Radikon Scripting System

An Object-Based Scripting System written in C#

Radikon Scripting System is an open source object-based Level Scripting Utility for game projects, designed for developers and students. This utility features a custom window for browsing nodes, as well as a collection of attributes for customising node appearances and filtering criteria. Additionally, Radikon Scripting System provides experienced programmers with an extensible base for projects looking to provide tools to developers familiar with Node-Based scripting systems.

This utility comes with a few nodes provided for control flow and object manipulation, however, custom nodes will need to be created on a per-project basis.

Current Release: Version 1.0

Overview

Full C# XML Documentation

All classes and members are documented within C#, using inheritance to trickle down to included and new sub-classes.

Node Browser

A custom editor window that can be used to find, display and filter nodes within any Resources/NodePrefabs folder. This browser also automatically aids with setting up an empty object to store all nodes, to prevent messing with scenes.

Customisation and Filtering Attributes

Developers can leverage a provided set of C# attributes to change the appearance and tool-tips for a Scripting Node, as well as how it is filtered by the Node Browser.

Custom Icons

A range of custom icons, created during the development of Radikon War: Lone Warrior, have been provided alongside the scripting system for use in Unity Editor only. Located in: **Utilities/ScriptingSystem/Editor/Resources**

ScriptingNode

A lightweight base class containing core members and methods for directing the control flow of nodes.

IScriptingSystemEntryPoint

An interface that marks a node as the Start Node for the Scripting System begin at from within a scene.

ScriptingCore

A singleton manager class that binds to the Runtime startup process, in order to detect nodes within loaded scenes and manage scripting control flow.

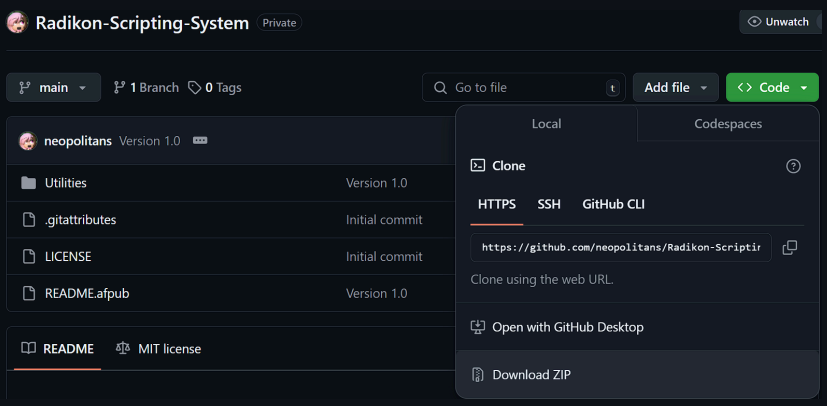
Notice

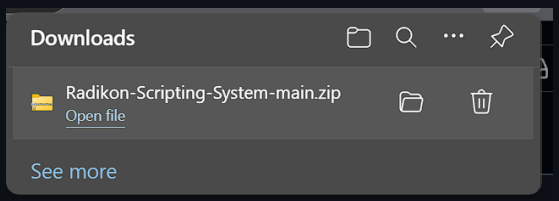
This Utility deals with a range of C# and .NET namespaces, such as Reflection and LINQ. Please test this utility inside of a test project first to determine if Radikon Scripting System is right for your needs. While this scripting system is entirely enclosed and should not interfere with other scripting systems, due to it's use of namespaces, there is no guarantee of compatibility.

This utility has been tested and confirmed to work with **Unity Engine 2022.3.2f1** and above. Unity Versions that have support for Unity's new UI Toolkit and UIElements namespace may be compatible, but please test first.

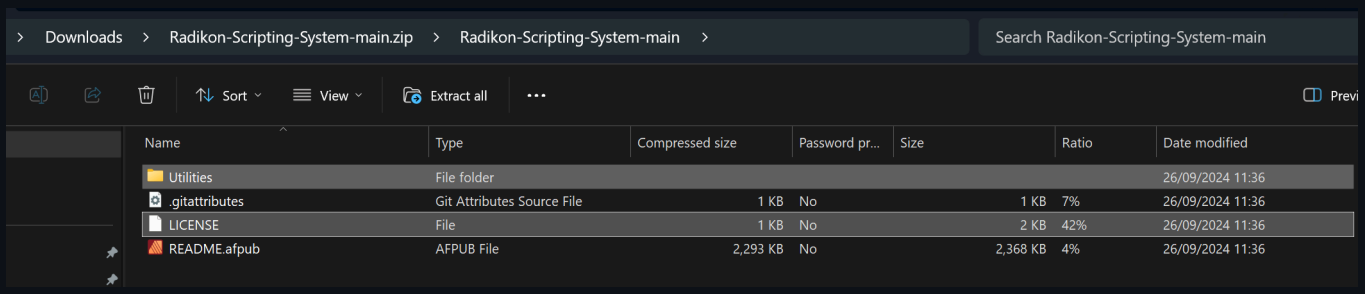
Installation

[**Step 1**] Download the GitHub repository, which will be a .zip file of all files and folders contained within.

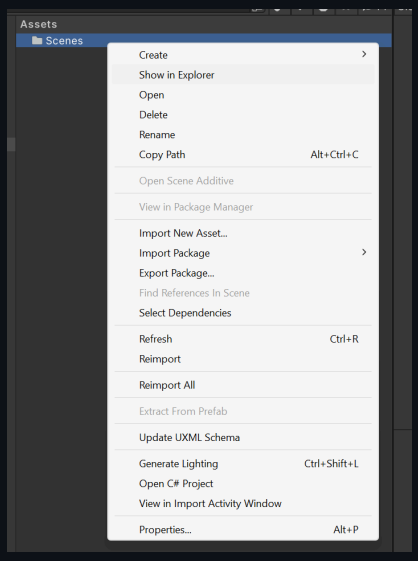


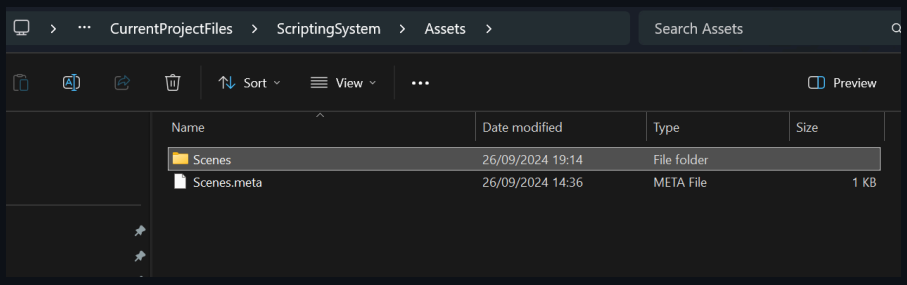


[**Step 2**] Open (or Extract) the .zip file, go into the folder then copy the Utilities Folder and the LICENSE file.

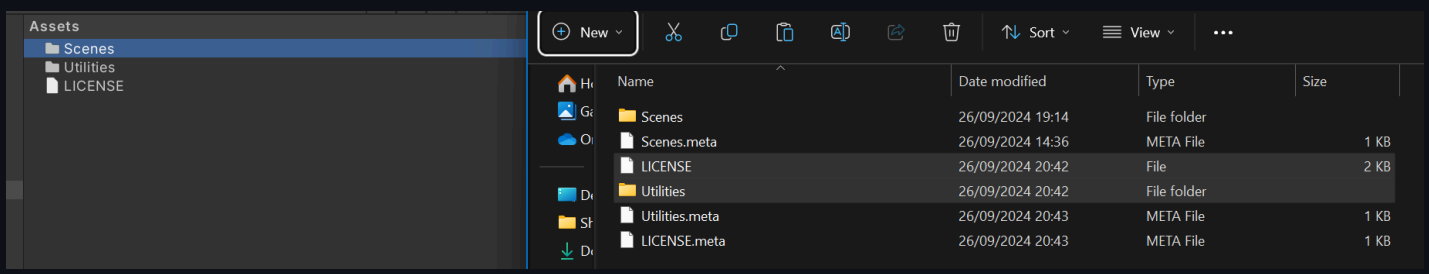


[**Step 3**] Go to the “Project” tab in Unity and open your project’s **Assets** folder in a file explorer.



