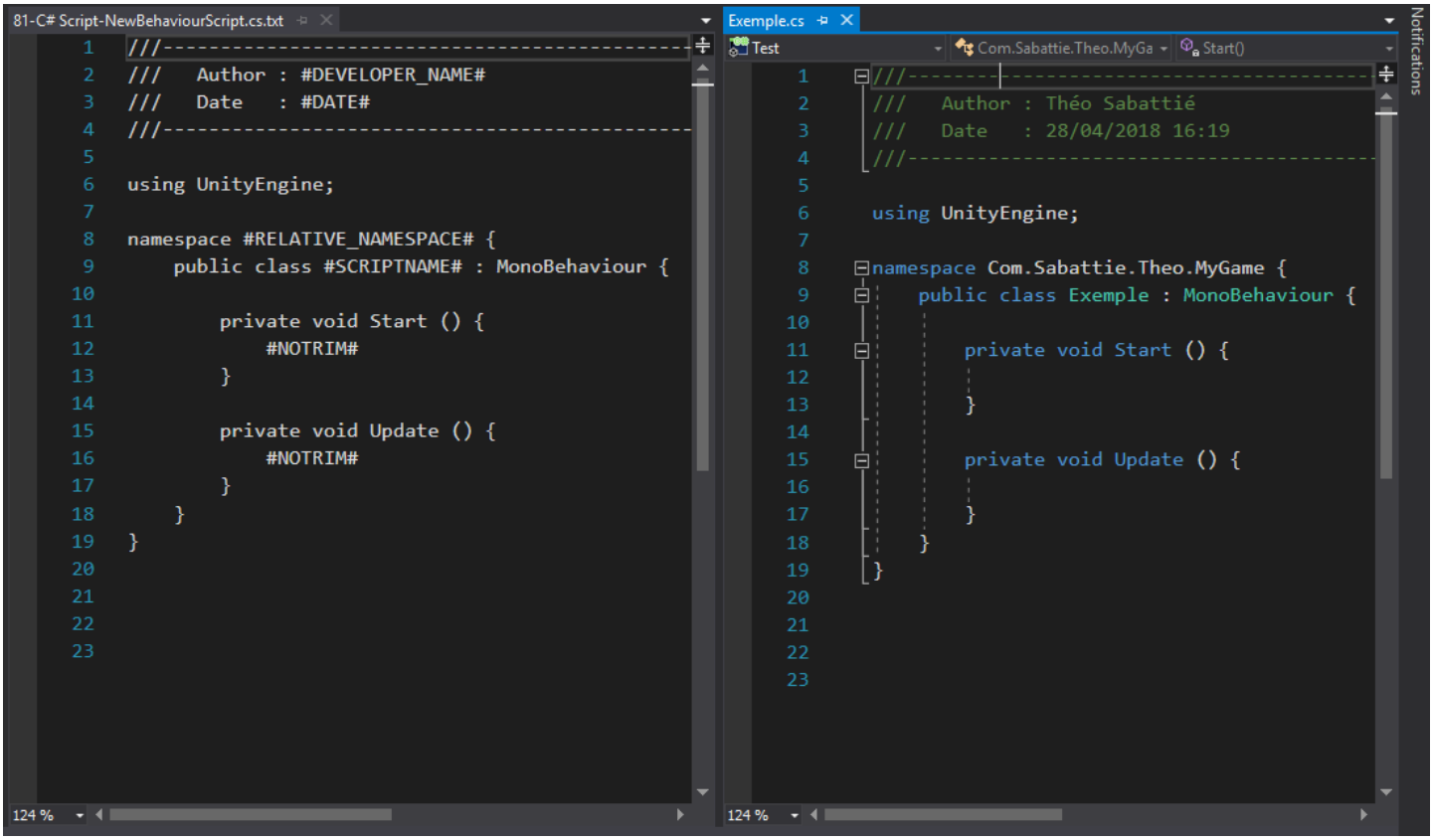
Table of Contents

Script Template Settings



Script Template Settings allows you to add your own script template keywords and use them for the default templates Unity or your script templates.

With this extension, your can add namespace, author, date, (...) to your new script automatically on creation.

The Keywords values can be serialized per project or globally for all your project, they can also be computed (Date, ScriptNumber, ...). You can write your keywords to make your own way to compute value.

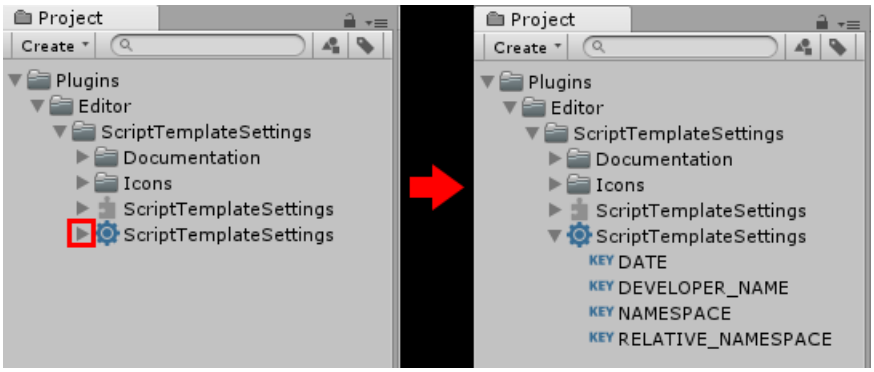
Unity Asset store

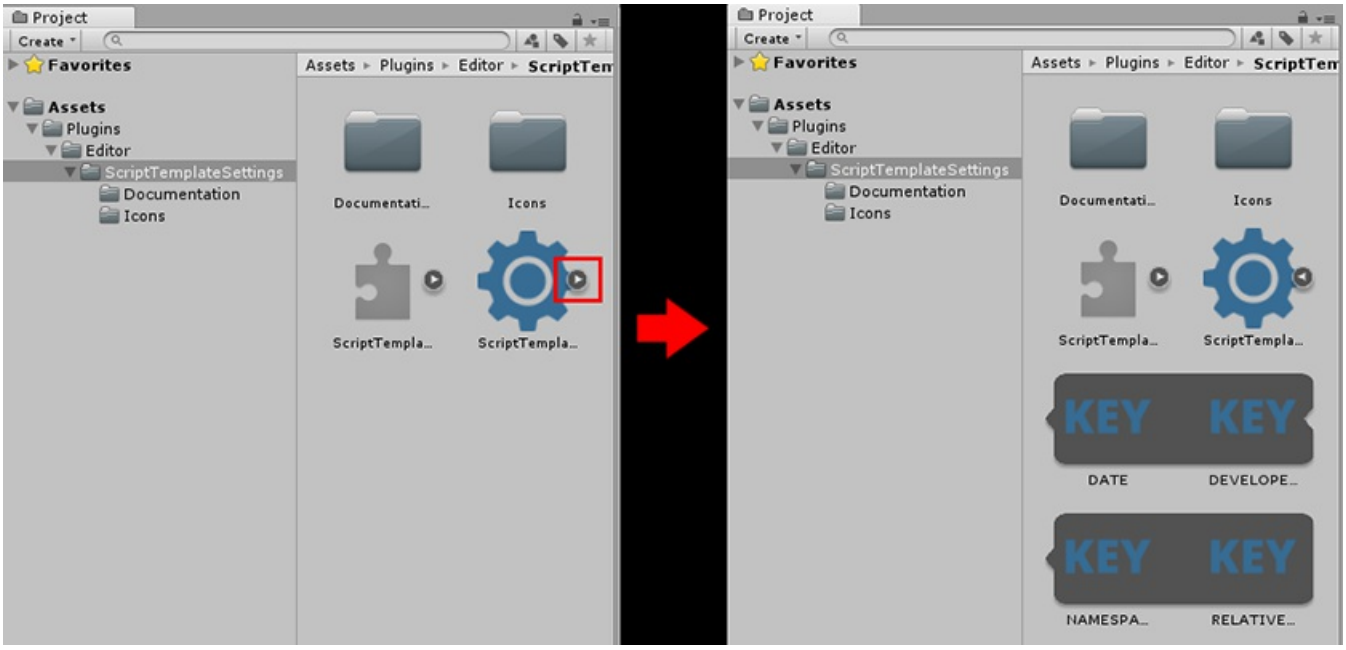
Forum

Twitter contact

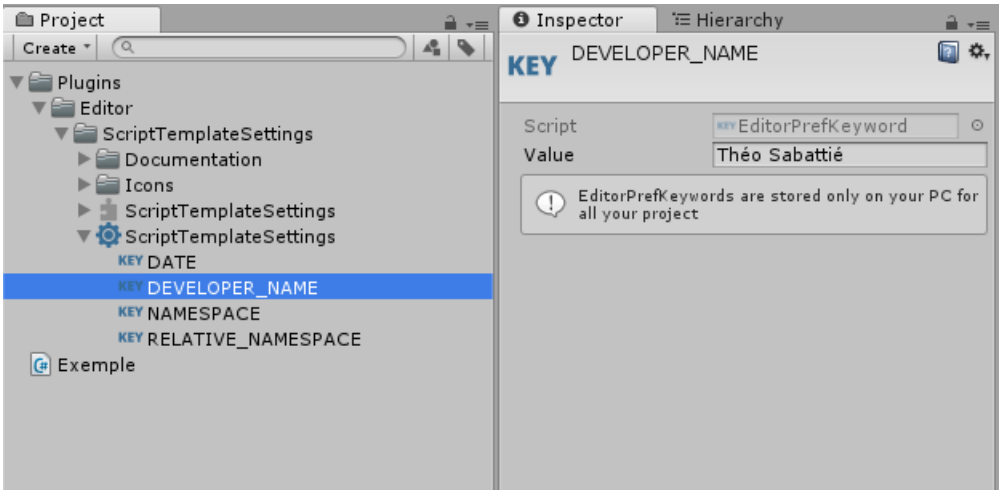
Basic Setup

By default 4 keywords are created : **DATE**, **DEVELOPER_NAME**, **NAMESPACE**, **RELATIVE_NAMESPACE**. They are on the ScriptTemplateSettings asset file.



