Contents

XML Documentation Comments (C# Programming Guide)

1/29/2019 • 2 minutes to read • Edit Online

In Visual C# you can create documentation for your code by including XML elements in special comment fields (indicated by triple slashes) in the source code directly before the code block to which the comments refer, for example:

```
/// <summary>
///  This class performs an important function.
/// </summary>
public class MyClass {}
```

When you compile with the /doc option, the compiler will search for all XML tags in the source code and create an XML documentation file. To create the final documentation based on the compiler-generated file, you can create a custom tool or use a tool such as DocFX or Sandcastle.

To refer to XML elements (for example, your function processes specific XML elements that you want to describe in an XML documentation comment), you can use the standard quoting mechanism (`<` and `>`). To refer to generic identifiers in code reference (`cref`) elements, you can use either the escape characters (for example, `cref="List&lt;T&gt;"`) or braces (`cref="List{T}"`). As a special case, the compiler parses the braces as angle brackets to make the documentation comment less cumbersome to author when referring to generic identifiers.

NOTE

The XML documentation comments are not metadata; they are not included in the compiled assembly and therefore they are not accessible through reflection.

In This Section

- Recommended Tags for Documentation Comments
- Processing the XML File
- Delimiters for Documentation Tags
- How to: Use the XML Documentation Features

Related Sections

For more information, see:

- /doc (Process Documentation Comments)

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide

Recommended Tags for Documentation Comments (C# Programming Guide)

1/29/2019 • 2 minutes to read • Edit Online

The C# compiler processes documentation comments in your code and formats them as XML in a file whose name you specify in the **/doc** command-line option. To create the final documentation based on the compiler-generated file, you can create a custom tool, or use a tool such as DocFX or Sandcastle.

Tags are processed on code constructs such as types and type members.

> **NOTE**
>
> Documentation comments cannot be applied to a namespace.

The compiler will process any tag that is valid XML. The following tags provide generally used functionality in user documentation.

Tags

<c>	<para>	<see>*
<code>	<param>*	<seealso>*
<example>	<paramref>	<summary>
<exception>*	<permission>*	<typeparam>*
<include>*	<remarks>	<typeparamref>
<list>	<returns>	<value>

(* denotes that the compiler verifies syntax.)

If you want angle brackets to appear in the text of a documentation comment, use `<` and `>`, as shown in the following example.

```
/// <summary cref="C < T >">
/// </summary>
```

See also

- C# Programming Guide
- /doc (C# Compiler Options)
- XML Documentation Comments

<c> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<c>text</c>
```

Parameters

`text`
The text you would like to indicate as code.

Remarks

The `<c>` tag gives you a way to indicate that text within a description should be marked as code. Use <code> to indicate multiple lines as code.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary><c>DoWork</c> is a method in the <c>TestClass</c> class.
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<code> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<code>content</code>
```

Parameters

`content`

The text you want marked as code.

Remarks

The <code> tag gives you a way to indicate multiple lines as code. Use <c> to indicate that text within a description should be marked as code.

Compile with /doc to process documentation comments to a file.

Example

See the <example> topic for an example of how to use the <code> tag.

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

cref Attribute (C# Programming Guide)

1/29/2019 • 2 minutes to read • Edit Online

The `cref` attribute in an XML documentation tag means "code reference." It specifies that the inner text of the tag is a code element, such as a type, method, or property. Documentation tools like DocFX and Sandcastle use the `cref` attributes to automatically generate hyperlinks to the page where the type or member is documented.

Example

The following example shows `cref` attributes used in <see> tags.

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// <code>
        /// class TestClass
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
        /// }
        /// </code>
        /// </example>
        public static int GetZero()
        {
            return 0;
        }

        /// <summary>
        /// The GetGenericValue method.
        /// </summary>
        /// <remarks>
        /// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
        /// </remarks>

        public static T GetGenericValue<T>(T para)
        public static T GetGenericValue<T>(T para)
        {
            return para;
        }
    }

