере осуществляется сопоставление для простого (не вложенно­го) элемента XML/HTML за счет поиска начального и конечного узлов с совпадаю­щим именем:

string regFind =

@"<(?'tag'\w+?).\*>" + // Соответствует первому дескриптору;

// назовем группу 'tag'

@" (?'text'.\*?)" + // Соответствует текстовому содержимому;

// назовем группу 'text'

@"</\k'tag'>"; // Соответствует последнему дескриптору, отмеченному 'tag'

Match m = Regex.Match ("<hl>hello</hl>", regFind);

Console.WriteLine (m.Groups ["tag"]); // hi

Console.WriteLine (m.Groups ["text"]); // hello

Замена и разделение текста

Метод RegEx. Replace работает подобно string. Replace за исключением того, что использует регулярное выражение. Следующий код заменяет строку cat строкой dog. В отличие от string. Replace, слово catapult не будет изменено на dogapult, поскольку соответствия ищутся по границам слов:

string find = @"\bcat\b";

string replace = "dog";

Console.WriteLine (Regex.Replace ("catapult the cat", find, replace));

ВЫВОД: catapult the dog

Строка замены может ссылаться на исходное совпадение с помощью подстановоч­ной конструкции $0. Следующий пример помещает числа внутри строки в угловые скобки:

string text = "10 plus 20 makes 30";

Console.WriteLine (Regex.Replace (text, @"\d+", @"<$0>"));

ВЫВОД: <10> plus <20> makes <30>

Обращаться к захваченным группам можно с помощью $1, $2, $3 и т.д. или посредс­твом *${имя}* для именованных групп. Чтобы продемонстрировать, когда это может быть удобно, вспомним регулярное выражение из предыдущего раздела, соответству­ющее простому элементу XML. За счет перестановки групп можно сформировать выражение замены, которое перемещает содержимое элемента в атрибут XML:

string regFind =

@"<(?'tag'\w+?).\*>" + // Соответствует первому дескриптору;

// назовем группу 'tag'

@"(?'text'.\*?)" + // Соответствует текстовому содержимому;

// назовем группу 'text'

@"</\k'tag'>"; // соответствует последнему дескриптору, отмеченному 'tag'

string regReplace =

@"<${tag}" + // <tag

@"value=""" + // value="

@"${text}" + // text

@"""/>"; // "/>

Console.Write (Regex.Replace ("<msg>hello</msg>", regFind, regReplace));

Вот результат:

<msg value="hello"/>

**Делегат MatchEvaluator**

Метод Replace имеет перегруженную версию, принимающую делегат MatchEvaluator, который вызывается для каждого совпадения. Это позволяет поручить построение содержимого строки замены коду С#, если язык регулярных выражений оказывается для этого недостаточно выразительным. Например:

Console.WriteLine (Regex.Replace ("5 is less than 10", @"\d+",

m => (int.Parse (m.Value) \* 10) . ToString()));

ВЫВОД: 50 is less than 100

В рецептурном справочнике будет показано, как использовать MatchEvaluator для защиты символом Unicode специально для HTML.

Разделение текста

Статический метод Regex. Split является более мощной версией метода string. Split, с регулярным выражением, обозначающим шаблон разделителя.

В следующем примере разделяется строка, в которой любая цифра считается раз­делителем:

foreach (string s in Regex.Split ("a5b7c", @"\d"))

Console.Write (s + " ") ; // a b c

Результат здесь не включает сами разделители. Тем не менее, включить раздели­тели можно, поместив выражение в положительный просмотр вперед. Следующий код разбивает строку в верблюжьем стиле на отдельные слова:

foreach (string s in Regex.Split ("oneTwoThree", @"(?=[A-Z])"))

Console .Write (s + " "); // one Two Three

Примеры регулярных выражений

Соответствие номеру карточки социального страхования/ телефонному номеру в США

string ssNum = @"\d{3}-\d{2}—\d{4}";

Console.WriteLine (Regex.IsMatch ("123-45-6789", ssNum)); // True

string phone = @"(?x)

( \d{3}[-\s] | \(\d{3}\)\s? )

\d{3}[-\s]?

