bhv

C#

СОВЕТЫ ПРОГРАММИСТАМ

+ cd

Использование стандартных алгоритмов

Использование особенностей интегрированной среды разработки IDE

Работа с формами, элементами управления

Использование технологий WMI, WSH

Вызов системных функций Windows API

АЛЕКСАНДР КЛИМОВ

УДК 681.3.068+800.92C#
ББК 32.973.26-018.1
К49

Климов А. П.

К49 C#. Советы программистам. — СПб.: БХВ-Петербург, 2008. — 544 с.: ил. + CD-ROM

ISBN 978-5-9775-0174-3

Книга представляет собой сборник советов, алгоритмов и готовых примеров программ на языке C# в среде MS Visual Studio 2005/2008 из различных областей: работа с формами и элементами управления, папками и файлами, мышью и клавиатурой, мультимедиа и графикой, использование технологий WMI и WSH, взаимодействие с MS Office и другими приложениями, работа в локальной сети и Интернете, особенности использования функций Windows API и др.

На компакт-диске размещены примеры из книги, а также демонстрационная версия справочника по функциям Windows API для .NET Framework и сообщениям Windows для Visual Basic .NET и C#.

Для программистов

УДК 681.3.068+800.92C#
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 26.02.08.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 43,86.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

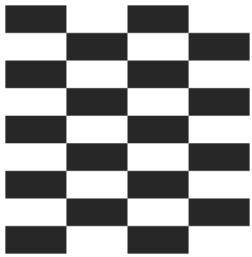
Санитарно-эпидемиологическое заключение на продукцию № 77.99.60.953.Д.002108.02.07
от 28.02.2007 г. выдано Федеральной службой по надзору
в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0174-3

Оглавление

Глава 2



Строки, даты, числа

В этой главе речь пойдет о базовых объектах, работать с которыми приходится практически в любом приложении.

Строки

Строки встречаются практически во всех приложениях, поэтому хорошее знание всех строковых методов и свойств необходимо любому программисту. Рассмотрим несколько советов. Необходимо помнить, что строки представлены классом `System.String` (или `string` в нотации C#) и любые операции, связанные со строками, на самом деле не изменяют саму строку, а создают новую строку. Поэтому во многих случаях лучше использовать класс `StringBuilder`, который в большей степени оптимизирован для манипуляции со строками.

Простейшие операции со строками

Для вставки одной строки в другую используется метод `Insert`, проиллюстрированный в листинге 2.1.

Листинг 2.1. Вставка одной строки в другую строку

```
string partBookTitle = "C#.советы";
string insertText = "Народные ";
string bookTitle = partBookTitle.Insert(3, insertText);
MessageBox.Show(bookTitle);
```

Для удаления подстроки из заданной строки используется метод `Remove`. Если вы хотите удалить подстроку с указанной позиции до конца строки, то достаточно указать индекс нужного символа, как это сделано в листинге 2.2.

Листинг 2.2. Удаление подстроки из заданной строки

```
string bookTitle = "C#.Народные советы";
// Удаляем подстроку с третьей позиции
bookTitle = bookTitle.Remove(2);
MessageBox.Show(bookTitle);
```

Для извлечения части строки из заданной строки используется метод `Substring`, применение которого показано в листинге 2.3.

Листинг 2.3. Извлечение подстроки из заданной строки

```
string bookTitle = "C#.Народные советы";
// Извлекаем подстроку с шестой позиции с размером в три символа
bookTitle = bookTitle.Substring(5, 3);
MessageBox.Show(bookTitle);
```

Примечание

Visual Basic.NET имеет в своем арсенале также такие методы и свойства, как `Left`, `Right`, `Len` и другие, унаследованные от Visual Basic 6.0. Вполне возможен вариант, что вам может попасться код, который использует эти устаревшие конструкции. Необходимо избавляться от подобных строчек кода, которым есть достойная альтернатива среди классов .NET Framework.

Входит ли строка в другую строку?

Метод `IndexOf`, проиллюстрированный в листинге 2.4, позволяет определить место вхождения подстроки в заданную строку.

Листинг 2.4. Определение вхождения подстроки в заданной строке

```
string str1 = "око";
string str2 = "Царь-колокол";
int i = str2.IndexOf(str1);
// Проверяем, входит ли строка око в слово Царь-колокол
if(i >= 0) MessageBox.Show(str1 + " входит в строку " + str2);
```

Также можно воспользоваться регулярным выражением, подключив пространство имен `System.Text.RegularExpressions`. В этом случае используется метод `Regex.IsMatch`. А если вы хотите написать собственный метод для нахождения строки в другой строке (для общего развития), то обратите внимание на статью "Find a string inside another string" на сайте **http://dirtydogstink.com/blog/2007/02/16/how2FindAStringInsideAnotherString.aspx**, чтобы не изобретать заново велосипед.

