ADVANCED INSPECTOR

TUTORIALS (v1.5)

Contents

Introduction

The Advanced Inspector is not an easy package to grasp, and we know that. There's many reasons for that; lot of features, drastically different design than Unity, no pandering with *easier, smaller* and more limited features, and tons of details to learn. As described, the goal of the Advanced Inspector is to make it useless for you to write custom editor. It's a tall order and to reach that the number of things to learn may appear overwhelming.

Another issue is that the main documentation is written in technical term. We acknowledge that Unity's users are not always professional - there is a lot of newcomers or indie studios - and this kind of wording may frighten you.

However, we cannot translate our technical documentation into simpler term without doubling the document size, which is already massive with over 40 pages. It means finding information would only become much harder.

This document have been created for attempting to compensate for this lack of easiness. It will simply contains very simple and progressive tutorials about how to achieve specific result with the Advanced Inspector.

Let it be clear, if you cannot find or understand how to do something, please let us know! We will be glad to help you. Contact us at admin@lightstrikersoftware.com and we will reply as soon as possible.

Converting an Existing Class

The most basic task you may have to do is to convert an existing class to take advantage of the Advanced Inspector.

By default, the Advanced Inspector does not change any standard behaviour on how your MonoBehaviour or your ScriptableObject are displayed in Unity's inspector. This behaviour of being "off" is wanted. So you have to explicitly tell the tool to handle the display of a specific class. Let's take the following example;

```
public class MyBehaviour : MonoBehaviour
{
    public float someNumber;
    public string someText;
}
```

The first thing you need to do to convert this class to the Advanced Inspector is to declare using the *AdvancedInspector* namespace like this;

```
using AdvancedInspector;

public class MyBehaviour : MonoBehaviour
{
    public float someNumber;
    public string someText;
}
```

This makes the tool available to access from your file. The second step is to flag the class you want to convert with the AdvancedInspector attribute as follow;

```
using AdvancedInspector;

[AdvancedInspector]
public class MyBehaviour : MonoBehaviour
{
    public float someNumber;
    public string someText;
}
```

If you check in Unity, at this point, you will see nothing in the Inspector. It means the Advanced Inspector is now performing its job and handling the display of your class, instead of Unity's inspector.

The first major difference between Unity and the Advanced Inspector; Unity only displays items it can save on disk. The Advanced Inspector has no such limitation.

However, it also means you have to explicitly tell the tool which item should be displayed, and which should not be.

I repeat again, because an item is displayed on the Advanced Inspector does ***NOT*** means it can or will be saved on disk. Unity's rules on serialization - saving items - are not changed. You should read about them; http://blogs.unity3d.com/2012/10/25/unity-serialization/ or google it, as there is dozen of articles about this topic.

Again, by default the Advanced Inspector display nothing. You have to explicitly tell it what should be displayed. The first and easiest way to convert an existing class is to use the "InspectDefaultItems" property from the AdvancedInspector attribute like this;

```
using AdvancedInspector;

[AdvancedInspector(InspectDefaultItems = true)]
public class MyBehaviour : MonoBehaviour
{
    public float someNumber;
    public string someText;
}
```

This flag tells the tool to display every items Unity would usually show. It follows the normal Unity's rules and display what Unity would save. It acts exactly like Unity would.

There is a second way to display items; with the Inspect attribute;

```
using AdvancedInspector;

[AdvancedInspector]
public class MyBehaviour : MonoBehaviour
{
    [Inspect]
    public float someNumber;
    [Inspect]
    public string someText;
}
```

This way, you can flag individual items to be display. You can then show items that Unity would otherwise not show, such as not-saveable fields, properties (getter/setter), or even parameter-less methods (a function with no parameters).

You can use "InspectorDefaultItems" with the Inspect attribute. This way, you only need to flag items that are not friendly to Unity.

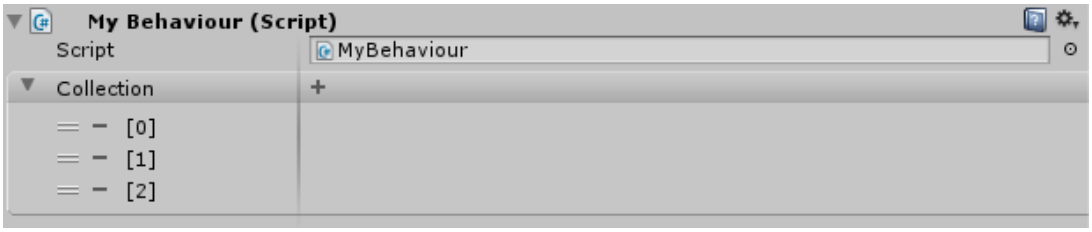
Expanding Object

Once your MonoBehaviour and ScriptableObject are displayed fine, you may need to diplay "sub-item" or object that does not derive from MonoBehaviour. It's take this example;

```
[AdvancedInspector(true, true)]
public class MyBehaviour : MonoBehaviour
{
    public MyObject[] collection;

    [Serializable]
    public class MyObject
    {
        public bool myBool;
        public float myFloat;
    }
}
```

