Tampere University of Applied Sciences

Differences Between C# and GDScript in Godot Game Engine

Jesse Vuorela

BACHELOR'S THESIS
December 2024

Business Information Systems
Games Production

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn tutkinto-ohjelma
Games Production

VUORELA, JESSE:
C# ja GDScriptin eroavaisuudet Godot-pelimoottorissa

Opinnäytetyö 30 sivua, joista liitteitä 5 sivua
Joulukuu 2024

Opinnäytetyössä vertailtiin C#- ja GDScript-ohjelmointikieliä Godot-pelimoottorissa. Työssä esitellään moottori ja ohjelmointikielet aloittelijatason Godot-pelikehittäjille. Kieliä tutkittiin ja testattiin vertailemalla niiden syntaksia, suorituskykyä ja yleisiä kyvykkyyksiä ohjelmoinnissa. Näiden tuloksien pohjalta voidaan arvioida, kumpi kieli sopii aloittelijalle paremmin.

Tulosten perusteella voidaan todeta, että vaikka GDScript on Godotille luotu ohjelmointikieli ja sillä on parempi kyvykkyys operoida moottorin ydinominaisuuksia, C#-kieli on sitä nopeampi. C# on myös parempi vaihtoehto rakentaa isompia ja monimutkaisempia projekteja kielen monimuotoisuuden vuoksi.

Vaikka C# on monipuolisempi kieli kuin GDScript, GDScript on silti suositeltavampi valinta aloittelijatason peliohjelmoijalle, koska se on helpompi oppia ja ydinominaisuudet ovat helpommin saatavilla. Jos ohjelmoijalla on jo jonkin verran taitoa C#-kielestä, C# saattaa olla siinä tapauksessa parempi vaihtoehto käyttää.

Avainsanat: Godot, C#, GDScript, peliohjelmointi, ohjelmointi

ABSTRACT

Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Games Production

VUORELA, JESSE:
Differences Between C# and GDScript in Godot Game Engine

Bachelor's thesis 30 pages, appendices 5 pages
December 2024

In this thesis C# and GDScript programming languages were compared on Godot game engine. The goal was to introduce the engine and these two languages to beginner-level Godot game developers. The languages were researched and experimented on by comparing their syntax, performance and overall capabilities in programming. Based on the results we can estimate which language is better for a beginner to use.

The results suggest that while GDScript is Godot's native language and has the upper hand on accessing the core features of the engine, C# is still generally faster than GDScript. C# also has more ways to structure code, which helps in maintaining bigger and complex projects.

Even though C# is more versatile language than GDScript, GDScript is still a better option for a beginner-level Godot game developer, since it is easier to learn, and the core features of the engine are more accessible on it. If a developer knows already some C#, it might be a better option to use then.

Key words: Godot, C#, GDScript, game programming, programming

CONTENTS

ABBREVIATIONS AND TERMS

.NET	A framework created by Microsoft
API	Application programming interface
C#	A programming language created by Microsoft
Dynamic typing	Typing system where a type is assigned to variables at run-time
GDExtension	Godot-specific technology created for interacting with native shared libraries during run-time
GDScript	A programming language created for Godot game engine
Godot	A game engine
IDE	Integrated development environment
Object-oriented programming	Programming paradigm based on the concept of objects
Static typing	Typing system where variables are bound to a data type during code compilation
Virtual machine	Emulation or virtualization of a computer system

1 INTRODUCTION

I have always been interested in video games and how they are made. When I started studying game development at Tampere's university of applied science, I got familiar with several game engines and learned how they work. Sometimes it is hard to find good and easily understandable information for a new developer, especially when the engine in question is fairly new and has just started to gain a lot of attraction.

The purpose of this thesis is to compare C# and GDScript programming languages on Godot game engine. Comparison of the syntax, performance and accessibility of the languages with examples help to determine which language might be better in game development for a new game developer. The goal is to introduce Godot to beginner level programmers.

Script execution time will be monitored by inserting time tracking methods in the script. To determine the performance of the languages, Godot's own built-in performance monitoring can also be used.

These questions are to be answered during comparison

- Are there any significant differences in the performance depending on the language?
- Is one of the languages easier to learn as a new Godot game developer?
- Are there any features that the other language lacks?

Both languages will be compared on Godot's .NET version 4.3. The engine will not have any additional modifications done; it is as it is as when downloaded freshly. Any changes made to the engine will be shown and explained.

2 GODOT

Godot is a free 2D and 3D, open-source game engine. It was released under the OSI-approved MIT license and was initially developed by 2 software developers: Juan Linietsky and Ariel Manzur. (Godot Foundation 2024e.) The first release of the engine was version 1.0 on 15th December 2014 (Godot Foundation 2024g). The engine is nowadays supported by a Dutch non-profit organization “The Godot Foundation” which administers charitable contributions made to the Godot Engine. It was formed on August 23rd, 2022. Before that, funding was managed by Software Freedom Conservancy (SFC). (Godot Foundation 2024f.)

Godot officially supports exporting projects on desktop platforms, web and mobile (Android and iOS). Also, consoles (PlayStation 4/5, Xbox One and Xbox Series X/S, Nintendo Switch and Steam Deck) are supported, but only Steam Deck is officially supported by Godot. The reason why other consoles aren’t officially supported is because Godot, or any other engine is not allowed to distribute console export templates to a developer, if they are not a licensed console developer. (Godot Foundation 2024b, 2024m.)

Godot has two main versions, standard and .NET versions. The .NET support was later added to the engine, in version 3.0. They are both on release version 4.3 which was released on 15th of August 2024, and they are both available for desktop platforms from Godot’s own web page. Digital store versions do not include .NET support. (Godot Foundation 2024a.)

The engine is lightweight and very “bare bones”. The engine is kept simple, and any extra additional features are not really added, since developers can modify the engine for their liking since it is modular and open source. Because of this, the engine runs well even on older hardware. The engine has its own built-in script editor, though its support is minimal for other languages than GDScript (Godot Foundation, 2024v).

Godot games consist of nodes, Godot’s building blocks. These nodes can be assigned as the parent or child of another node, resulting in a tree arrangement.

(Godot Foundation, 2024c.) This tree arrangement can be saved as a single scene, and these scenes allow you to structure the game however you like.

