

MolecularLib Docs

The documentation for the MolecularLib

USER GUIDE

Release 1.1.3

June 2023

Copyright (C) 2023 Gabriel Pasquale - All Rights Reserved

This manual, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. The content of this manual is furnished for informational use only, is subject to change without notice and should not be construed as a commitment by its authors. The author assumes no responsibility or liability for any errors or inaccuracies that may appear in this manual.

Table of Contents

Getting Started

Overview.....	6
Installation.....	6
The Demos.....	6

Polymorphic Variable

Overview.....	7
How to use.....	8
Limitations.....	9
How it works (In a nutshell)	9
Custom Serialization Support	9

Serializable Dictionary

Overview.....	10
How to use.....	10

Volatile Scriptable Object

Overview.....	11-12
How to use.....	13

Instantiate with arguments

Overview.....	14
Additional Notes.....	14

Auto Singleton

Overview.....	15
---------------	----

Timers

Overview.....	16
Timer Async.....	17
Timer Reference.....	17
Timer Coroutine.....	18

Table of Contents

Auto Assign

Overview.....	19
---------------	----

Color Helper

Overview.....	20
Methods.....	20-21

Vector Helper

Overview.....	22
Methods.....	22-23

String Helper

Overview.....	24
Methods.....	24-25

Ranges

Overview.....	26
The Different Range Types.....	26-28
MinMaxAttribute.....	28
Limitations of the generic Range class.....	28

Maths

Overview.....	29
Methods.....	29

Other Helpers

Play Status.....	30
Tag.....	30
Optional.....	31
Type Library.....	32
Type Variable.....	32

Table of Contents

Editor Helper

Overview.....	33
Methods.....	33-34

Enhanced Sprite Editor Drawer

Overview.....	35
How to disable it.....	35

Getting Started

Overview

This document provides a concise and not-too-long documentation for all the features included in the MolecularLib package. As a reminder, this package is open-source, with its source code hosted in [this](#) GitHub repository.

In this manual, for each topic there will be, in the start of the overview section, the following

Usage:

Namespace:

Where Namespace says which namespace you must have included in the using directives at the top of the file to use the feature. The Usage shows a quick example of how it's used.

In the case that you find any bugs while using the MolecularLib, and you are sure the bug is in the package code, please [send a bug report here](#) following the template that will be prompted to you.

In the case you would like to see a new feature being implemented, please [send a feature request here](#)

This documentation contains some gifs, which are only visible on the word document.

Installation

If you installed the package via the Asset Store or the package manager, you can just skip this. If you are installing it via the .unitypackage file, here is how to do it, go to: **"Assets->Import Package->Custom Package..."**. In the Import Asset window, find and select the MolecularLib.unitypackage file. When the "Importing package" window shows up, check all items to be imported and then click the Import button in the bottom right of the window.

The Demos

The project contains a Demo folder with a Demo scene, a demo Volatile Scriptable Object named Volatile SO, and some scripts that implement the functionality of the Demo scene, feel free to navigate this codes, they are written in a easy to understand way to serve as documentation as well. Two scripts worthy while looking are DemoAutoAssign and DemoDrawersScript.

The Demo folder is located at **"HandsomeDinosaur/MolecularLib/Core/Demo"**.

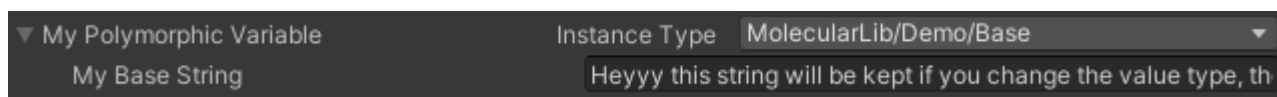
Polymorphic Variable

Overview

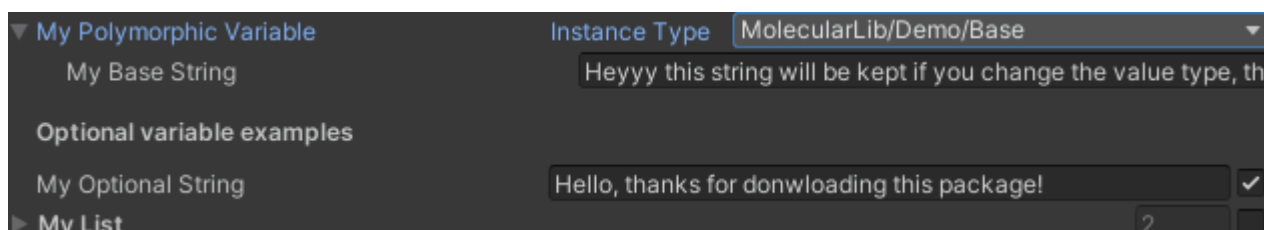
Declaration: PolymorphicVariable<TBase> (e.g. PolymorphicVariable<Weapon>)
Namespace: MolecularLib.PolymorphismSupport

The Polymorphic Variable adds support for polymorphism in unity serializable fields. If you don't know what polymorphism is, I strongly suggest [reading this page](#) (It is very short, mostly examples).

Basically, when you declare a Polymorphic Variable (like public PolymorphicVariable<MyBaseType> myPolymorphicVar or [SerializeField] private PolymorphicVariable<MyBaseType> myPolymorphicVar) in the editor it will appear like this:



Where the instance type defines what type the variable should be. As you change the Instance Type, the field prompted in the inspector will change.



(gif – viewable only in the word document)

You can access this value as TBase using myPolymorphicVar.Value or you can get it already converted using myPolymorphicVar.As<TDerived>(out TDerived, bool onlyPerfectTypeMatch) like this

```
if (myPolymorphicVariable.As<A>(out var asA))
    Debug.Log($"As A | aClassInt: {asA.aClassInt}");
else if (myPolymorphicVariable.As<B>(out var asB))
    Debug.Log($"As B | bClassInt: {asB.bClassInt}");
else if (myPolymorphicVariable.As<C>(out var asC))
    Debug.Log($"As C | cClassFloat: {asC.cClassFloat}");
else
    Debug.Log($"As Base | myBaseString: {myPolymorphicVariable.Value.myBaseString}");
```

Polymorphic Variable

How to use

First, define a `PolymorphicVariable<Your Base Type Here>` either as a public field or as a `[SerializeField]` private field. After that open the inspector for the object containing the script with this field and there you go, you have successfully declared the field. Keep in mind that the `PolymorphicVariable` will accept ANY type that derives from the provided base type, so if you put `System.Object` as the base type unity will get very lag, as the editor drawer will need to fetch all the types to be displayed in the popup (in this case around 70000).

In your code, you can access the value as the base provided type using `myPolymorphicVar.Value`. To check if the value is of a type and get the value as said type, use `myPolymorphicVar.As<ExpectedType>(out var value)` (this returns a bool, so you can put it inside a "if" statement). Using it like this, any type that derives from `ExpectedType` will NOT return true to this function. If you want those types to also return true, call the function like this: `myPolymorphicVar.As<ExpectedType>(out var value, false)`.

```
if (myPolymorphicVariable.As<B>(out var asB)) // This will match only the B type
    Debug.Log($"As B | bClassInt: {asB.bClassInt}");

// This will match both B classes and any derived from it, in the case of the
// Demo, B and C.
if (myPolymorphicVariable.As<B>(out var asB, false))
    Debug.Log($"As B | bClassInt: {asB.bClassInt}");
```

You can also access the selected type in the inspector using `myPolymorphicVar.SelectedPolymorphicType` and the actual value type with `myPolymorphicVar.ValueType`

There is also an implicit conversion operator defined to get the value as `TBase`, so you can use `(TBase)myPolymorphicVar`, or, depending of the context, `myPolymorphicVar` to get the `Value` property.