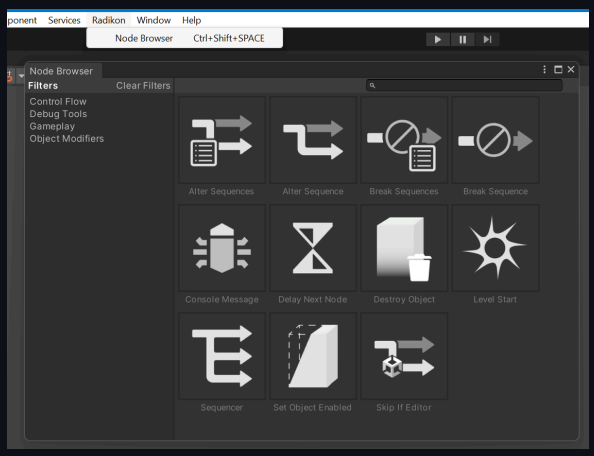


[**Step 4**] Paste the copied files into the Assets folder.

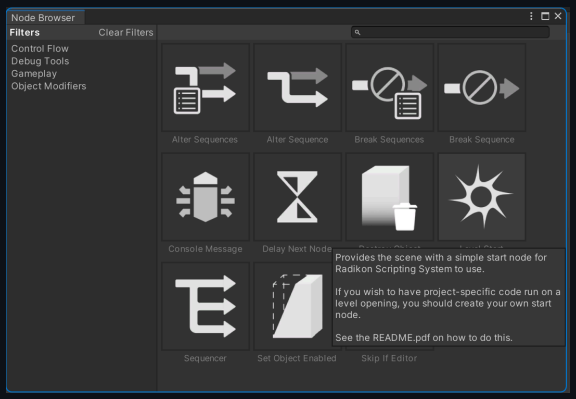


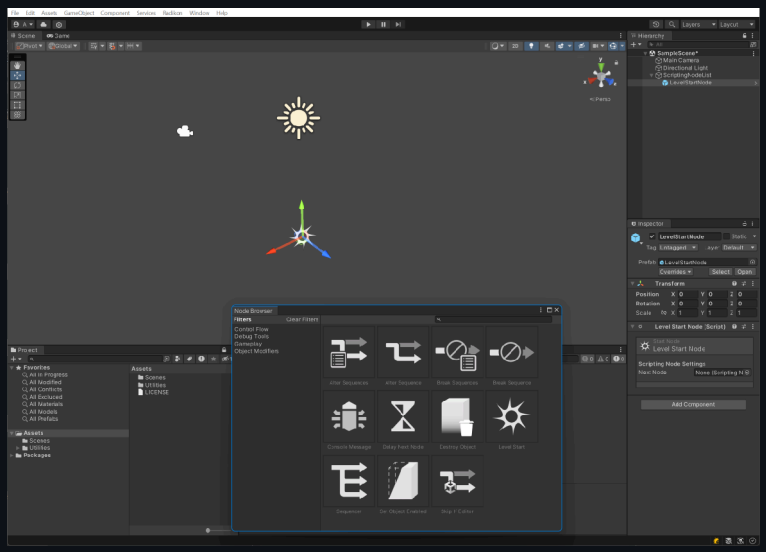
Getting Started

[**Step 1**] Press **CTRL + SHIFT + SPACE** to open the Node Browser at the top of the screen, or hover over the Radikon menu and click Node Browser. A dockable window featuring a list of nodes will appear.

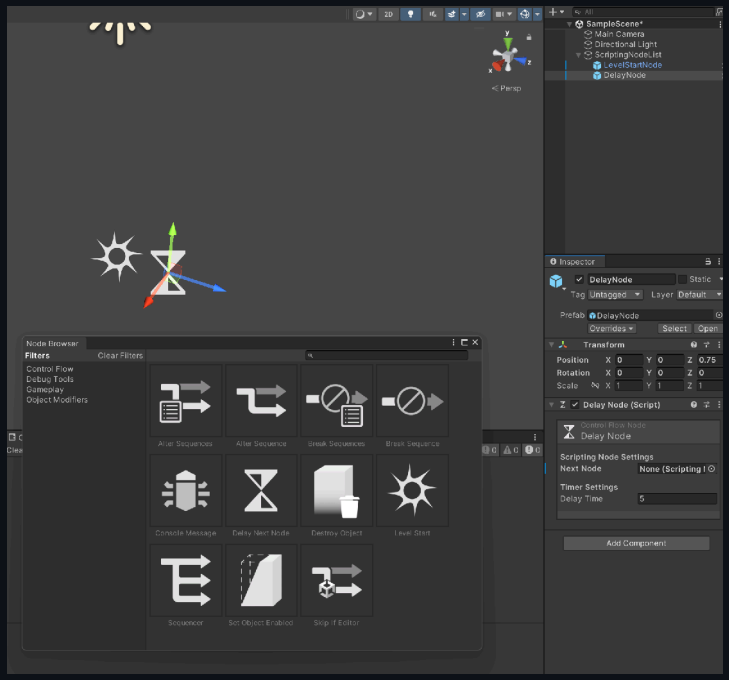


[**Step 2**] Click on the Level Start node. This will create two objects in your scene, **ScriptingNodeList** to hold all future nodes and an instance of **LevelStartNode** as a child of the list.

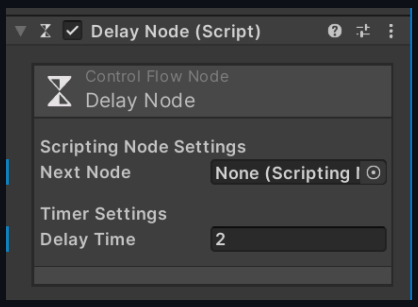




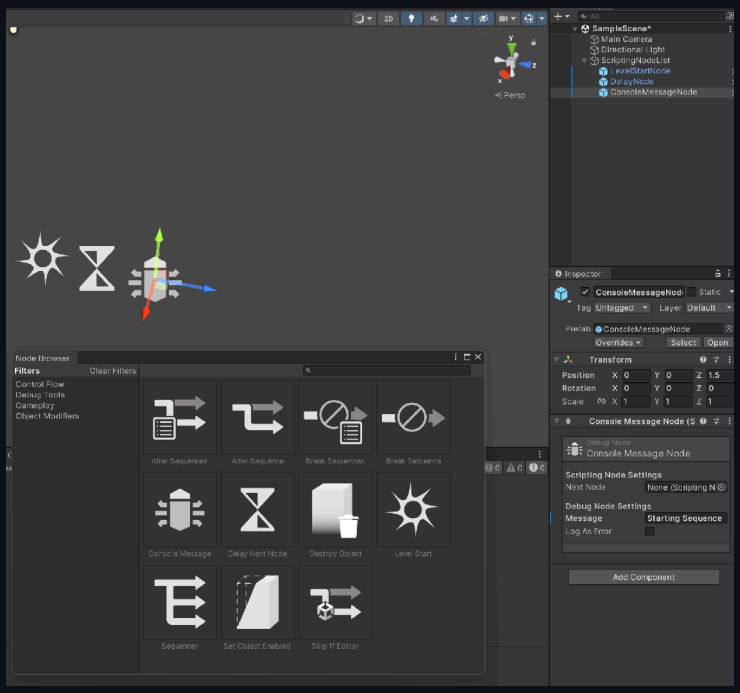
[**Step 2**] Make sure that **LevelStartNode** is still highlighted, then create any other node from the list. For this tutorial, the next node will be a Delay Node. If the nodes' icons overlap in the scene view, feel free to move one of them.