3 ENGINE LANGUAGES

The engine is written in C++, and it supports multiple programming languages. GDScript and C# are the officially supported languages to use in game development in Godot. But with the GDExtension technology, games can also be made with C or C++. It also supports integrating Software Development Kits (SDKs) and third-party libraries in the engine. (Godot Foundation 2024e, 2024h.) Visual scripting was also added to the engine in version 3.0 but was removed from the engine in version 4.0 for not gaining enough traction (Godot Foundation 2024i). Computationally intensive games that are unable to benefit from the built-in tools in the engine have the possibility of using C++ (Godot Foundation 2024j).

3.1 GDScript

GDScript was developed specifically for Godot. It replaced Lua scripting language in the early days and has been heavily supported since. There were many reasons for creating a custom scripting language, such as: no native vector types, poor support in most script virtual machines, difficulties integrating with the code editor and much more. (Godot Foundation 2024e.)

GDScript is an object-oriented, high-level, imperative and gradually typed programming language (Godot Foundation 2024k). Even though the language leans more towards dynamic typing, there is still an option to type statically (Godot Foundation 2024l). The syntax is inspired by Python, it is not based on it (Godot Foundation 2024k).

3.2 C#

The first widely distributed C# implementation was developed and released by Microsoft in 2000 as part of the .NET framework (Ecma International 2006). The

first version (1.0) was released in 2002. The latest version is C# 12, released in November 2023.

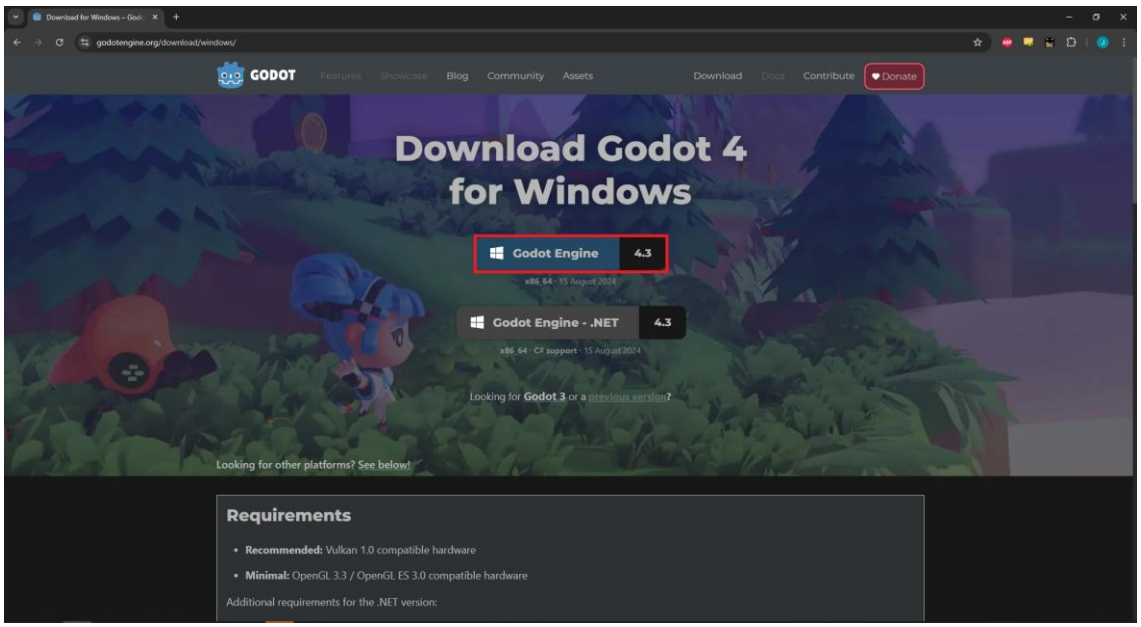
C# is a type-safe high-level, object-oriented and statically typed programming language. Its roots come from the C family which includes C, C++ and Java. (W3School, 2024.) It has a wide range of libraries. This helps when Godot does not provide something that is needed.

4 SETTING UP GODOT

Since there are two different versions of the engine, there are also two different ways to download the engine. The standard version is more straightforward to download and install compared to the .NET version. The .NET version needs Microsoft's .NET SDK to work. Version 6.0 is the minimum requirement (Godot Foundation, 2024m).

4.1 Godot Engine 4.3

The standard version of Godot is downloadable from Godot's own website (Godot Foundation, 2024a) (Picture 1). From there you can download the 4.3 version, or any of the older versions.

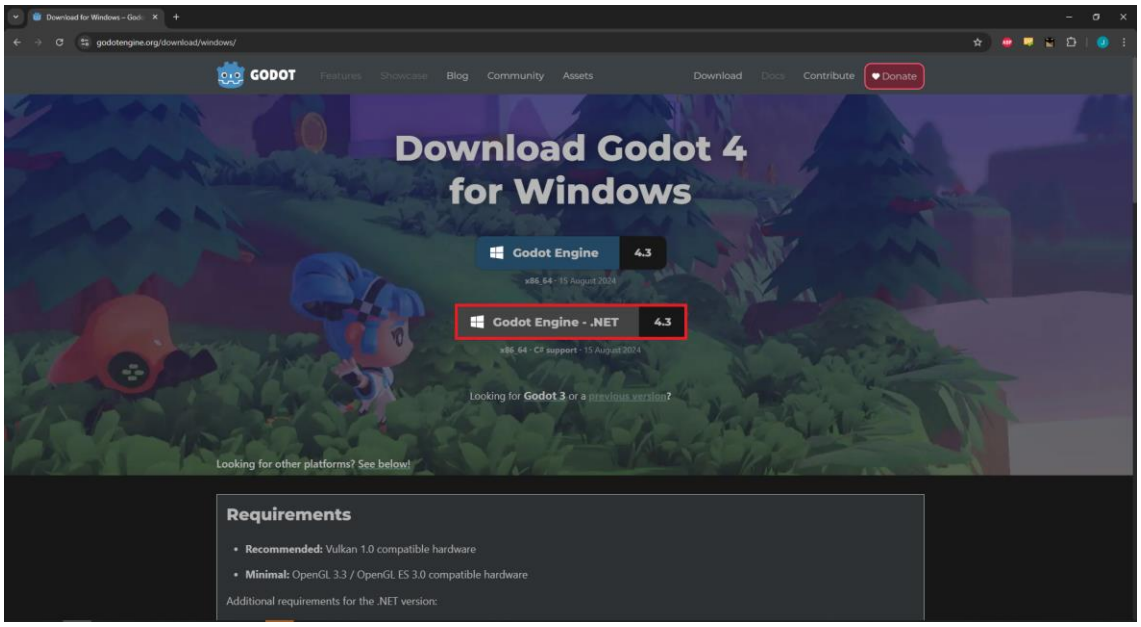


Picture 1. Where to download the standard version of the engine

After downloading the files, what the user needs to do is extract the downloaded content where they desire on their computer. When that is done, Godot is ready to start.