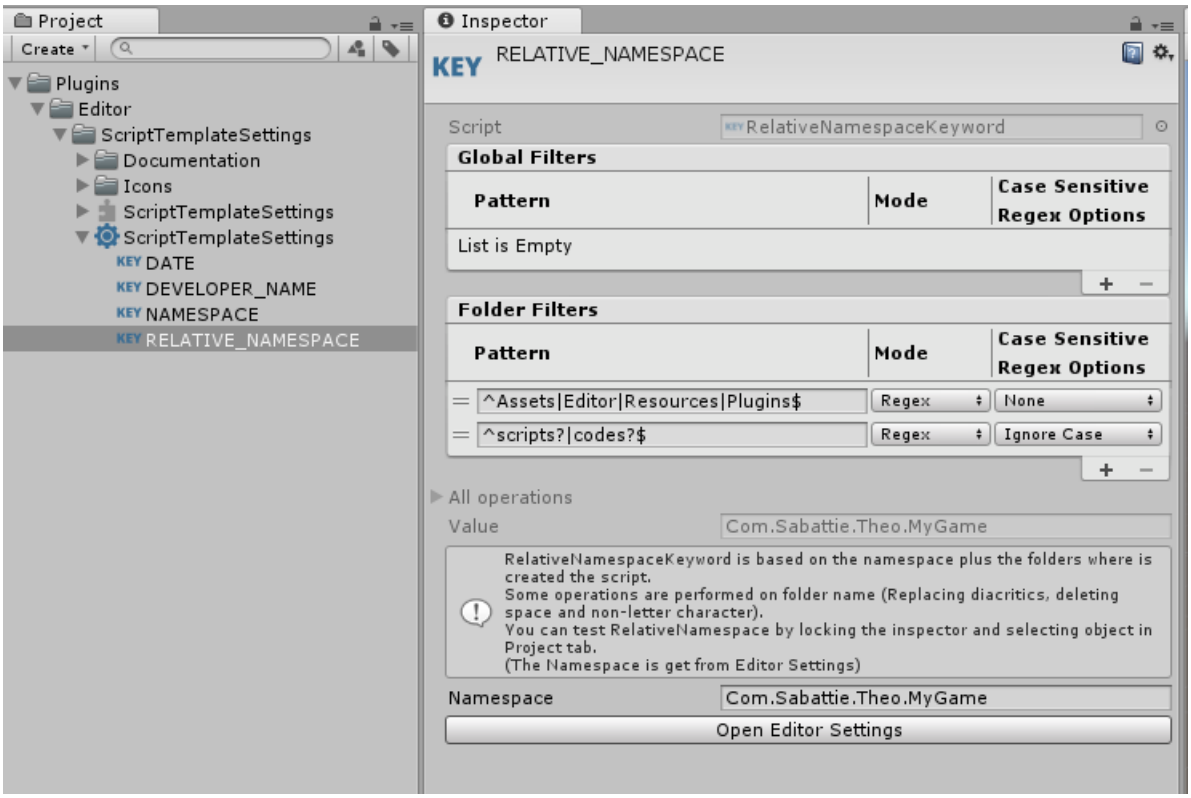


Configure the keyword **DEVELOPER_NAME** by selecting it and filling the Value field :



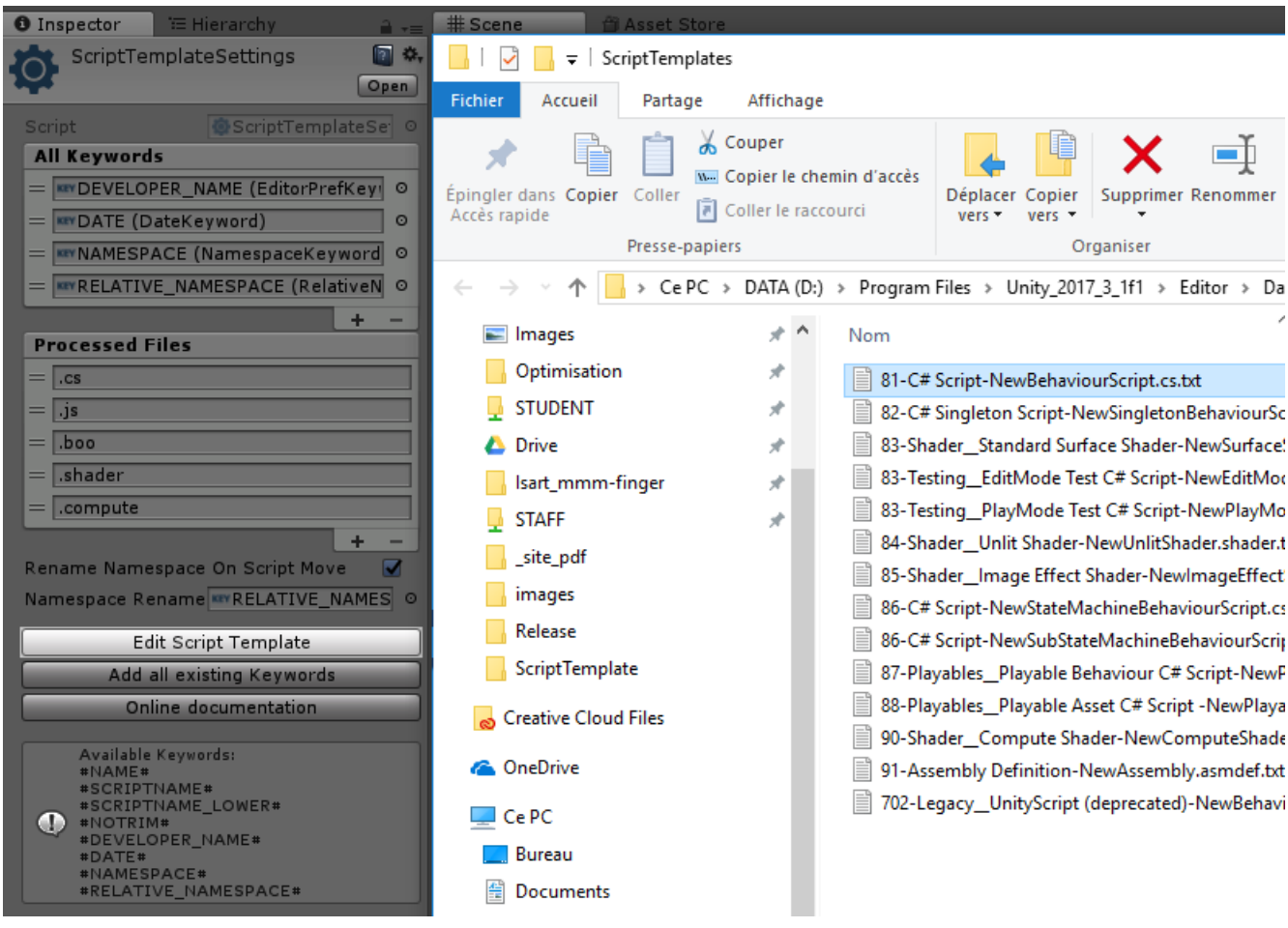
Note : **DEVELOPER_NAME** is a keyword with personnal value. The value is stored only on your computer.

Configure the keyword **RELATIVE_NAMESPACE** by selecting it and filling the Namespace field :



RELATIVE_NAMESPACE is a keyword computing the relative namespace using namespace plus a combinaison of folder where the script is placed. Some filters allow to ignore specials folders. More Informations

Edit the Unity basic template by selecting ScriptTemplateSettings and clicking on "Edit Script Template" button :