Note: IF SOME PRIVATE fields from your selected type aren't being saved, try setting them to public or have the `[DataMember]` attribute added. In the latter option you need to be 100% SURE that the class has both `[Serializable, DataContract]` attributes, like this:

```
[Serializable, DataContract]
public class MyClass
{
    public int myPublicField;
    [SerializeField, DataMember] private int myPrivateField;
}
```

Theoretically it shouldn't be needed though. Now **if you want to ignore a field and don't serialize it use the `[XmlIgnore]` attribute**

Polymorphic Variable

Limitations

- Can't do `PolymorphicVariable<UnityEngine.Object>` (not that that would make much sense for most cases anyway)
- All the field types (that do not derive from `UnityEngine.Object`) need to have a parameterless constructor (or no constructor) and be `Serializable`

First off, you cannot make a `PolymorphicVariable<UnityEngine.Object>` as it does not make sense. See, types deriving from the `UnityEngine.Object`, like `ScriptableObject`, or `MonoBehaviour` kind of already have polymorphism. You only assign references to those types.

Because of how it works, every type (that do not derive from `UnityEngine.Object` that will be saved by the `PolymorphicVariable` NEEDS to have a constructor with no parameters or no constructors at all and must be `Serializable`. If you have control over the class all you need to do is add `[System.Serializable]` to the class, additionally you might need to implement custom serialization for some user-defined classes, in that case implement the interface `IPolymorphicSerializationOverride` (See Custom Serialization Support below for more info)

How it works (In a nutshell)

Basically it uses reflection to fetch all the fields of the selected desired type and its values. Then it serializes all of that data using the [XmlSerializer class](#) (Except for fields that derive from `UnityEngine.Object`, those are saved normally in the same class of the `XML_serialized` field). This serialized data is then stored in a string field in the `SerializedPolymorphicField` class. What this means, and this is the whole purpose of this topic, is that if you will mess with anything relating `SerializedProperty` and `SerializedObject`, beware that the fields shown in the editor don't exist, they are all strings containing XML data.

Custom Serialization Support

If, for any reason, you need, or want, to implement your own serialization without using the `XmlSerializer` (Since it doesn't support all types), you can implement this interface `IPolymorphicSerializationOverride` in the type desired to have its own serialization. In case you want an example of that, check the code of the `SerializableDictionary`, it is used there.

Serializable Dictionary

Overview

Declaration: `SerializableDictionary<TKey, TValue>` (e.g. `SerializableDictionary<string, int>`)
Namespace: `MolecularLib`

As it is commonly known, unity serializes Lists, but not Dictionaries. But what is a serialized dictionary if not 2 lists, one for the keys, and one for the values? That's exactly what the `SerializableDictionary<TKey, TValue>` does! Right before unity serializes the class, it converts its dictionary to two list, keys and values, which are then serialized by unity. Then just after deserialization those lists are converted back to a dictionary. It also comes with a neat editor drawer! It will look like this in the inspector (if the variable is public or has `[SerializeField]`):



The `SerializableDictionary` class derives from `Dictionary`, so you can use it in runtime just like you would use a normal `Dictionary` class.

How to use

Well, as previously said, the `SerializableDictionary<TKey, TValue>` derives from `Dictionary<TKey, TValue>`, which makes its use exactly equal to the one you would have with a normal dictionary. If you want to learn more about dictionaries, go [here](#).

Volatile Scriptable Object

Overview

Declaration: `VolatileScriptableObject<TData>`

(e.g. `public class MyScriptableObject : VolatileScriptableObject<MyScriptableObjectData>`)

Namespace: `MolecularLib`

As you probably know, scriptable objects in unity have persistent data for entering and leaving play mode, so if you have an int in a scriptable object set to 100, and then, while in play mode, you change it to 50, when you exit play mode, it will still be 50, not 100. This is sometimes good and sometimes bad. One of these bad times is when you're using scriptable objects to hold changeable data, say player health. You would want for the player health to be reset when you leave play mode, but you also want to change it during play mode. If you are thinking this is a very strange way to code a game by the way, check [this](#), it actually has some very good benefits. So, what the `VolatileScriptableObject` class does is that it solves this problem! You can change the data in it, and when you leave play mode it won't be changed.

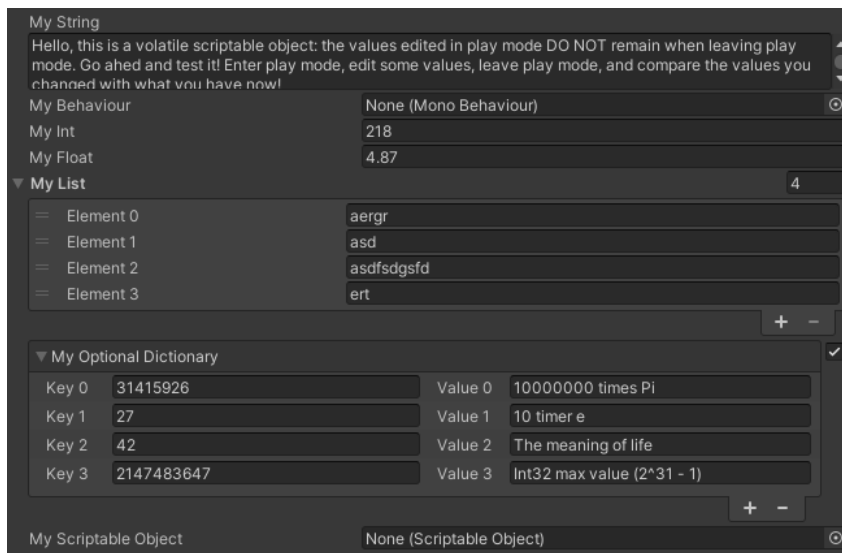
There is a very good demo and implementation of this in here "**Demo >**

TestVolatileScriptableObject.cs". But if you want a quick sneak-pic of it, here it is:

