1. Functions

A method (function) is a code block that contains a series of statements (nodes) which will be executed when function will be invoked.

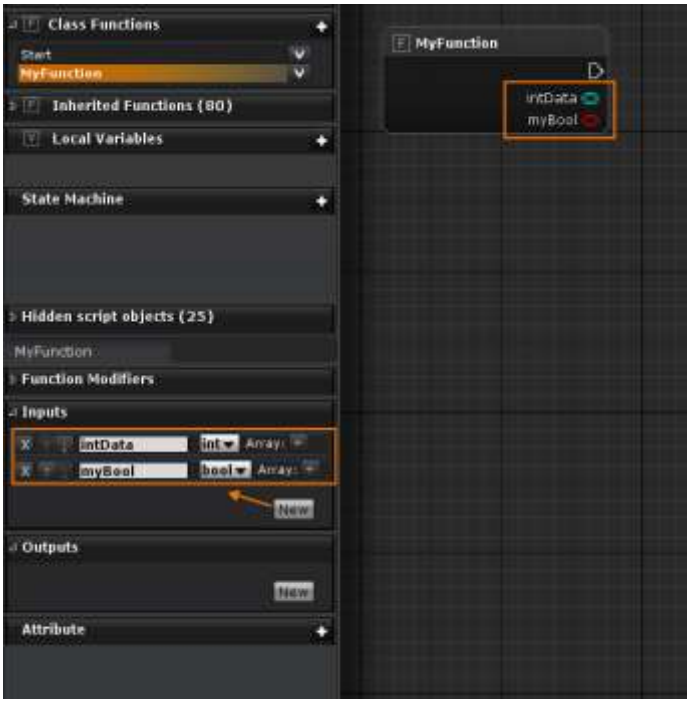
Read Node edit graph for more information about entry&return nodes, links between nodes and how to use sequence and data nodes.

1.1. Creating function

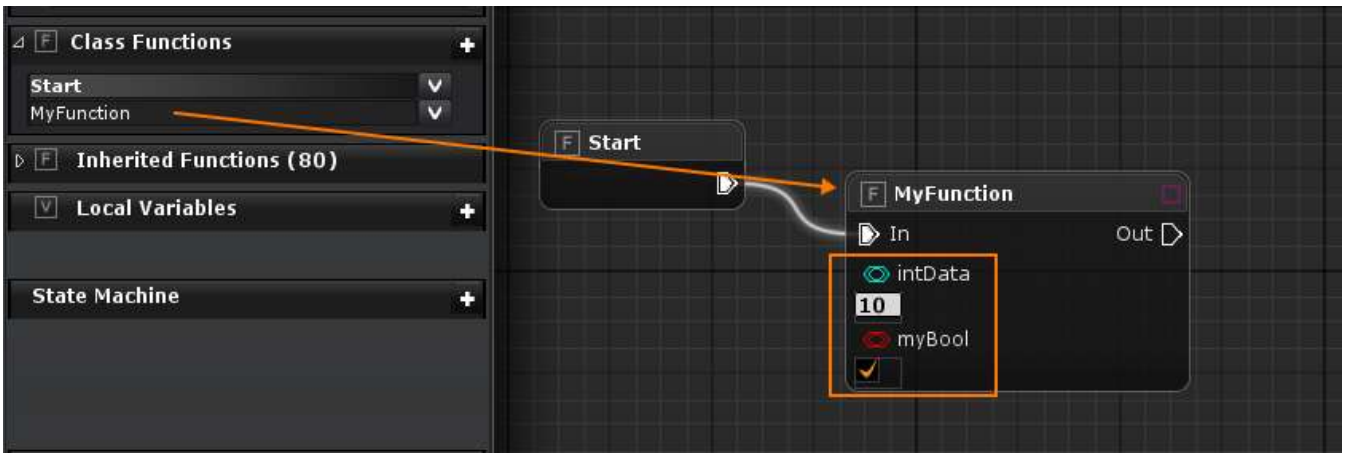
The functions can be created by pressing 'plus' button on Functions section:

1.2. Adding pins

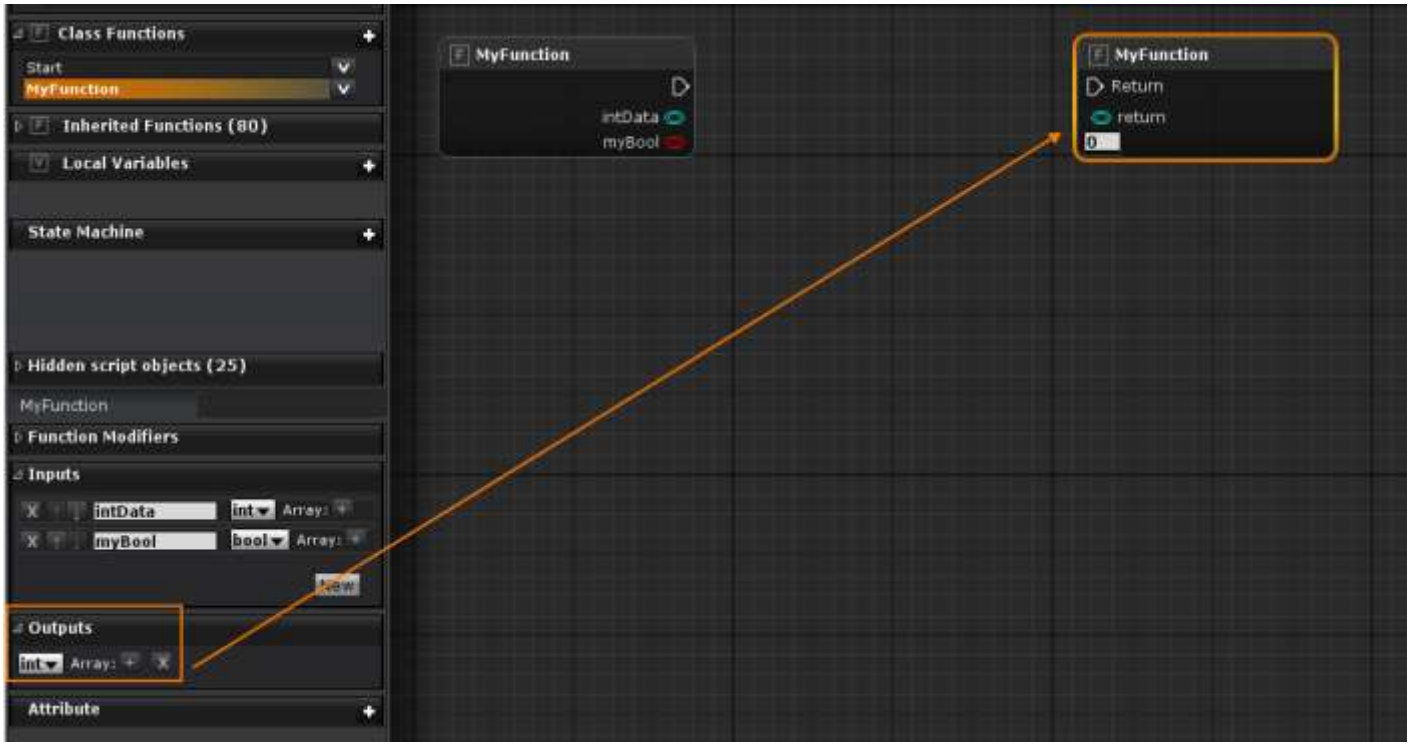
Each function can be invokes with a set of arguments (inputs), which allow to transfer some data (variables) inside function. To add new parameters to function press '**New**' button in '**Inputs**' section of function Details window. Each argument will have own name and type. Arguments with the same name are not allowed.



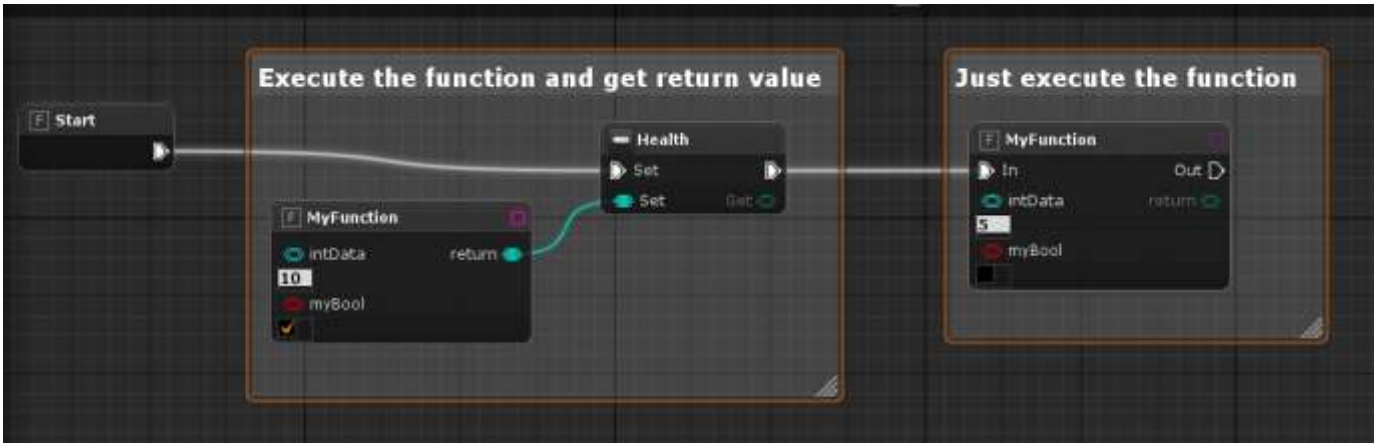
After adding the inputs to our function it can be executed with arguments:



If the function must return some data you can add out pin by '**New**' button in '**Outputs**' section of function Details window:



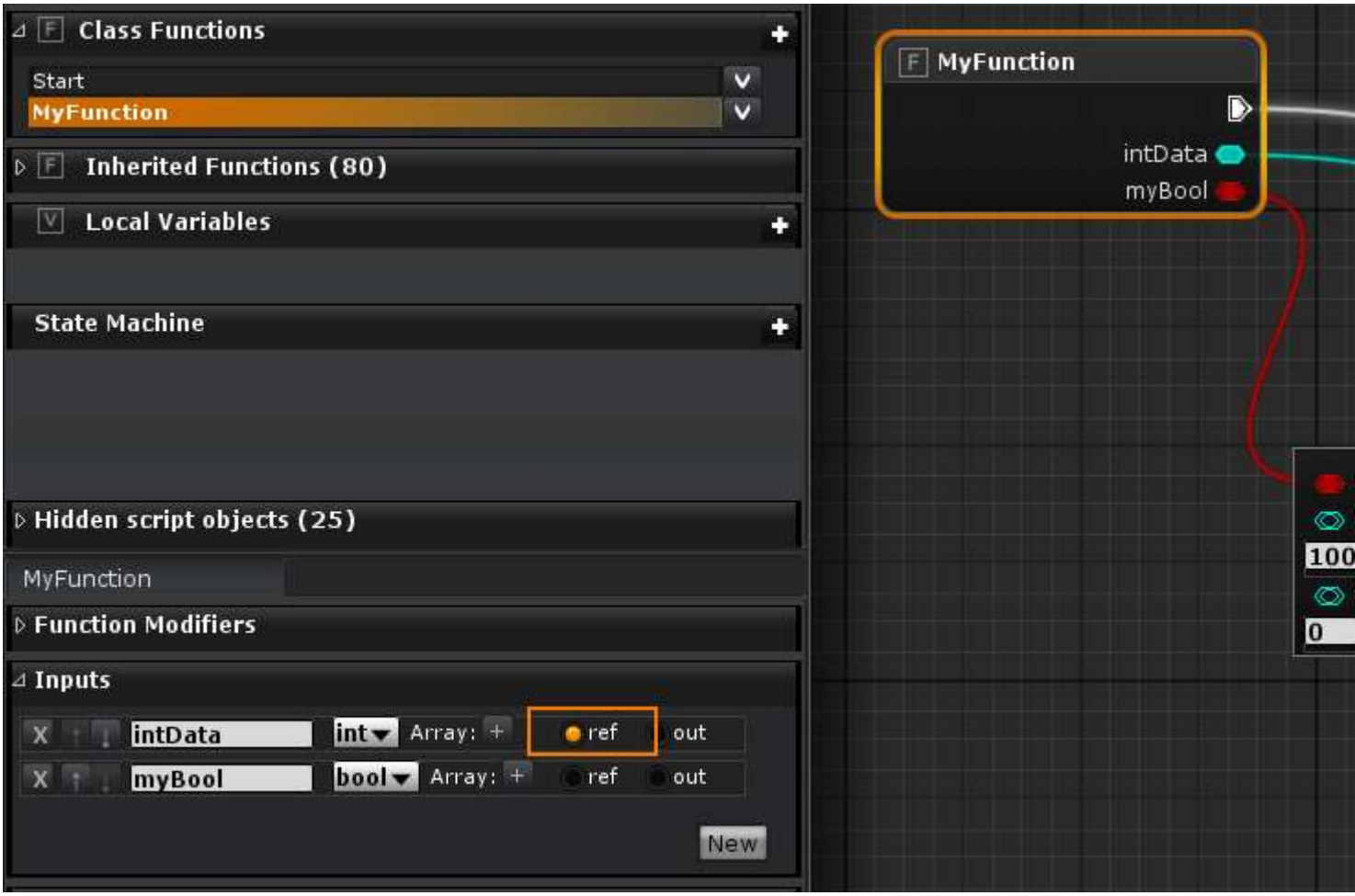
Now the function can return a value or can be executed by sequence connector as simple function:



1.3. Ref/Out parameters

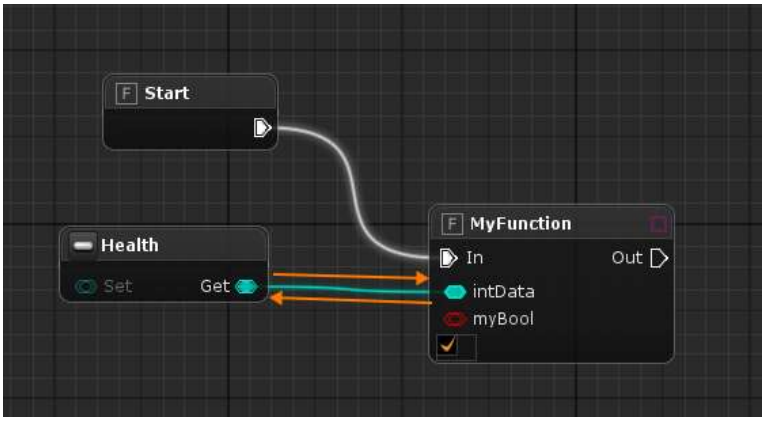
If the function is outside of a class and you need to return more than value from function you can use the '**ref**' and '**out**' parameters of arguments. The '**ref**' parameter causes an argument to be passed by reference, not by value (the value will be returned to a variable that connected to this). The '**out**' parameter is the same, but the value must be set before function finish executing code and exit.

To set the value of '**ref**' or '**out**' parameter use 'Set value' node:



(in this screenshot used the '**Conditional expression**' node, which identifies which value to return based on bool condition)

Now our function can return a value to a variable that connected to argument with 'ref' parameter:

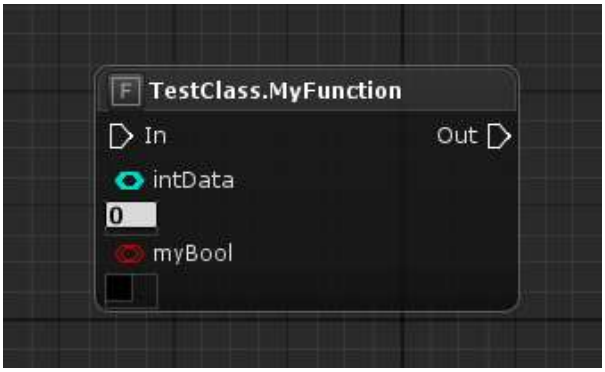


1.4. Function modifiers

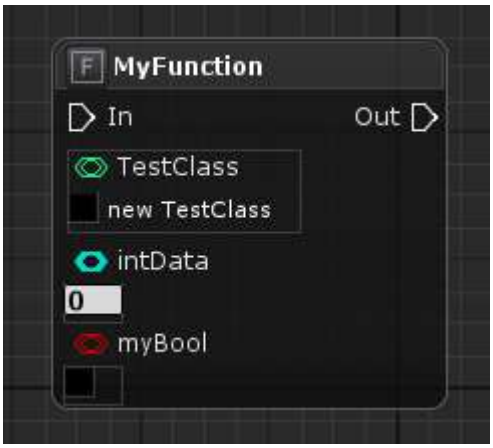
public - access is not restricted.

private - access is limited to the containing type.

static modifier is used to declare a static member, which belongs to the type itself rather than to a specific object. That means the public static function can be executed from type directly:



and not required a instance of class as public non-static function which shown on screenshot below:



Note: you can't use non-static variables in static function.

The **virtual** modifier is used to modify a method and allow for it to be overridden in a derived class. It means that you can create a class 'A' with a **virtual** function and you will be able to **override** the code of this function in class 'B' which derived from class 'A'.

Other modifiers:

protected - access is limited to the containing class or types derived from the containing class.

internal - access is limited to the current assembly.

protected internal - access is limited to the current assembly or types derived from the containing class.

1.5. Function attributes

Function attributes can be used to sign this function for some functionality. To add attribute to a variable press '**plus**' button in **Attribute** section and select it from list.