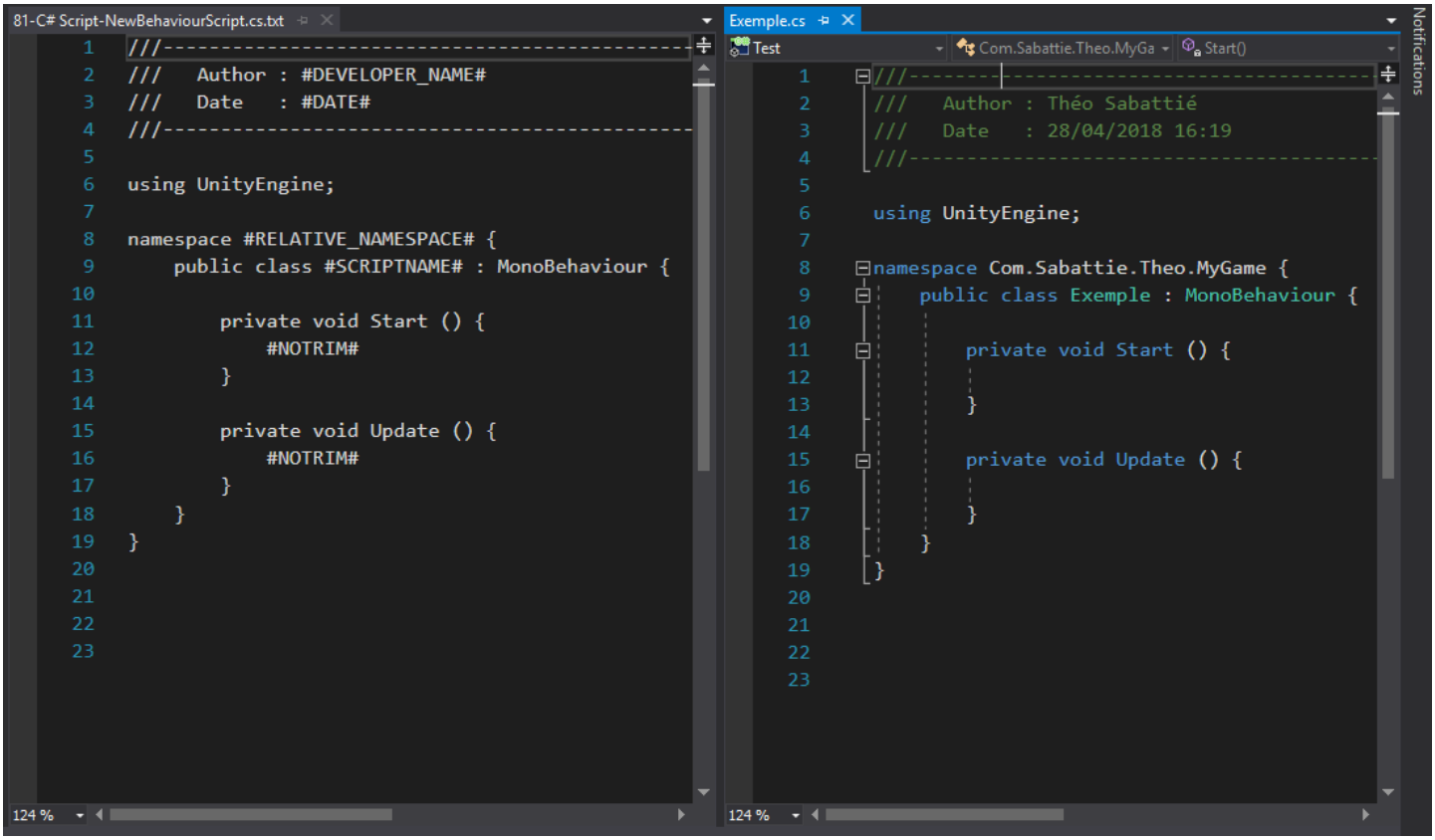


Open 81-C# Script-NewBehaviourScript.cs.txt
And copy paste the template below : (or setup your own template style)

```
///---------------------------------------------------------------
///   Author : #DEVELOPER_NAME#
///   Date   : #DATE#
///---------------------------------------------------------------

using UnityEngine;

namespace #RELATIVE_NAMESPACE# {
    public class #SCRIPTNAME# : MonoBehaviour {

        private void Start () {
            #NOTRIM#
        }

        private void Update () {
            #NOTRIM#
        }
    }
}
```

Save and close the template.
All the keywords will be replaced by the values.
DATE is a keyword with a computed value. You can edit the date format by selecting **DATE** keyword and updating the field Date Format.

Now, it's time to check if the setup is correct :
Create a script and open it.
You should get the result at the right (with your datas) :

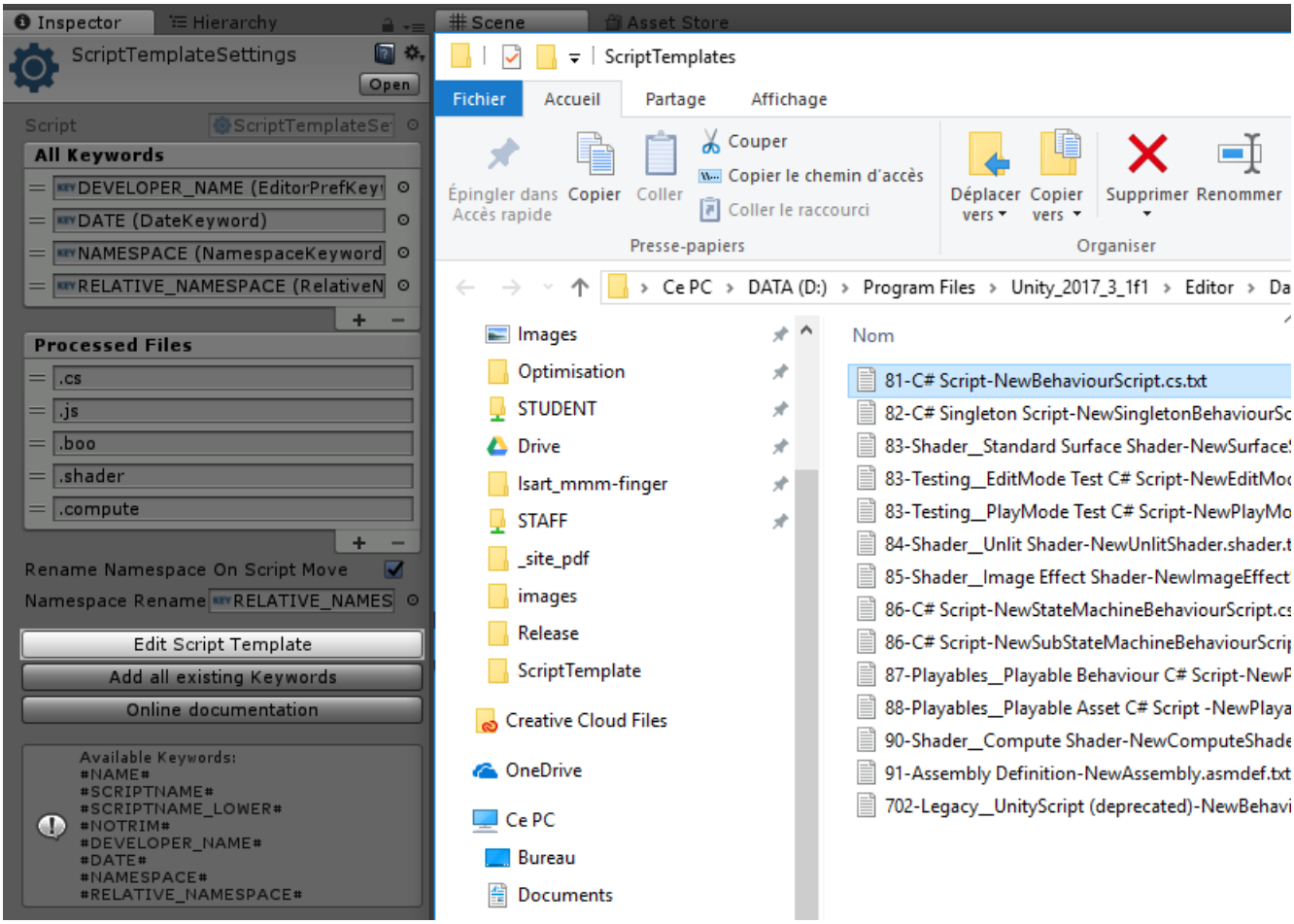


Note : There is ModificationProcessors to update automatically namespace when you move a script and to update automatically type when you rename a script.
Be careful ! That does not update all the other scripts referencing it.

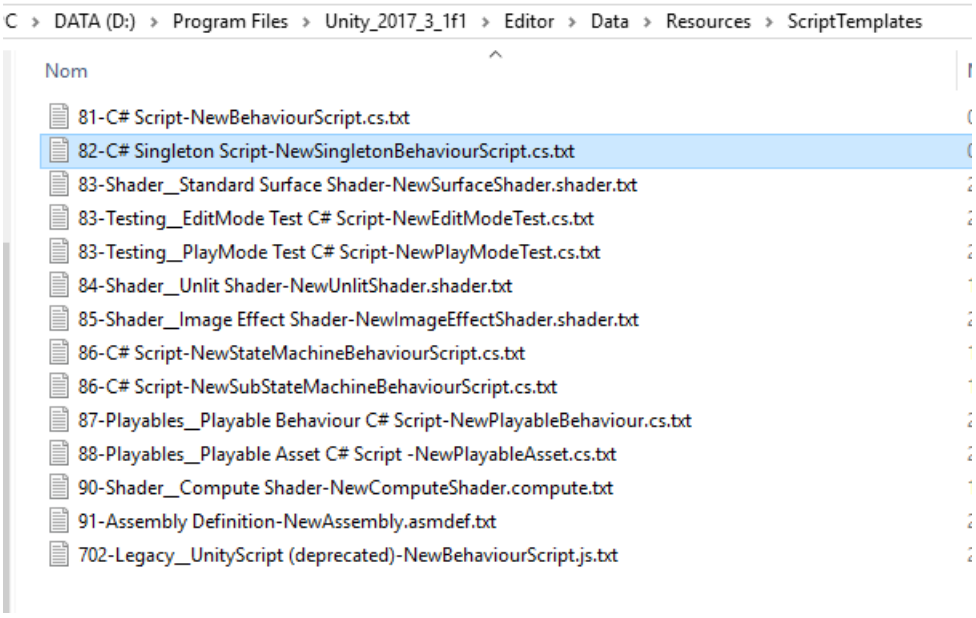
Add your own templates

Example with singleton

Add template to Unity templates folder by selecting ScriptTemplateSettings and clicking on "Edit Script Template" button :