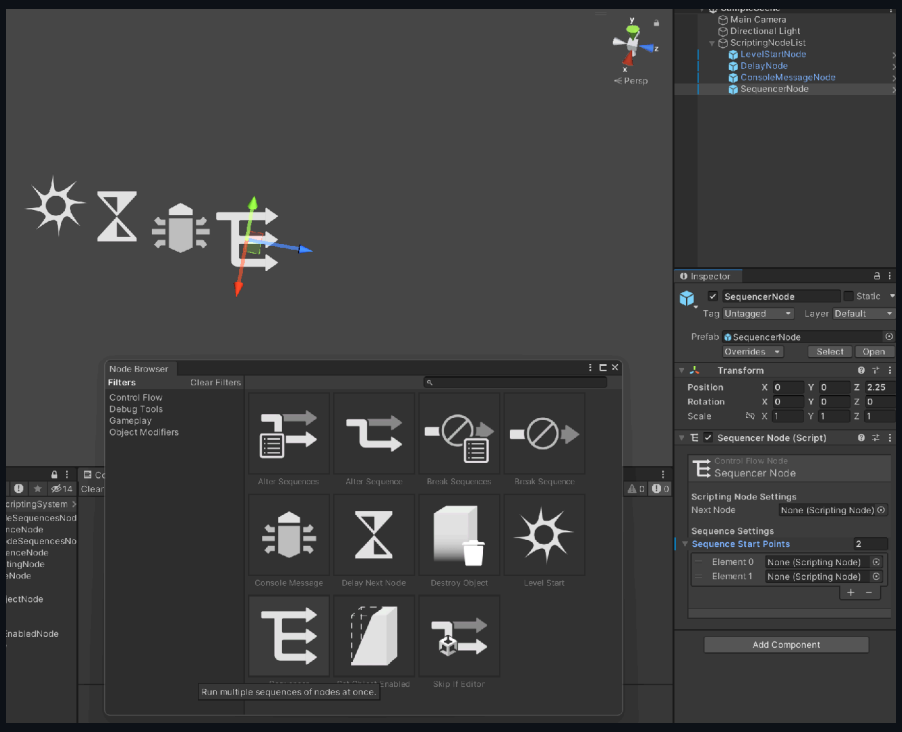
[**Step 4**] Select the **Delay Node** if it hasn't been selected already and change the Delay Time property to 2 seconds.



[**Step 5**] Create a **Console Message Node** and set it's Message property to "Starting Sequence".



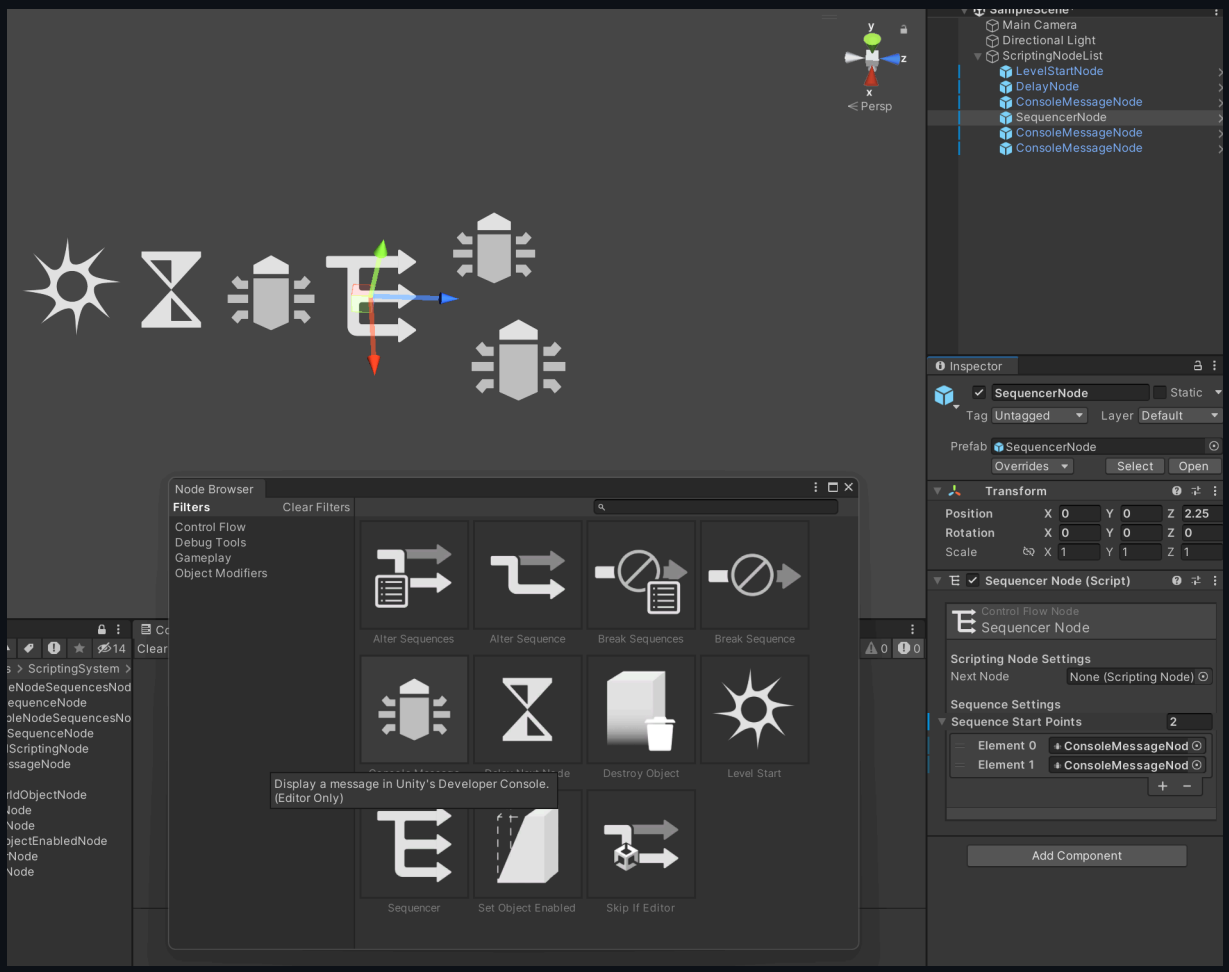
[**Step 6**] Create a **Sequencer Node** and set the length of it's Sequence Start Points list to 2. Then deselect the node.



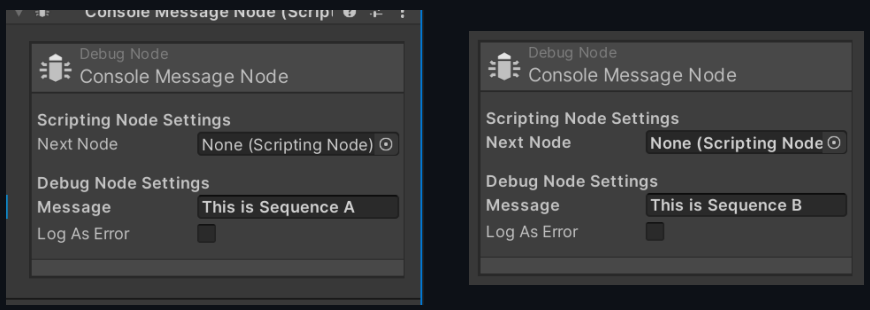
Sequencer Nodes have a unique function. After each sequence's start point is called, the main script tree will continue if there is a Next Node. Leave this empty if you want to split the tree into multiple paths.

[**Step 7**] With Sequence Node deselected, create two more **Console Message Nodes**. By deselecting the Sequencer node, the next node property isn't changed on the Sequencer Node by the Node Browser.