Преобразование строки в число

Не менее распространенная задача — преобразовать строку в число. В этом случае, как показано в листинге 2.5, можно использовать метод `Parse`.

Листинг 2.5. Преобразование строки в число

```
string tankman = "4";
string dog = "1";
textBox1.Text = tankman + dog; // получим 41
int all = int.Parse(tankman) + int.Parse(dog);
textBox1.Text += Environment.NewLine + all.ToString();  // получим 5
```

Как видите, сначала мы пытаемся сложить две строки, чтобы узнать численность экипажа танка из фильма "Четыре танкиста и собака". Результат, равный 41, нас явно не устроит. Поэтому сначала преобразуем строки в числа, а затем снова попытаемся сложить две переменные.

Существует еще один способ преобразования строки в число — методы класса `Convert`.

Вставка специального символа

Несмотря на то, что на клавиатуре имеется более 100 клавиш, некоторые символы на клавиатуре отсутствуют. Например, к разряду таких символов можно отнести знаки копирайта, знак зарегистрированной торговой марки, символы некоторых валют и т. д. В стандарте Unicode знак копирайта имеет десятичный код 169. В листинге 2.6 показано, как, используя класс `Convert`, можно преобразовать код нужного символа и получить нужный символ как часть строки, который затем можно вывести в текстовом поле (рис. 2.1).

Листинг 2.6. Вставка специального символа

```
int charCode = 169;
char specialChar = Convert.ToChar(charCode);
textBox1.Text = specialChar.ToString();
```

Можно использовать и шестнадцатеричное значение кода Unicode. Например, для вывода знака зарегистрированной торговой марки используем пример, приведенный в листинге 2.7.

Листинг 2.7. Вставка символа торговой марки

```
// Unicode-код для торговой марки
int charCode = 0x00AE;
char specialChar = Convert.ToChar(charCode);
textBox1.Text += specialChar.ToString();
```

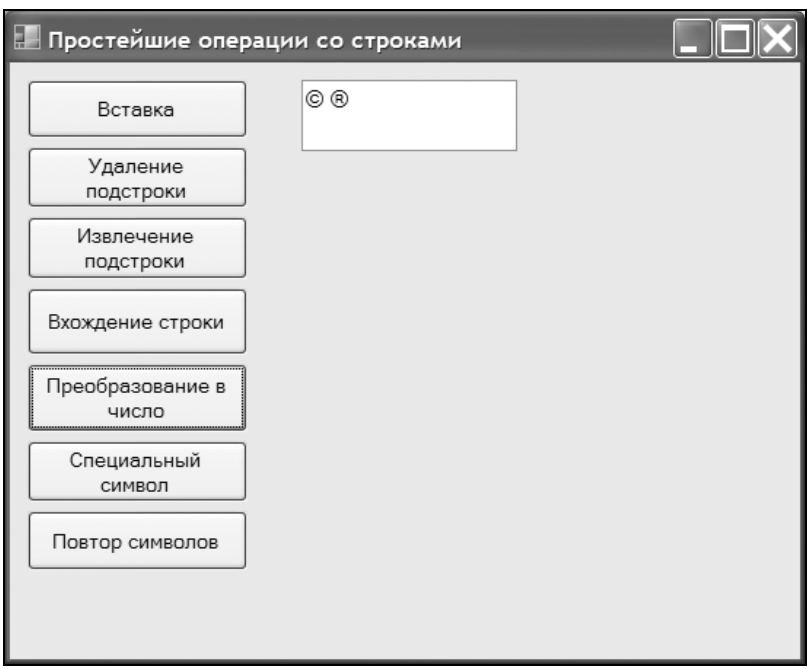


Рис. 2.1. Вывод специальных символов в текстовом поле

Создание строки из повторяющихся символов

Для создания строки, состоящей из одинаковых символов, достаточно (листинг 2.8) использовать простую конструкцию.

Листинг 2.8. Создание строки из повторяющихся символов

```
// Создаем строку из 5 звездочек
System.String FiveStars = new System.String('*', 5);
textBox1.Text = FiveStars;
```

Все остальные методы для работы со строками вы можете изучить самостоятельно. Комбинируя строковые методы, можно добиться некоторых интересных эффектов.

