# Just Logic

## *Introduction*

Just Logic is a Unity extension with the visual programming functionality. With JL you can create the logic for the game right in the inspector. Just Logic can access other scripts and objects and use any other plug-ins and dll libraries.

## *Platforms*

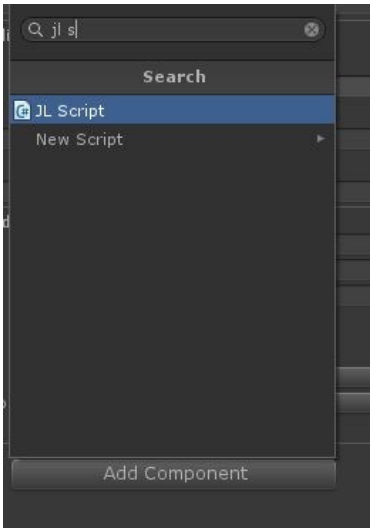
Standalone, Web Player, iOS and Android are target platforms. Other platforms possibly can run JustLogic but it's not guaranteed.

## *Installation*

Import JustLogic.unitypackage using "Assets/Import Package/Custom Package" menu.

Be aware that sometimes Unity behaves unstable when importing the package. This is a known issue. It's related to the "First Person Controller.prefab" (used in the Tutorials) inside Standard Assets. It should be ok after the editor restart. It's no connected to the JustLogic itself so if you have this issue please report this crash to the Unity support.

## *Units*




To make a new JL Script, create a new game object (Game Object / Create Empty) and add to it the component JL Script (by pressing the button "Add Component" inspector and entering the component name in the search box).

The script is composed of units - *actions* and *expressions*.

Each unit has its own settings called *parameters*. Usually parameter values - *the arguments* – are evaluated first before running the action itself. The inspector shows optional *parameters* in square brackets.

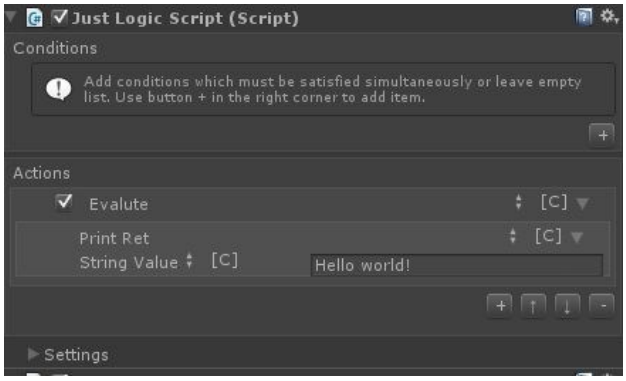
*An expression* differs from an *action*: it has returning value - the result of its work. This value can be immediately used as an *argument* for another unit.

Examples of *actions*: move the object, load level, slow down time.

Examples of *expressions*: find the main camera, compare two numbers, get the length of the string.

| | *Action* | *Expression* |
|---|---|---|
| Returns value | - | + |
| Parameters may contain another *action* | + | - |
| Parameters may contain another *expression* | + | + |
| Can be yielded (timers) | + | - |

Before launching the script the *conditions* check runs. They are specified in the inspector *Conditions* list. The script will not be started until all *conditions* are met.




Each *condition* is an *expression* that returns a value of type *bool* ("yes/no" — True/False). When all the *conditions* have returned True, the execution of the script begins.

*Actions* are specified in the Actions list and executed sequentially from top to bottom. The sequence of *actions* may be combined with the *Sequence* unit.

*Hint:* check box at the left corner of the *action* name disables the unit so it is not executed.

To replace an *expression* or *action* with another type, click on its name and select an appropriate option in the "Select Unit" window (for convenience, it can be docked in the editor). If the correct *expression* is not listed, perhaps it does not match the type of the parameter whose value you are setting.

*Hint:* the unit can be copied by selecting an appropriate menu item from the unit context menu (opens with right-click).

*Actions* are not available to be used as *expression* parameters. Mathematically speaking, *an expression* is a function that takes parameters. Each parameter can be a constant or another function and it must be specified. *An action* does not return a value, so it cannot be passed in the parameter.

If you want to execute *an expression* without using a return value (as if it were an *action*), it should be packed into a special *Evaluate action*.

