Contents

Strings (C# Programming Guide)

12/11/2018 • 12 minutes to read • Edit Online

A string is an object of type String whose value is text. Internally, the text is stored as a sequential read-only collection of Char objects. There is no null-terminating character at the end of a C# string; therefore a C# string can contain any number of embedded null characters ('\0'). The Length property of a string represents the number of `Char` objects it contains, not the number of Unicode characters. To access the individual Unicode code points in a string, use the StringInfo object.

string vs. System.String

In C#, the `string` keyword is an alias for String. Therefore, `String` and `string` are equivalent, and you can use whichever naming convention you prefer. The `String` class provides many methods for safely creating, manipulating, and comparing strings. In addition, the C# language overloads some operators to simplify common string operations. For more information about the keyword, see string. For more information about the type and its methods, see String.

Declaring and Initializing Strings

You can declare and initialize strings in various ways, as shown in the following example:

```
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

//Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Note that you do not use the new operator to create a string object except when initializing the string with an array of chars.

Initialize a string with the Empty constant value to create a new String object whose string is of zero length. The string literal representation of a zero-length string is "". By initializing strings with the Empty value instead of null, you can reduce the chances of a NullReferenceException occurring. Use the static IsNullOrEmpty(String) method to verify the value of a string before you try to access it.

Immutability of String Objects

String objects are *immutable*: they cannot be changed after they have been created. All of the String methods and C# operators that appear to modify a string actually return the results in a new string object. In the following example, when the contents of `s1` and `s2` are concatenated to form a single string, the two original strings are unmodified. The `+=` operator creates a new string that contains the combined contents. That new object is assigned to the variable `s1`, and the original object that was assigned to `s1` is released for garbage collection because no other variable holds a reference to it.

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Because a string "modification" is actually a new string creation, you must use caution when you create references to strings. If you create a reference to a string, and then "modify" the original string, the reference will continue to point to the original object instead of the new object that was created when the string was modified. The following code illustrates this behavior:

```
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
//Output: Hello
```

For more information about how to create new strings that are based on modifications such as search and replace operations on the original string, see How to: Modify String Contents.

Regular and Verbatim String Literals

Use regular string literals when you must embed escape characters provided by C#, as shown in the following example:

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
  Row 1
  Row 2
  Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

Use verbatim strings for convenience and better readability when the string text contains backslash characters, for example in file paths. Because verbatim strings preserve new line characters as part of the string text, they can be used to initialize multiline strings. Use double quotation marks to embed a quotation mark inside a verbatim string. The following example shows some common uses for verbatim strings:

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
   Thus on mine arm, most soothing sweet it is
   To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."
```

String Escape Sequences

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
\U	Unicode escape sequence for surrogate pairs.	\Unnnnnnnn
\u	Unicode escape sequence	\u0041 = "A"
\v	Vertical tab	0x000B
\x	Unicode escape sequence similar to "\u" except with variable length.	\x0041 or \x41 = "A"

> **NOTE**
> At compile time, verbatim strings are converted to ordinary strings with all the same escape sequences. Therefore, if you view a verbatim string in the debugger watch window, you will see the escape characters that were added by the compiler, not the verbatim version from your source code. For example, the verbatim string @"C:\files.txt" will appear in the watch window as "C:\\files.txt".

Format Strings

A format string is a string whose contents are determined dynamically at runtime. Format strings are created by embedding *interpolated expressions* or placeholders inside of braces within a string. Everything inside the braces (`{...}`) will be resolved to a value and output as a formatted string at runtime. There are two methods to create format strings: string interpolation and composite formatting.

String Interpolation

Available in C# 6.0 and later, *interpolated strings* are identified by the `$` special character and include interpolated expressions in braces. If you are new to string interpolation, see the String interpolation - C# interactive tutorial for a quick overview.

Use string interpolation to improve the readability and maintainability of your code. String interpolation achieves the same results as the `String.Format` method, but improves ease of use and inline clarity.

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of {jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) * 100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

Composite Formatting

The String.Format utilizes placeholders in braces to create a format string. This example results in similar output to the string interpolation method used above.

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.", pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.", pw.published, pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

For more information on formatting .NET types see Formatting Types in .NET.

Substrings

A substring is any sequence of characters that is contained in a string. Use the Substring method to create a new string from a part of the original string. You can search for one or more occurrences of a substring by using the IndexOf method. Use the Replace method to replace all occurrences of a specified substring with a new string. Like the Substring method, Replace actually returns a new string and does not modify the original string. For more information, see How to: search strings and How to: Modify String Contents.

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Accessing Individual Characters

You can use array notation with an index value to acquire read-only access to individual characters, as in the following example:

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

If the String methods do not provide the functionality that you must have to modify individual characters in a string, you can use a StringBuilder object to modify the individual chars "in-place", and then create a new string to store the results by using the StringBuilder methods. In the following example, assume that you must modify the original string in a particular way and then store the results for future use:

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS lOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Null Strings and Empty Strings

An empty string is an instance of a System.String object that contains zero characters. Empty strings are used often in various programming scenarios to represent a blank text field. You can call methods on empty strings because they are valid System.String objects. Empty strings are initialized as follows:

```
string s = String.Empty;
```