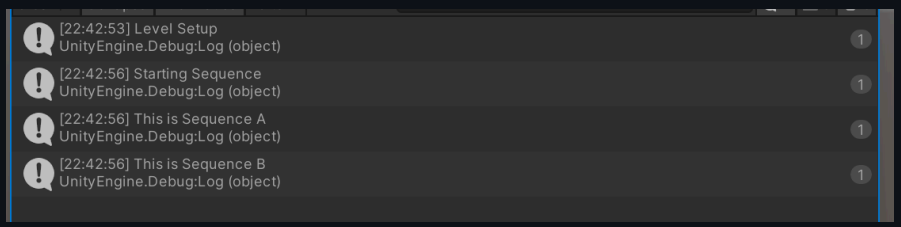
Then re-select the Sequencer Node and link the two new nodes to each start point created in the last step.



[**Step 8**] Set the message for the first Console Message Node to "This is Sequence A" and set the message for the second Console Message Node to "This is Sequence B". If either node has a Next Node assigned, make sure to set the property to None.



[**Step 9**] Test the scene. The first console message, sent by **Level Setup Node**, should display.
This should be followed by the Starting Sequence message, then the sequenced messages. If there is an error, check your nodes' connections for anything out of order.



This is just a quick guide to using the Node Browser for Developers, the next Section is for C# Programmers!

Creating Custom Nodes

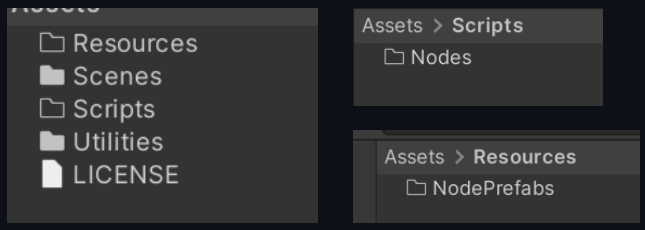
The nodes provided by default from Radikon Scripting System will only do so much on their own and aim to bring useful and simple features into the workflow. This utility is a blank canvas so that you can introduce nodes that leverage your project's mechanics and features for developers to use. Custom nodes act similarly to how **MonoBehaviour** classes do, with the addition of being connected to each other by object references.

When creating a custom node, there are two functions needed to make it work with the scripting system. **Execute** and **Next**. The latter already handled for most use cases and can be overridden if desired, but each custom node needs to override the **Execute** function in order for it to be called upon by the scripting system.

[**Step 1**] Create two new folders in your Assets folder, called **Scripts** and **Resources**.

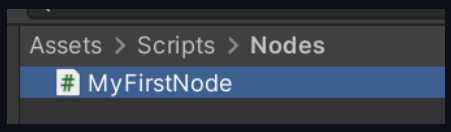
In **Scripts**, create another folder called **Nodes**, which will store the C# scripts for a node.

In **Resources**, create another folder called **NodePrefabs**, which will store prefabs for the Node Browser to use

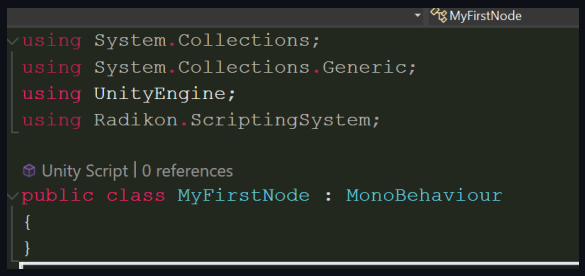


[**Step 2**] Create a new C# script called **MyFirstNode** (or any preferred name) in **Scripts/Nodes**.

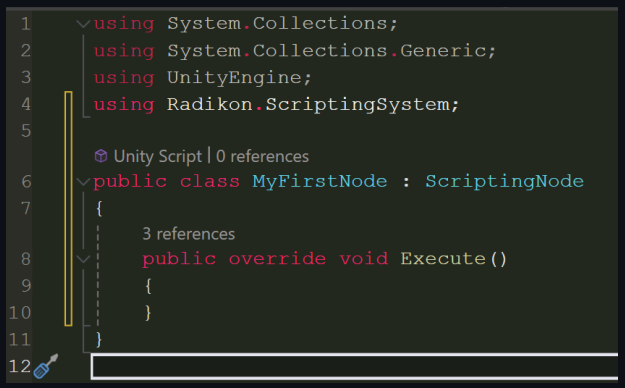
Open the script once the engine has finished compiling.



[**Step 3**] Remove the default Start and Update functions generated by Unity Engine and add a using declaration at the top for **Radikon.ScriptingSystem**. This namespace contains the **ScriptingNode** class and additional attributes.



[**Step 4**] Change the custom node's inheritance from **MonoBehaviour** to **ScriptingNode** and override the **Execute** function.

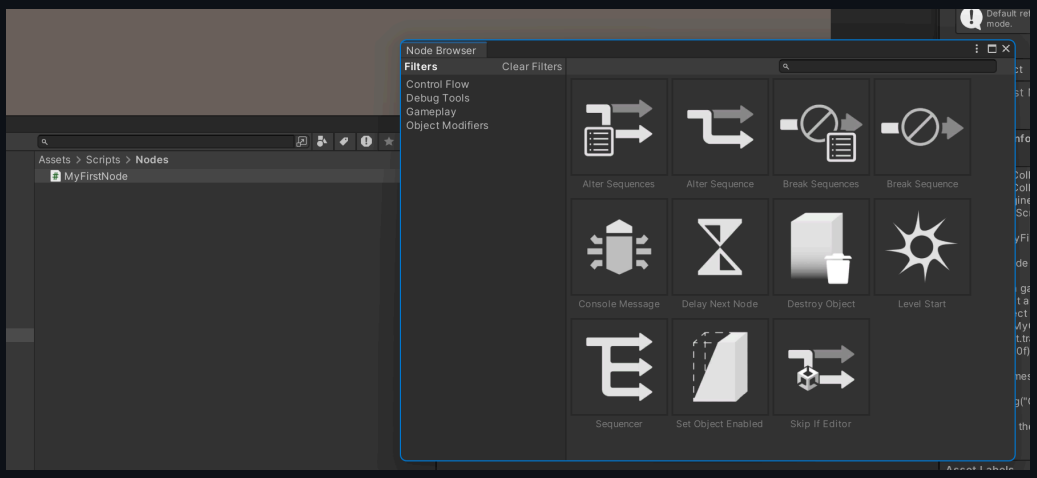


Since **ScriptingNode** is an extension of **MonoBehaviour**, it can access and run everything that a MonoBehaviour can. This includes Triggers and Collisions. For this node we'll be sticking with overriding **Execute** only, but you can call **Next** at any point after Execute is called. **This won't interrupt the scripting system unless the sequence that the node is on is the only sequence currently active in the level.**

[**Step 5**] In the **Execute** function, create a game object called “MyGameObject” and change it’s position to (0, 5, 0). Then, add a debug message for creating the object and call the **Next** function inherited from **ScriptingNode**.