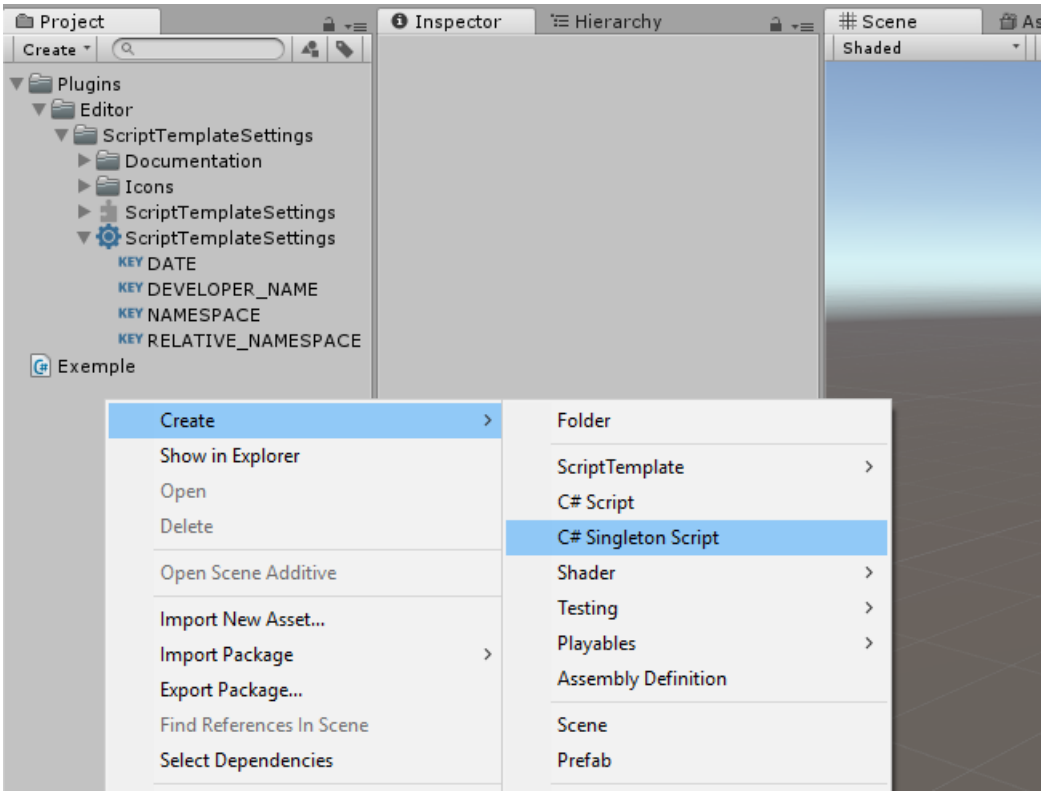
Create a new file and name it `82-C# Singleton Script-NewSingletonBehaviourScript.cs.txt`



Open this file and fill it :

```
///---------------------------------------------------------------
///   Author : #DEVELOPER_NAME#
///   Date   : #DATE#
///---------------------------------------------------------------

using UnityEngine;

namespace #RELATIVE_NAMESPACE# {
    public class #SCRIPTNAME# : MonoBehaviour {
        private static #SCRIPTNAME# _instance;
        public static #SCRIPTNAME# Instance { get { return _instance; } }

        private void Awake(){
            if (_instance){
                Destroy(gameObject);
                return;
            }

            _instance = this;
        }

        private void Start () {
            #NOTRIM#
        }

        private void Update () {
            #NOTRIM#
        }

        private void OnDestroy(){
            if (this == _instance)
                _instance = null;
        }
    }
}
```

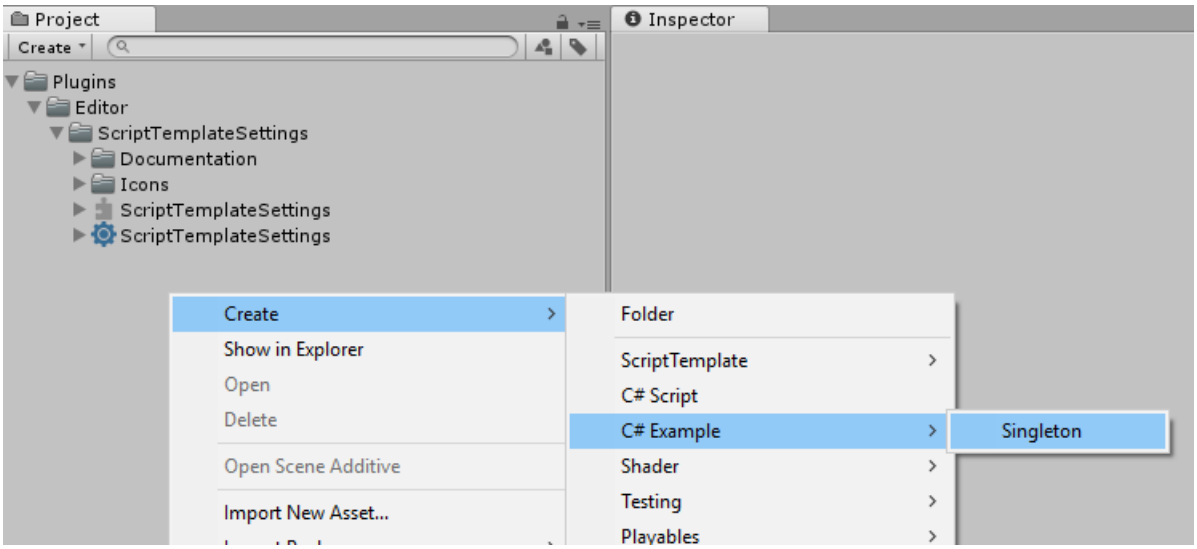
Unity generates automatically new menu when you add your templates to its resources. (Menus appear after restarting Unity)



Composition of the menu from the file name:
`82-C# Singleton Script-NewSingletonBehaviourScript.cs.txt`
`82`: Menu Item position.
`C# Singleton Script`: Menu name
`NewSingletonBehaviourScript.cs`: Default script name

You can also make submenu:
With:
`82-C# Example__Singleton-NewSingletonBehaviourScript.cs.txt`

You will get this result below :

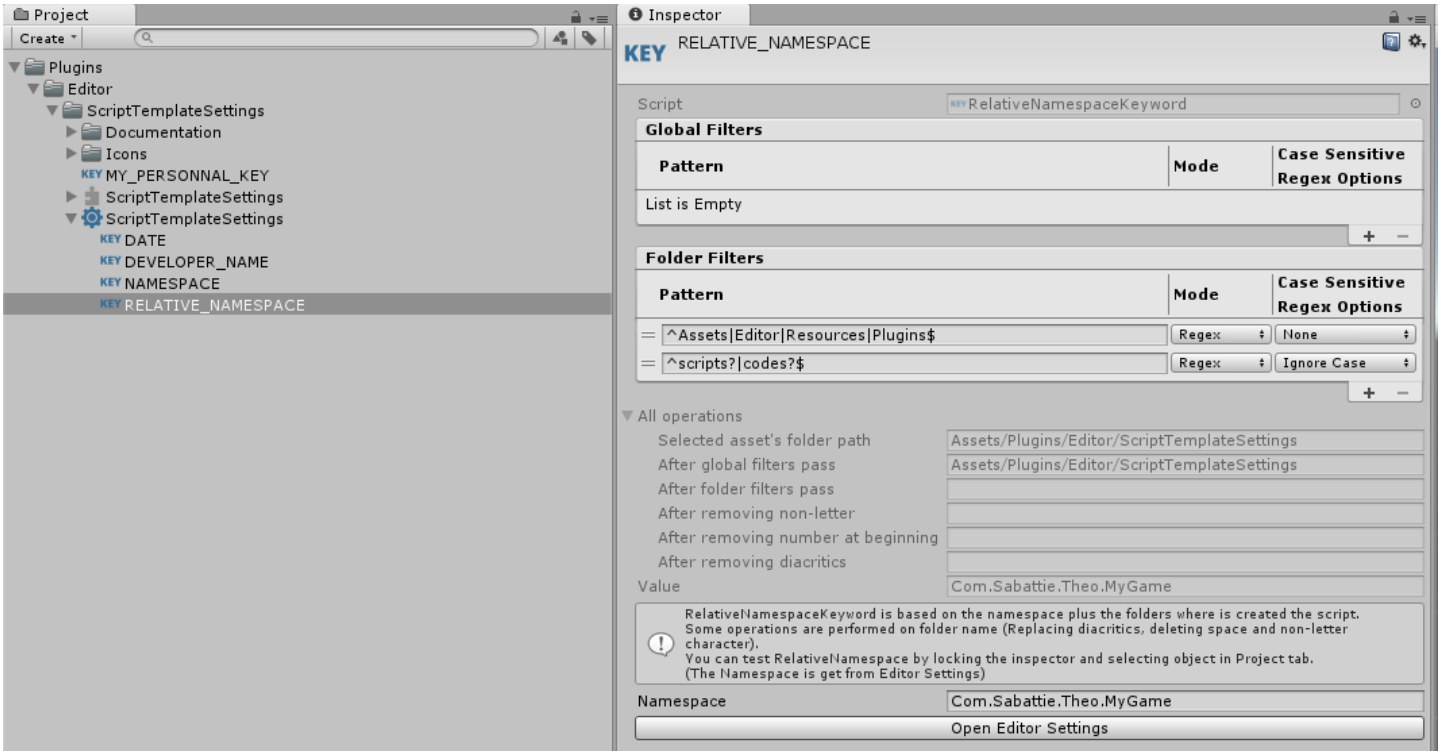


Each `__` represent a submenu. You can cumulate them.