    /// <summary>
    /// GenericClass.
    /// </summary>
    /// <remarks>
    /// This example shows how to specify the <see cref="GenericClass{T}"/> type as a cref attribute.
    /// </remarks>
    class GenericClass<T>
    {
        // Fields and members.
    }

    class Program
    {
        static int Main()
        {
            return TestClass.GetZero();
        }
    }
}
```

When compiled, the program produces the following XML file. Notice that the `cref` attribute for the `GetZero` method, for example, has been transformed by the compiler to `"M:TestNamespace.TestClass.GetZero"`. The "M:" prefix means "method" and is a convention that is recognized by documentation tools such as DocFX and Sandcastle. For a complete list of prefixes, see Processing the XML File.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>CRefTest</name>
    </assembly>
    <members>
        <member name="T:TestNamespace.TestClass">
            <summary>
            TestClass contains cref examples.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor">
            <summary>
            This sample shows how to specify the <see cref="T:TestNamespace.TestClass"/> constructor as a cref
attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor(System.Int32)">
            <summary>
            This sample shows how to specify the <see cref="M:TestNamespace.TestClass.#ctor(System.Int32)"/>
constructor as a cref attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.GetZero">
            <summary>
            The GetZero method.
            </summary>
            <example>
            This sample shows how to call the <see cref="M:TestNamespace.TestClass.GetZero"/> method.
            <code>
            class TestClass
            {
                static int Main()
                {
                    return GetZero();
                }
            }
            </code>
            </example>
        </member>
        <member name="M:TestNamespace.TestClass.GetGenericValue``1(``0)">
            <summary>
            The GetGenericValue method.
            </summary>
            <remarks>
            This sample shows how to specify the <see
cref="M:TestNamespace.TestClass.GetGenericValue``1(``0)"/> method as a cref attribute.
            </remarks>
        </member>
        <member name="T:TestNamespace.GenericClass`1">
            <summary>
            GenericClass.
            </summary>
            <remarks>
            This example shows how to specify the <see cref="T:TestNamespace.GenericClass`1"/> type as a cref
attribute.
            </remarks>
        </member>
    </members>    <members>
        <member name="T:TestNamespace.TestClass">
            <summary>
            TestClass contains two cref examples.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.GetZero">
            <summary>
            The GetZero method.
            </summary>
            <example> This sample shows how to call the <see cref="M:TestNamespace.TestClass.GetZero"/>
method.
            <code>
            class TestClass
            {
                static int Main()
                {
                    return GetZero();
                }
            }
            </code>
            </example>
        </member>
        <member name="M:TestNamespace.TestClass.GetGenericValue``1(``0)">
            <summary>
            The GetGenericValue method.
            </summary>
            <remarks>
            This sample shows how to specify the <see
cref="M:TestNamespace.TestClass.GetGenericValue``1(``0)"/> method as a cref attribute.
            </remarks>
        </member>
        <member name="T:TestNamespace.GenericClass`1">
            <summary>
            GenericClass.
            </summary>
            <remarks>
            This example shows how to specify the <see cref="T:TestNamespace.GenericClass`1"/> type as a cref
attribute.
            </remarks>
        </member>
    </members>
</doc>
```

See also

- XML Documentation Comments
- Recommended Tags for Documentation Comments

<example> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<example>description</example>
```

Parameters

`description`
A description of the code sample.

Remarks

The <example> tag lets you specify an example of how to use a method or other library member. This commonly involves using the <code> tag.

Compile with /doc to process documentation comments to a file.

Example

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// <code>
        /// class TestClass
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
        /// }
        /// </code>
        /// </example>
        public static int GetZero()
        {
            return 0;
        }

        /// <summary>
        /// The GetGenericValue method.
        /// </summary>
        /// <remarks>
        /// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
        /// </remarks>

        public static T GetGenericValue<T>(T para)
        {
            return para;
        }
    }

    /// <summary>
    /// GenericClass.
    /// </summary>
    /// <remarks>
    /// This example shows how to specify the <see cref="GenericClass{T}"/> type as a cref attribute.
    /// </remarks>
    class GenericClass<T>
    {
        // Fields and members.
    }

    class Program
    {
        static int Main()
        {
            return TestClass.GetZero();
        }
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<exception> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<exception cref="member">description</exception>
```

Parameters

cref = " `member` "
A reference to an exception that is available from the current compilation environment. The compiler checks that the given exception exists and translates `member` to the canonical element name in the output XML. `member` must appear within double quotation marks (" ").

For more information on how to create a cref reference to a generic type, see <see>.

`description`
A description of the exception.

Remarks

The <exception> tag lets you specify which exceptions can be thrown. This tag can be applied to definitions for methods, properties, events, and indexers.

Compile with /doc to process documentation comments to a file.

For more information about exception handling, see Exceptions and Exception Handling.

