

Table of Contents

[Version History 3](#_Toc441474938)

[Version 1.1 3](#_Toc441474939)

[Version 1.0 3](#_Toc441474940)

[Introduction 4](#_Toc441474941)

[Quick-Step Guide 6](#_Toc441474942)

[Importing the Asset 6](#_Toc441474943)

[Setup 6](#_Toc441474944)

[Using SOM 7](#_Toc441474945)

[SOMValue Attribute 8](#_Toc441474946)

[Keeping SOM up-to-date 10](#_Toc441474947)

[Preferences 10](#_Toc441474948)

[In-depth Guide 12](#_Toc441474949)

[Built-in Modules Guide 12](#_Toc441474950)

[Audio 12](#_Toc441474951)

[Input 12](#_Toc441474952)

[Layers 13](#_Toc441474953)

[Mecanim 13](#_Toc441474954)

[Navigation 14](#_Toc441474955)

[Resources 14](#_Toc441474956)

[Scenes 15](#_Toc441474957)

[Shaders 15](#_Toc441474958)

[Sorting Layers 16](#_Toc441474959)

[Tags 17](#_Toc441474960)

[Custom Modules 17](#_Toc441474961)

[Setup 17](#_Toc441474962)

[Refresh and the xml File 18](#_Toc441474963)

[Preferences 19](#_Toc441474964)

[Limitations and good practices 20](#_Toc441474965)

# Version History

## Version 1.1

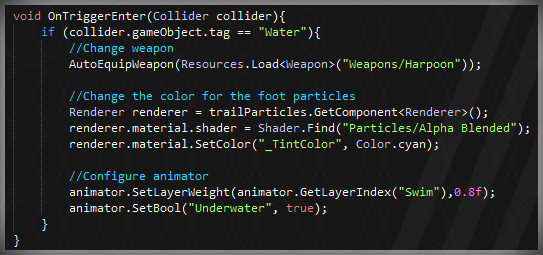
* Fix a bug in which AddModule wouldn’t add recursively inexistent parent Modules.
* Fixed a bug which prevented filters from serializing properly under certain circumstances.
* Resources:
  + Fixed a bug preventing folders to be filtered properly.
  + Folder paths are now added as a constant for each submodule.
* Shaders:
  + Added an option in the preferences to choose to separate built-in from Custom Shaders or not.
* Added SOMValueAttribute to acces SOM values from the Inspector.

## Version 1.0

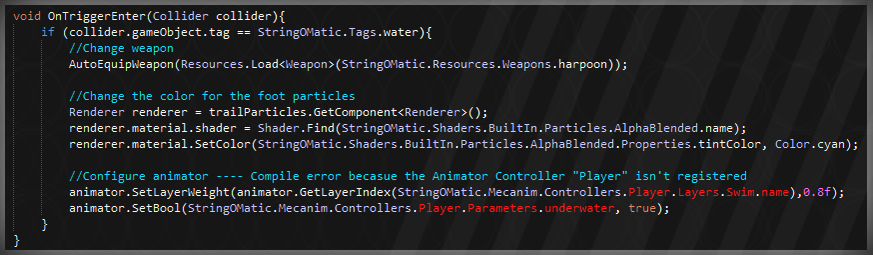
Initial release

# Introduction

Unity's policy for implementing magic strings in almost all of its systems is one the major sources of bugs and headaches, especially for amateurs. Using constant values instead of relying on magic strings heavily reduces the amount of runtime errors.  
  
String-O-Matic scans (SOM from now on) your project and generates constants holding references to all of those magic strings, effectively replacing soft runtime errors by robust compile errors.

Before SOM, anytime you had to compare some game object’s tag or layer, or load any resource, or change some animator parameter’s value, you had to write a hard-coded string. Somewhere in your code, you could easily have something like this:

The main problem with this approach is that the compiler isn’t aware of the existence of, for example, an animator parameter named “Underwater” for that particular animator. Basically, these error-prone magic strings have the evil power to proliferate errors at runtime, which is everything but good.

SOM, however, would make it easy for you to replace the code above with this:

Note how errors are caught at compile-time when, for example, there isn’t any animator parameter named “underwater” for the Player animator. Actually, the problem seems to be that there doesn’t even exist an animator named Player.

Concluding, SOM will let you catch common errors caused by Unity’s magic strings at compile-time, providing you a robust code to work with.

# Quick-Step Guide

Here in presented a quick-step guide on the basics of SOM. Reading this short guide will grant you an overview of the workings of SOM, allowing you to easily set it up and use it.

