**Класс String**

Класс String определен в пространстве имен System. В нем реализуются следующие

интерфейсы: IComparable, IComparable<string>, ICloneable, IConvertible,

IEnumerable, IEnumerable<char> и IEquatable<string>. Кроме того, String —

герметичный класс, а это означает, что он не может наследоваться. В классе String

предоставляются все необходимые функциональные возможности для обработки символьных

строк в С#. Он служит основанием для встроенного в C# типа string и является

составной частью среды .NET Framework. В последующих разделах представлено

подробное описание класса String.

**Конструкторы класса String**

В классе String определено несколько конструкторов, позволяющих создавать

строки самыми разными способами. Для создания строки из символьного массива

служит один из следующих конструкторов.

public String(char[] value)

public String(char[] value, int startIndex, int length)

Первая форма конструктора позволяет создать строку, состоящую из символов массива

*value. А* во второй форме для этой цели из массива *value* извлекается определенное

количество символов (*length*), начиная с элемента, указываемого по индексу

*startIndex.*

С помощью приведенного ниже конструктора можно создать строку, состоящую из

отдельного символа, повторяющегося столько раз, сколько потребуется:

public String(char с, int count)

где *с* обозначает повторяющийся символ; a *count* — количество его повторений.

Кроме того, строку можно создать по заданному указателю на символьный массив,

используя один из следующих конструкторов.

public String(char\* value)

public String(char\* value, int startIndex, int length)

Первая форма конструктора позволяет создать строку из символов, доступных

из массива по указателю *value.* При этом предполагается, что массив, доступный

по указателю *value*, завершается пустым символом, обозначающим конец строки.

А во второй форме конструктора для этой цели из массива, доступного по указателю

*value*, извлекается определенное количество символов (*length*), начиная с элемента,

указываемого по индексу *startIndex.* В этих конструкторах применяются указатели,

поэтому их можно использовать только в небезопасном коде.

И наконец, строку можно построить по заданному указателю на байтовый массив,

используя один из следующих конструкторов.

public String(sbyte\* value)

public String(sbyte\* value, int startIndex, int length)

public String(sbyte\* value, int startIndex, int length, Encoding enc)

Первая форма конструктора позволяет построить строку из отдельных байтов символов,

доступных из массива по указателю *value.* При этом предполагается, что массив,

доступный по указателю *value*, завершается признаком конца строки. Во второй

форме конструктора для этой цели из массива, доступного по указателю *value,* извлекается

определенное количество байтов символов (*length*), начиная с элемента,

указываемого по индексу *startIndex. А* третья форма конструктора позволяет указать

количество кодируемых байтов. Класс Encoding находится в пространстве имен

System.Text. В этих конструкторах применяются указатели, и поэтому их можно использовать

только в небезопасном коде.

При объявлении строкового литерала автоматически создается строковый объект.

Поэтому для инициализации строкового объекта зачастую оказывается достаточно

присвоить ему строковый литерал, как показано ниже.

string str = "новая строка";

**Поле, индексатор и свойство класса String**

В классе String определено единственное поле.

public static readonly string Empty

Поле Empty обозначает пустую строку, т.е. такую строку, которая не содержит символы.

Этим оно отличается от пустой ссылки типа String, которая просто делается на

несуществующий объект.

Помимо этого, в классе String определен единственный индексатор, доступный

только для чтения.

public char this[int index] { get; }

Этот индексатор позволяет получить символ по указанному индексу. Индексация

строк, как и массивов, начинается с нуля. Объекты типа String отличаются постоянством

и не изменяются, поэтому вполне логично, что в классе String поддерживается

индексатор, доступный только для чтения.

И наконец, в классе String определено единственное свойство, доступное только

для чтения.

public int Length { get; }

Свойство Length возвращает количество символов в строке.

**Операторы класса String**

В классе String перегружаются два следующих оператора: == и !=. Оператор ==

служит для проверки двух символьных строк на равенство. Когда оператор == применяется

к ссылкам на объекты, он обычно проверяет, делаются ли обе ссылки на один и тот

же объект. А когда оператор == применяется к ссылкам на объекты типа String, то на

предмет равенства сравнивается содержимое самих строк. Это же относится и к оператору

!=. Когда он применяется к ссылкам на объекты типа String, то на предмет неравенства

сравнивается содержимое самих строк. В то же время другие операторы отношения,

в том числе < и >=, сравнивают ссылки на объекты типа String таким же образом, как

и на объекты других типов. А для того чтобы проверить, является ли одна строка больше

другой, следует вызвать метод Compare(), определенный в классе String.

Как станет ясно дальше, во многих видах сравнения символьных строк используются

сведения о культурной среде. Но это не относится к операторам = = и !=. Ведь они

просто сравнивают порядковые значения символов в строках. (Иными словами, они

сравнивают двоичные значения символов, не видоизмененные нормами культурной

среды, т.е. региональными стандартами.) Следовательно, эти операторы выполняют

сравнение строк без учета регистра и настроек культурной среды.

**Сравнение строк**

Вероятно, из всех операций обработки символьных строк чаще всего выполняется

сравнение одной строки с другой. При порядковом сравнении строки просто упорядочиваются

на основании невидоизмененного значения каждого символа.

В классе String предоставляются самые разные методы сравнения строк, перечисленные

в табл. 22.1. Наиболее универсальным среди них является метод Compare().

Он позволяет сравнивать две строки полностью или частично, с учетом или без учета

регистра, способа сравнения, определяемого параметром типа StringComparison,

а также сведений о культурной среде, предоставляемых с помощью параметра типа

CultureInfo. Те перегружаемые варианты метода Compare(), которые не содержат

параметр типа StringComparison, выполняют сравнение символьных строк с учетом

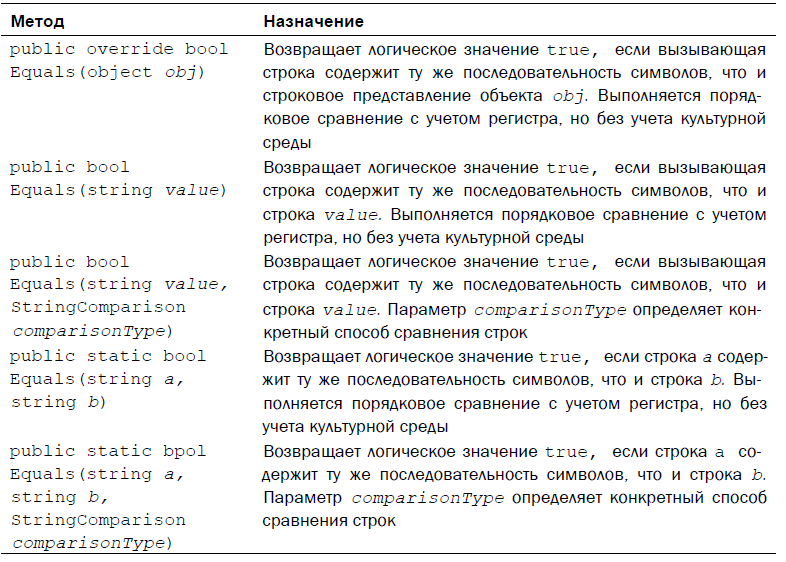
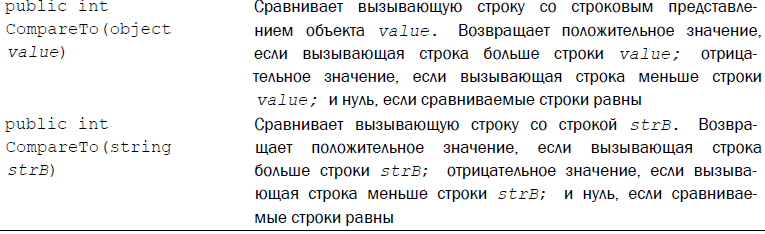
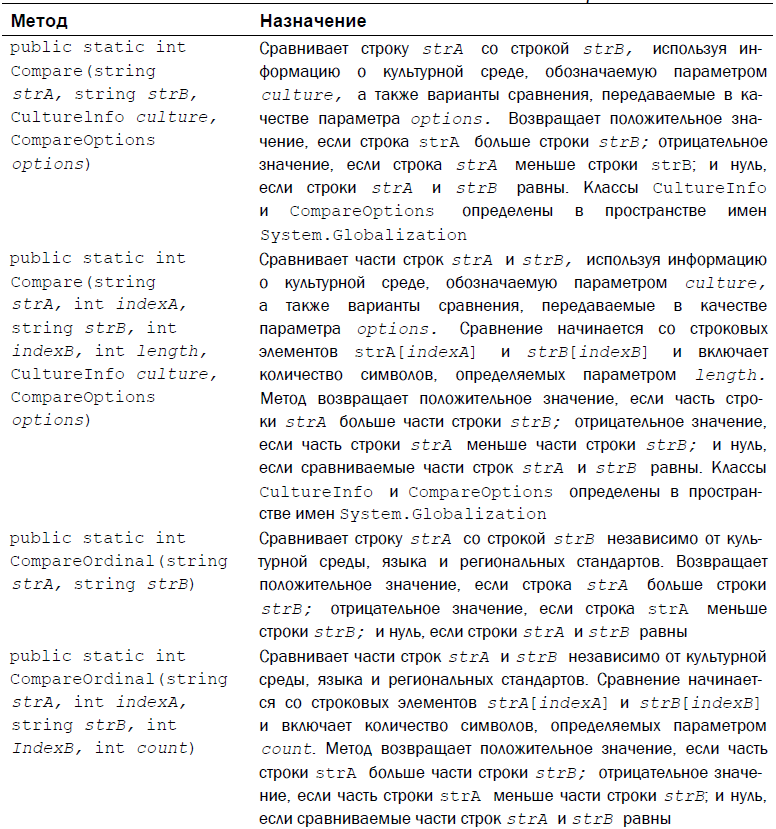
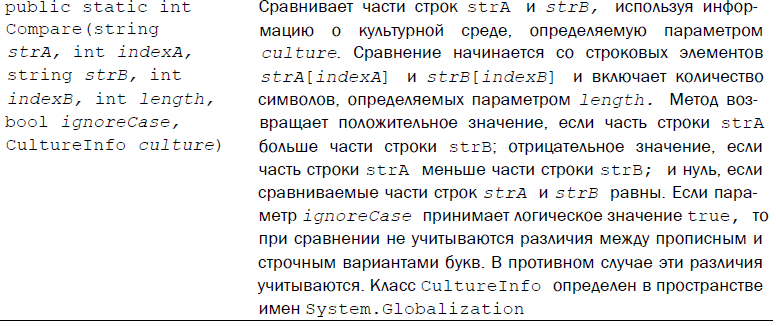
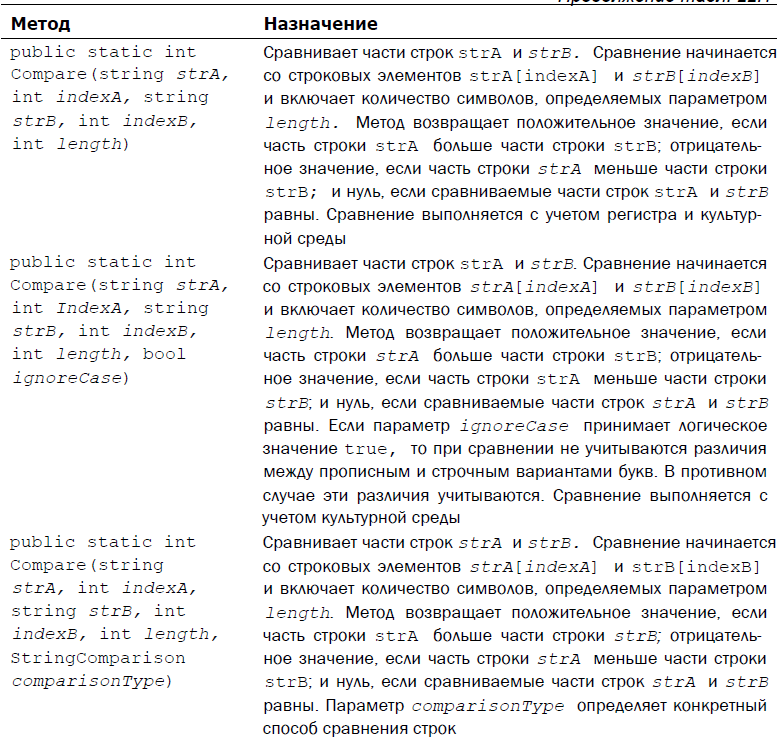
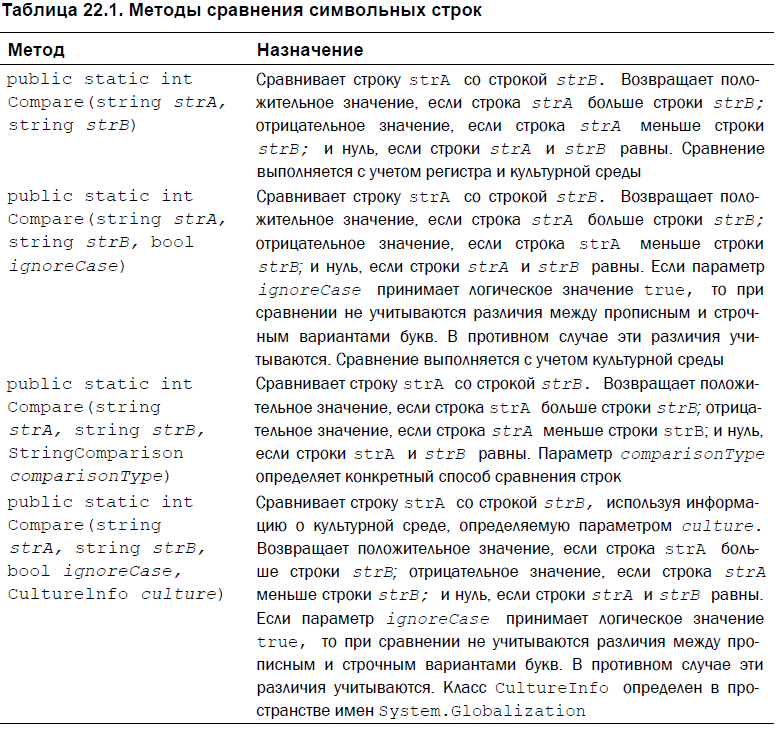
регистра и культурной среды. А в тех перегружаемых его вариантах, которые не содержат

