Contents

C# Special Characters

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

Special characters are predefined, contextual characters that modify the program element (a literal string, an identifier, or an attribute name) to which they are prepended. C# supports the following special characters:

- @, the verbatim identifier character.
- $, the interpolated string character.

See also

- C# Reference
- C# Programming Guide

$ - string interpolation (C# Reference)

1/15/2019 • 4 minutes to read • Edit Online

The `$` special character identifies a string literal as an *interpolated string*. An interpolated string is a string literal that might contain *interpolated expressions*. When an interpolated string is resolved to a result string, items with interpolated expressions are replaced by the string representations of the expression results. This feature is available in C# 6 and later versions of the language.

String interpolation provides a more readable and convenient syntax to create formatted strings than a string composite formatting feature. The following example uses both features to produce the same output:

```
string name = "Mark";
var date = DateTime.Now;

// Composite formatting:
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date);
// String interpolation:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");
// Both calls produce the same output that is similar to:
// Hello, Mark! Today is Wednesday, it's 19:40 now.
```

Structure of an interpolated string

To identify a string literal as an interpolated string, prepend it with the `$` symbol. You cannot have any white space between the `$` and the `"` that starts a string literal. Doing so causes a compile-time error.

The structure of an item with an interpolated expression is as follows:

```
{<interpolatedExpression>[,<alignment>][:<formatString>]}
```

Elements in square brackets are optional. The following table describes each element:

ELEMENT	DESCRIPTION
`interpolatedExpression`	The expression that produces a result to be formatted. String representation of the `null` result is String.Empty.
`alignment`	The constant expression whose value defines the minimum number of characters in the string representation of the result of the interpolated expression. If positive, the string representation is right-aligned; if negative, it's left-aligned. For more information, see Alignment Component.
`formatString`	A format string that is supported by the type of the expression result. For more information, see Format String Component.

The following example uses optional formatting components described above:

```
Console.WriteLine($"|{"Left",-7}|{"Right",7}|");

const int FieldWidthRightAligned = 20;
Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of the pi number");
Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three decimal digits of the pi number");
// Expected output is:
// |Left   |  Right|
//     3.14159265358979 - default formatting of the pi number
//                3.142 - display only three decimal digits of the pi number
```

Special characters

To include a brace, "{" or "}", in the text produced by an interpolated string, use two braces, "{{" or "}}". For more information, see Escaping Braces.

As the colon (":") has special meaning in an interpolated expression item, in order to use a conditional operator in an interpolated expression, enclose that expression in parentheses.

The following example shows how to include a brace in a result string and how to use a conditional operator in an interpolated expression:

```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a reply :-{{");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Expected output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-{
// Horace is 34 years old.
```

A verbatim interpolated string starts with the `$` character followed by the `@` character. For more information about verbatim strings, see the string and verbatim identifier topics.

NOTE

The `$` token must appear before the `@` token in a verbatim interpolated string.

Implicit conversions and specifying `IFormatProvider` implementation

There are three implicit conversions from an interpolated string:

1. Conversion of an interpolated string to a String instance that is the result of interpolated string resolution with interpolated expression items being replaced with the properly formatted string representations of their results. This conversion uses the current culture.

2. Conversion of an interpolated string to a FormattableString instance that represents a composite format string along with the expression results to be formatted. That allows you to create multiple result strings with culture-specific content from a single FormattableString instance. To do that call one of the following methods:

   - A ToString() overload that produces a result string for the CurrentCulture.
   - An Invariant method that produces a result string for the InvariantCulture.
   - A ToString(IFormatProvider) method that produces a result string for a specified culture.

   You also can use the ToString(IFormatProvider) method to provide a user-defined implementation of the IFormatProvider interface that supports custom formatting. For more information, see Custom Formatting with ICustomFormatter.

3. Conversion of an interpolated string to an IFormattable instance that also allows you to create multiple result strings with culture-specific content from a single IFormattable instance.

