# Символы и строки

Обработка текстовой информации является одной из самых распространенных задач современного программировании. С# предоставляет для ее решения широкий набор средств: символы char, неизменяемые строки string, изменяемые строки StringBuider и регулярные выражения Regex. В данном разделе мы рассмотрим работу с символами, неизменяемыми и изменяемыми строками.

#### Символы char

Символьный тип char предназначен для хранения символа в кодировке Unicode. Символьный тип относится к встроенным типам данных С# и соответствует стандартному классу Сhar библиотеки .Net из пространства имен System. В этом классе определены статические методы, позволяющие задавать вид и категорию символа, а также преобразовывать символ в верхний или нижний регистр, в число. Рассмотрим основные методы:

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| GetNumericValue | Возвращает числовое значение символа, если он является цифрой, и -1 в противном случае. |
| GetUnicodeCategory | Возвращает категорию Unicode-символа. В Unicode символы разделены на категории, например цифры ( DecimalDigitNumber ), римские цифры ( LetterNumber ), разделители строк ( LineSeparator ), буквы в нижнем регистре ( LowercaseLetter ) и т.д. |
| IsControl | Возвращает true, если символ является управляющим. |
| IsDigit | Возвращает true, если символ является десятичной цифрой. |
| IsLetter | Возвращает true, если символ является буквой. |
| IsLetterOrDigit | Возвращает true, если символ является буквой или десятичной цифрой. |
| IsLower | Возвращает true, если символ задан в нижнем регистре. |
| IsNumber | Возвращает true, если символ является числом (десятичным или шестнадцатеричным). |
| IsPunctuation | Возвращает true, если символ является знаком препинания. |
| IsSeparator | Возвращает true, если символ является разделителем. |
| IsUpper | Возвращает true, если символ задан в верхнем регистре. |
| IsWhiteSpace | Возвращает true, если символ является пробельным (пробел, перевод строки, возврат каретки). |
| Parse | Преобразует строку в символ (строка должна состоять из одного символа). |
| ToLower | Преобразует символ в нижний регистр |
| ToUpper | Преобразует символ в верхний регистр |

В следующем примере рассмотрим применение данных методов:

static void Main()

{

try

{

char b = 'B', c = '\x64', d = '\uffff';

Console.WriteLine("{0}, {1}, {2}", b, c, d);

Console.WriteLine("{0}, {1}, {2}", char.ToLower(b), char.ToUpper(c), char.GetNumericValue(d));

char a;

do //цикл выполнятеся до тех пор, пока не ввели символ e

{

Console.WriteLine("Введите символ: ");

a = char.Parse(Console.ReadLine());

Console.WriteLine("Введен символ {0}, его код {1}, его категория {2}", a,

(int)a, char.GetUnicodeCategory(a));

if (char.IsLetter(a)) Console.WriteLine("Буква");

if (char.IsUpper(a)) Console.WriteLine("Верхний регистр");

if (char.IsLower(a)) Console.WriteLine("Нижний регистр");

if (char.IsControl(a)) Console.WriteLine("Управляющий символ");

if (char.IsNumber(a)) Console.WriteLine("Число");

if (char.IsPunctuation(a)) Console.WriteLine("Разделитель");

} while (a != 'e');

}

catch

{

Console.WriteLine("Возникло исключение");

}

}

Используя символьный тип можно организовать массив символов и работать с ним на основе базового класса Array:

static void Main()

{

char[] a ={ 'm', 'a', 'Х', 'i', 'M', 'u', 'S' , '!', '!', '!' };

char [] b="кол около колокола".ToCharArray(); //преобразование строки в массив символов

PrintArray("Исходный массив а:", a);

for (int x=0;x<a.Length; x++)

if (char.IsLower(a[x])) a[x]=char.ToUpper(a[x]);

PrintArray("Измененный массив а:", a);

PrintArray("Исходный массив b:", b);

Array.Reverse(b);

PrintArray("Измененный массив b:", b);

}

static void PrintArray(string line, Array a)

{

Console.WriteLine(line);

foreach( object x in a) Console.Write(x);

Console.WriteLine('\n');

}

#### Неизменяемые строки string

Тип string, предназначенный для работы со строками символов в кодировке Unicode, является встроенным типом С#. Ему соответствует базовый тип класса System.String библиотеки .Net. Каждый объект string - это неизменяемая последовательность символов Unicode, т.е. методы, предназначенные для изменения строк, возвращают измененные копии, исходные же строки остаются неизменными.