Метод *String.Format*

Не забывайте о возможности использовать метод `String.Format` при выводе строк для представления текста в нужном виде. Например, чтобы вывести строчку "Стоимость BMW равна 12 000 р.", можно использовать код, показанный в листинге 2.9.

Листинг 2.9. Использование метода `String.Format`

```
private void butFormatStr_Click(object sender, EventArgs e)
{
    string AutoName;
    AutoName = "BMW";
    textBox1.Text =
        String.Format("Стоимость {0} равна {1:0.0;c}", AutoName, 12000);
}
```

Преобразование строки в объект *Color*

Если у вас возникнет необходимость преобразовать строку с названием цвета в сам объект `Color`, то воспользуйтесь классами `TypeDescriptor` и `TypeConverter`, который предоставляет унифицированный способ конвертирования типов значений в другие типы. Например, мы хотим преобразовать цвет `Color.Blue` в строку (или наоборот, преобразовать строку `"Green"` в объект `Color`). Делается это, как показано в листинге 2.10.

Листинг 2.10. Преобразование строки в объект `Color` и обратно

```
private void butColorName_Click(object sender, EventArgs e)
{
    // Задаем цвет
    Color clr = Color.Blue;
    // Получим имя выбранного цвета
    textBox1.Text =
         (TypeDescriptor.GetConverter(clr).ConvertToString(clr));

    // Обратная задача. Конвертируем название цвета Green в объект Color
    clr = (Color)TypeDescriptor.GetConverter(
                            typeof(Color)).ConvertFromString("Green");
    // Закрашиваем форму в выбранный цвет
    this.BackColor = clr;

}
```

Проверка строки на пустоту

Существует несколько способов проверить строку на пустоту:

- `if (myString == "") // пустая строка;`
- `if (myString == String.Empty) // пустая строка;`
- `if (myString.Length == 0) // строка с нулевой длиной;`
- `if (String.Equals(myString, String.Empty)).`

Ян Нельсон (Ian Nelson) в своем блоге **http://ianfnelson.com/blog/archive/2004/07/30/171.aspx** не поленился и подсчитал, сколько времени занимает каждый из этих вариантов при обработке 50 миллионов итераций. Результаты вы можете посмотреть на сайте, но самым быстрым вариантом стал третий способ. Но данный способ не учитывает, что строка может принимать значение `null`. Эксперименты проводились в 2004 году. С появлением .NET Framework 2.0 класс `System.String` обзавелся новым статическим методом `IsNullOrEmpty` (листинг 2.11), который позволяет быстро и просто проверить строку — пустая она или имеет значение `null`.

Листинг 2.11. Проверка строки на пустоту

```
public void SayHello(string name)
{
```

```
    if ( string.IsNullOrEmpty(name) )
         throw new ArgumentNullException("name");
    MessageBox.Show( string.Concat("Hello, ", name) );
}
```

Переворачиваем строку

Иногда встречается задача перевернуть строку наоборот. В Visual Basic 6.0 появилась удобная строковая функция `StrReverse`, которая позволяла быстро и удобно перевернуть строку. Разработчики Visual Studio оставили эту функцию для программистов Visual Basic .NET. Таким образом, вам нужно установить ссылку на пространство имен `Microsoft.VisualBasic` и воспользоваться функцией по своему назначению (листинг 2.12).

Листинг 2.12. Переворачиваем строку при помощи функции Visual Basic

```
using Microsoft.VisualBasic;

private void butReverseVB_Click(object sender, EventArgs e)
{
    // Взять текст из текстового поля
    // Например, А роза упала на лапу Азора
    string myString   = textBox1.Text;
    // Используем встроенную функцию Visual Basic
    textBox1.Text = Strings.StrReverse(myString);
}
```

Если вы по каким-то причинам не хотите пользоваться библиотекой Visual Basic, можно написать собственную процедуру переворачивания строки, приведенную в листинге 2.13.

Листинг 2.13. Переворачиваем строку при помощи C#

```
public static string ReverseString(string str)
{
    // Проверка на непустоту строки.
    if(string.IsNullOrEmpty(str))
    {
        return str;
    }
```

```
    // Создадим объект StringBuilder с нужной длиной.
    StringBuilder revStr = new StringBuilder(str.Length);
    // Перебираем в цикле все символы
    // и присоединяем каждый символ к StringBuilder
    for (int count = str.Length - 1; count > -1; count--)
    {
        revStr.Append(str[count]);
    }
    // Возвращаем перевернутую строку
    return revStr.ToString();
}

private void butReverseCS_Click(object sender, EventArgs e)
{
    textBox1.Text = ReverseString(textBox1.Text);
}
```

Примечание

Примеры с простейшими операциями со строками вы можете найти в папке SimpleStrings на прилагаемом диске.