The following example uses implicit conversion to FormattableString to create culture-specific result strings:

```
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3} km/s.";

System.Globalization.CultureInfo.CurrentCulture = System.Globalization.CultureInfo.GetCultureInfo("nl-NL");
string messageInCurrentCulture = message.ToString();

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-IN");
string messageInSpecificCulture = message.ToString(specificCulture);

string messageInInvariantCulture = FormattableString.Invariant(message);

Console.WriteLine($"{System.Globalization.CultureInfo.CurrentCulture,-10} {messageInCurrentCulture}");
Console.WriteLine($"{specificCulture,-10} {messageInSpecificCulture}");
Console.WriteLine($"{"Invariant",-10} {messageInInvariantCulture}");
// Expected output is:
// nl-NL      The speed of light is 299.792,458 km/s.
// en-IN      The speed of light is 2,99,792.458 km/s.
// Invariant  The speed of light is 299,792.458 km/s.
```

Additional resources

If you are new to string interpolation, see the String interpolation in C# interactive tutorial. Or you can try the String interpolation in C# tutorial locally on your machine.

See also

- String.Format
- System.FormattableString
- System.IFormattable
- Composite formatting
- Formatting numeric results table
- Strings
- C# Programming Guide
- C# Special Characters
- C# Reference

@ (C# Reference)

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

The `@` special character serves as a verbatim identifier. It can be used in the following ways:

1. To enable C# keywords to be used as identifiers. The `@` character prefixes a code element that the compiler is to interpret as an identifier rather than a C# keyword. The following example uses the `@` character to define an identifier named `for` that it uses in a `for` loop.

```
string[] @for = { "John", "James", "Joan", "Jamie" };
for (int ctr = 0; ctr < @for.Length; ctr++)
{
   Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
// The example displays the following output:
//      Here is your gift, John!
//      Here is your gift, James!
//      Here is your gift, Joan!
//      Here is your gift, Jamie!
```

2. To indicate that a string literal is to be interpreted verbatim. The `@` character in this instance defines a *verbatim string literal*. Simple escape sequences (such as `"\\"` for a backslash), hexadecimal escape sequences (such as `"\x0041"` for an uppercase A), and Unicode escape sequences (such as `"\u0041"` for an uppercase A) are interpreted literally. Only a quote escape sequence (`""`) is not interpreted literally; it produces a single quotation mark. Additionally, in case of a verbatim interpolated string brace escape sequences (`{{` and `}}`) are not interpreted literally; they produce single brace characters. The following example defines two identical file paths, one by using a regular string literal and the other by using a verbatim string literal. This is one of the more common uses of verbatim string literals.

```
string filename1 = @"c:\documents\files\u0066.txt";
string filename2 = "c:\\documents\\files\\u0066.txt";

Console.WriteLine(filename1);
Console.WriteLine(filename2);
// The example displays the following output:
//      c:\documents\files\u0066.txt
//      c:\documents\files\u0066.txt
```

The following example illustrates the effect of defining a regular string literal and a verbatim string literal that contain identical character sequences.

```
string s1 = "He said, \"This is the last \u0063hance\x0021\"";
string s2 = @"He said, ""This is the last \u0063hance\x0021""";

Console.WriteLine(s1);
Console.WriteLine(s2);
// The example displays the following output:
//      He said, "This is the last chance!"
//      He said, "This is the last \u0063hance\x0021"
```

3. To enable the compiler to distinguish between attributes in cases of a naming conflict. An attribute is a class that derives from Attribute. Its type name typically includes the suffix **Attribute**, although the compiler does not enforce this convention. The attribute can then be referenced in code either by its full type name (for

example, `[InfoAttribute]` or its shortened name (for example, `[Info]`). However, a naming conflict occurs if two shortened attribute type names are identical, and one type name includes the **Attribute** suffix but the other does not. For example, the following code fails to compile because the compiler cannot determine whether the `Info` or `InfoAttribute` attribute is applied to the `Example` class. See CS1614 for more information.

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
   private string information;

   public Info(string info)
   {
      information = info;
   }
}

[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute
{
   private string information;

   public InfoAttribute(string info)
   {
      information = info;
   }
}

[Info("A simple executable.")] // Generates compiler error CS1614. Ambiguous Info and InfoAttribute.
// Prepend '@' to select 'Info'. Specify the full name 'InfoAttribute' to select it.
public class Example
{
   [InfoAttribute("The entry point.")]
   public static void Main()
   {
   }
}
```

See also

- C# Reference
- C# Programming Guide
- C# Special Characters