параметр типа CultureInfo, сведения о культурной среде определяются текущей

средой выполнения. В примерах программ, приведенных в этой главе, параметр

типа CultureInfo не используется, а большее внимание уделяется использованию

параметра типа StringComparison.



Тип StringComparison представляет собой перечисление, в котором определяются

значения, приведенные в табл. 22.2. Используя эти значения, можно организовать сравнение

строк, удовлетворяющее потребностям конкретного приложения. Следовательно,

добавление параметра типа StringComparison расширяет возможности метода

Compare() и других методов сравнения, например, Equals(). Это дает также возможность

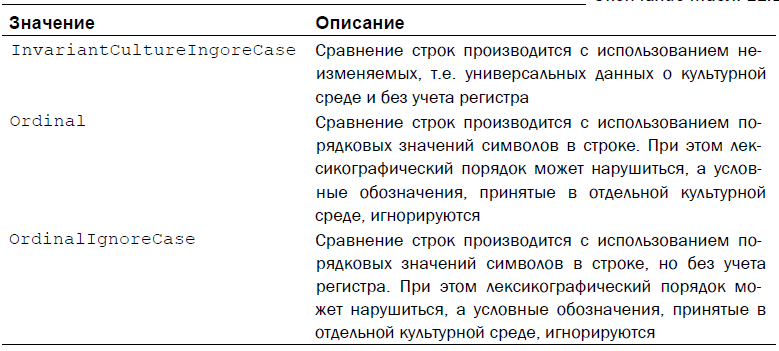
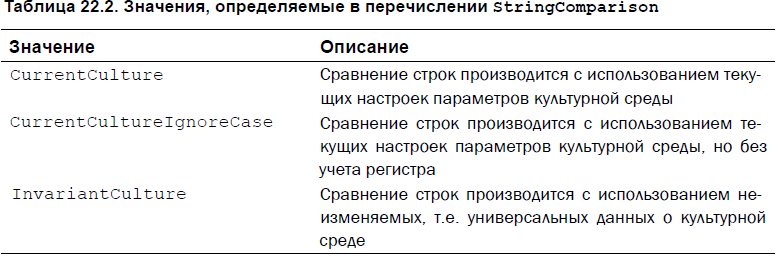
однозначно указывать способ предполагаемого сравнения строк. В силу имеющих

отличий между сравнением строк с учетом культурной среды и порядковым сравнением

очень важно быть предельно точным в этом отношении. Именно по этой причине

в примерах программ, приведенных в данной книге, параметр типа StringComparison

явно указывается в вызовах тех методов, в которых он поддерживается.



В любом случае метод Compare() возвращает отрицательное значение, если первая

сравниваемая строка оказывается меньше второй; положительное значение, если первая

сравниваемая строка больше второй; и наконец, нуль, если обе сравниваемые строки

равны. Несмотря на то что метод Compare() возвращает нуль, если сравниваемые

строки равны, для определения равенства символьных строк, как правило, лучше пользоваться

методом Equals() иди же оператором = =. Дело в том, что метод Compare()

определяет равенство сравниваемых строк на основании порядка их сортировки. Так,

если выполняется сравнение строк с учетом культурной среды, то обе строки могут

оказаться одинаковыми по порядку их сортировки, но не равными по существу. По

умолчанию равенство строк определяется в методе Equals(), исходя из порядковых

значений символов и без учета культурной среды. Следовательно, по умолчанию обе

строки сравниваются в этом методе на абсолютное, посимвольное равенство подобно

тому, как это делается в операторе = =.

Несмотря на большую универсальность метода Compare(), для простого порядкового

сравнения символьных строк проще пользоваться методом CompareOrdinal().

И наконец, следует иметь в виду, что метод CompareTo() выполняет сравнение строк

только с учетом культурной среды. На момент написания этой книги отсутствовали

перегружаемые варианты этого метода, позволявшие указывать другой способ сравнения

символьных строк.

В приведенной ниже программе демонстрируется применение методов Compare(),

Equals(), CompareOrdinal(), а также операторов = = и != для сравнения символьных

строк. Обратите внимание на то, что два первых примера сравнения наглядно демонстрируют

отличия между сравнением строк с учетом культурной среды и порядковым

сравнением в англоязычной среде.

**(glava22\_1)**

class CompareDemo

{

static void Main()

{

string str1 = "alpha";

string str2 = "Alpha";

string str3 = "Beta";

string str4 = "alpha";

string str5 = "alpha, beta";

int result;

//demonstrate difference betwen comparing string

//with cultural and order compare

result = String.Compare(str1, str2, StringComparison.CurrentCulture);

Console.Write("Comparising string with culture: ");

if (result < 0)

Console.WriteLine(str1 + " < " + str2);

else if (result > 0)

Console.WriteLine(str1 + " > " + str2);

else

Console.WriteLine(str1 + " = " + str2);

result = String.Compare(str1, str2, StringComparison.Ordinal);

Console.Write("Comparising ordinal: ");

if (result < 0)

Console.WriteLine(str1 + " < " + str2);

else if (result > 0)

Console.WriteLine(str1 + " > " + str2);

else

Console.WriteLine(str1 + " = " + str2);

//use method CompareOrdinal()

result = String.CompareOrdinal(str1, str2);

Console.Write("Compare with CompareOrdinal(): ");

if (result < 0)

Console.WriteLine(str1 + " < " + str2);

else if (result > 0)

Console.WriteLine(str1 + " > " + str2);

else

Console.WriteLine(str1 + " = " + str2);

Console.WriteLine();

//check equal string with operator ==

//this is ordinal comparing

if (str1 == str4) Console.WriteLine(str1 + " == " + str4);

//check incomparable with !=

if (str1 != str3) Console.WriteLine(str1 + " != " + str3);

if (str1 != str2) Console.WriteLine(str1 + " != " + str2);

Console.WriteLine();

//make ordinal comparing without register

//method Equals()

if (String.Equals(str1, str2, StringComparison.OrdinalIgnoreCase))

Console.WriteLine("Comparing with Equals() with parameter" +

" OrdinalIgnoreCase:\n" + str1 + " = " + str2);

Console.WriteLine();

if(String.Compare(str2, 0, str5, 0, 3, StringComparison.CurrentCulture) > 0)

{

Console.WriteLine("Comparing strings with culture:" +

"\n3 first symbols:" + str2 + " more, than 3 first of " + str5);

}

}

}

**Сцепление строк**

Строки можно сцеплять, т.е. объединять вместе, двумя способами. Во-первых, с помощью

оператора +, как было показано в главе 7. И во-вторых, с помощью одного из

методов сцепления, определенных в классе String. Конечно, для этой цели проще

всего воспользоваться оператором +, тем не менее методы сцепления служат неплохой

альтернативой такому подходу.

Метод, выполняющий сцепление строк, называется Concat(). Ниже приведена

одна из самых распространенных его форм.

public static string Concat(string str0, string str1)

Этот метод возвращает строку, состоящую из строки *str1,* присоединяемой путем

сцепления в конце строки *str0.* Ниже приведена еще одна форма метода Concat(),

в которой сцепляются три строки.

public static string Concat(string str0, string str1, string str2)

В данной форме метод Concat() возвращает строку, состоящую из последовательно

сцепленных строк *str0, str1* и *str2.*

Имеется также форма метода Concat(), в которой сцепляются четыре строки.

public static string Concat(string str0, string str1, string str2, string str3)

В этой форме метод Concat() возвращает строку, состоящую из четырех последовательно

сцепленных строк.

А в приведенной ниже еще одной форме метода Concat() сцепляется произвольное

количество строк:

public static string Concat(params string[] values)

где *values* обозначает переменное количество аргументов, сцепляемых для получения

возвращаемого результата. Если в этой форме метода Concat() допускается сцепление

произвольного количества строк, то зачем нужны все остальные его формы?

Они существуют ради повышения эффективности. Ведь передача методу от одного до

четырех аргументов оказывается намного эффективнее, чем использование для этой

цели переменного списка аргументов.

В приведенном ниже примере программы демонстрируется применение метода

Concat() в форме с переменным списком аргументов.

(***glava22\_2***)

class ConcatDemo

{

static void Main()

{

string result = String.Concat("This", "is", "test", "of", "method");

Console.WriteLine("Result: " + result);

}

}

Кроме того, существуют варианты метода Concat(), в которых он принимает

в качестве параметров ссылки на объекты, а не на строки. В этих вариантах метод

Concat() получает строковые представления вызывающих объектов, а возвращает

объединенную строку, сцепленную из этих представлений. (Строковые представления

объектов получаются с помощью метода ToString(), вызываемого для этих объектов.)