Example

```
// compile with: -doc:DocFileName.xml

/// Comment for class
public class EClass : System.Exception
{
    // class definition...
}

/// Comment for class
class TestClass
{
    /// <exception cref="System.Exception">Thrown when...</exception>
    public void DoSomething()
    {
        try
        {
        }
        catch (EClass)
        {
        }
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<include> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<include file='filename' path='tagpath[@name="id"]' />
```

Parameters

`filename`
The name of the XML file containing the documentation. The file name can be qualified with a path relative to the source code file. Enclose `filename` in single quotation marks (' ').

`tagpath`
The path of the tags in `filename` that leads to the tag `name`. Enclose the path in single quotation marks (' ').

`name`
The name specifier in the tag that precedes the comments; `name` will have an `id`.

`id`
The ID for the tag that precedes the comments. Enclose the ID in double quotation marks (" ").

Remarks

The <include> tag lets you refer to comments in another file that describe the types and members in your source code. This is an alternative to placing documentation comments directly in your source code file. By putting the documentation in a separate file, you can apply source control to the documentation separately from the source code. One person can have the source code file checked out and someone else can have the documentation file checked out.

The <include> tag uses the XML XPath syntax. Refer to XPath documentation for ways to customize your <include> use.

Example

This is a multifile example. The first file, which uses <include>, is listed below:

```
// compile with: -doc:DocFileName.xml

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
    static void Main()
    {
    }
}

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
    public void Test()
    {
    }
}
```

The second file, xml_include_tag.doc, contains the following documentation comments:

```
<MyDocs>

<MyMembers name="test">
<summary>
The summary for this type.
</summary>
</MyMembers>

<MyMembers name="test2">
<summary>
The summary for this other type.
</summary>
</MyMembers>

</MyDocs>
```

Program Output

The following output is generated when you compile the Test and Test2 classes with the following command line: `/doc:DocFileName.xml.` In Visual Studio, you specify the XML doc comments option in the Build pane of the Project Designer. When the C# compiler sees the <include> tag, it will search for documentation comments in xml_include_tag.doc instead of the current source file. The compiler then generates DocFileName.xml, and this is the file that is consumed by documentation tools such as DocFX and Sandcastle to produce the final documentation.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xml_include_tag</name>
    </assembly>
    <members>
        <member name="T:Test">
            <summary>
The summary for this type.
</summary>
        </member>
        <member name="T:Test2">
            <summary>
The summary for this other type.
</summary>
        </member>
    </members>
</doc>
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<list> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<list type="bullet" | "number" | "table">
    <listheader>
        <term>term</term>
        <description>description</description>
    </listheader>
    <item>
        <term>term</term>
        <description>description</description>
    </item>
</list>
```

Parameters

`term`
A term to define, which will be defined in `description`.

`description`
Either an item in a bullet or numbered list or the definition of a `term`.

Remarks

The <listheader> block is used to define the heading row of either a table or definition list. When defining a table, you only need to supply an entry for term in the heading.

Each item in the list is specified with an <item> block. When creating a definition list, you will need to specify both `term` and `description`. However, for a table, bulleted list, or numbered list, you only need to supply an entry for `description`.

A list or table can have as many <item> blocks as needed.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <item>
    /// <description>Item 1.</description>
    /// </item>
    /// <item>
    /// <description>Item 2.</description>
    /// </item>
    /// </list>
    /// </summary>
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<para> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<para>content</para>
```

Parameters

`content`
The text of the paragraph.

Remarks

The <para> tag is for use inside a tag, such as <summary>, <remarks>, or <returns>, and lets you add structure to the text.

Compile with /doc to process documentation comments to a file.

Example

See <summary> for an example of using <para>.

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<param> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<param name="name">description</param>
```

Parameters

`name`
The name of a method parameter. Enclose the name in double quotation marks (" ").

`description`
A description for the parameter.

Remarks

The <param> tag should be used in the comment for a method declaration to describe one of the parameters for the method. To document multiple parameters, use multiple <param> tags.

The text for the <param> tag will be displayed in IntelliSense, the Object Browser, and in the Code Comment Web Report.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    // Single parameter.
    /// <param name="Int1">Used to indicate status.</param>
    public static void DoWork(int Int1)
    {
    }

    // Multiple parameters.
    /// <param name="Int1">Used to indicate status.</param>
    /// <param name="Float1">Used to specify context.</param>
    public static void DoWork(int Int1, float Float1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide

- Recommended Tags for Documentation Comments

<paramref> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<paramref name="name"/>
```

Parameters

`name`
The name of the parameter to refer to. Enclose the name in double quotation marks (" ").