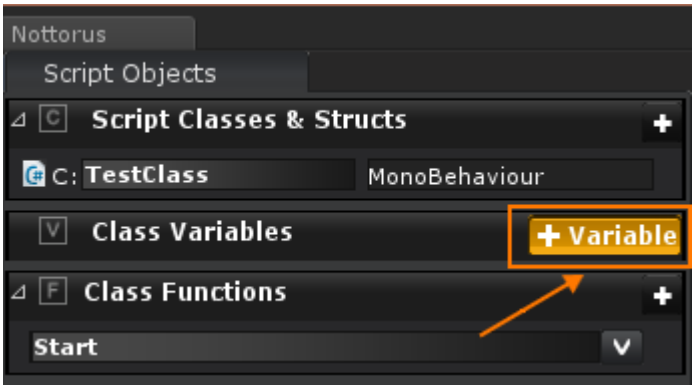
For example the most used function attribute is '**MenuItem**'. It used on static functions in class that derived from '**EditorWindow**' class to notify unity editor which function to execute to open editor window.

2. Class Variables

Variables are a 'boxes' to contain information of defined type.

2.1. Creating class variables

To create a class variable press '**plus**' button on the right side of **Class variables** section:



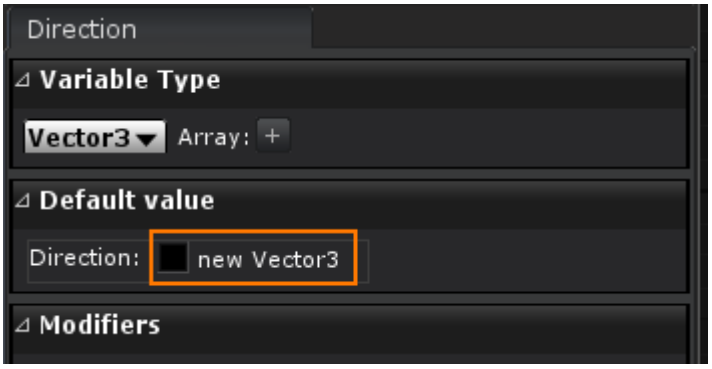
Renaming the variable can be done by double clicking on its name.

2.2. Variable type

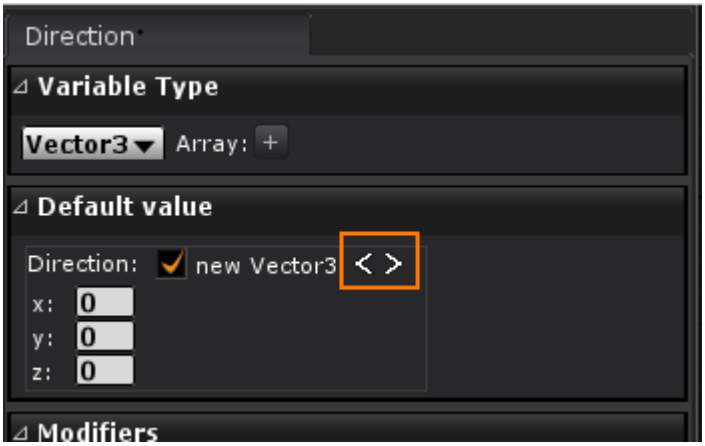
Variable type can be changed by pressing button in Variable type section of details window:

All the most using types placed at the top of type list.

Each variable can have a default value, which will be set initially. For default types the default value will be set automatically (read Default Values Table for more details)**.** For referencing or struct types there can be a constructor for initializing variable with parameters, which can be turned on by using checkbox:



If type has few constructors you can switch between them by pressing right/left arrow buttons.



2.3. Variable modifiers

Access modifiers control the access of variable for other classes and visibility in inspector window.

public - access is not restricted.

protected - access is limited to the containing class or types derived from the containing class.

internal - access is limited to the current assembly.

protected internal - access is limited to the current assembly or types derived from the containing class.

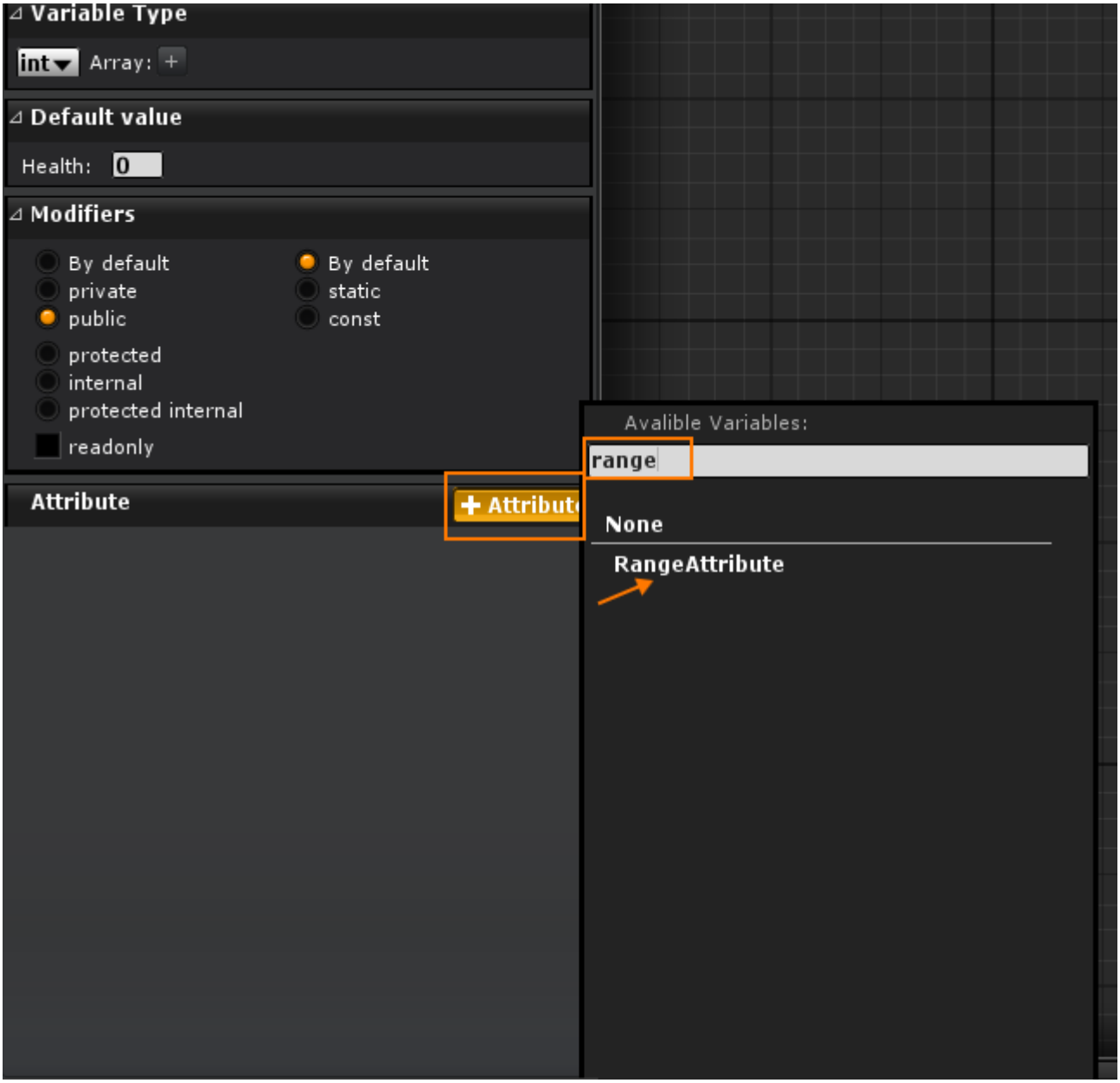
private - access is limited to the containing type.

static modifier is used to declare a static member, which belongs to the type itself rather than to a specific object.

const is used to declare a constant field or a constant local. Constant fields can’t be modified.

2.4. Variable Attributes

Attributes can add some functionality to a variable or change the appearance in inspector window. To add attribute to a variable press '**plus**' button in attribute section and select it from list. Also you can use search field to find the attribute faster.



The most used attributes:

RangeAttribute – change numeric field to a horizontal slider in inspector for a numeric variable;

HideInInspector – don't display variable in inspector;

SpaceAttribute – add spacing under variable in inspector;

TooltipAttribute – show a tooltipe when mouse on variable in inspector;

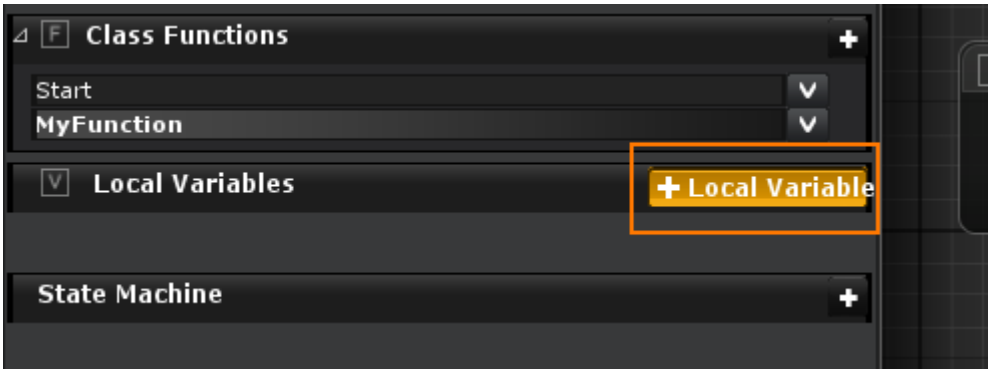
SerializeField – allow to change values of private variables in inspector.

3. Local Variables

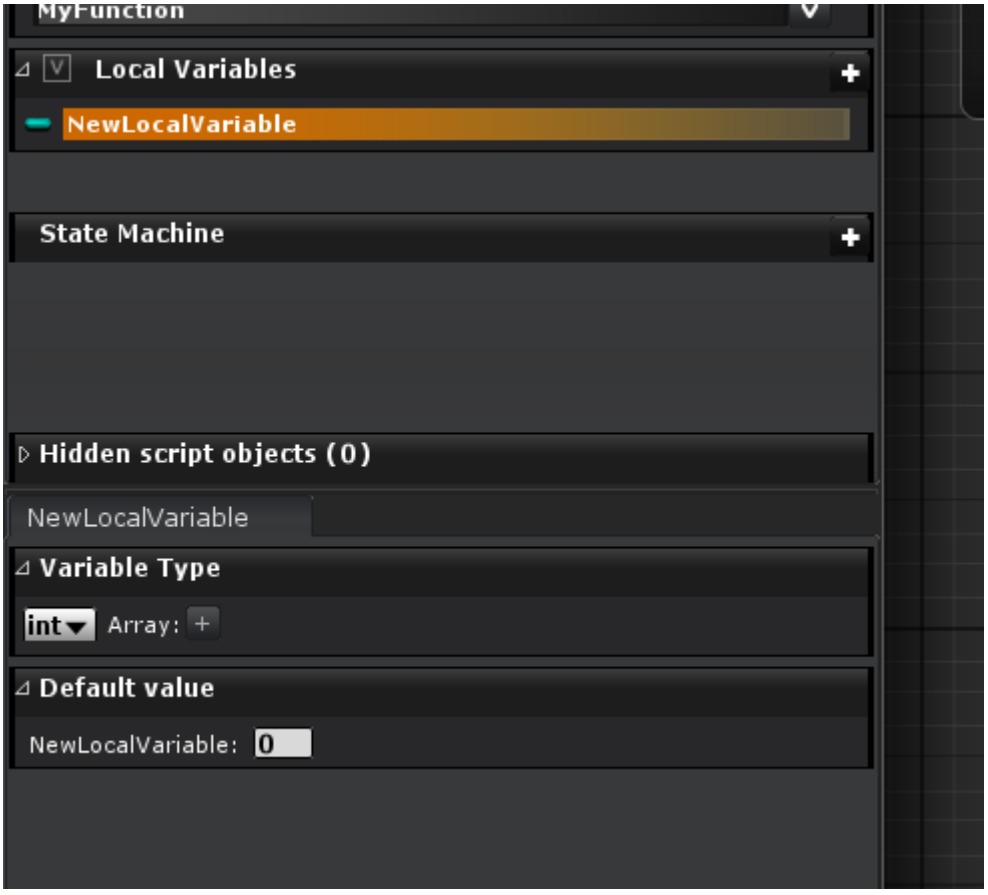
Local variable, as a Class variable, is a 'boxes' for temporary storage information of defined type while code is executing, but they used only in edit graph (in script functions, properties, constructors or state machine events).

3.1. Creating local variables

To create a local variable press '**plus**' button on the right side of **Class variables** section:



After local variable is created you can change only the name (double clicking on its name), the type of variable and its default value. For more information about this check [Class variables](#).

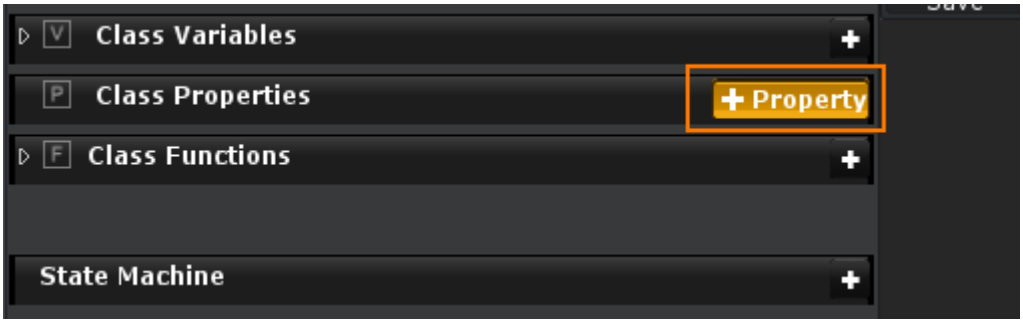


4. Properties

A **property** is a member that provides a flexible mechanism to read, write, or compute the value of a variable.

4.1. Creating properties

To create a property press ‘plus’ button on the right side of Class Properties section:



Note: If you can't find this section, check its visibility in the panel 'Hidden script objects' in the bottom of toolbar.

4.2. Using properties

The property has ‘**set**’ and ‘**get**’ assessors (edit graphs).

‘**Set**’ assessor - executing when we set the value of property (has input pin of new value):

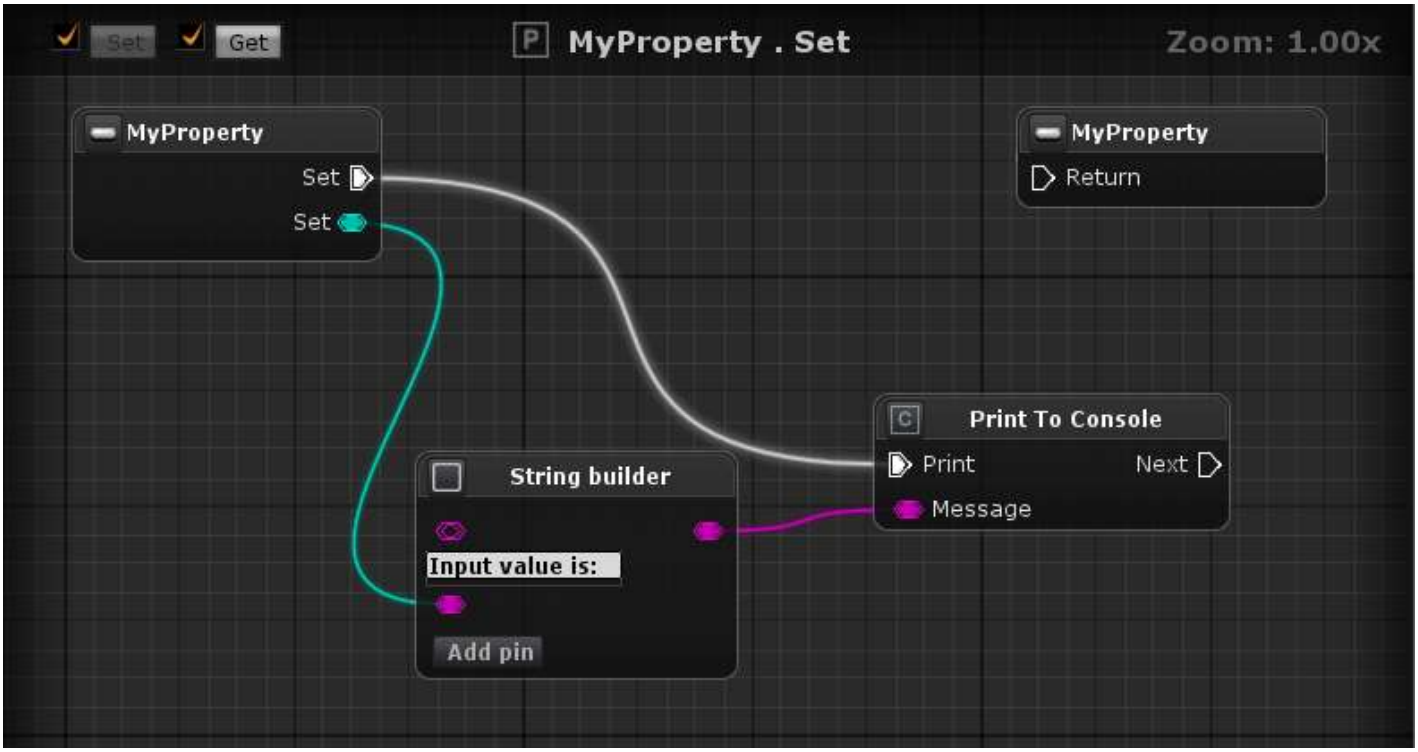
‘**Get**’ assessor - executing when we get the value (has output pin, must return a value):