By contrast, a null string does not refer to an instance of a System.String object and any attempt to call a method on a null string causes a NullReferenceException. However, you can use null strings in concatenation and comparison operations with other strings. The following examples illustrate some cases in which a reference to a null string does and does not cause an exception to be thrown:

```
static void Main()
{
    string str = "hello";
    string nullStr = null;
    string emptyStr = String.Empty;

    string tempStr = str + nullStr;
    // Output of the following line: hello
    Console.WriteLine(tempStr);

    bool b = (emptyStr == nullStr);
    // Output of the following line: False
    Console.WriteLine(b);

    // The following line creates a new empty string.
    string newStr = emptyStr + nullStr;

    // Null strings and empty strings behave differently. The following
    // two lines display 0.
    Console.WriteLine(emptyStr.Length);
    Console.WriteLine(newStr.Length);
    // The following line raises a NullReferenceException.
    //Console.WriteLine(nullStr.Length);

    // The null character can be displayed and counted, like other chars.
    string s1 = "\x0" + "abc";
    string s2 = "abc" + "\x0";
    // Output of the following line: * abc*
    Console.WriteLine("*" + s1 + "*");
    // Output of the following line: *abc *
    Console.WriteLine("*" + s2 + "*");
    // Output of the following line: 4
    Console.WriteLine(s2.Length);
}
```

Using StringBuilder for Fast String Creation

String operations in .NET are highly optimized and in most cases do not significantly impact performance. However, in some scenarios such as tight loops that are executing many hundreds or thousands of times, string operations can affect performance. The StringBuilder class creates a string buffer that offers better performance if your program performs many string manipulations. The StringBuilder string also enables you to reassign individual characters, something the built-in string data type does not support. This code, for example, changes the content of a string without creating a new string:

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//Outputs Cat: the ideal pet
```

In this example, a StringBuilder object is used to create a string from a set of numeric types:

```
class TestStringBuilder
{
    static void Main()
    {
        System.Text.StringBuilder sb = new System.Text.StringBuilder();

        // Create a string composed of numbers 0 - 9
        for (int i = 0; i < 10; i++)
        {
            sb.Append(i.ToString());
        }
        System.Console.WriteLine(sb);  // displays 0123456789

        // Copy one character of the string (not possible with a System.String)
        sb[0] = sb[9];

        System.Console.WriteLine(sb);  // displays 9123456789
    }
}
```

Strings, Extension Methods and LINQ

Because the String type implements IEnumerable<T>, you can use the extension methods defined in the Enumerable class on strings. To avoid visual clutter, these methods are excluded from IntelliSense for the String type, but they are available nevertheless. You can also use LINQ query expressions on strings. For more information, see LINQ and Strings.

Related Topics

TOPIC	DESCRIPTION
How to: Modify String Contents	Illustrates techniques to transform strings and modify the contents of strings.
How to: Compare Strings	Shows how to perform ordinal and culture specific comparisons of strings.
How to: Concatenate Multiple Strings	Demonstrates various ways to join multiple strings into one.
How to: Parse Strings Using String.Split	Contains code examples that illustrate how to use the `String.Split` method to parse strings.
How to: Search Strings	Explains how to use search for specific text or patterns in strings.
How to: Determine Whether a String Represents a Numeric Value	Shows how to safely parse a string to see whether it has a valid numeric value.
String interpolation	Describes the string interpolation feature that provides a convenient syntax to format strings.
Basic String Operations	Provides links to topics that use System.String and System.Text.StringBuilder methods to perform basic string operations.
Parsing Strings	Describes how to convert string representations of .NET base types to instances of the corresponding types.

TOPIC	DESCRIPTION
Parsing Date and Time Strings in .NET	Shows how to convert a string such as "01/24/2008" to a System.DateTime object.
Comparing Strings	Includes information about how to compare strings and provides examples in C# and Visual Basic.
Using the StringBuilder Class	Describes how to create and modify dynamic string objects by using the StringBuilder class.
LINQ and Strings	Provides information about how to perform various string operations by using LINQ queries.
C# Programming Guide	Provides links to topics that explain programming constructs in C#.

How to: Determine Whether a String Represents a Numeric Value (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

To determine whether a string is a valid representation of a specified numeric type, use the static `TryParse` method that is implemented by all primitive numeric types and also by types such as DateTime and IPAddress. The following example shows how to determine whether "108" is a valid int.

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

If the string contains nonnumeric characters or the numeric value is too large or too small for the particular type you have specified, `TryParse` returns false and sets the out parameter to zero. Otherwise, it returns true and sets the out parameter to the numeric value of the string.

> **NOTE**
> A string may contain only numeric characters and still not be valid for the type whose `TryParse` method that you use. For example, "256" is not a valid value for `byte` but it is valid for `int`. "98.6" is not a valid value for `int` but it is a valid `decimal`.

Example

The following examples show how to use `TryParse` with string representations of `long`, `byte`, and `decimal` values.

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
if (canConvert == true)
  Console.WriteLine("number1 now = {0}", number1);
else
  Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
canConvert = byte.TryParse(numString, out number2);
if (canConvert == true)
  Console.WriteLine("number2 now = {0}", number2);
else
  Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; //"27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
  Console.WriteLine("number3 now = {0}", number3);
else
  Console.WriteLine("number3 is not a valid decimal");
```

Robust Programming

Primitive numeric types also implement the `Parse` static method, which throws an exception if the string is not a valid number. `TryParse` is generally more efficient because it just returns false if the number is not valid.

.NET Framework Security

Always use the `TryParse` or `Parse` methods to validate user input from controls such as text boxes and combo boxes.

See also

- How to: Convert a byte Array to an int
- How to: Convert a String to a Number
- How to: Convert Between Hexadecimal Strings and Numeric Types
- Parsing Numeric Strings
- Formatting Types