Ниже приведены все подобные варианты и формы метода Concat().

public static string Concat(object arg0)

public static string Concat(object arg0, object arg1)

public static string Concat(object arg0, object arg1, object arg2)

public static string Concat(object arg0, object arg1, object arg2, object arg3)

public static string Concat(params object[] args)

В первой форме метод Concat() возвращает строку, эквивалентную объекту *arg0,*

а в остальных формах — строку, получаемую в результате сцепления всех аргументов

данного метода. Объектные формы метода Concat(), т.е. относящиеся к типу object,

очень удобны, поскольку они исключают получение вручную строковых представлений

объектов перед их сцеплением. В приведенном ниже примере программы наглядно

демонстрируется польза от подобных форм метода Concat().

(***glava22\_3***)

class MyClass

{

public static int Count = 0;

public MyClass() { Count++; }

}

class ConcatDemo

{

static void Main()

{

string result = String.Concat("value is " + 19);

Console.WriteLine("Result: " + result);

result = String.Concat("hi ", 88, " ", 20.0, " ", false, " ", 23.45M);

Console.WriteLine("Result: " + result);

MyClass me = new MyClass();

result = String.Concat(me, " current count is ", MyClass.Count);

Console.WriteLine("Result: " + result);

}

}

В данном примере метод Concat() сцепляет строковые представления различных

типов данных. Для каждого аргумента этого метода вызывается соответствующий метод

ToString(), с помощью которого получается строковое представление аргумента.

Следовательно, в следующем вызове метода Concat():

string result = String.Concat("value is " + 19);

метод Int32.ToString() вызывается для получения строкового представления целого

значения 19, а затем метод Concat() сцепляет строки и возвращает результат.

Обратите также внимание на применение объекта определяемого пользователем

класса MyClass в следующем вызове метода Concat().

result = String.Concat(me, " current count is ", MyClass.Count);

В данном случае возвращается строковое представление объекта типа MyClass,

сцепленное с указываемой строкой. По умолчанию это просто имя класса. Но если

переопределить метод ToString(), то вместо строки с именем класса MyClass может

быть возвращена другая строка.

В версию 4.0 среды .NET Framework добавлены еще две формы метода Concat(),

приведенные ниже.

public static string Concat<T>(IEnumerable<T> values)

public static string Concat(IEnumerable<string> values)

В первой форме этого метода возвращается символьная строка, состоящая из сцепленных

строковых представлений ряда значений, имеющихся в объекте, который

обозначается параметром *values* и может быть объектом любого типа, реализующего

интерфейс IEnumerable<T>. А во второй форме данного метода сцепляются строки,

обозначаемые параметром *values.* (Следует, однако, иметь в виду, что если приходится

выполнять большой объем операций сцепления символьных строк, то для этой

цели лучше воспользоваться средствами класса StringBuilder.)

**Поиск в строке**

В классе String предоставляется немало методов для поиска в строке. С их помощью

можно, например, искать в строке отдельный символ, строку, первое или последнее

вхождение того и другого в строке. Следует, однако, иметь в виду, что поиск может

осуществляться либо с учетом культурной среды либо порядковым способом.

Для обнаружения первого вхождения символа или подстроки в исходной строке

служит метод IndexOf(). Для него определено несколько перегружаемых форм.

Ниже приведена одна из форм для поиска первого вхождения символа в исходной

строке.

public int IndexOf(char value)

В этой форме метода IndexOf() возвращается первое вхождение символа *value*

в вызывающей строке. Если символ *value* в ней не найден, то возвращается значение -1.

При таком поиске символа настройки культурной среды игнорируются. Следовательно,

в данном случае осуществляется порядковый поиск первого вхождения символа.

Ниже приведены еще две формы метода IndexOf(), позволяющие искать первое

вхождение одной строки в другой.

public int IndexOf(String value)

public int IndexOf(String value, StringComparison comparisonType)

В первой форме рассматриваемого здесь метода поиск первого вхождения строки,

обозначаемой параметром *value,* осуществляется с учетом культурной среды. А во второй

форме предоставляется возможность указать значение типа StringComparison,

обозначающее способ поиска. В если искомая строка не найдена, то в обеих формах

данного метода возвращается значение -1.

Для обнаружения последнего вхождения символа или строки в исходной строке

служит метод LastIndexOf(). И для этого метода определено несколько перегружаемых

форм. Ниже приведена одна из форм для поиска последнего вхождения символа

в вызывающей строке.

public int LastlndexOf(char value)

В этой форме метода LastIndexOf() осуществляется порядковый поиск, а в итоге

возвращается последнее вхождение символа *value* в вызывающей строке или же значение

-1, если искомый символ не найден.

Ниже приведены еще две формы метода LastIndexOf(), позволяющие искать последнее

вхождение одной строки в другой.

public int LastlndexOf(string value)

public int LastlndexOf(string value, StringComparison comparisonType)

В первой форме рассматриваемого здесь метода поиск последнего вхождения

строки, обозначаемой параметром *value,* осуществляется с учетом культурной

среды. А во второй форме предоставляется возможность указать значение типа

StringComparison, обозначающее способ поиска. Если же искомая строка не найдена,

то в обеих формах данного метода возвращается значение -1.

В классе String предоставляются еще два интересных метода поиска в строке:

IndexOfAny() и LastIndexOfAny(). Оба метода обнаруживают первый символ, совпадающий

с любым набором символов. Ниже приведены простейшие формы этих

методов.

public int IndexOfAny(char[] anyOf)

public int LastIndexOfAny(char[] anyOf)

Метод IndexOfAny() возвращает индекс первого вхождения любого символа из

массива *anyOf,* обнаруженного в вызывающей строке, а метод LastIndexOfAny() —

индекс последнего вхождения любого символа из массива *anyOf,* обнаруженного в вызывающей

строке. Если совпадение символов не обнаружено, то в обоих случаях возвращается

значение -1. Кроме того, в обоих рассматриваемых здесь методах осуществляется

порядковый поиск.

При обработке символьных строк нередко оказывается полезно знать, начинается

ли строка заданной подстрокой или же оканчивается ею. Для этой цели служат методы

StartsWith() и EndsWith(). Ниже приведены их простейшие формы.

public bool StartsWith(string value)

public bool EndsWith(string value)

Метод StartsWith() возвращает логическое значение true, если вызывающая

строка начинается с подстроки, переданной ему в качестве аргумента *value.* А метод

EndsWith() возвращает логическое значение true, если вызывающая строка оканчивается

подстрокой, переданной ему в качестве аргумента *value.* В противном случае

оба метода возвращают логическое значение false.

В обоих рассматриваемых здесь методах поиск осуществляется с учетом культурной

среды. Для того чтобы указать конкретный способ поиска подстроки, можно воспользоваться

приведенными ниже вариантами этих методов с дополнительным параметром

типа StringComparison.

public bool StartsWith(string value, StringComparison comparisonType)

public bool EndsWith(string value, StringComparison comparisonType)

Оба варианта рассматриваемых здесь методов поиска действуют таким же образом,

как и предыдущие их варианты. Но в то же время они позволяют явно указать конкретный

способ поиска.

В приведенном ниже примере программы демонстрируется применение нескольких

методов поиска в строке.

**(glava22\_4)**

class StringSearchDemo

{

static void Main()

{

string str = "C# has effective tools to manage strings.";

int idx;

Console.WriteLine("String str: " + str);

idx = str.IndexOf('o');

Console.WriteLine("Index first o: " + idx);

idx = str.LastIndexOf('o');

Console.WriteLine("Index last o: " + idx);

idx = str.IndexOf("ls", StringComparison.Ordinal);

Console.WriteLine("Index first \"ls\": " + idx);

idx = str.LastIndexOf("ls", StringComparison.Ordinal);

Console.WriteLine("Index last \"ls\": " + idx);

char[] chrs = { 'a', 'b', 'c' };

idx = str.IndexOfAny(chrs);

Console.WriteLine("Index first of 'a', 'b', 'c' :" + idx);

if (str.StartsWith("C# has", StringComparison.Ordinal))

Console.WriteLine("String str starts with \"C2 has\".");

if(str.EndsWith("strings.", StringComparison.Ordinal))

Console.WriteLine("String str ends with \"strings.\".");

}

}

Во многих случаях полезным для поиска в строке оказывается метод Contains().

Его общая форма выглядит следующим образом.

public bool Contains(string value)

Метод Contains() возвращает логическое значение true, если вызывающая строка

содержит подстроку, обозначаемую параметром *value*, в противном случае — логическое

значение false. Поиск указываемой подстроки осуществляется порядковым

способом. Этот метод особенно полезен, если требуется только выяснить, находится

ли конкретная подстрока в другой строке. В приведенном ниже примере программы

демонстрируется применение метода Contains().

(***glava22\_5***)

class StringSearchDemo

{

static void Main()

{

string str = "C# has effective tools to manage strings.";

if (str.Contains("C# has"))

Console.WriteLine("String str has \"C# has\".");

if (str.Contains("effe"))

Console.WriteLine("String str has \"effe\".");