Создать строку можно несколькими способами:

1. string s; // инициализация отложена
2. string s=''кол около колокола''; //инициализация строковым литералом
3. string s=@'Привет!' //символ @ сообщает конструктору string, что Сегодня хорошая погода!!! '' // нужно воспринимать буквально, даже если она

// занимает несколько строк

1. string s=new string (' ', 20); //конструктор создает строку из 20 пробелов
2. int x = 12344556; //инициализировали целочисленную переменную
3. string s = x.ToString(); //преобразовали ее к типу string
4. char [] a={'a', 'b', 'c', 'd', 'e'}; //создали массив символов
5. string v=new string (a); // создание строки из массива символов
6. char [] a={'a', 'b', 'c', 'd', 'e'};

string v=new string (a, 0, 2) // создание строки из части массива символов, при

// этом: 0 показывает с какого символа, 2 - сколько символов

// использовать для инициализации

Класс string обладает богатым набором методов для сравнения строк, поиска в строке и других действий со строками. Рассмотрим эти методы.

|  |  |  |
| --- | --- | --- |
| **Название** | **Вид** | **Описание** |
| Compare | Статический метод | Сравнение двух строк в лексикографическом (алфавитном) порядке. Разные реализации метода позволяют сравнивать строки с учетом или без учета регистра. |
| CompareTo | Метод | Сравнение текущего экземпляра строки с другой строкой. |
| Concat | Статический метод | Слияние произвольного числа строк. |
| Copy | Статический метод | Создание копии строки |
| Empty | Статическое поле | Открытое статическое поле, представляющее пустую строку |
| Format | Статический метод | Форматирование строки в соответствии с заданным форматом |
| IndexOf, IndexOfAny, *LastIndexOf*, LastIndexOfAny | Экземплярные методы | Определение индексов первого и последнего вхождения заданной подстроки или любого символа из заданного набора в данную строку. |
| Insert | Экземплярный метод | Вставка подстроки в заданную позицию |
| Join | Статический метод | Слияние массива строк в единую строку. Между элементами массива вставляются разделители. |
| Length | Свойство | Возвращает длину строки |
| PadLeft, PadRigth | Экземплярные методы | Выравнивают строки по левому или правому краю путем вставки нужного числа пробелов в начале или в конце строки. |
| Remove | Экземплярный метод | Удаление подстроки из заданной позиции |
| Replace | Экземплярный метод | Замена всех вхождений заданной подстроки или символа новыми подстрокой или символом. |
| Split | Экземплярный метод | Разделяет строку на элементы, используя разные разделители. Результаты помещаются в массив строк. |
| StartWith, EndWith | Экземплярные методы | Возвращают true или false в зависимости от того, начинается или заканчивается строка заданной подстрокой. |
| Substring | *Экземплярный метод* | *Выделение подстроки*, начиная с заданной позиции |
| ToCharArray | *Экземплярный метод* | Преобразует строку в массив символов |
| ToLower, ToUpper | Экземплярные методы | Преобразование строки к нижнему или верхнему регистру |
| Trim, TrimStart, TrimEnd | Экземплярные методы | Удаление пробелов в начале и конце строки или только с одного ее конца. |

Напоминаем, что вызов статических методов происходит через обращение к имени класса, например, String.Concat(str1, str2), в остальных случаях через обращение к экземплярам класса, например, str.ToLower(). На примере рассмотрим использование данных свойств и методов.

static void Main()

{

string str1 ="Первая строка";

string str2 = string.Copy(str1);

string str3 = "Вторая строка";

string str4 = "ВТОРАЯ строка";

string strUp, strLow;

int result, idx;

Console.WriteLine("str1: " + str1);

Console.WriteLine("Длина строки str1: " +str1.Length);

// Создаем прописную и строчную версии строки str1.

strLow = str1.ToLower();

strUp = str1.ToUpper();

Console.WriteLine("Строчная версия строки str1: " +strLow);

Console.WriteLine("Прописная версия строки str1: " +strUp);

Console.WriteLine();

// Сравниваем строки,

result = str1.CompareTo(str3);

if (result == 0) Console.WriteLine("str1 и str3 равны.");

else if (result < 0) Console.WriteLine("str1 меньше, чем str3");

else Console.WriteLine("str1 больше, чем str3");

