

**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**РОССИЙСКОЙ ФЕДЕРАЦИИ**

Брянский государственный технический университет

**Утверждаю**

**Ректор университета**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_О.Н.Федонин**

**«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_2017 г.**

**Программирование в среде Microsoft .NET**

**Форматирование и обработка текста**

**Методические указания**

**к выполнению лабораторной работы №9**

**для студентов очной формы обучения по направлениям подготовки 02.03.03 – «Математическое обеспечение и администрирование информационных систем», 09.03.01 – «Информатика и вычислительная техника» и 09.03.04 – «Программная инженерия»**

**Брянск 2017**

УКД 004.43

Программирование в среде Microsoft .Net. Форматирование и обработка текста [Текст] + [Электронный ресурс]: Методические указания к выполнению лабораторной работы №9 для студентов очной формы обучения по направлениям подготовки 02.03.03 – «Математическое обеспечение и администрирование информационных систем», 09.03.01 – «Информатика и вычислительная техника» и 09.03.04 – «Программная инженерия». – Брянск: БГТУ, 2017. – 31 с.

Разработал:

Д.А.Коростелёв

канд. техн. наук, доц.

Рекомендовано кафедрой «Информатика и программное обеспечение» БГТУ (протокол №2 от 16.09.2016)

**Методические издания публикуются в авторской редакции**

**Цель работы**

Целью работы является изучение основ форматирования и обработки текса с помощью библиотеки классов среды Microsoft .Net.

Продолжительность работы – 2 часа.

**Региональные настройки**

Мир поделен на множество культур и регионов, и приложения должны уметь распознавать эти культурные и региональные различия. Под культурой в программировании подразумевается набор предпочтений, отражающих языковые и культурные традиции пользователя. В документе RFC 1766 (www.ietf.org/rfc/rfc1766.txt) содержится полный список наименований культур, которые применяются по всему миру, в зависимости от языка и страны или региона. В качестве примера можно упомянуть en-AU, en-CA, en-GB и en-US для английского языка в Австралии, Канаде, Великобритании и Соединенных Штатах соответственно.

Наиболее важным классом в пространстве имен System.Globalization, пожалуй, является класс CultureInfo. Этот класс позволяет представлять культуру и определять календари, формат дат и чисел, а также применяемый в данной культуре порядок сортировки строк.

Класс RegionInfo позволяет представлять региональные особенности (такие как валюта), а также указывать, применяется ли в регионе метрическая система. В некоторых регионах может использоваться сразу несколько языков. Одним из примеров является регион Испании, в котором есть баскская (eu-ES), каталонская (ca-ES), испанская (es-ES) и галисийская (gl-ES) культуры.

Подобно тому, как в одном регионе может использоваться множество языков, на одном языке могут говорить во множестве регионов, например, на испанском говорят в Мексике, Испании, Гватемале, Аргентине, Перу, и не только.

При применении культур в .NET Framework необходимо различать специфические, нейтральные и инвариантные культуры.

Специфические культуры ассоциируются с реальными существующими культурами, которые перечислены в упомянутом в предыдущем разделе документе RFC 1766. Они могут отображаться на нейтральные культуры. Например, de может служить нейтральной версией для таких специфических культур, как de-AT, de-DE, de-CH и т.д. de представляет собой сокращенное обозначение немецкого языка, a AT, DE и СН — сокращенное обозначение таких стран, как Австрия, Германия и Швейцария.

При переводе приложений выполнять перевод для каждого региона обычно не требуется, поскольку между вариантами немецкого языка, применяемого в Германии и Австрии, например, никакой особой разницы не существует. Поэтому вместо специфических культур для локализации приложений, как правило, достаточно использовать и просто нейтральные культуры.

Инвариантные культуры не зависят от реальных культур. При сохранении форматированных чисел и дат в файлах или отправке их по сети на сервер использование культуры, не зависящей ни от каких параметров пользователя, является наилучшим вариантом.

При настройке культур необходимо проводить различие между культурой, которая должна использоваться для пользовательской интерфейса, и той, что должна применяться для форматирования чисел и дат. Все культуры ассоциируются с потоком, и наличие этих двух типов культур, соответственно, означает, что к потоку могут применяться два параметра культуры. Класс Thread имеет свойства CurrentCulture и CurrentUICulture. Свойство CurrentCulture предназначено для указания культуры, которая должна использоваться для форматирования и сортировки, а свойство CurrentUICulture — той, что должна применяться для языка пользовательского интерфейса.

Пользователи могут изменять предлагаемое по умолчанию значение для CurrentCulture с помощью опций, предлагаемых в доступной через окно панели управления оснастке Regional and Language Options (Язык и региональные стандарты). С помощью этих опций также можно изменять предлагаемые по умолчания значения для формата чисел, времени и дат.

Значение CurrentUICulture от этих опций не зависит. Оно зависит от языка операционной системы. Хотя есть одно исключение: в случае установки Windows 7, Windows Vista или Windows ХР с многоязыковым пользовательским интерфейсом (MUI) язык пользовательского интерфейса может изменяться с помощью настроек региональных стандартов, и в таком случае они будут влиять на значение свойства CurrentUICulture.