}

}

Как следует из результата выполнения приведенной выше программы, метод

Contains() осуществляет поиск на совпадение произвольной последовательности

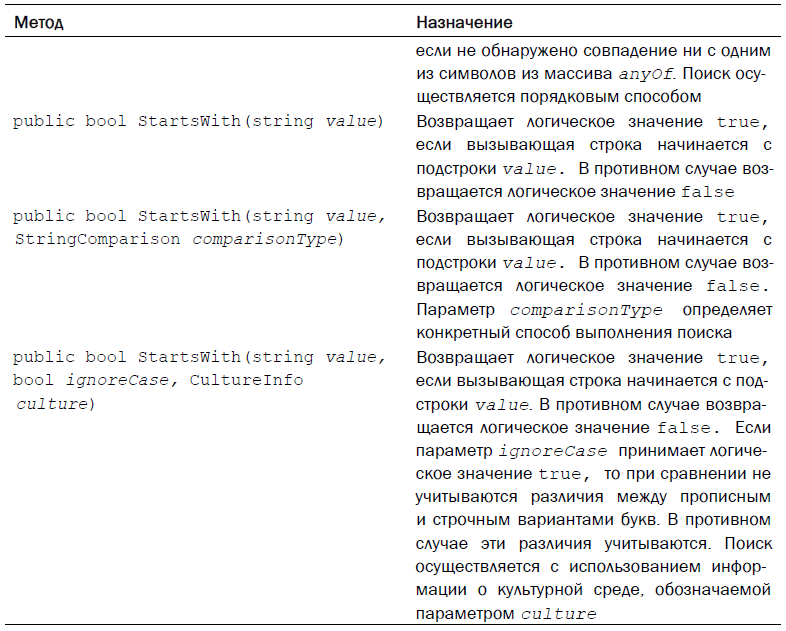
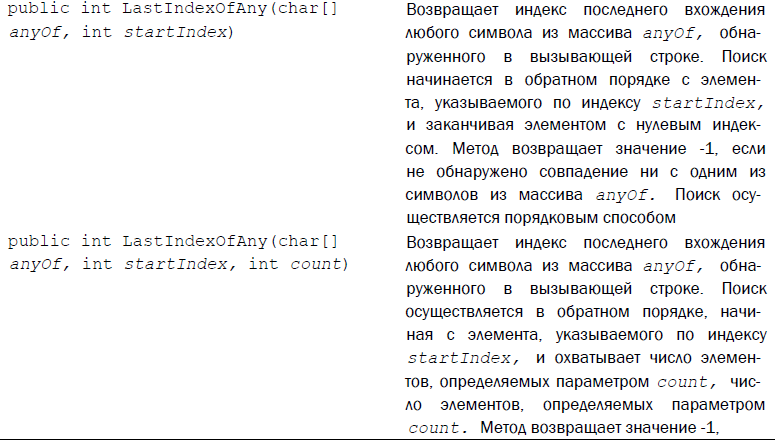
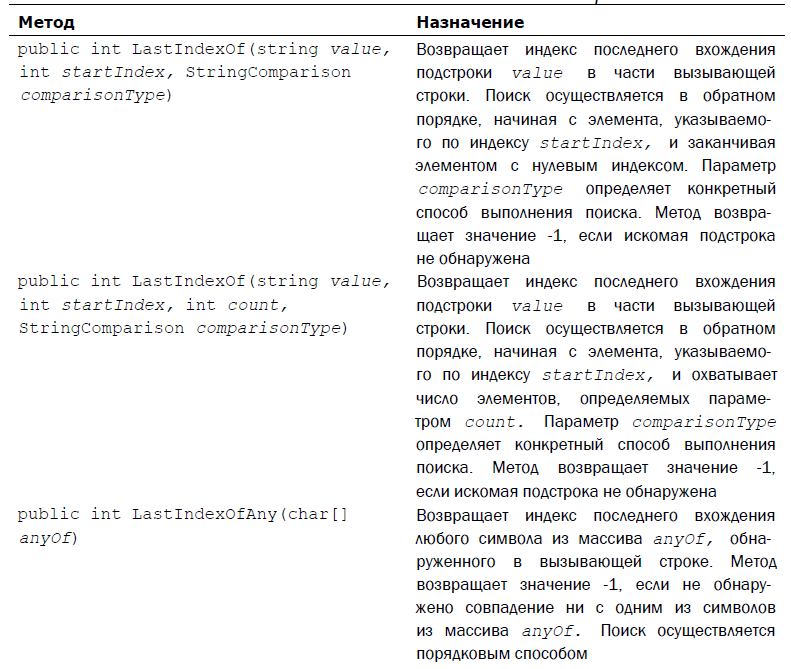
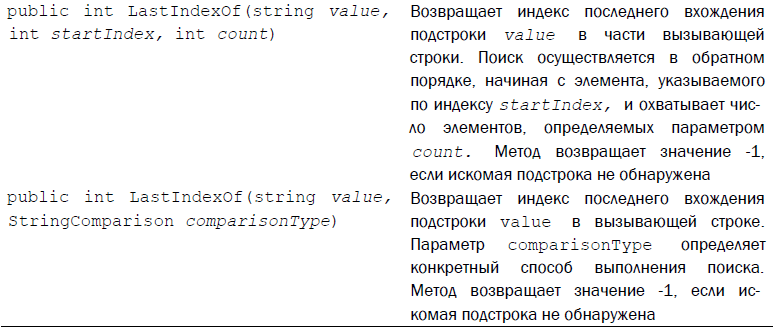
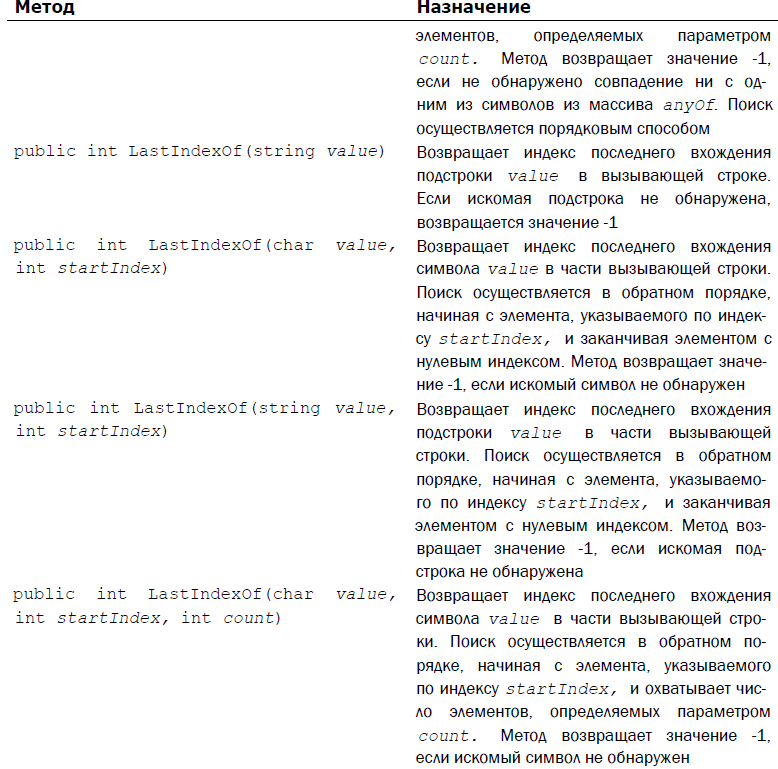
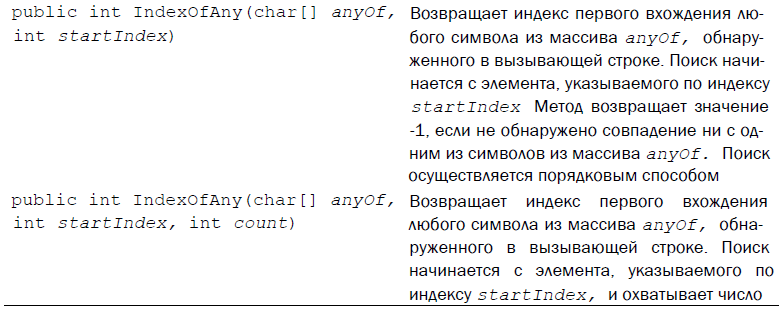
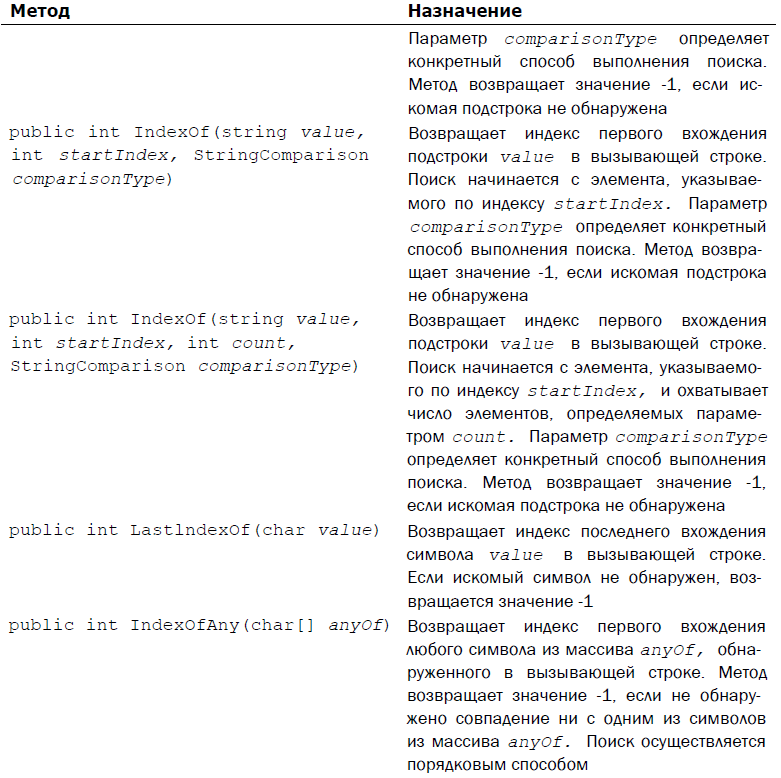
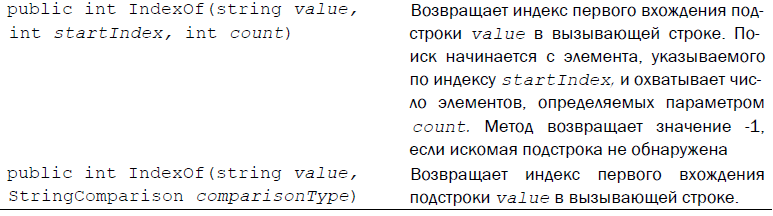
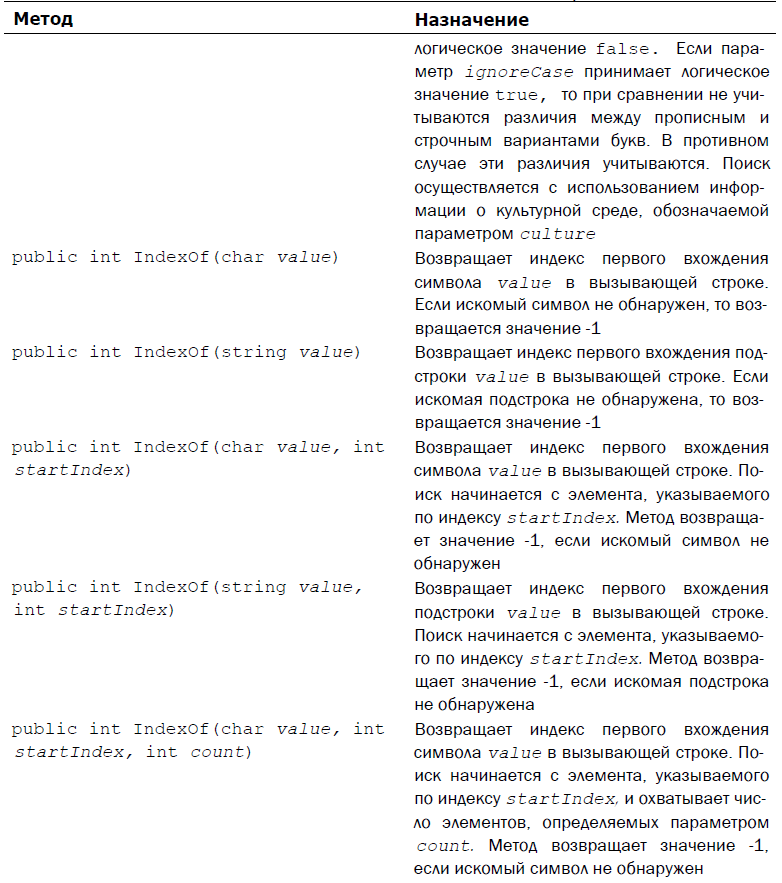
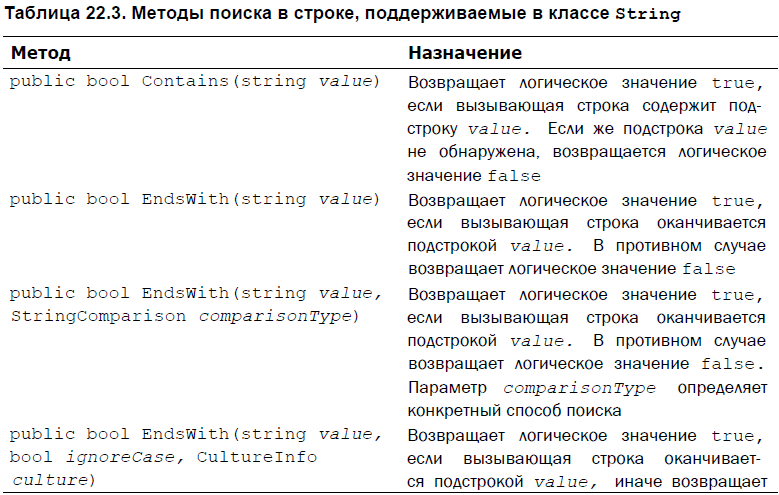
символов, а не только целых слов.

