Godot C# support

C# basics

C# features

C# API differences to GDScript

C# style guide

C# basics¶

Introduction¶

Warning

C# support is a new feature available since Godot 3.0. As such, you may still run into some issues, or find spots where the documentation could be improved. Please report issues with C# in Godot on the engine GitHub page, and any documentation issues on the documentation GitHub page.
This page provides a brief introduction to C#, both what it is and how to use it in Godot. Afterwards, you may want to look at how to use specific features, read about the differences between the C# and the GDScript API and (re)visit the Scripting section of the step-by-step tutorial.

C# is a high-level programming language developed by Microsoft. In Godot, it is implemented with the Mono 6.x .NET framework, including full support for C# 8.0. Mono is an open source implementation of Microsoft's .NET Framework based on the ECMA standards for C# and the Common Language Runtime. A good starting point for checking its capabilities is the Compatibility page in the Mono documentation.

Note

This is **not** a full-scale tutorial on the C# language as a whole. If you aren't already familiar with its syntax or features, see the Microsoft C# guide or look for a suitable introduction elsewhere.

Setting up C# for Godot¶

Prerequisites¶

Install the latest stable version of the .NET SDK, previously known as the .NET Core SDK.

From Godot 3.2.3 onwards, installing Mono SDK is not a requirement anymore, except it is required if you are building the engine from source.

Godot bundles the parts of Mono needed to run already compiled games. However, Godot does not bundle the tools required to build and compile games, such as MSBuild and the C# compiler. These are included in the .NET SDK, which needs to be installed separately.

In summary, you must have installed the .NET SDK **and** the Mono-enabled version of Godot.

Additional notes¶

Be sure to install the 64-bit version of the SDK(s) if you are using the 64-bit version of Godot.

If you are building Godot from source, install the latest stable version of Mono, and make sure to follow the steps to enable Mono support in your build as outlined in the Compiling with Mono page.

Configuring an external editor¶

C# support in Godot's built-in script editor is minimal. Consider using an external IDE or editor, such as Visual Studio Code or MonoDevelop. These provide autocompletion, debugging, and other useful features for C#. To select an external editor in Godot, click on **Editor → Editor Settings** and scroll down to **Mono**. Under **Mono**, click on **Editor**, and select your external editor of choice. Godot currently supports the following external editors:

- Visual Studio 2019
- Visual Studio Code
- MonoDevelop
- Visual Studio for Mac
- JetBrains Rider

See the following sections for how to configure an external editor:

JetBrains Rider¶

After reading the "Prerequisites" section, you can download and install JetBrains Rider.

In Godot's **Editor → Editor Settings** menu:

- Set **Mono -> Editor -> External Editor** to **JetBrains Rider**.
- Set **Mono -> Builds -> Build Tool** to **dotnet CLI**.

In Rider:

- Set **MSBuild version** to **.NET Core**.
- Install the **Godot support** plugin.

Visual Studio Code¶

After reading the "Prerequisites" section, you can download and install Visual Studio Code (aka VS Code).

In Godot's **Editor** → **Editor Settings** menu:

- Set **Mono** -> **Editor** -> **External Editor** to **Visual Studio Code**.

In Visual Studio Code:

- Install the C# extension.
- Install the Mono Debug extension.
- Install the C# Tools for Godot extension.

Note

If you are using Linux you need to install the Mono SDK for the C# tools plugin to work.
To configure a project for debugging open the Godot project folder in VS Code. Go to the Run tab and click on **Add Configuration...**. Select **C# Godot** from the dropdown menu. Open the `tasks.json` and `launch.json` files that were created. Change the executable setting in `launch.json` and command settings in `tasks.json` to your Godot executable path. Now, when you start the debugger in VS Code, your Godot project will run.

Visual Studio (Windows only)¶

Download and install the latest version of Visual Studio. Visual Studio will include the required SDKs if you have the correct workloads selected, so you don't need to manually install the things listed in the "Prerequisites" section.