Все эти настройки обеспечивают очень хорошее поведение по умолчанию, которое в большинстве случаев изменять не требуется. При необходимости, однако, параметры культуры можно легко модифицировать, изменяя значения обоих отвечающих за культуру свойств потока, скажем, таким образом, чтобы они представляли испанскую культуру, как показано в следующем фрагменте кода:

System.Globalization.CultureInfo ci = new System.Globalization.CultureInfo("es-ES");

System.Threading.Thread.CurrentThread.CurrentCulture = ci;

System.Threading.Thread.CurrentThread.CurrentUICulture = ci;

Теперь, когда понятно, как устанавливать культуру, можно рассмотреть вопросы, связанные с форматированием чисел и дат, на которое оказывает влияние параметр CurrentCulture.

Числовые структуры Int16, Int32, Int64 и т.д. в пространстве имен System имеют перегруженные версии метода ToString(). Этот метод может применяться для создания различных представлений чисел в зависимости от локальных настроек. В случае структуры Int32 ToString() он имеет четыре таких перегруженных версии:

public string ToString();

public string ToString(IFormatProvider);

public string ToString(string);

public string ToString(string, IFormatProvider);

В версии без аргументов ToString() возвращает строку без всяких возможных вариантов форматирования. Однако этот метод может также принимать строку и класс, реализующий интерфейс IFormatProvider.

В строке указывается желаемый формат представления данных. Этот формат может представлять собой как стандартную, так и шаблонную строку форматирования чисел. В первом случае используются предопределенные обозначения: С — формат валюты; D — десятичный формат; Е — научный формат; F — формат с фиксированной точкой; G — общий формат; N — числовой формат; X — шестнадцатеричный формат.

В шаблонной строке можно задавать количество разрядов, разделители групп и дробной части, обозначение процентов и т.д. Например, шаблонная строка форматирования вида ###.### означает, что вывод должен генерироваться в формате трехзначных десятичных блоков с разделителем групп.

Интерфейс IFormatProvider реализован классами NumberFormatInfо, DateTimeFormatInfо и CultureInfo. Этот интерфейс имеет единственный метод GetFormat(), который возвращает объект формата.

Класс NumberFormatInfo может применяться для определения специализированных форматов для чисел. В случае применения в нем конструктора по умолчанию создается независимый от культуры или инвариантный объект. С помощью свойства NumberFormatInfo можно изменять все опции форматирования, например знак положительных чисел, символ процента, разделитель групп чисел, символ валюты и многое другое. Статическое свойство InvariantInfо возвращает доступный только для чтения, независимый от культуры объект NumberFormatInfo, а статическое свойство CurrentInfo — объект NumberFormatInfo, в котором значения форматирования основаны на значениях свойства CultureInfo текущего потока.

Создадим простой консольный проект. В этом проекте первый пример демонстрирует отображение числа в формате культуры текущего потока. Во втором примере используется метод ToString() с аргументом IFormatProvider. Класс CultureInfo реализует интерфейс IFormatProvider, поэтому далее создается объект CultureInfo с использованием культуры French. И, наконец, в третьем примере демонстрируется изменение культуры текущего потока. Здесь культура изменяется на German с помощью свойства CurrentCulture экземпляра Thread:

using System;

using System.Globalization;

using System.Threading;

namespace NumberAndDateFormatting

{

class Program

{

static void Main(string[] args)

{

NumberFormatDemo();

}

private static void NumberFormatDemo()

{

int val = 1234567890;

// Культура текущего потока

Console.WriteLine(val.ToString("N"));

// Интерфейс IFormatProvider

Console.WriteLine(val.ToString("N", new CultureInfo("fr-FR")));

// Изменение культуры потока

Thread.CurrentThread.CurrentCulture = new CultureInfo("de-DE");

Console.WriteLine(val.ToString("N"));

}

}

}

**Кодировки**

Кодирование — это процесс преобразования набора символов Юникода в последовательность байтов. Декодирование, наоборот, представляет собой процесс преобразования последовательности закодированных байтов в набор символов Юникода.

Для кодирования в .NET используется класс Encoding, который предназначен для выполнения преобразований между разными кодировками и Юникодом. Обычно для своего приложения следует выбирать один из производных классов Юникода.

Платформа .NET Framework предоставляет следующие реализации класса Encoding для обеспечения поддержки текущих кодировок Юникода и других кодировок:

* ASCIIEncoding кодирует символы Юникода как отдельные 7-разрядные ASCII-символы.
* UTF7Encoding кодирует символы Юникода с использованием кодировки UTF-7. Данная кодировка поддерживает все значения символов Юникода.
* UTF8Encoding кодирует символы Юникода с использованием кодировки UTF-8. Данная кодировка поддерживает все значения символов Юникода.
* UnicodeEncoding кодирует символы Юникода с использованием кодировки UTF-16.
* UTF32Encoding кодирует символы Юникода с использованием кодировки UTF-32.