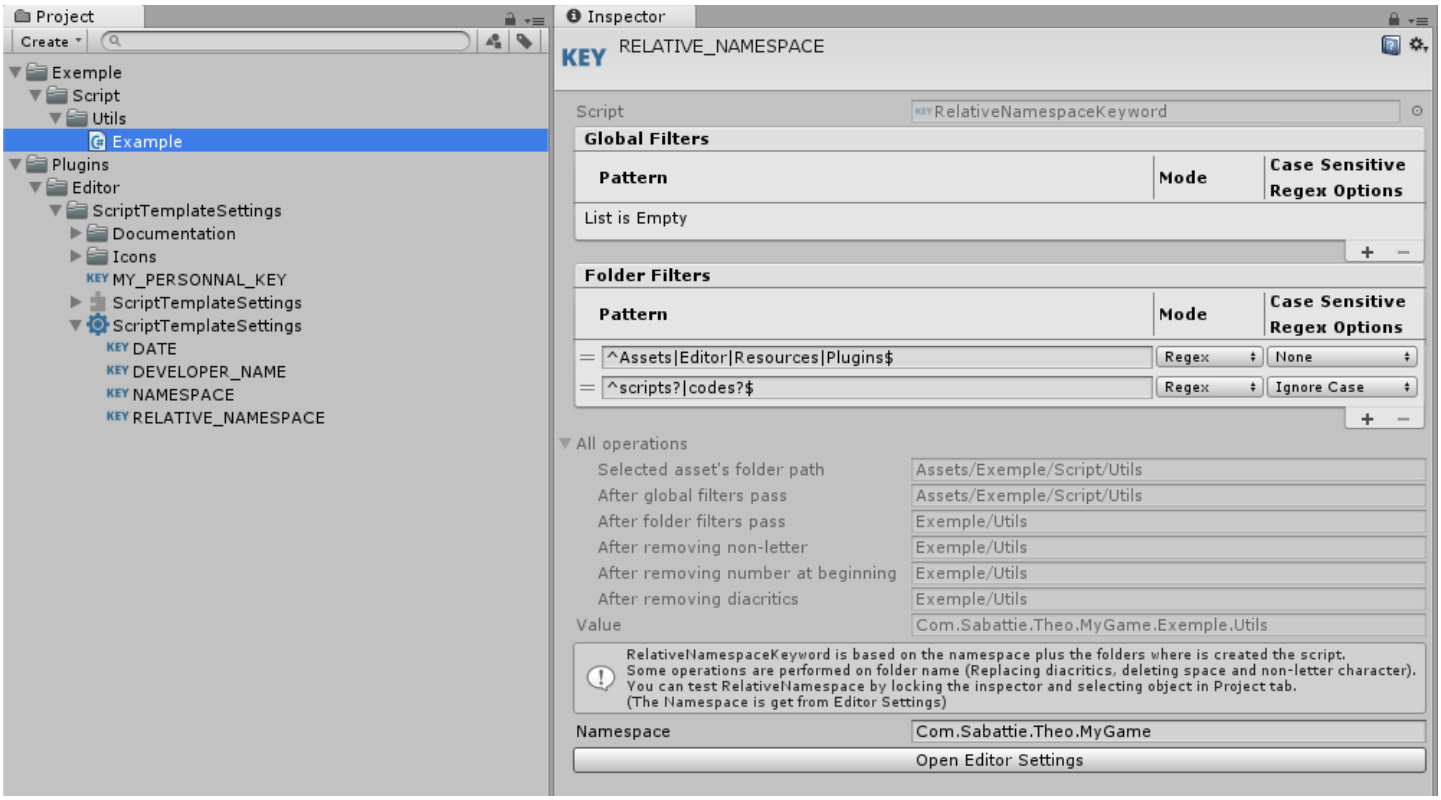
Note: If you want create script from template in another location Unity will not create menu but you can add your own MenuItem and use a method in ScriptTemplateUtils to create script.

RELATIVE_NAMESPACE keyword

RELATIVE_NAMESPACE is a keyword computing the relative namespace using namespace concatened with the folders where the script is placed.
Filter prevent usage of some part of the path for namespace.
There are global filters working directly on the path.
And per folder filters.
Each match remove a part of the path.
Relative namespace removes also diacritics, non-letter char and number at beginning to make a valid namespace.
All operations are visible and can be observed:

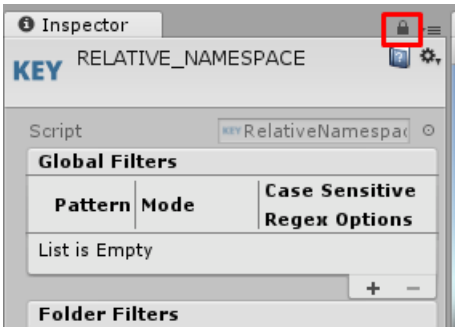


They are relative to the selected file.

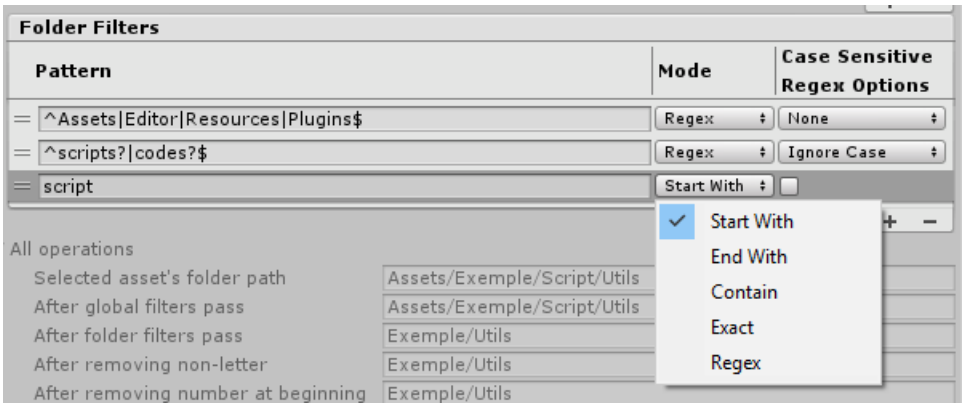


The resulting value can be observed in Value field. (That is not editable because it is a computed value)

Tips:
I conserved the focus on **RELATIVE_NAMESPACE** by locking the inspector on it.