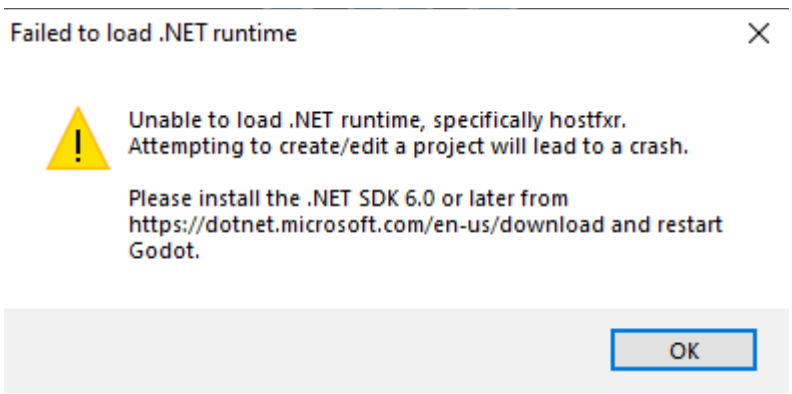
4.2 Godot Engine .NET 4.3

The .NET version of the engine can also be downloaded from the same webpage as the standard version (Picture 2). The .NET version contains all the bindings from C# to Godot C++ APIs. For this version, there are additional requirements to do before the engine is usable.



Picture 2. Where to download the .NET version of the engine

After Godot's files have been downloaded and extracted, you need to install the .NET SDK files. If the files have not been installed, Godot fails to launch. A message is prompted to the user if the .NET runtime fails to load (Picture 3).



Picture 3. The prompted message after trying to launch .NET version of Godot without .NET installed on the hardware.

The .NET SDK files can be found from Microsoft's website (Microsoft, 2024). There you can find the latest version (8.0) and download it. After downloading the installer, the user should open and follow the instructions on the installer to install the .NET SDK. When that is done, Godot is functional.

4.2.1 .NET SDK versions

Depending on the platform the game is exported to, you may need a higher .NET version than the minimum version of 6.0. Projects written in C# currently cannot be exported to the web using Godot 4 (Godot Foundation 2024o). The minimum required .NET versions for all different platforms can be found on the table below (Picture 4).

Platform	Runtimes supported	Minimum required .NET version
Windows	CoreCLR, Mono, NativeAOT	6.0 (CoreCLR), 7.0 (Mono, NativeAOT)
macOS	CoreCLR, Mono, NativeAOT	6.0 (CoreCLR), 7.0 (Mono, NativeAOT)
Linux	CoreCLR, Mono, NativeAOT	6.0 (CoreCLR), 7.0 (Mono, NativeAOT)
Android	Mono	7.0
iOS	NativeAOT	8.0
Web	-	-

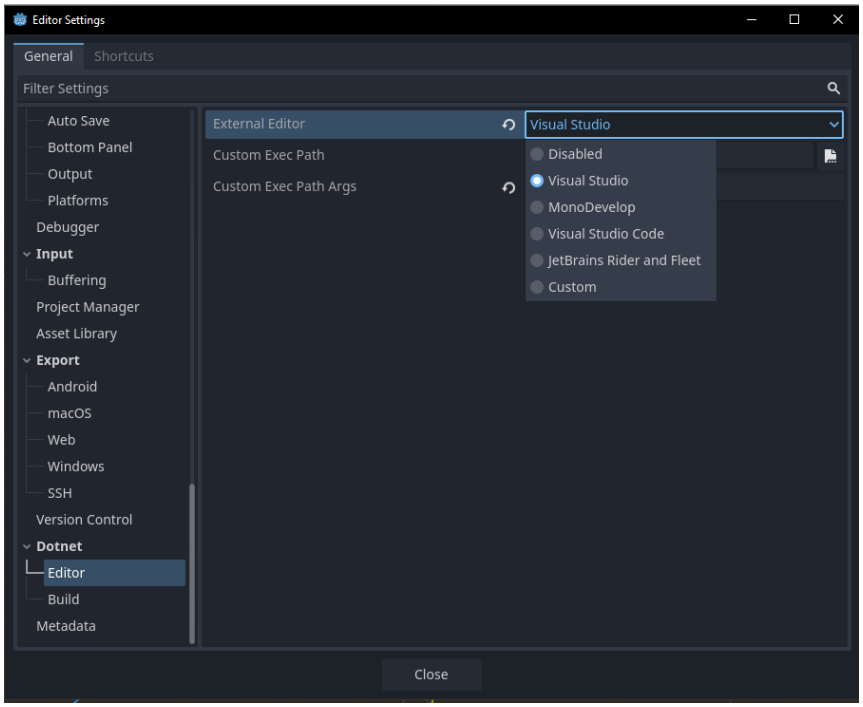
Picture 4. The current .NET platform support as of version 4.2 (Godot Foundation 2024m).

4.2.2 External IDE

Even though Godot has a built-in script editor, the C# support is minimal. You should consider using an external IDE. Godot supports many external IDEs, and they have a different configuration process. For this I am going to use Visual Studio 2022.

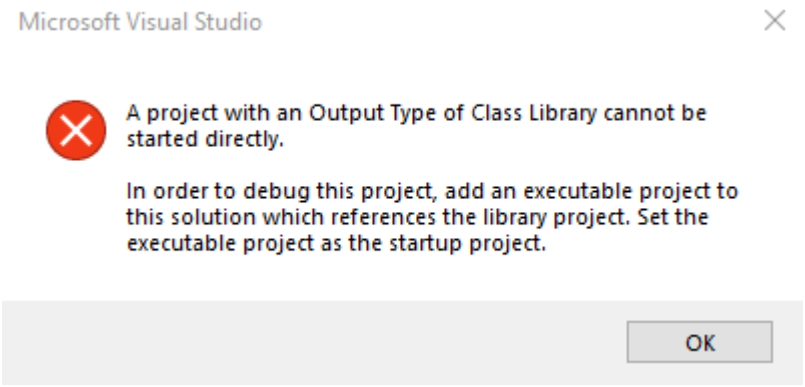
A solution file is required for Visual Studio to work on a project. This solution file can be generated by creating a C# script in Godot. After creating a new script, a C# Project file, Visual Studio Solution and some utility files are created in the project folder.