Для получения других кодировок в приложениях используется метод GetEncoding. Для получения списка всех кодировок следует использовать метод GetEncodings.

Пример использования кодировок:

string unicodeString = "This string contains the unicode character Pi(\u03a0)";

// Создаем объекты двух кодировок.

Encoding ascii = Encoding.ASCII, unicode = Encoding.Unicode;

// Перевод строки в массив байт (byte[]).

byte[] unicodeBytes = unicode.GetBytes(unicodeString);

// Выполняем конвертацию из одной кодировки в другую.

byte[] asciiBytes = Encoding.Convert(unicode, ascii, unicodeBytes);

// Конвертация массива байт в массив символов и строку.

// Пример использования GetCharCount/GetChars.

char[] asciiChars = new char[ascii.GetCharCount(asciiBytes, 0, asciiBytes.Length)];

ascii.GetChars(asciiBytes, 0, asciiBytes.Length, asciiChars, 0);

string asciiString = new string(asciiChars);

// Отображаем результаты.

File.WriteAllText("unicode.txt", String.Format("Original string: {0}", unicodeString));

File.WriteAllText("ascii.txt", String.Format("Ascii converted string: {0}", asciiString));

**Шифрование**

В .NET Framework классы для шифрования находятся в пространстве имен System.Security.Cryptography и реализуют несколько симметричных и асимметричных алгоритмов. Здесь можно найти разные классы алгоритмов для множества различных целей.

Некоторые из классов, которые были добавлены в .NET 3.5, имеют префикс или суффикс Cng, который представляет собой сокращение от Cryptography Next Generation (Криптография следующего поколения; CNG) и может применяться, начиная с Windows Vista и Windows Server 2008 и далее. Такой API-интерфейс позволяет писать не зависящие от алгоритма программы за счет применения модели, основанной на поставщиках.

В случае, если целевой платформой является Windows Server 2003, то выбору классов для шифрования необходимо уделять особое внимание. В табл. 1 приведен перечень всех доступных в пространстве имен System.Security.Cryptography классов для шифрования с краткими описаниями их назначения:

**Таблица 1**

*Классы для шифрования*

|  |  |  |
| --- | --- | --- |
| Категория | Классы | Описание |
| Хеш-шифрование | MD5, MD5Cng, SHA1, SHA1Managed, SHA1Cng, SHA256, SHA256Managed, SHA256Cng, SHA384, SHA384Managed, SHA384Cng, SHA512, SHA512Managed, SHA512Cng | Задачей хеш-алгоритмов является создание хеш-значения фиксированной длины из двоичных строк произвольной длины. Эти алгоритмы применяются для создания цифровых подписей и для обеспечения целостности данных. В случае повторного хеширования той же бинарной строки возвращается такой же результат.  MD5 (Message Digest Algorithm 5) был разработан в RSA Laboratories и работает быстрее, чем SHA1. Алгоритм SHA1 является более устойчивым к атакам с применением методов грубой силы. Алгоритмы SHA разрабатывались в NSA (National Security Agency — Агентство национальной безопасности). В MD5 размер хеш-значения составляет 128 бит, а в SHA1 — 160 бит. В других алгоритмах SHA размер хеш-значения указан прямо в названии. Алгоритм SHA512 является самым надежным из всех, потому что подразумевает использование хеш-значения размером 512 бит, но при этом, конечно же, и работает медленнее всех остальных. |
| Симметричное шифрование | DES, DESCryptoServiceProvider, TripleDES, TripleDESCryptoServiceProvider, Aes, AesCryptoServiceProvider, AesManaged, RC2, RC2CryptoServiceProvider, Rijandel, RijandelManaged | Алгоритмы шифрования с помощью симметричного ключа предусматривают использование одного и того же ключа как для шифрования, так и для дешифрации данных. Алгоритм DES (Data Encryption Standard — стандарт шифрования данных) теперь считается небезопасным, поскольку основан на применении для ключа значения размером всего лишь 56 бит, которое может быть взломано меньше, чем за 24 часа.  Алгоритм Triple-DES является следующей версией DES и предусматривает использование ключей размером 168 бит, но эффективно обеспечивает безопасность только при 112 битах. Алгоритм AES (Advanced Encryption Standard — усовершенствованный стандарт шифрования) поддерживает ключи размером 128,192 и 256 бит.  Алгоритм Rijandel очень похож на AES, но просто предлагает больше вариантов для размеров ключей. Алгоритм AES был утвержден правительством США в качестве стандартного алгоритма для шифрования. |
| Ассиметричное шифрование | DSA, DSACryptoServiceProvider, ECDsa, ECDsaCng, ECDiffieHellman, ECDiffieHellmanCng, RSA, RSACryptoServiceProvider | Ассиметричные алгоритмы предусматривают применение для шифрования и дешифрации разных ключей. RSA (Rivest, Shamir, Adleman — Райвест, Шамир, Адлеман) был первым алгоритмом, который применялся для добавления подписей, а также шифрования. Он и по сей день широко используется в протоколах, связанных с областью электронной коммерции.  DSA (Digital Signature Algorithm — алгоритмцифровой подписи) является утвержденным федеральным правительством США стандартом для цифровых подписей.  ECDSA (Elliptic Curve DSA) и ECDiffieHellman используют алгоритмы, основанные на группах эллиптических кривых. Эти алгоритмы обеспечивают большую безопасность в случае применения более коротких ключей. Например, использование ключа длиной 1024 бита для DSA по уровню безопасности аналогично применению ключа длиной 160 бита для ECDSA. В результате получается, что ECDSA будет работать гораздо быстрее. ECDiffieHellman — это алгоритм, который применяется для обеспечения безопасного обмена секретными ключами по открытому каналу. |