У некоторых методов поиска в строке имеются дополнительные формы, позволяющие

начинать поиск по указанному индексу или указывать пределы для поиска в строке.

В табл. 22.3 сведены все варианты методов поиска в строке, которые поддерживаются

в классе String.



**Разделение и соединение строк**

К основным операциям обработки строк относятся разделение и соединение. При

*разделении* строка разбивается на составные части, а при соединении строка составляется

из отдельных частей. Для разделения строк в классе String определен метод

Split(), а для соединения — метод Join().

Существует несколько вариантов метода Split(). Ниже приведены две формы

этого метода, ставшие наиболее часто используемыми, начиная с версии C# 1.0.

public string[] Split(params char[] separator)

public string[] Split(params char[] separator, int count)

В первой форме метода Split() вызывающая строка разделяется на составные

части. В итоге возвращается массив, содержащий подстроки, полученные из вызывающей

строки. Символы, ограничивающие эти подстроки, передаются в массиве

*separator.* Если массив *separator* пуст или ссылается на пустую строку, то в качестве

разделителя подстрок используется пробел. А во второй форме данного метода

возвращается количество подстрок, определяемых параметром *count.*

Существует несколько форм метода Join(). Ниже приведены две формы, ставшие

доступными, начиная с версии 2.0 среды .NET Framework.

public static string Join(string separator, string[] value)

public static string Join(string separator, string[] value,

int startIndex, int count)

В первой форме метода Join() возвращается строка, состоящая из сцепляемых

подстрок, передаваемых в массиве *value.* Во второй форме также возвращается строка,

состоящая из подстрок, передаваемых в массиве value, но они сцепляются в определенном

количестве *count,* начиная с элемента массива *value*[*startIndex*]*.* В обеих

формах каждая последующая строка отделяется от предыдущей разделительной

строкой, определяемой параметром *separator.*

В приведенном ниже примере программы демонстрируется применение методов

Split() и Join().

(***glava22\_6***)

class SplitAndJoinDemo

{

static void Main()

{

string str = "One at land, another at in sea.";

char[] seps = { ' ', '.', ',' };

//separate string apart

string[] parts = str.Split(seps);

Console.WriteLine("Results of separating: ");

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

Console.WriteLine(parts[i]);

//now join parts

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

Console.Write("Result of joining: ");

Console.WriteLine(whole);

}

}

Существует ряд других форм метода Split(), принимающих параметр типа

StringSplitOptions. Этот параметр определяет, являются ли пустые строки частью

разделяемой в итоге строки. Ниже приведены все эти формы метода Split().

public string[] Split(params char[] separator, StringSplitOptions options)

public string[] Split(string[] separator, StringSplitOptions options)

public string[] Split(params char[] separator, int count,

StringSplitOptions options)

public string[] Split(string[] separator, int count,

StringSplitOptions options)

В двух первых формах метода Split() вызывающая строка разделяется на части

и возвращается массив, содержащий подстроки, полученные из вызывающей строки.

Символы, разделяющие эти подстроки, передаются в массиве *separator.* Если

массив *separator* пуст, то в качестве разделителя используется пробел. А в третьей

и четвертой формах данного метода возвращается количество строк, ограничиваемое

параметром *count.* Но во всех формах параметр *options* обозначает конкретный способ

обработки пустых строк, которые образуются в том случае, если два разделителя

оказываются рядом. В перечислении StringSplitOptions определяются только два

значения: None и RemoveEmptyEntries. Если параметр *options* принимает значение

None, то пустые строки включаются в конечный результат разделения исходной строки,

как показано в предыдущем примере программы. А если параметр *options* принимает

значение RemoveEmptyEntries, то пустые строки исключаются из конечного

результата разделения исходной строки.

Разделение является очень важной процедурой обработки строк, поскольку с его

помощью нередко получают отдельные *лексемы,* составляющие исходную строку. Так,

в программе ведения базы данных может возникнуть потребность разделить с помощью

метода Split() строку запроса *"*показать все остатки больше 100*"* на отдельные

части, включая подстроки *"*показать*"* и *"*100*"*. В процессе разделения исключаются

разделители, поэтому в итоге получается подстрока *"*показать*"* (без начальных

и конечных пробелов), а не подстрока *"*показать*"*. Этот принцип демонстрируется

в приведенном ниже примере программы, где строки, содержащие такие бинарные

математические операции, как 10 + 5, преобразуются в лексемы, а затем эти операции

выполняются и выводится конечный результат.

(***glava22\_6***)

class TokenizeDemo

{

static void Main()

{

string[] input =

{

"100 + 19",

"100 / 3,3",

"-3 \* 9",

"100 - 87"

};

char[] seps = { ' ' };

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

{

//split string apart

string[] parts = input[i].Split(seps);

Console.Write("Command: ");

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

Console.Write(parts[j] + " ");

Console.Write(", result: ");

double n = Double.Parse(parts[0]);

double n2 = Double.Parse(parts[2]);

switch (parts[1])

{

case "+":

Console.WriteLine(n + n2);

break;

case "-":

Console.WriteLine(n - n2);

break;

case "\*":

Console.WriteLine(n \* n2);

break;

case "/":

Console.WriteLine(n / n2);

break;

}

}

}

}

Начиная с версии 4.0, в среде .NET Framework стали доступными следующие дополнительные

формы метода Join().

public static string Join(string separator, params object[] values)

public static string Join(string separator, IEnumerable<string>[] values)

public static string Join<T>(string separator, IEnumerable<T>[] values)

В первой форме рассматриваемого здесь метода возвращается строка, содержащая

строковое представление объектов из массива *values.* Во второй форме возвращается

строка, содержащая результат сцепления коллекции строк, обозначаемой параметром

*values.* И в третьей форме возвращается строка, содержащая результат сцепления

строковых представлений объектов из коллекции, обозначаемой параметром values.

Во всех трех случаях каждая предыдущая строка отделяется от последующей разделителем,

определяемым параметром *separator*.

**Заполнение и обрезка строк trim()**

Иногда в строке требуется удалить начальные и конечные пробелы. Такая операция

называется *обрезкой* и нередко требуется в командных процессорах. Например, программа

ведения базы данных способна распознавать команду *"*print*"*, но пользователь

может ввести эту команду с одним иди несколькими начальными и конечными пробелами.

Поэтому перед распознаванием введенной команды необходимо удалить все

подобные пробелы. С другой стороны, строку иногда требуется заполнить пробелами,

чтобы она имела необходимую минимальную длину. Так, если подготавливается вывод

результатов в определенном формате, то каждая выводимая строка должна иметь

определенную длину, чтобы сохранить выравнивание строк. Для упрощения подобных

операций в C# предусмотрены соответствующие методы.

Для обрезки строк используется одна из приведенных ниже форм метода Trim().

public string Trim()

public string Trim(params char[] trimChars)

В первой форме метода Trim() из вызывающей строки удаляются начальные и

конечные пробелы. А во второй форме этого метода удаляются начальные и конечные

вхождения в вызывающей строке символов из массива *trimChars.* В обеих формах

возвращается получающаяся в итоге строка.

Строку можно заполнить символами слева или справа. Для заполнения строки слева

служат такие формы метода PadLeft().

public string PadLeft(int totalWidth)

public string PadLeft(int totalWidth, char paddingChar)

В первой форме метода PadLeft() вводятся пробелы с девой стороны вызывающей

строки, чтобы ее общая длина стала равной значению параметра *totalWidth.* А во

второй форме данного метода символы, обозначаемые параметром *paddingChar,* вводятся

с левой стороны вызывающей строки, чтобы ее общая длина стала равной значению

параметра *totalWidth.* В обеих формах возвращается получающаяся в итоге

строка. Если значение параметра *totalWidth* меньше длины вызывающей строки, то

возвращается копия неизмененной вызывающей строки.

Для заполнения строки справа служат следующие формы метода PadRight().

public string PadRight(int totalWidth)

public string PadRight(int totalWidth, char paddingChar)

В приведенном ниже примере программы демонстрируются обрезка и заполнение

строк.

(***glava22\_7***)

class TrimPadDemo

{

static void Main()

{

string str = "test";

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

//full string with spaces from left

str = str.PadLeft(10);

Console.WriteLine(" | " + str + " | ");

//full string spaces right

str = str.PadRight(20);

Console.WriteLine(" | " + str + " | ");

//cut spaces

str = str.Trim();

Console.WriteLine(" | " + str + " | ");

//fill string with # symbols left

str = str.PadLeft(10, '#');

Console.WriteLine(" | " + str + " | ");

//fill string #symbols right

str = str.PadRight(20, '#');

Console.WriteLine(" | " + str + " | ");

//cut symbols #

str = str.Trim('#');

Console.WriteLine(" | " + str + " | ");

}

}

**Вставка, удаление и замена строк**

Для вставки одной строки в другую служит приведенный ниже метод Insert():

public string Insert(int startIndex, string value)

где *value* обозначает строку, вставляемую в вызывающую строку по индексу

*startIndex.* Метод возвращает получившуюся в итоге строку.

Для удаления части строки служит метод Remove(). Ниже приведены две его формы.

public string Remove(int startIndex)

public string Remove(int startIndex, int count)

В первой форме метода Remove() удаление выполняется, начиная с места, указываемого

по индексу *startIndex,* и продолжается до конца строки. А во второй форме

данного метода из строки удаляется количество символов, определяемое параметром

*count,* начиная с места, указываемого по индексу *startIndex.* В обеих формах возвращается

получающаяся в итоге строка.

Для замены части строки служит метод Replace(). Ниже приведены две его формы.

public string Replace(char oldChar, char newChar)

public string Replace(string oldValue, string newValue)

В первой форме метода Replace() все вхождения символа *oldChar* в вызывающей

строке заменяются символом *newChar. А* во второй форме данного метода все вхождения

строки *oldValue* в вызывающей строке заменяются строкой *newValue.* В обеих

формах возвращается получающаяся в итоге строка.

В приведенном ниже примере демонстрируется применение методов Insert(),

Remove() и Replace().

(***glava22\_8***)

class InsRepRevDemo

{

static void Main()

{

string str = "This is a test";

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

//input string

str = str.Insert(4, "simple");

Console.WriteLine(str);

//change string

str = str.Replace("simple", "dificult");

Console.WriteLine(str);

//change symbols in string

str = str.Replace('t', 'X');

Console.WriteLine(str);

//delete string

str = str.Remove(4, 5);

Console.WriteLine(str);

}

}

**Смена регистра**

В классе String предоставляются два удобных метода, позволяющих сменить регистр

букв в строке, — ToUpper() и ToLower(). Ниже приведены их простейшие

формы.

public string ToLower()

public string ToUpper()

Метод ToLower() делает строчными все буквы в вызывающей строке, а метод

ToUpper() делает их прописными. В обоих случаях возвращается получающаяся в

итоге строка. Имеются также следующие формы этих методов, в которых можно указывать

информацию о культурной среде и способы преобразования символов.

public string ToLower(CultureInfo culture)

public string ToUpper(CultureInfo culture)

С помощью этих форм можно избежать неоднозначности в исходном коде по отношению

к правилам смены регистра. Именно для таких целей эти формы и рекомендуется

применять.

Кроме того, имеются следующие методы T o U p p e r I n v a r i a n t ( ) и

ToLowerInvariant().

public string ToUpperlnvariant()

public string ToLowerlnvariant()

Эти методы аналогичны методам ToUpper() и ToLower(), за исключением того,

что они изменяют регистр букв в вызывающей строке безотносительно к настройкам

культурной среды.