Сжатие длинных имен файлов

Если вам часто приходится работать со строками, которые представляют собой длинные полные пути к файлам, то возможно вам пригодится следующий пример. С помощью функции Windows API `PathCompactPathEx` можно заменить часть длинного пути к файлу многоточием, например, как это сделано в листинге 2.14.

Листинг 2.14. Замена длинного пути многоточием

```
using System.Runtime.InteropServices;

[DllImport("shlwapi.dll", CharSet = CharSet.Auto, SetLastError = true)]
private static extern bool PathCompactPathEx(
    System.Text.StringBuilder pszOut,
    string pszSrc,
    Int32 cchMax,
    Int32 dwFlags);
```

```
private void button1_Click(object sender, EventArgs e)
{
    // длинный путь к файлу
    string strPathFile =
              "c:/program files/My SuperProgram/skins/sample.txt";
    StringBuilder sb = new StringBuilder(260);

    // оставляем 20 символов, остальное заменяем многоточием
    bool b = PathCompactPathEx(sb, strPathFile, 20+1, 0);

    // Выводим результат в текстовое поле
    textBox1.Text = sb.ToString();
}
```

Примечание

Пример со сжатием строки находится в папке CompactPathDemo на прилагаемом диске.

Печатающийся текст

Чтобы создать видимость того, что текст печатается на печатной машинке, нужно использовать таймер, с помощью которого через определенный промежуток времени мы считываем все символы текста с первой буквы до текущей и выводим их на экран. Счетчик текущего количества букв постоянно обновляется (листинг 2.15), что позволяет повторять эффект печатающегося текста бесконечно, пока пользователь не остановит этот процесс.

Листинг 2.15. Эффект печатающегося текста

```
private void button1_Click(object sender, EventArgs e)
{
    if (button1.Text == "Старт")
    {
        timer1.Enabled = true;
        button1.Text = "Стоп";
    }
    else
    {
        timer1.Enabled = false;
```

```
        button1.Text = "Старт";
    }
}

public static int counter = 0;

private void timer1_Tick(object sender, EventArgs e)
{
    string typingText = "C#.Народные советы";

    this.Text = typingText.Substring(0,counter);
    counter++;

    if (counter > typingText.Length)
        counter = 0;
}
```

Бегущая строка

Второй распространенный эффект со строками — создание бегущей строки. Алгоритм создания эффекта бегущей строки заключается в следующем — из исходной строки удаляем один символ слева и добавляем его с правой стороны текста.

Листинг 2.16. Создание бегущей строки

```
private void butScroll_Click(object sender, EventArgs e)
{
    timer2.Enabled = true;
}

// Исходная строка. Для большего удобства добавлено несколько пробелов
// в конец строки
private string scrollText = "C#.Народные советы    ";

private void timer2_Tick(object sender, EventArgs e)
{
    // Удаляем один символ слева и прибавляем его с правой стороны
    scrollText = scrollText.Substring(1,
```

```
            (scrollText.Length - 1)) + scrollText.Substring(0, 1);
    this.Text = scrollText;
}
```

Примечание

Примеры находятся в проекте StringsFX на прилагаемом диске.

Как соединять строки

Во многих книжках пишут, что для сцепления строк нужно использовать класс `StringBuilder`, как самый эффективный способ.

Листинг 2.17. Соединение строк при помощи `StringBuilder`

```
StringBuilder sb = new StringBuilder();
sb.Append("строка 1");
sb.Append("строка 2");
this.Text = sb.ToString();
```

Однако, на самом деле, это не всегда самый лучший выбор. В одном из выпусков журнала "MSDN Magazine" была статья, в которой говорилось, что существуют случаи, когда использование класса `StringBuilder` для каждой операции сцепления является не самым быстрым решением. Использование класса `StringBuilder` наиболее эффективно, если сцепляется много строк, если число строк неизвестно или если строки длинные. Однако если в вашей программе сцепляются лишь несколько строк, или если они относительно коротки, то лучшим выбором часто является использование стандартной операции сцепления. В этих случаях можно использовать либо стандартную конструкцию сцепления используемого языка (для языка C# это символ +), либо метод `String.Concat`. Такие способы обеспечивают сравнимую производительность. Дело в том, что при использовании класса `StringBuilder` происходят дополнительные затраты ресурсов, которые не возникают при использовании стандартных строк. К тому же, класс `StringBuilder`, в конечном счете, необходимо все равно преобразовывать в строку, что опять приводит к дополнительным расходам. Следовательно, если вы занимаетесь оптимизацией вашего приложения, то вам необходимо убедиться в том, что используемые методы сцепления строк себя оправдывают. В указанной статье рассматривается пример сцепления переменного количества строк длиной 25 символов каждая и показываются графики зависимости времени, не-

обходимого для сцепления нескольких двадцатипятисимвольных строк с объектом класса `StringBuilder` и со строкой, от их количества. Результаты показывают, что если число сцеплений больше семи, лучше использовать класс `StringBuilder`, а если сцепляемых строк меньше, обычные способы соединения строк дают лучшую производительность.

