



Formatting best practices for C# scripting in Unity

While there might not be one right way to format your C# code, agreeing on a consistent style across your team can result in a cleaner, more readable and scalable codebase. This page offers tips and key considerations to keep in mind for your classes, methods, and comments when creating your own style guide.

Note: The recommendations shared here are based on those provided by Microsoft. The best code style guide rules are the ones that work for your team's needs.

You can find a code style guide example [here](#) or download the full e-book, [Create a C# style guide: Write cleaner code that scales](#).

Formatting your code

Properties

Expression-bodied properties

Auto-implemented property

Serialization

Brace or indentation style

Decide on a uniform indentation

Don't omit braces

Keep braces for clarity in multiline statements

Standardize switch statements

Formatting your code

Along with naming, formatting helps reduce guesswork and improves code clarity. By following a standardized style guide, code reviews become less about how the code looks and more about what it does.

Aim to personalize the way your team will format the code. Consider each of the following code formatting suggestions when setting up your Unity style guide. You can choose to omit, expand, or modify these sample rules to fit your team's needs.

In all cases, reflect on how your team will implement each formatting rule, and then have everyone apply it uniformly. Refer back to your team's style guide to resolve any discrepancies. The less you think about formatting, the more productively – and creatively – you can work.

Let's take a look at some formatting guidelines.

Properties

A property provides a flexible mechanism to read, write, or compute class values. Properties behave as if they're public member variables, but in fact they are special methods called **accessors**.

Each property has a get and set method to access a private field, called a **backing field**. This way, the property **encapsulates the data**, hiding it from unwanted changes by the user or external objects. The "getter" and "setter" each have their own access modifier, allowing your property to be **read-write**, **read-only**, or **write-only**.

You can also use the accessors to validate or convert the data (e.g., to verify that the data fits your preferred format or changes a value to a particular unit).

The syntax for properties can vary, so your style guide should define how to format them. See the following examples for tips on how to keep properties consistent in your code.

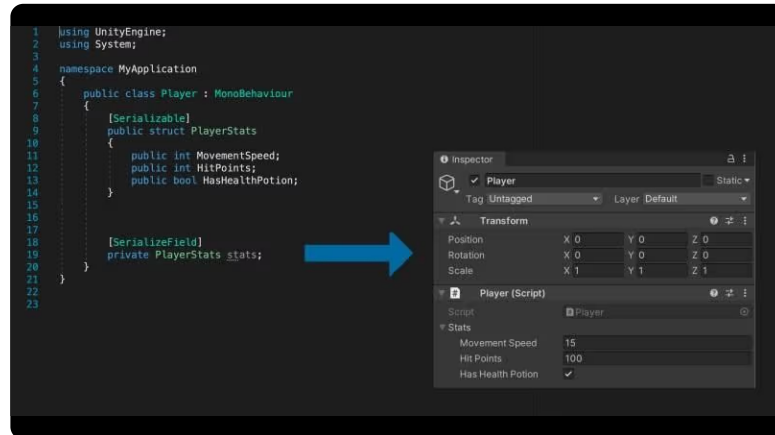
Expression-bodied properties

private backing field.

Auto-implemented property

Everything else uses the **expression-bodied { get; set; }** syntax: If you just want to expose a public property without specifying a backing field, use the **auto-implemented property**.

Apply the expression-bodied syntax for the set and get accessors. Remember to make the "setter" private if you don't want to provide write access. Align the closing with the opening brace for multiline code blocks.



A SERIALIZABLE CLASS OR STRUCT CAN HELP ORGANIZE THE INSPECTOR.

Serialization

Script serialization is the automatic process of transforming data structures or object states into a format that Unity can store and reconstruct later. For performance reasons, Unity handles serialization differently than in other programming environments.

Serialized fields appear in the **Inspector**, but you cannot serialize static, constant, or read-only fields. They must be either public or tagged with the **[SerializeField]** attribute. Unity only serializes certain field types, so refer to the [documentation](#) for the complete set of serialization rules.

Observe these basic guidelines when working with serialized fields:

Use the [SerializeField] attribute: The `SerializeField` attribute can work with private or protected variables to make them appear in the Inspector. This encapsulates the data better than marking the variable public, and prevents an external object from overwriting its values.

Use the Range attribute to set minimum and maximum values: The `[Range(min, max)]` attribute is handy if you want to limit what the user can assign to a numeric field. It also conveniently represents the field as a slider in the Inspector.

Group data in serializable classes or structs to clean up the Inspector: Define a public class or struct and mark it with the `[Serializable]` attribute. Define public variables for each type you want to expose in the Inspector.

Reference this serializable class from another class. The resulting variables appear within collapsible units in the Inspector.