Классы без суффикса Cng, Managed и CryptoServiceProvider, такие как MD5, являются абстрактными базовыми. Суффикс Managed означает, что данный алгоритм реализуется с помощью управляемого кода; другие классы могут служить оболочкой для вызовов собственного кода из API-интерфейса Windows. Суффикс CryptoServiceProvider присутствует в именах тех классов, которые реализуют абстрактный базовый класс, а суффикс Cng – в тех, которые предусматривают применение нового API-интерфейса Cryptography CNG.

Теперь давайте рассмотрим примеры того, как эти алгоритмы могут использоваться программно.

В первом примере демонстрируется создание подписи с использованием алгоритма ECDSA. В этом примере Алиса создает подпись, которая шифруется с помощью ее секретного ключа и может быть расшифрована с применением ее открытого ключа. Такой подход гарантирует, что подпись действительно принадлежит Алисе.

Сначала давайте посмотрим, как выглядят главные шаги в методе Main(): первым делом создаются ключи Алисы, далее производится подписание строки «Alice» и, наконец, осуществляется проверка того, действительно ли подпись принадлежит Алисе, за счет применения открытого ключа. Затем подписанное сообщение с помощью класса Encoding преобразуется в массив байтов. После этого для вывода зашифрованной подписи в окне консоли массив байтов, содержащий подпись, преобразуется в строку с помощью метода Convert.ToBase64String():

using System;

using System.Security.Cryptography;

using System.Text;

namespace SecurityCSharp

{

class Program

{

internal static CngKey aliceKeySignature;

internal static byte[] alicePubKeyBlob;

static void Main()

{

CreateKeys();

byte[] aliceData = Encoding.UTF8.GetBytes("Alice");

byte[] aliceSignature = CreateSignature(aliceData, aliceKeySignature);

Console.WriteLine("Alice created signature: {0}",

Convert.ToBase64String(aliceSignature));

if (VerifySignature(aliceData, aliceSignature, alicePubKeyBlob))

{

Console.WriteLine("Подпись Алисы была успешно проверена");

}

}

...

Новая пара ключей для Алисы создается с помощью метода CreateKeys(). Эта пара ключей сохраняется в статическом поле, чтобы к ней можно было получать доступ из других методов. Метод Create() класса CngKey в качестве аргумента получает алгоритм. С помощью метода Export() из пары ключей экспортируется открытый ключ. Этот открытый ключ может быть предоставлен Бобу, чтобы он мог проверять действительность подписи.

Секретный ключ остается у Алисы. Вместо того чтобы создавать пары ключей с помощью класса CngKey, можно также открыть существующие ключи из специального хранилища ключей. В реальных условия у Алисы, скорее всего, имеется сертификат, содержащий пару ключей и хранящийся в ее личном хранилище, к которому можно получить доступ с помощью метода CngKey.Open():

static void CreateKeys()

{

aliceKeySignature = CngKey.Create(CngAlgorithm.ECDsaP256);

alicePubKeyBlob = aliceKeySignature.Export(CngKeyBlobFormat.GenericPublicBlob);

}

Имея в распоряжении пару ключей, Алиса может создать подпись с помощью класса ECDsaCng. Конструктор этого класса принимает объект CngKey, в котором содержится открытый и секретный ключи. Далее этот секретный ключ используется для подписания данных вызовом метода SignData():

static byte[] CreateSignature(byte[] data, CngKey key)

{

byte[] signature;

var signingAlg = new ECDsaCng(key);

signature = signingAlg.SignData(data);

signingAlg.Clear();

return signature;

}

Для проверки, действительно ли подпись принадлежит Алисе, Боб извлекает ее с применением полученного от Алисы открытого ключа. Для этого сначала массив байтов, содержащий этот открытый ключ, импортируется в объект CngKey с помощью статического метода Import(), а затем для верификации подписи вызывается метод VerifyData() класса ECDsaCng:

static bool VerifySignature(byte[] data, byte[] signature, byte[] pubKey)

{

bool retValue = false;

using (CngKey key = CngKey.Import(pubKey, CngKeyBlobFormat.GenericPublicBlob))

{

var signingAlg = new ECDsaCng(key);

retValue = signingAlg.VerifyData(data, signature);

signingAlg.Clear();

}

return retValue;

}

**Регулярные выражения**

Регулярные выражения — это часть небольшой технологической области, невероятно широко используемой в огромном диапазоне программ. Регулярные выражения можно представить себе как мини-язык программирования, имеющий одно специфическое назначение: находить подстроки в больших строковых выражениях.