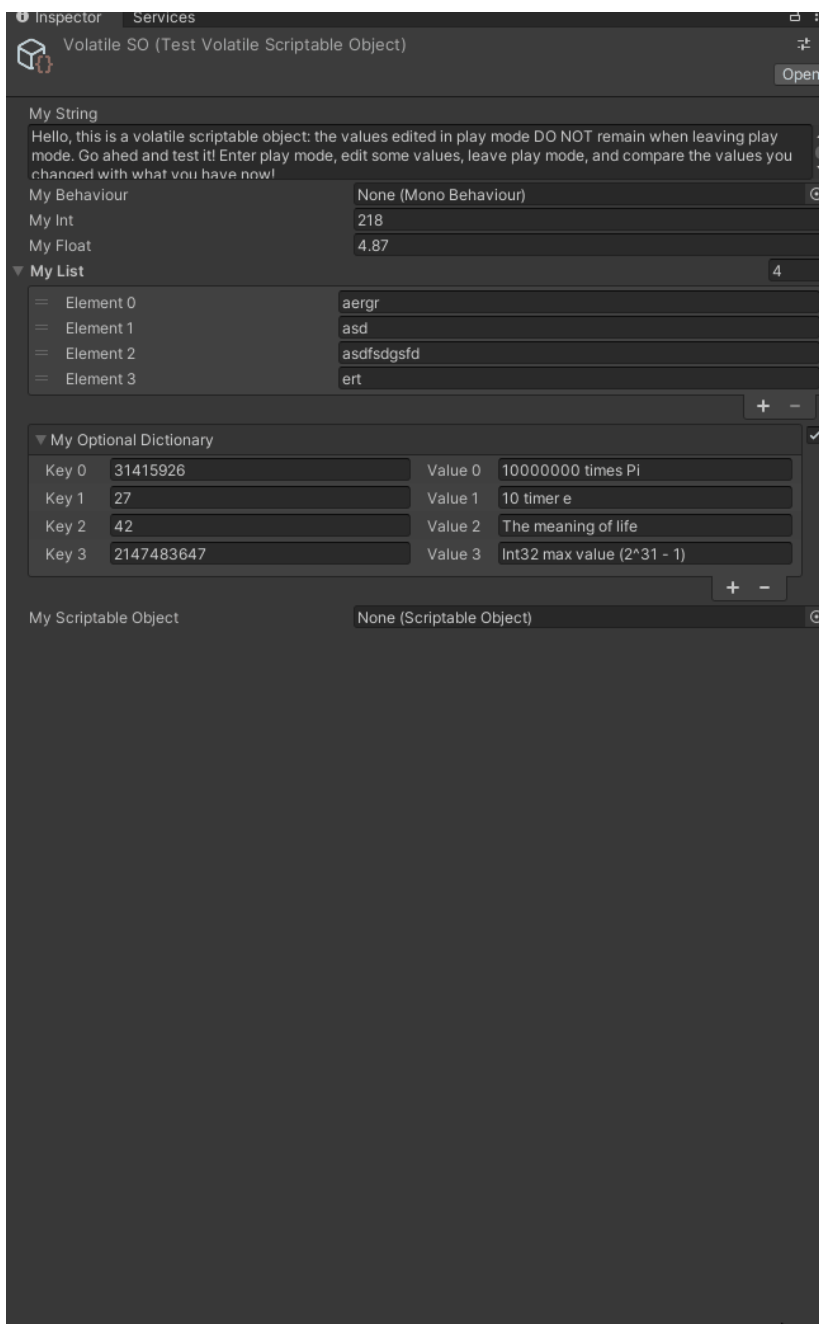
```
[CreateAssetMenu(fileName = "Volatile SO", menuName = "New Volatile SO", order = 0)]
public class TestVolatileScriptableObject :
    VolatileScriptableObject<TestVolatileScriptableObject.Data>
{
    // Here is a quick way of accessing the data. Can be done in other ways too.
    public Data VolatileData
    {
        get => Value;
        set => Value = value;
    }

    // Here you will put all the data you want to be volatile
    [Serializable]
    public class Data
    {
        [TextArea] public string myString;
        public MonoBehaviour myBehaviour;
        public int myInt;
        public float myFloat;
        public List<string> myList;
        public Optional<SerializableDictionary<int, string>> myOptionalDictionary;
        public ScriptableObject myScriptableObject;
    }
}
```

In the editor (not in play mode), this will look like this:



During play mode it will be like this:



(gif – viewable only in the word document)

Volatile Scriptable Object

How to use

Well, as previously shown, you will make your own ScriptableObject derive from VolatileScriptableObject<T> where T will be a class or struct that will contain all the data to be volatile.

As protected values (which you can access from your derived class) you will have

- Value
- runtimeValue
- editorSavedValue

And as a protected method you will have:

- CopyRuntimeValuesToEditorSaved() :void

Where runtimeValue and editorSavedValue are very self-explanatory, one is the value during runtime, the other, not. But you will most probably only use the Value property, and here's why: during runtime (or in builds) it will return always the runtimeValue, so it can be edited without changing the editorSavedValue. When not on play mode, it will always return the editorSavedValue, so it can be edited.

Technical Details

If you are wondering how this works, it basically stores a copy of all the values on the scriptable object. If now you are wondering how this works with reference types (e.g. List<some type>) it uses a DeepCopy method present on the VolatileScriptableObject class that serialized the data to JSON and than deserializes it using the JsonUtility class. This method is only called while entering play mode or during the editor when values are changed.

Instantiate with arguments

Overview

Usage: `Molecular.Instantiate(original :GameObject, [parente :Transform], [pos :Vector3], [rot :Quaternion], arg1 :T1, arg2 :T2 ...)`
Namespace: `MolecularLib`

First off to use this, the object that you want to instantiate needs have this interface: `IArgsInstantiable<>`. There are a total of 10 overrides of this interface, one with no parameters, one with 1 parameter, one with 2 parameters ... and one with 10 parameters. The function that this interface implements will as parameters the types passed into the interface, like so:

```
public class TestArgInstantiable : MonoBehaviour, IArgsInstantiable<float, int, string, GameObject>
{
    public float floatArg;
    public int intArg;
    public string stringArg;
    public GameObject gameObjectArg;

    public void Initialize(float arg1, int arg2, string arg3, GameObject arg4)
    {
        Debug.Log(
            $"I was instantiated with args: args1: {arg1} args2: {arg2} args3: {arg3} args4: {arg4}");
        floatArg = arg1;
        intArg = arg2;
        stringArg = arg3;
        gameObjectArg = arg4;
    }
}
```

This function "Initialize" will then be called just after the awake and after the object is in fact instantiated. To instantiate this object, for example, you would do it like this:

```
var floatIntStringGo = Molecular.Instantiate(prefab, 23.3f, 342, "I am a string", someGo);
```

It is simple like that to use it and implement it!

In the **Demo folder** there is a great example of this called `InstantiateWithArgsDemoObject`

Additional Notes

- You can have more than one these interfaces per object, just make sure that they are well defined.
- Reflections is NOT used at all in this system, so it adds virtually none overhead to just instantiating normally the prefab

Auto Singleton

Overview

Usage: `AutoSingleton<T>` (e.g. `MyMonoBehaviour : AutoSingleton<MyMonoBehaviour>`)
Namespace: `MolecularLib`

Need to use a singleton but don't want to write the same stuff every time? Just do this:
In your `MonoBehaviour` class, change the parent class from `MonoBehaviour` to `AutoSingleton<YOUR TYPE HERE>`, so that it looks like this:

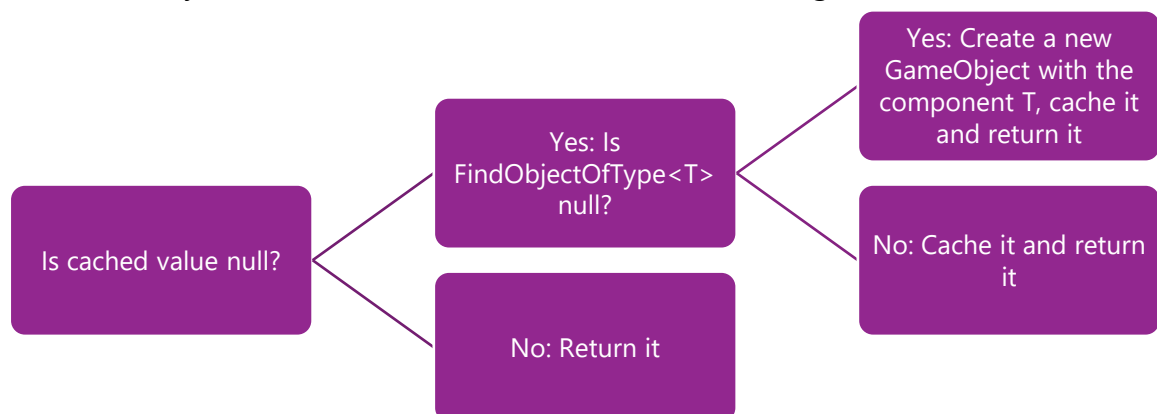
```
public class TimerManager : AutoSingleton<TimerManager>
```

Using it this way, from anywhere in the code you will be able to access `TimerManager.Current` and get a `TimerManager` instance!