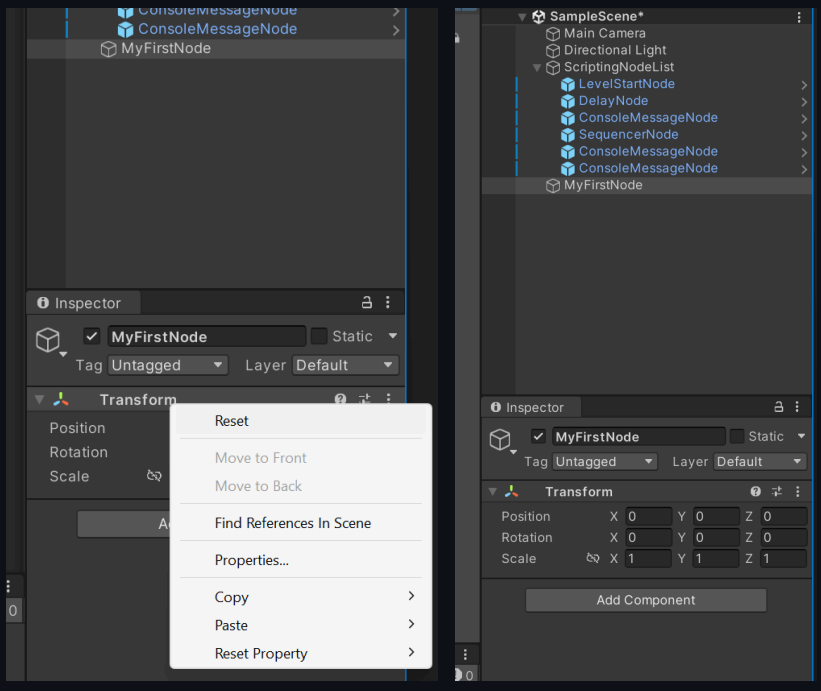
```
Unity Script | 0 references
public class MyFirstNode : ScriptingNode
{
    3 references
    public override void Execute()
    {
        // Create a game object named MyGameObject and set it to 0,5,0 in the air
        GameObject newObject = new GameObject("MyGameObject");
        newObject.transform.position = new Vector3(0f, 5f, 0f);

        // Send a message to the console that it was created.
        Debug.Log("Object Created!");

        // Move to the next node, if there is one.
        Next();
    }
}
```

Save the script and go back to the editor. This forms the node, though the node browser still won’t be able to find it as we’ve not created a prefab with default settings.

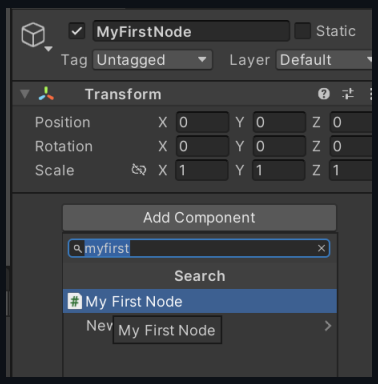


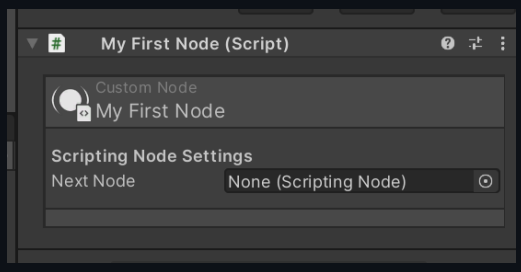
While this may be inconvenient at first, and at scale with a larger project. The reason the node browser relies on prefabs rather than automating the entire process is so that developers can customise the default settings of a prefab in a fixed place.

[**Step 6**] Create a new Game Object called “MyFirstNode” and reset it’s transform. This will become the prefab that the Node Browser will search for.

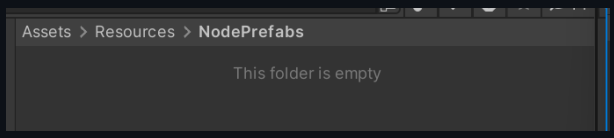


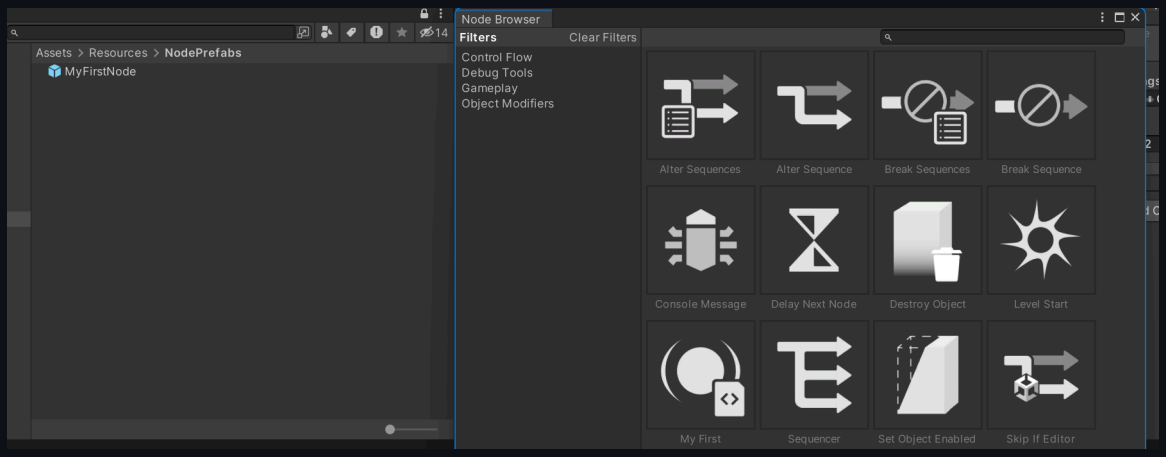
[**Step 7**] Add your node script as a new component to the game object you just created.
The node inspector will look simple, but we'll customise the appearance later using C# Attributes.





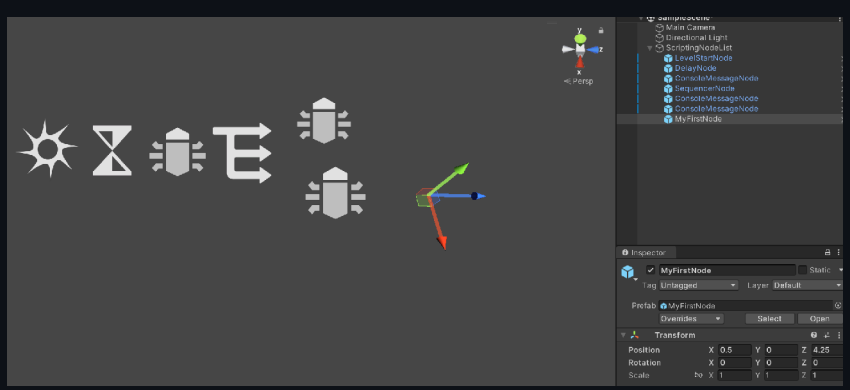
[**Step 8**] Open the **Resources/NodePrefabs** folder you created earlier and drag this game object into the folder.
This will allow the Node Browser to discover your custom node. So long as it's in a folder called **NodePrefabs** within any Resources folder that can be found by Unity's Player, it will be safe to store your node prefab there.

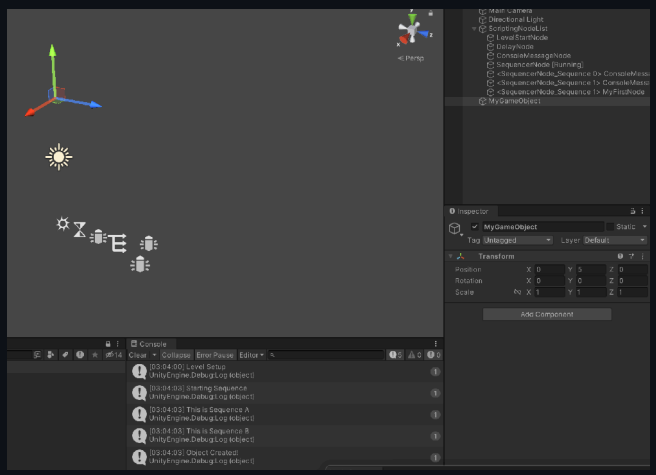




Go to the Node Browser and find your custom node. In this case, it's at the bottom left and has the name "My First" rather than "My First Node". This is because the browser normally filters out "Node" as if it were a suffix if it's at the end of the class name of a custom **ScriptingNode**.

[**Step 9**] If you're following on from the **Getting Started** tutorial, or have a scene setup with a start point already, select a node with no NextNode assigned and then click on the button for your custom node. Once created, test the scene.