## *Events*

A script cannot be run by itself. Usually it is started by a specific *event handler*. Each handler has its own defined list of conditions that allows instant checking of the triggered event parameters.

General procedure for running the script looks like:


1. An *event handler* is being triggered
2. Checking *the event handler conditions*
3. Triggering a script
4. Checking the script *conditions*
5. Executing the script *actions*

*Event handler* is a component ("JL Event Handler"), which can be added to the same game object to which the triggering script is added or to any other game object in the scene.

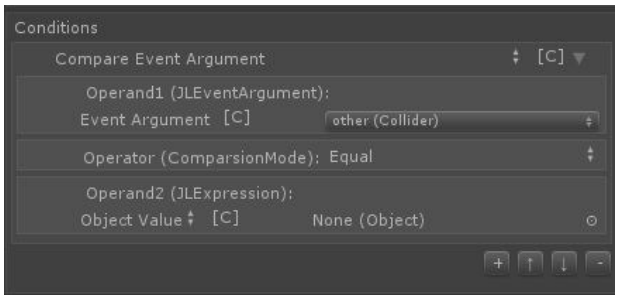

Parameter "Event" specifies the event when the handler will be triggered.

Learn more about Unity events at [http://docs.unity3d.com/Documentation/Manual/EventFunctions.html](http://docs.unity3d.com/Documentation/Manual/EventFunctions.html)

Documentation on all Unity events is here
http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.html

The list of expressions “JLScripts” specifies scripts which must be triggered.

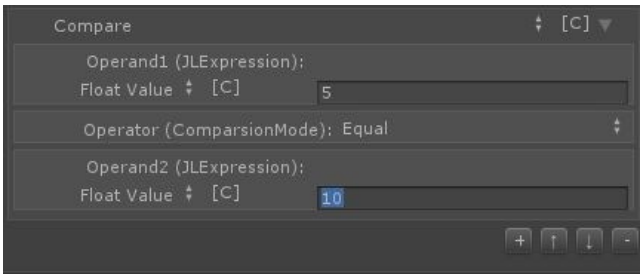
The list “Conditions” specifies the *conditions* of the handler. Only if all *the conditions* are met, the script will be executed. It's recommended to debug the script without specifying the *conditions*, and add them only after making sure the script works as expected.




The *Event* parameter (like *other* in OnCollisionTrigger event) can be compared with any value using the expression “Event / Compare Event Argument”.

## *Expression: Compare*

The expression is intended to compare two values. Returns bool value: True or False.




This expression differs from the mentioned “Compare Event Argument” only in that its “Operand1” is available for editing.

Where possible, the type conversion is automatic. For ~~an~~ example you can compare Game Object and Component on this or another Game Object.

| | |
|---|---|
| Returning value | *bool* |
| Parameters “Operand1” and “Operand2” | Two compared operands (*expressions*). |
| Parameter “Operator” | The comparison type: equality, inequality, greater than, smaller than, greater or equal, smaller or equal. |

## *Action: Branch / If*

*If* accepts *bool* argument and depending on its value executes one of two *actions*.

Hint: use “Branch / Sequence” *action* to combine a list of actions into one.

Parameters:

| | |
|---|---|
| Value | *bool expression,* |
| Then | action is executed if “Value” equals *True,* |
| Else | action, is executed if “Value” equals *False.* |

## *Expressions group: Logical*

The following units are designed to compare the *arguments* of the *bool* type (*True* or *False*). The result of comparison returns as a *bool* value.

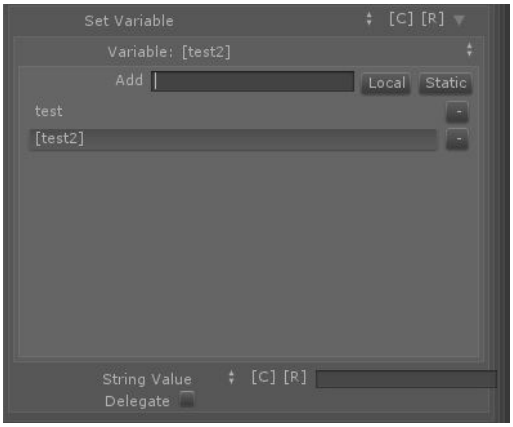
*The* expressions below return *True* in case that

| | |
|---|---|
| And | all argument values equal *True* or they are not specified, |
| Or | at least one argument value equals *True*, |
| Xor | only one argument value equals *True*, |
| Not | the argument value equals False (inverted value). |

In other cases these *expressions* return *False*.

