Strings and string literals

Article • 05/27/2023

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's 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. It's recommended to use the provided alias `string` as it works even without `using System;`. 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:

C#

```csharp
// 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);
```

You don't 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 strings

String objects are *immutable*: they can't be changed after they've 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.

C#

```
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:

C#

```
string str1 = "Hello ";
string str2 = str1;
str1 += "World";

System.Console.WriteLine(str2);
//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.

Quoted string literals

Quoted string literals start and end with a single double quote character (`"`) on the same line. Quoted string literals are best suited for strings that fit on a single line and don't include any escape sequences. A quoted string literal must embed escape characters, as shown in the following example:

C#

```
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
```

Verbatim string literals

Verbatim string literals are more convenient for multi-line strings, strings that contain backslash characters, or embedded double quotes. Verbatim strings preserve new line characters as part of the string text. Use double quotation marks to embed a quotation mark inside a verbatim string. The following example shows some common uses for verbatim strings:

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

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."
```

Raw string literals

Beginning with C# 11, you can use *raw string literals* to more easily create strings that are multi-line, or use any characters requiring escape sequences. *Raw string literals* remove the need to ever use escape sequences. You can write the string, including whitespace formatting, how you want it to appear in output. A *raw string literal*:

- Starts and ends with a sequence of at least three double quote characters (`"""`). You're allowed more than three consecutive characters to start and end the sequence in order to support string literals that contain three (or more) repeated quote characters.
- Single line raw string literals require the opening and closing quote characters on the same line.
- Multi-line raw string literals require both opening and closing quote characters on their own line.
- In multi-line raw string literals, any whitespace to the left of the closing quotes is removed from all lines of the raw string literal.

- In multi-line raw string literals, whitespace following the opening quote on the same line is ignored.
- In multi-line raw string literals, whitespace only lines following the opening quote are included in the string literal.

The following examples demonstrate these rules:

C#

```
string singleLine = """Friends say "hello" as they pass by.""";
string multiLine = """
    "Hello World!" is typically the first program someone writes.
    """;
string embeddedXML = """
       <element attr = "content">
           <body style="normal">
               Here is the main text
           </body>
           <footer>
               Excerpts from "An amazing story"
           </footer>
       </element >
       """;
// The line "<element attr = "content">" starts in the first column.
// All whitespace left of that column is removed from the string.

string rawStringLiteralDelimiter = """"
    Raw string literals are delimited
    by a string of at least three double quotes,
    like this: """
    """";
```

The following examples demonstrate the compiler errors reported based on these rules:

C#

```
// CS8997: Unterminated raw string literal.
var multiLineStart = """This
    is the beginning of a string
    """;

// CS9000: Raw string literal delimiter must be on its own line.
var multiLineEnd = """
    This is the beginning of a string """;

// CS8999: Line does not start with the same whitespace as the clos-
ing line
// of the raw string literal
var noOutdenting = """
    A line of text.
```

```
Trying to outdent the second line.
    """;
```

The first two examples are invalid because multiline raw string literals require the opening and closing quote sequence on its own line. The third example is invalid because the text is outdented from the closing quote sequence.

You should consider raw string literals when you're generating text that includes characters that require escape sequences when using quoted string literals or verbatim string literals. Raw string literals will be easier for you and others to read because it will more closely resemble the output text. For example, consider the following code that includes a string of formatted JSON:

```C#
string jsonString = """
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
    "2019-08-02T00:00:00-07:00"
  ],
  "TemperatureRanges": {
    "Cold": {
      "High": 20,
      "Low": -10
    },
    "Hot": {
      "High": 60,
      "Low": 20
    }
            },
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
}
""";
```

Compare that text with the equivalent text in our sample on JSON serialization, which doesn't make use of this new feature.

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
\v	Vertical tab	0x000B
\u	Unicode escape sequence (UTF-16)	`\uHHHH` (range: 0000 - FFFF; example: `\u00E7` = "ç")
\U	Unicode escape sequence (UTF-32)	`\U00HHHHHH` (range: 000000 - 10FFFF; example: `\U0001F47D` = "👽")
\x	Unicode escape sequence similar to "\u" except with variable length	`\xH[H][H][H]` (range: 0 - FFFF; example: `\x00E7` or `\x0E7` or `\xE7` = "ç")

⚠ Warning

When using the `\x` escape sequence and specifying less than 4 hex digits, if the characters that immediately follow the escape sequence are valid hex digits (i.e. 0-9, A-F, and a-f), they will be interpreted as being part of the escape sequence. For example, `\xA1` produces "¡", which is code point U+00A1. However, if the next character is "A" or "a", then the escape sequence will instead be interpreted as being `\xA1A` and produce "ਚ", which is code point U+0A1A. In such cases, specifying all 4 hex digits (for example, `\x00A1`) prevents any possible misinterpretation.

ⓘ Note

At compile time, verbatim and raw strings are converted to ordinary strings with all the same escape sequences. Therefore, if you view a verbatim or raw string in the debugger watch window, you will see the escape characters that were added by the compiler, not the verbatim or raw 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 run time. 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 run time. There are two methods to create format strings: string interpolation and composite formatting.

String interpolation

Interpolated strings are identified by the `$` special character and include interpolated expressions in braces. If you're 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.

C#

```csharp
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, pub-
lished: 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.
```

Beginning with C# 10, you can use string interpolation to initialize a constant string when all the expressions used for placeholders are also constant strings.

Beginning with C# 11, you can combine *raw string literals* with string interpolations. You start and end the format string with three or more successive double quotes. If your output string should contain the `{` or `}` character, you can use extra `$` characters to specify how many `{` and `}` characters start and end an interpolation. Any sequence of fewer `{` or `}` characters is included in the output. The following example shows how you can use that feature to display the distance of a point from the origin, and place the point inside braces:

```C#
int X = 2;
int Y = 3;

var pointMessage = $$"""The point {{{X}}, {{Y}}} is {{Math.Sqrt(X * X
+ Y * Y)}} from the origin.""";

Console.WriteLine(pointMessage);
// Output:
// The point {2, 3} is 3.605551275463989 from the origin.
```

Verbatim string interpolation

C# also allows verbatim string interpolation, for example across multiple lines, using the `$@` or `@$` syntax.

To interpret escape sequences literally, use a verbatim string literal. An interpolated verbatim string starts with the `$` character followed by the `@` character. You can use the `$` and `@` tokens in any order: both `$@"..."` and `@$"..."` are valid interpolated verbatim strings.

```C#
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, pub-
lished: 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}.");

// Output:
// Jupiter Hammon
//     was an African American poet born in 1711.
// He was first published in 1761
// at the age of 50.
```

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.

C#

```csharp
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 doesn't modify the original string. For more information, see How to search strings and How to modify string contents.

C#

```csharp
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:

```C#
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 don't 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:

```C#
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're valid System.String objects. Empty strings are initialized as follows:

C#

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

By contrast, a null string doesn't 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 doesn't cause an exception to be thrown:

C#

```
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 don't 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 doesn't support. This code, for example, changes the content of a string without creating a new string:

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

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

```C#
var sb = new StringBuilder();

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

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

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're available nevertheless. You can also use LINQ query expressions on strings. For more information, see LINQ and Strings.

Related articles

- 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 articles 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.
- 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.