While installing Visual Studio, select these workloads:

- Mobile development with .NET
- .NET Core cross-platform development

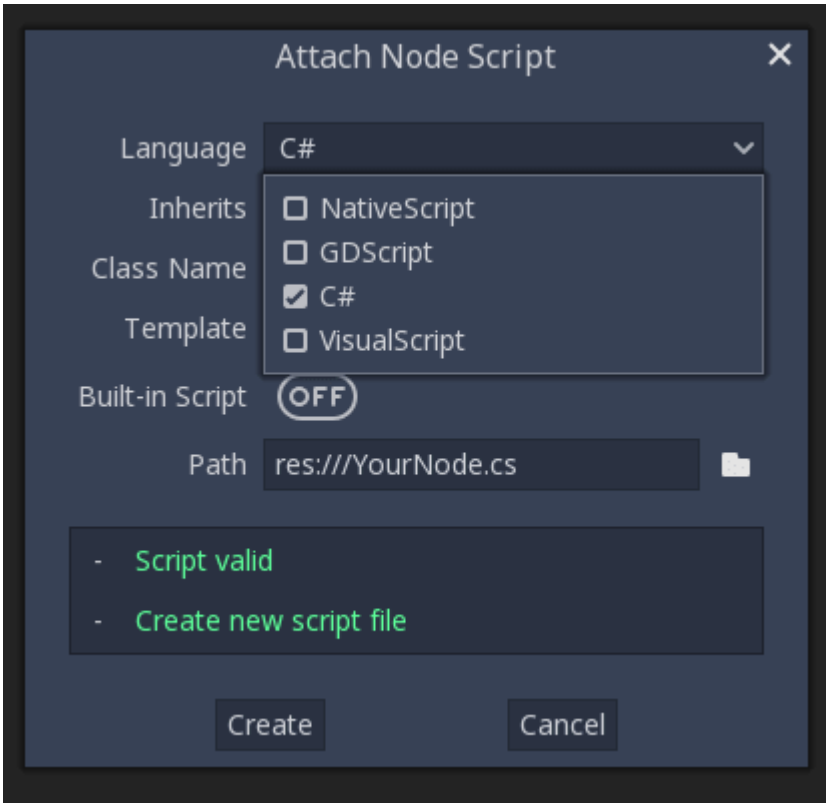
In Godot's **Editor** → **Editor Settings** menu:

- Set **Mono** -> **Editor** -> **External Editor** to **Visual Studio**.

Next, you need to download the Godot Visual Studio extension from github here. Double click on the downloaded file and follow the installation process.

Creating a C# script¶

After you successfully set up C# for Godot, you should see the following option when selecting **Attach Script** in the context menu of a node in your scene:



Note that while some specifics change, most concepts work the same when using C# for scripting. If you're new to Godot, you may want to follow the tutorials on Scripting languages at this point. While some places in the documentation still lack C# examples, most concepts can be transferred easily from GDScript.

Project setup and workflow¶

When you create the first C# script, Godot initializes the C# project files for your Godot project. This includes generating a C# solution (`.sln`) and a project file (`.csproj`), as well as some utility files and folders (`.mono` and `Properties/AssemblyInfo.cs`). All of these but `.mono` are important and should be committed to your version control system. `.mono` can be safely added to the ignore list of your VCS. When troubleshooting, it can sometimes help to delete the `.mono` folder and let it regenerate.

Example¶

Here's a blank C# script with some comments to demonstrate how it works.

```
using Godot;
using System;

public class YourCustomClass : Node
{
    // Member variables here, example:
    private int a = 2;
    private string b = "textvar";

    public override void _Ready()
    {
        // Called every time the node is added to the scene.
        // Initialization here.
        GD.Print("Hello from C# to Godot :)");
    }

    public override void _Process(float delta)
    {
        // Called every frame. Delta is time since the last frame.
        // Update game logic here.
    }
}
```

As you can see, functions normally in global scope in GDScript like Godot's `print` function are available in the `GD` class which is part of the `Godot` namespace. For a list of methods in the `GD` class, see the class reference pages for @GDScript and @GlobalScope.