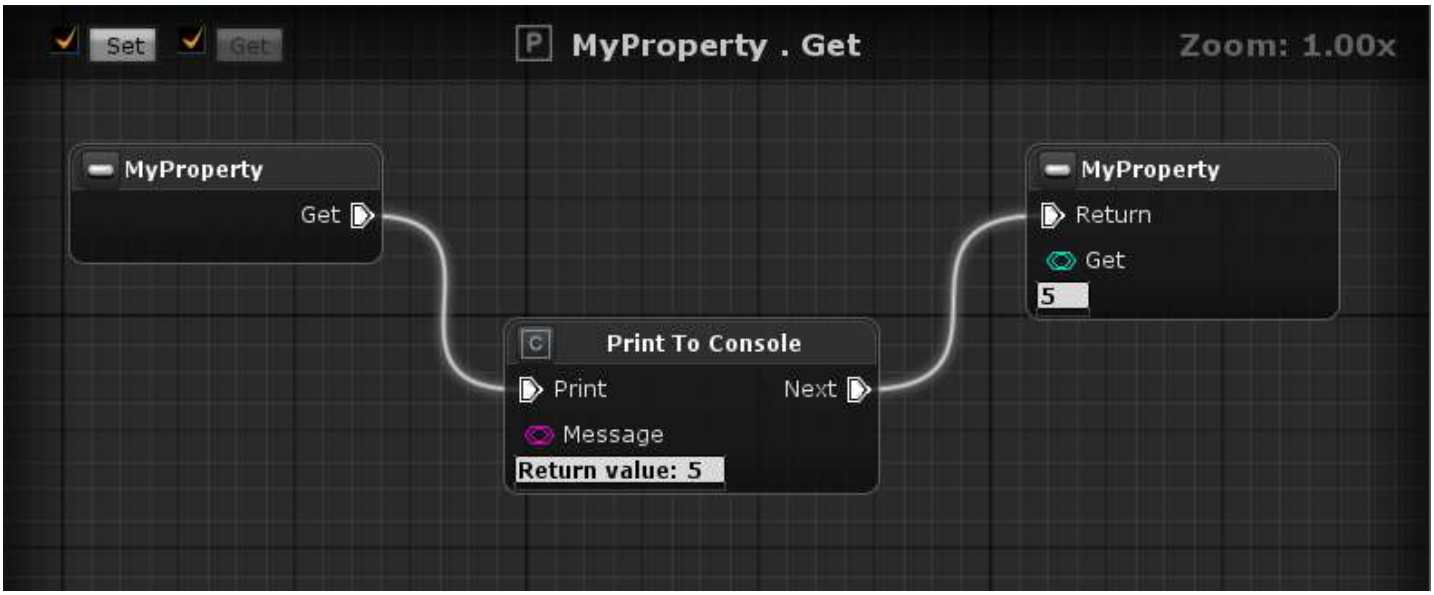
Note: ‘Return’ node allways must be linked with sequence connector in get accessor. The value must be returned at the end of code sequence.

You can switch between ‘**set**’ or ‘**get**’ assessors by pressing ‘**Get**’ and ‘**Set**’ buttons at the top left of edit graph. To disable ‘**set**’ or ‘**get**’ assessors use checkboxes near this buttons.

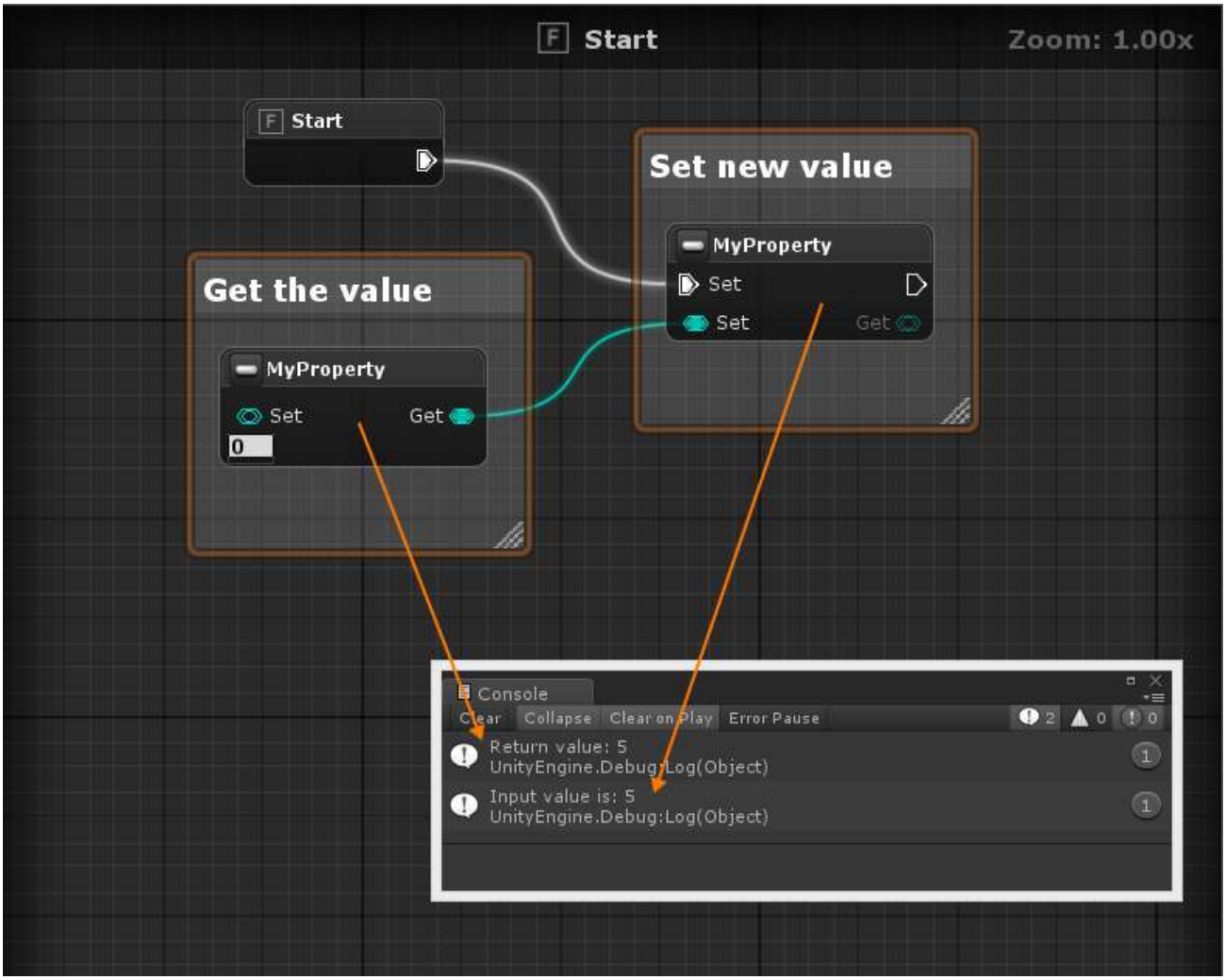
4.3. Example

In this example we will debug values when new value will be set or get in property:





Now test it. We get the value, set it again to this property and get the console messages:



4.4. Property modifiers

Access modifiers control the access of property for other classes and visibility in inspector window.

public - access is not restricted.

protected - access is limited to the containing class or types derived from the containing class.

internal - access is limited to the current assembly.

protected internal - access is limited to the current assembly or types derived from the containing class.

private - access is limited to the containing type.

const is used to declare a constant field or a constant local. Constant fields can't be modified.

static modifier is used to declare a static member, which belongs to the type itself rather than to a specific object.

Note: you can't use non-static variables in static property.

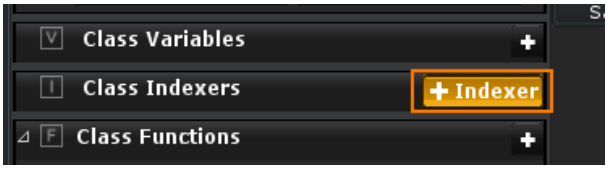
The **virtual** modifier is used to modify a property and allow for it to be **overridden** in a **derived** class. It means that you can create a class 'A' with a virtual property and you will be able to override the code of this property in class 'B' which derived from class 'A'.

5. Indexers

Indexers allow instances of a class or struct to be indexed just like arrays. Indexers resemble properties except that their accessors take parameters.

5.1. Creating indexer

The indexer can be created by pressing 'plus' button on **Class Indexers** section:



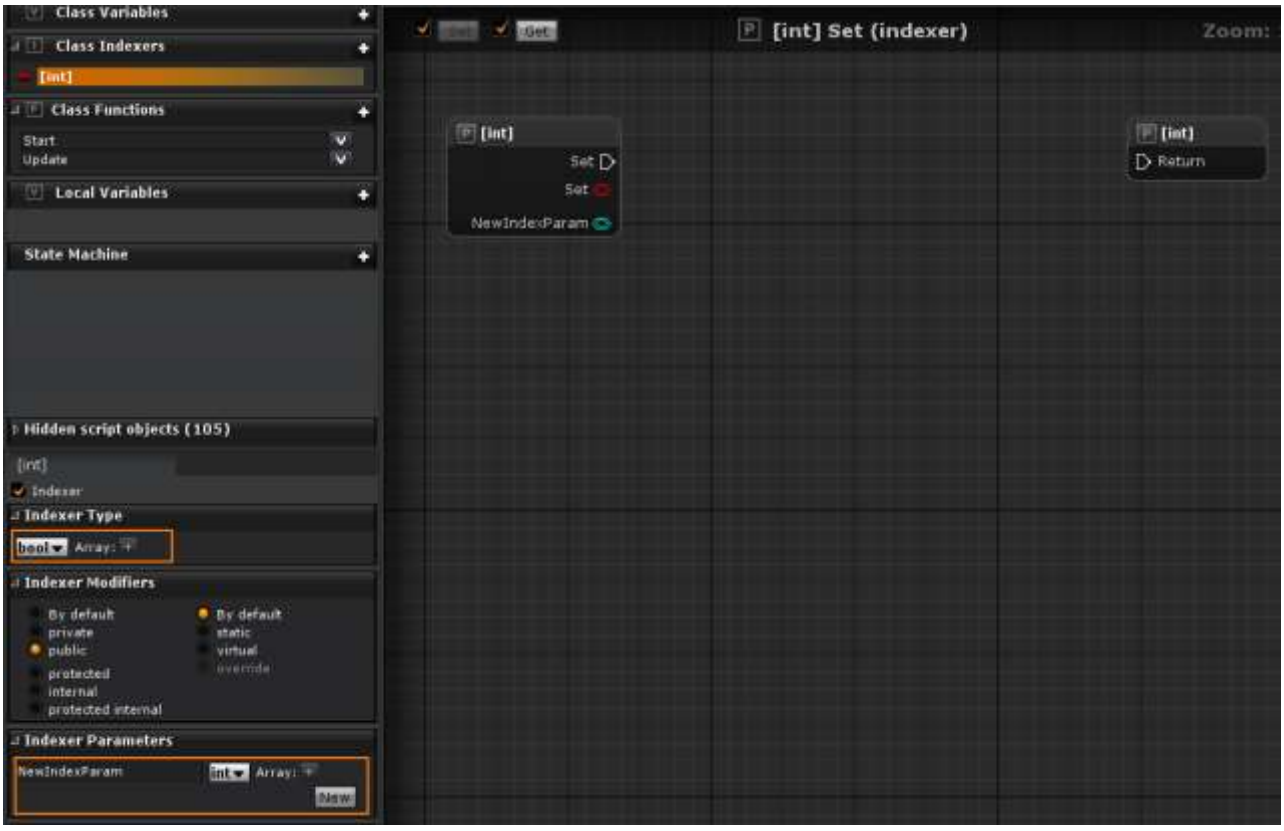
Note: If you can't find this section, check its visibility in the panel 'Hidden script objects' in the bottom of toolbar.

The indexer it's like accessing to some element of array: it has the index of element (**int** type) and the value of element. You can **get** or **set** it.

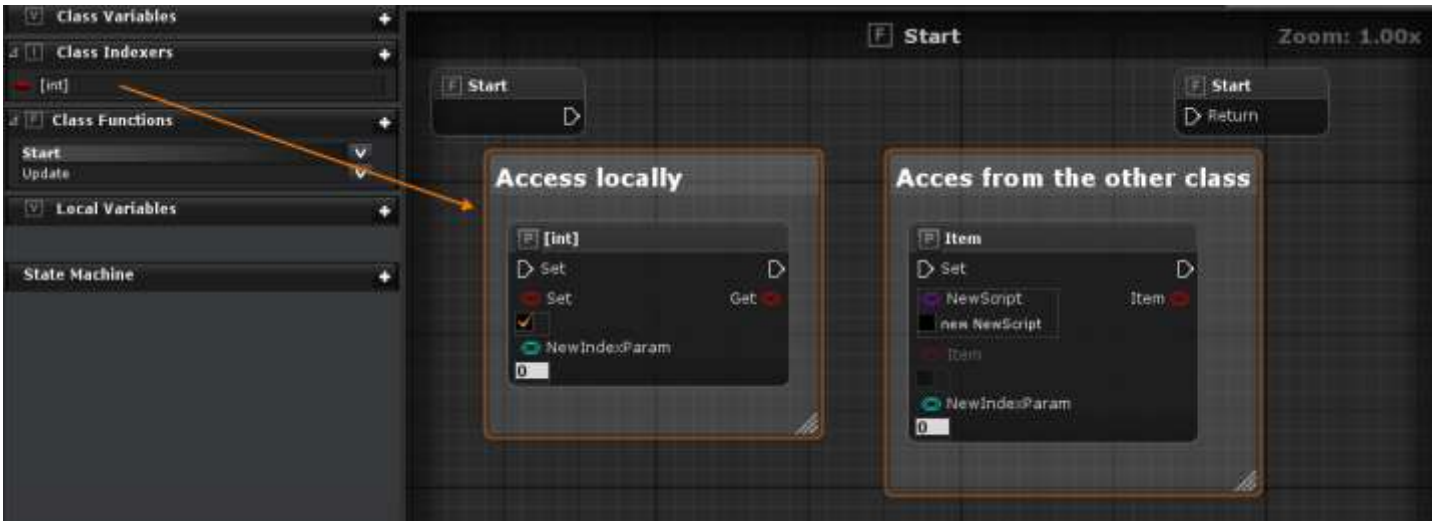
The main properties of indexers:

- Contains "**get**" and '**set**' accessors (like a property);
- Can have **multiple parameters** in accessor (at least one);
- Don't have a **name**;
- The class **can't have few indexers** with the **same parameter types**.

You can set indexer parameters and indexer type in Details window of indexer:



After compiling you will be able to use it in code:

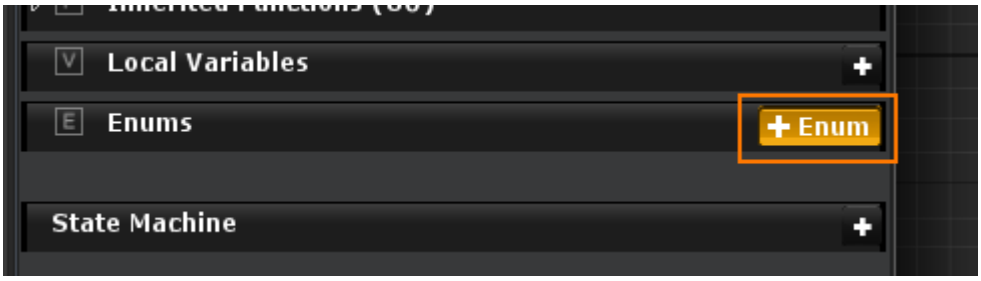


6. Enums

The enum is used to declare an enumeration, which is a set of named constants. It's like an array of "names" and each name can has own value.