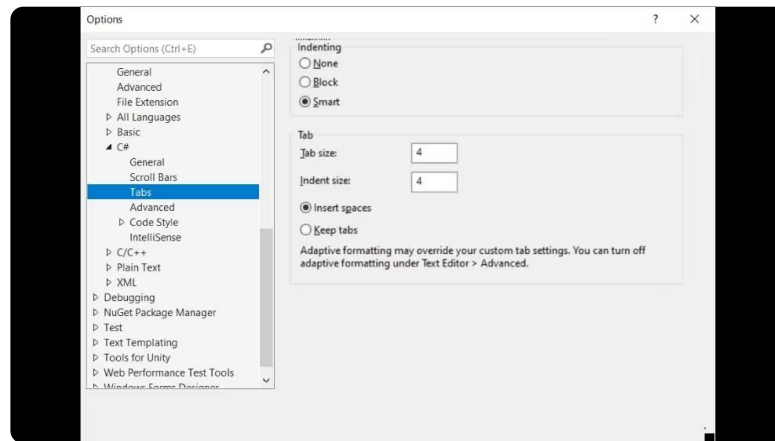
Brace or indentation style

There are two common indentation styles in C#:

The Allman style, also known as the BSD style (from BSD Unix), places the opening curly braces on a new line.

The K&R style, or “one true brace style,” keeps the opening brace on the same line as the previous header.

There are variations on these indentation styles as well. The examples in this guide use the Allman style from the Microsoft framework design guidelines. Regardless of which one you choose as a team, make sure that everyone follows the same indentation and brace style. You can also try the tips in the following sections.



TABS SETTINGS IN VISUAL STUDIO FOR WINDOWS

Decide on a uniform indentation

The indentation is typically two or four spaces. Get everyone on your team to agree on a setting in your Editor preferences without igniting a tabs versus spaces flame war.

In Visual Studio for **Windows**, navigate to **Tools > Options > Text Editor > C# > Tabs**.

In Visual Studio for **Mac**, navigate to **Preferences > Source Code > C# Source Code**. Select the **Text Style** to adjust the settings.

Note: Visual Studio provides the option to convert tabs to spaces.

Don't omit braces

Don't omit braces – not even for single-line statements. Braces increase consistency, which makes your code easier to read and maintain. In this example, the braces clearly separate the action, `DoSomething`, from the loop.

If you need to add a Debug line or run `DoSomethingElse` later on, the braces will already be in place. Keeping the clause on a separate line enables you to simply add a breakpoint.

Keep braces for clarity in multiline statements

Don't remove braces from nested multiline statements. Removing braces in this case won't throw an error, but will likely cause confusion. Apply braces for clarity, even if they're optional.

Standardize switch statements

Formatting can vary, so document your team preferences in your style guide and standardize your **switch** statements accordingly.

Here is one example of indenting the case statements.

Horizontal spacing
Add spaces
Spacing after a comma
No spacing after parenthesis
No space between a function and parenthesis
Avoid spaces inside brackets
Spacing before flow control conditions
Spacing with comparison operators
Readability tips
Vertical spacing and regions
Code formatting in Visual Studio

Horizontal spacing

Something as simple as spacing can enhance your code's appearance onscreen. While personal formatting preferences can vary, consider the suggestions below to improve readability.

Add spaces

Add spaces to decrease code density. The extra whitespace gives the sense of visual separation between parts of a line.

Spacing after a comma

Use a single space after a comma between function arguments.

No spacing after parenthesis

Don't add a space after the parenthesis and function arguments.

No space between a function and parenthesis

Don't use spaces between a function name and parenthesis.

Avoid spaces inside brackets

As much as possible, avoid spaces inside brackets.

Spacing before flow control conditions

Use a single space before flow control conditions and add a space between the flow comparison operator and parentheses.

Spacing with comparison operators

Use a single space before and after comparison operators.

Readability tips

Keep lines short and consider horizontal whitespace. Decide on a standard line width (80–120 characters), and break a long line into smaller statements rather than letting it overflow.

As discussed earlier, try to maintain indentation/hierarchy. Indenting your code can increase legibility.

Don't use column alignment unless it's required for readability. While this type of spacing aligns the variables, it can complicate pairing the type with the name.

Column alignment, however, can be useful for bitwise expressions or structs with a lot of data. Just be aware that it might create more work for you to maintain the column alignment as you add more items. Some auto-formatters might also change which part of the column gets aligned.

Vertical spacing and regions

You can use vertical spacing to your advantage as well. Keep related parts of the script together and use blank lines to your advantage. Try the following to organize your code from top to bottom:

- Group dependent or similar methods together: Code needs to be logical and coherent. Keep methods that do the same thing next to one another, so that someone reading your logic doesn't have to jump around the file.
- Use the vertical whitespace to separate distinct parts of your class: For example, you can add two blank lines between:
 - Variable declarations and methods
 - Classes and interfaces
 - if-then-else blocks (if it helps readability)

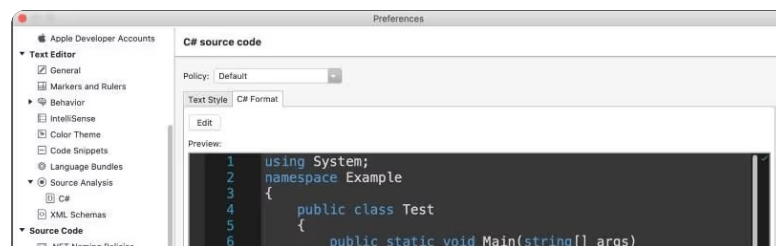
Keep this to a minimum, and if possible, track it in your style guide.