Note

Keep in mind that the class you wish to attach to your node should have the same name as the `.cs` file. Otherwise, you will get the following error and won't be able to run the scene: *"Cannot find class XXX for script res://XXX.cs"*

General differences between C# and GDScript¶

The C# API uses `PascalCase` instead of `snake_case` in GDScript/C++. Where possible, fields and getters/setters have been converted to properties. In general, the C# Godot API strives to be as idiomatic as is reasonably possible.

For more information, see the C# API differences to GDScript page.

Warning

You need to (re)build the project assemblies whenever you want to see new exported variables or signals in the editor. This build can be manually triggered by clicking the word **Build** in the top right corner of the editor. You can also click **Mono** at the bottom of the editor window to reveal the Mono panel, then click the **Build Project** button.

You will also need to rebuild the project assemblies to apply changes in "tool" scripts.

Current gotchas and known issues¶

As C# support is quite new in Godot, there are some growing pains and things that need to be ironed out. Below is a list of the most important issues you should be aware of when diving into C# in Godot, but if in doubt, also take a look over the official issue tracker for Mono issues.

- Writing editor plugins is possible, but it is currently quite convoluted.
- State is currently not saved and restored when hot-reloading, with the exception of exported variables.
- Attached C# scripts should refer to a class that has a class name that matches the file name.
- There are some methods such as `Get()`/`Set()`, `Call()`/`CallDeferred()` and signal connection method `Connect()` that rely on Godot's `snake_case` API naming conventions. So when using e.g. `CallDeferred("AddChild")`, `AddChild` will not work because the API is expecting the original `snake_case` version `add_child`. However, you can use any custom properties or methods without this limitation.

Exporting Mono projects is supported for desktop platforms (Linux, Windows and macOS), Android, HTML5, and iOS. The only platform not supported yet is UWP.

Performance of C# in Godot¶

According to some preliminary benchmarks, the performance of C# in Godot — while generally in the same order of magnitude — is roughly **~4×** that of GDScript in some naive cases. C++ is still a little faster; the specifics are going to vary according to your use case. GDScript is likely fast enough for most general scripting workloads. C# is faster, but requires some expensive marshalling when talking to Godot.

Using NuGet packages in Godot¶

NuGet packages can be installed and used with Godot, as with any C# project. Many IDEs are able to add packages directly. They can also be added manually by adding the package reference in the `.csproj` file located in the project root:

```
    <ItemGroup>
        <PackageReference Include="Newtonsoft.Json" Version="11.0.2" />
    </ItemGroup>
    ...
</Project>
```

As of Godot 3.2.3, Godot automatically downloads and sets up newly added NuGet packages the next time it builds the project.

Profiling your C# code¶

- Mono log profiler is available for Linux and macOS. Due to a Mono change, it does not work on Windows currently.
- External Mono profiler like JetBrains dotTrace can be used as described here.

C# features¶

This page provides an overview of the commonly used features of both C# and Godot and how they are used together.

Type conversion and casting¶

C# is a statically typed language. Therefore, you can't do the following:

```
var mySprite = GetNode("MySprite");
mySprite.SetFrame(0);
```

The method `GetNode()` returns a `Node` instance. You must explicitly convert it to the desired derived type, `Sprite` in this case.

For this, you have various options in C#.

Casting and Type Checking

Throws `InvalidCastException` if the returned node cannot be cast to Sprite. You would use it instead of the `as` operator if you are pretty sure it won't fail.

```
Sprite mySprite = (Sprite)GetNode("MySprite");
mySprite.SetFrame(0);
```

Using the AS operator

The `as` operator returns `null` if the node cannot be cast to Sprite, and for that reason, it cannot be used with value types.

```
Sprite mySprite = GetNode("MySprite") as Sprite;
// Only call SetFrame() if mySprite is not null
mySprite?.SetFrame(0);
```

Using the generic methods

Generic methods are also provided to make this type conversion transparent.