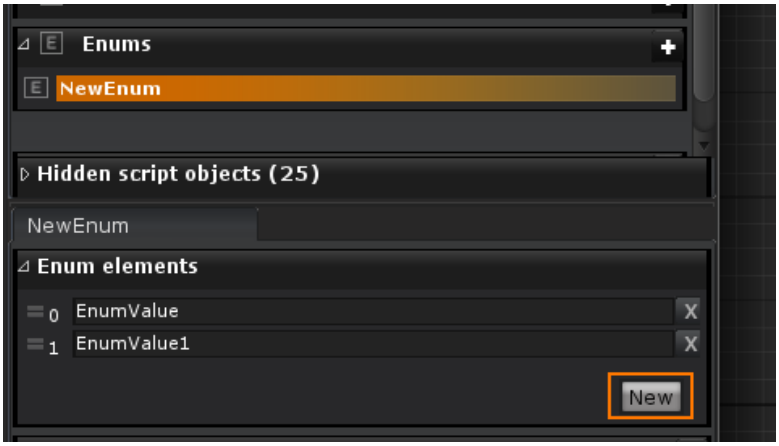
6.1. Creating enum

To create new enum press 'plus' button on the right side of Enums section:

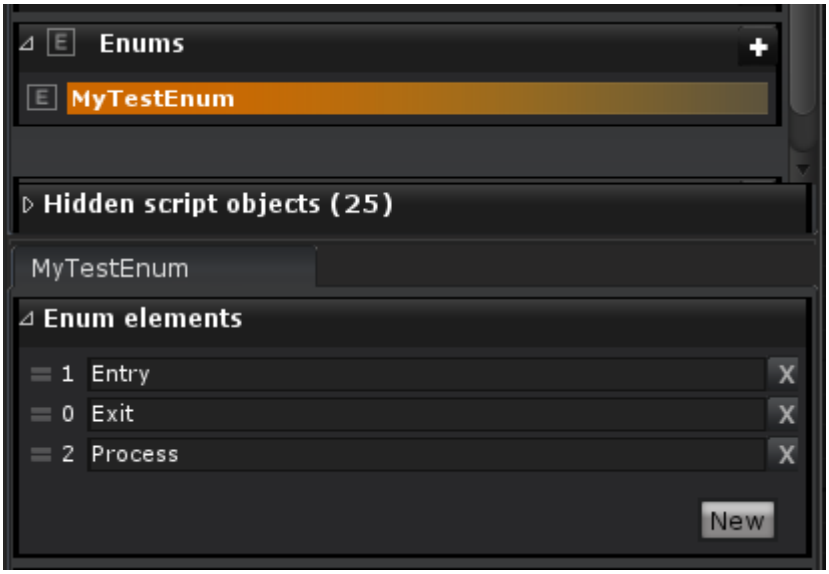


Note: If you can't find this section, check its visibility in the panel 'Hidden script objects' in the bottom of toolbar.

To add new elements to enumeration press '**New**' button in 'Enum elements' section:

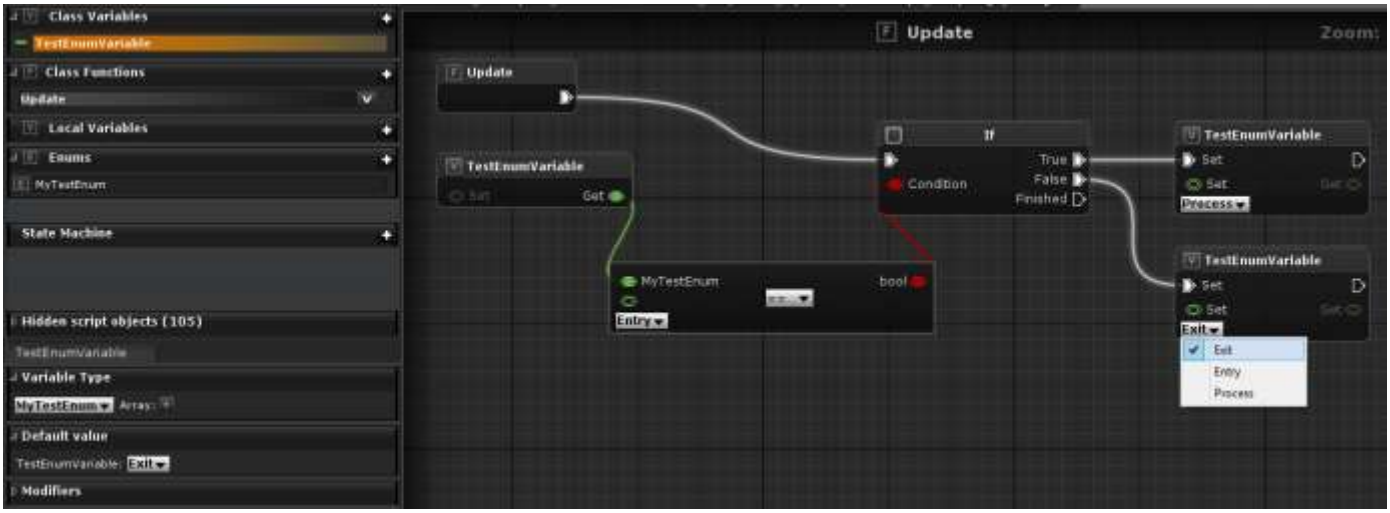


To delete elements from enumeration press [X] button on the right side of element that must to be deleted. You can change the order of elements by dragging sorting button on the left side of each element. Rename elements by double clicking on its name.

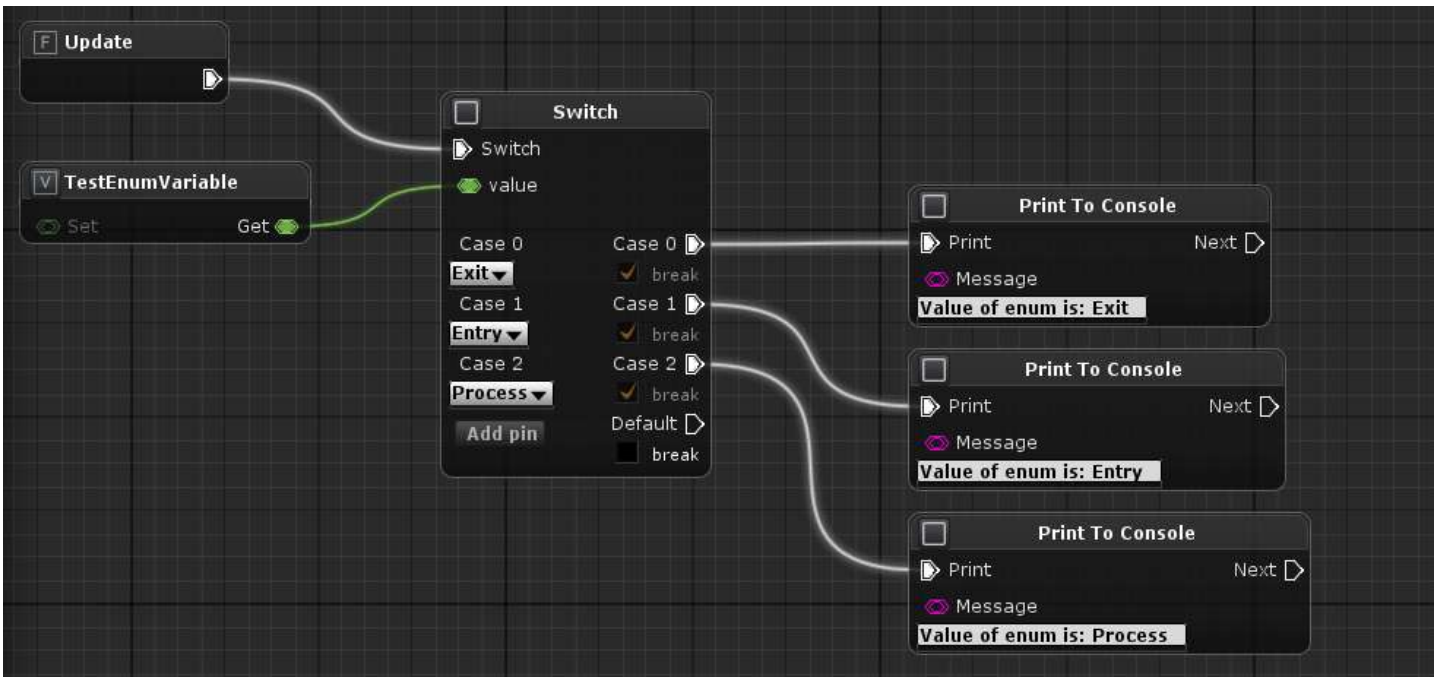


6.2. Using enum

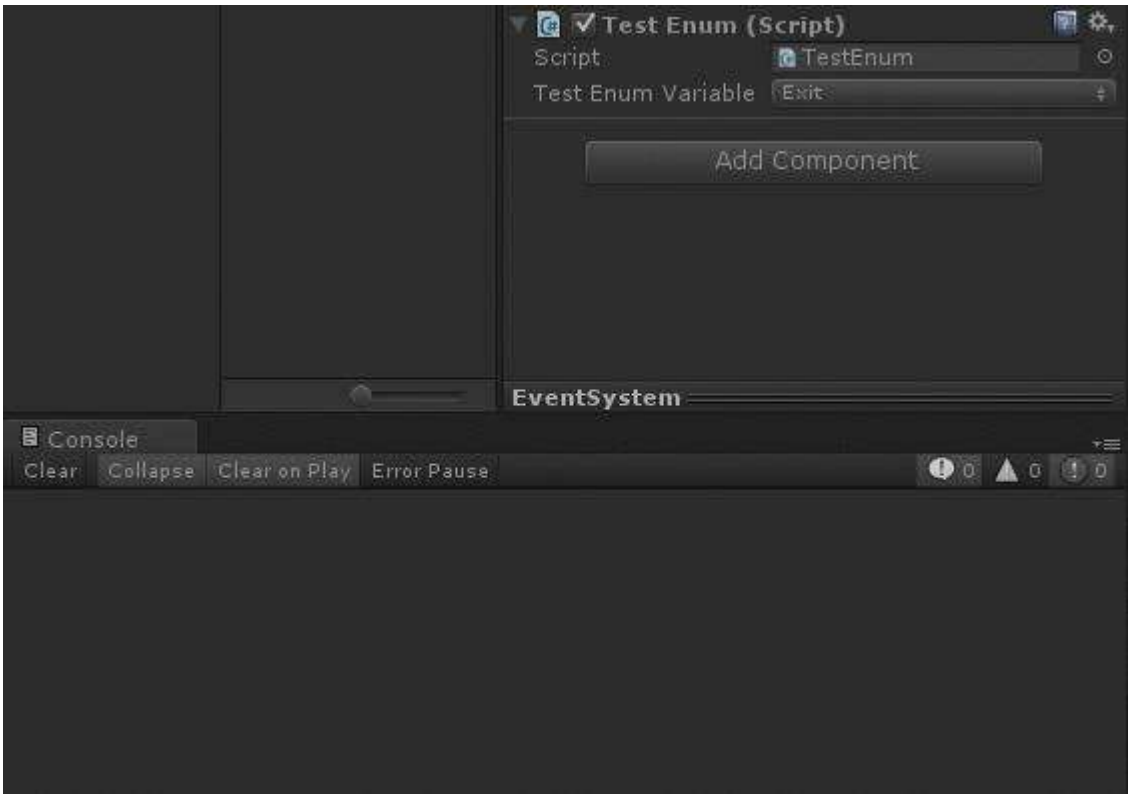
After compiling script you will able to use this enum in code. Create variable with type of your Enum and use it in code:



Enums works fine with '**Switch**' node, which identifies what statement to run based on value of enum:



In inspector enums are displayed as popup button:



7. Delegates

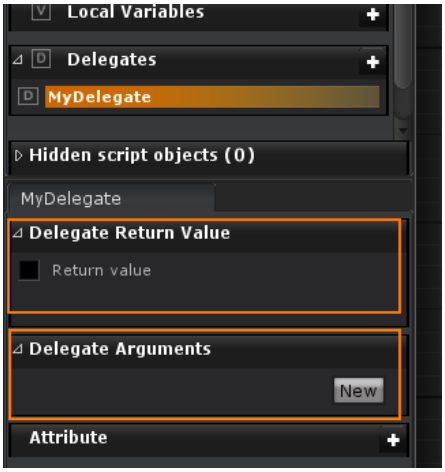
A **delegate** is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

7.1. Creating delegates

To create new delegate press ‘plus’ button on the right side of **Delegates** section:

Note: If you can't find this section, check its visibility in the panel 'Hidden script objects' in the bottom of toolbar.

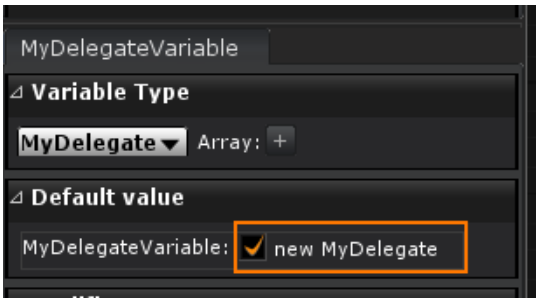
If you associate this delegate with function that has arguments or return type you can set these arguments or return type in Details window of this delegate:



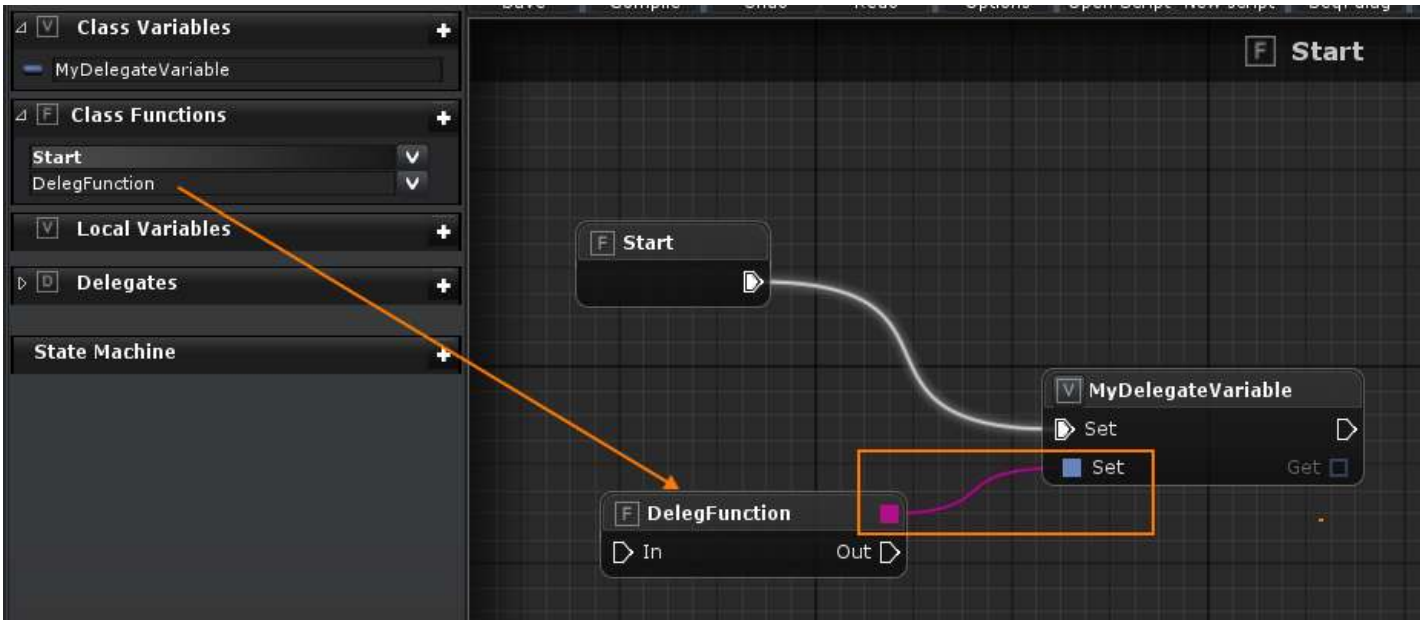
7.2. Using delegates

After compiling script you will able to use this delegate as variable type:

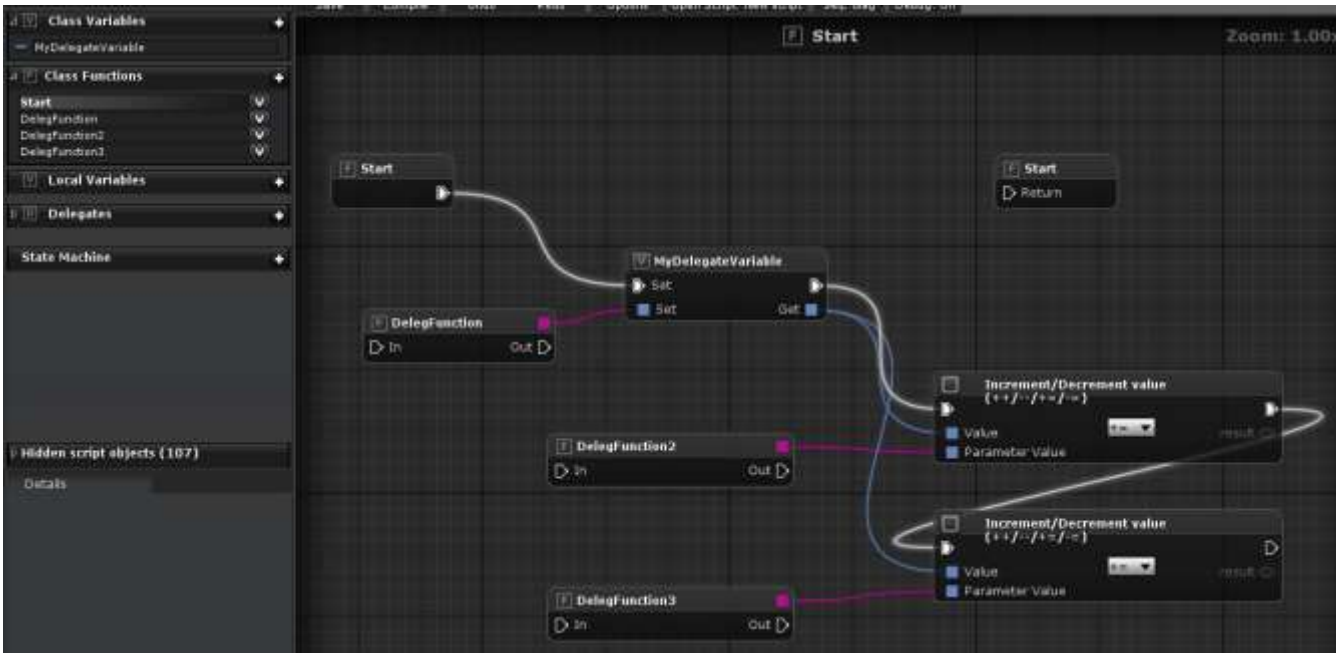
Don’t forget to initialize the start value, otherwise executing of null delegate variable will cause en error.



