Tutorial: Write a custom string interpolation handler

Article • 04/06/2023

In this tutorial, you'll learn how to:

- ✓ Implement the string interpolation handler pattern
- ✓ Interact with the receiver in a string interpolation operation.
- ✓ Add arguments to the string interpolation handler
- ✓ Understand the new library features for string interpolation

Prerequisites

You'll need to set up your machine to run .NET 6, including the C# 10 compiler. The C# 10 compiler is available starting with Visual Studio 2022 or .NET 6 SDK.

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

New outline

C# 10 adds support for a custom *interpolated string handler*. An interpolated string handler is a type that processes the placeholder expression in an interpolated string. Without a custom handler, placeholders are processed similar to String.Format. Each placeholder is formatted as text, and then the components are concatenated to form the resulting string.

You can write a handler for any scenario where you use information about the resulting string. Will it be used? What constraints are on the format? Some examples include:

- You may require none of the resulting strings are greater than some limit, such as 80 characters. You can process the interpolated strings to fill a fixed-length buffer, and stop processing once that buffer length is reached.
- You may have a tabular format, and each placeholder must have a fixed length. A custom handler can enforce that, rather than forcing all client code to conform.

In this tutorial, you'll create a string interpolation handler for one of the core performance scenarios: logging libraries. Depending on the configured log level, the work to construct a log message isn't needed. If logging is off, the work to construct a string from an interpolated string expression isn't needed. The message is never printed, so any string concatenation can be skipped. In addition, any expressions used in the placeholders, including generating stack traces, doesn't need to be done.

An interpolated string handler can determine if the formatted string will be used, and only perform the necessary work if needed.

Initial implementation

Let's start from a basic `Logger` class that supports different levels:

C#

```
public enum LogLevel
{
    Off,
    Critical,
    Error,
    Warning,
    Information,
    Trace
}

public class Logger
{
    public LogLevel EnabledLevel { get; init; } = LogLevel.Error;

    public void LogMessage(LogLevel level, string msg)
    {
        if (EnabledLevel < level) return;
        Console.WriteLine(msg);
    }
}
```

This `Logger` supports six different levels. When a message won't pass the log level filter, there's no output. The public API for the logger accepts a (fully formatted) string as the message. All the work to create the string has already been done.

Implement the handler pattern

This step is to build an *interpolated string handler* that recreates the current behavior. An interpolated string handler is a type that must have the following characteristics:

- The System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute applied to the type.
- A constructor that has two `int` parameters, `literalLength` and `formattedCount`. (More parameters are allowed).

- A public `AppendLiteral` method with the signature: `public void AppendLiteral(string s)`.
- A generic public `AppendFormatted` method with the signature: `public void AppendFormatted<T>(T t)`.

Internally, the builder creates the formatted string, and provides a member for a client to retrieve that string. The following code shows a `LogInterpolatedStringHandler` type that meets these requirements:

C#

```
[InterpolatedStringHandler]
public ref struct LogInterpolatedStringHandler
{
    // Storage for the built-up string
    StringBuilder builder;

    public LogInterpolatedStringHandler(int literalLength, int for-
mattedCount)
    {
        builder = new StringBuilder(literalLength);
        Console.WriteLine($"\tliteral length: {literalLength}, for-
mattedCount: {formattedCount}");
    }

    public void AppendLiteral(string s)
    {
        Console.WriteLine($"\tAppendLiteral called: {{{s}}}");

        builder.Append(s);
        Console.WriteLine($"\tAppended the literal string");
    }

    public void AppendFormatted<T>(T t)
    {
        Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of
type {typeof(T)}");

        builder.Append(t?.ToString());
        Console.WriteLine($"\tAppended the formatted object");
    }

    internal string GetFormattedText() => builder.ToString();
}
```

You can now add an overload to `LogMessage` in the `Logger` class to try your new interpolated string handler:

C#

```
public void LogMessage(LogLevel level, LogInterpolatedStringHandler
builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

You don't need to remove the original `LogMessage` method, the compiler will prefer a method with an interpolated handler parameter over a method with a `string` parameter when the argument is an interpolated string expression.

You can verify that the new handler is invoked using the following code as the main program:

C#