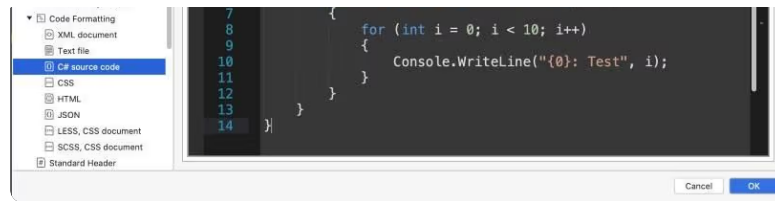
Using regions in your code

The `#region` directive enables you to collapse and hide sections of code in `C#` files, making large files more manageable and easier to read.

However, if you follow the general advice for classes from this guide, your class size should be manageable and the `#region` directive superfluous. Break your code into smaller classes instead of hiding code blocks behind regions. You will be less inclined to add a region if the source file is short.

Note: Many developers consider regions to be code smells or anti-patterns. Decide as a team on which side of the debate you fall.





THE PREVIEW WINDOW SHOWS OFF YOUR STYLE GUIDE CHOICES.

Code formatting in Visual Studio

Don't despair if these formatting rules seem overwhelming. Modern IDEs make it efficient to set up and enforce them. You can create a template of formatting rules and then convert your project files at once.

To set up formatting rules for the script Editor:

- In Visual Studio for **Windows**, navigate to **Tools > Options**, then locate **Text Editor > C# > Code Style Formatting**.
 - Use the settings to modify the **General**, **Indentation**, **New Lines**, **Spacing**, and **Wrapping** options.
- In Visual Studio for **Mac**, select **Visual Studio > Preferences**, then navigate to **Source Code > Code Formatting > C# Source Code**.
 - Select the **Policy** at the top, then go to the **Text Style** tab. In the **C# Format** tab, adjust the **Indentation**, **New Lines**, **Spacing**, and **Wrapping** settings.

To force your script file to conform to the style guide:

- In Visual Studio for **Windows**, go to **Edit > Advanced > Format Document** (Ctrl + K, Ctrl + D hotkey chord). If you want to only format whitespaces and tab alignment, you can also use **Run Code Cleanup** (Ctrl + K, Ctrl + E) at the bottom of the Editor.
- In Visual Studio for **Mac**, go to **Edit > Format Document** (Ctrl + I hotkey).

On Windows, you can also share your Editor settings from **Tools > Import and Export Settings**.

Export a file with the style guide's C# code formatting and then have every team member import that file.

Visual Studio helps you adhere to the style guide. Formatting then becomes as simple as using a hotkey.

Note: You can configure an [EditorConfig](#) file (see above) instead of importing and exporting Visual Studio settings. Doing this facilitates sharing formatting across different IDEs. It also has the added benefit of working with version control. See the [.NET code style rule options](#) for more information.

Though this isn't specific to clean code, be sure to check out the [10 ways to speed up your programming workflow](#) in [Unity with Visual Studio](#). Remember that it is more convenient to format and refactor clean code if you apply these productivity tips.

Get more code style tips

Learn more about naming conventions [here](#) or check out the [full e-book](#). You can also explore our detailed [code style guide example](#).

Language

English Deutsch 日本語 Français Português 中文 Español Русский 한국어

Social

[YouTube](#) [LinkedIn](#) [Twitter](#) [Facebook](#) [Instagram](#) [TikTok](#) [Discord](#)

Currency

EUR

Purchase

Education

Download


Unity Labs
Labs
Publications

Resources

Unity

Products	Students	Unity Hub	Learn platform	Our Company
Unity Ads	Educators	Download Archive	Community	Newsletter
Subscription	Institutions	Beta Program	Documentation	Blog
Unity Asset Store	Certification		Unity QA	Events
Resellers	Learn		FAQ	Careers
	Center of Excellence		Services Status	Help
			Case Studies	Press
			Made with Unity	Partners
				Investors
				Affiliates
				Security
				Social Impact
				Inclusion & Diversity
				Contact us

Copyright © 2025 Unity Technologies

[Legal](#) [Privacy Policy](#) [Cookies](#) [Do Not Sell or Share My Personal Information](#)  [Your Privacy Choices \(Cookie Settings\)](#)

"Unity", Unity logos, and other Unity trademarks are trademarks or registered trademarks of Unity Technologies or its affiliates in the U.S. and elsewhere ([more info here](#)). Other names or brands are trademarks of their respective owners.