

# QUICK-START C#

## Documenting your code with XML

### Why XML?

*You can skip this. It's just the opening. Head to [Getting Started](#) for the real stuff.*

Some programming software contains tools that let us see a list of methods and variables that we can access, as well as what those methods and variables are used for. For those of us who are learning with software that uses C# there is a special type of commenting we can do to 'integrate' the ability to have notes and information attached to our methods. By using XML within a special type of C# comment above our member we can have that message display while we're writing a call to that member later.



**Members** in C# refer to any method, variable or property (as well as other types) in a Class or Struct.

<https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/members>

Instead of having to go back to that script later on whilst we're working on another game mechanic or tool, we can get that information as we're typing away; an invaluable addition to our collection of tools and tricks as programmers. *So...what's the catch?*

### The Catch

We've all been there; writing large blocks of comments to describe how a member in our class works. The catch with XML is we're still doing a large block of commenting, though a good chunk of it is XML's syntax which we won't see. These snippets also can be larger for methods and constructors if you are passing through parameters and want documentation specifically for each parameter. Finally, this tutorial is written with examples done in **Visual Studio Community 2019 (or later)**. If you're still on board with this, *let's go!*

## Getting Started

### Reference

```
/// <summary> (this is a start tag)
///
/// </summary> (this is an end tag)
private void Update() { /* ... */ }
```

Hovering over the method's name on the left will show this intellisense tip.

```
void OurObjectClass.Update()
Any text we add will appear here, under our class member...

Unity: Update is called every frame, if MonoBehaviour is enabled
```

Our first example uses the **Update()** method generated in a **MonoBehaviour**. You may be familiar with this, it is a staple of how we make games in Unity Engine.

The first thing we want to do to start using XML is by creating a new line above our Update Method, adding three forward-slashes to it. For Visual Studio, this will provide our first XML tags, **summary**, across three lines. Summary is just the main "body" of the tip for us to add our own text to. In between the first and last line that has appeared, we want to add our text.

*Adding **before** <summary> or **after** </summary> tags will result in the text not displaying in our intellisense tip.*

Right now, our intellisense is empty, so we'll add some flavour text in the tutorial. Make sure to add text relevant to what your class' member is for if you're using this in code for an assignment or project.

### Sample

```
/// <summary>
/// We're calling some methods in our game object here.
/// </summary>
private void Update() { /* ... */ }
```

```
void OurObjectClass.Update()
We're calling some methods in our game object here.

Unity: Update is called every frame, if MonoBehaviour is enabled
```

This will work for any member of a class or struct though there are a lot of extra bells and whistles that XML has. For our purposes the next few sections of this guide will contain everything that you would generally need or want to know around XML documentation.

**As for what can go wrong, there are four things you want to check for immediately if your changed intellisense tip doesn't appear:**



#### 1. You did not close a start tag.

*XML will break down and cry if you forget an end tag.*

```
/// <summary> custom C# method.
private void Method() { /* ... */ }
```

#### 2. The text is placed before a start tag or after an end tag.

*XML is partially blind and can't see outside of the tags.*

```
/// Our <summary> custom C# method. </summary>
private void Method() { /* ... */ }
```

#### 3. The start tag and end tag don't match.

*XML is (case) sensitive and will bite you.*

```
/// <summary> Our custom C# method. </Summary>
private void Method() { /* ... */ }
```

#### 4. Your XML start tags for two separate style options may end out of order.

*XML loves order. It will kick and scream until you listen.*

```
/// <summary>
/// <b><i>Our custom C# method.</b></i>
/// </summary>
private void Method() { /* ... */ }
```

*If you create a start tag just after another start tag, that tag **must** have an end tag first or it will not work.*

## Quick XML Reference

This section is not comprehensive to everything XML has to offer, just the core things that developers will most likely use throughout their projects. Microsoft has an extensive, detailed breakdown of every XML feature included with C# XML Documentation.

For more information see: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/xml/doc/recommended-tags>

### Empty Tags

In XML Empty Tags are start and end tags that have no content between them. An empty tag below is the same as doing the start and end tag without writing anything in between (where applicable). **You only need one empty tag in place of a start and end tag.**

For more information see: [https://www.tutorialspoint.com/xml/xml\\_tags.htm](https://www.tutorialspoint.com/xml/xml_tags.htm)

Additional Notice: The Intellisense View font is different due to Fira Mono lacking italic support.