Note: `AutoSingleton<T>` derives from `MonoBehaviour`

How it gets the instance?

In some ways: First, before you call `Current` for the first time, it has no idea where is this instance. When you call it the first time it will do the following:



This way it should never return null.

Timers

Overview

Namespace: MolecularLib.Timers

Well, it is kind of obvious what it does, so let's jump right into how to use the timers. There are two types of timers in the library:

- Timer Async
- Timer Coroutine

And the timer Async can generate a TimerReference class.

Both types of timers have the same functionalities:

- OnComplete / OnFinish
- Duration
- ElapsedSeconds
- Repeat

The main difference between Timer Async and Timer Coroutine is the way they work: Timer Async, as the name suggests, uses an asynchronous function to count the time using `Task.Delay(secondsToFinish)` and the `ElapsedSeconds` is calculated via `UnityEngine.Time.time - Timer.StartTime` (Timer.StartTime is set to `UnityEngine.Time.time` when the timer starts).

Now the Timer Coroutine uses a coroutine to count time, using `yield return new WaitForSeconds(delay)` and the `ElapsedSeconds` is also calculated via `UnityEngine.Time.time - Timer.StartTime` (Timer.StartTime is set to `UnityEngine.Time.time` when the timer starts).

Aside from how they work, **TimerCoroutine will instantiate 1, and only one thru the entire program execution, GameObject** with the TimerManager script (It has a singleton by the way). While the Async timer will not. **The Timer Async CANNOT be paused or resumed, can only be permanently stopped in repeat mode. The Timer Coroutine CAN be paused and resumed, even in repeat mode.**

The way I suggest looking at this is like so: The Async timer is a lightweight more basic version of the TimerCoroutine. If you just need a timer, real quick for some real basic stuff, use the Async, if you need it to be more controllable, use the Coroutine.

Timers

Timer Async

Namespace: MolecularLib.Timers

Usage:

- Timer.TimerAsync(seconds :float, callback :Action)
- Timer.TimerAsyncReference(seconds :float, repeat = false :bool)

These methods will create start a timer async (virtually no memory allocation on the first one, little on the second) which will use asynchronous functions to count time. The first method is used like this:

```
Timer.TimerAsync(TimerDelay, () => Debug.Log("Finished"));
```

The second one is used like this:

```
var timerReference = Timer.TimerAsyncReference(TimerDelay);

timerReference.OnFinish += () => Debug.Log("Finished");
if (timerReference.HasFinished) Debug.Log("Already finished");
Debug.Log($"There has elapsed {timerReference.ElapsedSeconds} seconds");
```

Where there is written "() =>", if you don't know what that is, it's a [lambda or anonymous function](#), don't worry about it, they aren't needed, you can just pass a normal void parameterless function.

Timer Reference

In the TimerReference class you have:

- bool Repeat { get; }
- float StartTime { get; }
- int DurationInMilliseconds { get; }
- bool HasFinished { get; }
- Action OnFinish { get; set; }
- float ElapsedSeconds { get; }
- int ElapsedMilliseconds { get; }
- StopOnNextCycle() :void

If you are unfamiliar with "{ get; set; }", it just means whether you can set, get or both to that field. Check out [properties](#)

Timers

Timer Coroutine

Namespace: MolecularLib.Timers

Usage: Timer.CreateTimer(seconds :float, callback :Action, repeat :bool)

This method will create a new Timer, and if doesn't already exist, the TimerManager GameObject, with don't destroy on load and hide in inspector hide flag. You can create the timer like this:

```
var timer = Timer.Create(TimerDelay, () => Debug.Log("Finished"));
```

and for repeat:

```
var timer = Timer.Create(TimerDelay, () => Debug.Log("Finished"), true);
```

the returned value "timer" is an instance of the Timer class, containing the following:

- Methods
 - StartTimer() // You don't need to call this to start the timer using Timer.Create
 - ResumeTimer() // Resumes a paused timer
 - StopTimer() // Stops(pauses) the timer
 - RestartTimer() // Restart the timer as if it was starting now
- Properties
 - float StartTime { get; }
 - bool Repeat { get; set; }
 - bool IsStopped { get; }
 - float DurationInSeconds { get; }
 - float ElapsedSeconds { get; }
 - bool HasFinished { get; }
- Events
 - OnComplete

Note: Both the Async and Coroutine timer's 'HasFinished', in REPEAT timers, will only ever get to be true if the timer is stopped, but even than, it's better to use IsStopped or just use the event.

Auto Assign

Overview

Usage: [GetComponent] **OR** [GetComponentInChild] **OR** [GetComponents] **OR** [GetComponentsInChild] **OR** [Find(GAMEOBJECT NAME)] **OR** [FindWithTag(TAG)] **OR** [FindGameObjectsWithTag(TAG)] **OR** [FindObjectOfType(typeof(TYPE))] **OR** [FindObjectsOfType(typeof(TYPE))] **OR** [LoadResource] **on fields or properties** **AND derive** you class from **AutoAssignMonoBehaviour** **OR [UseAutoAssign]** on **class** and call **this.AutoAssign(); in Awake or Start**

Namespace: MolecularLib.AutoAssign

Instead of having several lines in the Awake function calling various functions like GetComponent, FindObjectOfType, etc... you can use the attributes stated in the Usage section in fields or properties so they are automatically assigned when this.AutoAssign() is called. For classes that derive from AutoAssignMonoBehaviour that function is called automatically on the Awake function. Here are some examples of usage, not all.

```
// Remember to add this attribute here on the class if you don't derive it from
AutoAssignMonoBehaviour!
//[UseAutoAssign]
public class DemoAutoAssign : AutoAssignMonoBehaviour
{
    [GetComponent] private Rigidbody2D _rigidbody2D;
    [GetComponent] private Rigidbody2D _rigidbody2DProp { get; set; }
    [GetComponents(typeof(BoxCollider2D))] private List<Collider2D> _colliders;
    [GetComponents(typeof(BoxCollider2D))] private List<Collider2D> CollidersProp { get; set; }
}

[GetComponentInChildren] private Collider2D _collider;
[Find("DEMO DRAWERS")] private GameObject _drawersProp { get; set; }
[FindWithTag("MainCamera")] private GameObject _camera;
[FindWithTag("MainCamera")] private GameObject _cameraProp { get; set; }
[FindGameObjectsWithTag("GameController")] private GameObject[] _gameControllers;
[FindGameObjectsWithTag("GameController")] private GameObject[] _gameControllersProp {
get; set; }
[FindObjectOfType(typeof(DemoDrawersScript))] private DemoDrawersScript _drawersScript;
[FindObjectOfType(typeof(DemoDrawersScript))] private DemoDrawersScript
_drawersScriptProp { get; set; }
[FindObjectsOfType(typeof(DemoDrawersScript))] private DemoDrawersScript[]
_drawersScripts;
[FindObjectsOfType(typeof(DemoDrawersScript))] private DemoDrawersScript[]
_drawersScriptsProp { get; set; }

/* If you can't derive from AutoAssignMonoBehaviour, you can just call the function below
like that
private void Awake()
{
    this.AutoAssign();
}*/
}
//Taken from DemoAutoAssign.cs in the demos folder, in the scene you can test it with the
test method in the context menu
```