When using external IDE, you should change Godot to open it automatically when opening scripts from the engine. This can be done in the editor settings (Picture 5).



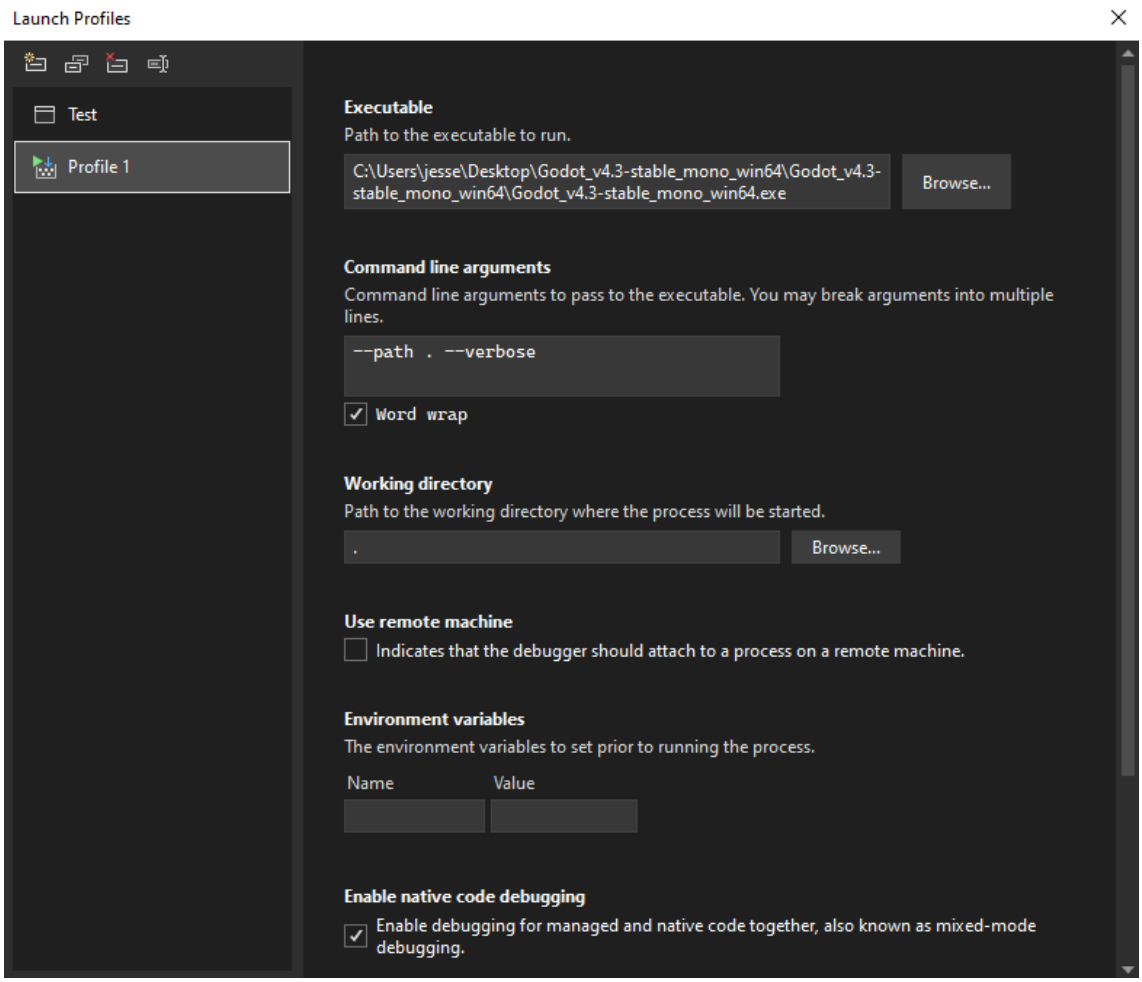
Picture 5. Setting the external editor in Godot editor settings.

After changing the external editor, Visual Studio is ready to use, but it is still missing its powerful debugger feature. If a user tries to debug their project without setting the launch options for the debugger, the user is met with an error message (Picture 6).



Picture 6. The error message when the debugger is not configured in the project

The debugger launch options can be set from Visual Studio's debug properties by creating a new executable launch profile. When the executable is created, some arguments must be set to get it working. You must assign the Godot's .exe path for the executable to run and add "—path ." and "—verbose" to the command line arguments. Godot has its own comprehensive list of command line references on their documentation webpage (Godot Foundation, 2024p). After the arguments have been set, set the working directory to just "." to refer to the current directory name component, and enable native code debugging. A ready working launch profile example is below (Picture 7). The launch profile is now set, and all the user needs to do is set a main scene in the Godot project settings and the debugger is ready to use. Restarting Godot and Visual Studio is recommended at this point.



Picture 7. The executable arguments for Visual Studio debugger

5 DIFFERENCES BETWEEN GDSCRIPT AND C#

There are a lot of differences between GDScript and C#. Since C# was created and is maintained by Microsoft it is much more versatile and polished than GDScript. But since GDScript was made especially for Godot, it has the upper hand on accessing core features of the engine compared to the other languages.

Here I will demonstrate how the two languages differ from each other. Comparing the syntax, performance and accessibility of the languages will give a good perspective on how the languages perform and which is more user-friendly to new developers.

5.1 Static and dynamic typing

Since GDScript is a gradually typed language, it is capable of using both static and dynamic typing. While GDScript is easier and quicker to write dynamically, it is less performant compared to static typing. Because of this, it is important to try and type statically while making more complex games. It also helps in making your code more error safe (Godot Foundation, 2024l).

If typing dynamically, all variables are “variant”-like, meaning that their type is not fixed (Godot Foundation, 2024j). Statically typed code must have all data types assigned. This promotes type safety and error detection at an early stage. Static typing also gives better code completion options compared to dynamic. (Godot Foundation, 2024l.)