Примечание

Более подробную информацию вы можете прочитать в статье "CLR Inside Out" в январском номере 2006 года журнала "MSDN Magazine". Существует также русскоязычная онлайн-версия статьи "CLR вдоль и поперек" на сайте **http://msdn.microsoft.com:80/msdnmag/issues/06/01/CLRInsideOut/default.aspx?loc=ru**.

Что лучше: *Parse* или *TryParse*?

В статье из предыдущего примера также сравниваются возможности методов `Parse` и `TryParse`. В .NET Framework 2.0 появился новый метод для базовых типов `TryParse`, который похож на метод `Parse`. Разница в их поведении заключается в том, что метод `Parse` при неудачной проверке данных вызывает исключение, а метод `TryParse` просто переходит к инструкции `else`. Нас интересует, что лучше использовать в программах для повышения производительности. Оказывается, что при успешном анализе значения строки `someString` производительность обоих методов в точности одинакова. Таким образом, использование метода `Parse` не приводит к дополнительным затратам, если точно известно, что имеющаяся строка соответствует тому типу, в который она преобразуется. Однако производительность падает, если вам попадется хотя бы одна строка, которая не может быть успешно преобразована. В этом случае при использовании метода `Parse` управление будет передано в блок обработки исключения. А исключения практически всегда замедляют работу. Если данная ситуация возникает не слишком часто, то это не очень страшно. В противном случае приходится платить значительным ухудшением производительности. В противоположность методу `Parse` метод `TryParse` при такой ситуации использует блок `if-else`. При переходе к блоку `else` производительность не страдает. Метод `TryParse` всегда работает с той же или большей производительностью, что и метод `Parse`, но разница между ними иногда может быть значительной. В статье приводятся такие цифры: в самом крайнем случае, когда ни одну строку не удается преобразовать в базовый тип, использование метода

`Parse` может занять в 150 раз больше времени, чем использование метода `TryParse`. С другой стороны, если анализируется 100 строк, и только одна из них не может быть успешно преобразована, метод `TryParse` работает всего лишь на 30 процентов быстрее. Таким образом, подтверждается общее правило, что не следует использовать исключения в качестве механизма управления логической последовательностью операций. Хотя использование метода `TryParse` более предпочтительно, если необходимо определить, соответствует ли строка другому типу данных, все же в некоторых ситуациях вместо него следует использовать метод `Parse`. Одной из них является ситуация, при которой ожидается, что анализ будет успешным, и, следовательно, неудачный анализ действительно является исключительным условием, а снижение производительности проблемы не составляет. Другой важный случай — когда необходимо определить конкретные причины неудачного преобразования, например, различить исключительные ситуации, связанные с переполнением и с неверным форматом. Для обоих примеров весьма подходящим является метод `Parse`. Исключение `FormatException`, возникающее в результате неудачи метода `Parse`, означает, что неудача была неожиданной, и случай действительно является исключительным по определению.

Еще раз советую перечитать статью, ссылка на которую дана в предыдущем разделе.

Сравнение и сортировка строк

И, наконец, в указанной выше статье сказано несколько слов о сравнении строк. Строки в среде .NET являются довольно мощным инструментом в том смысле, что их можно сравнивать с учетом и без учета особенностей культурной среды. По умолчанию большинство строковых операций подобные особенности учитывают. Если нет необходимости сравнивать строки с учетом особенностей культурной среды, то для сравнения строк всегда следует использовать сравнение по порядковым номерам. Это же касается и сортировки строк. Экономия при использовании этого способа сортировки или сравнения получается значительной. Для класса `System.String` существуют специальные строковые методы, например, `String.CompareOrdinal`, подчеркивающие ценность такого подхода. Если вам не нужно учитывать регистр строк при сравнении, но при этом важно сохранить производительность, следует воспользоваться методом `StringComparer.OrdinalIgnoreCase`. Он работает намного быстрее, чем метод, учитывающий особенности культурной