To assign function to a delegate just link it with special pin at the corner of the function node:



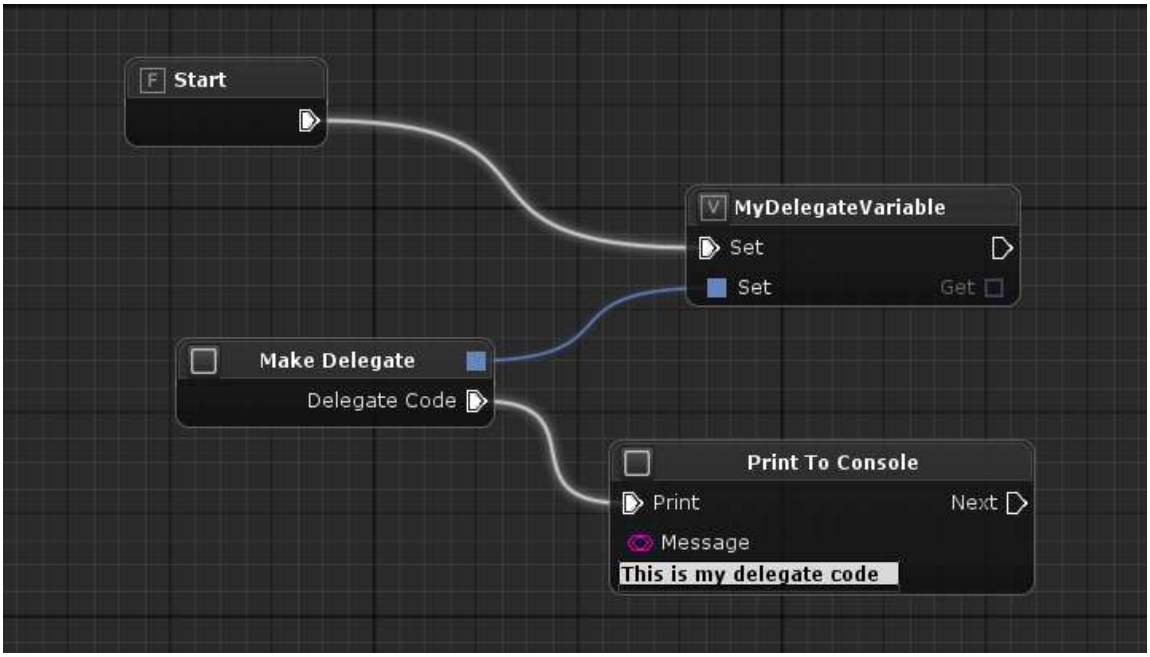
Also you can add other links to this delegate (multidelegate) using ‘**Increment/Decrement node**’ with operation “**+=**”, and when delegate will be executed all the functions which assigned to this delegate will be executed too:



Note: executing functions with delegates faster than direct function execution.

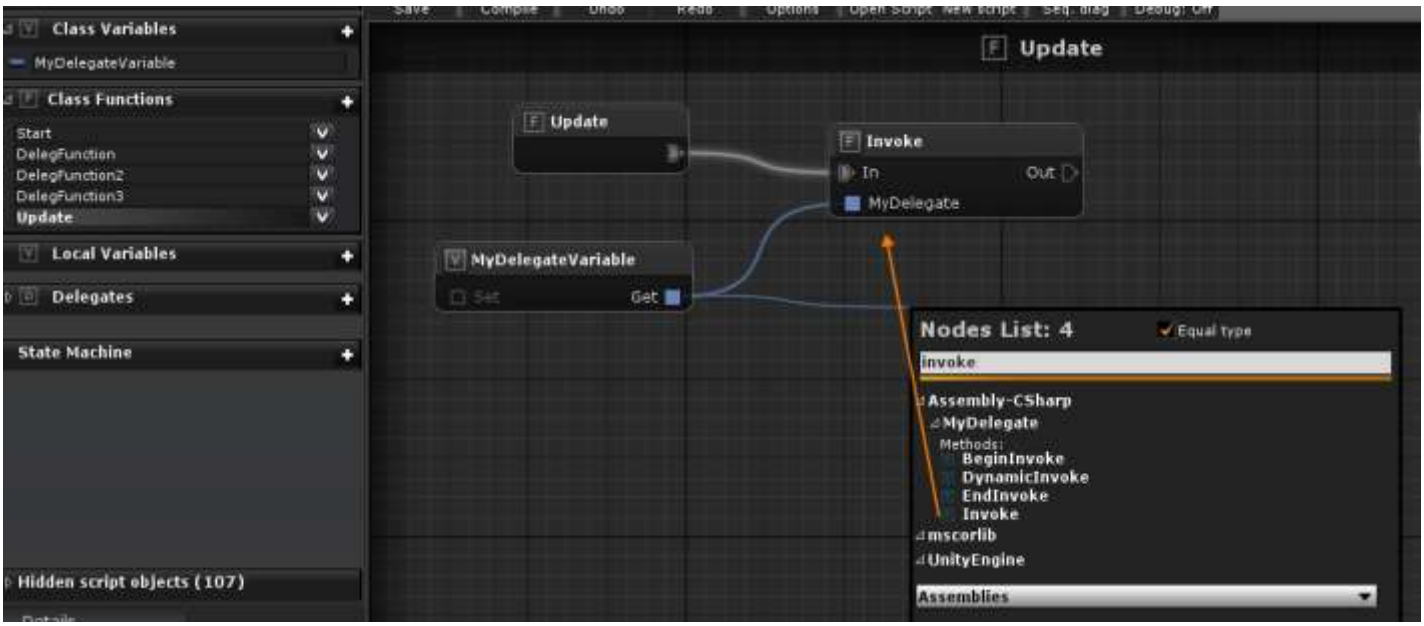
You can remove your functions from delegate by using the same '**Increment/Decrement node**' node but changing operation to "**-=**".

Also you can make a code and assign it to delegate with the '**Make delegate**' node:



7.3. Executing delegates

To execute method find '**Invoke**' method from delegate variable. Invoking delegate cause executing all code and functions in delegate.



8. Classes

A **class** is a construct that enables you to create your own custom types by grouping together variables of other types, properties, functions etc. A class is like a blueprint. It defines the data and behavior of a type.

8.1. Creating classes

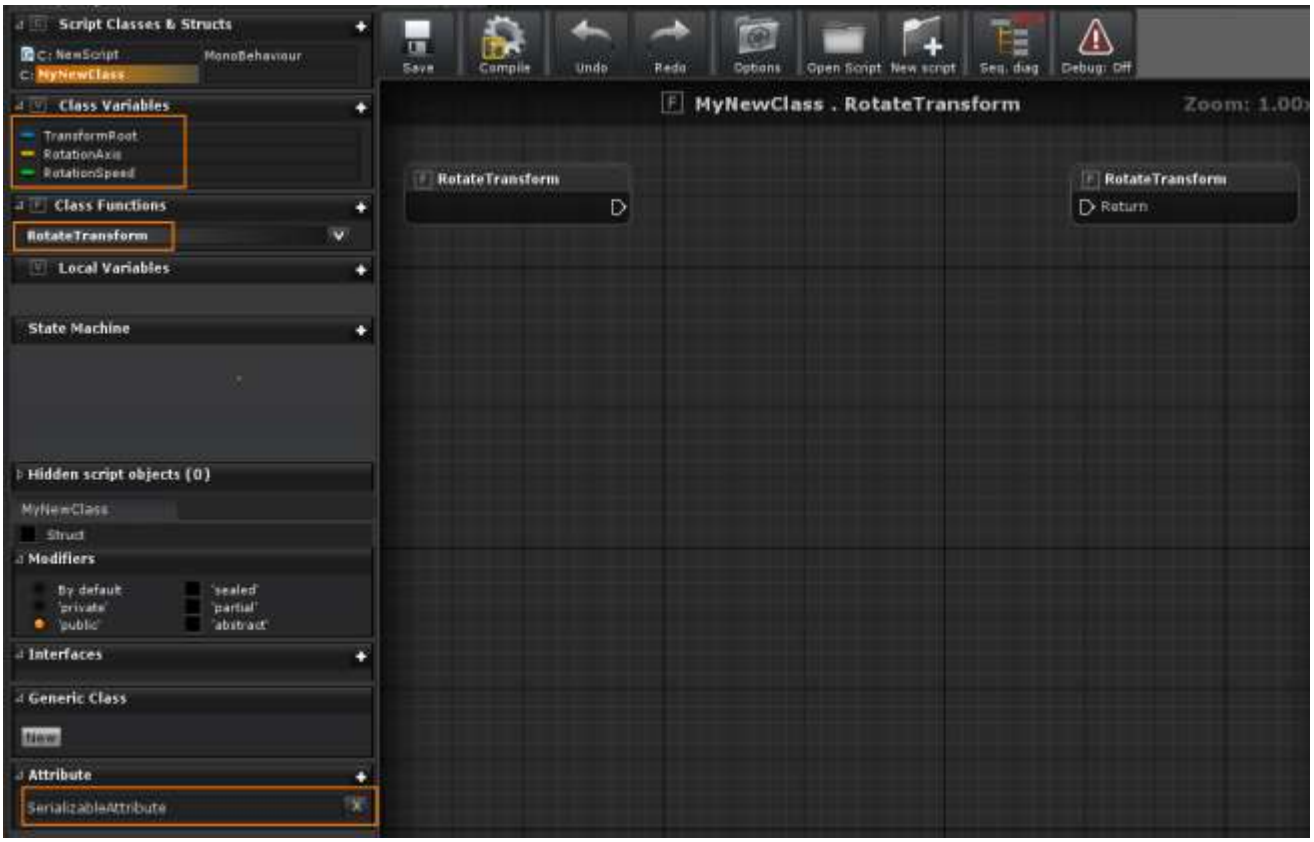
New classes can be created by pressing ‘plus’ button on **Script Classes & Structs** section:

The class can contain different types of code objects (functions, variables, properties, etc) that can be created in the left toolbar of editor. All unused members can be hidden in bottom menu of this toolbar. In this article described each of them.

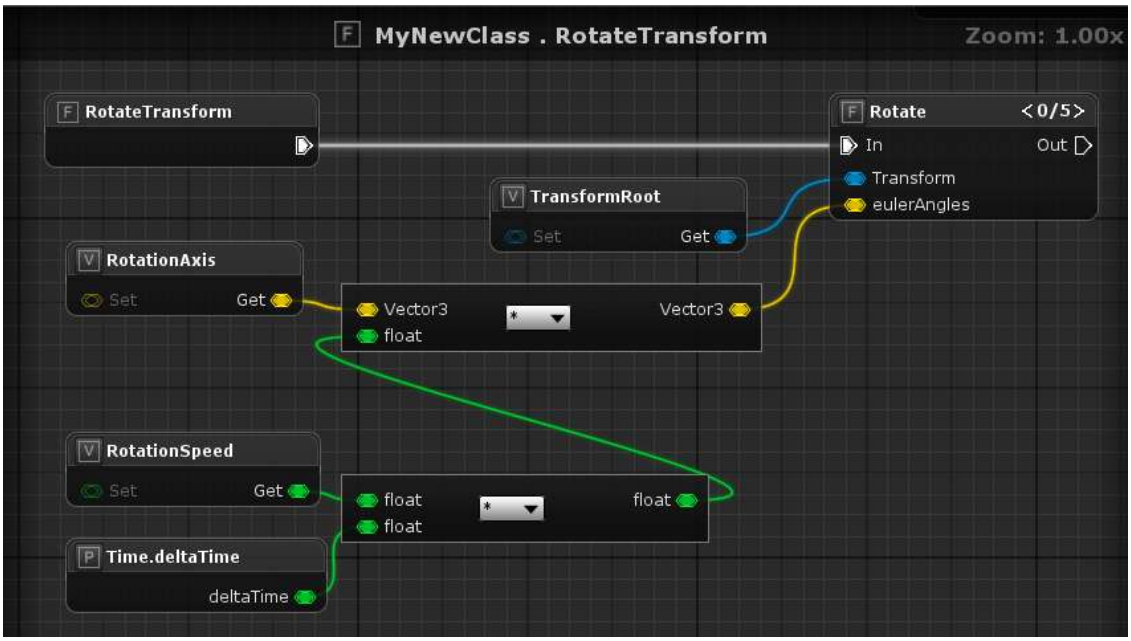
8.2. Using classes

Classes can be used for holding some groups of information of same type and internal calculation of them. For example we will make a new class that will hold transform and rotate it on defined axis and speed.

Create a class with function “RotateTransform” and variables: TransformRoot (Transform), RotationAxis (Vector3), RotationSpeed(float). Then set “Serializable” attribute to the class to display it in inspector when we will use this class as a variable:



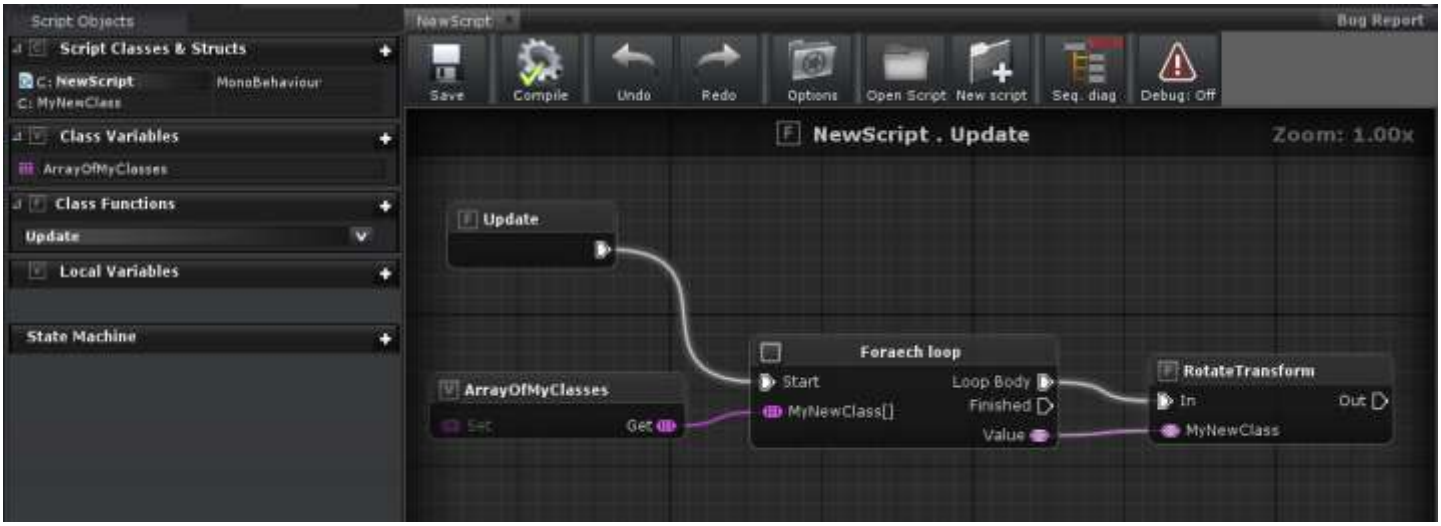
Select all variables, drag them to edit graph and find method “**Rotate**” from **TransformRoot** variable.
Multiply **RotationAxis** vector by **RotationSpeed** multiplied by **Time.deltaTime** and set them to a second argument of “Rotate” method:



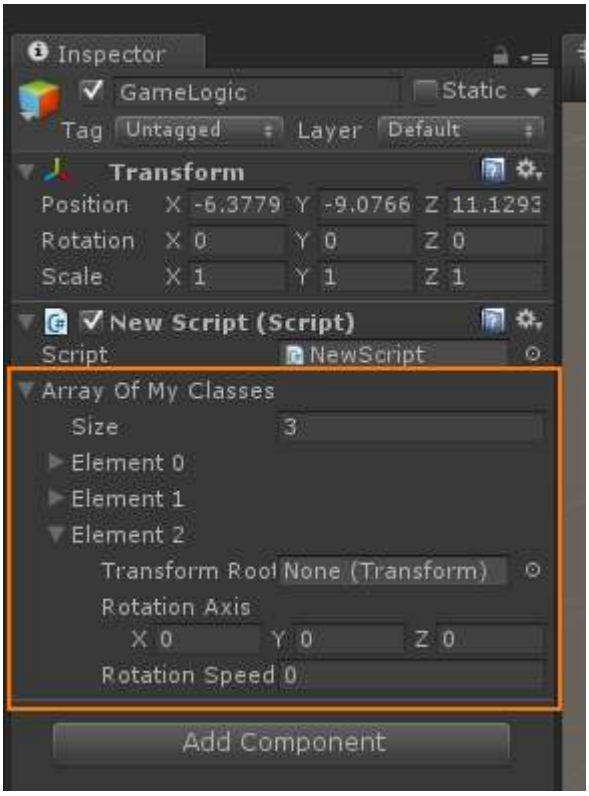
After compiling script we can use our new class as a variable (as array) in other class:

Note: to create one dimensional array of some type just press “+” button near selected type in Variable Type section.