## Importing the Asset

When importing the Asset, you can place it anywhere you want in your code. Just make sure the internal structure of the folder isn’t modified.

## Setup

Before using SOM, you will have to set it up. To do so, press the shortcut Ctrl+Shift+R on Windows or Cmd+Shift+R on Mac. You can also click on the Refresh menu button under:

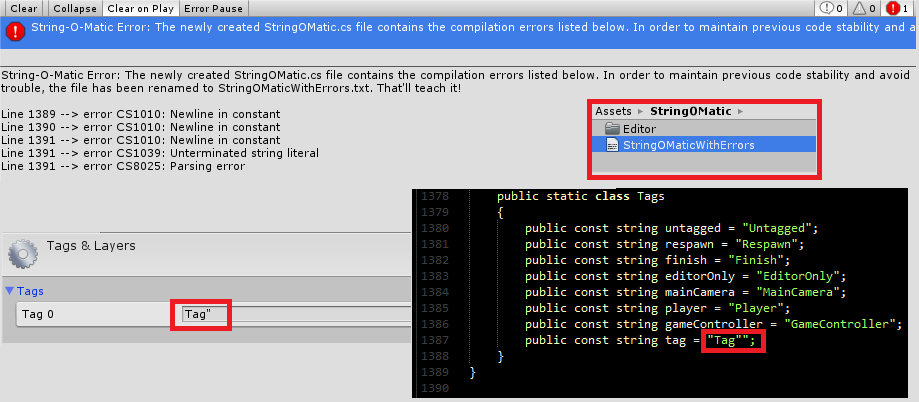


SOM will then scan your project and make a “StringOMatic.cs” file under the SOM root folder.



Now you are ready to use it.

**Note**: It may happen sometimes that the file cannot be created due to some compilation errors. In this case, an error message will be prompted to the console listing those errors. Also, the new file will be created as a plain .txt file that you can use to identify the source of the error.



In this case, the error was caused by a bad naming of one of the tags.

## Using SOM

SOM is constructed under a modular system. Each module adds functionality and is independent of the rest. Built-in modules are located in the String-O-Matic /Editor/Modules folder, but you can also make your own modules (more on that in the in-depth guide).

When refreshed, SOM generates a file containing all the information provided by each of the modules. Those modules are represented as static classes, and the information contained, as string constants. The file structure follows the next format:

StringOMatic

Module1

Constant1

Constant2

…

SubModule1

SubModule2

Module2

Constant1

Constant2

…

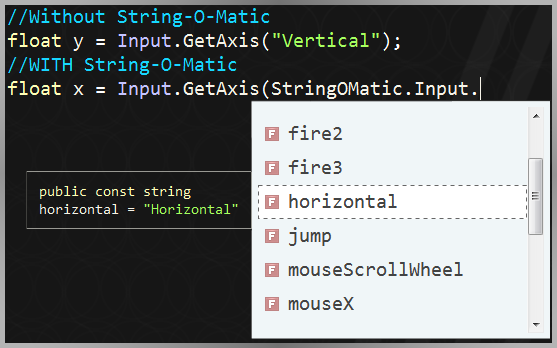
SubModule1

SubModule2

…

Note that the structure is recursive, which means that each module can have any number of submodules and/or constants, which, in turn, may also contain more submodules and constants; following this pattern to infinity and beyond.

So, in order to use SOM, once you are familiar with the structure, you just have to replace your magic strings with the value referenced in the appropriate constant. Here is an example using the Input module:



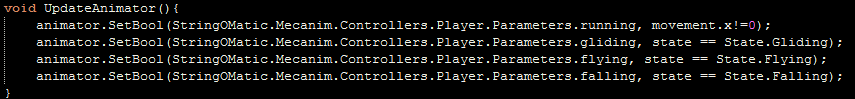
Note, too, that the module’s structure is unique for each one.

## SOMValue Attribute

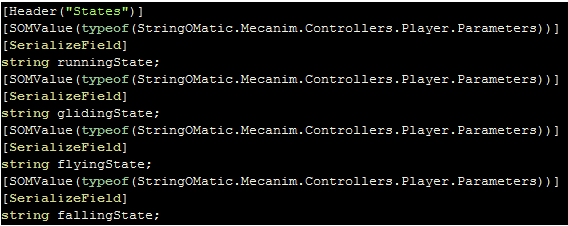
Probably, the easiest way to use SOM is using the SOMValue attribute. This attribute, when applied to a string, lets you reference a SOM constant from the inspector; efficiently highlighting missing references and easily changing its value.