Note that "InspectDefaultItems" can also be flagged by the standard "bool, bool" constructor. The first argument is about displaying the "script field" at the top of the inspector.

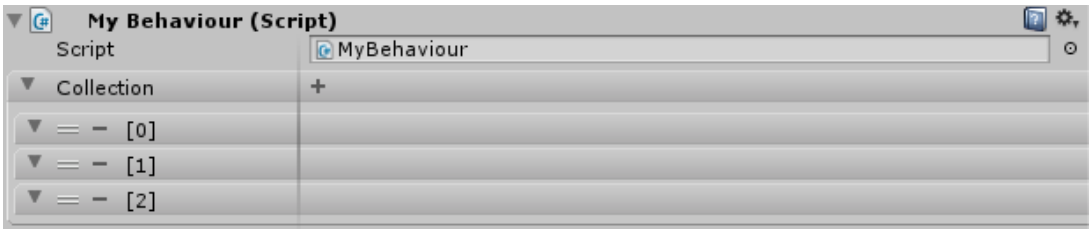
If you run the above, you will see something like this on the Inspector;



This is most likely not what you want to see. Once again, the Advanced Inspector makes no assumption on your intention. You have to explicitly tell it how you want it to handle your classes. In this case, we want to tell the tool that the class "MyObject" can be expanded and it can display items in it. It is done with the Expandable attribute, like so;

```
[AdvancedInspector(true, true)]
public class MyBehaviour : MonoBehaviour
{
    public MyObject[] collection;

    [Serializable, Expandable]
    public class MyObject
    {
        public bool myBool;
        public float myFloat;
    }
}
```

By flagging the class "MyObject" with the Expandable attribute, you claim that his type can be opened up and inspected in a recursive manner, exposing its inner properties. However, if you only do that, you will see this;



Now each items has an expansion arrow besides them, but once again, nothing is displayed. This is the same issue as explained in "Converting an Existing Class", in which the Advanced Inspector does not assume of anything on what should be displayed or not.

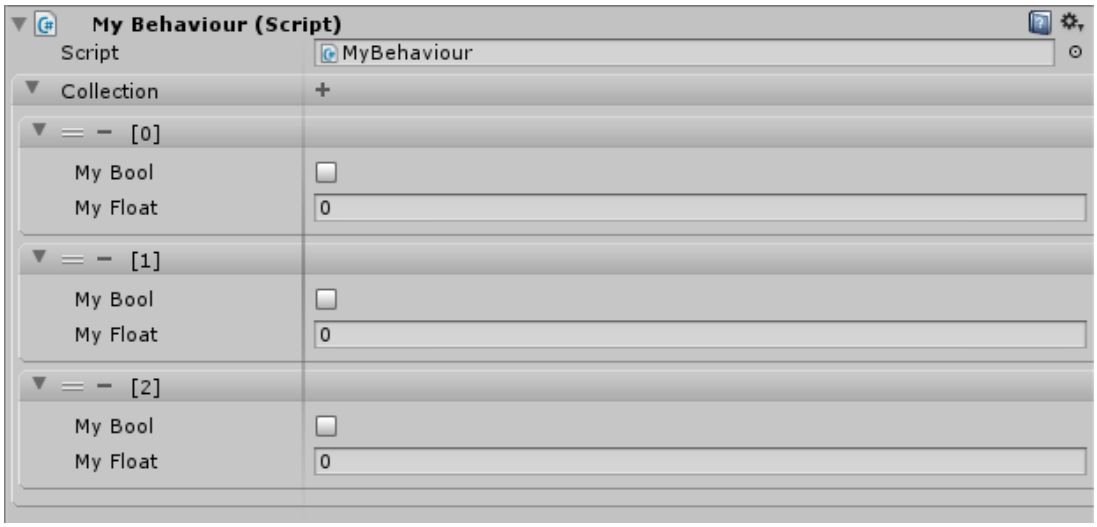
Once again, you can flag your fields with the Inspect attribute, or you can use the "InspectorDefaultItems" property that exist in the Expandable attribute.

```
[AdvancedInspector(true, true)]
public class MyBehaviour : MonoBehaviour
{
    public MyObject[] collection;

    [Serializable, Expandable(InspectDefaultItems = true)]
    public class MyObject
    {
        public bool myBool;
        public float myFloat;
    }
}
```

or;

```
[AdvancedInspector(true, true)]
public class MyBehaviour : MonoBehaviour
{
    public MyObject[] collection;

    [Serializable, Expandable]
    public class MyObject
    {
        [Inspect]
        public bool myBool;
        [Inspect]
        public float myFloat;
    }
}
```