*Expression IIf* is similar to action *If,* but it executes one of the two expressions depending on the *bool* Value.

## *Variables*




*Variables* allow temporary storage of some data. You can store a value to use it later in your script. "Variable / Set" writes data into the *variable* and "Variable / Get" reads them.

Hint: *Variable / Set* is an *expression* so just stored value can be immediately passed to another unit as an argument.

In addition to the usual variables that hold a value only during the execution of a script, there are another two types of variables:

- *Static variables* retain their value after the execution of a script, and can be read at its next start.

  The inspector displays the name of a *static variable* in square brackets.

- *Global variables* work in the same manner, but they are shared between all scripts at once. They can help you transfer data to other scripts.

To check whether a variable contains a value, use bool-expression Variable / Is Set.

Setting an *expression* as a value of a variable it is possible to delay the calculation of *expression* until the variable is read. To use this option, make sure "Delegate" check box (in "Variable / Set") is on. Every time you read a *variable the expression* will be evaluated again. Note that this option has no effect for *global variables.*

The stored numeric value of a variable can be increased or decreased by one with the "Variable / Increment" and "Variable / Decrement" *expressions* respectively. These *expressions* return the new value of a *variable*.

## *Actions group: Time*

"SetTimeScale" unit determines the speed of the game time (default is 1).

The rest of *actions* in this group add possibility to pause the script for a while:

| | |
|---|---|
| Wait for game time | waiting for a game time (specified in seconds), |

| | |
|---|---|
| Wait for real time | waiting for real time (specified in seconds), |
| Wait for next frame | waiting for the next *Update event*, |
| Wait for next fixed update frame | waiting for the next FixedUpdate event. |

## *Units group: External*

Units of this group are designed to appeal to other scripts and plugins.

***Action "TriggerScript"*** starts the specified JL Script (independently, as it's triggered by an *event handler*). In this case, the current script will be continued even if the started script yields (Wait For ...).

***Action "Call JL Script (scene or prefab)"*** differs from the previous one in that it runs only actions of the target script (ignoring conditions). The current script does not continue until the called script is completed (for example, in the case of using yield timers). Check of the target script conditions is not executed.

***Action "Call JL Script (asset)"*** allows execution of a script that is stored in the form of an asset. This script is stored out of the scene and can be called from several different scenes. To create a script of such type, use the menu "Assets / Create / Just Logic Script".

Similarly, the ***expression "Call Expression (asset)"*** allows running an expression that is stored in the form of an asset (created with menu "Assets / Create / Just Logic Expression"). The return value can be used in the current script.

***Expression "Call Expression (scene or prefab)"*** launches an expression that is stored in the scene (JL Expression component).

***Expression "Invoke"*** allows calling any method of the specified object. You can also specify a type and call the static method on it (such methods in the list have square brackets around the arguments.)


***Action "SendMessage"*** calls a method on a specified object or a script. Learn more about SendMessage at: http://docs.unity3d.com/Documentation/ScriptReference/Component.SendMessage.html

## *Units group: Branch*

| | |
|---|---|
| Return | Stops the current script execution (will not return to the caller script if called in JL Script). |
| Return Script | Stop the current JL Script execution and returns to its caller. |
| Noop | Does nothing. |
| If | Executes *an action* depending of the value of *an expression*. |
| IIf | Returns one of the two values depending of the value of *an expression*. |
| Sequence | Executes specified sequence of *actions*. |
| Evalute Expression | Executes an *expression.* |

| | |
|---|---|
| Exception group | Units for exception handling. |

## *Actions group: Loop*

Units of this group perform a specified action repeatedly.

| | |
|---|---|
| SimpleFor | Executes *an action* specified number of times. |
| For | Executes *an action*, while setting the value of *an expression* from “From” till “To” *arguments*. |
| ForEach | Executes *an action* while ~~setting~~ filling the specified *variable* value with elements of the specified list. |
| While | Executes *an action* while all the specified *conditions* are met. |
| Continue | Jumps to the next loop iteration (inside sequences) |
| Break | Stops an execution of the current loop |