To type statically with GDScript, the type of a variable, parameter, function or constant must be defined. For variables this can be done by writing a colon after the name, followed by its type. To define the return type of a function, you must type an arrow (dash and a right-angle bracket) after function declaration, followed by the return type (Godot Foundation, 2024l) (Pictures 8, 9.) With C# typing is always static.

```
var health = 100
var person = "Tom"

func _ready():
```

Picture 8. Variables and a function typed dynamically

```
var health: int = 100
var person: String = "Tom"

func _ready() -> void:
```

Picture 9. Variables and a function typed statically

5.2 Interfaces and duck typing

If the same code is needed in many classes, interfaces can be used to implement the same functionality to several classes. An interface defines a contract. Any class, struct or record implementing the contract must provide an implementation defined in the interface. An interface may define static virtual or static abstract members. (Microsoft, 2024a.) In picture 10 an interface defines a function and a health property. A class inheriting the interface must implement all definitions in the interface as in picture 11.

```
1 reference
interface IInterface
{
    2 references
    void DoThisFunction();

    5 references
    float Health
    {
        get;
        set;
    }
}
```

Picture 10. An interface class in C#.

```
7 references
public partial class InterfaceScript : Node, IInterface
{
    5 references
    public float Health
    {
        get;
        set;
    }

    2 references
    public void DoThisFunction()
    {
        GD.Print("This function is derived from the IInterface class");
    }
```

Picture 11. A class implementing the interface class.

While interfaces are supported on C#, GDScript lacks support for it, instead it relies largely on duck typing. Duck typing is a term used in dynamically typed languages to describe the polymorphic behaviour of objects. With duck typing the class or type of an object is not important, only the method it defines. This way we do not have to care about inheriting functions.

In picture 12, we have an animal class that has a function that prints to the console. We can do a reference to that class in another class as in picture 13 and call for that function by its name. Godot will try to use the given object at runtime and raises an error if the object does not have this function.

```
extends Node

class_name Animal

func fly():
    print("Animal flies!")
```

Picture 12. The animal class.

```
extends Node

class_name NodeClass

var animal = Animal.new()

func _ready():
    animal.fly()
```

Picture 13. Another class referencing to the animal class and trying to execute the fly() function. All this class cares about is that the referenced object has the fly() function, no type checking needed.

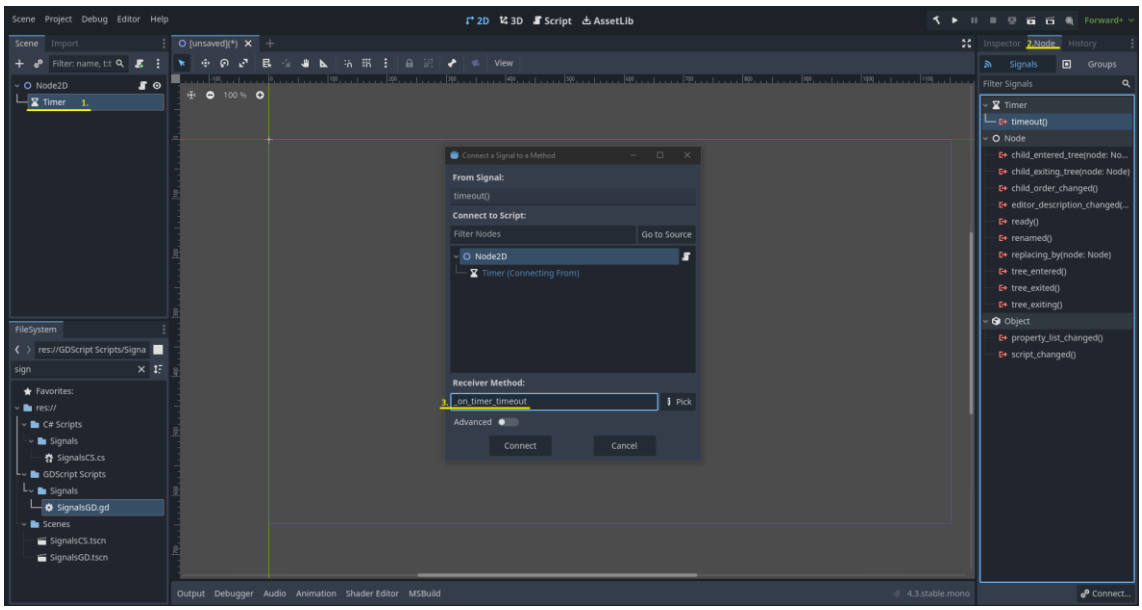
5.3 Signals

Signals are a built-in delegation mechanism in the engine and both GDScript and C# are supported. Signals are messages emitted by nodes when something specific happens to them. (Godot Foundation, 2024d.) This way different nodes can communicate with each other.

To provide more type-safety, signals in C# are implemented using C# events. C# events work like signals, they enable a class or object to notify other objects or classes when something specific happens to them.

Signals emitting from a node can be connected to other nodes' scripts from the node dock, which displays a list of available signals on the selected node. When a signal is selected, a receiver method must be defined to connect the signal. After connecting, Godot makes a function automatically for GDScript.

In this example a child timer node is connected to its parent node's script (Picture 14). The timer has properties set from the inspector menu, so it will automatically start counting to one second. After the timer is done, it will automatically stop counting. The created receiver method in the parent node's script will print to console when the counting is done (Picture 15). The script is made in GDScript.