Это не новая технология, изначально она появилась в среде UNIX и обычно используется в языке программирования Perl. Разработчики из Microsoft перенесли ее в Windows, где до недавнего времени эта технология применялась в основном со сценарными языками. Однако теперь регулярные выражения поддерживаются множеством классов .NET из пространства имен System.Text.RegularExpressions. Случаи применения регулярных выражений можно встретить во многих частях среды .NET Framework. В частности, вы найдете их в серверных элементах управления проверкой ASP.NET.

Введение в регулярные выражения

Язык регулярных выражений предназначен специально для обработки строк. Он включает два средства:

* Набор управляющих кодов для идентификации специфических типов символов
* Система для группирования частей подстрок и промежуточных результатов таких действий

С помощью регулярных выражений можно выполнять достаточно сложные и высокоуровневые действия над строками:

* Идентифицировать (и возможно, помечать к удалению) все повторяющиеся слова в строке
* Сделать заглавными первые буквы всех слов
* Преобразовать первые буквы всех слов длиннее трех символов в заглавные
* Обеспечить правильную капитализацию предложений
* Выделить различные элементы в URI

Главным преимуществом регулярных выражений является использование метасимволов — специальные символы, задающие команды, а также управляющие последовательности, которые работают подобно управляющим последовательностям C#. Это символы, предваренные знаком обратного слеша (\) и имеющие специальное назначение.

В табл. 2 специальные метасимволы регулярных выражений C# сгруппированы по смыслу:

**Табл. 2**

*Метасимволы, используемые в регулярных выражениях C#*

|  |  |  |  |
| --- | --- | --- | --- |
| **Символ** | **Значение** | **Пример** | **Соответствует** |
| Классы символов | | | |
| [...] | Любой из символов, указанных в скобках | [a-z] | В исходной строке может быть любой символ английского алфавита в нижнем регистре |
| [^...] | Любой из символов, не указанных в скобках | [^0-9] | В исходной строке может быть любой символ кроме цифр |
| . | Любой символ, кроме перевода строки или другого разделителя Unicode-строки |  |  |
| \w | Любой текстовый символ, не являющийся пробелом, символом табуляции и т.п. |  |  |
| \W | Любой символ, не являющийся текстовым символом |  |  |
| \s | Любой пробельный символ из набора Unicode |  |  |
| \S | Любой непробельный символ из набора Unicode. Обратите внимание, что символы \w и \S - это не одно и то же |  |  |
| \d | Любые ASCII-цифры. Эквивалентно [0-9] |  |  |
| \D | Любой символ, отличный от ASCII-цифр. Эквивалентно [^0-9] |  |  |
| Символы повторения | | | |
| {n,m} | Соответствует предшествующему шаблону, повторенному не менее n и не более m раз | s{2,4} | "Press", "ssl", "progressss" |
| {n,} | Соответствует предшествующему шаблону, повторенному n или более раз | s{1,} | "ssl" |
| {n} | Соответствует в точности n экземплярам предшествующего шаблона | s{2} | "Press", "ssl", но не "progressss" |
| ? | Соответствует нулю или одному экземпляру предшествующего шаблона; предшествующий шаблон является необязательным | Эквивалентно {0,1} |  |
| + | Соответствует одному или более экземплярам предшествующего шаблона | Эквивалентно {1,} |  |
| \* | Соответствует нулю или более экземплярам предшествующего шаблона | Эквивалентно {0,} |  |
| Символы регулярных выражений выбора | | | |
| | | Соответствует либо подвыражению слева, либо подвыражению справа (аналог логической операции ИЛИ). |  |  |
| (...) | Группировка. Группирует элементы в единое целое, которое может использоваться с символами \*, +, ?, | и т.п. Также запоминает символы, соответствующие этой группе для использования в последующих ссылках. |  |  |
| (?:...) | Только группировка. Группирует элементы в единое целое, но не запоминает символы, соответствующие этой группе. |  |  |
| Якорные символы регулярных выражений | | | |
| ^ | Соответствует началу строкового выражения или началу строки при многострочном поиске. | ^Hello | "Hello, world", но не "Ok, Hello world" т.к. в этой строке слово "Hello" находится не в начале |
| $ | Соответствует концу строкового выражения или концу строки при многострочном поиске. | Hello$ | "World, Hello" |
| \b | Соответствует границе слова, т.е. соответствует позиции между символом \w и символом \W или между символом \w и началом или концом строки. | \b(my)\b | В строке "Hello my world" выберет слово "my" |
| \B | Соответствует позиции, не являющейся границей слов. | \B(ld)\b | Соответствие найдется в слове "World", но не в слове "ld" |

Безуcловно, задачу поиска и замены подстроки в строке можно решить на C# с использованием различных методов System.String и System.Text.StringBuilder. Однако в некоторых случаях это потребует написания большого объема кода C#. Если вы используете регулярные выражения, то весь этот код сокращается буквально до нескольких строк. По сути, вы создаете экземпляр объекта RegEx, передаете ему строку для обработки, а также само регулярное выражение (строку, включающую инструкции на языке регулярных выражений) — и все готово.