Remarks

The <paramref> tag gives you a way to indicate that a word in the code comments, for example in a <summary> or <remarks> block refers to a parameter. The XML file can be processed to format this word in some distinct way, such as with a bold or italic font.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// The <paramref name="int1"/> parameter takes a number.
    /// </summary>
    public static void DoWork(int int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<permission> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<permission cref="member">description</permission>
```

Parameters

cref = " `member` "
A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and translates `member` to the canonical element name in the output XML. *member* must appear within double quotation marks (" ").

For information on how to create a cref reference to a generic type, see <see>.

`description`
A description of the access to the member.

Remarks

The <permission> tag lets you document the access of a member. The PermissionSet class lets you specify access to a member.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

class TestClass
{
    /// <permission cref="System.Security.PermissionSet">Everyone can access this method.</permission>
    public static void Test()
    {
    }

    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<remarks> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<remarks>description</remarks>
```

Parameters

`Description`

A description of the member.

Remarks

The <remarks> tag is used to add information about a type, supplementing the information specified with <summary>. This information is displayed in the Object Browser window.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this class.
/// </summary>
/// <remarks>
/// You may have some additional information about this class.
/// </remarks>
public class TestClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<returns> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<returns>description</returns>
```

Parameters

`description`
A description of the return value.

Remarks

The <returns> tag should be used in the comment for a method declaration to describe the return value.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <returns>Returns zero.</returns>
    public static int GetZero()
    {
        return 0;
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<see> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<see cref="member"/>
```

Parameters

cref = " `member` "

A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes `member` to the element name in the output XML. Place *member* within double quotation marks (" ").

Remarks

The <see> tag lets you specify a link from within text. Use <seealso> to indicate that text should be placed in a See Also section. Use the cref Attribute to create internal hyperlinks to documentation pages for code elements.

Compile with -doc to process documentation comments to a file.

The following example shows a <see> tag within a summary section.

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
cref="System.Console.WriteLine(System.String)"/> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<seealso> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<seealso cref="member"/>
```

Parameters

cref = " `member` "

A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes `member` to the element name in the output XML. `member` must appear within double quotation marks (" ").

For information on how to create a cref reference to a generic type, see <see>.

Remarks

The <seealso> tag lets you specify the text that you might want to appear in a See Also section. Use <see> to specify a link from within text.

Compile with /doc to process documentation comments to a file.

Example

See <summary> for an example of using <seealso>.

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<summary> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<summary>description</summary>
```

Parameters

`description`
A summary of the object.

Remarks

The <summary> tag should be used to describe a type or a type member. Use <remarks> to add supplemental information to a type description. Use the cref Attribute to enable documentation tools such as DocFX and Sandcastle to create internal hyperlinks to documentation pages for code elements.

The text for the <summary> tag is the only source of information about the type in IntelliSense, and is also displayed in the Object Browser Window.

Compile with /doc to process documentation comments to a file. To create the final documentation based on the compiler-generated file, you can create a custom tool, or use a tool such as DocFX or Sandcastle.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
cref="System.Console.WriteLine(System.String)"/> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

The previous example produces the following XML file.

```xml
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>YourNamespace</name>
    </assembly>
    <members>
        <member name="T:DotNetEvents.TestClass">
            text for class TestClass
        </member>
        <member name="M:DotNetEvents.TestClass.DoWork(System.Int32)">
            <summary>DoWork is a method in the TestClass class.
            <para>Here's how you could make a second paragraph in a description. <see
cref="M:System.Console.WriteLine(System.String)"/> for information about output statements.</para>
            <seealso cref="M:DotNetEvents.TestClass.Main"/>
            </summary>
        </member>
        <member name="M:DotNetEvents.TestClass.Main">
            text for Main
        </member>
    </members>
</doc>
```

Example

The following example shows how to make a `cref` reference to a generic type.

```csharp
// compile with: -doc:DocFileName.xml

// the following cref shows how to specify the reference, such that,
// the compiler will resolve the reference
/// <summary cref="C{T}">
/// </summary>
class A { }

// the following cref shows another way to specify the reference,
// such that, the compiler will resolve the reference
// <summary cref="C &lt; T &gt;">

// the following cref shows how to hard-code the reference
/// <summary cref="T:C`1">
/// </summary>
class B { }

/// <summary cref="A">
/// </summary>
/// <typeparam name="T"></typeparam>
class C<T> { }
```

The previous example produces the following XML file.

```xml
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>YourNamespace</name>
    </assembly>
    <members>
        <member name="T:ExtensionMethodsDemo1.A">
            <summary cref="T:ExtensionMethodsDemo1.C`1">
            </summary>
        </member>
        <member name="T:ExtensionMethodsDemo1.B">
            <summary cref="T:C`1">
            </summary>
        </member>
        <member name="T:ExtensionMethodsDemo1.C`1">
            <summary cref="T:ExtensionMethodsDemo1.A">
            </summary>
            <typeparam name="T"></typeparam>
        </member>
    </members>
</doc>
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<typeparam> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<typeparam name="name">description</typeparam>
```

Parameters

`name`
The name of the type parameter. Enclose the name in double quotation marks (" ").