The usage of SOMValueAttribute trades robustness off for flexibility and abstraction, which is a plus in many occasions.

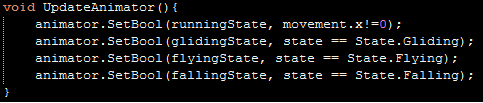
While the code below is robust and leads to compile-time errors, it can sometimes be hard to debug and maintain.



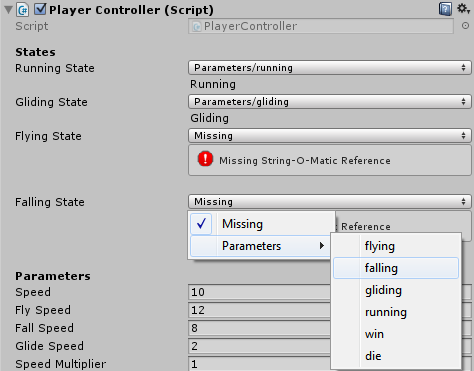
Instead, SOMValueAttribute can be used to add flexibility and debug ease.



Resulting in the following code:



In the inspector, the string will show as a popup listing every possible value. The root for the popup is passed as a parameter to the attribute (In this example, the animator parameters for the player controller) and, if none is passed, it takes SOM as root.



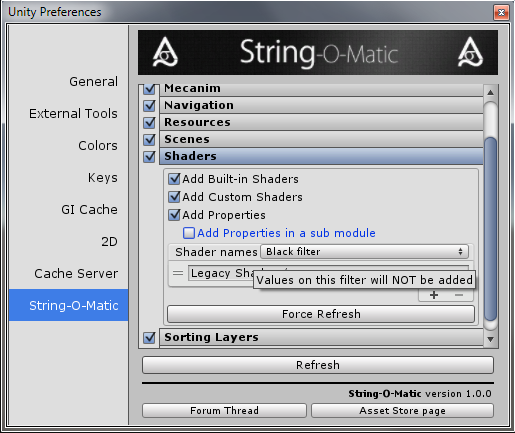
The SOMValue attribute will show the selected constant path, its value below and an error message in case the reference is missing.

## Keeping SOM up-to-date

While SOM provides you with a solid structure and core functionality, it cannot detect changes made in your project’s magic strings (maybe it will someday in a future version), so only you are responsible for refreshing SOM whenever needed. If taken as a habit, it will help a big deal, as errors caused by renaming tags or layers or changing some resource name will be caught at compile-time.

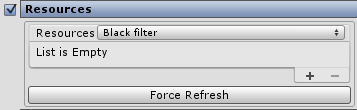
## Preferences

To access SOM preferences, you can either open the Unity preferences (under the Edit/Preferences menu) and open the SOM section; or you can directly access them under the Tools/SOM/Preferences menu (or pressing Ctrl+Shift+C on Windows and Cmd+Shift+C on Mac).



The preferences window will show a list of all the available modules together with a toggle. When a refresh is done, only those modules whose toggle is checked will be refreshed. This way, you can choose which modules to update and which not to.

Also, when a module is clicked, further options for that specific module are shown. In the image above, you can see the options for the shaders module. Most of the options make reference to the way that module hierarchy is structured (more on this options in the in-depth guide). Also, every module includes a “Force Refresh” button which, if clicked, makes a refresh of that module alone.



Some modules offer a Filter List. The way this filter is treated depends on the module itself, but general practices should be followed:

* A black filter indicates exceptions that should NOT be processed.
* A white filter indicates that ONLY those exceptions should be processed.
* Also, values containing forward slashes “/” should be used as a means to separate modules/sub-modules.

As updates come along, new customization features will be added.

# In-depth Guide

Here is presented an in-depth guide on the usage of SOM. Before proceeding, though, make sure you have read the basics presented in the quick-step guide right above.

## Built-in Modules Guide

For each module, a brief description is presented, followed by a graph of its structure. A final section lists Unity API where this code would become useful.

### Audio

The Audio Module lists every mixer in the project, together with its properties and its snapshots.

#### Structure

Audio

Mixers

Mixer1

Name

Parameters

Param1

Param2

…

Snapshots

Snapshot1

Snapshot2

…

Mixer2

…

#### Relevant API

This module is useful for modifying audio mixers’ parameters through scripting.