#### Bold

##### Description:

Highlight text in the intellisense tip with **bold** font styling.

##### Start Tag

<b>


##### End Tag

</b>

##### Code

```
/// <summary>
/// Our custom <b>C# method</b>.
/// </summary>
private void Method() { /* ... */ }
```

##### Intellisense View

 void OurObjectClass.Method()  
Our custom **C# Method**.

#### Italic

##### Description:

Highlight text in the intellisense tip with *italic* font styling.

##### Start Tag

<i>

##### End Tag


</i>

##### Code

```
/// <summary>
/// Our custom <i>C# method</i>.
/// </summary>
private void Method() { /* ... */ }
```

##### Intellisense View

 void OurObjectClass.Method()  
Our custom *C# Method*.

 Visual Studio will try to auto-fill this as <inheriteddoc>. Simply press **ESC** to cancel the intellisense auto-fill and close the declaration with a greater-than symbol.

#### Line Break

##### Description:

Start a new text line in the intellisense tip.

##### Start Tag

<br>

##### End Tag

</br>


##### Empty Tag

<br/>

##### Code

```
/// <summary>
/// Our custom <br/> C# method.
/// </summary>
private void Method() { /* ... */ }
```

##### Intellisense View

 void OurObjectClass.Method()  
Our custom  
C# Method.

#### Parameter Definitions

##### Description:

Create a parameter definition that appears when developers hover over the method's parameter declaration or when they are providing an object while writing a call to the method.

##### Code

```
/// <summary>
/// Our custom C# method with parameters.
/// </summary>
/// <param name="objParameter">Our custom method's parameter.</param>
private void Method(object objParameter)
{
    // ...
}
```


##### Start Tag

<param name="(parameter name here)">


##### End Tag

</param>

##### Intellisense View - Method

 void OurObjectClass.Method()  
Our custom C# Method with parameters.

##### Intellisense View - Parameter

 (parameter) object objParameter  
Our custom method's parameter.

## Returns

### Description:

Create a sub-definition that appears under the method's definition for the object(s) the method returns.

### Code

```
/// <summary> Our custom C# method with parameters. </summary>
/// <param name="objParameter">Our custom method's parameter.</param>
/// <returns> The object we just put in. </summary>
private OurCustomType Method(OurCustomType objParameter)
{
    return objParameter; // ...
}
```


### Start Tag

<returns>

### End Tag

</returns>

### Intellisense View

 **OurCustomType** **OurObjectClass**.Method()  
Our custom C# Method with parameters .

Returns:  
The object we just put in.

## See - C# Reference

### Description:

Create an in-text link that will redirect the developer to a C# Class, Type or Struct.

### Start Tag

<see cref="{object-type here}">

### End Tag

</see>


### Empty Tag

<see cref="{object-type here}"/>

### Code (Start-End Tag Variant)

```
/// <summary>
/// Our custom C# method. <br/>
/// Takes an <see cref="OurCustomType">Our Custom Type</see> object.
/// </summary>
/// <param name="objParameter">Our custom method's parameter.</param>
private void Method(OurCustomType objParameter) { /* ... */ }
```


### Intellisense View - Start-End Tag Variant

 **OurCustomType** **OurObjectClass**.Method()  
Our custom C# Method.  
Takes an **Our Custom Type** object .

### Code (Empty Tag Variant)

```
/// <summary>
/// Our custom C# method. <br/>
/// Takes an <see cref="OurCustomType"/> object.
/// </summary>
/// <param name="objParameter">Our custom method's parameter.</param>
private void Method(OurCustomType objParameter) { /* ... */ }
```

### Intellisense View - Empty Tag Variant

 **OurCustomType** **OurObjectClass**.Method()  
Our custom C# Method.  
Takes an **OurCustomType** object .

## See - Hyperlink Reference

### Description:

Create an in-text link that will redirect the developer to an external resource. **This will open a page in their browser.**

### Start Tag

<see href="{object-type here}">

### End Tag

</see>


### Empty Tag

<see href="{object-type here}"/>

### Code (Start-End Tag Variant)

```
/// <summary>
/// Our custom C# method. <br/>
/// Read more at <see href="insert-hyperlink-here">Our Scripting Website</see>.
/// </summary>
/// <param name="objParameter">Our custom method's parameter.</param>
private void Method(OurCustomType objParameter) { /* ... */ }
```