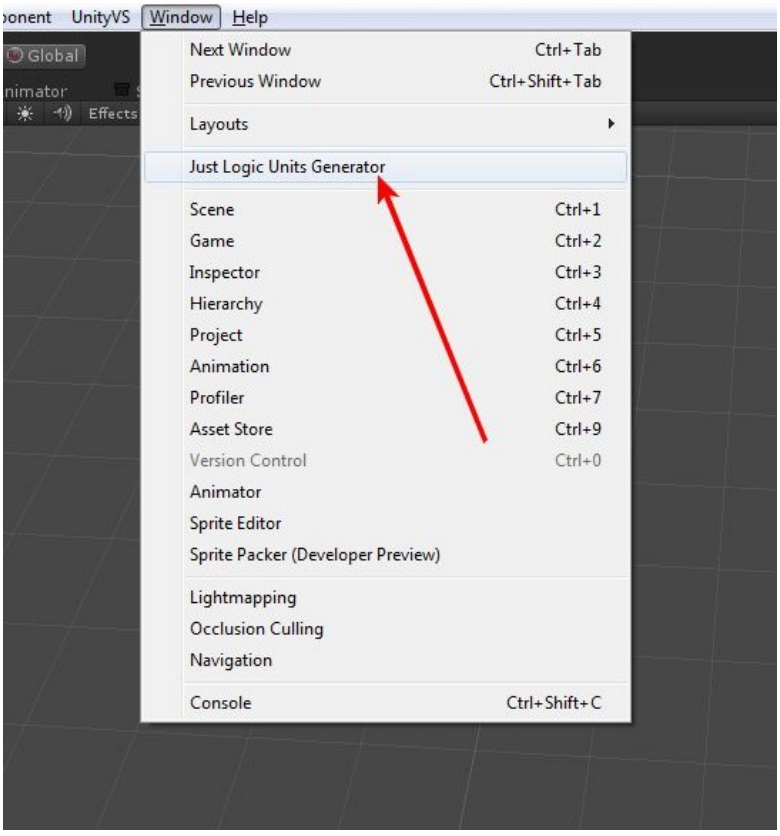
## *Other units*

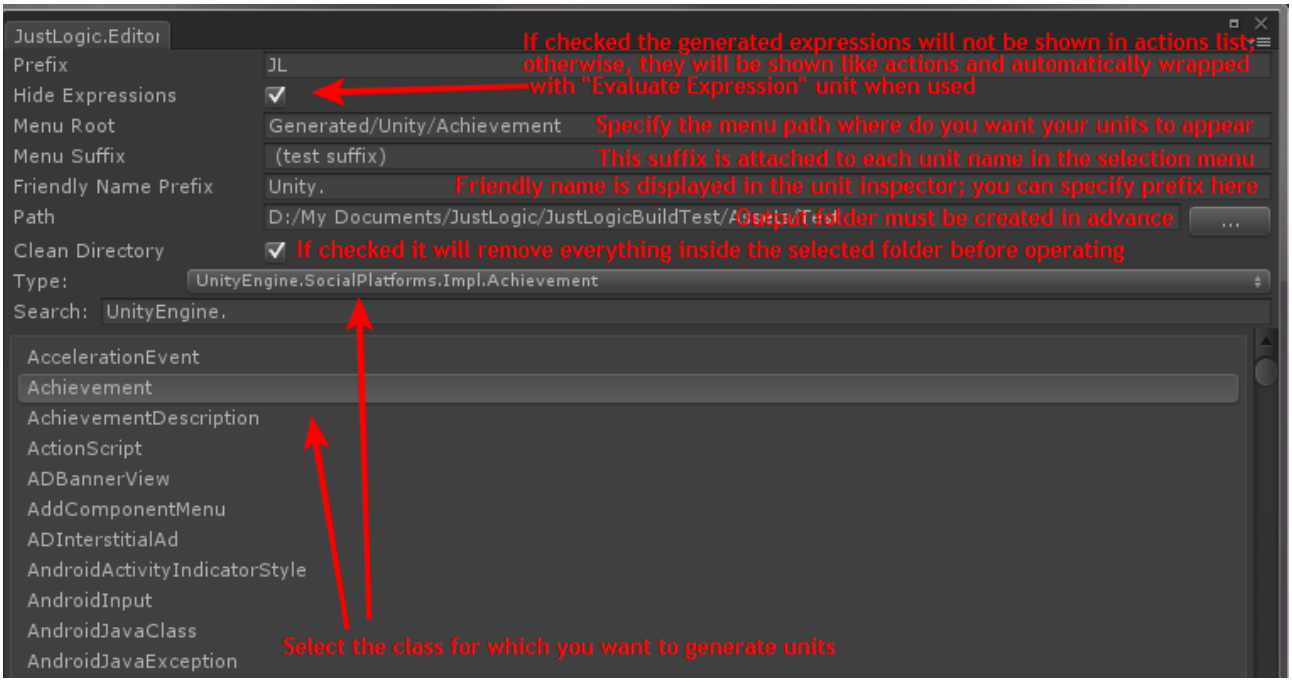
The purposes of the units which were not described here match their name. Normally, Unity documentation describes the function with the same name as that of the unit. For information about such functions please visit the Unity documentation: http://docs.unity3d.com/