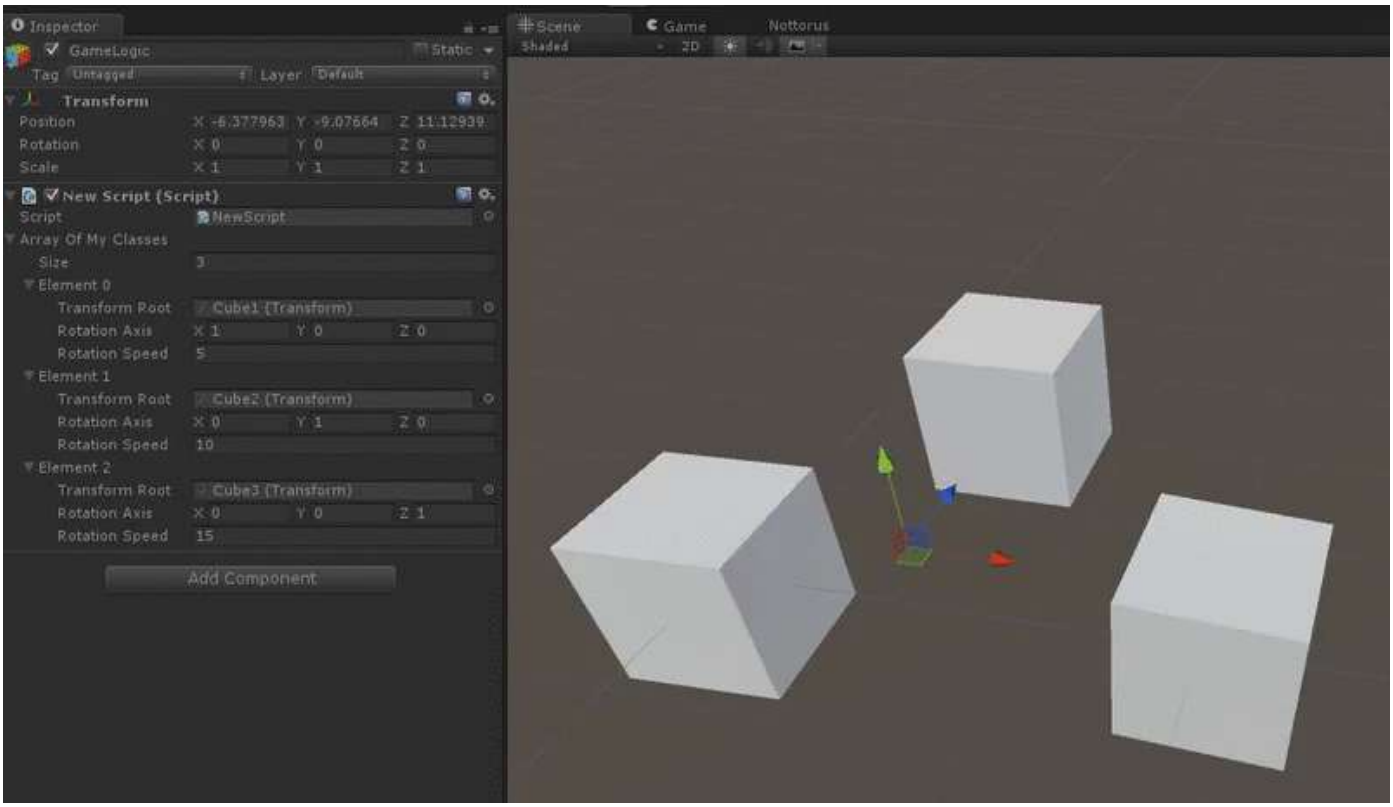
The last step is to iterate through all instances of class in our array and execute function “**RotateTransform**” in each of them. Connect array variable to the '**Foreach loop**' node then find method “**RotateTransform**” from out pin of **Foreach** node and connect it to a ‘**Loop body**’ connector:



Now compile the script and assign it into some gameobject:

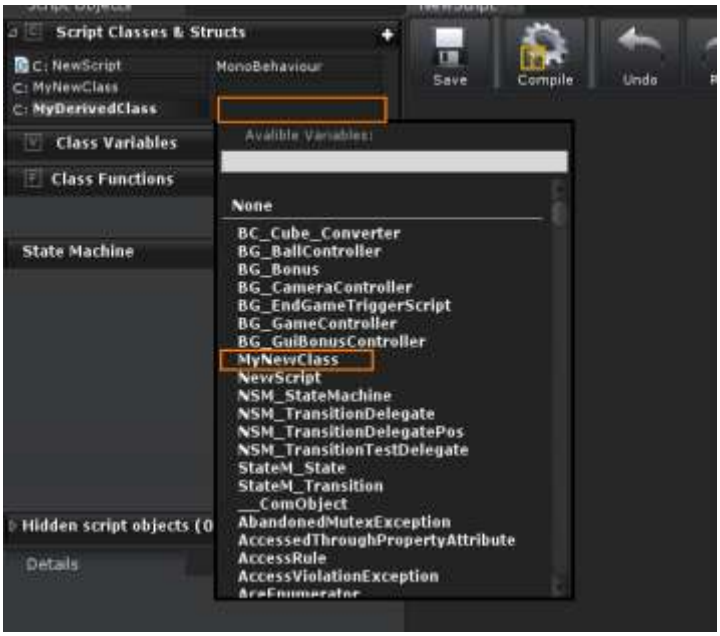


As you can see the array of our classes displayed correctly (we set the Serializable attribute) and we can assign some transforms which we want to rotate and angle with speed to each of them personally:

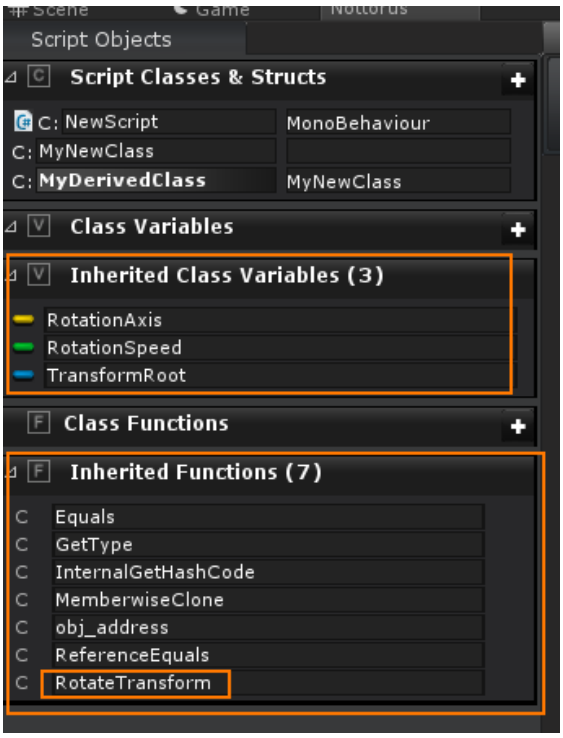


8.3. Inheritance

Each class can be inherited from other class and inherit all functionality (variables, functions and other script objects which has '**public**' modifier) from derived class. To inherit class from other press on the box on the right side of class name field and select the class from the list:

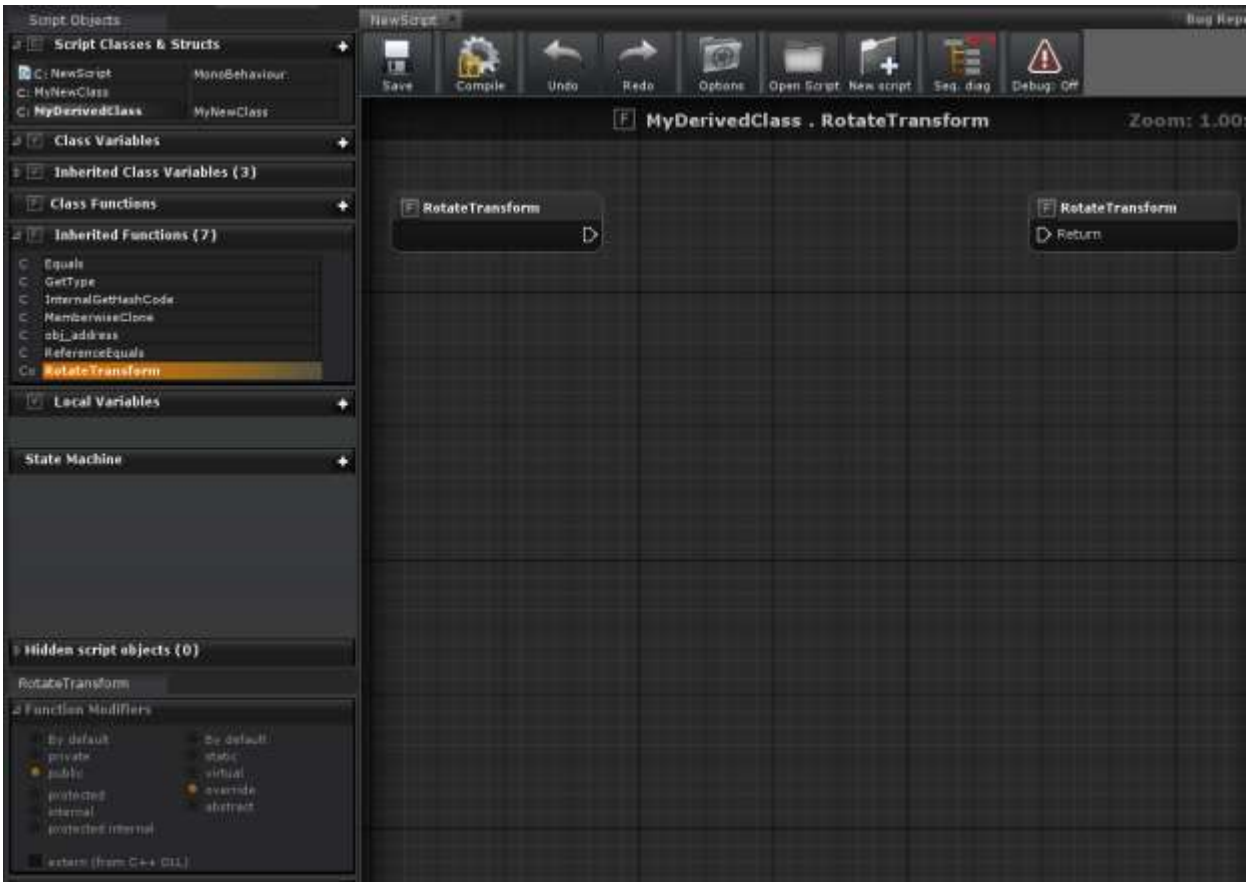


After that you can see that all variables and functions were inherited from derived class:



8.4. Overriding methods

To override some method of derived class in inherited class you must to set ‘virtual’ modifier in the function in derived class. After compiling you will able to override it in inherited class:



Note: if after compiling you can’t override the method try to inherit again.

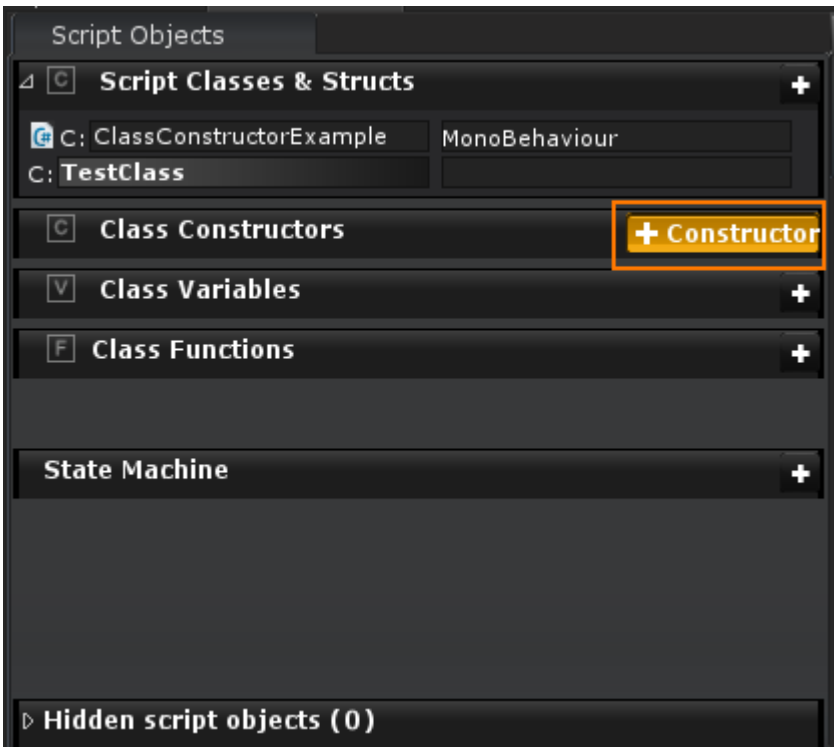
9. Constructors

Whenever a **class** or **struct** is created, its **constructor** is called. A class or struct may have multiple constructors that take different arguments.

In other words the constructor looks like a function that will be called when you create instance of class or struct. Also it affects the number of arguments when you create a new instance of class.

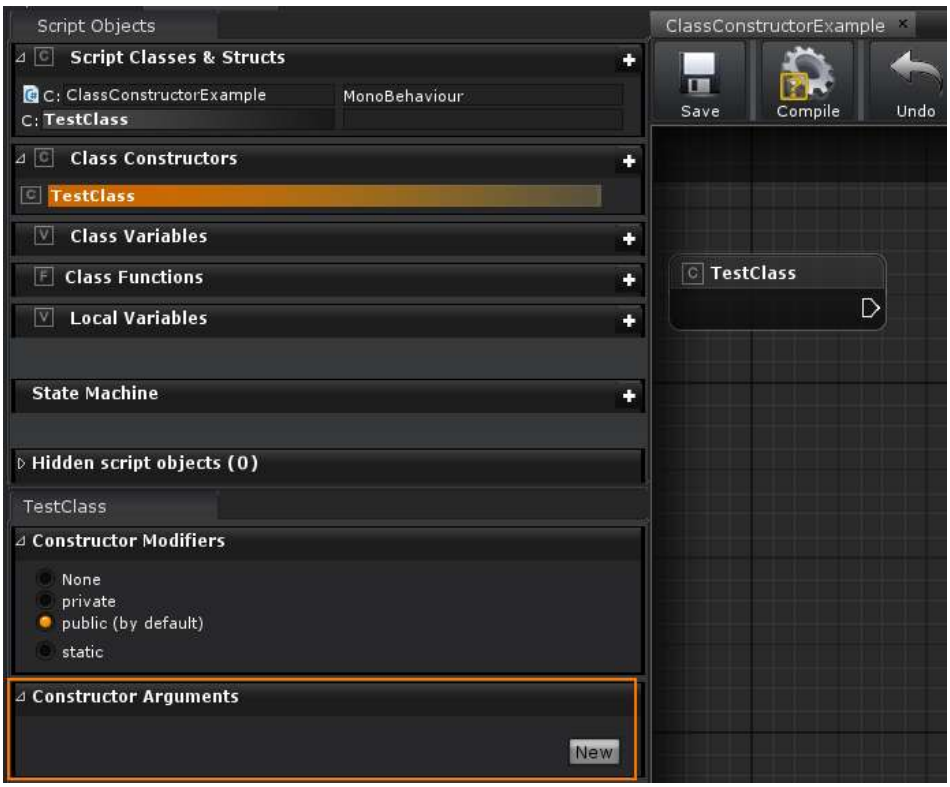
9.1. Creating constructors

The constructor of class can be created by pressing 'plus' button on **Class Constructors** section:



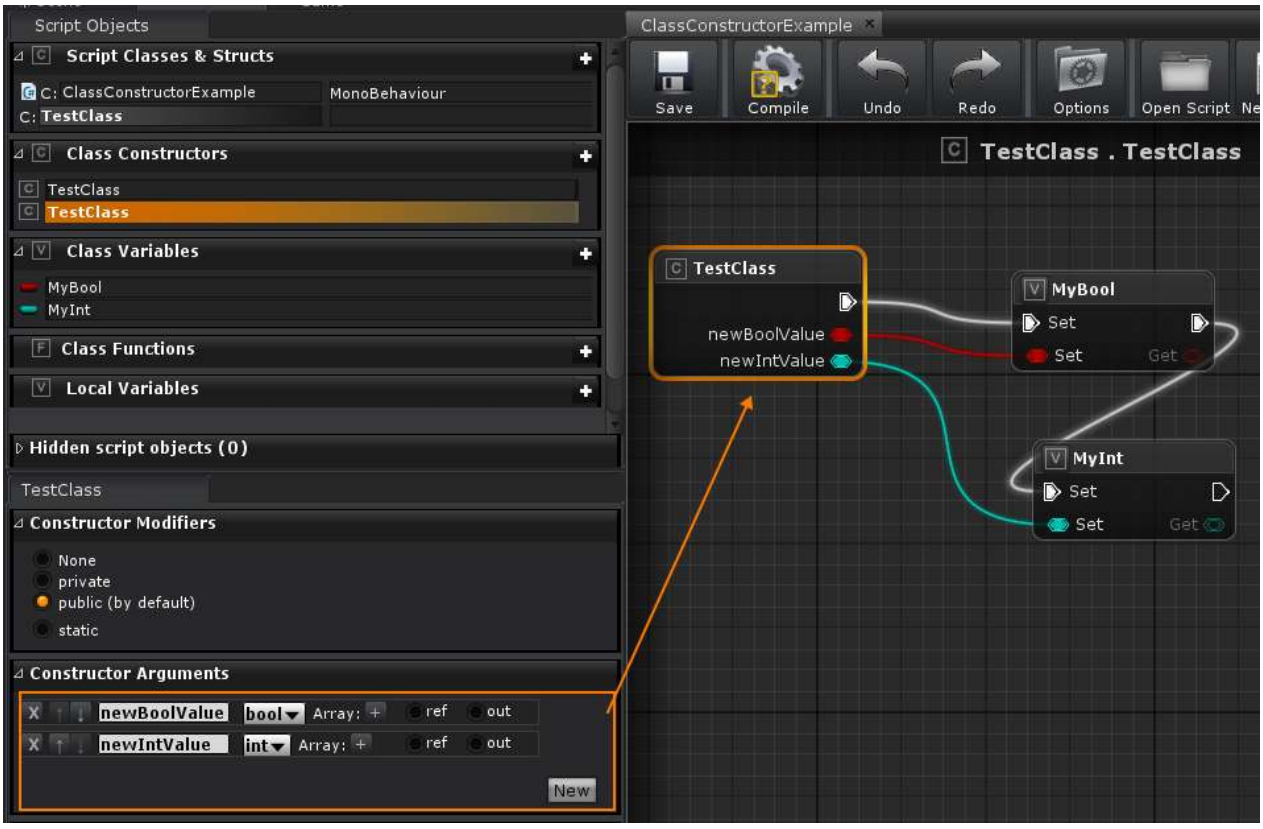
Note: If you can't find this section, check its visibility in the panel 'Hidden script objects' in the bottom of toolbar.