`description`
A description for the type parameter.

Remarks

The `<typeparam>` tag should be used in the comment for a generic type or method declaration to describe a type parameter. Add a tag for each type parameter of the generic type or method.

For more information, see Generics.

The text for the `<typeparam>` tag will be displayed in IntelliSense, the Object Browser Window code comment web report.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

See also

- C# Reference
- C# Programming Guide
- Recommended Tags for Documentation Comments

<typeparamref> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<typeparamref name="name"/>
```

Parameters

`name`
The name of the type parameter. Enclose the name in double quotation marks (" ").

Remarks

For more information on type parameters in generic types and methods, see Generics.

Use this tag to enable consumers of the documentation file to format the word in some distinct way, for example in italics.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

<value> (C# Programming Guide)

1/30/2019 • 2 minutes to read • Edit Online

Syntax

```
<value>property-description</value>
```

Parameters

`property-description`
A description for the property.

Remarks

The <value> tag lets you describe the value that a property represents. Note that when you add a property via code wizard in the Visual Studio .NET development environment, it will add a <summary> tag for the new property. You should then manually add a <value> tag to describe the value that the property represents.

Compile with /doc to process documentation comments to a file.

Example

```
// compile with: -doc:DocFileName.xml

/// text for class Employee
public class Employee
{
    private string _name;

    /// <summary>The Name property represents the employee's name.</summary>
    /// <value>The Name property gets/sets the value of the string field, _name.</value>

    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }
}

/// text for class MainClass
public class MainClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

See also

- C# Programming Guide
- Recommended Tags for Documentation Comments

Processing the XML File (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

The compiler generates an ID string for each construct in your code that is tagged to generate documentation. (For information about how to tag your code, see Recommended Tags for Documentation Comments.) The ID string uniquely identifies the construct. Programs that process the XML file can use the ID string to identify the corresponding .NET Framework metadata/reflection item that the documentation applies to.

The XML file is not a hierarchical representation of your code; it is a flat list that has a generated ID for each element.

The compiler observes the following rules when it generates the ID strings:

- No white space is in the string.
- The first part of the ID string identifies the kind of member being identified, by way of a single character followed by a colon. The following member types are used:

CHARACTER	DESCRIPTION
N	namespace You cannot add documentation comments to a namespace, but you can make cref references to them, where supported.
T	type: class, interface, struct, enum, delegate
F	field
P	property (including indexers or other indexed properties)
M	method (including such special methods as constructors, operators, and so forth)
E	event
!	error string The rest of the string provides information about the error. The C# compiler generates error information for links that cannot be resolved.

- The second part of the string is the fully qualified name of the item, starting at the root of the namespace. The name of the item, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they are replaced by the hash-sign ('#'). It is assumed that no item has a hash-sign directly in its name. For example, the fully qualified name of the String constructor would be "System.String.#ctor".
- For properties and methods, if there are arguments to the method, the argument list enclosed in parentheses follows. If there are no arguments, no parentheses are present. The arguments are separated by commas. The encoding of each argument follows directly how it is encoded in a .NET Framework signature:
  - Base types. Regular types (ELEMENT_TYPE_CLASS or ELEMENT_TYPE_VALUETYPE) are

represented as the fully qualified name of the type.