| **Units group** | **Descriptions** |
|---|---|
| **Value** | *Expressions* used for setting constant values, as opposed to their calculation during the script execution. |

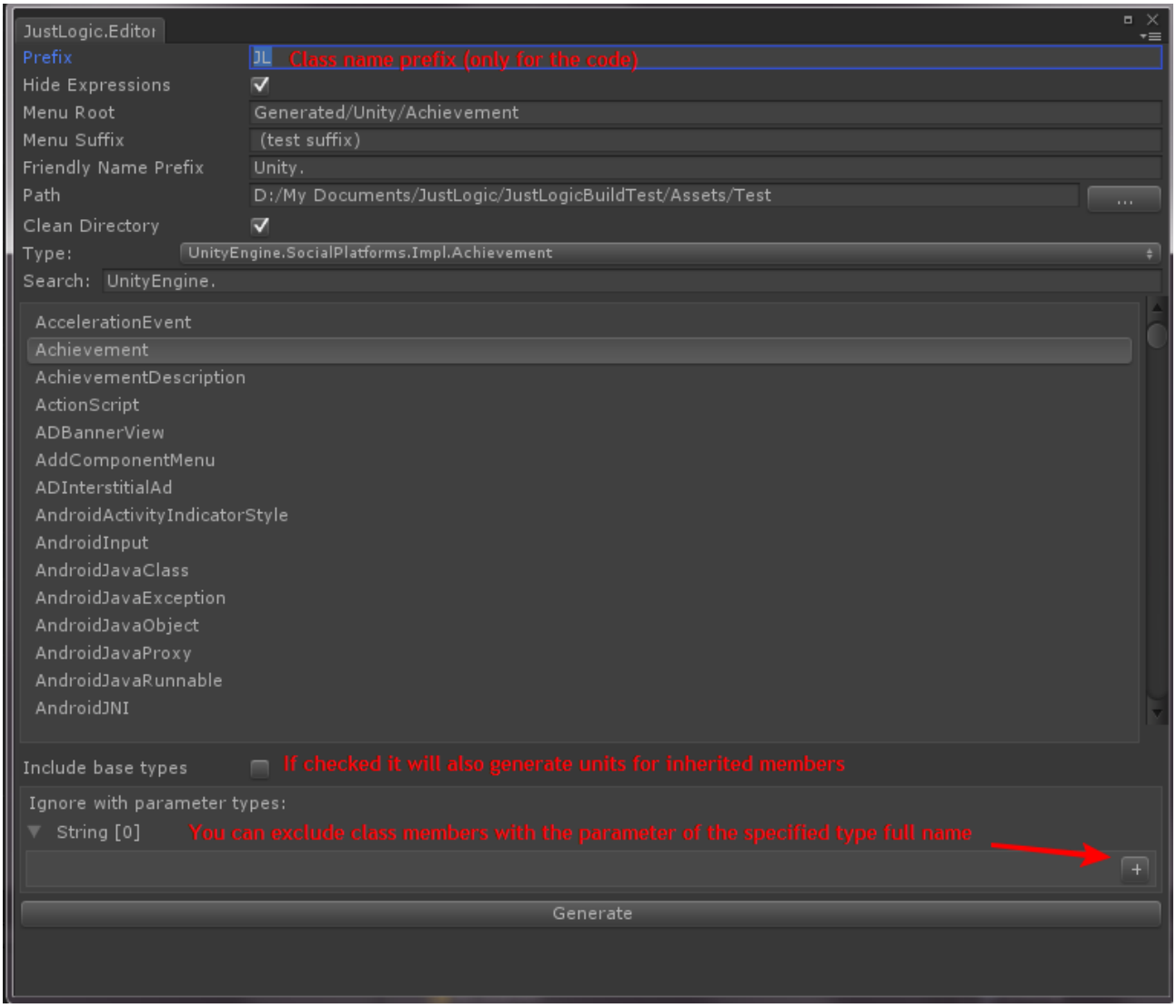
## *Generating units*

You can easily generate units for any class to use in JustLogic. For an example, let’s generate units for the Unity “Achievements” class. Afterwise, you can use them like any other standard JL units. With the same way you can create units for any asset (like NGUI).

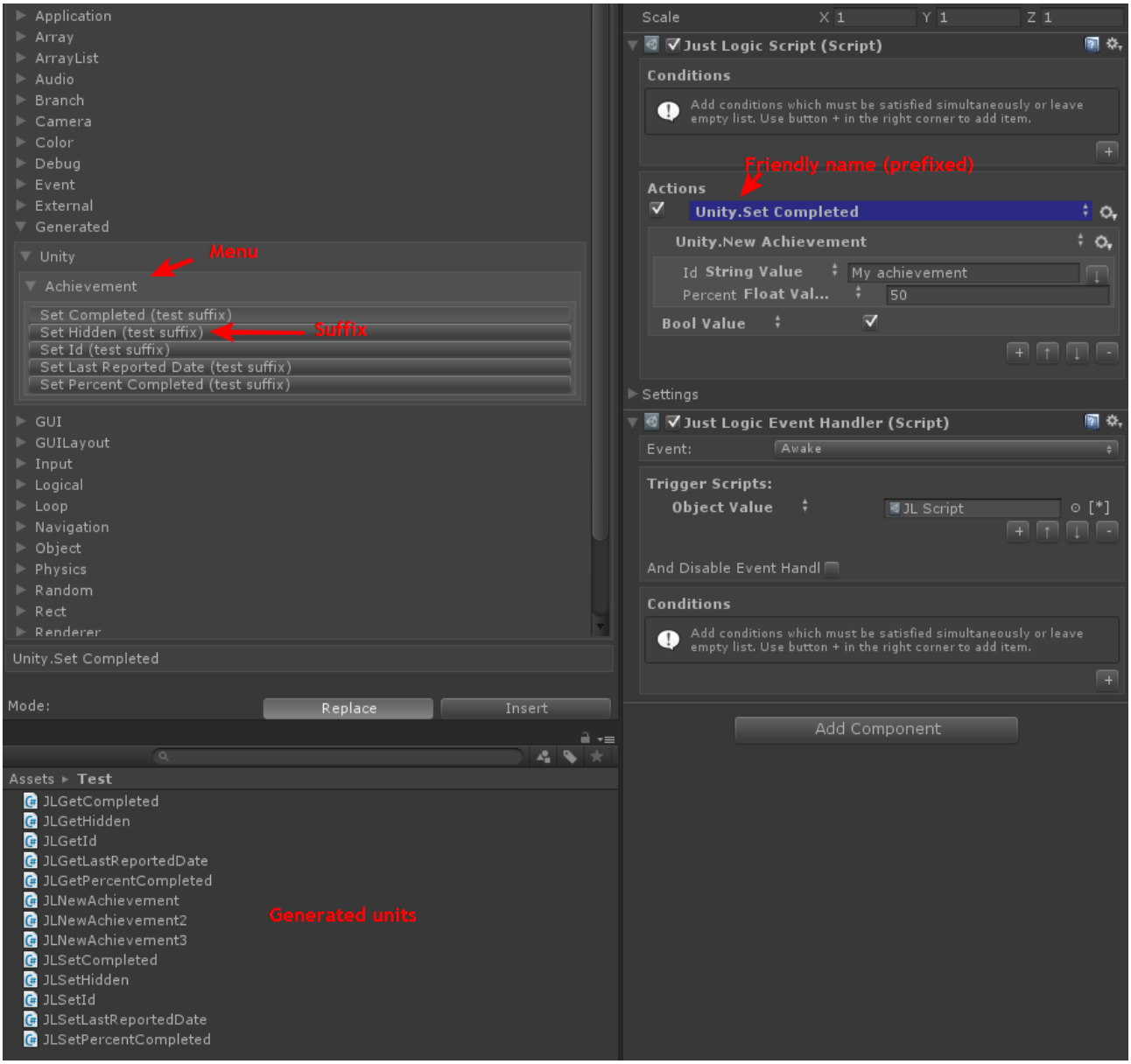





You can also specify advanced parameters like class name prefix and exclude conditions:




Everything is ready, let’s press “Generate”. And what will you see?..