`GetNode<T>()` casts the node before returning it. It will throw an `InvalidCastException` if the node cannot be cast to the desired type.

```
Sprite mySprite = GetNode<Sprite>("MySprite");
mySprite.SetFrame(0);
```

`GetNodeOrNull<T>()` uses the `as` operator and will return `null` if the node cannot be cast to the desired type.

```
Sprite mySprite = GetNodeOrNull<Sprite>("MySprite");
// Only call SetFrame() if mySprite is not null
mySprite?.SetFrame(0);
```

Type checking using the IS operator

To check if the node can be cast to Sprite, you can use the `is` operator. The `is` operator returns false if the node cannot be cast to Sprite, otherwise it returns true.

```
if (GetNode("MySprite") is Sprite)
{
    // Yup, it's a sprite!
}
```

For more advanced type checking, you can look into Pattern Matching.

C# signals¶

For a complete C# example, see the **Handling a signal** section in the step by step Scripting languages tutorial.

Declaring a signal in C# is done with the `[Signal]` attribute on a delegate.

```
[Signal]
delegate void MySignal();

[Signal]
delegate void MySignalWithArguments(string foo, int bar);
```

These signals can then be connected either in the editor or from code with `Connect`. If you want to connect a signal in the editor, you need to (re)build the project assemblies to see the new signal. This build can be manually triggered by clicking the “Build” button at the top right corner of the editor window.

```
public void MyCallback()
```

```
{
    GD.Print("My callback!");
}

public void MyCallbackWithArguments(string foo, int bar)
{
    GD.Print("My callback with: ", foo, " and ", bar, "!");
}

public void SomeFunction()
{
    instance.Connect("MySignal", this, "MyCallback");
    instance.Connect(nameof(MySignalWithArguments), this, "MyCallbackWithArguments");
}
```

Emitting signals is done with the `EmitSignal` method.

```
public void SomeFunction()
{
    EmitSignal(nameof(MySignal));
    EmitSignal("MySignalWithArguments", "hello there", 28);
}
```

Notice that you can always reference a signal name with the `nameof` keyword (applied on the delegate itself).

It is possible to bind values when establishing a connection by passing a Godot array.

```
public int Value { get; private set; } = 0;

private void ModifyValue(int modifier)
{
    Value += modifier;
}

public void SomeFunction()
{
    var plusButton = (Button)GetNode("PlusButton");
    var minusButton = (Button)GetNode("MinusButton");
```

```
    plusButton.Connect("pressed", this, "ModifyValue", new Godot.Collections.Array { 1 });
    minusButton.Connect("pressed", this, "ModifyValue", new Godot.Collections.Array { -1
});
}
```

Signals support parameters and bound values of all the built-in types and Classes derived from Godot.Object. Consequently, any `Node` or `Reference` will be compatible automatically, but custom data objects will need to extend from *Godot.Object* or one of its subclasses.

```
public class DataObject : Godot.Object
{
    public string Field1 { get; set; }
    public string Field2 { get; set; }
}
```

Finally, signals can be created by calling `AddUserSignal`, but be aware that it should be executed before any use of said signals (with `Connect` or `EmitSignal`).

```
public void SomeFunction()
{
    AddUserSignal("MyOtherSignal");
    EmitSignal("MyOtherSignal");
}
```

Preprocessor defines¶

Godot has a set of defines that allow you to change your C# code depending on the environment you are compiling to.

Note

If you created your project before Godot 3.2, you have to modify or regenerate your *csproj* file to use this feature (compare `<DefineConstants>` with a new 3.2+ project).

Examples¶

For example, you can change code based on the platform:

```
    public override void _Ready()
    {
```

```
#if GODOT_SERVER
        // Don't try to load meshes or anything, this is a server!
        LaunchServer();
#elif GODOT_32 || GODOT_MOBILE || GODOT_WEB
        // Use simple objects when running on less powerful systems.
        SpawnSimpleObjects();
#else
        SpawnComplexObjects();
#endif
    }
```

Or you can detect which engine your code is in, useful for making cross-engine libraries:

```
    public void MyPlatformPrinter()
    {
#if GODOT
        GD.Print("This is Godot.");
#elif UNITY_5_3_OR_NEWER
        print("This is Unity.");
#else
        throw new InvalidWorkflowException("Only Godot and Unity are supported.");
#endif
    }
```

Full list of defines¶

- `GODOT` is always defined for Godot projects.
- One of `GODOT_64` or `GODOT_32` is defined depending on if the architecture is 64-bit or 32-bit.
- One of `GODOT_X11`, `GODOT_WINDOWS`, `GODOT_OSX`, `GODOT_ANDROID`, `GODOT_IOS`, `GODOT_HTML5`, or `GODOT_SERVER` depending on the OS. These names may change in the future. These are created from the `get_name()` method of the OS singleton, but not every possible OS the method returns is an OS that Godot with Mono runs on.