-AudioMixer.SetFloat

-AudioMixer.GetFloat

-AudioMixer.FinsSnapshot

### Input

The Input Module offers the names of every input axis configured through the Input Manager.

#### Structure

Input

Axis1

Axis2

…

#### Relevant API

This module is to be used in conjunction with the Input class.

-Input.GetAxis

-Input.GetButton

### Layers

The Layers Module offers the name of all the layers configured in the Tags & Layers configuration panel.

#### Structure

Layers

Layer1

Layer2

…

#### Relevant API

You will mainly be using this module to get the LayerMask for a given layer name.

-LayerMask.NameToLayer

### Mecanim

The Mecanim Module lists every animator controller in the project, together with its layers, parameters, states and sub-states machines.

Please, note that for every sub-state machines, its states and sub-state machines are added recursively. Also, Animator Controllers under any Editor folder are ignored.

#### Structure

Mecanim

Controllers

Controller1

Name

Parameters

Param1

Param2

…

Layers

Layer1

Name

States

State1

State2

…

State Machines

StateMachine1

Name

Recursive States

Recursive State Machines

StateMachine2

…

Layer2

…

Controller2

…

#### Relevant API

One of the most common situations in which this module can become useful is when changing some animator parameter’s value or the weight of some of its layers.

-Animator.SetInt

-Animator.GetBool

-Animator.GetLayerIndex

### Navigation

The Navigation offers a list of every Navigation Area name.

#### Structure

Navigation

Area1

Area2

…

#### Relevant API

This module is to be used with any of the methods of the NavMesh class.

-NavMesh.GetAreaFromName

### Resources

The Resources Module lets you access any of the resources in your Resources folders by path and name of the resource. The internal structure of this module mimics the hierarchical folder structure of your Resources folder. Plus, it adds the path of each folder.

When using the filter, use “/” to indicate folder paths. For example:

* “Weapons/” means the whole weapons folder is filtered.
* “Weapons/Debug/” means the whole weapons/debug folder is filtered.
* “Weapons/flamethrower” means only that weapon is filtered.

Note that Editor folders are completely ignored.

#### Structure

Same as that of your Resources folder

#### Relevant API

Basically, you will want to use this module when calling any of the Resources API.

-Resources.Load

-Resources.LoadAll

### Scenes

The Scenes module offers the name of every scene added to the project’s build settings.

#### Structure

Scenes

Scene1

Scene2

…

#### Relevant API

This module is useful when loading a new scene by its name.

-Application.LoadLevel

### Shaders

The Shaders Module offers the name every shader in the project sorted by its category, along with their parameters.

Built-in modules for the used version of Unity are included, and also the custom ones found in the project.

Note that the name of the shader file is not relevant, only the name given within the code.

A filter can be added specifying a full category or a concrete shader name.

* “Standard” will filter the Standard shader.
* “Legacy Shaders/” will filter the whole category.

Note that shaders placed under Editor folders are completely ignored.

#### Structure

Shaders

Built-in

Category1

Shader1

Name

Properties

Property1

Property2

…

Shader2

…

Category2

…

Custom

#### Relevant API

It is sometimes useful, especially when creating a procedural material, to load a shader manually by its name; or change some material properties at runtime. In any of those cases, this module can prove itself helpful.

-Shader.Find

-Material.SetFloat

### Sorting Layers

The Sorting Layers Module offers the name of all the sorting layers configured in the Tags & Layers configuration panel.

#### Structure

Sorting Layers

Sorting Layer1

Sorting Layer2

…

#### Relevant API

You will mainly be using this module to set a Renderer’s sorting layer.

-Renderer.sortingLayerName

### Tags

The Tags Module offers the name of all the tags configured in the Tags & Layers configuration panel.

#### Structure

Tags

Tag1

Tag2

…

#### Relevant API

You will mainly be using this module to compare some gameobject’s tag.

-GameObject.CompareTag

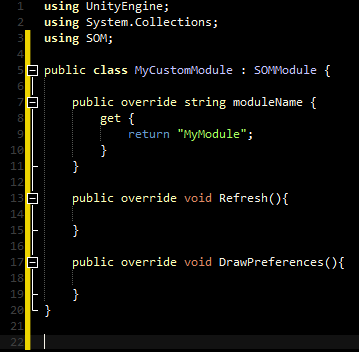
## Custom Modules

SOM also allows you to make custom modules. You can use these together with other assets or with whatever you find useful.

Although I find that the best way to learn how to make custom modules is by playing with the built-in ones and taking them as an example, here I present a quick guide.