В табл. 3 показана часть информации о перечислении RegexOptions, экземпляр которого можно передать конструктору класса RegEx.

**Таблица 3**

*Структура перечисления RegexOptions*

|  |  |
| --- | --- |
| **Член** | **Описание** |
| CultureInvariant | Предписывает игнорировать национальные установки строки |
| ExplicitCapture | Модифицирует способ поиска соответствия, обеспечивая только буквальное соответствие |
| IgnoreCase | Игнорирует регистр символов во входной строке |
| IgnorePatternWhitespace | Удаляет из строки не защищенные управляющими символами пробелы и разрешает комментарии, начинающиеся со знака фунта или хеша |
| Multiline | Изменяет значение символов ^ и $ так, что они применяются к началу и концу каждой строки, а не только к началу и концу всего входного текста |
| RightToLeft | Предписывает читать входную строку справа налево вместо направления по умолчанию — слева направо (что удобно для некоторых азиатских и других языков, которые читаются в таком направлении) |
| Singleline | Специфицирует однострочный режим, в котором точка (.) символизирует соответствие любому символу |

После создания шаблона регулярного выражения с ним можно осуществить различные действия, в зависимости от того, что вам необходимо. Можно просто проверить, существует ли текст, соответствующий шаблону, в исходной строке. Для этого нужно использовать метод IsMatch(), который возвращает логическое значение (рис. 1)

using System;

using System.Text.RegularExpressions;

class Example

{

static void Main()

{

// Массив тестируемых строк

string[] test = {

"Wuck World", "Hello world", "My wonderful world"

};

// Проверим, содержится ли в исходных строках слово World

// при этом мы не укажем опции RegexOption

Regex regex = new Regex("World");

Console.WriteLine("Регистрозависимый поиск: ");

foreach (string str in test)

{

if (regex.IsMatch(str))

Console.WriteLine("В исходной строке: \"{0}\" есть совпадения!", str);

}

Console.WriteLine();

// Теперь укажем поиск, не зависимый от регистра

regex = new Regex("World", RegexOptions.IgnoreCase);

Console.WriteLine("РегистроНЕзависимый поиск: ");

foreach (string str in test)

{

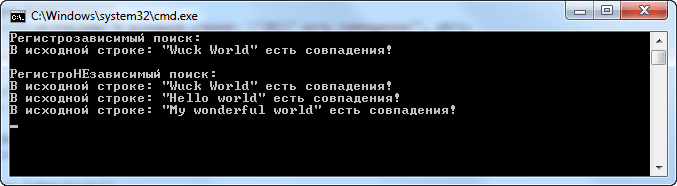
if (regex.IsMatch(str))

Console.WriteLine("В исходной строке: \"{0}\" есть совпадения!", str);

}

}

}



***Рис. 1. Результаты работы программы***

Если нужно вернуть найденное соответствие из исходной строки, то можно воспользоваться методом Match(), который возвращает объект класса Match, содержащий сведения о первой подстроке, которая сопоставлена шаблону регулярного выражения. В этом классе имеется свойство Success, которое возвращает значение true, если найдено следующее совпадение, которое можно получить с помощью вызова метода Match.NextMatch(). Эти вызовы метода можно продолжать пока свойство Match.Success не вернет значение false. Например (рис. 2):

using System;

using System.Text.RegularExpressions;

class Example

{

static void Main()

{

// Допустим в исходной строке нужно найти все числа,

// соответствующие стоимости продукта

string input = "Добро пожаловать в наш магазин, вот наши цены: " +

"1 кг. яблок - 20 руб. " +

"2 кг. апельсинов - 30 руб. " +

"0.5 кг. орехов - 50 руб.";

string pattern = @"\b(\d+\W?руб)";

Regex regex = new Regex(pattern);

// Получаем совпадения в экземпляре класса Match

Match match = regex.Match(input);

// отображаем все совпадения

while (match.Success)

{

// Т.к. мы выделили в шаблоне одну группу (одни круглые скобки),

// ссылаемся на найденное значение через свойство Groups класса Match

Console.WriteLine(match.Groups[1].Value);

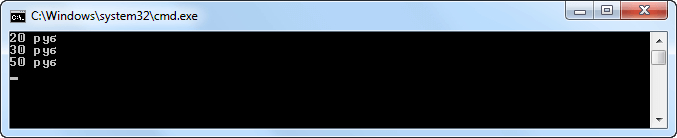
// Переходим к следующему совпадению

match = match.NextMatch();

}

}

}



***Рис. 2. Результаты работы программы***

Извлечь все совпадения можно и более простым способом, используя метод Regex.Matches(), который возвращает объект класса MatchCollection, который, в свою очередь, содержит сведения обо всех совпадениях, которые обработчик регулярных выражений находит во входной строке. Например, предыдущий пример может быть переписан для вызова метода Matches вместо метода Match и метода NextMatch:

using System;

using System.Text.RegularExpressions;

class Example

{

static void Main()