Color Helper

Overview

Usage: ColorHelper.METHOD() or myColorVar.EXTENSIONMETHOD()
(e.g. ColorHelper.Random(); e.g. backgroundColor.TextForegroundColor())
Namespace: MolecularLib.Helpers

The ColorHelper static class provides helper functions for the Color class in unity. It contains both static methods and [extension methods](#) for colors, here they are:

Methods

Static Methods

- **Random**
 - Returns a random color
- **Random(byte minRGB = 0, byte maxRGB = 255)**
 - Returns a random color that falls in the range(inclusive) of minRGB and maxRGB
- **FromString(string value, byte minRGB = 0, byte maxRGB = 255)**
 - Uses the GetHashCode function to convert a string to to a color, for the same string it will return the same color always. The color is also on the inclusive range of minRGB and maxRGB
- **GetTextColorFromBackground(Color background)**
 - Based on the appearance luminance of the background color, calculated via "luminance = 0.299 * background.r + 0.587 * background.g + 0.114 * background.b", if the luminance is greater than .5f, it will return a dark color, r: 16 g: 16 b: 16, if the luminance is smaller than .5f a bright color, r: 201 g: 201 b: 201
- **TextColorShouldBeDark(Color background)**
 - Exactly what the method above does except that it returns a bool instead of a color, so if the text color should be dark, return true, if should be bright, false
- **NormalizeToColor(byte r, byte g, byte b, byte a = 255)**
 - Returns a new Color based on values going from 0-255 instead of 0-1
- **FromHex(string hex)**
 - Returns a color based on a hex color string (# is optional, can be in formats: #AABBCC, #ABC, #ABCD, #AABBCCDD, AABBCC, ABC, ABCD, AABBCCDD)

Color Helper

Methods

Extension Methods

- **ToColor32()**
 - Converts the color to a Color32
- **ToHexString(bool addSharpAtStart = true)**
 - Converts the color to a hex string
- **ToHexStringNoAlpha(bool addSharpAtStart = true, bool add00ToEnd = true)**
 - Converts the color to a hex string ignoring the alpha component
- **TextForegroundColor()**
 - Exactly what you would get calling **ColorHelper.GetTextColorFromBackground()** (documentation in the static methods section, the last page)
- **TextForegroundColorShouldBeDark()**
 - Exactly what you would get calling **ColorHelper.TextColorShouldBeDark()** (documentation in the static methods section, the last page)
- **WithR(byte r) | WithR(float r)**
 - Returns a new color with the same values of the provided one, except the "r" value, which will be overridden with the provided one.
- **WithG(byte g) | WithG(float g)**
 - Returns a new color with the same values of the provided one, except the "g" value, which will be overridden with the provided one.
- **WithB(byte b) | WithB(float b)**
 - Returns a new color with the same values of the provided one, except the "b" value, which will be overridden with the provided one.
- **WithA(byte a) | WithA(float a)**
 - Returns a new color with the same values of the provided one, except the "a" value, which will be overridden with the provided one.

Vector Helper

Overview

Usage: myVector.SOME_EXTENSION_METHOD (e.g. position = position.WithY(y => y + 3))
Namespace: MolecularLib.Helpers

There are only extension methods, and the class they are contained in is the VectorHelperExtensionMethods. So they are all used like this: myVector.METHOD()

Methods

"With" and "Without" Methods

The "With" methods are present for every vector type (Vector2, Vector3, Vector4, Vector2Int, Vector3Int). All these methods do is they return the same value except the one that was changed. So "WithX(newX)" will return the exact same Vector except for the x value, which will be newX now. There are two variants for all of these methods: With(newValue) and With(currentValue => [return new value here]). They would be used like the following:

```
var newVec2 = vec2.WithX(x => x + 1);  
var newVec2 = vec2.WithX(5);
```

Remember that Vectors are structs, so they are value types, which means you need to use the return value to get the edited vector. If you only call vec2.With(3), the vec2 var will still have the old value.

The Without methods will simply call With(0), so set an value to 0

```
var newVec2NoX = vec2.WithoutX();  
var newVec2NoY = vec2.WithoutY();
```

Vector Helper

Methods

"To" Methods

The "To" methods are conversion methods that will take an original Vector and change it to an VectorInt or VectorInt -> Vector. Say Vector2 -> Vector2Int, Vector4 -> Vector3Int, Vector2Int -> Vector3. It will use the provided values to create the new vector and RoundToInt to convert from float to int. Values there are created will be 0(Say, converting a Vector2Int to Vector3, the z value will be 0);

Usage will look like this:

```
vec3.ToVec2Int()  
vec3.ToVec3Int()  
vec3Int.ToVec2()  
vec3Int.ToVec3()  
vec3Int.ToVec4()  
vec2.ToVec3Int()
```

"IsBetween" and "IsWithin" Methods

Usage: providedVector.IsBetween(minVector, maxVector);

The "IsBetween" methods will return true if all the components of the provided Vector falls between(less/greater NOT equal) the ones provided as min and max. False if even one of the components aren't

```
minVec4.IsBetween(minVec4, maxVec4); // Returns False  
maxVec4.IsBetween(minVec4, maxVec4); // Returns False  
  
// Returns True  
(new Vector3(.5f, .5f, .5f)).IsBetween(Vector3.zero, Vector3.one);
```

Usage: providedVector.IsWithin(minVector, maxVector);

The "IsWithin" methods will return true if all the components of the provided Vector falls within(less/greater or equal) the ones provided as min and max. False if even one of the components aren't

```
minVec4.IsWithin(minVec4, maxVec4); // Returns True  
maxVec4.IsWithin(minVec4, maxVec4); // Returns True  
  
// Returns True  
(new Vector3(.5f, .5f, .5f)).IsWithin(Vector3.zero, Vector3.one);
```

String Helper

Overview

Usage: myString.SOME_EXTENSION_METHOD (e.g. playerName = playerName.Color(Color.white))

Namespace: MolecularLib.Helpers

This class offers useful methods for working with rich text strings and some for just normal strings. For example:

```
"Hello World!"  
    .ToBuilder()  
    .Color(UnityEngine.Color.green)  
    .Bold()  
    .ToString()  
    .Ellipsis(50, "Label");
```

Methods

The following rich text methods have two overloads, one which manipulates a string, and one that manipulates a StringBuilder. If you will call more than one of these methods in a chain, like how it's done in the example above, it's highly recommended to use the StringBuilder version, as it's way more memory efficient and quicker. To use the string builder version, first get your string and call ".ToBuilder()"