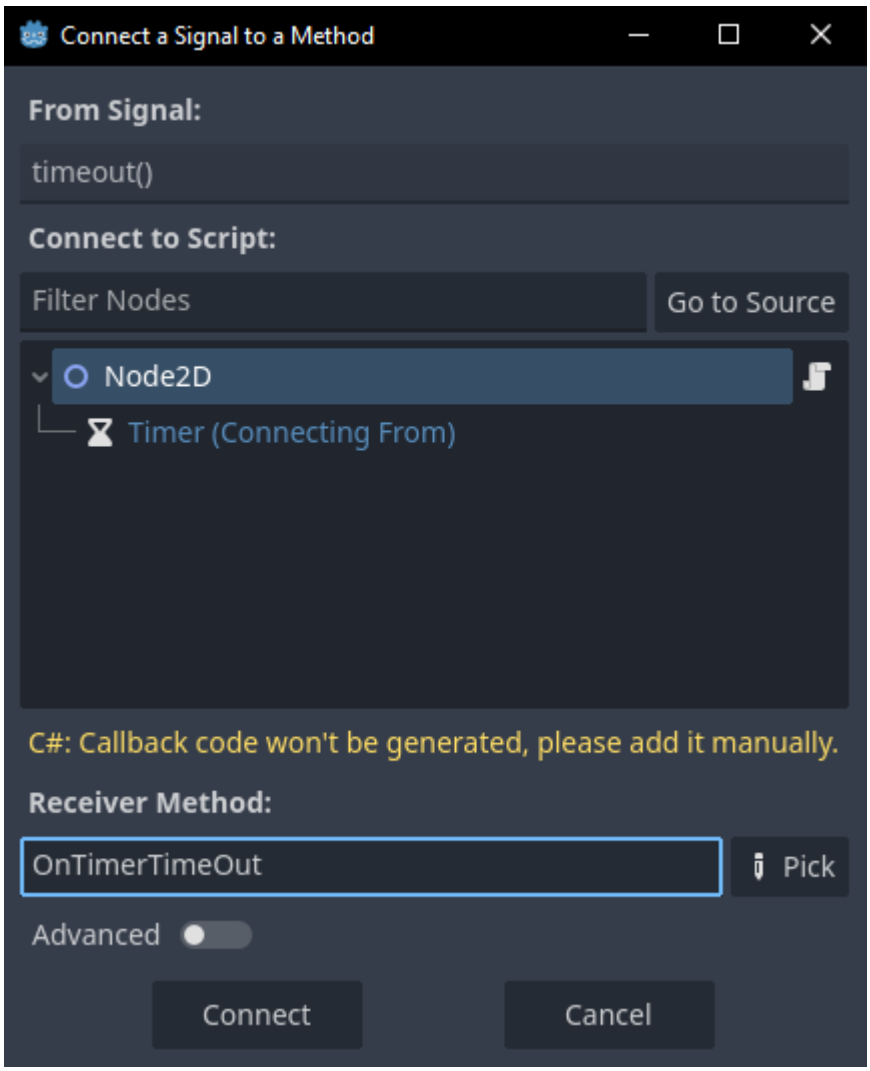


Picture 14. Connecting a signal from a child node to parent node's script. 1. A timer node has been made and selected. 2. Timer node's signal menu, timeout() function is selected. 3. Defining the receiver method on the parent node's script.

```
13  # This function is connected from the "Timer" node's signal menu.
14  func _on_timer_timeout() -> void:
15      print("Timer counted to one!")
```

Picture 15. The created receiver method on the parent node's script in GDScript. This will print to the console after the timer has counted to one. Notice the green arrow on left, this indicates that the function is connected to a signal.

Signals can be connected to C# the same way as in GDScript. The only difference is that the receiver method will not be generated automatically, it must be defined in the script by hand as we see in pictures 16 and 17.



Picture 16. Connecting a signal to a receiver method in C#. Method name is written in C# style. This function must be defined in the parent node's C# script by hand after connecting.

```
/// <summary>
/// This method is connected from the node's
/// signal menu.
/// </summary>
1 reference
private void OnTimerTimeout()
{
    GD.Print("Timer has counted to one!");
}
```

Picture 17. The created receiver method on the parent node's script in C#. This must be written by hand; it is not generated automatically.

Signals can also be connected via code instead of using the editor. This is important if nodes are instantiated from a script and signal connecting is impossible. They can be connected by first fetching the needed node, in this case the timer

node, and store the reference to it in a variable. After that we can connect the timer's timeout signal to the node which the script is attached to as seen in picture 18.

```
5  # Timer node is fetched and connected to
6  # _print_on_timeout function.
7  func _ready() -> void:
8      var timer = get_node("Timer")
9      timer.timeout.connect(_print_on_timeout)
```

Picture 18. Connecting a signal via code from a child node to the parent node's script in GDScript.

C# implementation is almost the same as in GDScript, only the syntax is a bit different. By inspecting picture 19, the C# implementation must have the definitions of types based on static typing's nature. Also the connect() method is replaced with C#'s event handling operator "+=".

```
/// <summary>
/// This way you can connect a signal without
/// connecting it from the node's signal menu.
/// </summary>
1 reference
public override void _Ready()
{
    Timer timer = GetNode<Timer>("Timer");
    timer.Timeout += PrintSignalOnTimeout;
}
/// <summary>
/// This method prints when timer has counted to one.
/// </summary>
2 references
private void PrintSignalOnTimeout()
{
    GD.Print("This signal function is connected from the script!");
}
```

Picture 19. Signal connected via code in C#.

Custom signals can also be defined in a script. This can be helpful if Godot does not offer readily available signals. Custom signals can be defined with "signal" attribute in GDScript and C#. The signal then becomes visible in the node's signal menu and this signal can then be connected as needed. Custom signals can be emitted from the script with emit() function in GDScript and EmitSignal() function

in C#. In C# the project must be built first to make the signal visible in the signal menu. In picture 20 we have an example of custom signal creation in GDScript and in picture 21 in C#.

```
# Assigning a custom signal
signal print_signal

func _ready() -> void:
    # Emitting a signal with the custom signal
    print_signal.emit()

# This function has been connected with a custom signal and will be emitted
# with the emit() function.
func _on_print_signal() -> void:
    print("Printed with custom signal!")
```

Picture 20. Emitting a custom signal in GDScript. This signal is connected to the on_print_signal() function from the node's signal menu.

```
// Assigning a custom signal
[Signal]
public delegate void PrintSignalEventHandler();


1 reference
public override void _Ready()
{
    // Emitting a custom signal
    EmitSignal(SignalName.PrintSignal);
}
/// <summary>
/// This function has been connected with a custom signal
/// and will be emitted with the EmitSignal() function.
/// </summary>
1 reference
private void OnPrintSignal()
{
    GD.Print("Printed with custom signal!");
}
```

Picture 21. Emitting a custom signal in C#. The signal is connected to the On-PrintSignal() function from the node's signal menu. The name of the delegate must end with "EventHander" and this part must be removed when calling the signal.

5.4 Array sorting

To compare the performance of the two languages, I have done three different random array sorting scripts. One is for GDScript and two are for C#. The reason for the two different C# scripts is that there are different ways to do this in C#. One way is to use C#'s own collections which have their own methods, the other way is to use Godot's own collections.

All scripts generate a random array with thousand random numbers from zero to hundred, and sort them based on their value (from small to big). These scripts were repeated ten times, and an average was taken as the result. Time is tracked in C# using its System.Diagnostics collections. The collection allows us to track time in ticks and see how long each part of the script takes. A single tick is equal to one hundred nanoseconds. GDScript has also the ability to track time the same way as in C#, though the results come in this case as seconds. In the following results for clarity's sake the ticks in C# have been converted to milliseconds, the same applies to GDScript's seconds.

By inspecting chart 1, we can notice that C#'s own collections are much faster in randomizing and sorting the array compared to Godot's own. Even though C#'s collections are faster than Godot's, the case might not be the same in every aspect. While indeed C# has generally faster collections, we still end up with slower performance in this case because we have to convert the array to Godot's own variant type to print the unsorted and sorted arrays in the console. In chart 2 we can see how big of a performance dip this conversion is. It is good practice to avoid using Godot.Variant unless you do not have other options but to use it.