With this, your inspector should not look like this;



You might wonder what is the difference between AdvancedInspector attribute and Expandable attribute. You can see the AdvancedInspector attribute as the ignition key that start or stop the tool. The first class - the entry point - must display this attribute for the tool to do anything, otherwise it reverts to the default Unity behaviour.

As for Expandable attribute, it is used as a flag to know which class can be opened up and which should not be. There is many cases were a class or a struct should not be expanded, such as when there is a specific FieldEditor for it, if it's a value type, or if it's a type that should not be edited "inline".

Note that the Expandable attribute can also be placed on field and property to override the class definition.

Using a Condition to Hide an Item

One of the major advantage of Advanced Inspector is that many of the attributes are dynamic. The behaviour their trigger can change based on condition you provide. If you look at the Inspect attribute, you will see this;

```
public class InspectAttribute : Attribute, IRuntimeAttribute
```

The "IRuntimeAttribute" is an interface that defines an attribute that can take the name of a function as constructor parameter. Knowing this, the tool retrieves the function and invoke it when required.

Often you would wish to hide an item if some condition of another variable is met or not. A good example is how the Light in Unity shows and hides variable depending on the type of the light. For example, if you select a directional light, you won't have the "range" option showing up.

Let's take the following;

```
[AdvancedInspector]
public class MyBehaviour : MonoBehaviour
{
    public GameObject go;

    public bool activateLight = false;
}
```

Let's say you want the "activateLight" variable to show up on the Inspector only if the GameObject in "go" has a Light component.

The Inspect attribute can take the name of a function as constructor. You can go see the signature this function should take directly from the attribute definition;

```
public delegate bool InspectDelegate();
```

This means the function you would write must have no parameter, and must return a boolean. A function that test if a GameObject has a Light will look like this;

```
    private bool HasLight()
    {
        if (go == null)
            return false;

        Light light = go.GetComponent<Light>();
        return light != null;
    }
```

This function returns "true" if the GameObject is not null and if it has a Light component. What is left to do is tell the Inspect attribute which function to use to determine if the item should be displayed or not;

```
[AdvancedInspector]
public class MyBehaviour : MonoBehaviour
{
    [Inspect]
    public GameObject go;

    [Inspect("HasLight")]
    public bool activateLight = false;

    private bool HasLight()
    {
        return go != null && go.GetComponent<Light>() != null;
    }
}
```

Runtime attribute are extremely powerful and give great dynamism to how your class are inspected. They can hide, change name, change color, limit choices and so on. You can even write your own attribute that implement IRuntimeAttribute.

Forcing a List of Choices

You often ends up having an item that display too many choices, or even the wrong choices. With Unity, you would be forced to write a Custom Editor just for the sake of controlling what data is displayed. (See "Using a Condition to Hide an Item")

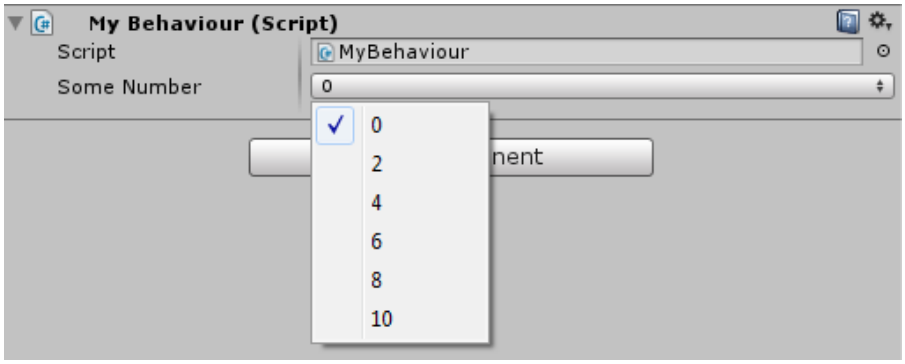
With the Advanced Inspector, you can do that by writing a single function. First, add the "Restrict" attribute to the item you want limited;

```
[AdvancedInspector]
public class MyBehaviour : MonoBehaviour
{
    [Inspect, Restrict("ValidValues")]
    public float someNumber;
}
```