Console.WriteLine();

//сравниваем строки без учета регистра

result = String.Compare(str3,str4,true);

if (result == 0) Console.WriteLine("str3 и str4 равны без учета регистра.");

else Console.WriteLine("str3 и str4 не равны без учета регистра.");

Console.WriteLine();

//сравниваем части строк

result = String.Compare(str1, 4, str2, 4, 2);

if (result == 0) Console.WriteLine("часть str1 и str2 равны");

else Console.WriteLine("часть str1 и str2 не равны");

Console.WriteLine();

// Поиск строк.

idx = str2.IndexOf("строка");

Console.WriteLine("Индекс первого вхождения подстроки строка: " + idx);

idx = str2.LastIndexOf("о");

Console.WriteLine("Индекс последнего вхождения символа о: " + idx);

//конкатенация

string str=String.Concat(str1, str2, str3, str4);

Console.WriteLine(str);

//удаление подстроки

str=str.Remove(0,str1.Length);

Console.WriteLine(str);

//замена подстроки "строка" на пустую подстроку

str=str.Replace("строка","");

Console.WriteLine(str);

}

Очень важными методами обработки строк, являются методы разделения строки на элементы Split и слияние массива строк в единую строку Join.

static void Main()

{

string poems = "тучки небесные вечные странники";

char[] div = { ' '}; //создаем массив разделителей

// Разбиваем строку на части,

string[] parts = poems.Split(div);

Console.WriteLine("Результат разбиения строки на части: ");

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

Console.WriteLine(parts[i]);

// Теперь собираем эти части в одну строку, в качестве разделителя используем символ |

string whole = String.Join(" | ", parts);

Console.WriteLine("Результат сборки: ");

Console.WriteLine(whole);

}

**Задание**. Измените программу так, чтобы слова в предложении записывались в обратном порядке.

В общем случае строка может содержать и другие разделители:

static void Main()

{

string poems = "Тучки небесные, вечные странники...";

char[] div = { ' ', ',', '.'}; //создаем массив разделителей

// Разбиваем строку на части,

string[] parts = poems.Split(div);

Console.WriteLine("Результат разбиения строки на части: ");

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

Console.WriteLine(parts[i]);

// Теперь собираем эти части в одну строку,

string whole = String.Join(" | ", parts);

Console.WriteLine("Результат сборки: ");

Console.WriteLine(whole);

}

Рассмотрим другой пример - используя метод Split вводить двумерный массив можно не поэлементно, а построчно:

static void Main()

{

try

{

int[][] MyArray;

Console.Write("введите количество строк: ");

int n = int.Parse(Console.ReadLine());

MyArray = new int[n][];

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

{

string line = Console.ReadLine();

string[] mas = line.Split(' ');

MyArray[i] = new int[mas.Length];

for (int j = 0; j < MyArray[i].Length; j++)

{

MyArray[i][j] = int.Parse(mas[j]);

}

}

PrintArray("исходный массив:", MyArray);

for (int i = 0; i < MyArray.Length; i++) Array.Sort(MyArray[i]);

PrintArray("итоговый массив", MyArray);

}

catch

{

Console.WriteLine("возникло исключение");

}

}

static void PrintArray(string a, int[][] mas)

{

Console.WriteLine(a);

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

{

foreach (int x in mas[i]) Console.Write("{0} ", x);

Console.WriteLine();

}

}

В этом примере могут возникнуть исключительные ситуации, если введенная строка элементов массива будет содержать лишние пробелы. Следовательно, от этих пробелов нужно избавиться:

static void Main()

{

try

{

int[][] MyArray;

Console.Write("введите количество строк: ");

string line= Console.ReadLine()

int n = int.Parse(line.Trim());

MyArray = new int[n][];

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

{

line = Console.ReadLine();

line=line.Trim(); //удалаяем пробелы в начале и конце строки

//удаляем линшие пробелы внутри строки

n = line.IndexOf(" ");

while (n > 0)

{

line = line.Remove(n, 1);

n = line.IndexOf(" ");

}

string[] mas = line.Split(' ');

MyArray[i] = new int[mas.Length];

for (int j = 0; j < MyArray[i].Length; j++)

{

MyArray[i][j] = int.Parse(mas[j]);