You can specify constructor arguments in the details window of constructor:



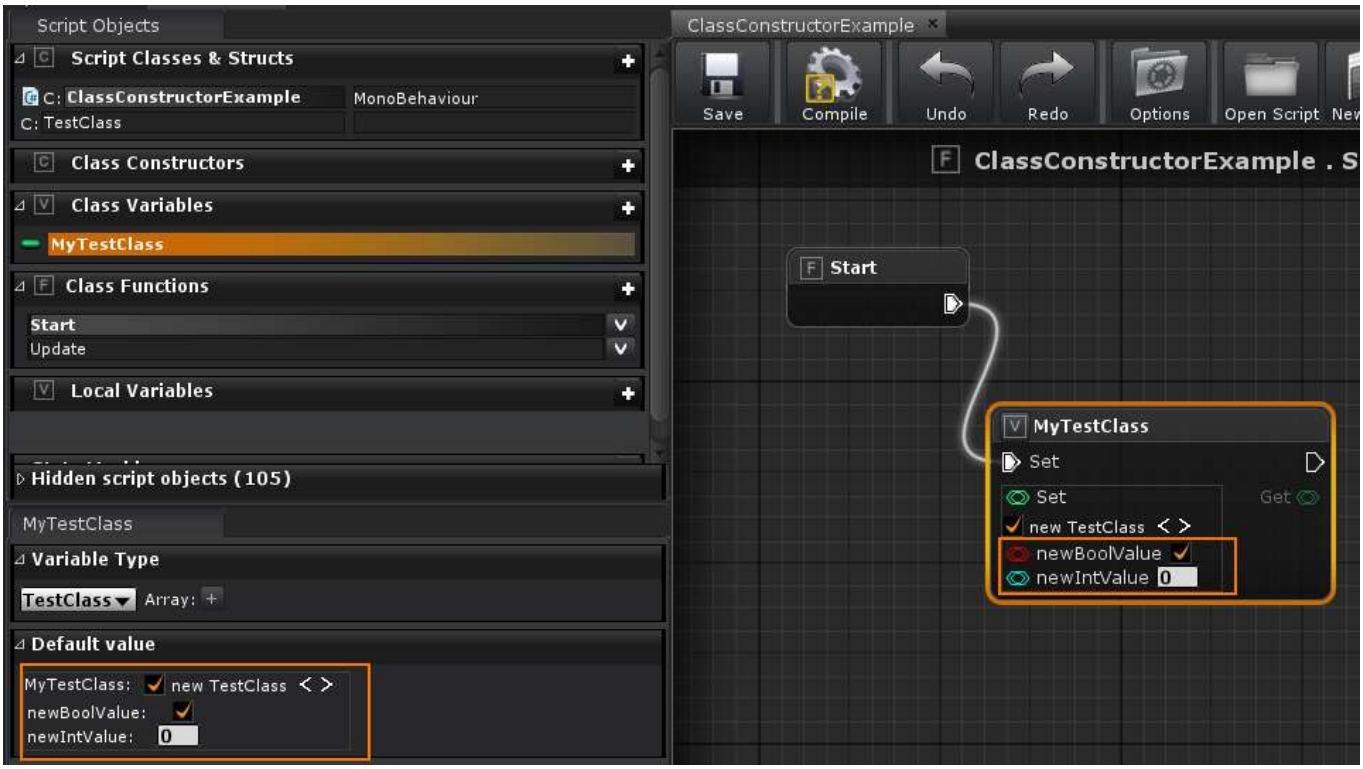
9.2. Using constructors

For example you have custom class and you want to set the values of some variables from arguments when class will be instantiated. So create a new constructor and add required arguments and connect (set) them to the variables:



In this example we leave one (first) constructor without arguments to be able to instantiate a class without overriding variables.

After compilation you will able to use this constructor to create an instance of class with arguments that will set the values of variables from arguments:



Note: to select wanted constructor just select it by pressing “**<**” and “**>**” buttons.

10.Structs

The **struct** is very similar to the **class**, but there are some differences between them:

A **class** is a reference type. When an object of the class is created, the variable to which the object is assigned holds only a reference to that memory.

A **struct** is a value type. When a struct is created, the variable to which the struct is assigned holds the struct's actual data. **When the struct is assigned to a new variable, it is copied.**

Also:

- Variables in structs cannot be initialized (you can't set the default value)
- Structs cannot be inherited from other structs or classes.

10.1. Creating structs

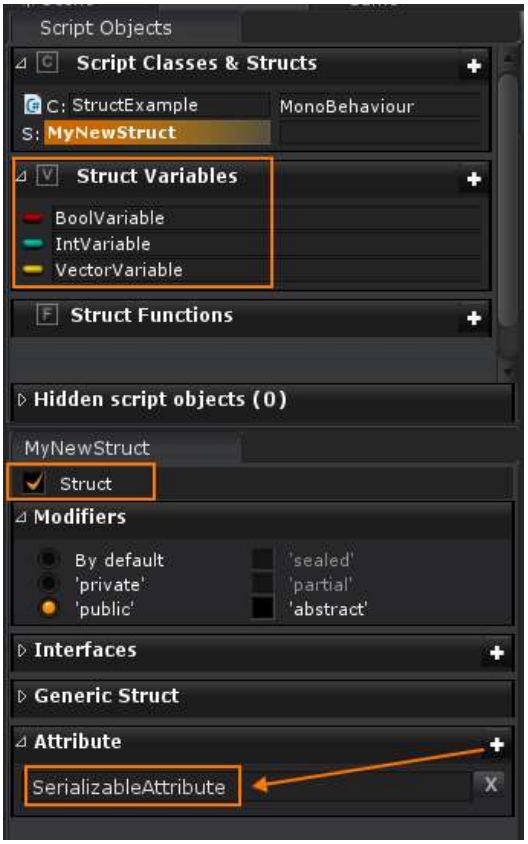
To create new struct press 'plus' button on the right side of **Classes & Structs** section (as you do this to create the class) and check '**Struct**' option:

Note: If you can't find this section, check its visibility in the panel 'Hidden script objects' in the bottom of toolbar.

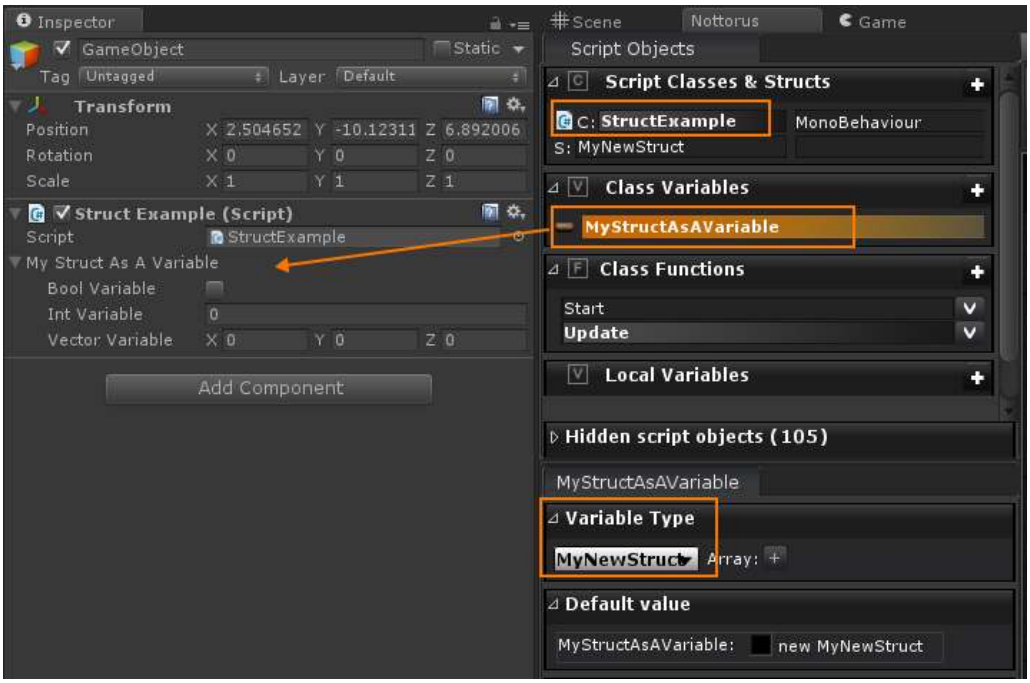
10.2. Using structs

Structs are best suited for **small data structures** that contain primarily data that is **not intended to be modified after the struct is created**.

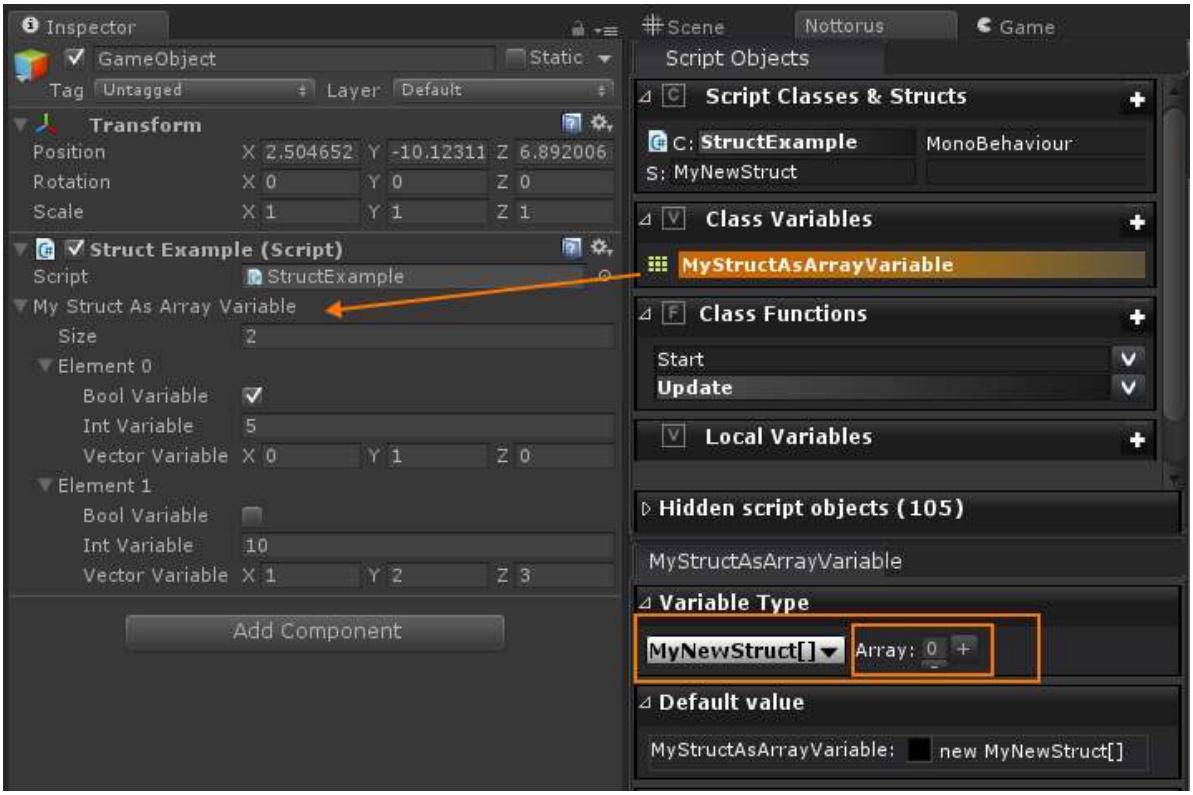
For example you can use it for keeping some configuration in Monobehaviour script and be able to edit its values in inspector. Create a struct set the "**Serializable**" attribute (to show engine that we went this struct will be displayed in inspector):



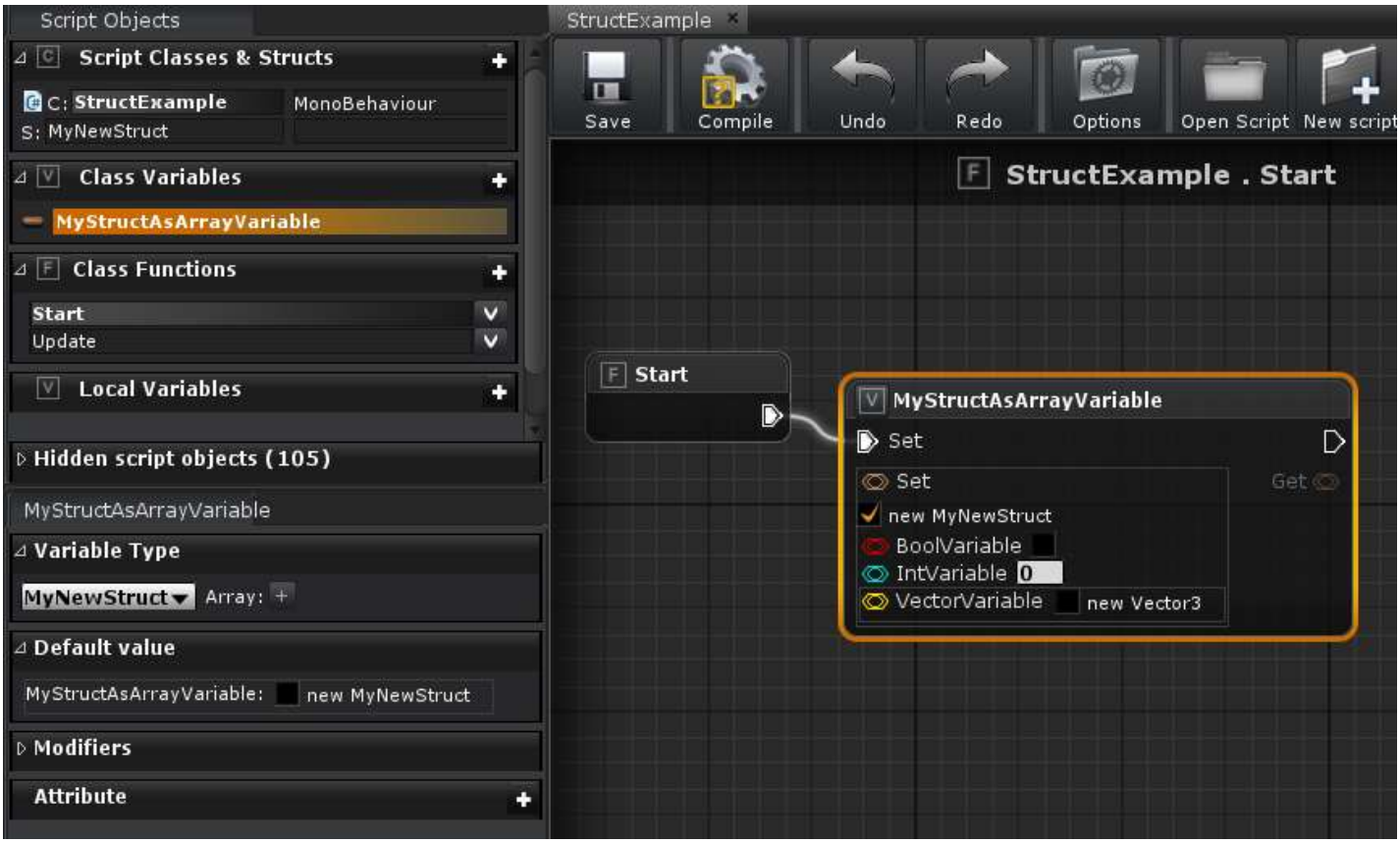
After compiling you will able to use it as variable:



Or as an array:



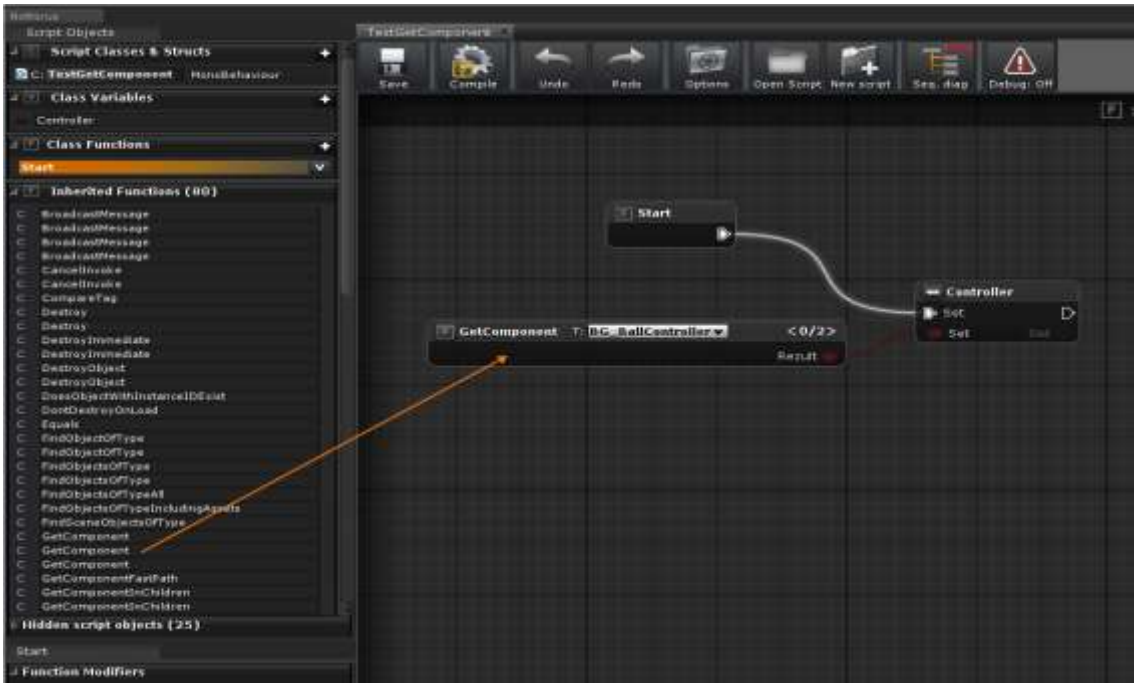
Also you don't need to create constructor for a struct to set its variables when initializing the struct, they will be shown as an arguments:



11.Using Generics

11.1. Using generic functions

One of the most using generic functions is **GetComponent<T>**, where **T** is a generic type parameter which defines a type of component which is required:



Generic parameter type can be selected from dropdown list by pressing generic parameter button on top of node. After that the out pin of node will return component of defined type.