You may want to rename, tweak, or remove some of them. They are editable like any other cs files.
To avoid crashes remember: it's necessary that each unit class has the same name as its file name.

## *Source code*

Some parts of the asset are accessible in the form of a source code.

- Units are merged into the main JustLogic.dll assembly.
  You can found the source code archive in the JustLogic\SourceCode\JustLogicUnitsSourceCode.zip
- Editor source code with some of standard drawers: JustLogic\Editor\Code

## *Compiling units dll*

You may want to precompile your units into a dll assembly (to speed up the compilation) or make some changes to the standard units source code. You don't need to do this operation if you create units directly inside your Unity project. You also shouldn't merge in-editor assemblies (but note that if you inherit Unity Editor classes (not JL drawers) from another dll (like JustLogicCoreEditor.dll) you also need to inherit your classes in Unity project, otherwise they may not become recognized).

In Unity you can't split your scriptable objects from one dll in two or more separate dlls without losing script references. For compatibility causes it's recommended that all units are merged into one big JustLogic.dll assembly.

This instruction is not about using MonoDevelop or Visual Studio to create a new dll project for Unity. Please learn your favorite IDE from its own documentation and community.

You can find the original JustLogic.dll assembly without standard units in the source code archive. Your assembly may (and is supposed to) reference that JustLogic.dll. After build you can use IL Merge for Windows or alternatives for Mac to merge all the dlls into JustLogic.dll.

On Windows I use a post build cmd file. Assume you have your Unity project at path "D:\Project" (and Assets folder at "D:\Project\Assets").

1. Place all the dlls (original JustLogic.dll, YourCompiled.dll, YourCompiled2.dll, …) to the "D:\Project\Assets\JustLogic\Plugins" folder. Note that it should not contain UnityEngine.dll or UnityEditor.dll otherwise Unity will not compile your project.

2. Place the ILMerge.exe, your UnityEngine.dll and UnityEditor.dll from "c:\Program Files (x86)\Unity\Editor\Data\Managed\" to the "D:\Project" directory.

3. Create "D:\Project\merge.cmd":

```
cd Assets\JustLogic\Plugins

rmdir Merged /S /Q

mkdir Merged

..\..\..\ILMerge /out:Merged\JustLogic.dll *.dll /lib:..\..\..\ /xmldocs /wildcards

@if %errorlevel% neq 0 pause

del *.dll

del *.xml

xcopy Merged\* Merged\.. /E /Y

rmdir Merged /S /Q

pause
```

   It creates a new *Merged* folder for output dll, runs ILMerge with all the dlls in the *Plugins* folder, clears *Plugins* folder and copies output from *Merged* back to the *Plugins*. Than temporary *Merged* folder will be removed.

   If you can't copy the text from this document, see JustLogic\SourceCode\merge_cmd.txt.

4. Launch it. All done.

It should be easy to do the same procedure using IL Merge alternatives for Mono.

## *Tips*

- Always pay attention to the Console. Even though everything works fine, the console may contain useful information.

- Be careful with object references (*ObjectValue*)! If you pass a destroyed or missing object to an expression, most likely, the exception will be thrown and the execution of the whole script will stop. All exceptions appear in the Console.

- Some items may go self-destructed over time. To detect such objects, watch your script in the inspector while testing the game.

- While testing, you can manually run the script from the inspector. To do this, right-click on the script header in the inspector and select “Start Execution” from the context menu.

- Read the Unity documentation, including the Scripting section. With the help of *the “Invoke” expression* you can use any of the Unity functions. Only the method of script creating is different but the functionality is the same.

Author: Vlad Taranov
Skype: vbprogr
YouTube channel: http://www.youtube.com/user/aqlasolutions
More at http://www.aqla.net