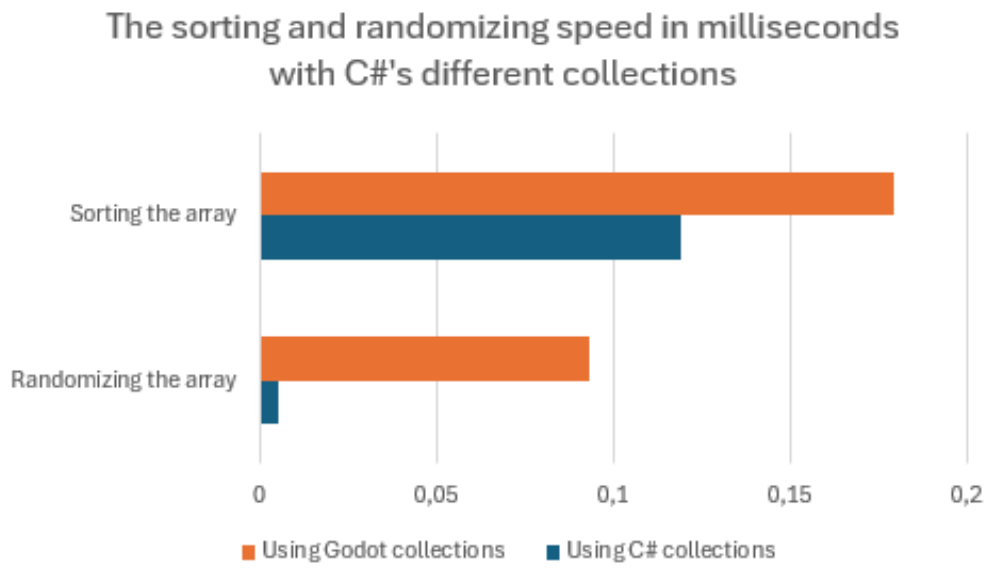


Chart 1. The speed differences of C# collections.

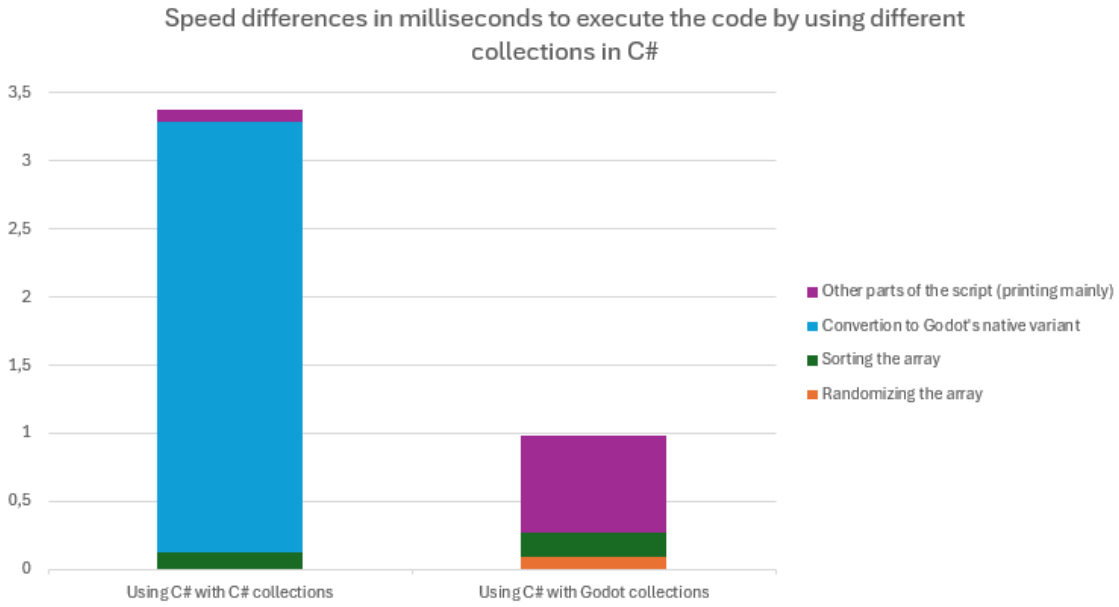


Chart 2. The whole script execution time in milliseconds by using different C# collections.

While GDScript is the native language of the engine, it is still slower compared to C# (Chart 3). While the chart shows that GDScript is faster than C# using its own collections, GDScript is still overall slower since converting the array to Godot's own variant type takes over 90% of the code execution time as seen previously. This means that C# with its own native collections perform otherwise better compared to both GDScript and Godot's own collections in C#.

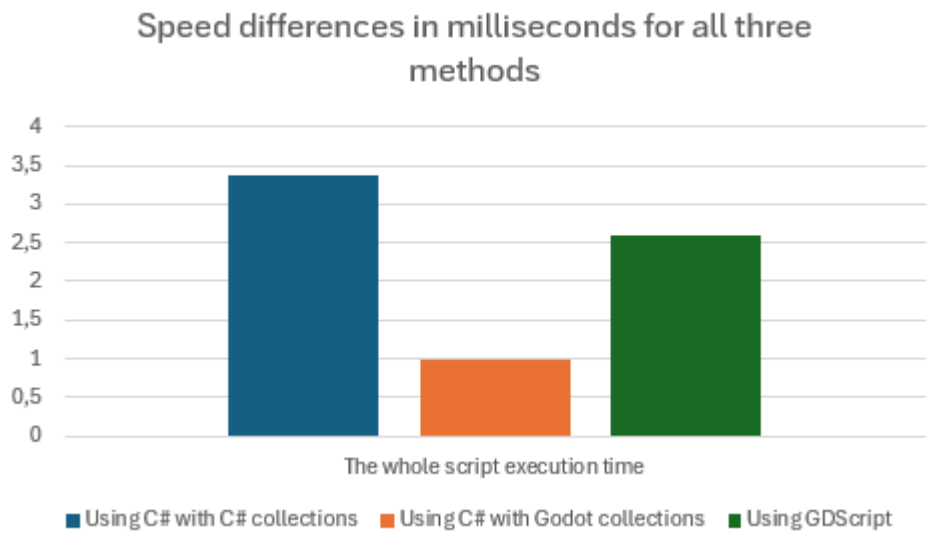


Chart 3. The whole script execution times in GDScript and in C# using different collections.

6 DISCUSSION

The purpose of this thesis was to get acquainted with Godot game engine and to compare GDScript and C# in Godot game development. Some of important aspects in Godot game development were discussed and we can conclude that GDScript is probably the language to go with when started developing with Godot.