When **exporting**, the following may also be defined depending on the export features:

- One of `GODOT_PC`, `GODOT_MOBILE`, or `GODOT_WEB` depending on the platform type.
- One of `GODOT_ARM64_V8A` or `GODOT_ARMEABI_V7A` on Android only depending on the architecture.

- One of `GODOT_ARM64` or `GODOT_ARMV7` on iOS only depending on the architecture.
- Any of `GODOT_S3TC`, `GODOT_ETC`, and `GODOT_ETC2` depending on the texture compression type.
- Any custom features added in the export menu will be capitalized and prefixed: `foo` -> `GODOT_FOO`.

To see an example project, see the OS testing demo: https://github.com/godotengine/godot-demo-projects/tree/master/misc/os_test

C# API differences to GDScript¶

This is a (incomplete) list of API differences between C# and GDScript.

General differences¶

As explained in the C# basics, C# generally uses `PascalCase` instead of the `snake_case` used in GDScript and C++.

Global scope¶

Global functions and some constants had to be moved to classes, since C# does not allow declaring them in namespaces. Most global constants were moved to their own enums.

Constants¶

Global constants were moved to their own enums. For example, `ERR_*` constants were moved to the `Error` enum.

Special cases:

GDScript	C#
`SPKEY`	`GD.SpKey`
`TYPE_*`	`Variant.Type` enum
`OP_*`	`Variant.Operator` enum

Math functions¶

Math global functions, like `abs`, `acos`, `asin`, `atan` and `atan2`, are located under `Mathf` as `Abs`, `Acos`, `Asin`, `Atan` and `Atan2`. The `PI` constant can be found as `Mathf.Pi`.

Random functions¶

Random global functions, like `rand_range` and `rand_seed`, are located under `GD`. Example: `GD.RandRange` and `GD.RandSeed`.

Other functions¶

Many other global functions like `print` and `var2str` are located under `GD`.
Example: `GD.Print` and `GD.Var2Str`.

Exceptions:

GDScript	C#
`weakref(obj)`	`Object.WeakRef(obj)`
`is_instance_valid(obj)`	`Object.IsInstanceValid(obj)`

Tips¶

Sometimes it can be useful to use the `using static` directive. This directive allows to access the members and nested types of a class without specifying the class name.

Example:

```
using static Godot.GD;

public class Test
{
    static Test()
    {
        Print("Hello"); // Instead of GD.Print("Hello");
    }
}
```

Export keyword¶

Use the `[Export]` attribute instead of the GDScript `export` keyword. This attribute can also be provided with optional PropertyHint and `hintString` parameters. Default values can be set by assigning a value.

Example:

```
using Godot;

public class MyNode : Node
```

```
{
    [Export]
    private NodePath _nodePath;

    [Export]
    private string _name = "default";

    [Export(PropertyHint.Range, "0,100000,1000,or_greater")]
    private int _income;

    [Export(PropertyHint.File, "*.png,*.jpg")]
    private string _icon;
}
```

Signal keyword¶

Use the `[Signal]` attribute to declare a signal instead of the GDScript `signal` keyword. This attribute should be used on a *delegate*, whose name signature will be used to define the signal.

```
[Signal]
delegate void MySignal(string willSendsAString);
```

See also: C# signals.

onready keyword¶

GDScript has the ability to defer the initialization of a member variable until the ready function is called with *onready* (cf. onready keyword). For example:

```
onready var my_label = get_node("MyLabel")
```

However C# does not have this ability. To achieve the same effect you need to do this.

```
private Label _myLabel;

public override void _Ready()
{
    _myLabel = GetNode<Label>("MyLabel");
}
```

Singletons¶

Singletons are available as static classes rather than using the singleton pattern. This is to make code less verbose than it would be with an `Instance` property.