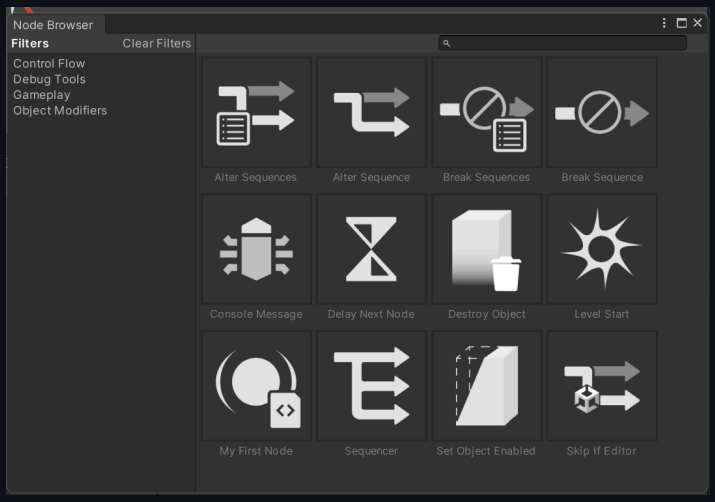




You should see an empty game object spawned into the air if you go back to the scene view during the test.

[**Step 10**] Go back to **MyFirstNode.cs** and at the top of the class definition, add the attribute **BrowserDisplayName** with the display name as “MyFirstNode” and a second parameter as “false”. The boolean controls whether the browser filters out “Node” from the end of a node’s class name. Save the script and go back to the node browser.

```
[BrowserDisplayName("MyFirstNode", false)]
Unity Script (1 asset reference) | 0 references
public class MyFirstNode : ScriptingNode
{
    3 references
    public override void Execute()...
}
```

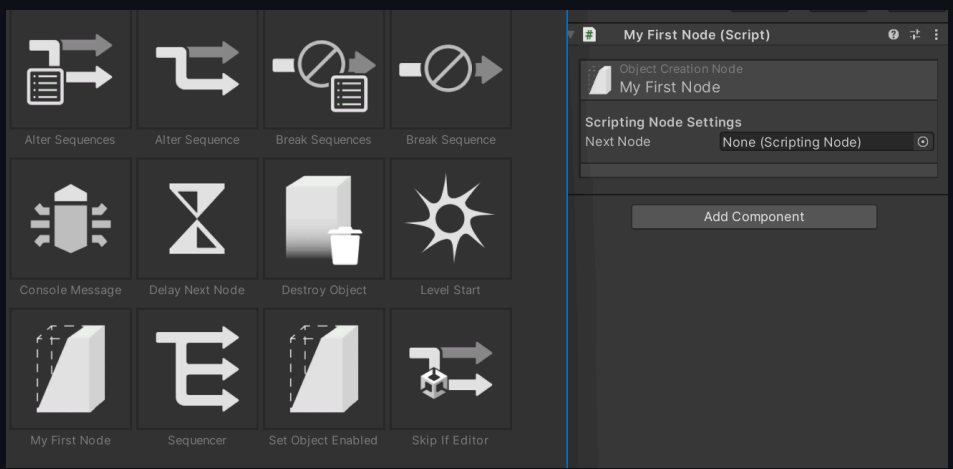


You should see the display name of the node change to now show as “My First Node”. This isn’t the only attribute though! We’ll be adding the rest to the node, to make it easier to discover and understand for developers.

[**Step 11**] Above the previous attribute, add a new one called **NodeType**. This attribute takes a string that will be displayed in the inspector as a description of what the node does, and an icon directory. The icon can be any asset within a Resources folder, so long as the URL points to a **Texture2D**. Since this node deals with object creation, we can use one of the included icons that another object-related node uses.

The Path for this icon is: **ScriptingSystemIcons/SetWorldObjectEnabledNode**

```
[NodeType("Object Creation Node", "ScriptingSystemIcons/SetWorldObjectEnabledNode")]
[BrowserDisplayName("MyFirstNode", false)]
Unity Script (1 asset reference) | 0 references
public class MyFirstNode : ScriptingNode
{
    3 references
    public override void Execute()...
}
```



Save the script and check both the Browser and Inspector for your node. You should see the visuals change. If you see an error, check the path you’re using for the custom icon.

[**Step 12**] Below both previous attributes, add another attribute called **BrowserTooltip**. This attribute only takes a string which is then displayed when hovering over your node in the Browser. For this, we'll describe what happens when **Execute** is called. Add the following description (or your own), then save the script and hover over your node in the Node Browser.

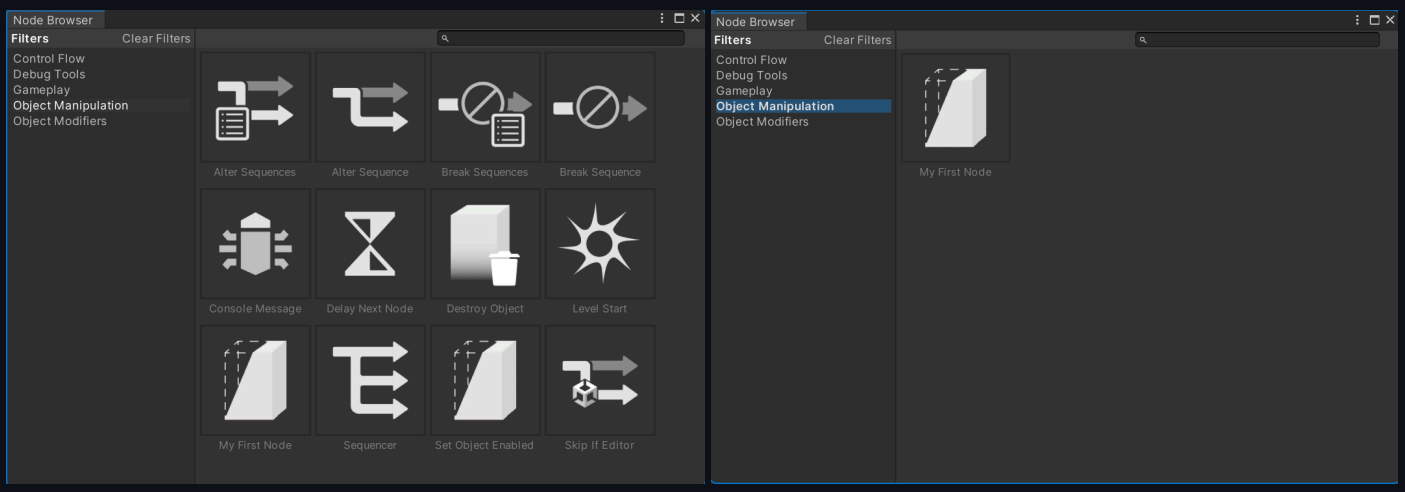
```
[NodeType("Object Creation Node", "ScriptingSystemIcons/SetWorldObjectEnabledNode")]
[BrowserDisplayName("MyFirstNode", false)]
[BrowserTooltip("Creates a game object named \"MyGameObject\", sets it's position and prints a message once done.")]
Unity Script (1 asset reference) | 0 references
public class MyFirstNode : ScriptingNode
{
    3 references
    public override void Execute()...
}
```

If the text isn't properly formatting, or you get errors, make sure to use backslashes when trying to show speech marks in the string.

[**Step 13**] Next, to make it faster to filter the node, we'll add a **NodeBrowserCategory** attribute. The browser filters for these automatically, every time the editor has recompiled scripts and displays them as buttons on the left-hand side. For this, set the category to "ObjectManipulation", then save the script and go over to the Node Browser.

```
[NodeType("Object Creation Node", "ScriptingSystemIcons/SetWorldObjectEnabledNode")]
[BrowserDisplayName("MyFirstNode", false)]
[BrowserTooltip("Creates a game object named \"MyGameObject\", sets it's position and prints a message once done.")]
[NodeBrowserCategory("ObjectManipulation")]
Unity Script (1 asset reference) | 0 references
public class MyFirstNode : ScriptingNode
{
    3 references
    public override void Execute()...
}
```

The category name will be split by camel case, and selecting it should filter out all but your custom node(s) depending on how many nodes have the same category.