**Применение метода Substring()**

Для получения части строки служит метод Substring(). Ниже приведены две его

формы.

public string Substring(int startIndex)

public string Substring(int startIndex, int length)

В первой форме метода Substring() подстрока извлекается, начиная с места,

обозначаемого параметром *startIndex,* и до конца вызывающей строки. А во второй

форме данного метода извлекается подстрока, состоящая из количества символов,

определяемых параметром *length,* начиная с места, обозначаемого параметром

*startIndex.* В обеих формах возвращается получающаяся в итоге подстрока.

В приведенном ниже примере программы демонстрируется применение метода

Substring().

(***glava22\_9***)

class SubstringDemo

{

static void Main()

{

string str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

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

Console.Write("Substring str.Substring(15): ");

string substr = str.Substring(15);

Console.WriteLine(substr);

Console.Write("Substring str.Substring(0, 15): ");

substr = str.Substring(0, 15);

Console.WriteLine(substr);

}

}

**Методы расширения класса String**

Как упоминалось ранее, в классе String реализуется обобщенный интерфейс

IEnumerable<T>. Это означает, что, начиная с версии C# 3.0, для объекта класса

String можно вызывать методы расширения, определенные в классах Enumerable

и Queryable, которые находятся в пространстве имен System.Linq. Эти методы расширения

служат главным образом для поддержки LINQ, хотя некоторые из них могут

использоваться в иных целях, в том числе и в определенных видах обработки строк.

Подробнее о методах расширения см. в главе 19.

**Форматирование**

Для форматирования данных числовых типов в C# предусмотрен целый ряд методов, включая методы Console.WriteLine(), String.Format() и ToString(). Во всех этих методах применяется

один и тот же подход к форматированию. Поэтому освоив один из них, вы

сможете без особого труда применять и другие.

**Общее представление о форматировании**

Форматирование осуществляется с помощью двух компонентов: *спецификаторов*

*формата* и *поставщиков формата.* Конкретная форма строкового представления

отдельного значения зависит от спецификатора формата. Следовательно, спецификатор

формата определяет, в какой именно удобочитаемой форме будут представлены

данные. Например, для вывода числового значения в экспоненциальном представлении

(т.е. в виде мантиссы и порядка числа) используется спецификатор формата Е.

Для учета культурных и языковых отличий в C# предусмотрены поставщики формата.

В частности, поставщик формата определяет порядок интерпретации спецификатора

формата. Поставщик формата создается путем реализации интерфейса

IFormatProvider, в котором определяется метод GetFormat().

Применение спецификаторов формата рассматривается далее на примере метода

Console.WriteLine(), хотя аналогичный подход применим и к другим методам,

поддерживающим форматирование.

Для форматирования выводимых данных служит следующая форма метода

WriteLine().

WriteLine("форматирующая строка", arg0, arg1, ... , argN);

В этой форме аргументы метода WriteLine() разделяются запятой, а не знаком +.

А форматирующая строка состоит из двух следующих элементов: обычных печатаемых

символов, отображаемых в исходном виде, а также команд форматирования.

Ниже приведена общая форма команд форматирования:

{argnum, width: fmt}

где *argnum* — это номер отображаемого аргумента, начиная с нуля; *width* — минимальная

ширина поля, a *fmt* — спецификатор формата. Параметры *width* и *fmt* не

являются обязательными. Поэтому в своей простейшей форме команда форматирования

просто указывает конкретные аргументы для отображения. Например, команда

{0} указывает аргумент *arg0,* команда {1} — аргумент *arg1* и т.д.

Если в команде форматирования указывается параметр *fmt,* то данные отображаются

в указываемом формате. В противном случае используется формат, выбираемый по

умолчанию. Если же в команде форматирования указывается параметр *width,* то выводимые

данные дополняются пробелами для достижения минимально необходимой

ширины поля. При положительном значении параметра *width* выводимые данные выравниваются

по правому краю, а при отрицательном значении — по левому краю.

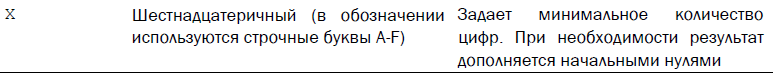
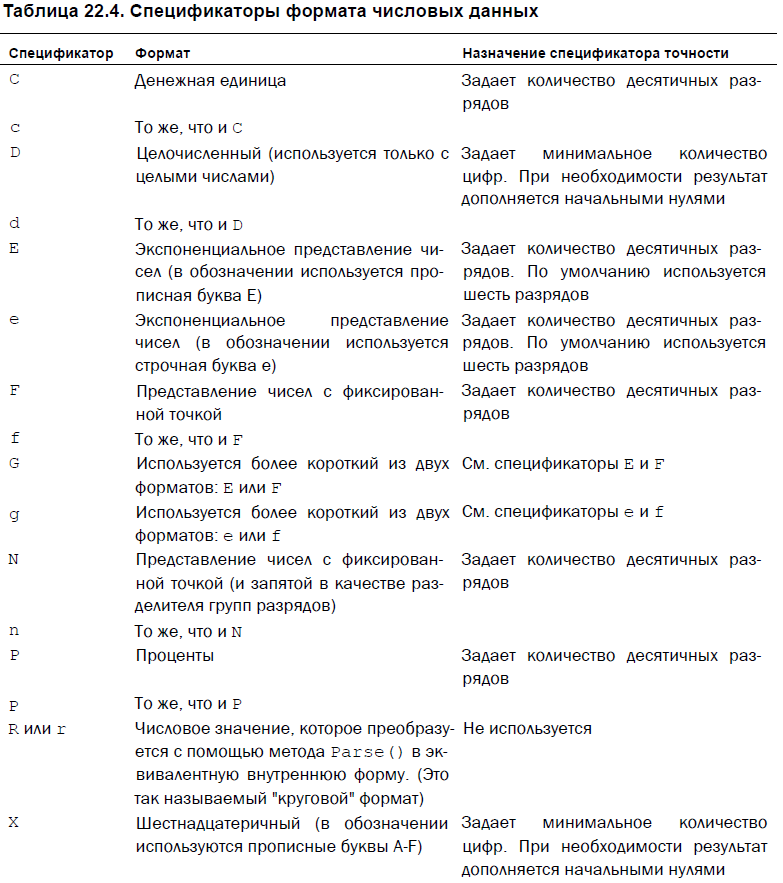
Для числовых данных определено несколько спецификаторов формата, сведенных

в табл. 22.4. Каждый спецификатор формата может включать в себя дополнительный,

но необязательный спецификатор точности. Так, если числовое значение требуется

указать в формате с фиксированной точкой и двумя десятичными разрядами в дробной

части, то для этой цели служит спецификатор F2.



Для большинства пользователей используемая по умолчанию информация о

культурной среде соответствует их региональным стандартам и языковым особенностям.

Поэтому один и тот же спецификатор формата может использоваться без учета

культурного контекста, в котором выполняется программа.

В приведенной ниже программе демонстрируется применение нескольких спецификаторов

формата числовых данных.

**(glava22\_10)**

class FormatDemo

{

static void Main()

{

double v = 17688.65849;

double v2 = 0.15;

int x = 21;

Console.WriteLine("{0:F2}", v);

Console.WriteLine("{0:N5}", v);

Console.WriteLine("{0:e}", v);

Console.WriteLine("{0:r}", v);

Console.WriteLine("{0:p}", v);

Console.WriteLine("{0:x}", v);

Console.WriteLine("{0:D12}", v);

Console.WriteLine("{0:C}", v);

}

}

**Представление о номерах аргументов**

Следует иметь в виду, что аргумент, связанный со спецификатором формата, определяется

номером аргумента, а не его позицией в списке аргументов. Это означает,

что один и тот же аргумент может указываться неоднократно в одном вызове метода

WriteLine(). Эта также означает, что аргументы могут отображаться в той последовательности,

в какой они указываются в списке аргументов. В качестве примера рассмотрим

следующую программу.

(glava22\_11)

class FormatDemo2

{

static void Main()

{

//format the one argument

Console.WriteLine("{0:F2} {0:e} {0:F3}", 10.12345);

//show not in order

Console.WriteLine("{2:d} {0:d} {1:d}", 1, 2, 3);

}

}

В первом операторе вызова метода WriteLine() один и тот же аргумент 10.12345

форматируется тремя разными способами. Это вполне допустимо, поскольку каждый

спецификатор формата в этом вызове обозначает первый и единственный аргумент.

А во втором вызове метода WriteLine() три аргумента отображаются не по порядку.

Не следует забывать, что каких-то особых правил, предписывающих обозначать аргументы

в спецификаторах формата в определенной последовательности, не существует.

Любой спецификатор формата может обозначать какой угодно аргумент.

**Применение методов String.Format()**

**и ToString() для форматирования данных**

Вообще говоря, отформатированное строковое представление отдельного значения

может быть получено двумя способами. Один из них состоит в применении

метода String.Format(), а другой — в передаче спецификатора формата методу

ToString(), относящемуся к одному из встроенных в С# числовых типов данных. Оба

способа рассматриваются далее по порядку.

**Применение метода String.Format() для форматирования значений**

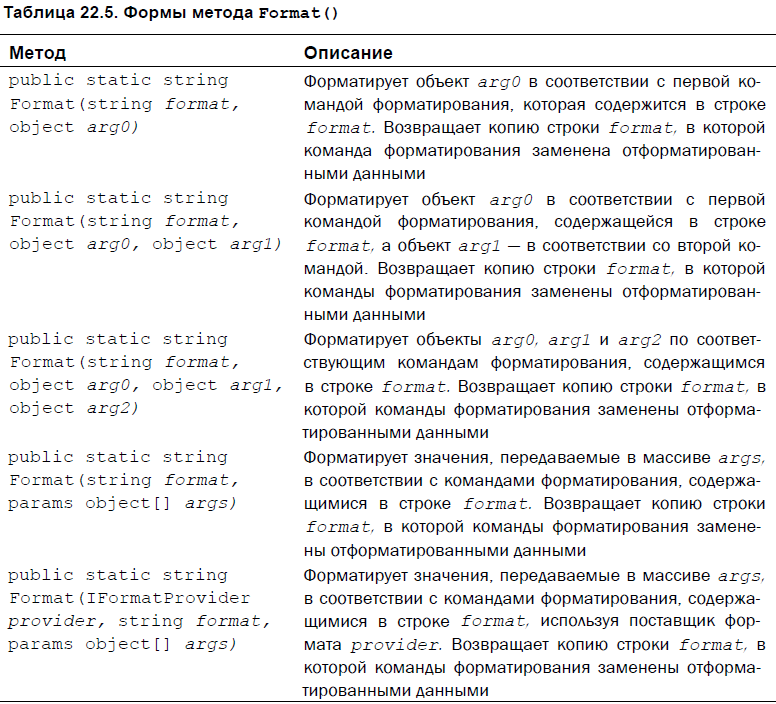
Для получения отформатированного значения достаточно вызвать метод Format(),

определенный в классе String, в соответствующей его форме. Все формы этого метода

перечислены в табл. 22.5. Метод Format() аналогичен методу WriteLine(), за исключением

того, что он возвращает отформатированную строку, а не выводит ее на

консоль.