Example:

```
Input.IsActionPressed("ui_down")
```

However, in some very rare cases this is not enough. For example, you may want to access a member from the base class `Godot.Object`, like `Connect`. For such use cases we provide a static property named `Singleton` that returns the singleton instance. The type of this instance is `Godot.Object`.

Example:

```
Input.Singleton.Connect("joy_connection_changed", this,
nameof(Input_JoyConnectionChanged));
```

String¶

Use `System.String` (`string`). Most of Godot's String methods are provided by the `StringExtensions` class as extension methods.

Example:

```
string upper = "I LIKE SALAD FORKS";
string lower = upper.ToLower();
```

There are a few differences, though:

- `erase`: Strings are immutable in C#, so we cannot modify the string passed to the extension method. For this reason, `Erase` was added as an extension method of `StringBuilder` instead of string. Alternatively, you can use `string.Remove`.
- `IsSubsequenceOf`/`IsSubsequenceOfi`: An additional method is provided, which is an overload of `IsSubsequenceOf`, allowing you to explicitly specify case sensitivity:

```
str.IsSubsequenceOf("ok"); // Case sensitive
str.IsSubsequenceOf("ok", true); // Case sensitive
str.IsSubsequenceOfi("ok"); // Case insensitive
str.IsSubsequenceOf("ok", false); // Case insensitive
```

- `Match` / `Matchn` / `ExprMatch`: An additional method is provided besides `Match` and `Matchn`, which allows you to explicitly specify case sensitivity:

```
str.Match("*.txt"); // Case sensitive
str.ExprMatch("*.txt", true); // Case sensitive
str.Matchn("*.txt"); // Case insensitive
str.ExprMatch("*.txt", false); // Case insensitive
```

Basis¶

Structs cannot have parameterless constructors in C#. Therefore, `new Basis()` initializes all primitive members to their default value. Use `Basis.Identity` for the equivalent of `Basis()` in GDScript and C++.

The following method was converted to a property with a different name:

GDScript	C#
`get_scale()`	`Scale`

Transform2D¶

Structs cannot have parameterless constructors in C#. Therefore, `new Transform2D()` initializes all primitive members to their default value. Please use `Transform2D.Identity` for the equivalent of `Transform2D()` in GDScript and C++.

The following methods were converted to properties with their respective names changed:

GDScript	C#
`get_rotation()`	`Rotation`
`get_scale()`	`Scale`

Plane¶

The following method was converted to a property with a *slightly* different name:

GDScript	C#
`center()`	`Center`

Rect2¶

The following field was converted to a property with a *slightly* different name:

GDScript	C#
`end`	`End`

The following method was converted to a property with a different name:

GDScript	C#
`get_area()`	`Area`

Quat¶

Structs cannot have parameterless constructors in C#. Therefore, `new Quat()` initializes all primitive members to their default value. Please use `Quat.Identity` for the equivalent of `Quat()` in GDScript and C++.

The following methods were converted to a property with a different name:

GDScript	C#
`length()`	`Length`
`length_squared()`	`LengthSquared`

Array¶

This is temporary. PoolArrays will need their own types to be used the way they are meant to.

GDScript	C#
`Array`	`Godot.Collections.Array`
`PoolIntArray`	`int[]`
`PoolByteArray`	`byte[]`
`PoolFloatArray`	`float[]`
`PoolStringArray`	`String[]`
`PoolColorArray`	`Color[]`
`PoolVector2Array`	`Vector2[]`
`PoolVector3Array`	`Vector3[]`

`Godot.Collections.Array<T>` is a type-safe wrapper around `Godot.Collections.Array`. Use the `Godot.Collections.Array<T>(Godot.Collections.Array)` constructor to create one.

Dictionary¶

Use `Godot.Collections.Dictionary`.

`Godot.Collections.Dictionary<T>` is a type-safe wrapper around `Godot.Collections.Dictionary`. Use the `Godot.Collections.Dictionary<T>(Godot.Collections.Dictionary)` constructor to create one.

Variant¶

`System.Object` (`object`) is used instead of `Variant`.

Communicating with other scripting languages¶

This is explained extensively in Cross-language scripting.