- Intrinsic types (for example, ELEMENT_TYPE_I4, ELEMENT_TYPE_OBJECT, ELEMENT_TYPE_STRING, ELEMENT_TYPE_TYPEDBYREF. and ELEMENT_TYPE_VOID) are represented as the fully qualified name of the corresponding full type. For example, System.Int32 or System.TypedReference.
- ELEMENT_TYPE_PTR is represented as a '*' following the modified type.
- ELEMENT_TYPE_BYREF is represented as a '@' following the modified type.
- ELEMENT_TYPE_PINNED is represented as a '^' following the modified type. The C# compiler never generates this.
- ELEMENT_TYPE_CMOD_REQ is represented as a '|' and the fully qualified name of the modifier class, following the modified type. The C# compiler never generates this.
- ELEMENT_TYPE_CMOD_OPT is represented as a '!' and the fully qualified name of the modifier class, following the modified type.
- ELEMENT_TYPE_SZARRAY is represented as "[]" following the element type of the array.
- ELEMENT_TYPE_GENERICARRAY is represented as "[?]" following the element type of the array. The C# compiler never generates this.
- ELEMENT_TYPE_ARRAY is represented as [*lowerbound*:`size`,*lowerbound*:`size`] where the number of commas is the rank - 1, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is simply omitted. If the lower bound and size for a particular dimension are omitted, the ':' is omitted as well. For example, a 2-dimensional array with 1 as the lower bounds and unspecified sizes is [1:,1:].
- ELEMENT_TYPE_FNPTR is represented as "=FUNC:`type`(*signature*)", where `type` is the return type, and *signature* is the arguments of the method. If there are no arguments, the parentheses are omitted. The C# compiler never generates this.

The following signature components are not represented because they are never used for differentiating overloaded methods:

- calling convention
- return type
- ELEMENT_TYPE_SENTINEL

- For conversion operators only (op_Implicit and op_Explicit), the return value of the method is encoded as a '~' followed by the return type, as encoded above.
- For generic types, the name of the type is followed by a backtick and then a number that indicates the number of generic type parameters. For example:

  `<member name="T:SampleClass`2">` is the tag for a type that is defined as `public class SampleClass<T, U>`.

  For methods taking generic types as parameters, the generic type parameters are specified as numbers prefaced with backticks (for example `0,`1). Each number representing a zero-based array notation for the type's generic parameters.

Examples

The following examples show how the ID strings for a class and its members would be generated:

```
namespace N
{
    /// <summary>
    /// Enter description here for class X.
    /// ID string generated is "T:N.X".
    /// </summary>
    public unsafe class X
    {
        /// <summary>
        /// Enter description here for the first constructor.
        /// ID string generated is "M:N.X.#ctor".
        /// </summary>
        public X() { }


        /// <summary>
        /// Enter description here for the second constructor.
        /// ID string generated is "M:N.X.#ctor(System.Int32)".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public X(int i) { }


        /// <summary>
        /// Enter description here for field q.
        /// ID string generated is "F:N.X.q".
        /// </summary>
        public string q;


        /// <summary>
        /// Enter description for constant PI.
        /// ID string generated is "F:N.X.PI".
        /// </summary>
        public const double PI = 3.14;


        /// <summary>
        /// Enter description for method f.
        /// ID string generated is "M:N.X.f".
        /// </summary>
        /// <returns>Describe return value.</returns>
        public int f() { return 1; }


        /// <summary>
        /// Enter description for method bb.
        /// ID string generated is "M:N.X.bb(System.String,System.Int32@,System.Void*)".
        /// </summary>
        /// <param name="s">Describe parameter.</param>
        /// <param name="y">Describe parameter.</param>
        /// <param name="z">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int bb(string s, ref int y, void* z) { return 1; }


        /// <summary>
        /// Enter description for method gg.
        /// ID string generated is "M:N.X.gg(System.Int16[],System.Int32[0:,0:])".
        /// </summary>
        /// <param name="array1">Describe parameter.</param>
        /// <param name="array">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public int gg(short[] array1, int[,] array) { return 0; }


        /// <summary>
        /// Enter description for operator.
        /// ID string generated is "M:N.X.op_Addition(N.X,N.X)".
        /// </summary>
        /// <param name="x">Describe parameter.</param>
        /// <param name="xx">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static X operator +(X x, X xx) { return x; }


        /// <summary>
        /// Enter description for property.
        /// ID string generated is "P:N.X.prop".
        /// </summary>
        public int prop { get { return 1; } set { } }


        /// <summary>
        /// Enter description for event.
        /// ID string generated is "E:N.X.d".
        /// </summary>
        public event D d;


        /// <summary>
        /// Enter description for property.
        /// ID string generated is "P:N.X.Item(System.String)".
        /// </summary>
        /// <param name="s">Describe parameter.</param>
        /// <returns></returns>
        public int this[string s] { get { return 1; } }


        /// <summary>
        /// Enter description for class Nested.
        /// ID string generated is "T:N.X.Nested".
        /// </summary>
        public class Nested { }


        /// <summary>
        /// Enter description for delegate.
        /// ID string generated is "T:N.X.D".
        /// </summary>
        /// <param name="i">Describe parameter.</param>
        public delegate void D(int i);


        /// <summary>
        /// Enter description for operator.
        /// ID string generated is "M:N.X.op_Explicit(N.X)~System.Int32".
        /// </summary>
        /// <param name="x">Describe parameter.</param>
        /// <returns>Describe return value.</returns>
        public static explicit operator int(X x) { return 1; }

    }
}
```