{

// Допустим в исходной строке нужно найти все числа,

// соответствующие стоимости продукта

string input = "Добро пожаловать в наш магазин, вот наши цены: " +

"1 кг. яблок - 20 руб. " +

"2 кг. апельсинов - 30 руб. " +

"0.5 кг. орехов - 50 руб.";

string pattern = @"\b(\d+\W?руб)";

Regex regex = new Regex(pattern);

// Достигаем того же результата что и в предыдущем примере,

// используя метод Regex.Matches() возвращающий MatchCollection

foreach (Match match in regex.Matches(input))

{

Console.WriteLine(match.Groups[1].Value);

}

}

}

Наконец, можно не просто извлекать совпадения в исходной строке, но и заменять их на собственные значения. Для этого используется метод Regex.Replace(). В качестве замены методу Replace() можно передавать как строку, так и шаблон замены. В табл. 4 показано как формируются метасимволы для замены.

**Таблица 4**

*Метасимволы замены в регулярных выражениях C#*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Символ** | **Описание** | **Пример шаблона** | **Пример шаблона замены** | **Результат (входная -> результирующая строки)** |
| $ number | Замещает часть строки, соответствующую группе number | \b(\w+)(\s)(\w+)\b | $3$2$1 | "один два" -> "два один" |
| $$ | Подставляет литерал "$" | \b(\d+)\s?USD | $$$1 | "103 USD" -> "$103" |
| $& | Замещает копией полного соответствия | (\$\*(\d\*(\.+\d+)?){1}) | \*\*$& | "$1.30" -> "\*\*$1.30\*\*" |
| $` | Замещает весь текст входной строки до соответствия | B+ | $` | "AABBCC" -> "AAAACC" |
| $' | Замещает весь текст входной строки после соответствия | B+ | $' | "AABBCC" -> "AACCCC" |
| $+ | Замещает последнюю захваченную группу | B+(C+) | $+ | "AABBCCDD" -> "AACCDD" |
| $\_ | Замещает всю входную строку | B+ | $\_ | "AABBCC" -> "AAAABBCCCC" |

Давайте рассмотрим метод Regex.Replace() на примере (рис. 3):

using System;

using System.Text.RegularExpressions;

class Example

{

static void Main()

{

// Допустим в исходной строке нужно заменить "руб." на "$",

// а стоимость переместить после знака $

string input = "Добро пожаловать в наш магазин, вот наши цены: \n" +

"\t 1 кг. яблок - 20 руб. \n" +

"\t 2 кг. апельсинов - 30 руб. \n" +

"\t 0.5 кг. орехов - 50 руб. \n";

Console.WriteLine("Исходная строка:\n {0}", input);

// В шаблоне используются 2 группы

string pattern = @"\b(\d+)\W?(руб.)";

// Строка замены "руб." на "$"

string replacement1 = "$$$1"; // Перед первой группой ставится знак $,

// вторая группа удаляется без замены

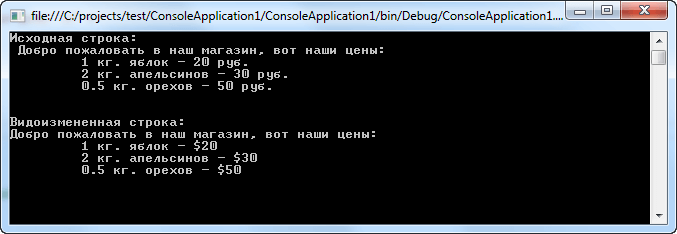
input = Regex.Replace(input, pattern, replacement1);

Console.WriteLine("\nВидоизмененная строка: \n" +input);

}

}

Для закрепления темы давайте рассмотрим еще один пример использования регулярных выражений, где будем искать в исходном тексте слово «сериализация» и его однокоренные слова, при этом выделяя в консоли их другим цветом (рис. 4).



***Рис. 3. Результаты работы программы***

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.RegularExpressions;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

string myText = @"Сериализация представляет собой процесс сохранения объекта на диске.

В другой части приложения или даже в совершенно отдельном приложении может производиться

десериализация объекта, возвращающая его в состояние, в котором он пребывал до сериализации.";

const string myReg = "со";

MatchCollection myMatch = Regex.Matches(myText,myReg);

Console.WriteLine("Все вхождения строки \"{0}\" в исходной строке: ",myReg);

foreach (Match i in myMatch)

Console.Write("\t"+i.Index);

// Усложним шаблон регулярного выражения

// введя в него специальные метасимволы

const string myReg1 = @"\b[с,д]\S\*ериализац\S\*";

MatchCollection match1 = Regex.Matches(myText,myReg1,RegexOptions.IgnoreCase);

findMyText(myText,match1);

Console.ReadLine();

}

static void findMyText(string text, MatchCollection myMatch)

{

Console.WriteLine("\n\nИсходная строка:\n\n{0}\n\nВидоизмененная строка:\n",text);

// Реализуем выделение ключевых слов в консоли другим цветом

for (int i = 0; i < text.Length; i++)

{

foreach (Match m in myMatch)