}

}

PrintArray("исходный массив:", MyArray);

for (int i = 0; i < MyArray.Length; i++) Array.Sort(MyArray[i]);

PrintArray("итоговый массив", MyArray);

}

catch

{

Console.WriteLine("возникло исключение");

}

}

static void PrintArray(string a, int[][] mas)

{

Console.WriteLine(a);

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

{

foreach (int x in mas[i]) Console.Write("{0} ", x);

Console.WriteLine();

}

}

При работе с объектами класса string нужно учитывать их свойство неизменяемости, т.е. тот факт, что методы изменяют не сами строки, а их копии. Рассмотрим фрагмент программы:

string a="";

for (int i = 1; i <= 100; i++) a +="!";

Console.WriteLine(a);

В этом случае в памяти компьютера будет сформировано 100 различных строк вида:

!

!!

!!!

…

!!!...!!

И только последняя строка будет храниться в переменной а. Ссылки на все остальные строчки будут потеряны, но эти строки будут храниться в памяти компьютера и засорять память. Бороться с таким засорением придется сборщику мусора, что будет сказываться на производительности программы. Поэтому если нужно изменять строку, то лучше пользоваться классом StringBuilder.

#### Изменяемые строки

Чтобы создать строку, которую можно изменять, в С# предусмотрен класс StringBuilder, определенный в пространстве имен System.Text. Объекты этого класса всегда объявляются с явным вызовом конструктора класса (через операцию new ) . Примеры создания изменяемых строк:

StringBuilder a =new StringBuilder(); //создание пустой строки, размер по умолчанию 16 символов

//инициализация строки и выделение необходимой памяти

StringBuilder b = new StringBuilder("abcd");

//создание пустой строки и выделение памяти под 100 символов

StringBuilder с = new StringBuilder(100);

//инициализация строки и выделение памяти под 100 символов

StringBuilder d = new StringBuilder("abcd", 100);

//инициализация подстрокой "bcd", и выделение памяти под 100 символов

StringBuilder d = new StringBuilder("abcd", 1, 3,100);

Основные элементы класса приведены в таблице:

|  |  |  |
| --- | --- | --- |
| **Название** | **Вид** | **Описание** |
| Append | *Экземплярный метод* | Добавление данных в конец строки. Разные варианты метода позволяют добавлять в строку величины любых встроенных типов, массивы символов, строки и подстроки string. |
| AppendFormat | *Экземплярный метод* | Добавление форматированной строки в конец строки |
| Capacity | свойство | Получение и установка емкости буфера. Если устанавливаемое значение меньше текущей длины строки или больше максимального, то генерируется исключение ArgumentOutOfRangeException |
| Insert | *Экземплярный метод* | Вставка подстроки в заданную позицию |
| Length | изменяемое свойство | Возвращает длину строки. Присвоение ему значения 0 сбрасывает содержимое и очищает строку |
| MaxCapacity | неизменное свойство | Возвращает наибольшее количество символов, которое может быть размещено в строке |
| Remove | *Экземплярный метод* | Удаление подстроки из заданной позиции |
| Replace | *Экземплярный метод* | Замена всех вхождений заданной подстроки или символа новой подстрокой или символом |
| ToString | *Экземплярный метод* | Преобразование в строку типа string |
| Chars | изменяемое свойство | Возвращает из массива или устанавливает в массиве символ с заданным индексом. Вместо него можно пользоваться квадратными скобками [] |
| Equals | *Экземплярный метод* | Возвращает true, только если объекты имеют одну и ту же длину и состоят из одних и тех же символов |
| CopyTo | *Экземплярный метод* | Копирует подмножество символов строки в массив char |

Как видим, методы класса StringBuilder менее развиты, чем методы класса String, но они позволяют более эффективно использовать память за счет работы с изменяемыми строками. Рассмотрим примеры использования данных методов.

static void Main()

{

try

{

StringBuilder str=new StringBuilder("Площадь");

PrintString(str);

str.Append(" треугольника равна");

PrintString(str);

str.AppendFormat(" {0:f2} см ", 123.456);

PrintString(str);

str.Insert(8, "данного ");

PrintString(str);

str.Remove(7, 21);

PrintString(str);

str.Replace("а", "о");

PrintString(str);

StringBuilder str1=new StringBuilder(Console.ReadLine());