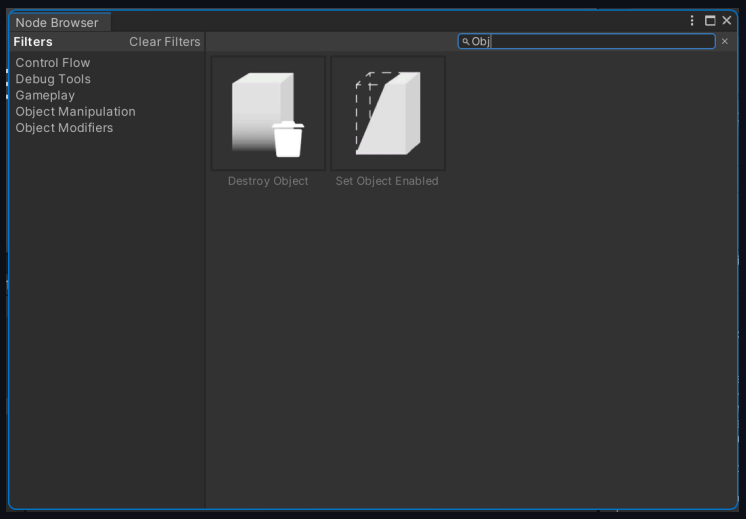
You can select multiple filters at once, if you know all the categories a node has applied. Nodes can have multiple **NodeBrowserCategory** attributes added to them, just remember to not add too many to every node you create or finding specific ones may be harder.

[**Step 14**] The last attribute we'll be adding is the **BrowserKeywords** attribute. This takes any amount of single-word strings which Node Browser attempt to find nodes linked to, if a direct name match cannot be found. This is useful for cases where different developers may use different words for the same node, or only remember how it functions rather than it's name.

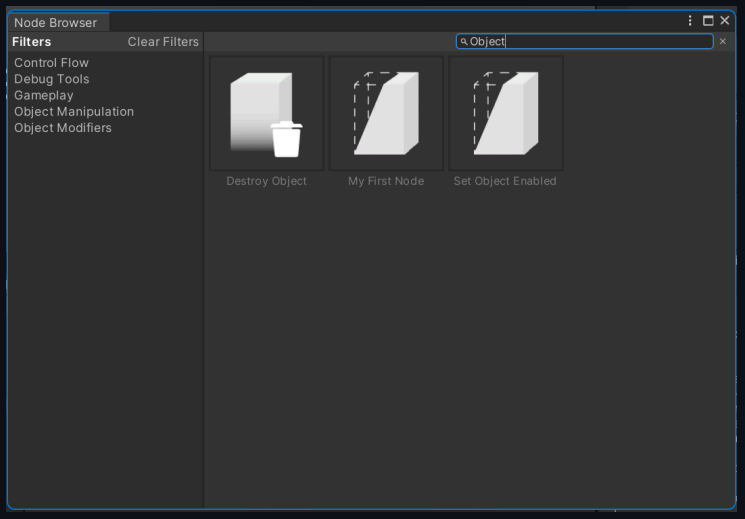
For our attribute, add the Keywords **Object**, **Creation**, **Custom** and **My**. Then save the script and head back to the Node Browser.

```
[NodeType("Object Creation Node", "ScriptingSystemIcons/SetWorldObjectEnabledNode")]
[BrowserDisplayName("MyFirstNode", false)]
[BrowserTooltip("Creates a game object named \"MyGameObject\", sets it's position and prints a message once done.")]
[NodeBrowserCategory("ObjectManipulation")]
[BrowserKeywords("Object", "Creation", "Custom", "My")]
Unity Script (1 asset reference) | 0 references
public class MyFirstNode : ScriptingNode
{
    3 references
    public override void Execute()...
}
```

The Node Browser **will** match partial search queries to a node's class name first, and then attempt to filter by keywords second. So keywords need to be typed in completely if you wish to search by them instead.

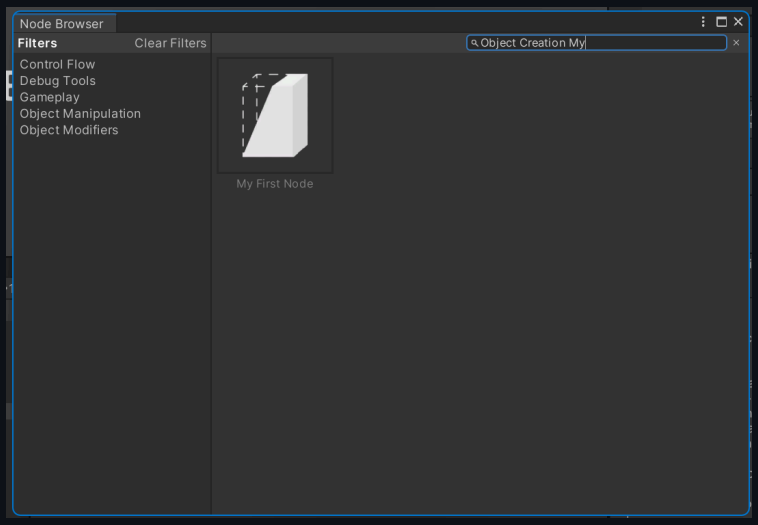


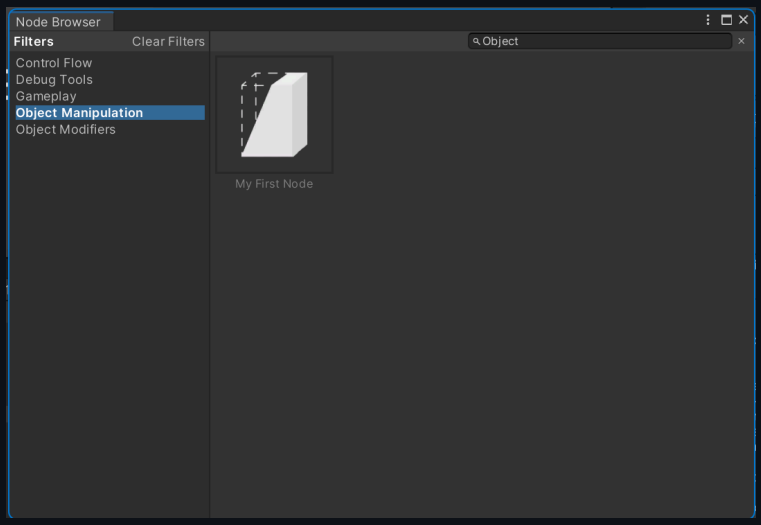
Partial Word - Obj: Only returns nodes with the partial word in their class name



Full Word - Object: Returns nodes with the word in their class name and nodes with the word set as a keyword.

Multiple keywords can also be searched for, in order to make the results more granular. These can also be stacked with category filtering to further increase the precision of the nodes you are looking for.





Please remember to not overload a node with too many irrelevant keywords or too many categories. The more you add, the more that the browser has to filter for and the more inaccurate results may be when searching for nodes in a project that has many of them.

Creating Custom Start Nodes

In some cases with projects, you may prefer to have the game spawn the Player and any other assets only when the level is loaded, instead of keeping them in the editor. While the included **Level Start Node** provides a simple entry point, you can create your own ones by also inheriting **IScriptingSystemEntryPoint**, an interface that adds a new required method to the inheriting node called **StartLevel**.

Note: Only one Start Node should exist per scene, otherwise only the first one that Unity Engine returns is used.

This section expects you to have either created a new node, or followed the tutorial above.

