|  |  |
| --- | --- |
|  | .NET Bio Commenting Conventions:  How to Write Documentation Comments in Source Code  Version 1.0 - June 2011 |

Abstract

These guidelines describe how to write useful documentation comments when tools will be used to harvest comments to create reference pages.

This paper is intended for developers who need to write documentation comments, either to help programmer-writers create documentation or to create an API reference directly.

Contents

[Introduction: Why Use Document Comment Conventions? 3](#_Toc256774564)

[General Guidelines 4](#_Toc256774565)

[How to Write Documentation Comments 5](#_Toc256774566)

[Comment Guidelines for Reference Components 7](#_Toc256774567)

[Topic Description 7](#_Toc256774568)

[Types 7](#_Toc256774569)

[Parameter Descriptions 8](#_Toc256774570)

[Return Values 9](#_Toc256774571)

[Exceptions 10](#_Toc256774572)

[Remarks 10](#_Toc256774573)

[Examples 12](#_Toc256774574)

[See Also 12](#_Toc256774575)

[Appendix A: XML Comment Tags for .NET Documentation 13](#_Toc256774576)

**Disclaimer:** This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2011 The Outercurve Foundation.

Distributed under Creative Commons Attribution 3.0 Unported License.

Microsoft, Visual Studio, and Windows are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

# Introduction: Why Use Document Comment Conventions?

Comments have always been an important—though often neglected—aspect of programming. Originally, programmers used comments to document code for themselves and other programmers who might work in the file. Such comments were typically a variable combination of internal implementation details and content intended for users.

However, there was no formal way to distinguish between user-oriented and internal/personal comments. The only way to harvest user-oriented comments was for a writer to comb the files line by line and manually transfer any useful information to formal documentation.

To support documenting code, we now have:

* Formatting conventions that can distinguish documentation comments from implementation comments.
* Standardized tags that allow developers to identify different types of content, such as parameter descriptions or remarks.

Several tools can harvest the documentation comments:

* Some tools can automatically create a complete API reference for the project, with text derived from documentation comments.
* Generating an API reference directly from code comments has some limitations, but can be quite useful for certain types of project.
* Documentation comments provide an easy way for developers to provide information about how the API works to the writer who will produce the associated reference documentation.

In either case, the quality of the comments is important:

* For comment-based documentation, the text is drawn verbatim from the documentation comments, so comment quality completely controls the quality of the reference.
* For comments that will be handled by a writer, good comments help the writer to work more efficiently and reduce their need to ask clarifying questions.

This document discusses how to write good documentation comments. The advice is specifically directed to projects that plan to produce an API reference directly from code comments. The quality bar is high for such projects, because there is little opportunity to modify the text after it is harvested. However, writing comments to this standard is encouraged for any project.

Note: This document does not discuss the mechanics of writing code comments in detail, because each tool has its own conventions. You should refer to the tool’s documentation for details.

# General Guidelines

This section contains general guidelines for writing documentation comments. Subsequent sections provide guidelines for documenting different components of a reference page.

Don’t use documentation comments for internal/personal purposes.

Anything that you put in a documentation comment will be harvested by the tools. If you produce the reference directly from comments, the comments will be published verbatim. Putting internal information in documentation comments poses risks ranging from embarrassment to exposure of intellectual property to a variety of serious legal issues.

For .NET Framework documentation, only “///” comments are harvested, so put all internal or personal details in “//” comments.

IMPORTANT: Internal information should be placed only in standard comments that are not harvested by the documentation tools.

Choose your words carefully.

The key things to avoid are:

Embarrassing language

This category includes “naughty” words, humor (what’s funny in one culture can be offensive in another), rude comments (“Engineers of that other code are wienies”), or any other content that could cause embarrassment if published.

Geopolitical gaffes

This category includes words and phrases that are known to be problematic with certain countries.

Legally actionable content

This category includes, but is not limited to, intellectual property, trademark, and copyright infringement issues.

Security and privacy issues

Be careful about including any compromising content. Check your comments to ensure the following:

No security-related references exposed, such as your local \\server\share names.

No private information exposed, such as personal e-mail addresses.

Use standard English.

Write your comments in textbook English, including:

Use complete sentences, with appropriate spelling, grammar, punctuation, and capitalization.

Keep it simple and straightforward.

Avoid complex sentence structure and obscure words. Stick to simple declarative sentences, the shorter the better.

Don’t use slang, which specifically includes text-messaging abbreviations.

Stick to standard English plus well-established technical terminology.

Use jargon sparingly.