StringBuilder str2=new StringBuilder(Console.ReadLine());

Console.WriteLine(str1.Equals(str2));

}

catch

{

Console.WriteLine("Вознико исключение");

}

}

static void PrintString(StringBuilder a)

{

Console.WriteLine("Строка: "+a);

Console.WriteLine("Текущая длина строки " +a.Length);

Console.WriteLine("Объем буфера "+a.Capacity);

Console.WriteLine("Максимальный объем буфера "+a.MaxCapacity);

Console.WriteLine();

}

С изменяемой строкой можно работать не только как с объектом, но как с массивом символов:

static void Main()

{

StringBuilder a = new StringBuilder("2\*3=3\*2");

Console.WriteLine(a);

int k=0;

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

if (char.IsDigit(a[i])) k+=int.Parse(a[i].ToString());

Console.WriteLine(k);

}

На практике часто комбинируют работу с изменяемыми и неизменяемыми строками. Однако если необходимо изменять строку, то в этом случае используют StringBuilder.

**Пример**. Дана строка, в которой содержится осмысленное текстовое сообщение. Слова сообщения разделяются пробелами и знаками препинания. Вывести все слова сообщения, которые начинаются и заканчиваются на одну и ту же букву.

static void Main()

{

Console.WriteLine("Введите строку: ");

StringBuilder a = new StringBuilder(Console.ReadLine());

Console.WriteLine("Исходная строка: "+a);

for (int i=0; i<a.Length;)

if (char.IsPunctuation(a[i])) a.Remove(i,1);

else ++i;

string str=a.ToString();

string []s=str.Split(' ');

Console.WriteLine("Искомые слова: ");

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

if (s[i][0]==s[i][s.Length-1]) Console.WriteLine(s[i]);

}

# *Регулярные выражения*

Стандартный *класс* *string* позволяет выполнять над строками различные *операции*, в том числе *поиск*, замену, вставку и удаление подстрок. Тем не менее, есть классы задач по обработке символьной информации, где стандартных возможностей явно не хватает. Чтобы облегчить решение подобных задач, в *Net* Framework встроен более мощный аппарат работы со строками, основанный на регулярных выражениях.

Регулярные выражения предназначены для обработки текстовой информации и обеспечивают:

1. Эффективный поиск в тексте по заданному шаблону;
2. Редактирование текста;
3. Формирование итоговых отчетов по результатам работы с текстом.

Подробно рассмотрим первые два аспекта применения регулярных выражений.

#### Метасимволы в регулярных выражениях

Регулярное выражение - это шаблон, по которому выполняется поиск соответствующего фрагмента текста. Язык описания регулярных выражений состоит из символов двух видов: обычных символов и метасимволов. Обычный символ представляет в выражении сам себя, а метасимвол - некоторый *класс символов*.

Рассмотрим наиболее употребительные метасимволы:

|  |  |  |
| --- | --- | --- |
| **Класс символов** | **Описание** | **Пример** |
| . | Любой символ, кроме \n. | Выражение c. t соответствует фрагментам: cat, cut, c#t, c{t и т.д. |
| [] | Любой одиночный символ из последовательности, записанной внутри скобок. Допускается использование *диапазонов символов*. | Выражение c[*aui*]t соответствует фрагментам: cat, cut, *cit*. Выражение c[a-c]t соответствует фрагментам: cat, *cbt*, cct. |
| [^] | Любой одиночный символ, не входящий в последовательность, записанную внутри скобок. Допускается использование *диапазонов символов*. | Выражение c[^*aui*]t соответствует фрагментам: *cbt*, cct, c2t и т.д. Выражение c[^a-c]t соответствует фрагментам: cdt, *cet*, c%t и т.д. |
| \w | Любой алфавитно - цифровой символ. | Выражение c\wt соответствует фрагментам: *cbt*, cct, c2t и т.д., но не соответствует фрагментам c%t, c{t и т.д. |
| \W | Любой не алфавитно - цифровой символ. | Выражение c\Wt соответствует фрагментам: c%t, c{t, c. t и т.д., но не соответствует фрагментам *cbt*, cct, c2t и т.д. |
| \s | Любой пробельный символ. | Выражение \s\w\w\w\s соответствует любому слову из трех букв, окруженному пробельными символами. |
| \S | Любой не пробельный символ. | Выражение \s\S\S\S\s соответствует любым трем непробельным символам, окруженным пробельными. |
| \d | Любая десятичная цифра | Выражение c\dt соответствует фрагментам: c1t, c2t, c3t и т.д. |
| \D | Любой символ, не являющийся десятичной цифрой | Выражение c\Dt не соответствует фрагментам: c1t, c2t, c3t и т.д. |