```
var logger = new Logger() { EnabledLevel = LogLevel.Warning };
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}.
This is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time}.
This won't be printed.");
logger.LogMessage(LogLevel.Warning, "Warning Level. This warning is a
string, not an interpolated string expression.");
```

Running the application produces output similar to the following text:

PowerShell

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be
printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error.
It will be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This won't be printed.}
        Appended the literal string
```

```
Warning Level. This warning is a string, not an interpolated string
expression.
```

Tracing through the output, you can see how the compiler adds code to call the handler and build the string:

- The compiler adds a call to construct the handler, passing the total length of the literal text in the format string, and the number of placeholders.
- The compiler adds calls to `AppendLiteral` and `AppendFormatted` for each section of the literal string and for each placeholder.
- The compiler invokes the `LogMessage` method using the `CoreInterpolatedStringHandler` as the argument.

Finally, notice that the last warning doesn't invoke the interpolated string handler. The argument is a `string`, so that call invokes the other overload with a string parameter.

Add more capabilities to the handler

The preceding version of the interpolated string handler implements the pattern. To avoid processing every placeholder expression, you'll need more information in the handler. In this section, you'll improve your handler so that it does less work when the constructed string won't be written to the log. You use System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute to specify a mapping between parameters to a public API and parameters to a handler's constructor. That provides the handler with the information needed to determine if the interpolated string should be evaluated.

Let's start with changes to the Handler. First, add a field to track if the handler is enabled. Add two parameters to the constructor: one to specify the log level for this message, and the other a reference to the log object:

C#

```csharp
private readonly bool enabled;

public LogInterpolatedStringHandler(int literalLength, int formatted-
Count, Logger logger, LogLevel logLevel)
{
    enabled = logger.EnabledLevel >= logLevel;
    builder = new StringBuilder(literalLength);
    Console.WriteLine($"\tliteral length: {literalLength}, formatted-
Count: {formattedCount}");
}
```

Next, use the field so that your handler only appends literals or formatted objects when the final string will be used:

```C#
public void AppendLiteral(string s)
{
    Console.WriteLine($"\tAppendLiteral called: {{{s}}}");
    if (!enabled) return;

    builder.Append(s);
    Console.WriteLine($"\tAppended the literal string");
}

public void AppendFormatted<T>(T t)
{
    Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of type
{typeof(T)}");
    if (!enabled) return;

    builder.Append(t?.ToString());
    Console.WriteLine($"\tAppended the formatted object");
}
```

Next, you'll need to update the `LogMessage` declaration so that the compiler passes the additional parameters to the handler's constructor. That's handled using the System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute on the handler argument:

```C#
public void LogMessage(LogLevel level, [InterpolatedStringHandlerAr-
gument("", "level")] LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

This attribute specifies the list of arguments to `LogMessage` that map to the parameters that follow the required `literalLength` and `formattedCount` parameters. The empty string (""), specifies the receiver. The compiler substitutes the value of the `Logger` object represented by `this` for the next argument to the handler's constructor. The compiler substitutes the value of `level` for the following argument. You can provide any number of arguments for any handler you write. The arguments that you add are string arguments.

You can run this version using the same test code. This time, you'll see the following results:

PowerShell

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be
printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error.
It will be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
        AppendLiteral called: {. This won't be printed.}
Warning Level. This warning is a string, not an interpolated string
expression.
```

You can see that the `AppendLiteral` and `AppendFormat` methods are being called, but they aren't doing any work. The handler has determined that the final string won't be needed, so the handler doesn't build it. There are still a couple of improvements to make.

First, you can add an overload of `AppendFormatted` that constrains the argument to a type that implements System.IFormattable. This overload enables callers to add format strings in the placeholders. While making this change, let's also change the return type of the other `AppendFormatted` and `AppendLiteral` methods, from `void` to `bool` (if any of these methods have different return types, then you'll get a compilation error). That change enables *short circuiting*. The methods return `false` to indicate that processing of the interpolated string expression should be stopped. Returning `true` indicates that it should continue. In this example, you're using it to stop processing when the resulting string isn't needed. Short circuiting supports more fine-grained actions. You could stop processing the expression once it reaches a certain length, to support fixed-length buffers. Or some condition could indicate remaining elements aren't needed.