Rich text methods (string version)

- Color(Color color) :string
- Bold() :string
- Italic() :string
- Size(float pixelSize) :string
- NewLine() :string

Rich text methods (StringBuilder version)

- ToBuilder() :StringBuilder
- Color(Color color) :StringBuilder
- Bold() : StringBuilder
- Italic() : StringBuilder
- Size(float pixelSize) : StringBuilder
- NewLine() : StringBuilder

String Helper

Methods

Other Methods

- ToColor() :Color
 - Just calls the ColorHelper.FromString() method, and return it. All that applies to that function applies to this one too. The documentation for that is on page 19
- Ellipsis(float maxWidth, Func<GUIContent, float> calcWidthFunc) :string
 - This function returns a possibly shortened string ending with "..." or the normal string, depending on whether the string fits in the provided maxWidth. In order to check if the string fits the maxWidth, a character to size function conversion is needed. That is what calcWidthFunc is.
- Ellipsis(float maxWidth, GUIStyle style) :string
 - This function returns a possibly shortened string ending with "..." or the normal string, depending on whether the string fits in the provided maxWidth. In order for this to work you need to pass the currently used GUIStyle for the place you will show the string. If you don't have the GUIStyle, check the function above, it is way more versatile.

Ranges

Overview

Usage:

```
public Range myFloatRange;
```

```
[SerializeField] private RangeInteger myIntRange;
```

```
[SerializeField] private Range<MyComparableType> myCustomTypeRange;
```

Namespace: MolecularLib.Helpers

These are a collection of helper classes to express ranges (Not the C# 9 indexes ranges [3..4]). These are really useful for expressing a patrol range for some enemy, or a temperature range for a map generator, or a height range for a NPC, etc...

Moreover, all range classes derive from the master class `Range<T>` which derives from `IRange<T>` which derives from `IRange` where `T` needs to implement `IComparable`. So, if you have a class that implements `IComparable`, it will work with a nice editor and everything in a `Range`.

The `IRange<T>` interface contains two properties:

```
T Min { get; set; }
```

```
T Max { get; set; }
```

The `Range` interface contains one methods:

```
ValidateMinMaxValues();
```

The different range types

Range<T>

The `Range<T>` class is the master class for all the ones bellow, all that's valid for this is valid for the following ones.

It contains two properties:

```
T Min { get; set; }
```

```
T Max { get; set; }
```

And a set of methods:

- `IsInRange(T value) :bool`
- `Clamp(T value) :T`
- `ClampCeil(T value) :T`
- `ClampFloor(T value) :T`
- `ValidateMinMaxValues()`

Ranges

The different range types

The `Range<T>` can also be deconstructed as so:

```
var (min, max) = myRange;
```

and it also implements implicit operators so you can convert your range to and from a `Range`.

Range

The `Range` class inherits from `Range<float>` and implement some extra useful methods and properties:

- `MidPoint :float`
- `Lerp(float t) :float`
- `LerpUnclamped(float t) :float`
- `InverseLerp(float value) :float`
- `InverseLerpUnclamped(float value) :float`
- `Random() :float`

RangeInteger

The `Range` class inherits from `Range<int>` and implement some extra useful methods and properties:

- `MidPoint :int`
- `Lerp(float t) : int`
- `LerpUnclamped(float t) : int`
- `InverseLerp(int value) : float`
- `InverseLerpUnclamped(int value) : float`
- `Random() : int`

RangeVector

The `RangeVector` classes (`RangeVector2`, `RangeVector3`, `RangeVetor2Int`, `RangeVector3Int`) all derive from a tuple, like this: `RangeVector2 : Range<(float x, float y)>`. Because `Vector` doesn't implement `Comparable`, that is needed. Apart from that, it works normally, as the properties are overridden to return a `Vector` and not a tuple.

Ranges

The different range types

RangeVector

The RangeVector classes implements the following:

- Lerp(float t) : Vector
- LerpUnclamped(float t) : Vector
- InverseLerp(Vector value) : float
- InverseLerpUnclamped(Vector value) : float
- Random() : Vector
- GetBoundingBox() : Bounds

The int version of these vector ranges also implement GetBoundingIntBox() :BoundsInt

All the ranges look like this:



The MinMaxRange Attribute

Usage: [MinMaxRange(0, 1)] public Range myLimitedRange ;

This will make so the range can only accept values from between the ones specified in the attribute; Only supported on Range and RangeInteger. The editor will look like this:



Limits of the generic range class

The generic type must implement IComparable.

Maths

Overview

Usage: Maths.SOME_METHOD()

Namespace: MolecularLib.Helpers

This is a static class that implements some little useful function involving maths. Here are the functions:

Methods

- Lerp(float a, float b, float t) :float
- InvLerp(float a, float b, float value) :float
- InvLerp(Vector2 a, Vector2 b, Vector2 value) :float
- InvLerp(Vector3 a, Vector3 b, Vector3 value) :float
- InvLerpClamped(Vector2 a, Vector2 b, Vector2 value) :float
- InvLerpClamped(Vector3 a, Vector3 b, Vector3 value) :float
- Remap(float iMin, float iMax, float oMin, float oMax, float v) :float
- RemapClamped(float iMin, float iMax, float oMin, float oMax, float v) :float
- IsAllGreaterThan(Vector3 reference, Vector3 toBeGreater) :bool
- IsAllGreaterThan(Vector2 reference, Vector2 toBeGreater) :bool
- IsAllSmallerThan(Vector3 reference, Vector3 toBeGreater) :bool
- IsAllSmallerThan(Vector2 reference, Vector2 toBeGreater) :bool
- IsWithin(this float v, float min, float max) :bool
- IsBetween(this float v, float min, float max) :bool

Lerp stands for Linear interpolation. The lerp functions here are unclamped, for the clamped ones and the ones for Vectors check `UnityEngine.Mathf.Lerp` and `Vector.Lerp`.

The InverseLerp methods do the opposite of the lerp, instead of getting a value that is t% of the way between a and b, it gets the t% value of a given value v between a and b. For a more in-depth explanation of this, check [this video](#) by Freya Holmer.

The Remap methods remap a value from the given input range to the output range. It's useful if you need to change the domain of a value. Say you have a world generator that gives you values from -1 to 1, but you need those to be in the range of 0 to 10, you would use the `Remap(-1, 1, 0, 10, value)` to get the value between 0 and 10 and not between -1 and 1. For a more in-depth explanation of this, check [this video](#) by Freya Holmer.

`IsWithin` checks if the value is greater or equal and smaller or equal than the min and max respectively. `IsBetween` checks if the value is greater and smaller than the min and max respectively. The other methods are very self-explanatory.