Кроме метасимволов, обозначающие классы символов, могут применяться уточняющие метасимволы:

|  |  |
| --- | --- |
| **Уточняющие символы** | **Описание** |
| ^ | Фрагмент, совпадающий с регулярными выражениями, следует искать только в начале строки |
| $ | Фрагмент, совпадающий с регулярными выражениями, следует искать только в конце строки |
| \А | Фрагмент, совпадающий с регулярными выражениями, следует искать только в начале многострочной строки |
| \Z | Фрагмент, совпадающий с регулярными выражениями, следует искать только в конце многострочной строки |
| \b | Фрагмент, совпадающий с регулярными выражениями, начинается или заканчивается на границе слова, т.е. между символами, соответствующими метасимволам \w и \W |
| \B | Фрагмент, совпадающий с регулярными выражениями, не должен встречаться на границе слов |

В регулярных выражениях часто используются повторители - метасимволы, которые располагаются непосредственно после обычного символа или группы символов и задают количество его повторений в выражении.

|  |  |  |
| --- | --- | --- |
| **Повторители** | **Описание** | **Пример** |
| \* | Ноль или более повторений предыдущего элемента | Выражение ca\*t соответствует фрагментам: ct, cat, caat, caaat и т.д. |
| + | Одно или более повторений предыдущего элемента | Выражение ca+t соответствует фрагментам: cat, caat, caaat и т.д. |
| ? | Не более одного повторения предыдущего элемента | Выражение ca?t соответствует фрагментам: ct, cat. |
| {n} | Ровно n повторений предыдущего элемента | Выражение ca{3}t соответствует фрагменту: cаааt. Выражение (cat){2} соответствует фрагменту: c а tcat. |
| {n,} | По крайней мере n повторений предыдущего элемента | Выражение ca{3, }t соответствует фрагментам: c ааа t, caaaat, caaaaaaat и т.д. Выражение (cat){2, } соответствует фрагментам: catcat, catcatcat и т.д. |
| {n, m} | От n до m повторений предыдущего элемента | Выражение ca{2, 4}t соответствует фрагментам: c аа t, caaat, caaaat. |

Регулярное выражение записывается в виде строкового литерала, причем перед строкой необходимо ставить символ @, который говорит о том, что строку нужно будет рассматривать и в том случае, если она будет занимать несколько строчек на экране. Однако символ @ можно не ставить, если в качестве шаблона используется шаблон без метасимволов.

**Замечание**. Если нужно найти какой-то символ, который является метасимволом, например, точку, можно это сделать защитив ее обратным слэшем. Т.е. просто точка означает любой одиночный символ, а \. означает просто точку.

Примеры регулярных выражений:

1. слово rus -

@"rus" или "rus"

1. номер телефона в формате xxx-xx-xx - @"\d\d\d-\d\d-\d\d" или @"\d{3}(-\d\d){2}"
2. номер автомобиля - @"[A-Z]\d{3}[A-Z]{2}\d{2,3}RUS"

#### Поиск в тексте по шаблону

Пространство имен библиотеки базовых классов System.Text.RegularExpressions содержит все объекты платформы .NET Framework, имеющие отношение к регулярным выражениям. Важнейшим классом, поддерживающим регулярные выражения, является класс Regex, который представляет неизменяемые откомпилированные регулярные выражения. Для описания регулярного выражения в классе определено несколько перегруженных конструкторов:

1. Regex() - создает пустое выражение;
2. Regex(String) - создает заданное выражение;
3. Regex(String, RegexOptions) - создает заданное выражение и задает параметры для его обработки с помощью элементов перечисления RegexOptions (например, различать или нет прописные и строчные буквы).

Поиск фрагментов строки, соответствующих заданному выражению, выполняется с помощью методов IsMatch, Match, Matches класса Regex.