Technical jargon is normal and acceptable to a point, but avoid using terms that might meaningful to only a small group.

Ask yourself: would anyone outside your immediate team understand a particular term or phrase?

Document usage, not implementation.

The focus of your comments should be on how to use the API in an application.

Comment early and often.

Write your documentation comments while everything is fresh in your mind. Six months later, you will likely have a much harder time remembering details.

Update as needed.

If the API changes, don’t forget to modify the documentation comments, too.

Be complete.

Give users all the information they need. Don’t leave large “left as an exercise for the reader” gaps for the user to figure out.

Don’t assume too much of the user.

When you have been working on a project for considerable time, many issues will be so obvious to you that they won’t seem to be worth mentioning. Put yourself in the mindset of someone seeing this technology for the first time—many of those issues will not be obvious at all. In case of doubt, document it. Users rarely complain that the documentation is too thorough.

# How to Write Documentation Comments

This section walks through different components of a method reference page, and provides advice on how to write the corresponding code comments. Method topics are the chosen example, because they are the most complicated reference pages. Other page types contain a subset of the basic elements—or different elements that serve a similar purpose—so adapting these guidelines to other page types is straightforward.

The following figure shows an annotated skeleton method reference page. The syntax block and the various headings are automatically generated by the tools. The associated descriptions—the underlined text—come from code comments.

  
Figure 1. Annotated Skeleton method Reference Page

The two primary tools for creating an API reference from documentation comments are Sandcastle and DOxygen.

* Sandcastle uses .NET documentation tags, which have the following format:

///<tagname>Content</tagname>

where <tagname> is an XML tag that indicates the reference page component.

* See the DOxygen documentation for an explanation of the format and available tags.

Recommended Tags for Documentation Comments (C# Programming Guide)

<http://msdn.microsoft.com/en-us/library/5ast78ax.aspx>

DOxygen

<http://www.stack.nl/~dimitri/doxygen>

# Comment Guidelines for Reference Components

The following sections provide guidelines for documenting each section of a standard reference page. For convenience, each section includes the standard .NET tag for the associated reference page component.

The .NET Framework also includes number of additional tags that can be used for various specialized purposes. Appendix A provides a list of the available .NET tags and a brief usage description. For information about DOxygen tags, see the DOxygen documentation.

## Topic Description

<summary>

Required.

All topics must have a description that briefly describes how the API is used in an application. It should normally be limited a single sentence, or two at most. Put detailed discussions elsewhere, typically in Remarks. The topic descriptions for class members are often used as descriptions in the class reference page’s Members table.

### Example

The Multiply method performs element-wise multiplication on two one-dimensional integer arrays.

Fair Documentation Comment

///<summary>Multiplies two arrays</summary>

Good Documentation Comment

///<summary>This method multiplies each element of a one-dimensional

///integer array by the corresponding element of a second

///one-dimensional integer array.</summary>

The “Fair” comment is too terse and could be misinterpreted. “Array multiplication” could also be understood as referring to a scalar product, matrix-vector multiplication, and so on, which are quite different operations than element-wise multiplication of 1-D arrays.

The “Good” comment makes it clear that the method performs element-wise multiplication on a pair of one-dimensional arrays. It is also a complete, properly punctuated sentence.

## Types

<typeparam name=”***type\_name***”>

Required, for topics that use generics.

Describe the generic types.

### Example

Consider the following fictitious method, which takes a generic input and returns a generic output, not necessarily of the same type.

public static OutObject<K> MyMethod(InObject<T> inputData)

Fair Documentation Comment

///<typeparam name="T">A generic type</typeparam>

///<typeparam name="K">A generic type</typeparam>

Good Documentation Comment

///<typeparam name="T">The element type of the input object.</typeparam>

///<typeparam name="K">The element type of the return value.</typeparam>

This section often needs little explanation and can be omitted if the type information is obvious, especially if there is only a single type. The “Good” example indicates the purpose of each type, which can be helpful. Also consider:

* Using type names, such as TSource or TResult, that better indicate how the type is used.
* Mentioning what actual types these generics can correspond to, especially if the range of types is limited. However, Type Parameters descriptions should be brief, so such a discussion might be better handled in the Remarks section or in an associated conceptual document.

## Parameter Descriptions

<param name="param\_name">

Required, for methods with parameters.

Describe how each parameter is used. Parameter usage is often straightforward and requires only a brief sentence. Details that can be added to clarify usage include:

* Prefix the description with [IN], [OUT], or [IN/OUT], if the direction isn’t apparent. This is especially helpful for pointers, where it isn’t always obvious how they are used.
* Indicate whether indexes or other ordinals are zero-based or one-based.
* Indicate whether character counts are in bytes or characters.
* For parameters of indeterminate type—such as void\* or Object—describe what actual types are expected, or least give some general guidance.