Ниже приведен вариант предыдущего примера программы форматирования, измененный

с целью продемонстрировать применение метода String.Format(). Этот

вариант дает такой же результат, как и предыдущий.

class FormatDemo

{

static void Main()

{

double v = 17688.65849;

double v2 = 0.15;

int x = 21;

string str = String.Format("{0:F2}", v);

Console.WriteLine(str);

str = String.Format("{0:N5}", v);

Console.WriteLine(str);

str = String.Format("{0:e}", v);

Console.WriteLine(str);

str = String.Format("{0:r}", v);

Console.WriteLine(str);

str = String.Format("{0:p}", v2);

Console.WriteLine(str);

str = String.Format("{0:X}", x);

Console.WriteLine(str);

str = String.Format("{0:D12}", x);

Console.WriteLine(str);

str = String.Format("{0:C}", 189.99);

Console.WriteLine(str);

}

}

Аналогично методу WriteLine(), метод String.Format() позволяет встраивать

в свой вызов обычный текст вместе со спецификаторами формата, причем в вызове

данного метода может быть указано несколько спецификаторов формата и значений.

В качестве примера рассмотрим еще одну программу, отображающую текущую сумму

и произведение чисел от 1 до 10.

class FormatDemo2

{

static void Main()

{

int i;

int sum = 0;

int prod = 1;

string str;

/\* Отобразить текущую сумму и произведение чисел

от 1 до 10. \*/

for (i = 1; i <= 10; i++)

{

sum += i;

prod \*= i;

str = String.Format("Сумма:{0,3:D} Произведение:{1,8:D}",

sum, prod);

Console.WriteLine(str);

}

}

}

str = String.Format("Сумма:{0,3:D} Произведение:{1,8:D}",

sum, prod);

В этом операторе содержится вызов метода Format() с двумя спецификаторами

формата: одним — для суммы (в переменной sum), а другим — для произведения

(в переменной prod). Обратите также внимание на то, что номера аргументов указываются

таким же образом, как и в вызове метода WriteLine(), и что в вызов метода

Format() включается обычный текст, как, например, строка "Сумма: ". Этот текст

передается данному методу и становится частью выводимой строки.

**Применение метода ToString() для форматирования данных**

Для получения отформатированного строкового представления отдельного значения

любого числового типа, которому соответствует встроенная структура, например

Int32 или Double, можно воспользоваться методом ToString(). Этой цели служит

приведенная ниже форма метода ToString().

public string ToString("форматирующая строка")

В этой форме метод ToString() возвращает строковое представление вызывающего

объекта в том формате, который определяет спецификатор "форматирующая

строка", передаваемый данному методу. Например, в следующей строке кода формируется

строковое представление значения 188.99 в формате денежной единицы с

помощью спецификатора формата С.

string str = 189.99.ToString("С");

Обратите внимание на то, что спецификатор формата передается методу

ToString() непосредственно.

Ниже приведен вариант примера предыдущей программы форматирования, измененный

с целью продемонстрировать применение метода ToString() для получения

отформатированных строк. Этот вариант дает такой же результат, как и предыдущий.

(glava22\_12)

class ToStringDemo

{

static void Main()

{

double v = 17688.65849;

double v2 = 0.15;

int x = 21;

string str = v.ToString("F2");

Console.WriteLine(str);

str = v.ToString("N5");

Console.WriteLine(str);

str = v.ToString("e");

Console.WriteLine(str);

str = v.ToString("r");

Console.WriteLine(str);

str = v2.ToString("p");

Console.WriteLine(str);

str = x.ToString("X");

Console.WriteLine(str);

str = x.ToString("D12");

Console.WriteLine(str);

str = 189.99.ToString("C");

Console.WriteLine(str);

}

}

**Определение пользовательского формата числовых данных**

Несмотря на всю полезность предопределенных спецификаторов формата числовых

данных, в C# предоставляется также возможность определить пользовательский,

т.е. свой собственный, формат, используя средство, называемое *форматом изображения.*

Своим происхождением термин *формат изображения* обязан тому обстоятельству,

что специальный формат пользователь определяет, задавая пример внешнего вида

(т.е. изображение) выводимых данных. Такой подход вкратце упоминался в части I

этой книги, а здесь он рассматривается более подробно.

**Символы-заполнители специального формата числовых данных**

Когда пользователь определяет специальный формат, он задает этот формат в виде

примера (иди изображения) того, как должны выглядеть выводимые данные. Для этой

цеди используются символы, перечисленные в табл. 22.6. Они служат в качестве заполнителей

и рассматриваются далее по очереди.

Символ точки обозначает местоположение десятичной точки.

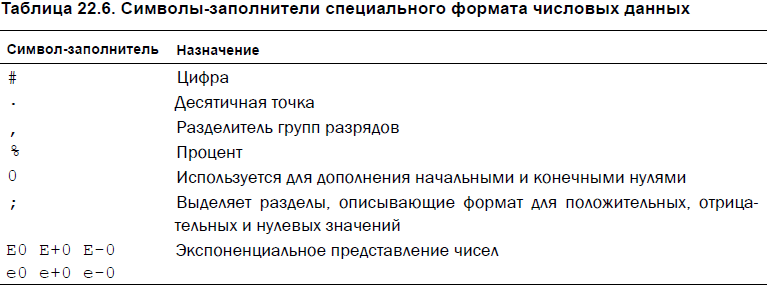
Символ-заполнитель # обозначает цифровую позицию, иди разряд числа. Этот

символ может указываться слева иди справа от десятичной точки либо отдельно.

Так, если справа от десятичной точки указывается несколько символов #, то они обозначают

количество отображаемых десятичных цифр в дробной части числа.

Когда же символы # указываются слева от десятичной точки, то они обозначают количество отображаемых десятичных цифр в целой части числа.



Символ-заполнитель 0 обусловливает дополнение форматируемого числового значения

начальными или конечными нулями, чтобы обеспечить минимально необходимое

количество цифр в строковом представлении данного значения. Этот символ

может указываться как слева, как и справа от десятичной точки. Например, следующая

строка кода:

Console.WriteLine("{0:00##.#00}", 21.3);

выводит такой результат.

0021.300

Значения, состоящие из большего количества цифр, будут полностью отображаться

слева от десятичной точки, а округленные — справа.

При отображении больших числовых значений отдельные группы цифр могут отделяться

друг от друга запятыми, для чего достаточно вставить запятую в шаблон, состоящий

из символов #. Например, следующая строка кода:

Console.WriteLine("{0:#,###.#}", 3421.3);

выводит такой результат..

3,421.3.

Указывать запятую на каждой позиции совсем не обязательно. Если указать запятую

в шаблоне один раз, то она будет автоматически вставляться в форматируемом

числовом значении через каждые три цифры слева от десятичной запятой. Например,

следующая строка кода:

Console.WriteLine("{0:#,###.#}", 8763421.3);

дает такой результат.

8,763,421.3.

У запятой имеется и другое назначение. Если запятая вставляется непосредственно

перед десятичной точкой, то она выполняет роль масштабного коэффициента. Каждая

запятая делит форматируемое числовое значение на 1000. Например, следующая

строка кода:

Console.WriteLine("Значение в тысячах: {0:#,###,.#}", 8763421.3);

дает такой результат.

Значение в тысячах: 8,763.4

Как показывает приведенный выше результат, числовое значение выводится масштабированным

в тысячах.

При необходимости в форматируемой строке можно также указывать такие управляющие

последовательности, как \t или \n.

Знак ; служит разделителем в различных форматах вывода положительных, отрицательных

и нулевых значений. Ниже приведена общая форма пользовательского

спецификатора формата, в котором используется знак ;.

***положительный\_формат; отрицательный\_формат; нулевой\_формат***

Console.WriteLine("{0:#.##; (#.##);0.00}", num);

Если значение переменной num положительно, то оно отображается с двумя разрядами

после десятичной точки. Если же значение переменной num отрицательно,

то оно также отображается с двумя разрядами после десятичной точки, но в круглых

скобках. А если значение переменной num равно нулю, то оно отображается в виде

строки 0.00.

В приведенном ниже примере программы демонстрируется лишь несколько специальных

форматов, которые могут быть определены пользователем.

(***glava22\_13***)

class PictureFormatDemo

{

static void Main()

{

double num = 64354.2345;

Console.WriteLine("Default format: " + num);

Console.WriteLine("Value with 2 decade levels: " + "(0:#.##)", num);

Console.WriteLine("Add columns: {0:#,###.##}", num);

}

}

**Форматирование перечислений**

В C# допускается также форматировать значения, определяемые в перечислении.

Вообще говоря, значения из перечисления могут отображаться как по имени, так и

по значению. Спецификаторы формата перечислений сведены в табл. 22.10. Обратите

особое внимание на форматы G и F. Перед перечислениями, которые должны представлять

битовые поля, следует указывать атрибут Flags. Как правило, в битовых полях

хранятся значения, обозначающие отдельные двоичные разряды и упорядоченные

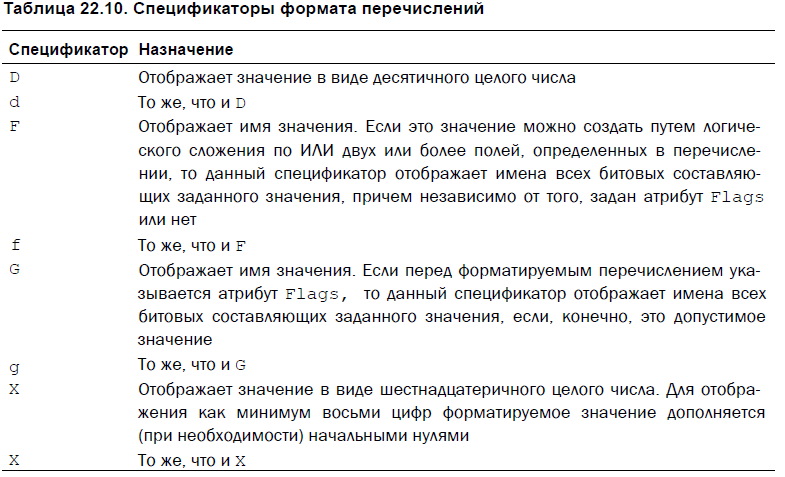
по степени числа 2. При наличии атрибута Flags имена всех битовых составляющих

форматируемого значения, если, конечно, это действительное значение, отображаются

с помощью спецификатора G. А с помощью спецификатора F отображаются имена

всех битовых составляющих форматируемого значения, если оно составляется путем

логического сложения по ИЛИ двух иди более полей, определяемых в перечислении.



class EnumFmtDemo

{

enum Direction { North, South, East, West }

[Flags]

enum Status

{

Ready = 0x1, OffLine = 0x2,

Waiting = 0x4, TransmitOK = 0x8,

ReceiveOK = 0x10, OnLine = 0x20

}

static void Main()

{

Direction d = Direction.West;

Console.WriteLine("{0:G}", d);

Console.WriteLine("{0:F}", d);

Console.WriteLine("{0:D}", d);

Console.WriteLine("{0:X}", d);

Status s = Status.Ready | Status.TransmitOK;

Console.WriteLine("{0:G}", s);

Console.WriteLine("{0:F}", s);

Console.WriteLine("{0:D}", s);

Console.WriteLine("{0:X}", s);

}

}

Ниже приведен результат выполнения этой программы.

West

West

3

00000003

Ready, TransmitOK

Ready, TransmitOK

9

00000009

До главы 22===========================================================================

С точки зрения регулярного программирования строковый тип данных ***string*** относится

к числу самых важных в С#. Этот тип определяет и поддерживает символьные

строки. В целом ряде других языков программирования строка представляет собой

массив символов(C++). А в C# строки являются объектами. Следовательно, тип ***string*** относится

к числу ссылочных.

