Determine caller information using attributes interpreted by the C# compiler

Article • 06/15/2022

Using info attributes, you obtain information about the caller to a method. You obtain the file path of the source code, the line number in the source code, and the member name of the caller. To obtain member caller information, you use attributes that are applied to optional parameters. Each optional parameter specifies a default value. The following table lists the Caller Info attributes that are defined in the System.Runtime.CompilerServices namespace:

Attribute	Description	Type
CallerFilePathAttribute	Full path of the source file that contains the caller. The full path is the path at compile time.	`String`
CallerLineNumberAttribute	Line number in the source file from which the method is called.	`Integer`
CallerMemberNameAttribute	Method name or property name of the caller.	`String`
CallerArgumentExpressionAttribute	String representation of the argument expression.	`String`

This information helps you with tracing and debugging, and helps you to create diagnostic tools. The following example shows how to use caller info attributes. On each call to the `TraceMessage` method, the caller information is inserted for the arguments to the optional parameters.

C#

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
        [CallerMemberName] string memberName = "",
        [CallerFilePath] string sourceFilePath = "",
        [CallerLineNumber] int sourceLineNumber = 0)
{
    Trace.WriteLine("message: " + message);
    Trace.WriteLine("member name: " + memberName);
    Trace.WriteLine("source file path: " + sourceFilePath);
```

```
    Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
//  message: Something happened.
//  member name: DoProcessing
//  source file path: c:\Visual Studio
Projects\CallerInfoCS\CallerInfoCS\Form1.cs
//  source line number: 31
```

You specify an explicit default value for each optional parameter. You can't apply caller info attributes to parameters that aren't specified as optional. The caller info attributes don't make a parameter optional. Instead, they affect the default value that's passed in when the argument is omitted. Caller info values are emitted as literals into the Intermediate Language (IL) at compile time. Unlike the results of the StackTrace property for exceptions, the results aren't affected by obfuscation. You can explicitly supply the optional arguments to control the caller information or to hide caller information.

Member names

You can use the `CallerMemberName` attribute to avoid specifying the member name as a `String` argument to the called method. By using this technique, you avoid the problem that **Rename Refactoring** doesn't change the `String` values. This benefit is especially useful for the following tasks:

- Using tracing and diagnostic routines.
- Implementing the INotifyPropertyChanged interface when binding data. This interface allows the property of an object to notify a bound control that the property has changed. The control can display the updated information. Without the `CallerMemberName` attribute, you must specify the property name as a literal.

The following chart shows the member names that are returned when you use the `CallerMemberName` attribute.

Calls occur within	Member name result
Method, property, or event	The name of the method, property, or event from which the call originated.
Constructor	The string ".ctor"
Static constructor	The string ".cctor"
Finalizer	The string "Finalize"

Calls occur within	Member name result
User-defined operators or conversions	The generated name for the member, for example, "op_Addition".
Attribute constructor	The name of the method or property to which the attribute is applied. If the attribute is any element within a member (such as a parameter, a return value, or a generic type parameter), this result is the name of the member that's associated with that element.
No containing member (for example, assembly-level or attributes that are applied to types)	The default value of the optional parameter.

Argument expressions

You use the System.Runtime.CompilerServices.CallerArgumentExpressionAttribute when you want the expression passed as an argument. Diagnostic libraries may want to provide more details about the *expressions* passed to arguments. By providing the expression that triggered the diagnostic, in addition to the parameter name, developers have more details about the condition that triggered the diagnostic. That extra information makes it easier to fix.

The following example shows how you can provide detailed information about the argument when it's invalid:

C#

```
public static void ValidateArgument(string parameterName, bool condi-
tion, [CallerArgumentExpression("condition")] string? message=null)
{
    if (!condition)
    {
        throw new ArgumentException($"Argument failed validation:
<{message}>", parameterName);
    }
}
```

You would invoke it as shown in the following example:

C#

```
public void Operation(Action func)
{
    Utilities.ValidateArgument(nameof(func), func is not null);
```

```
    func();
}
```

The expression used for `condition` is injected by the compiler into the `message` argument. When a developer calls `Operation` with a `null` argument, the following message is stored in the `ArgumentException`:

.NET CLI

```
Argument failed validation: <func is not null>
```

This attribute enables you to write diagnostic utilities that provide more details. Developers can more quickly understand what changes are needed. You can also use the CallerArgumentExpressionAttribute to determine what expression was used as the receiver for extension methods. The following method samples a sequence at regular intervals. If the sequence has fewer elements than the frequency, it reports an error:

C#

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sequence,
int frequency,
    [CallerArgumentExpression(nameof(sequence))] string? message =
null)
{
    if (sequence.Count() < frequency)
        throw new ArgumentException($"Expression doesn't have enough
elements: {message}", nameof(sequence));
    int i = 0;
    foreach (T item in sequence)
    {
        if (i++ % frequency == 0)
            yield return item;
    }
}
```

The previous example uses the nameof operator for the parameter `sequence`. That feature is available in C# 11. Before C# 11, you'll need to type the name of the parameter as a string. You could call this method as follows:

C#

```
sample = Enumerable.Range(0, 10).Sample(100);
```

The preceding example would throw an ArgumentException whose message is the following text:

.NET CLI

```
Expression doesn't have enough elements: Enumerable.Range(0, 10) (Pa-
rameter 'sequence')
```

See also

- Named and Optional Arguments
- System.Reflection
- Attribute
- Attributes