### Example

Consider the following fictitious method, which takes an enumeration, a string, and an object of unspecified type as input. The inputObject object is, among other things, listed in the UI.

public void MyMethod( SomeEnum scheme,

string name,

object inputObject)

Fair Documentation Comment

///<param name="scheme">The scheme</param>

///<param name="name">The object name</param>

///<param name="index">The list index</param>

///<param name="inputObject">The input object</param>

Good Documentation Comment

///<param name="scheme">The data scheme used by the input object.

///This parameter is set to one of the values from the

///<see cref="MyNamespace.SomeEnum"/> enumeration.</param>

///<param name="name">The name of the input object. The user

///interface uses this name to identify the object.</param>

///<param name="index">The zero-based index of the inputObject

///object’s position in the list.</param>

///<param name="inputObject">The input object. This object can be

///any of the following types: InObject1, InObject2,

///or InObject3.</param>

The “Fair” descriptions tell you little more than what you could figure out from looking at the syntax block. The “Good” descriptions provide much more:

* The scheme description tells you what kind of scheme, what the scheme applies to, and provides a link to the enumeration that defines the available schemes. For short enumerations, you could simply put the values in a list or table.
* The name description tells you what the name is used for. Since this is fictitious, the description is a little vague. With a real UI, you could be more precise about how the UI uses the name.
* The index description tells you that the index is zero-based, something that the user might otherwise have to figure out by trial and error.
* The inputObject description tells you what kind of objects can be assigned to this parameter. That’s often not defined sufficiently that you can simply list the possible object types, but you can usually provide some general guidance or point to a conceptual document that explains the issue in detail.

## Return Values

<returns>

Required, for methods that return a value.

Describe what the method returns. This is often straightforward and requires only a brief sentence. Details that can be added to clarify usage include:

* If the method returns an enumerable set of values, such as a set of error or HRESULT values, list them.

If the method returns values from an enumeration, you can simply link to the enumeration, but if it’s only a few items, consider listing them for convenience. If the method returns only some of an enumeration’s values, indicate which ones.

* For return values of indeterminate type—such as void\* or object—describe what actual types are returned, or least the range of types.

### Example

Consider the following fictitious method, which takes an input object that represents a collection of data, and filters the data based on a specified scheme. It then returns the filtered data in an object of the same type as the input object.

public object MyMethod( SomeEnum scheme,

object inputObject)

Fair Documentation Comment

///<returns> Returns the filtered object.</returns>

Good Documentation Comment

///<returns> Returns the filtered object, which is the same type

///as the input object.</returns>

The “Fair” description doesn’t tell you that the two object types are the same. It’s likely that they are, but better to be explicit, as in the “Good” explanation.

Note that the “Good” explanation implicitly assumes that the filtering process is explained elsewhere in the topic, so that it’s apparent what is meant by “filtered object.”

## Exceptions

<exception cref="member">

Optional, but recommended.

List any exceptions that are method-specific or that the user should be prepared to handle routinely—for example, if the method throws an exception to indicate a common error condition.

## Remarks

<remarks>

Optional, but recommended.

Remarks is a free-form section that can be used for any content that doesn’t belong in the other sections, or requires more discussion that can be easily accommodated by the other sections. It can include such topics as:

* A detailed discussion of how to use the API.
* Issues that affect the API, but don’t belong in other sections.
* Quirks that the user should be aware of, and possible workarounds.
* Compatibility issues.
* Security issues.

A useful rule of thumb is that Remarks sections should be limited to a few paragraphs. If a Remarks section is significantly longer—or if the same content is used in multiple topics—consider putting the content in a conceptual document and linking to the document from the relevant reference topics.

Most topics will benefit from well-considered remarks. However, some API members are so straightforward and obvious that there really nothing to add. There is no need for a Remarks section in that case.

### Examples

The following are a few random examples of Remarks, with commentary on why they are or are not useful or appropriate.

Fair Example

///<remarks>Setting this property to true sets the ALLOWNULL bit in

///NULLFIELDS to enable nullability</remarks>

This remark discusses the internal workings of the property, not how to use the property in an application, so it isn’t of much value to a user.

Good Example

///<remarks>Setting this field to true allows all object references

///to have null values. If the AllowNullFields field is not set,

///applications must explicitly annotate those fields that can have

///null values by applying the [Field](#_Field_Attribute) attribute, as follows:

///<code>[Field(CanBeNull=true)]</code></remarks>

This remark:

Describes how the method works.