### Setup

Use the code below as a framework. Needless to say, the file must be placed under the Editor folder in your project.



It’s essential that your module inherits from SOMModule, which is the base class for all modules. That class, and many other we will be using, is placed under the SOM namespace, so you will need to include that.

Your class also needs to override the ”moduleName” property, returning the name of the module. This will be using mostly for display and as keyName for storing settings. The “Refresh” method needs to be overridden too, and an optional “DrawPreferences” method too.

### Refresh and the xml File

When a refresh is triggered, the “Refresh” method of every module is called. Here is where the action happens and the core functionality of the module takes place.

You may have noticed that an .xml file is created along with the “StringOMatic.cs” file when a refresh is done. That is because modules are first generated in a xml format and then converted to classes and constants.

So, in this method, you will need to address to that XML file in the project. A full interface for dealing with that file is available through the SOMXmlHandler wrapper class. The class offers some methods for dealing with modules, sub-modules and constants.

The XML file mimics the structure of a directory system, that is, every submodule needs to follow a forward slash “/”. Most of the methods in that wrapper ask for a path as an argument, which is where you need to perform the operation. For example, the method SOMXmlHandler.AddModule(“Weapons/Modern”) will add a module called “Modern” as a sub-module of the “Weapons” module.

Note that many exceptions can be thrown caused by a wrong xml file manipulation. You should be aware, for example, that when executing the command explained above, an exception will be thrown if the “Weapons” module does not exist. Note, too, that when a Refresh is triggered on a module, that module is previously cleaned in the xml file.

### Preferences

The “DrawPreferences” is optional, but, id added, you should make sure to call “base.DrawPreferences()” to include the “Force Refresh” button.

This method should be populated with GUI calls, as it will be called on the OnGUI method of the Preferences Window when the module is selected.

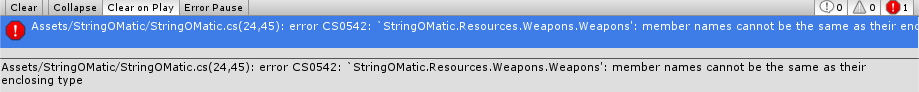
If you want to use a Filter List as those of Resources or Shaders modules, use the SOMUtils.FilterList class.

You may have noticed, too, that a file called SOMPreferences.asset is created along with the .xml file. This file, which is controlled by the SOMPresefernces class is responsible for storing values the same way EditorPrefs or PlayerPrefs do, except that it is editor-only, project-dependant. The API is not exactly the same yet, but you can still use it to store values for your preferences.

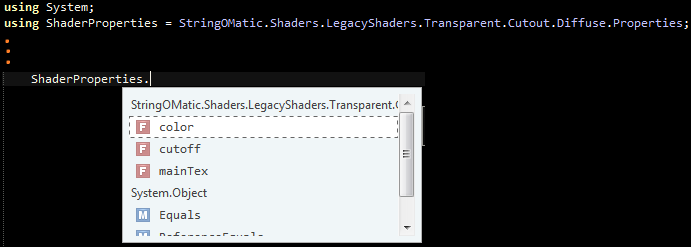
## Limitations and good practices

There are some limitations and good practices one should be aware of when using SOM:

* You should be specially aware with the Shaders and Resources module, as they are more likely to produce the error CS0542. This is a limitation with C# in which nested types cannot have the same name as their enclosing types.



As a rule of thumb, and until further updates (for which I will probably provide a solution), try not to have Resources folders with the same name as their parent folders; and try to avoid having nested shader categories with the same name.

* Adding a constant name to the xml with containing spaces will not prompt any error, contrary to what happens with modules; but they will throw a compilation error when refreshing; so try by all means having constant names that are valid identifiers. Built-in modules prevent these by calling SOMUtils.NicifyConstantName, which camelCases the given string, removes some special characters and prevents having number-only identifiers (this one is done by placing an “n” before a number-only string).
* The usage of the SOMValue attribute is highly recommended. However, when “hardcoding” the reference to a SOM value, in order to avoid repeatedly referencing the same Module and having long references in your code; I encourage you to make use of the “using” directive.
* When using SOMValueAttribute, keep in mind that the string only stores the value of the constant reference. It’s path (the route you see in the popup) is deduced in the editor and does not affect the outcome of the value. Thus, if you have multiple constants containing the same value, it may happen that the wrong path shows in the popup. That does not matter, however: as the important thing is the value itself.