\d{4}";

Console.WriteLine (Regex.IsMatch ("123-456-7890", phone)); // True

Console.WriteLine (Regex.IsMatch ("(123) 456-7890", phone)); // True

Извлечение пар "имя = значение" (по одной в строке)

Обратите внимание на применение в начале директивы (?m);

string r = @"(?m)^\s\*(?'name'\w+)\s\*=\s\*(?'value'.\*)\s\*(?=\r?$)";

string text =

@"id = 3

secure = true

timeout = 30";

foreach (Match m in Regex.Matches (text, r) )

Console.WriteLine (m.Groups["name"] + " is " + m.Groups["value"]);

*id is 3 secure is true timeout is 30*

Проверка сильных паролей

Следующий код проверяет, что пароль состоит, по меньшей мере, из шести симво­лов и включает цифру, символ или знак пунктуации:

string r = @"(?х)^(?=.\* ( \d | \р{Р} | \р{S} )).{6,}";

Console.WriteLine(Regex.IsMatch ("abcl2", r) ); // False

Console.WriteLine(Regex.IsMatch ("abcdef", r) ); // False

Console.WriteLine(Regex.IsMatch ("ab88yz", r) ); // True

Строки, содержащие, по крайней мере, 80 символов

string r = @"(?m)^.{80,)(?=\r?$)";

string fifty = new string ('x', 50);

string eighty = new string('x', 80);

string text = eighty + "\r\n" + fifty + "\r\n" + eighty;

Console.WriteLine (Regex.Matches (text, r).Count); // 2

Разбор даты/времени

Это выражение поддерживает разнообразные числовые форматы даты и работает независимо от того, указан год в начале или конце. Директива (?х) улучшает читабельность, разрешая применение пробельных символов; директива (?i) отключает чувствительность к регистру символов (для необязательного указателя АМ/РМ). Затем к компонентам совпадения можно обращаться через коллекцию Groups:

string r = @"(?х)(?i)

(\d{l,4}) [./-]

(\d{l,2}) [./-]

(\d{1,4}) [\sT]

(\d+):(\d+):(\d+) \s? (A\.?M\.?|P\.?M\.?)?";

string text = "01/02/2008 5:20:50 PM";

foreach (Group g in Regex.Match (text, r).Groups)

Console.WriteLine (g.Value + " ");

01/02/2008 5:20:50 PM 01 02 2008 5 20 50 PM

(Разумеется, это не проверяет правильность даты/времени.)

Соответствие римским числам

string r =

@"(?i)\bm\*" +

@"(d?c{0,3}|с[dm])" +

@"(1?х{0,3}|x[lc])" +

@" (v?i{0,3}|i[vx])" +

@"\b";

Console.WriteLine (Regex.IsMatch ("MCMLXXXIV", r) ) ; //True

Удаление повторяющихся слов

Здесь захватываем именованную группу dupe:

string r = @"(?'dupe'\w+)\W\k'dupe'";

string text = "In the the beginning...";

Console.WriteLine (Regex.Replace (text, r, "${dupe}"));

In the beginning

Подсчет слов

string r = @"\b(\w|[-‘])+\b";

string text = "It's all mumbo-jumbo to me";

Console.WriteLine (Regex.Matches (text, r).Count); // 5

Соответствие GUID

string r =

@"(?i)\b" +

@"[0-9a-fA-F]{8}\-" +

@"[0-9a-fA-F]**{4}\-**" **+**

@"[0-9a-fA-F]{4}\-" +

@"[0-9a-fA-F]{4}\-" +

@"[O-9a-fA-F]{12}" +

@"\b";

string text = "Its key is {3F2504E0-4F89-UD3-9A0C-0305E82C3301}.";

Console.WriteLine (Regex.Match (text, r).Index); // 12

Разбор дескриптора XML/HTML

Класс Regex удобен при разборе фрагментов HTML-разметки — особенно, когда документ может быть сформирован некорректно:

string r =

@"<(?'tag'**\w+?)**.\*>" + // соответствует первому дескриптору;

// назовем группу 'tag'

@"(?'text'.\*?)" + // соответствует текстовому содержимому;

// назовем группу 'text'

@"</\k'tag'>";// соответствует последнему дескриптору, отмеченному 'tag'

string text = "<hl>hello</hl>";

Match m = Regex.Match (text, r);

Console.WriteLine (m.Groups ["tag"]); // hi

Console.WriteLine (m.Groups ["text"]); // hello

Разделение на слова в верблюжьем стиле

Это требует положительного просмотра вперед, чтобы включить разделители в верхнем регистре:

string r = @"(?=[A-Z])";

foreach (string s in Regex.Split ("oneTwoThree", r) )