{

if ((i >= m.Index) && (i < m.Index+m.Length))

{

Console.BackgroundColor = ConsoleColor.Green;

Console.ForegroundColor = ConsoleColor.Black;

break;

}

else

{

Console.BackgroundColor = ConsoleColor.Black;

Console.ForegroundColor = ConsoleColor.White;

}

}

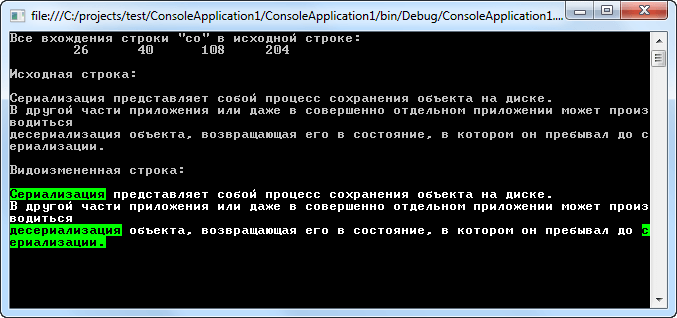
Console.Write(text[i]);

}

}

}

}



***Рис. 4. Результаты работы программы***

**Задание на лабораторную работу**

1. Разработать Windows Forms приложение для просмотра и сохранения файлов в разных кодировках. Программа должна позволять при отображении содержимого файла выбирать и менять кодировку, а также сохранять файл в выбранной кодировке.
2. Разработать программу, позволяющую выделять из файла информацию с использованием регулярных выражений в соответствии с вариантом (табл. 5).

**Таблица 5**

*Варианты заданий*

|  |  |
| --- | --- |
| №  варианта | Информация, которую необходимо найти в файле |
|  | Наименования всех не статических классов на языке C# |
|  | Наименования всех статических классов на языке C# |
|  | Наименования всех не статических свойств класса на языке C# |
|  | Наименования всех статических свойств класса на языке C# |
|  | Наименования всех не статических полей с данными класса на языке C# |
|  | Наименования всех статических полей с данными класса на языке C# |
|  | Наименования всех не статических методов класса на языке C# |
|  | Наименования всех статических методов класса на языке C# |
|  | Содержимое всех блоков get свойств класса на языке C# |
|  | Содержимое всех блоков set свойств класса на языке C# |
|  | Варианты для всех перечислений (enum) на языке C# |
|  | Все текстовые константы, используемые в программе на языке C# |
|  | Список аргументы с указанием их типов для всех методов класса на языке C# |
|  | Содержимое всех циклов типа for языка C# |
|  | Содержимое всех циклов типа while языка C# |
|  | Содержимое всех циклов типа do-while языка C# |
|  | Содержимое всех циклов типа foreach языка C# |
|  | Значения вариантов выбора case в блоке switch для языка C# |
|  | Все объявления локальных переменных для языка C# |
|  | Наименования всех событий класса на языке C# |

**Контрольные вопросы**

1. Какие классы и как используются для работы с регулярные выражения в .NET?
2. Какие классы и как используются для работы с кодировками?
3. Какие классы и как используются для работы с шифрованием?

# Список рекомендуемой литературы

1. Нейгел, К. C# 5.0 и платформа .NET 4.5 для профессионалов / К.Нейгел, Б.Ивьен, Дж.Глинн, М.Скиннер, К.Уотсон. – М.: Диалектика, 2013. – 1440 с.
2. Неш, Т. C# 2010: ускоренный курс для профессионалов / Т.Неш – М.: Вильямс, 2010. – 592 с.
3. Снелл, М. Microsoft Visual Studio 2008 / М.Снелл, Л.Пауэрс. – СПб: БХВ-Петербург, 2009. – 1200 с.
4. Троелсен, Э. Язык программирования С# 5.0 и платформа .NET 4.5 / Э.Троелсен. – 6-е изд. – М.: Вильямс, 2015. – 1312 с.
5. Рихтер, Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C# / Дж.Рихтер. – СПб.: Питер, 2016. – 896 с.

Программирование в среде Microsoft .Net. Форматирование и обработка текста [Текст] + [Электронный ресурс]: Методические указания к выполнению лабораторной работы №9 для студентов очной формы обучения по направлениям подготовки 02.03.03 – «Математическое обеспечение и администрирование информационных систем», 09.03.01 – «Информатика и вычислительная техника» и 09.03.04 – «Программная инженерия». – Брянск: БГТУ, 2017. – 31 с.

КОРОСТЕЛЁВ ДМИТРИЙ АЛЕКСАНДРОВИЧ

Научный редактор Д.И. Булатицкий

Компьютерный набор Д.А. Коростелёв

Иллюстрации Д.А. Коростелёв

Подписано в печать 07.07.2017 г. Формат 60х84 1/16 Бумага офсетная. Офсетная печать. Усл.печ.л. 0,63 Уч.-изд.л. 0,63 Тираж 1 экз.

Брянский государственный технический университет

Кафедра «Информатика и программное обеспечение», тел. 56-09-84

241035, Брянск, бульвар 50 лет Октября, 7 БГТУ