[**Step 1**] Add the interface **IScriptingSystemEntryPoint** to the list of inheritance for your custom node.
This will appear like it's an error at first, but just add a new public function called **SetupLevel** to your node.

```
Unity Script (1 asset reference) | 0 references
public class MyFirstNode : ScriptingNode, IScriptingSystemEntryPoint
{
    3 references
    public override void Execute()...

    2 references
    public void SetupLevel()
    {
    }
}
```

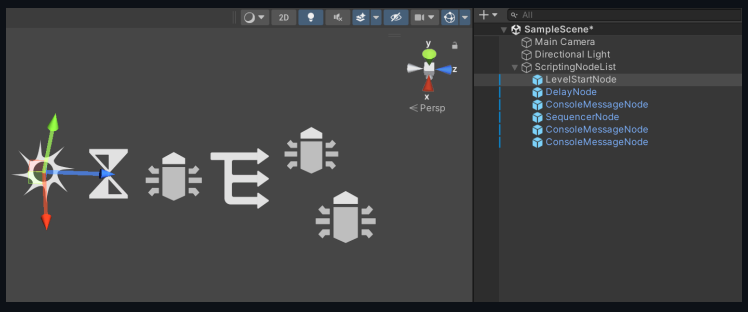
SetupLevel is always called first when a scene is loaded. Any objects can be loaded here for the level in advance.

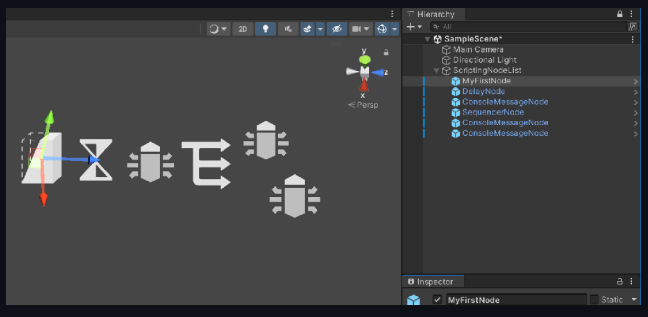
[**Step 2**] In the new **SetupLevel** function, create a game object called "VitalObject" and print a message saying "Level Setup.". Save the script and head back to the Editor.

```
2 references
public void SetupLevel()
{
    GameObject vitalGameObj = new GameObject("VitalObject");
    Debug.Log("Level Setup.");
}
```

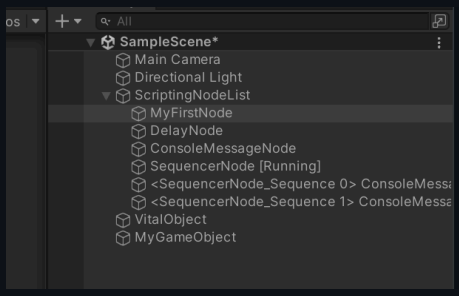
[**Step 3**] If you're using the scene from earlier, or are otherwise using the **Level Setup** node. Delete that and replace it with custom node you made. If it was tested as a normal node previously, make sure to delete other instances of it and reset the NextNode of the node(s) it was linked to. Then, set it's NextNode to the second node in the sequence.

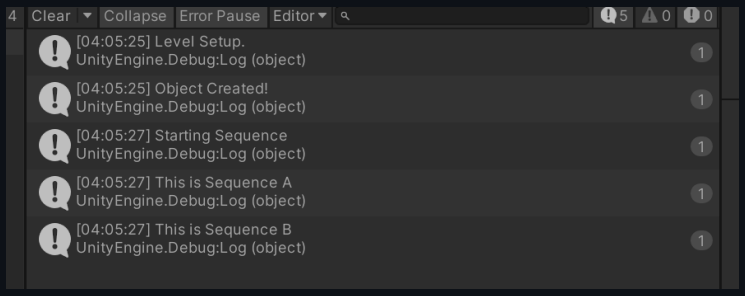
If you're using a new scene then click on your node and the browser will setup the level.





Lastly, test the scene.





The "VitalObject" game object will have been the first thing created for the scene, before the rest could load up.

Setup and Finalizer Methods for Nodes

There may also be times where you wish to not change how **Next** works, or do some setup before **Execute** is called. This can be done in Awake or Start, but two interfaces are provided with Radikon Scripting System just in case those aren't desired. These interfaces are **ISetupMethodProvider** and **IFinalizeMethodProvider** , which provide the methods **SetupNode** and **FinalizeNode** respectively. The use cases for these are extremely limited, but they may prove useful.

When the Scripting System is about to change which node is running, it first checks if the last node has a finalizer method. If the node does, that method is called first for cleanup. Then if the running node has a setup method, it is called before **Execute** so that the node can create any objects ahead of time.

To use either method, the process is much simpler than for creating a custom start node.

To utilise the extra setup method for a custom node, inherit from **ISetupMethodProvider** and add a new public method called **SetupNode**. As seen below:

```
Unity Script (1 asset reference) | 0 references
public class MyFirstNode : ScriptingNode, ISetupMethodProvider
{
    3 references
    public override void Execute()...

    2 references
    public void SetupNode() {}
}
```

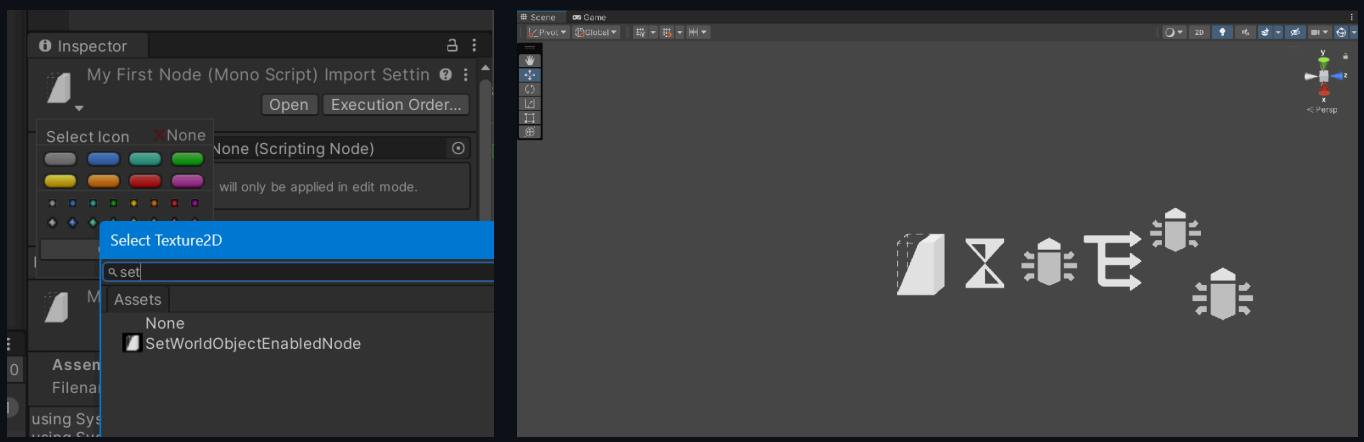
To utilise the finalizer method for a custom node, inherit from **IFinalizeMethodProvider** and add a new public method called **FinalizeNode**. As seen below:

```
Unity Script (1 asset reference) | 0 references
public class MyFirstNode : ScriptingNode, IFinalizeMethodProvider
{
    3 references
    public override void Execute()...

    2 references
    public void FinalizeNode() {}
}
```

The scripting system will handle the rest at runtime, so you don't need to perform any additional node setup past this point.

Custom Node Icons in the Scene View

Lastly, during the tutorials, some readers may have wondered how the custom node icons are being displayed in the scene view. To achieve this, you can set a custom icon for **Mono Scripts**. Click on the C# script you wish to change the icon of and go to it's inspector. The icon in the top left can be changed and you can select a custom Texture2D for it.



Once changed, the new icon will appear in the Scene View, Inspector and Project Explorer. Radikon Scripting System leverages this so that it's own classes look distinctive from standard C# scripts.