Yield¶

Something similar to GDScript's `yield` with a single parameter can be achieved with C#'s yield keyword.

The equivalent of yield on signal can be achieved with async/await and `Godot.Object.ToSignal`.

Example:

```
await ToSignal(timer, "timeout");
GD.Print("After timeout");
```

Other differences¶

`preload`, as it works in GDScript, is not available in C#.
Use `GD.Load` or `ResourceLoader.Load` instead.

Other differences:

GDScript	C#
`Color8`	`Color.Color8`
`is_inf`	`float.IsInfinity`
`is_nan`	`float.IsNaN`
`dict2inst`	TODO
`inst2dict`	TODO

C# style guide¶

Having well-defined and consistent coding conventions is important for every project, and Godot is no exception to this rule.

This page contains a coding style guide, which is followed by developers of and contributors to Godot itself. As such, it is mainly intended for those who want to contribute to the project, but since the conventions and guidelines mentioned in this article are those most widely adopted by the users of the language, we encourage you to do the same, especially if you do not have such a guide yet.

Note

This article is by no means an exhaustive guide on how to follow the standard coding conventions or best practices. If you feel unsure of an aspect which is not covered here, please refer to more comprehensive documentation, such as C# Coding Conventions or Framework Design Guidelines.

Language specification¶

Godot currently uses **C# version 7.0** in its engine and example source code. So, before we move to a newer version, care must be taken to avoid mixing language features only available in C# 7.1 or later.

For detailed information on C# features in different versions, please see What's New in C#.

Formatting¶

General guidelines¶

- Use line feed (**LF**) characters to break lines, not CRLF or CR.
- Use one line feed character at the end of each file, except for *csproj* files.
- Use **UTF-8** encoding without a byte order mark.
- Use **4 spaces** instead of tabs for indentation (which is referred to as "soft tabs").
- Consider breaking a line into several if it's longer than 100 characters.

Line breaks and blank lines¶

For a general indentation rule, follow the "Allman Style" which recommends placing the brace associated with a control statement on the next line, indented to the same level:

```
// Use this style:
if (x > 0)
{
    DoSomething();
}

// NOT this:
if (x > 0) {
    DoSomething();
}
```

However, you may choose to omit line breaks inside brackets:

- For simple property accessors.
- For simple object, array, or collection initializers.
- For abstract auto property, indexer, or event declarations.

```
// You may put the brackets in a single line in following cases:
public interface MyInterface
{
    int MyProperty { get; set; }
}

public class MyClass : ParentClass
{
    public int Value
    {
        get { return 0; }
        set
        {
            ArrayValue = new [] {value};
        }
    }
}
```

Insert a blank line:

- After a list of `using` statements.
- Between method, properties, and inner type declarations.

- At the end of each file.

Field and constant declarations can be grouped together according to relevance. In that case, consider inserting a blank line between the groups for easier reading.

Avoid inserting a blank line:

- After `{`, the opening brace.
- Before `}`, the closing brace.
- After a comment block or a single-line comment.
- Adjacent to another blank line.

```
using System;
using Godot;
                                                    // Blank line after `using` list.
public class MyClass
{                                   // No blank line after `{`.
    public enum MyEnum
    {
        Value,
        AnotherValue                // No blank line before `}`.
    }
                                                    // Blank line around inner types.
    public const int SomeConstant = 1;
    public const int AnotherConstant = 2;

    private Vector3 _x;             // Related constants or fields can be
    private Vector3 _y;             // grouped together.

    private float _width;
    private float _height;

    public int MyProperty { get; set; }
                                                    // Blank line around properties.
    public void MyMethod()
    {
        // Some comment.
        AnotherMethod();            // No blank line after a comment.
```

```
    }
                                        // Blank line around methods.
    public void AnotherMethod()
    {
    }
}
```

Using spaces¶

Insert a space:

- Around a binary and tertiary operator.
- Between an opening parenthesis and `if`, `for`, `foreach`, `catch`, `while`, `lock` or `using` keywords.
- Before and within a single line accessor block.
- Between accessors in a single line accessor block.
- After a comma which is not at the end of a line.
- After a semicolon in a `for` statement.
- After a colon in a single line `case` statement.
- Around a colon in a type declaration.
- Around a lambda arrow.
- After a single-line comment symbol (`//`), and before it if used at the end of a line.