Other Helpers

Play Status

Usage: `PlayStatus.IsPlaying`

Namespace: `MolecularLib.Helpers`

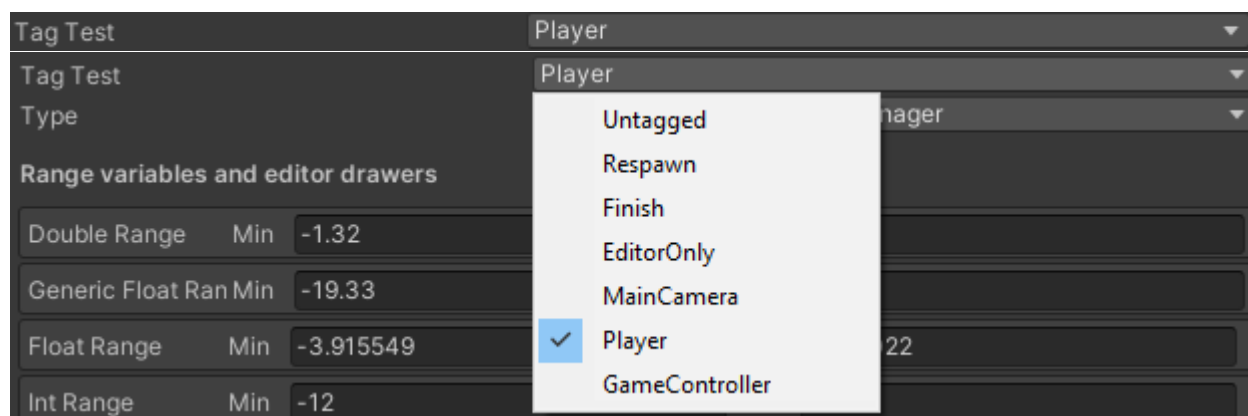
This is the simplest class in this library. When the game scene load, it sets its `IsPlaying` property to true. Yes, it does the same than `Application.IsPlaying`. This class only exists because not always you can call `Application.IsPlaying`.

Tag

Usage: `public Tag playerTag;`

Namespace: `MolecularLib.Helpers`

This provides a safe tag wrapper that makes so you don't need to use string bindings at all while working with tags in unity! One less point of failure! Just declare as a field in your `MonoBehaviour` a tag like this: `public Tag myTag;` and check out the editor. It should look like this:



You can then call

```
tagTest.CompareTag(gameObject);  
// Or  
gameObject.CompareTag(tagTest);
```

You can also get the tag as a string using the `Tag.TagName` property. It also offers implicit operators to do the conversion from `Tag` to string and string to `Tag`. The tag class also offers static `Tag` values for the default Unity Tags that can be accessed like `Tag.MainCamera`.

Other Helpers

Optional

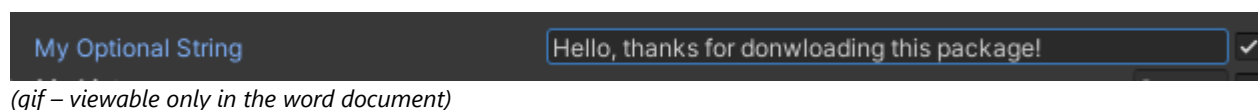
Usage: `public Optional<int> myOptionalInt`

Namespace: `MolecularLib.Helpers`

This is extremely useful for when you want to have a value that can be optional. In the code it will look like this:

```
[SerializeField] private Optional<string> myOptionalString;
```

In the editor it will look like this:



Where you can toggle the right toggle to enable or disable the value. In code you will access that like this:

```
if (myOptionalString.HasValue)
    Debug.Log(myOptionalString.Value);

// Or simply (Using implicit operators)

if (myOptionalString)
    Debug.Log(myOptionalString);
```

Timed Scope

Usage: `using (var timer = new TimedScope("Timer Completed")) {}`

Namespace: `MolecularLib.Helpers`

Creates a scope in which the code executed within will have its time of execution measured and outputted to the debug console in the following pattern:
YOUR MESSAGE completed in TIMEms

Other Helpers

Type Library

Usage: `TypeLibrary.AllAssembliesTypes`

Namespace: `MolecularLib`

Well, sometimes it is needed to access all the types that are present in the project (mostly for the next helper topic). To simplify that process and avoid it being done twice or more, the `TypeLibrary` static class will, after the assemblies are loaded, fetch and cache all the types present in the app domain. If you don't want to use it, and neither want to use the `Type Variable` nor the `PolymorphicVariable`, remove `USE_TYPE_LIBRARY` from the **Scripting Define Symbols** in **"Project Settings > Player > Script Compilation > Scripting Define Symbols"**.

The class provides 3 properties:

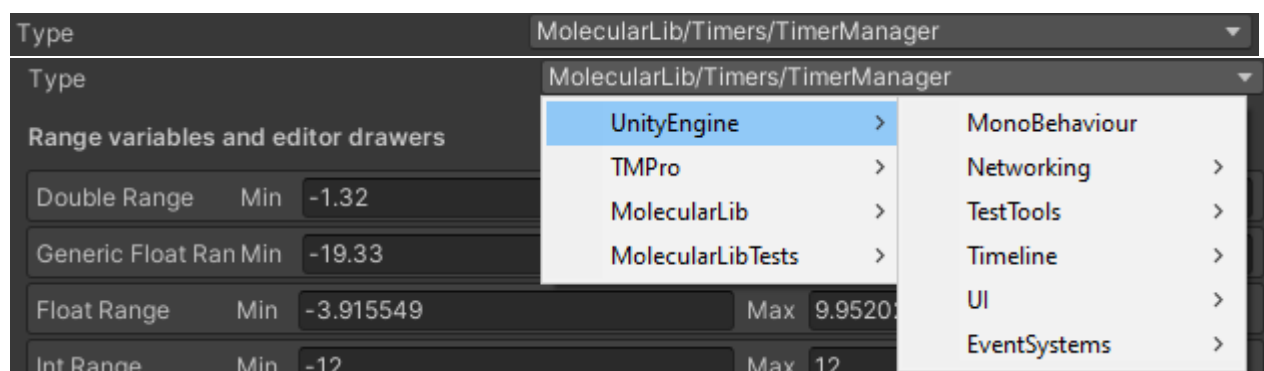
- `AllAssembliesTypes { get; } :IEnumerable<Type>`
- `AllNonUnityAssembliesTypes { get; } :IEnumerable<Type>`
- `AllAssemblies { get; } :IDictionary<string, Assembly>`

Type Variable

Usage: `public TypeVariable<Base> myType;`

Namespace: `MolecularLib.Helpers`

There are two ways of using the type variable: `TypeVariable<Base>` and `[TypeVariableBaseType(typeof(Base))] TypeVariable`. But both of them accomplish the same result: A variable that stores a type in the object:



If you go back to the first section of this manual, `PolymorphicVariable<Base>`, you will see that it uses this in its implementation. In the code you access the type via the implicit conversion or using `TypeVariable.Type`.

