Use string interpolation to construct formatted strings

Article • 09/15/2021

This tutorial teaches you how to use C# string interpolation to insert values into a single result string. You write C# code and see the results of compiling and running it. The tutorial contains a series of lessons that show you how to insert values into a string and format those values in different ways.

This tutorial expects that you have a machine you can use for development. The .NET tutorial Hello World in 10 minutes has instructions for setting up your local development environment on Windows, Linux, or macOS. You can also complete the interactive version of this tutorial in your browser.

Create an interpolated string

Create a directory named *interpolated*. Make it the current directory and run the following command from a console window:

.NET CLI

```
dotnet new console
```

This command creates a new .NET Core console application in the current directory.

Open *Program.cs* in your favorite editor, and replace the line `Console.WriteLine("Hello World!");` with the following code, where you replace `<name>` with your name:

C#

```
var name = "<name>";
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

Try this code by typing `dotnet run` in your console window. When you run the program, it displays a single string that includes your name in the greeting. The string included in the WriteLine method call is an *interpolated string expression*. It's a kind of template that lets you construct a single string (called the *result string*) from a string that includes embedded code. Interpolated strings are particularly useful for inserting values into a string or concatenating (joining together) strings.

This simple example contains the two elements that every interpolated string must have:

- A string literal that begins with the `$` character before its opening quotation mark character. There can't be any spaces between the `$` symbol and the quotation mark character. (If you'd like to see what happens if you include one, insert a space after the `$` character, save the file, and run the program again by typing `dotnet run` in the console window. The C# compiler displays an error message, "error CS1056: Unexpected character '$'".)
- One or more *interpolation expressions*. An interpolation expression is indicated by an opening and closing brace (`{` and `}`). You can put any C# expression that returns a value (including `null`) inside the braces.

Let's try a few more string interpolation examples with some other data types.

Include different data types

In the previous section, you used string interpolation to insert one string inside of another. The result of an interpolation expression can be of any data type, though. Let's include values of various data types in an interpolated string.

In the following example, we first define a class data type `Vegetable` that has a `Name` property and a `ToString` method, which overrides the behavior of the Object.ToString() method. The public access modifier makes that method available to any client code to get the string representation of a `Vegetable` instance. In the example the `Vegetable.ToString` method returns the value of the `Name` property that is initialized at the `Vegetable` constructor:

C#

```
public Vegetable(string name) => Name = name;
```

Then we create an instance of the `Vegetable` class named `item` by using the new operator and providing a name for the constructor `Vegetable`:

C#

```
var item = new Vegetable("eggplant");
```

Finally, we include the `item` variable into an interpolated string that also contains a DateTime value, a Decimal value, and a `Unit` enumeration value. Replace all of the C#

code in your editor with the following code, and then use the `dotnet run` command to run it:

C#

```csharp
using System;

public class Vegetable
{
   public Vegetable(string name) => Name = name;

   public string Name { get; }

   public override string ToString() => Name;
}

public class Program
{
   public enum Unit { item, kilogram, gram, dozen };

   public static void Main()
   {
      var item = new Vegetable("eggplant");
      var date = DateTime.Now;
      var price = 1.99m;
      var unit = Unit.item;
      Console.WriteLine($"On {date}, the price of {item} was {price} per
{unit}.");
   }
}
```

Note that the interpolation expression `item` in the interpolated string resolves to the text "eggplant" in the result string. That's because, when the type of the expression result is not a string, the result is resolved to a string in the following way:

- If the interpolation expression evaluates to `null`, an empty string ("", or String.Empty) is used.

- If the interpolation expression doesn't evaluate to `null`, typically the `ToString` method of the result type is called. You can test this by updating the implementation of the `Vegetable.ToString` method. You might not even need to implement the `ToString` method since every type has some implementation of this method. To test this, comment out the definition of the `Vegetable.ToString` method in the example (to do that, put a comment symbol, `//`, in front of it). In the output, the string "eggplant" is replaced by the fully qualified type name ("Vegetable" in this example), which is the default behavior of the Object.ToString()

method. The default behavior of the `ToString` method for an enumeration value is to return the string representation of the value.

In the output from this example, the date is too precise (the price of eggplant doesn't change every second), and the price value doesn't indicate a unit of currency. In the next section, you'll learn how to fix those issues by controlling the format of string representations of the expression results.

Control the formatting of interpolation expressions

In the previous section, two poorly formatted strings were inserted into the result string. One was a date and time value for which only the date was appropriate. The second was a price that didn't indicate its unit of currency. Both issues are easy to address. String interpolation lets you specify *format strings* that control the formatting of particular types. Modify the call to `Console.WriteLine` from the previous example to include the format strings for the date and price expressions as shown in the following line:

```C#
Console.WriteLine($"On {date:d}, the price of {item} was {price:C2} per
{unit}.");
```

You specify a format string by following the interpolation expression with a colon (":") and the format string. "d" is a standard date and time format string that represents the short date format. "C2" is a standard numeric format string that represents a number as a currency value with two digits after the decimal point.

A number of types in the .NET libraries support a predefined set of format strings. These include all the numeric types and the date and time types. For a complete list of types that support format strings, see Format Strings and .NET Class Library Types in the Formatting Types in .NET article.

Try modifying the format strings in your text editor and, each time you make a change, rerun the program to see how the changes affect the formatting of the date and time and the numeric value. Change the "d" in `{date:d}` to "t" (to display the short time format), "y" (to display the year and month), and "yyyy" (to display the year as a four-digit number). Change the "C2" in `{price:C2}` to "e" (for exponential notation) and "F3" (for a numeric value with three digits after the decimal point).

In addition to controlling formatting, you can also control the field width and alignment of the formatted strings that are included in the result string. In the next section, you'll learn how to do this.

Control the field width and alignment of interpolation expressions

Ordinarily, when the result of an interpolation expression is formatted to string, that string is included in a result string without leading or trailing spaces. Particularly when you work with a set of data, being able to control a field width and text alignment helps to produce a more readable output. To see this, replace all the code in your text editor with the following code, then type `dotnet run` to execute the program:

C#

```csharp
using System;
using System.Collections.Generic;

public class Example
{
   public static void Main()
   {
      var titles = new Dictionary<string, string>()
      {
          ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
          ["London, Jack"] = "Call of the Wild, The",
          ["Shakespeare, William"] = "Tempest, The"
      };

      Console.WriteLine("Author and Title List");
      Console.WriteLine();
      Console.WriteLine($"|{"Author",-25}|{"Title",30}|");
      foreach (var title in titles)
         Console.WriteLine($"|{title.Key,-25}|{title.Value,30}|");
   }
}
```

The names of authors are left-aligned, and the titles they wrote are right-aligned. You specify the alignment by adding a comma (",") after an interpolation expression and designating the *minimum* field width. If the specified value is a positive number, the field is right-aligned. If it is a negative number, the field is left-aligned.

Try removing the negative signs from the `{"Author",-25}` and `{title.Key,-25}` code and run the example again, as the following code does:

C#

```
Console.WriteLine($"|{"Author",25}|{"Title",30}|");
foreach (var title in titles)
   Console.WriteLine($"|{title.Key,25}|{title.Value,30}|");
```

This time, the author information is right-aligned.

You can combine an alignment specifier and a format string for a single interpolation expression. To do that, specify the alignment first, followed by a colon and the format string. Replace all of the code inside the `Main` method with the following code, which displays three formatted strings with defined field widths. Then run the program by entering the `dotnet run` command.

C#

```
Console.WriteLine($"[{DateTime.Now,-20:d}] Hour [{DateTime.Now,-10:HH}]
[{1063.342,15:N2}] feet");
```

The output looks something like the following:

Console

```
[04/14/2018          ] Hour [16        ] [       1,063.34] feet
```

You've completed the string interpolation tutorial.

For more information, see the String interpolation topic and the String interpolation in C# tutorial.