See also

- C# Programming Guide
- /doc (C# Compiler Options)
- XML Documentation Comments

Delimiters for Documentation Tags (C# Programming Guide)

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

The use of XML doc comments requires delimiters, which indicate to the compiler where a documentation comment begins and ends. You can use the following kinds of delimiters with the XML documentation tags:

`///`
Single-line delimiter. This is the form that is shown in documentation examples and used by the Visual C# project templates. If there is a white space character following the delimiter, that character is not included in the XML output.

NOTE
The Visual Studio IDE has a feature called Smart Comment Editing that automatically inserts the <summary> and </summary> tags and moves your cursor within these tags after you type the `///` delimiter in the Code Editor. You can turn this feature on or off in the Options dialog box.

`/** */`
Multiline delimiters.

There are some formatting rules to follow when you use the `/** */` delimiters.

- On the line that contains the `/**` delimiter, if the remainder of the line is white space, the line is not processed for comments. If the first character after the `/**` delimiter is white space, that white space character is ignored and the rest of the line is processed. Otherwise, the entire text of the line after the `/**` delimiter is processed as part of the comment.
- On the line that contains the `*/` delimiter, if there is only white space up to the `*/` delimiter, that line is ignored. Otherwise, the text on the line up to the `*/` delimiter is processed as part of the comment, subject to the pattern-matching rules described in the following bullet.
- For the lines after the one that begins with the `/**` delimiter, the compiler looks for a common pattern at the beginning of each line. The pattern can consist of optional white space and an asterisk (`*`), followed by more optional white space. If the compiler finds a common pattern at the beginning of each line that does not begin with the `/**` delimiter or the `*/` delimiter, it ignores that pattern for each line.

The following examples illustrate these rules.

- The only part of the following comment that will be processed is the line that begins with `<summary>`. The three tag formats produce the same comments.

```
/** <summary>text</summary> */

/**
<summary>text</summary>
*/

/**
 * <summary>text</summary>
*/
```

- The compiler identifies a common pattern of " * " at the beginning of the second and third lines. The pattern is not included in the output.

```
/**
 * <summary>
 * text </summary>*/
```

- The compiler finds no common pattern in the following comment because the second character on the third line is not an asterisk. Therefore, all text on the second and third lines is processed as part of the comment.

```
/**
 * <summary>
   text </summary>
*/
```

- The compiler finds no pattern in the following comment for two reasons. First, the number of spaces before the asterisk is not consistent. Second, the fifth line begins with a tab, which does not match spaces. Therefore, all text from lines two through five is processed as part of the comment.

```
/**
  * <summary>
  * text
 *  text2
    *  </summary>
*/
```

See also

- C# Programming Guide
- XML Documentation Comments
- /doc (C# Compiler Options)
- XML Documentation Comments

How to: Use the XML documentation features

1/23/2019 • 4 minutes to read • Edit Online

The following sample provides a basic overview of a type that has been documented.

Example

```
// If compiling from the command line, compile with: -doc:YourFileName.xml

/// <summary>
/// Class level summary documentation goes here.
/// </summary>
/// <remarks>
/// Longer comments can be associated with a type or member through
/// the remarks tag.
/// </remarks>
public class TestClass : TestInterface
{
    /// <summary>
    /// Store for the Name property.
    /// </summary>
    private string _name = null;

    /// <summary>
    /// The class constructor.
    /// </summary>
    public TestClass()
    {
        // TODO: Add Constructor Logic here.
    }

    /// <summary>
    /// Name property.
    /// </summary>
    /// <value>
    /// A value tag is used to describe the property value.
    /// </value>
    public string Name
    {
        get
        {
            if (_name == null)
            {
                throw new System.Exception("Name is null");
            }
            return _name;
        }
    }

    /// <summary>
    /// Description for SomeMethod.
    /// </summary>
    /// <param name="s"> Parameter description for s goes here.</param>
    /// <seealso cref="System.String">
    /// You can use the cref attribute on any tag to reference a type or member
    /// and the compiler will check that the reference exists.
    /// </seealso>
    public void SomeMethod(string s)
    {
    }
    /// <summary>
    /// Some other method.
    /// </summary>
    /// <returns>
    /// Return values are described through the returns tag.
    /// </returns>
    /// <seealso cref="SomeMethod(string)">
    /// Notice the use of the cref attribute to reference a specific method.
    /// </seealso>
    public int SomeOtherMethod()
    {
        return 0;
    }

    public int InterfaceMethod(int n)
    {
        return n * n;
    }

    /// <summary>
    /// The entry point for the application.
    /// </summary>
    /// <param name="args"> A list of command line arguments.</param>
    static int Main(System.String[] args)
    {
        // TODO: Add code to start application here.
        return 0;
    }
}

/// <summary>
/// Documentation that describes the interface goes here.
/// </summary>
/// <remarks>
/// Details about the interface go here.
/// </remarks>
interface TestInterface
{
    /// <summary>
    /// Documentation that describes the method goes here.
    /// </summary>
    /// <param name="n">
    /// Parameter n requires an integer argument.
    /// </param>
    /// <returns>
    /// The method returns an integer.
    /// </returns>
    int InterfaceMethod(int n);
}
```