Console.Write (s + " "); // one Two Three

Получение допустимого имени файла

string input = "My \"good\" <recipes>.txt";

char[] invalidChars = System.10.Path.GetlnvalidPathChars();

string invalidstring = Regex.Escape (new string (invalidChars));

string valid = Regex.Replace (input, "[" + invalidstring + "]","") ;

Console.WriteLine (valid);

My good recipes.txt

Отмена символов Unicode для HTML

string htmlFragment = "© 2007";

string result = Regex.Replace (htmlFragment, @"[\u0080-\uFFFF]",

m => @"&#" + ( (int)m.Value [0]) .ToString() + ";");

Console.WriteLine (result); // &#169; 2007

Преобразование символов в строке запроса HTTP

string sample = "C%23 rocks";

string result = Regex.Replace (

sample,

@"%[0-9a-f][0-9a-f]",

m => ((char) Convert.ToByte (m.Value.Substring(1), 16)).ToString(),

RegexOptions.IgnoreCase

) ;

Console.WriteLine (result); // C# rocks

Разбор поисковых терминов Google из журнала веб-статистики

Это должно использоваться в сочетании с предыдущим примером преобразования символов в строке запроса:

string sample =

"http://google.com/search?hl=en&q=greedy+quantifiers+regex&btnG=Search";

Match m = Regex.Match (sample, @"(?<=google\..+search\?.\*q=).+?(?=(&!$))");

string[] keywords = m.Value.Split ( new[] { ' + ' },

StringSplitOptions.RemoveEmptyEntries) ;

foreach (string keyword in keywords)

Console.Write (keyword + " ") ; // greedy quantifiers regex

ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ

**Задание 1**. Используя знания об элементах языка регулярных выражений интерпретируйте следующий шаблон:

1. gr[ae]y\s\S+?[\s|\p{P}]
2. \b[A-Z]\w\*\b
3. \bth[^o]\w+\b
4. \b.\*[.?!;:](\s|\z)
5. \P{Sc})+
6. (\w)\1
7. \b(\w+)(\W){1,2}
8. \b\w+(e)?s(\s|$)
9. \b(\S+)\s?
10. ^(\(?\d{3}\)?[\s-])?\d{3}-\d{4}$
11. ^\D\d{1,5}\D\*$
12. ^[0-9-[2468]]+$
13. (\w+)\s(\1)
14. \D+(?<digit>\d+)\D+(?<digit>\d+)?
15. \b91\*9\*\b
16. \ban+\w\*?\b
17. \ban?\b
18. \b\d+\,\d{3}\b
19. \b\d{2,}\b\D+
20. \b\w\*?oo\w\*?\b
21. \b\w+?\b
22. \b[A-Z](\w\*\s+){1,10}?[.!?]
23. \b(?(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b

**Задание 2**. Составьте шаблон регулярного выражения для проверки корректного ввода номера паспорта. Протестируйте шаблон.

ИНФОРМАЦИОННЫЕ ИСТОЧНИКИ

1. Албахари, Дж. C#5.0. Справочник. Полное описание языка / Дж. Албахари, Б. Албахари. – 5-е изд. – М.: ООО «И.Д. Вильямс», 2014. – 1008 с.: ил.
2. Гойвертс, Я. Регулярные выражения. Сборник рецептов / Я. Гойвертс, Ст. Левитан. – 2-е изд. – СПб.: Символ-Плюс, 2015. – 704 с.: ил.
3. Фицджеральд, М. Регулярные выражения: основы / М. Фицджеральд. – М.: ООО «И.Д. Вильямс», 2015. – 144 с.: ил.
4. Троелсен, Э. Язык программирования C# 5.0 и платформа .NET 4.5 / Э. Троелсен. – 6-е изд. – М.: ООО «И.Д. Вильямс», 2013. – 1312 с.: ил.
5. Шилдт, Г. C#4.0: Полное руководство / Г. Шилдт. – М.: ООО «И.Д. Вильямс», 2011. – 1056 с.: ил.
6. Полное руководство по языку программирования С# 6.0 и платформе .NET 4.6 [Электронный ресурс] / Сайт о программировании. – metanit.com, 2012-2016. – Режим доступа: [http://metanit.com/sharp/ tutorial](http://metanit.com/sharp/%20tutorial). – Дата доступа: 20.08.2016.

Преподаватель Е.В. Багласова

|  |
| --- |
| Рассмотрено на заседании цикловой  комиссии программного обеспечения информационных технологий №10  Протокол №\_\_\_\_\_от «\_\_\_»\_\_\_\_2016  Председатель ЦК *( )* Т.Г.Багласова |