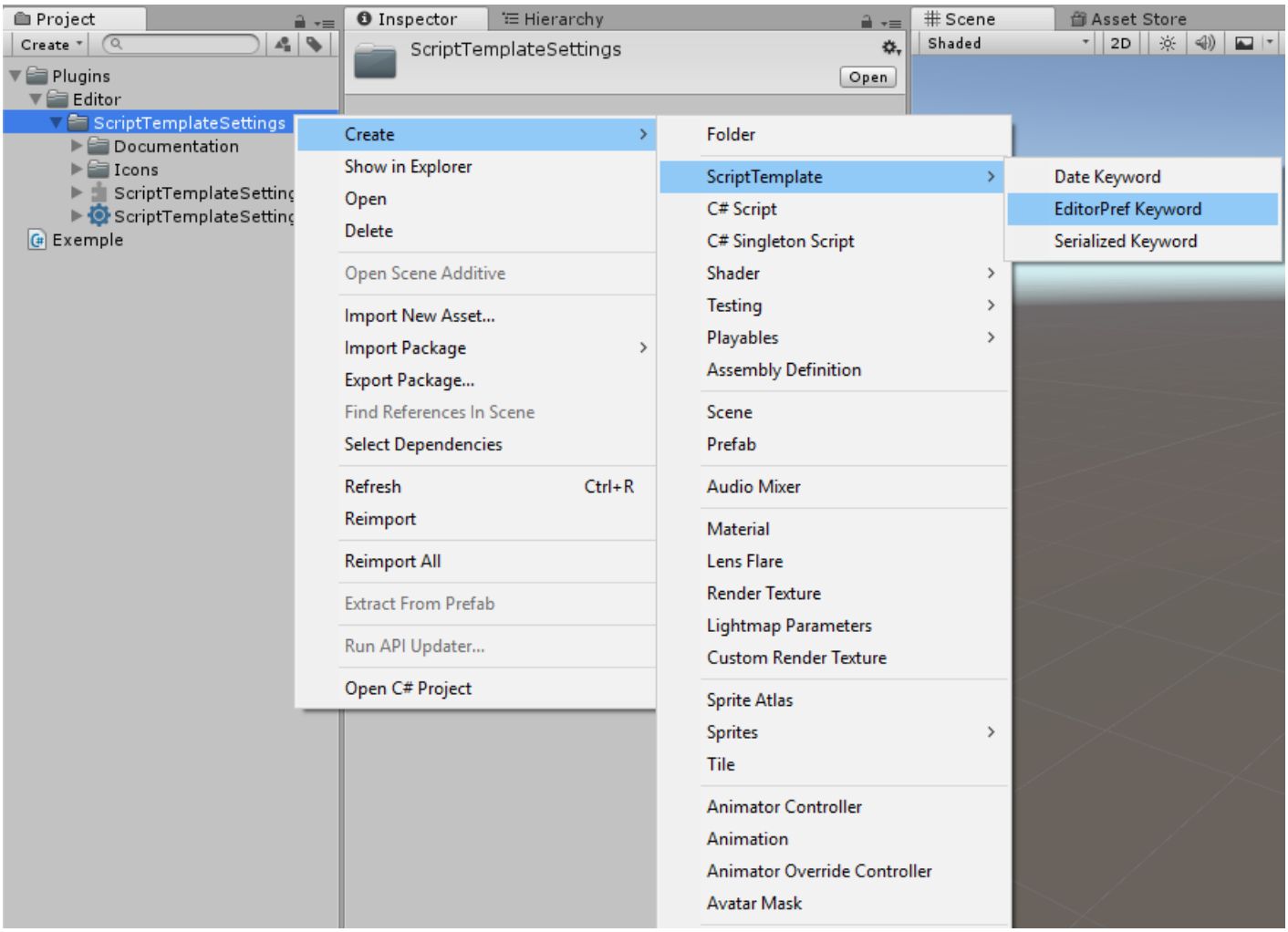


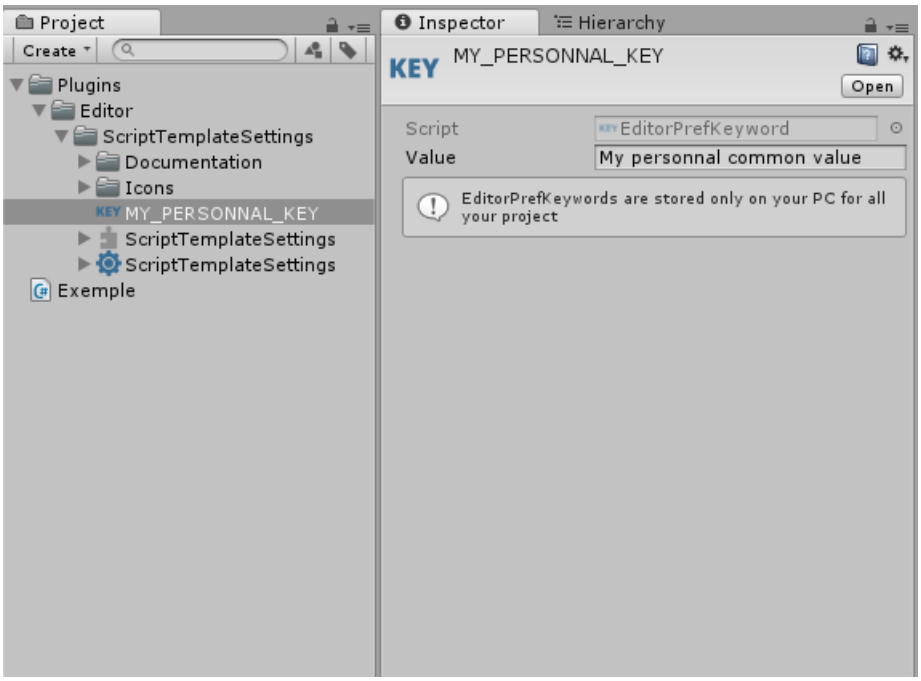
There are some alternatives if you aren't confortable with regex.



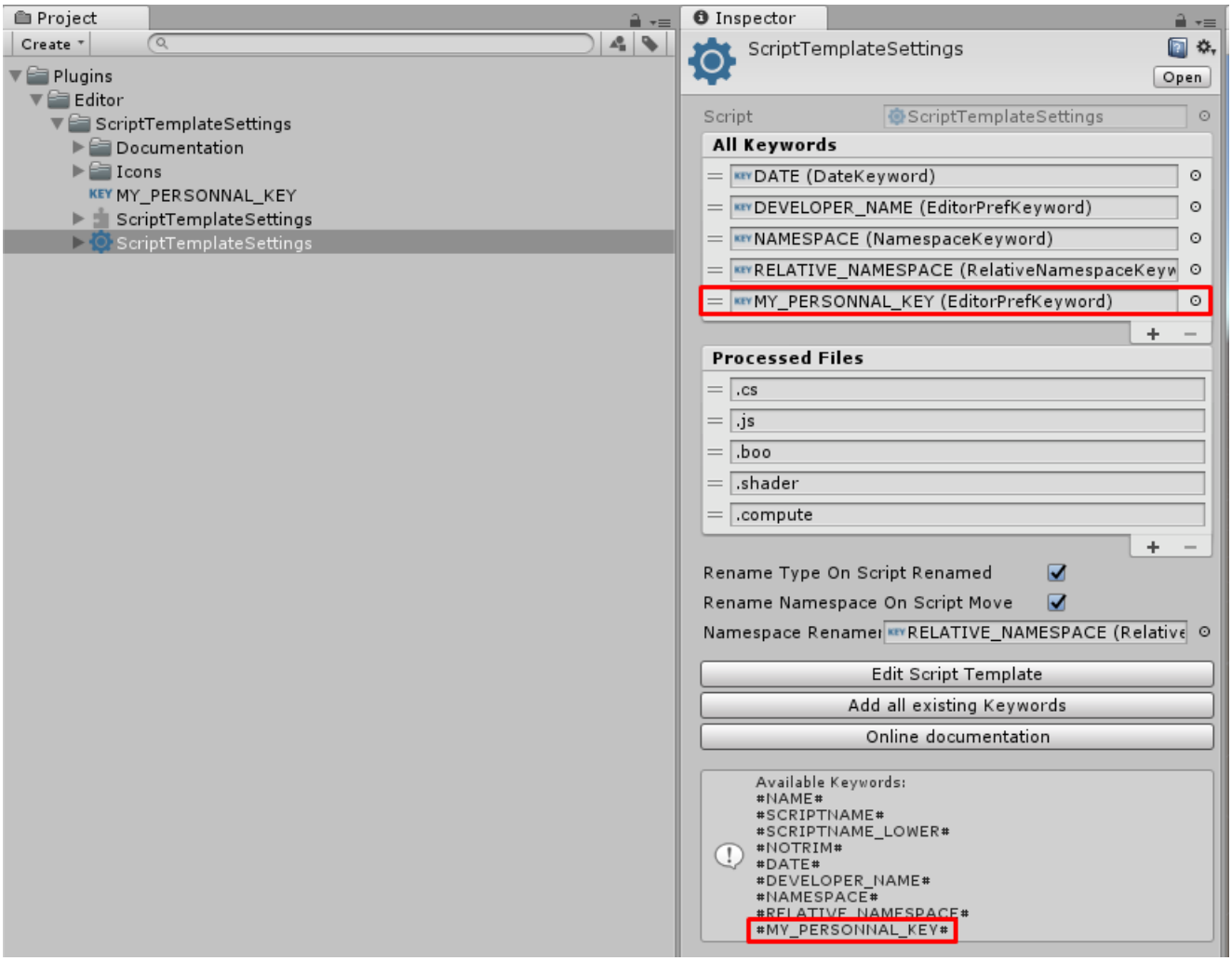
Keyword with personnal value

Those keywords values are not stored per project but globaly per user. They are stored in EditorPrefs. You can add your own personnal keyword by the menu below:





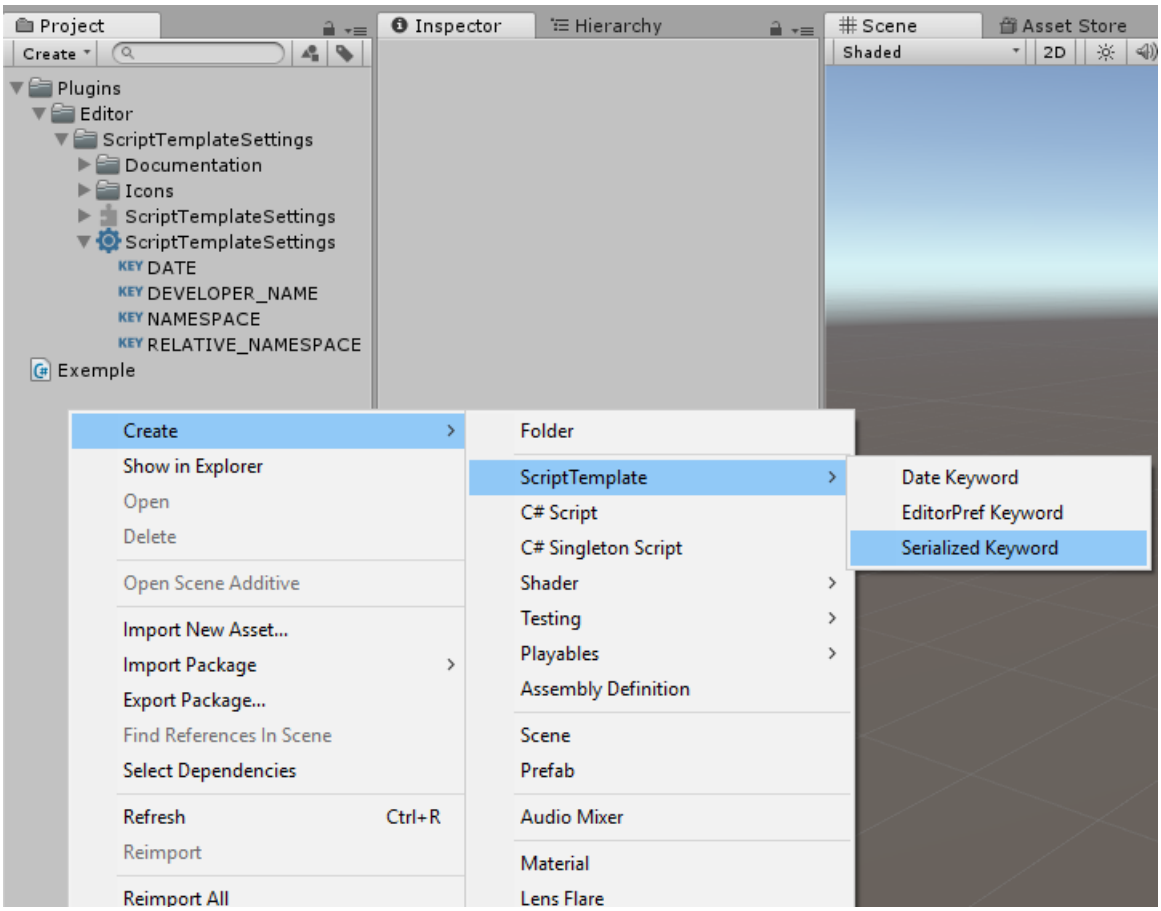
After that, you need to add that keyword to the ScriptTemplateSettings:

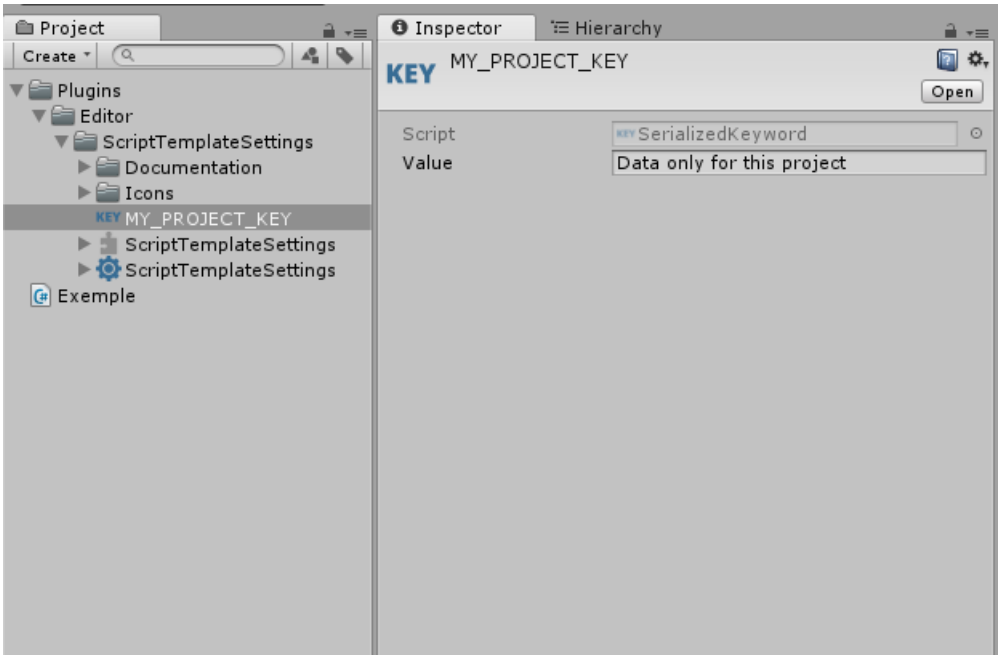


Note: if you create a personnal keyword in another project with the same name, you will get the same value.

Keyword with project value

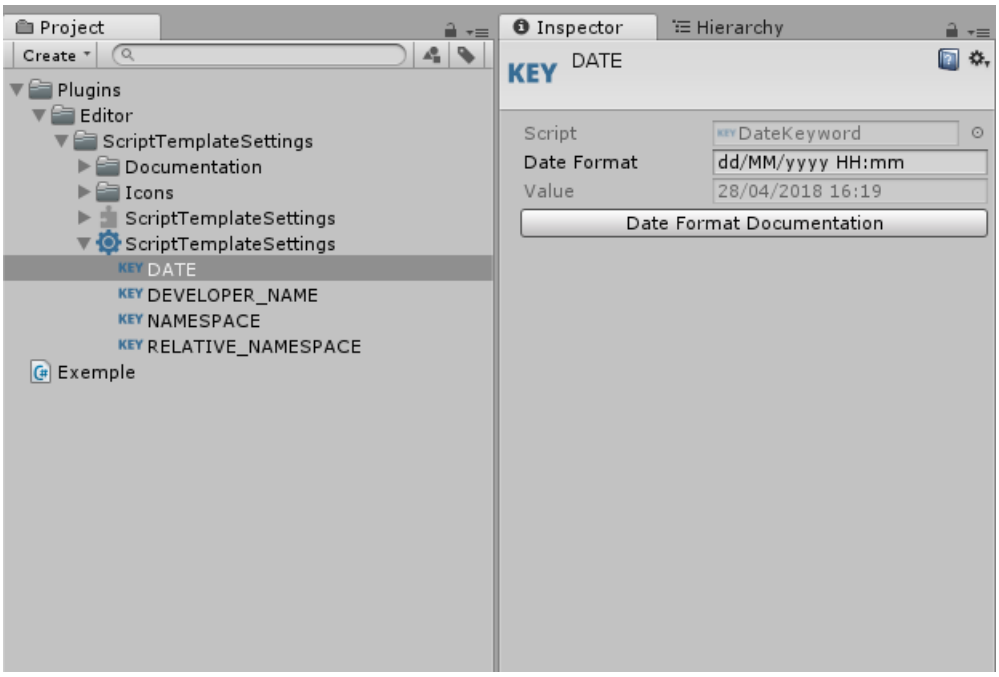
For project keyword, create a Serialized Keyword.

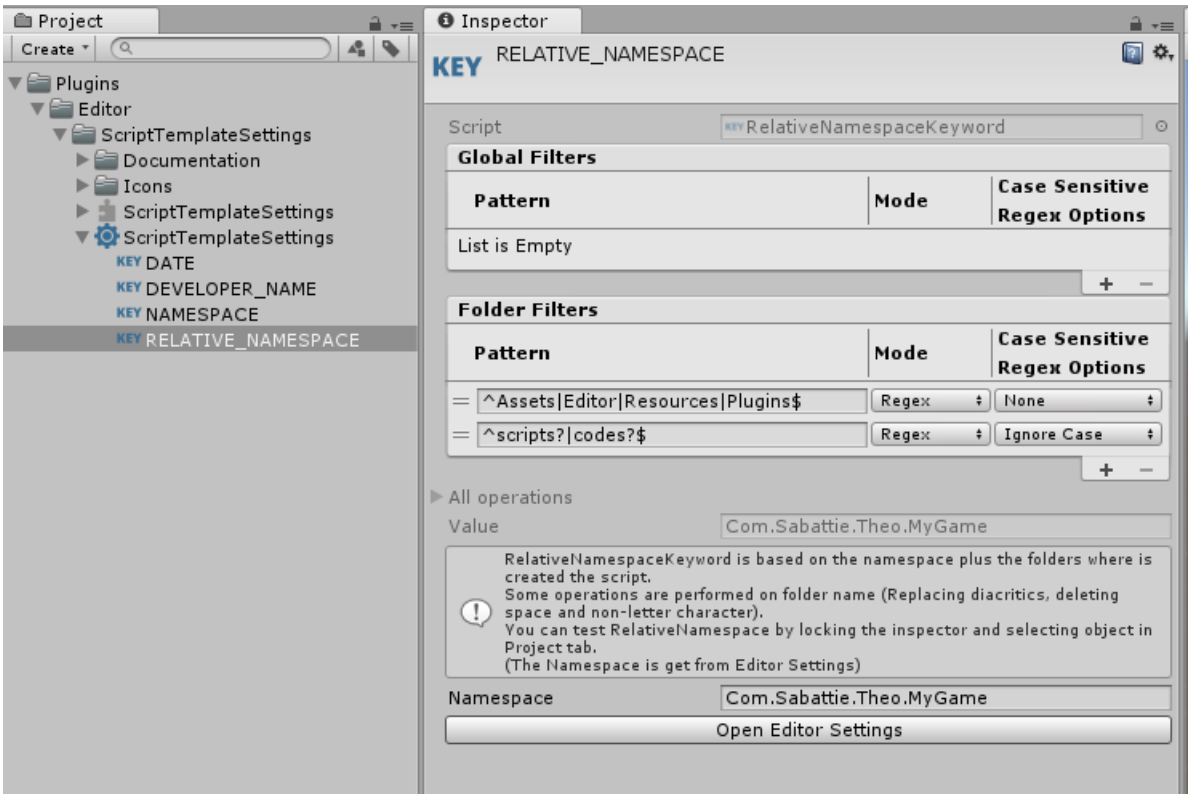




Keyword with computed value

Some keywords already exist : **DATE**, **RELATIVE_NAMESPACE**.





When GetValue is call on thoses Keyword, they compute the resulting value. You can create your own Keyword class to compute value as you want.

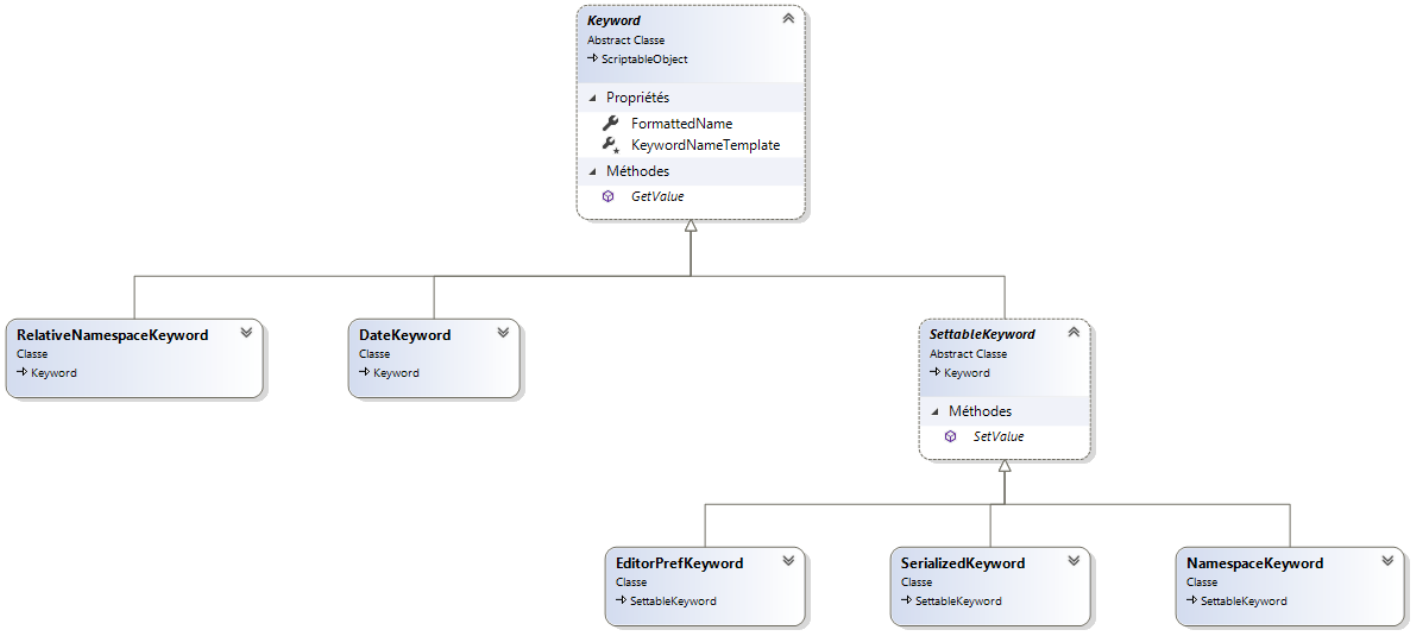
Script Modification Processors

ScriptModificationProcessors are executed when a script is moved or renamed.
By default, 2 Script Modification Processors are created : **Update Namespace On Script Moved Processor**, **Update Type On Script Renamed Processor**.
They processes are executed if there are active.