C#

```
public void AppendFormatted<T>(T t, string format) where T :
IFormattable
{
    Console.WriteLine($"\tAppendFormatted (IFormattable version)
called: {t} with format {{{format}}} is of type {typeof(T)},");

    builder.Append(t?.ToString(format, null));
    Console.WriteLine($"\tAppended the formatted object");
}
```

With that addition, you can specify format strings in your interpolated string expression:

```C#
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}.
The time doesn't use formatting.");
logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime:
{time:t}. This is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime:
{time:t}. This won't be printed.");
```

The `:t` on the first message specifies the "short time format" for the current time. The previous example showed one of the overloads to the `AppendFormatted` method that you can create for your handler. You don't need to specify a generic argument for the object being formatted. You may have more efficient ways to convert types you create to string. You can write overloads of `AppendFormatted` that takes those types instead of a generic argument. The compiler will pick the best overload. The runtime uses this technique to convert System.Span<T> to string output. You can add an integer parameter to specify the *alignment* of the output, with or without an IFormattable. The System.Runtime.CompilerServices.DefaultInterpolatedStringHandler that ships with .NET 6 contains nine overloads of AppendFormatted for different uses. You can use it as a reference while building a handler for your purposes.

Run the sample now, and you'll see that for the `Trace` message, only the first `AppendLiteral` is called:

```PowerShell
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:18:29 PM is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:18:29 PM. The time doesn't
use formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021
12:18:29 PM with format {t} is of type System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:18 PM. This is an error. It will be
printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: Trace Level. CurrentTime:
Warning Level. This warning is a string, not an interpolated string
expression.
```

You can make one final update to the handler's constructor that improves efficiency. The handler can add a final `out bool` parameter. Setting that parameter to `false` indicates that the handler shouldn't be called at all to process the interpolated string expression:

```C#
public LogInterpolatedStringHandler(int literalLength, int formatted-
Count, Logger logger, LogLevel level, out bool isEnabled)
{
    isEnabled = logger.EnabledLevel >= level;
    Console.WriteLine($"\tliteral length: {literalLength}, formatted-
Count: {formattedCount}");
    builder = isEnabled ? new StringBuilder(literalLength) : de-
fault!;
}
```

That change means you can remove the `enabled` field. Then, you can change the return type of `AppendLiteral` and `AppendFormatted` to `void`. Now, when you run the sample, you'll see the following output:

```PowerShell
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:19:10 PM is of type
System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. The time doesn't
use formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021
```

```
12:19:10 PM with format {t} is of type System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:19 PM. This is an error. It will be
printed.
        literal length: 50, formattedCount: 1
Warning Level. This warning is a string, not an interpolated string
expression.
```

The only output when `LogLevel.Trace` was specified is the output from the constructor. The handler indicated that it's not enabled, so none of the `Append` methods were invoked.

This example illustrates an important point for interpolated string handlers, especially when logging libraries are used. Any side-effects in the placeholders may not occur. Add the following code to your main program and see this behavior in action:

C#

```csharp
int index = 0;
int numberOfIncrements = 0;
for (var level = LogLevel.Critical; level <= LogLevel.Trace; level++)
{
    Console.WriteLine(level);
    logger.LogMessage(level, $"{level}: Increment index a few times
{index++}, {index++}, {index++}, {index++}, {index++}");
    numberOfIncrements += 5;
}
Console.WriteLine($"Value of index {index}, value of numberOfIncre-
ments: {numberOfIncrements}");
```

You can see the `index` variable is incremented five times each iteration of the loop. Because the placeholders are evaluated only for `Critical`, `Error` and `Warning` levels, not for `Information` and `Trace`, the final value of `index` doesn't match the expectation:

PowerShell

```
Critical
Critical: Increment index a few times 0, 1, 2, 3, 4
Error
Error: Increment index a few times 5, 6, 7, 8, 9
Warning
Warning: Increment index a few times 10, 11, 12, 13, 14
Information
Trace
Value of index 15, value of numberOfIncrements: 25
```

Interpolated string handlers provide greater control over how an interpolated string expression is converted to a string. The .NET runtime team has already used this feature to improve performance in several areas. You can make use of the same capability in your own libraries. To explore further, look at the System.Runtime.CompilerServices.DefaultInterpolatedStringHandler. It provides a more complete implementation than you built here. You'll see many more overloads that are possible for the `Append` methods.