Describes an alternative that provides more granularity than the method itself.

Fair Example

///<remarks>This method normalizes the elements of an

///object</remarks>

Normalization can be implemented in a variety of ways, so this remark is too vague to give the user any real understanding of how the method works.

Good Example

///<remarks>This method normalizes the elements of an object by using ///the following formula:

///<code>f1' = f1a / sqrt(f1a\*\*2 + f2a\*\*2 + f3a\*\*2)

///f2' = f2a / sqrt(f1a\*\*2 + f2a\*\*2 + f3a\*\*2)

///f3' = f3a / sqrt(f1a\*\*2 + f2a\*\*2 + f3a\*\*2)

///f4' = f4a / sqrt(f1a\*\*2 + f2a\*\*2 + f3a\*\*2)</code>

///For example, assume that a represents the following array of

///Float4 structures:

///<code>{1, 2, 3, 4} {3 ,4 ,5 ,6} {5 ,6 ,7 ,8}</code>

///This method returns an object that represents the following array.

///For convenience, the values are rounded to two decimal places.

///<code>{0.27, 0.53, 0.80, 1.1} {0.42, 0.57, 0.71, 0.85}

///{0.48, 0.57, 0.67, 0.76}</code>

This remark is an example of where exposing internal details provides useful information to the reader about how the method works, without exposing details that might create problems later. This information could perhaps have been in the Return Values section, but it’s usually better to put lengthy discussions in Remarks.

## Examples

<example>

Optional.

Examples is a free-form section that can be used for any examples, including:

* Code examples.
* Command line examples.
* Data format examples, such as XML schemas.

Typically, the section consists of a descriptive sentence followed by one or more blocks of sample code, enclosed in <code> tags.

Examples should do more than just demonstrate correct syntax. Users generally prefer substantive examples that show how the API is used in a functional application. A recommended best practice is to source all example code from complete functional samples. “Freehanding” an example is not recommended; it is too easy to make unobvious mistakes. Sourcing your examples from working sample code ensures that the example contains valid code.

IMPORTANT: Users often cut and paste sample code, so make sure that examples compile and run correctly, and don’t forget to update examples if the method changes.

## See Also

<seealso>

Optional.

SeeAlso is a list of links to related information, usually to related API elements. Include any links that a user might find interesting or convenient, and aren’t linked elsewhere in the topic. You usually don’t need to include obvious things such as a link a method’s class page.

# Appendix A: XML Comment Tags for .NET Documentation

For ease of reference, the following table contains a list of commonly used .NET documentation XML comment tags, with a brief description. For details, see [“Recommended Tags for Documentation Comments (C# Programming Guide)”](http://msdn.microsoft.com/en-us/library/5ast78ax(VS.80).aspx) on MSDN.

|  |  |
| --- | --- |
| Tag | Description |
| **<c>** | Inline descriptive text, typically API names. The displayed text is bold.   * Inner text: The text. |
| **<code>** | A code block, typically in an **Example** section.   * Inner text: The code. |
| **<description>** | A description of a list item:   * Inner text: The description. |
| **<example>** | An **Example** section.   * Inner text: Example description. * Child elements: Usually contains a <code> block. |
| **<exception>** | An exception.   * cref: A reference to the exception class. * Inner text: The description. |
| **<item>** | An item in a list.   * Child elements: The item name and/or a description, as <term> and <description> elements, respectively. |
| **<list>** | A list.   * Child elements: The list items, as either <listheader> or <item> child elements. |
| **<para>** | A paragraph.   * Inner text: The paragraph text. |
| **<param>** | A parameter.   * name: The parameter name. * Inner text: A description. |
| **<paramref>** | An inline reference to a parameter.   * name: The parameter name. |
| **<remarks>** | A **Remarks** section.   * Inner text: The remarks. * Child elements: Can contain child elements such as <para> that format the remarks. |
| **<returns>** | A **Return Value** section.   * Inner text: A description. |
| **<see>** | An inline link to a member or field.   * cref: The member or field. |
| **<seealso>** | A link to a member or field for the **See Also** section.   * cref: The member or field. |
| **<summary>** | The topic description.   * Inner text: The description. |
| **<term>** | The name of an item in a list.   * Inner text: The name. |
| **<typeparam>** | A generic type:   * name: The type name. * Inner text: Description. |
| **<typeparamref>** | An inline reference to a generic type.   * name: The type name. |
| **<value>** | A **Value** section for a property topic.   * Inner text: A description. |