Do not use a space:

- After type cast parentheses.
- Within single line initializer braces.

The following example shows a proper use of spaces, according to some of the above mentioned conventions:

```
public class MyClass<A, B> : Parent<A, B>
{
    public float MyProperty { get; set; }

    public float AnotherProperty
    {
        get { return MyProperty; }
    }
```

```
    public void MyMethod()
    {
        int[] values = {1, 2, 3, 4}; // No space within initializer brackets.
        int sum = 0;

        // Single line comment.
        for (int i = 0; i < values.Length; i++)
        {
            switch (i)
            {
                case 3: return;
                default:
                    sum += i > 2 ? 0 : 1;
                    break;
            }
        }

        i += (int)MyProperty; // No space after a type cast.
    }
}
```

Naming conventions¶

Use **PascalCase** for all namespaces, type names and member level identifiers (i.e. methods, properties, constants, events), except for private fields:

```
namespace ExampleProject
{
    public class PlayerCharacter
    {
        public const float DefaultSpeed = 10f;

        public float CurrentSpeed { get; set; }

        protected int HitPoints;
```

```
        private void CalculateWeaponDamage()
        {
        }
    }
}
```

Use **camelCase** for all other identifiers (i.e. local variables, method arguments), and use an underscore (`_`) as a prefix for private fields (but not for methods or properties, as explained above):

```
private Vector3 _aimingAt; // Use a `_` prefix for private fields.

private void Attack(float attackStrength)
{
    Enemy targetFound = FindTarget(_aimingAt);

    targetFound?.Hit(attackStrength);
}
```

There's an exception with acronyms which consist of two letters, like `UI`, which should be written in uppercase letters where PascalCase would be expected, and in lowercase letters otherwise.

Note that `id` is **not** an acronym, so it should be treated as a normal identifier:

```
public string Id { get; }

public UIManager UI
{
    get { return uiManager; }
}
```

It is generally discouraged to use a type name as a prefix of an identifier,
like `string strText` or `float fPower`, for example. An exception is made, however, for interfaces, which **should**, in fact, have an uppercase letter `I` prefixed to their names,
like `IInventoryHolder` or `IDamageable`.

Lastly, consider choosing descriptive names and do not try to shorten them too much if it affects readability.

For instance, if you want to write code to find a nearby enemy and hit it with a weapon, prefer:

```
FindNearbyEnemy()?.Damage(weaponDamage);
```

Rather than:

```
FindNode()?.Change(wpnDmg);
```

Member variables¶

Don't declare member variables if they are only used locally in a method, as it makes the code more difficult to follow. Instead, declare them as local variables in the method's body.

Local variables¶

Declare local variables as close as possible to their first use. This makes it easier to follow the code, without having to scroll too much to find where the variable was declared.

Implicitly typed local variables¶

Consider using implicitly typing (`var`) for declaration of a local variable, but do so **only when the type is evident** from the right side of the assignment:

```
// You can use `var` for these cases:

var direction = new Vector2(1, 0);

var value = (int)speed;

var text = "Some value";

for (var i = 0; i < 10; i++)
{
}

// But not for these:

var value = GetValue();

var velocity = direction * 1.5;
```

```
// It's generally a better idea to use explicit typing for numeric values, especially with
// the existence of the `real_t` alias in Godot, which can either be double or float
// depending on the build configuration.

var value = 1.5;
```

Other considerations¶

- Use explicit access modifiers.
- Use properties instead of non-private fields.
- Use modifiers in this order: `public` / `protected` / `private` / `internal` / `virtual` / `override` / `abstract` / `new` / `static` / `readonly`.
- Avoid using fully-qualified names or `this.` prefix for members when it's not necessary.
- Remove unused `using` statements and unnecessary parentheses.
- Consider omitting the default initial value for a type.
- Consider using null-conditional operators or type initializers to make the code more compact.
- Use safe cast when there is a possibility of the value being a different type, and use direct cast otherwise.