### Intellisense View - Start-End Tag Variant

 **OurCustomType** **OurObjectClass**.Method()  
Our custom C# Method.  
Read more at: [Our Scripting Website](#) .

### Code (Empty Tag Variant)


```
/// <summary>
/// Our custom C# method. <br/>
/// Read more at <see href="insert-hyperlink-here"/>.
/// </summary>
/// <param name="objParameter">Our custom method's parameter.</param>
private void Method(OurCustomType objParameter) { /* ... */ }
```

### Intellisense View - Empty Tag Variant

 **OurCustomType** **OurObjectClass**.Method()  
Our custom C# Method.  
Read more at: [\[ full hyperlink will appear here \]](#) .

One final thing, have you ever wondered how you could do what **Unity Technologies** does with MonoBehaviour methods in intellisense? How could they get “Unity” to display as if it were a C# keyword?

```
public void OurObjectClass.Update()  
Any text we add will appear here, under our class member...
```

 **Unity:** Update is called every frame, if MonoBehaviour is enabled

That’s where this final variation of “see” comes in. It highlights a word (or any text provided) as if it were a C# language keyword. It’s meant to be used for highlighting C# keywords specifically but we can use it to highlight whatever we want and it’s affected by font styling tags too.

## See - Language Word

### Description:

Create an in-text highlight for a language keyword.  
**Alternatively, highlight your own keyword.**


### Empty Tag

`<see langword="{your-word-here}"/>`

### Code (Empty Tag Variant)

```
/// <summary>  
/// Our custom C# method. <br/><br/>  
/// <see langword="TODO:"/> Add method functionality.  
/// </summary>  
/// <param name="objParameter">Our custom method's parameter.</param>  
private void Method(OurCustomType objParameter) { /* ... */ }
```

### Intellisense View - Empty Tag Variant

 **OurObjectClass.Method()**  
Our custom C# Method.  
  
**TODO:** Add method functionality.

## Final Tips

Using XML can make your scripts a bit harder to scroll through. It’s a trade-off you take for always-accessible documentation that you can find whenever you type out the class or member name elsewhere in your project. Having these comments can be extremely handy and could make the difference between remembering what you wrote and not. These are quick tips you can use to help quickly get up and running with C# XML Documentation.

### 1. Keep it simple.

*You don’t need to include what each parameter does within the main summary body. You also don’t need to include what each part of the code inside a method is doing, that’s what comments are for. **All you should aim to have is a concise, direct description of what the class/struct member is for.***

### 2. Don’t repeat yourself.

*It is entirely possible for you to accidentally repeat the exact same XML Documentation twice. Sometimes this can just be a case of you have the same method in multiple places and you could really do with restructuring your program.*

*Most often though, you’ve inherited a class and started to write XML for a member override. **XML documentation also applies to the inheriting class’ overriding members.** You should only write over your documentation for an overriding member if it is doing something that mostly or entirely replaces the core functionality of the original class’ member.*

### 3. Don’t rely on XML.

*That’s the most contradictory point of this document, but it makes sense. XML is for long-term use and it’s for when you work on projects that have tens of scripts which communicate with each other. It should not be used as a primary tool for documenting what you learnt, **only what the thing you’ve made does.***

*You can also tack on **TODO** lists but that’s probably going to be forgotten about unless you’re actively using the class/struct member heavily as you work on other mechanics.*

### 4. Experiment, experiment, experiment.

*This document is a basic introduction that covers the essential curios of the ability to integrate XML documentation Into C#. A lot of this information wasn’t found just through research and reading; it was trial and error too.*

*For an example, the way you get a less-than or greater-than symbol in XML documentation is through **XML Character Entities**. For a less-than symbol, you add [ &lt; ]. For a greater-than symbol, you add [ &gt; ].*

For more information see: [https://www.tutorialspoint.com/xml/xml\\_character\\_entities.htm](https://www.tutorialspoint.com/xml/xml_character_entities.htm)

XML is just one of the many tools in our arsenal as developers. It’s immensely useful when applied properly and can really bring a whole new level to the way you develop, changing your pipeline to be more efficient in the process. Don’t replace comments wholesale though!