The example generates an .xml file with the following contents:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xmlsample</name>
    </assembly>
    <members>
        <member name="T:TestClass">
            <summary>
            Class level summary documentation goes here.
            </summary>
            <remarks>
            Longer comments can be associated with a type or member through
            the remarks tag.
            </remarks>
        </member>
```

```xml
        <member name="F:TestClass._name">
            <summary>
            Store for the Name property.
            </summary>
        </member>
        <member name="M:TestClass.#ctor">
            <summary>
            The class constructor.
            </summary>
        </member>
        <member name="P:TestClass.Name">
            <summary>
            Name property.
            </summary>
            <value>
            A value tag is used to describe the property value.
            </value>
        </member>
        <member name="M:TestClass.SomeMethod(System.String)">
            <summary>
            Description for SomeMethod.
            </summary>
            <param name="s"> Parameter description for s goes here.</param>
            <seealso cref="T:System.String">
            You can use the cref attribute on any tag to reference a type or member
            and the compiler will check that the reference exists.
            </seealso>
        </member>
        <member name="M:TestClass.SomeOtherMethod">
            <summary>
            Some other method.
            </summary>
            <returns>
            Return values are described through the returns tag.
            </returns>
            <seealso cref="M:TestClass.SomeMethod(System.String)">
            Notice the use of the cref attribute to reference a specific method.
            </seealso>
        </member>
        <member name="M:TestClass.Main(System.String[])">
            <summary>
            The entry point for the application.
            </summary>
            <param name="args"> A list of command line arguments.</param>
        </member>
        <member name="T:TestInterface">
            <summary>
            Documentation that describes the interface goes here.
            </summary>
            <remarks>
            Details about the interface go here.
            </remarks>
        </member>
        <member name="M:TestInterface.InterfaceMethod(System.Int32)">
            <summary>
            Documentation that describes the method goes here.
            </summary>
            <param name="n">
            Parameter n requires an integer argument.
            </param>
            <returns>
            The method returns an integer.
            </returns>
        </member>
    </members>
</doc>
```

Compiling the code

To compile the example, type the following command line:

```
csc XMLsample.cs /doc:XMLsample.xml
```

This command creates the XML file *XMLsample.xml*, which you can view in your browser or by using the TYPE command.

Robust programming

XML documentation starts with ///. When you create a new project, the wizards put some starter /// lines in for you. The processing of these comments has some restrictions:

- The documentation must be well-formed XML. If the XML is not well-formed, a warning is generated and the documentation file will contain a comment that says that an error was encountered.
- Developers are free to create their own set of tags. There is a recommended set of tags (see Recommended tags for documentation comments). Some of the recommended tags have special meanings:
  - The <param> tag is used to describe parameters. If used, the compiler verifies that the parameter exists and that all parameters are described in the documentation. If the verification failed, the compiler issues a warning.
  - The `cref` attribute can be attached to any tag to provide a reference to a code element. The compiler verifies that this code element exists. If the verification failed, the compiler issues a warning. The compiler respects any `using` statements when it looks for a type described in the `cref` attribute.
  - The <summary> tag is used by IntelliSense inside Visual Studio to display additional information about a type or member.

    NOTE

    The XML file does not provide full information about the type and members (for example, it does not contain any type information). To get full information about a type or member, the documentation file must be used together with reflection on the actual type or member.

See also

- C# Programming Guide
- /doc (C# Compiler Options)
- XML Documentation Comments