Editor Helper

Overview

Usage: `EditorHelper.METHOD()`

Namespace: `MolecularEditor`

This class offers useful functions for handling editor drawers and custom inspectors. Most of the methods are used internally, but as they still are pretty useful, the class and methods are left as public. The methods can be split up in 3 categories: `UIElements`; `IMGUI`; `General Utilities`;

Methods

“UIElements”

- `LinkNodes(this GraphView grapView, Port outputPort, Port inputPort)`
- `ButtonWithText(Action clickEvent, string text) :Button`
- `ToolBarButtonWithText(Action clickEvent, string text) :ToolBarButton`
- `MakeBorder(this VisualElement container, float borderWidth, Color borderColor, float borderCornerRadius = 0)`
- `MakeTopBorder(this VisualElement container, float borderWidth, Color borderColor, float borderCornerRadius = 0)`
- `MakeBottomBorder(this VisualElement container, float borderWidth, Color borderColor, float borderCornerRadius = 0)`
- `MakeLeftBorder(this VisualElement container, float borderWidth, Color borderColor, float borderCornerRadius = 0)`
- `MakeRightBorder(this VisualElement container, float borderWidth, Color borderColor, float borderCornerRadius = 0)`

“IMGUI”

- `DrawBoxWithTitle(Rect totalPos, GUIContent tittle) :Rect`
- `BeginBoxWithTittle(GUIContent tittle, params GUILayoutOption[] options) :Rect`
 - Use this only in custom editor, not in drawer
- `EndBoxWithTittle()`
 - Use this only in custom editor, not in drawer
- `AutoTypeFieldInfo(ref Rect rect, FieldInfo fi, object targetObj, string uniqueIdentifier, string label = null)`
 - Check the item below the item below, this method just calls it with conversions

- `AutoTypeInfo(ref Rect rect, PropertyInfo pi, object targetObj, string uniqueIdentifier, string label = null)`
 - Check the item below, this method just calls it with conversions

Editor Helper

Methods

- `AutoTypeField(ref Rect rect, Type valueType, object value, string uniqueIdentifier, string, labelStr = null, IList<CustomAttributeData> attributes = null) :object`
 - For whenever you have a field, but it's not in a `SerializedProperty`, but you would really like to use `EditorGUI.PropertyField()`. This function simply does the right editor for the provided type. It even supports Custom Property Drawers and Property Decorators! (You need to pass the parameter attributes of the field for property decoratos)
- `AutoTypeFieldGetHeight(Type valueType, object value, string uniqueIdentifier, string, labelStr = null, IList<CustomAttributeData> attributes = null) :float`
 - Gets the height of the to be drawn field using `AutoTypeField()`;
- `TypeField<TBaseClass>(Rect rect, string label, Type currentValue, bool showBaseType) :Type`
- `TypeField(Rect rect, string label, Type currentValue, Type baseType, bool showBaseType) :Type`
- `DrawTypeField(Rect rect, string label, List<Type> types, Type current) :Type`

"General Utilities"

- `UnitySerializesBindingFlags :BindingFlags { get; }`
- `AllTypes :IReadOnlyList<Type> { get; }`
 - List of all types gotten from the `TypeLibrary` class mentioned in page 31
- `GetTypesForPopup<TBaseClass>(bool showBaseType) :List<Type>`
- `GetTypesForPopup(Type baseType, bool showBaseType) :List<Type>`
- `Tex2DOfColorScreenSize(Color32 color) :Texture2D`
- `Tex2DOfColorAndSize(Color32 color, int width, int height) :Texture2D`
- `Tex2DOfColorScreenSize(Color color) :Texture2D`
- `Tex2DOfColorAndSize(Color color, int width, int height) :Texture2D`
- `Tex2DOfColor(Color color) :Texture2D`
- `GetTargetValue<T>(SerializedProperty property) :T`
 - This functions returns the actual value contained inside a `SerializedProperty`. A great example of its use is in the `SerializableDictionary` and `Range` editors. If you are doing a custom editor to a custom type of yours that doesn't derives from `UnityEngine.Object` you probably got stuck trying to access the actual value of the property. This function does exactly that.
- `GetTargetValue (SerializedProperty property) :object`
 - Same as above, except it's not generic.

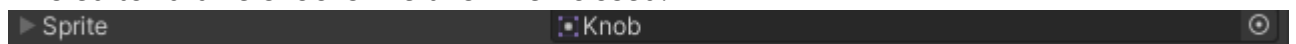
- `GetPropertyDrawer(Type classType) :Type`
 - This function returns the type of the custom property drawer for the `classType` if it exists, else it returns null.

Enhanced Sprite Editor Drawer

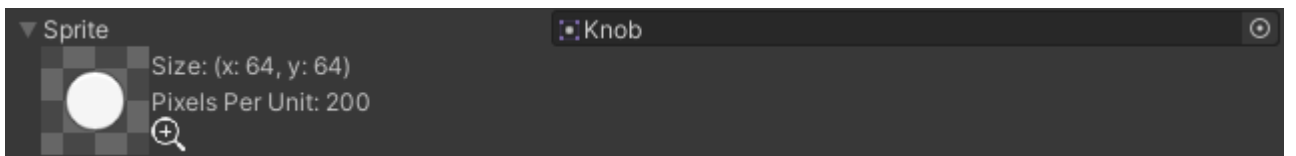
Overview

Usage: Just go on normally with your day declaring normal Sprites
 Namespace: Not Applicable ;)

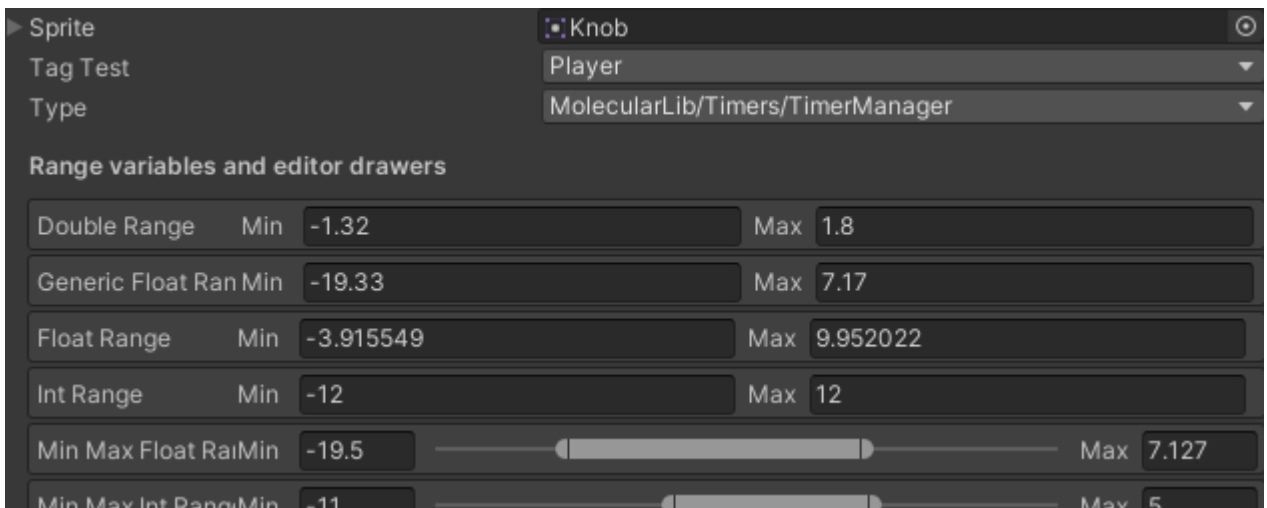
This editor drawers looks like this when closed:



And like this when opened



Where if you click and drag the zoom icon you can increase or decrease the size of the preview.



(gif – viewable only in the word document)

How to disable it

In case you don't like this editor drawer, or want to do your own, just go here
"HandsomeDynosaur/MolecularLib/Core/Editor" and inside that folder delete the file
"SpriteEditorDrawer.cs". Just keep in mind that if you update the package the file will
 come back.

Know bug

If you use the Sprite drawer inside Lists and arrays, you'll see that it works fine except when trying to zoom in and out. For now, I haven't been able to fix it.