This attribute takes a string as paramater, which should be the name of one function in your class. The function can be private or public, it doesn't matter. The name is case sensitive! So let's write a function that return a list of number I want this field to be limited to;

```
[AdvancedInspector]
public class MyBehaviour : MonoBehaviour
{
    [Inspect, Restrict("ValidValues")]
    public float someNumber;

    private IList ValidValues()
    {
        return new float[] { 0, 2, 4, 6, 8, 10 };
    }
}
```

This function should return a collection that can be cast to a IList. For example, you can use an array, ArrayList, List<> or any collection that implement the IList interface. In the inspector, your class will now look like this;



Restricting displayed values is a very powerful feature. It allows you to guaranty that some data is never wrong or that someone would be unable to input the wrong one. It also makes it much easier to change complex data.

You can list values, types, files, names or anything that you want limited in choices.

The Restrict attribute is a tool that will be used many times in the course of this document.

Conversion between Display and Serialization

One thing that is asked often on Unity's forum or ask website is "How do I save a Type?". Usually the answer goes along the lines of "You cannot" or "You need to write some complex Custom Editor".

With the Advanced Inspector, it's easy as pie.

The first thing to understand about "Type" is that you cannot directly save it. This object is only created at runtime and simply cannot be saved. However, you can save something that is a indirect reference to it; the Assembly Qualified Name, which is a string.

Since Unity has no issue saving strings, let's start by making a field to save it;

```
[AdvancedInspector]
public class MyBehaviour : MonoBehaviour
{
    [SerializeField]
    private string assemblyQualifiedName;
}
```

The field is private because we don't want anybody to directly access this data. While the Assembly Qualified Name makes sense to the .NET Framework, it is just gibberish for us. To convert this string back and from the proper Type, we will use a property (getter/setter);

```
[AdvancedInspector]
public class MyBehaviour : MonoBehaviour
{
    [SerializeField]
    private string assemblyQualifiedName;

    public Type myType
    {
        get { return Type.GetType(assemblyQualifiedName); }
        set { assemblyQualifiedName = value.AssemblyQualifiedName; }
    }
}
```

As you can see here, the getter/setter is serving as a translation layer between the saved data, and the data we want to use or see.

The second issue here is that Unity or the Advanced Inspector has no idea how to display a "Type", and it's normal as there is not much useful in it to display. However, the Restrict attribute - as seen in the chapter "Forcing a List of Choices" - allows us to build a list of items we wish the user to choose from, even types.

Let's say we want to display a list of all the class that derive from "MonoBehaviour", which in most case would be a list of the class we wrote. For this, we will use the System.Reflection namespace. It's an incredibly powerful set of tool, and if you don't know about it, you should definitely give it a look. For now, let's just say that the way to get the list of classes is like this;

```
private IList GetTypes()
{
    List<Type> types = new List<Type>();
    foreach (Assembly assembly in AppDomain.CurrentDomain.GetAssemblies())
    {
        foreach (Type type in assembly.GetTypes())
        {
            if (type != typeof(MonoBehaviour) && !type.IsAbstract &&
                typeof(MonoBehaviour).IsAssignableFrom(type))
            {
                types.Add(type);
            }
        }
    }

    return types;
}
```

This function loops in all the loaded assemblies and get all the types. It checks that the type is not directly "MonoBehaviour", that is not abstract and that it can be assigned as "MonoBehaviour" - in essence, that it is deriving from MonoBehaviour.

What is left to do is tell the Restrict attribute to use this method;

```
[Inspect, Restrict("GetTypes")]
public Type myType
{
    get { return Type.GetType(assemblyQualifiedName); }
    set { assemblyQualifiedName = value.AssemblyQualifiedName; }
}
```

If you have many script, you may end up with a very long list and it would be very hard to navigate in the huge drop down list. Luckily, the Restrict attribute also have another parameter;

```
[Inspect, Restrict("GetTypes", InspectorDisplay.Toolbox)]
```

This way, instead of a drop down list, a "+" icon will show up. When clicked on it, a dialog box with a search field will show up listing all the values we returned in "GetTypes". This method of display works best with large collection of choice.

Contact

The Advanced Inspector was a huge undertaking, and for that reason the first few versions are expected to have hidden issues that we are not aware of. We will fix those bugs as quickly as we can, however we need to learn about them. To send us any bug report or feature request, contact us at;

Email : admin@lightstrikersoftware.com

Website : www.lightstrikersoftware.com