Метод IsMatch возвращает true, если фрагмент, соответствующий выражению, в заданной строке найден, и false в противном случае. Например, попытаемся определить, встречается ли в заданном тексте слово собака:

static void Main()

{

Regex r = new Regex("собака",RegexOptions.IgnoreCase);

string text1 = "Кот в доме, собака в конуре.";

string text2 = "Котик в доме, собачка в конуре.";

Console.WriteLine(r.IsMatch(text1));

Console.WriteLine(r.IsMatch(text2));

}

*Замечание*. RegexOptions.IgnoreCase - означает, что регулярное выражение применяется без учета регистра символов.

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

static void Main(string[] args)

{

Regex r = new Regex("собака|кот",RegexOptions.IgnoreCase);

string text1 = "Кот в доме, собака в конуре.";

string text2 = "Котик в доме, собачка в конуре.";

Console.WriteLine(r.IsMatch(text1));

Console.WriteLine(r.IsMatch(text2));

}

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

static void Main()

{

Regex r = new Regex(@"\d{2,3}(-\d\d){2}");

string text1 = "tel:123-45-67";

string text2 = "tel:no";

string text3 = "tel:12-34-56";

Console.WriteLine(r.IsMatch(text1));

Console.WriteLine(r.IsMatch(text2));

Console.WriteLine(r.IsMatch(text3));

}

Метод Match класса Regex не просто определяет, содержится ли текст, соответствующий шаблону, а возвращает объект класса Match - последовательность фрагментов текста, совпавших с шаблоном. Следующий пример позволяет найти все номера телефонов в указанном фрагменте текста:

static void Main()

{

Regex r = new Regex(@"\d{2,3}(-\d\d){2}");

string text = @"Контакты в Москве tel:123-45-67, 123-34-56; fax:123-56-45

Контакты в Саратове tel:12-34-56; fax:12-56-45";

Match tel = r.Match(text);

while (tel.Success)

{

Console.WriteLine(tel);

tel = tel.NextMatch();

}

}

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

static void Main()

{

Regex r = new Regex(@"[-+]?\d+");

string text = @"5\*10=50 -80/40=-2";

Match teg = r.Match(text);

int sum = 0;

while (teg.Success)

{

Console.WriteLine(teg);

sum += int.Parse(teg.ToString());

teg = teg.NextMatch();

}

Console.WriteLine("sum=" + sum);

}

Метод Matches класса Regex возвращает объект класса MatchCollection - коллекцию всех фрагментов заданной строки, совпавших с шаблоном. При этом метод Matches многократно запускает метод Match, каждый раз начиная поиск с того места, на котором закончился предыдущий поиск.

static void Main(string[] args)

{

string text = @"5\*10=50 -80/40=-2";

Regex theReg = new Regex(@"[-+]?\d+");

MatchCollection theMatches = theReg.Matches(text);

foreach (Match theMatch in theMatches)

{

Console.Write("{0} ", theMatch.ToString());

}

Console.WriteLine();

}

}

#### Редактирование текста

Регулярные выражения могут эффективно использоваться для редактирования текста. Например, метод Replace класса Regex позволяет выполнять замену одного фрагмента текста другим или удаление фрагментов текста:

**Пример** *1*. Изменение номеров телефонов:

static void Main(string[] args)

{

string text = @"Контакты в Москве tel:123-45-67, 123-34-56; fax:123-56-45.

Контакты в Саратове tel:12-34-56; fax:11-56-45";

Console.WriteLine("Старые данные\n"+text);

string newText=Regex.Replace(text, "123-", "890-");

Console.WriteLine("Новые данные\n" + newText);

}

**Пример 2**. Удаление всех номеров телефонов из текста:

static void Main()

{

string text = @"Контакты в Москве tel:123-45-67, 123-34-56; fax:123-56-45.

Контакты в Саратове tel:12-34-56; fax:12-56-45";

Console.WriteLine("Старые данные\n"+text);

string newText=Regex.Replace(text, @"\d{2,3}(-\d\d){2}", "");

Console.WriteLine("Новые данные\n" + newText);

}

}

**Пример** 3. Разбиение исходного текста на фрагменты:

static void Main()

{

string text = @"Контакты в Москве tel:123-45-67, 123-34-56; fax:123-56-45.

Контакты в Саратове tel:12-34-56; fax:12-56-45";

string []newText=Regex.Split(text,"[ ,.:;]+");

foreach( string a in newText)

Console.WriteLine(a);

}