**Построение строк**

Самый простой способ построить символьную строку — воспользоваться строковым

литералом.

string str = "Strings in C# pretty effective.";

Объект типа string можно также создать из массива типа char. Например:

char [] charray = { 't', 'e', 's', 't' };

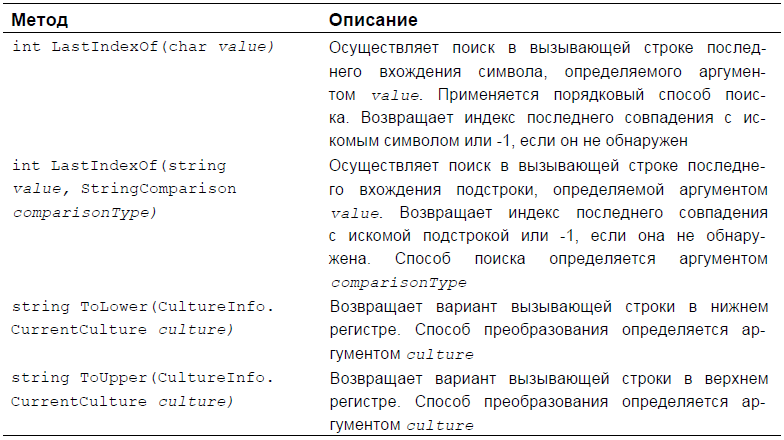
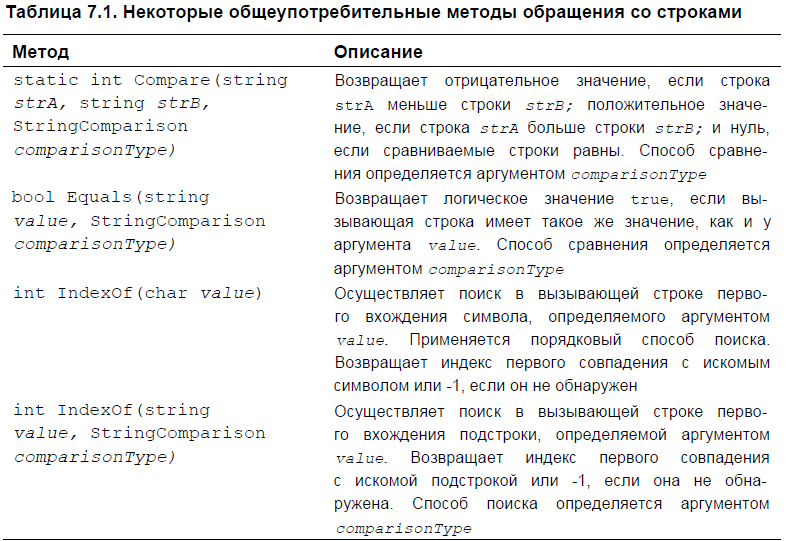
string str = new string(charray);

Как только объект типа ***string*** будет создан, его можно использовать везде, где

только требуется строка текста, заключенного в кавычки.

Класс типа ***string*** содержит ряд методов для обращения со строками. Некоторые

из этих методов перечислены в табл. 7.1.



Обратите внимание на то, что некоторые методы принимают параметр типа ***StringComparison***. Это перечислимый тип, определяющий различные значения, которые определяют порядок сравнения символьных строк. Нетрудно догадаться, что символьные строки можно сравнивать разными способами. Например, их можно сравнивать на основании двоичных значений символов, из которых они состоят. Такое сравнение называется *порядковым.* Строки можно также

сравнивать с учетом различных особенностей культурной среды, например, в лексикографическом порядке. Это так называемое сравнение *с учетом культурной среды.*

(Учитывать культурную среду особенно важно в локализуемых приложениях.) Кроме

того, строки можно сравнивать *с учетом* или *без учета* регистра. Несмотря на то что

существуют перегружаемые варианты методов Compare(), Equals(), IndexOf()

и LastIndexOf(), обеспечивающие используемый по умолчанию подход к сравнению

символьных строк, в настоящее время считается более приемлемым явно указывать

способ требуемого сравнения, чтобы избежать неоднозначности, а также упростить

локализацию приложений. Именно поэтому здесь рассматривают разные способы

сравнения символьных строк.

Как правило и за рядом исключений, для сравнения символьных строк с учетом

культурной среды (т.е. языковых и региональных стандартов) применяется

способ ***StringComparison.CurrentCulture***. Если же требуется сравнить строки

только на основании значений их символов, то лучше воспользоваться способом

***StringComparison.Ordinal***, а для сравнения строк без учета регистра — одним

из двух способов: ***StringComparison.CurrentCultureIgnoreCase*** или

***StringComparison.OrdinalIgnoreCase***. Кроме того, можно указать сравнение

строк без учета культурной среды (подробнее об этом — в главе 22).

Обратите внимание на то, что метод Compare() объявляется в табл. 7.1 как static.

Подробнее о модификаторе static речь пойдет в главе 8, а до тех пор вкратце поясним,

что он обозначает следующее: метод Compare() вызывается по имени своего

класса, а не по его экземпляру. Следовательно, для вызова метода Compare() служит

следующая общая форма:

**результат = string.Compare(str1, str2, способ);**

где *способ* обозначает конкретный подход к сравнению символьных строк.

Обратите также внимание на методы ***ToUpper()*** и ***ToLower()***, преобразующие содержимое

строки в символы верхнего и нижнего регистра соответственно. Их формы,

представленные в табл. 7.1, содержат параметр ***CultureInfо***, относящийся к классу,

в котором описываются атрибуты культурной среды, применяемые для сравнения.

В примерах, приведенных в этой книге, используются текущие настройки культурной

среды (т.е. текущие языковые и региональные стандарты). Эти настройки указываются

при передаче методу аргумента ***CultureInfo.CurrentCulture.*** Класс ***CultureInfo***

относится к пространству имен ***System.Globalization***. Любопытно, имеются варианты

рассматриваемых здесь методов, в которых текущая культурная среда используется

по умолчанию, но во избежание неоднозначности в примерах из этой книги

аргумент ***CultureInfo.CurrentCulture*** указывается явно.

Объекты типа ***string*** содержат также свойство ***Length***, где хранится длина строки.

Отдельный символ выбирается из строки с помощью индекса, как в приведенном

ниже фрагменте кода.

***string str = "тест";***

***Console.WriteLine(str [0]);***

Следует, однако, иметь в виду, что с помощью индекса нельзя присвоить новое значение символу в строке. Индекс может служить только для выборки символа из строки.

Для проверки двух строк на равенство служит оператор ==. Как правило, если оператор

== применяется к ссылкам на объект, то он определяет, являются ли они ссылками

на один и тот же объект. Совсем иначе обстоит дело с объектами типа string.

Когда оператор == применяется к ссылкам на две строки, он сравнивает содержимое

этих строк. Это же относится и к оператору !=. В обоих случаях выполняется порядковое

сравнение. Для проверки двух строк на равенство с учетом культурной среды

служит метод Equals(), где непременно нужно указать способ сравнения в виде аргумента

StringComparison.CurrentCulture. Следует также иметь в виду, что метод

Compare() служит для сравнения строк с целью определить отношение порядка,

например для сортировки. Если же требуется проверить символьные строки на равенство,

то для этой цели лучше воспользоваться методом Equals() или строковыми

операторами. (***glava7\_11.1***)

string str1 = "Hello there";

string str2 = "Hello there";

string str3 = "Im here where right there";

string strUp, strLow;

int result, idx;

strLow = str1.ToLower(CultureInfo.CurrentCulture);

strUp = str1.ToUpper(CultureInfo.CurrentCulture);

result = string.Compare(str3, str1, StringComparison.CurrentCulture);

//search for symbol

idx = str2.IndexOf("One", StringComparison.Ordinal);

idx = str2.LastIndexOf("One", StringComparison.Ordinal);

Прежде чем читать дальше, обратите внимание на то, что метод Compare() вызывается

следующим образом.

***result = string.Compare(str1, str3, StringComparison.CurrentCulture);***

Как пояснялось ранее, метод Compare() объявляется как static, и поэтому он вызывается

по имени, а не по экземпляру своего класса.

С помощью ***оператора +*** можно сцепить (т.е. объединить вместе) две строки.

Например, в следующем фрагменте кода:

***string str1 = "Один";***

***string str2 = "Два";***

***string str3 = "Три";***

***string str4 = str1 + str2 + str3;***

переменная str4 инициализируется строкой "ОдинДваТри".

**Массивы строк**

Аналогично данным любого другого типа, строки могут быть организованы в массивы. (***glava7\_12***)

class StringArrays

{

static void Main()

{

string [] str = { "This", "is", "simple", "test" };

Console.WriteLine("Array str: ");

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

Console.Write(str[i] + " ");

Console.WriteLine();

//change str

str [1] = "as well";

str [3] = "till the end!";

Console.WriteLine("Changed Array str: ");

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

Console.Write(str[i] + " ");

Console.WriteLine();

}

}

**Постоянство строк**

Как ни странно, содержимое объекта типа string не подлежит изменению. Это

означает, что однажды созданную последовательность символов изменить нельзя. Но

данное ограничение способствует более эффективной реализации символьных строк.

Поэтому этот, на первый взгляд, очевидный недостаток на самом деле превращается в

преимущество. Так, если требуется строка в качестве разновидности уже имеющейся

строки, то для этой цели следует создать новую строку, содержащую все необходимые

изменения. А поскольку неиспользуемые строковые объекты автоматически собираются

в "мусор", то о дальнейшей судьбе ненужных строк можно даже не беспокоиться.

Следует, однако, подчеркнуть, что переменные ссылки на строки (т.е. объекты типа

string) подлежат изменению, а следовательно, они могут ссылаться на другой объект.

Но содержимое самого объекта типа string не меняется после его создания.

Для того чтобы стало понятнее, почему неизменяемые строки не являются помехой,

воспользуемся еще одним методом обращения со строками: ***Substring ().*** Этот метод

возвращает новую строку, содержащую часть вызывающей строки. В итоге создается

новый строковый объект, содержащий выбранную подстроку, тогда как исходная

строка не меняется, а следовательно, соблюдается принцип постоянства строк. Ниже

приведена рассматриваемая здесь форма метода ***Substring ():***

***string Substring (int индекс\_начала, int длина)***

где ***индекс\_начала***обозначает начальный индекс исходной строки, а ***длина***— длину

выбираемой подстроки. (***glava7\_13***)

string orgstr = "C# make using simple.";

//make sub str

string substr = orgstr.Substring(5, 10);

//string substr = orgstr.Substring(0, orgstr.Length);

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

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

Как видите, исходная строка из переменной orgstr не меняется, а выбранная из

нее подстрока содержится в переменной substr.

И последнее замечание: несмотря на то, что постоянство строк обычно не является

ни ограничением, ни помехой для программирования на С#, иногда оказывается

полезно иметь возможность видоизменять строки. Для этой цели в C# имеется класс

StringBuilder, который определен в пространстве имен System.Text. Этот класс

позволяет создавать строковые объекты, которые можно изменять. Но, как правило,

в программировании на C# используется тип string, а не класс StringBuilder.

**Применение строк в операторах switch**

Объекты типа string могут использоваться для управления оператором switch.

Это единственный нецелочисленный тип данных, который допускается применять

в операторе switch. Благодаря такому применению строк в некоторых сложных ситуациях

удается найти более простой выход из положения, чем может показаться на

первый взгляд. Например, в приведенной ниже программе выводятся отдельные цифры,

соответствующие словам "один", "два" и "три". (***glava7\_14***)

string [] strs = { "one", "two", "three", "two", "one" };

foreach (string s in strs)

{

switch(s)

{

case "one": Console.Write(1); break;

case "two": Console.Write(2); break;

case "three": Console.Write(3); break;

}

}