While C# can be used to structure code better, it may be a little overwhelming for a beginner programmer. Even though C# is performance wise faster, usually beginner-level programmers' projects are small enough to manage with GDScript. Also, it has simpler syntax and the need to write less code. Even though GDScript does not support everything that C# has to offer, it has its own ways to do things. The core features are also more easily accessible, and the language is immediately available to use without installing any frameworks or anything else besides the engine. While more performance tests could have been done, the array sorting example give a good enough hint of the overall performance of the languages.

While GDScript should be considered to use as the first language to use with Godot, C# is still a good option to try and use since it can be used to so many other things than only game development. Especially if a developer already knows some C#, it is a really good choice to begin with.

REFERENCES

Godot Foundation, 2024a. Referenced 8.10 https://godotengine.org/download/windows/

Godot Foundation, 2024b. Referenced 8.10 https://docs.godotengine.org/en/stable/tutorials/platform/consoles.html

Godot Foundation, 2024c. Referenced 18.11 https://docs.godotengine.org/en/stable/classes/class_node.html

Godot Foundation, 2024d. Referenced 18.11 https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html

Godot Foundation, 2024e. Referenced 10.10 https://docs.godotengine.org/en/stable/about/faq.html#

Godot Foundation, 2024f. Referenced 12.10 https://godot.foundation/

Godot Foundation, 2024g. Referenced 12.10 https://godotengine.org/download/archive/

Godot Foundation, 2024h. Referenced 12.10 https://docs.godotengine.org/en/stable/getting_started/introduction/introduction_to_godot.html#

Godot Foundation, 2024i. Referenced 12.10 https://docs.godotengine.org/en/3.5/tutorials/scripting/visual_script/index.html

Godot Foundation, 2024j. Referenced 12.10 https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_advanced.html

Godot Foundation, 2024k. Referenced 12.10 https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html

Godot Foundation, 2024l. Referenced 12.10 https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/static_typing.html

Godot Foundation, 2024m. Referenced 13.10 https://godotengine.org/article/platform-state-in-csharp-for-godot-4-2/

Godot Foundation, 2024o. Referenced 13.10 https://docs.godotengine.org/en/stable/tutorials/export/exporting_for_web.html

Microsoft, 2024. Referenced 23.10 https://dotnet.microsoft.com/en-us/download

Godot Foundation, 2024p. Referenced 7.11 https://docs.godotengine.org/en/stable/tutorials/editor/command_line_tutorial.html#doc-command-line-tutorial

Microsoft, 2024a. Referenced 14.11 https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface

Ecma International, 2006. Referenced 19.11 https://www.ecma-international.org/wp-content/uploads/ECMA-334_4th_edition_june_2006.pdf

W3Schools, 2024. Referenced 19.11 https://www.w3schools.com/cs/cs_intro.php

APPENDICES

Appendix 1. Signals in GDScript

```gdscript
extends Node2D

# The class name
class_name Signals

# Assigning a custom signal
signal print_signal

func _ready() -> void:
	# Emitting a signal with the custom signal
	print_signal.emit()
	# Connect a signal via code from a child node
	var timer = get_node("Timer")
	timer.timeout.connect(_print_on_timeout)

# This function has been connected with a custom signal and will be emitted
# with the emit() function.
func _on_print_signal() -> void:
	print("Printed with custom signal!")

# This function is connected in the ready function.
func _print_on_timeout() -> void:
	print("Hello from a custom signal!")

# This function is connected from the "Timer" node's signal menu.
func _on_timer_timeout() -> void:
	print("Hello!")
```

Appendix 2. Signals in C#

```
using Godot;

11 references
public partial class SignalsCS : Node2D
{
    // Assigning a custom signal
    [Signal]
    public delegate void PrintSignalEventHandler();

    1 reference
    public override void _Ready()
    {
        // Emitting a custom signal
        EmitSignal(SignalName.PrintSignal);
        // This way you can connect a signal without
        // connecting it from the node's signal menu.
        Timer timer = GetNode<Timer>("Timer");
        timer.Timeout += PrintSignalOnTimeout;
    }

    /// <summary>
    /// This function has been connected with a custom signal
    /// and will be emitted with the EmitSignal() function.
    /// </summary>
    1 reference
    private void OnPrintSignal()
    {
        GD.Print("Printed with custom signal!");
    }

    /// <summary>
    /// This method has been connected via code in the _ready() function.
    /// </summary>
    2 references
    private void PrintSignalOnTimeout()
    {
        GD.Print("This signal function is connected from the script!");
    }

    /// <summary>
    /// This method is connected from the node's
    /// signal menu.
    /// </summary>
    1 reference
    private void OnTimerTimeout()
    {
        GD.Print("Timer has counted to one!");
    }
}
```

Appendix 3. Sorting in C# with Godot collections

```
using Godot;
using System;

7 references
public partial class SortingCS2 : Node
{
    public Godot.Collections.Array array = new Godot.Collections.Array();

    Random randNum = new Random();

    int count = 1000;

    1 reference
    public override void _Ready()
    {

        for (int i = 0; i < count; i++)
        {
            array.Add(randNum.Next(0, 100));
        }

        GD.Print("Unsorted");
        GD.Print(array);

        array.Sort();

        GD.Print("Sorted");
        GD.Print(array);
    }
}
```

Appendix 4. Sorting in C# with C# collections

```
using Godot;
using System;

7 references
public partial class SortingCS : Node
{
    int[] array = new int[1000];
    Random randNum = new Random();

    // Called when the node enters the scene tree for the first time.
    1 reference
    public override void _Ready()
    {

        for (int i = 0; i < array.Length; i++)
        {
            array[i] = randNum.Next(0, 100);
        }

        GD.Print("Unsorted");
        GD.Print(Variant.From(array));

        Array.Sort(array);

        GD.Print("Sorted");
        GD.Print(Variant.From(array));
    }
}
```

Appendix 5. Sorting in GDScript

```
1   extends Node
2
3   var array = []
4   var count = 1000
5
6   # Called when the node enters the scene tree for the first time.
7   func _ready():
8   
9       for i in count:
10  
11          var randNum = RandomNumberGenerator.new()
12  
13          randNum.randomize()
14          var number = randNum.randi_range(0, 100)
15  
16          array.insert(0, number)
17
18      print("Unsorted")
19      print(array)
20  
21      array.sort()
22  
23      print("Sorted")
24      print(array)
```