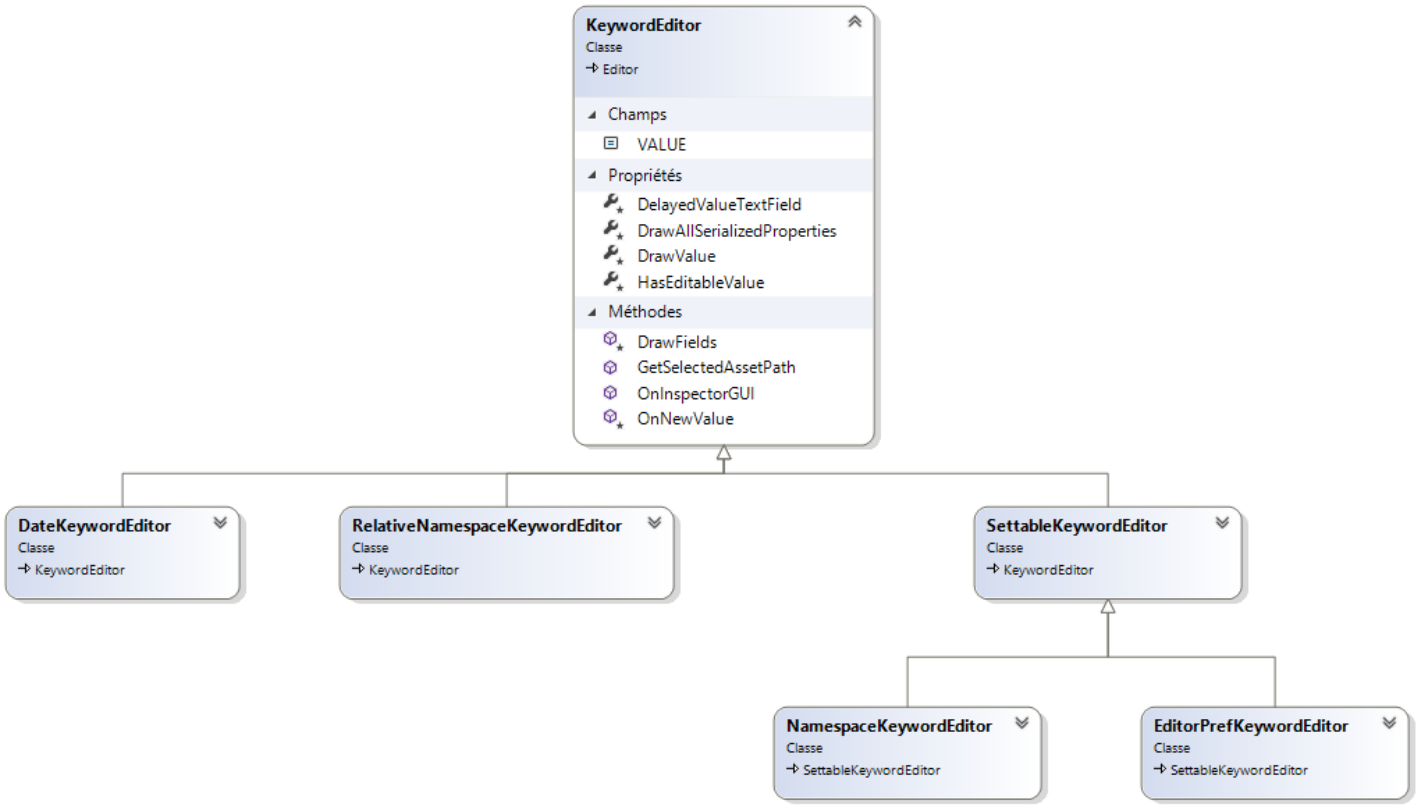
Be careful ! That does not update all the other scripts referencing it.
You can also add your own Script Modification Processor by extending `ScriptModificationProcessor` and implement `OnScriptMoved` method.

Make your own Keyword

If you need to add a keyword with a computed value, extend Keyword class and implement GetValue method.
If you need to add a keyword with a serialized value, extend SettableKeyword class and implement GetValue and SetValue method.

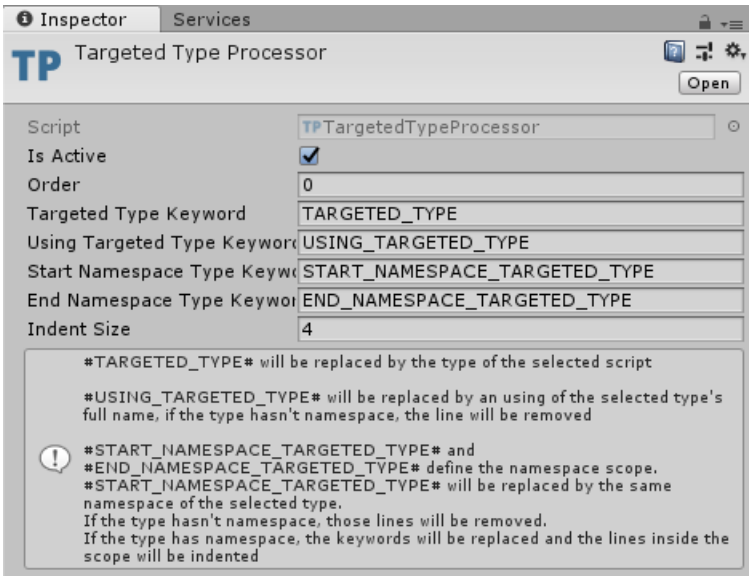


If you want make your own Editor:



Script Template Processors

ScriptTemplateProcessors are executed when a script is created.
Sometime, replace a keyword is not enough to get the result than you want, so, if you need to perform several operations, you can create a class extending `ScriptTemplateProcessor` and implement the `Process` method.
By default, 1 Script Template Processor is created : **Targeted Type Processor**.



That processor replaces keyword of delete them, and also inject some tabulations if necessary.
That is used for creating Editor and Property Drawer. (Select Script then create from menu `C# Editor/Editor`)

Other templates

All basics templates are available in Documention/templates.zip

Scriptable Object (`81-C# Scriptable Object-NewScriptableObject.cs.txt`)

```
///------------------------------------------------------------------
///   Author : #DEVELOPER_NAME#
///   Date   : #DATE#
///------------------------------------------------------------------

using UnityEngine;

namespace #RELATIVE_NAMESPACE# {
    [CreateAssetMenu(menuName = "#PRODUCT#/#SCRIPTNAME#")]
    public class #SCRIPTNAME# : ScriptableObject {

    }
}
```

EditorWindow (`81-Editor__C# EditorWindow-NewEditorWindow.cs.txt`)

```
///------------------------------------------------------------------
///   Author : #DEVELOPER_NAME#
///   Date   : #DATE#
///------------------------------------------------------------------

using UnityEditor;

namespace #RELATIVE_NAMESPACE#
{
    public class #SCRIPTNAME# : EditorWindow
    {
        [MenuItem("Window/#PRODUCT#/#SCRIPTNAME#")]
        public static void Open()
        {
            GetWindow<#SCRIPTNAME#>().Show();
        }

        private void OnEnable()
        {

        }

        private void OnGUI()
        {

        }
    }
}
```

Property Attribute (`81-Editor__C# Property Attribute-NewPropertyAttribute.cs.txt`)

```
///------------------------------------------------------------------
///   Author : #DEVELOPER_NAME#
///   Date   : #DATE#
///------------------------------------------------------------------

namespace #RELATIVE_NAMESPACE#
{
    public class #SCRIPTNAME# : PropertyAttribute
    {
    }
}
```

Editor (`Editor/81-Editor__C# Editor-NewEditor.cs.txt`)

```
///------------------------------------------------------------------
///   Author : #DEVELOPER_NAME#
///   Date   : #DATE#
///------------------------------------------------------------------

using UnityEditor;
using UnityEngine;

#START_NAMESPACE_TARGETED_TYPE#
[CustomEditor(typeof(#TARGETED_TYPE#))]
public class #SCRIPTNAME# : Editor
{
    public sealed override void OnInspectorGUI()
    {
        base.OnInspectorGUI();
        serializedObject.Update();

        // Do what you want

        serializedObject.ApplyModifiedProperties();
    }
}
#END_NAMESPACE_TARGETED_TYPE#
```

Property Drawer (`Editor/81-Editor__C# Property Drawer-NewPropertyDrawer.cs.txt`)

```
///------------------------------------------------------------------
///   Author : #DEVELOPER_NAME#
///   Date   : #DATE#
///------------------------------------------------------------------

using UnityEditor;
using UnityEngine;

#START_NAMESPACE_TARGETED_TYPE#
[CustomPropertyDrawer(typeof(#TARGETED_TYPE#))]
public class #SCRIPTNAME# : PropertyDrawer
{
    public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)
    {
        // Draw what you want with EditorGUI
    }

    public override float GetPropertyHeight(SerializedProperty property, GUIContent label)
    {
        return EditorGUIUtility.singleLineHeight; // To tune in
    }
}
#END_NAMESPACE_TARGETED_TYPE#
```