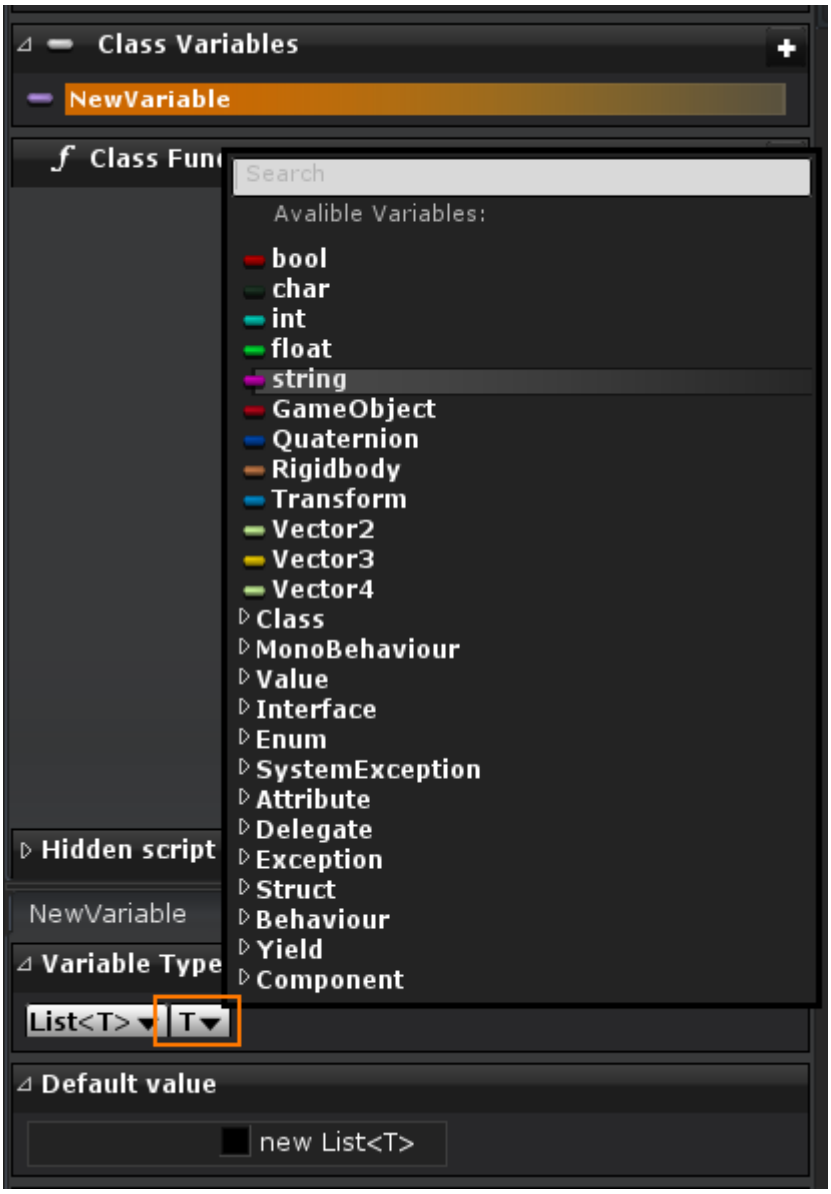
11.2. Using generic variables

Generic types can be holder or collection of defined variable type.

Example (Generic collections):

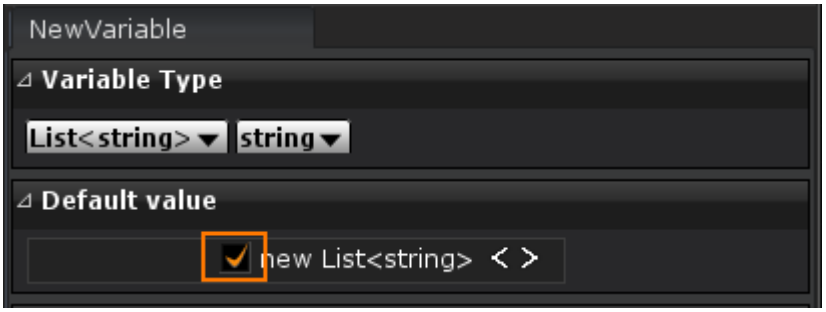
Create **List<T>** variable: (if you can't find it in list make sure you connected all required assemblies (System.Core))

Define the generic <**T**> parameter as a string:



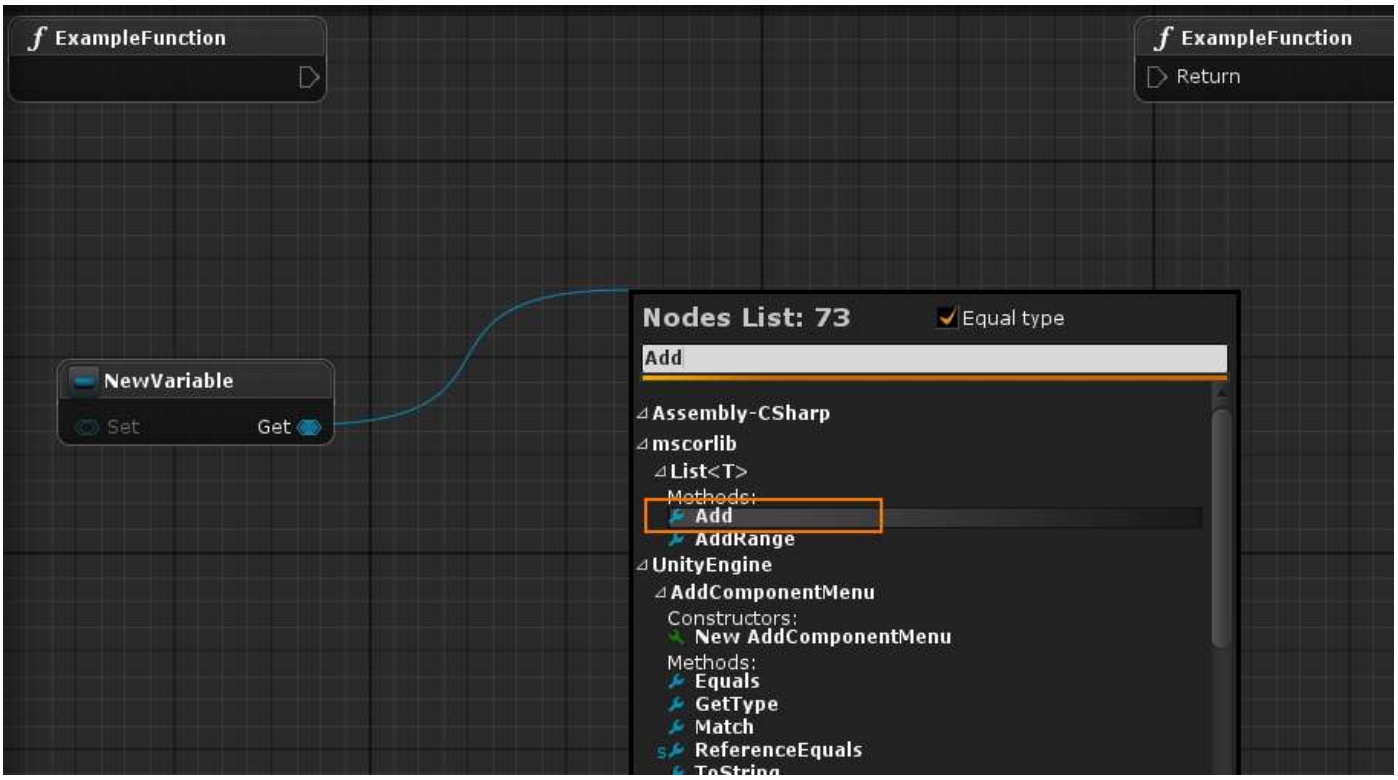
Now **List<string>** is a dynamic collection of string objects.

Check option to initialize variable, otherwise it will be “null” and will cause “NullReferenceException” error:



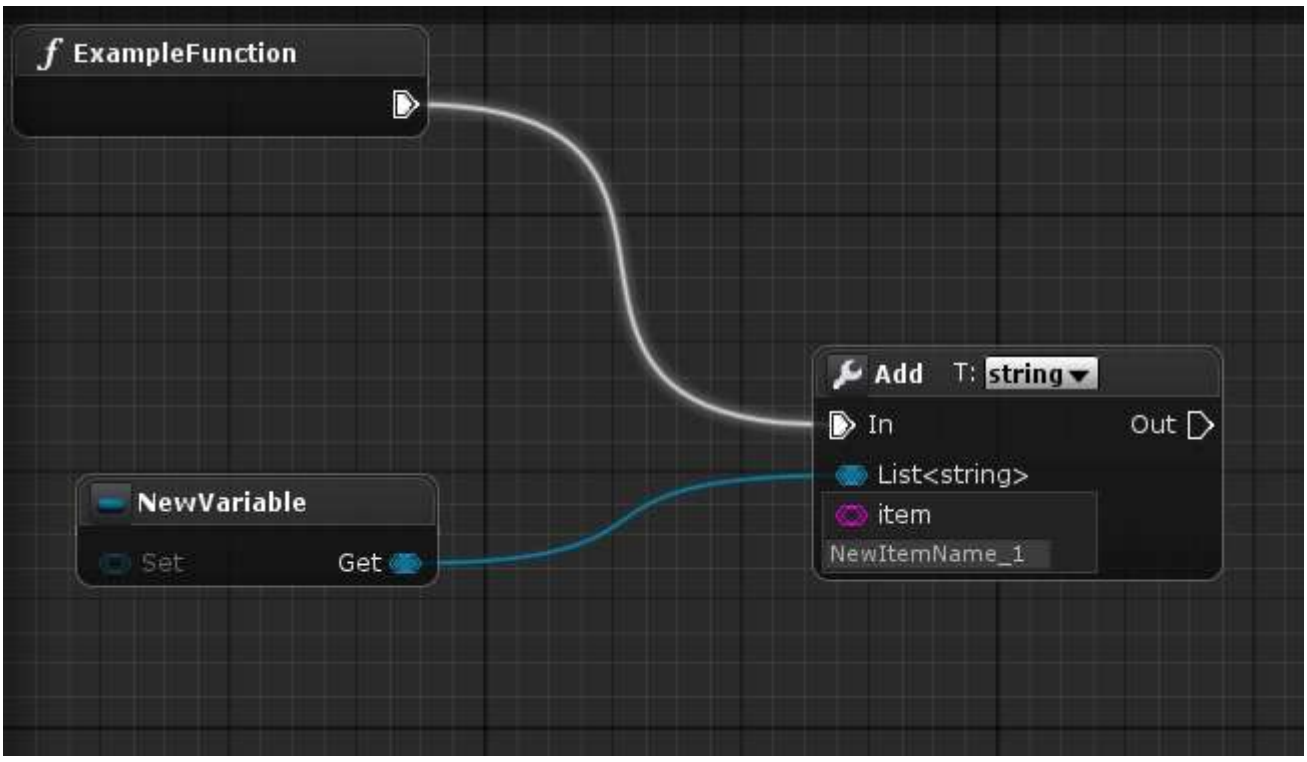
11.3. Adding new objects to collection.

Find method “**Add**” from dragging out pin of variable with “Equal type” option is enabled. Owner class must be **List<T>**:

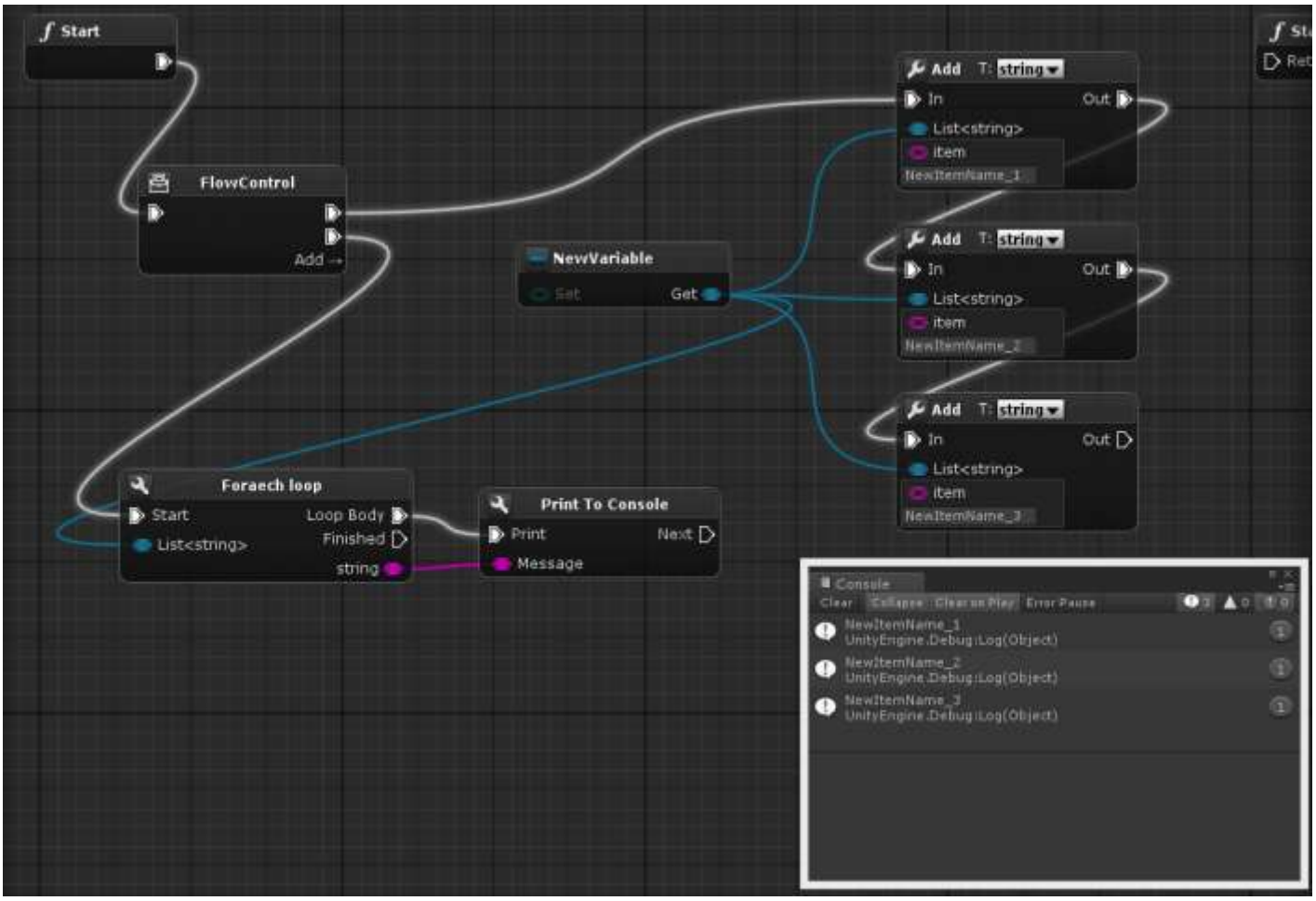


After creation node it will be linked with this function node and its generic type will be automatically defined as “string”.

Connect node with sequence connector and define new adding item name:



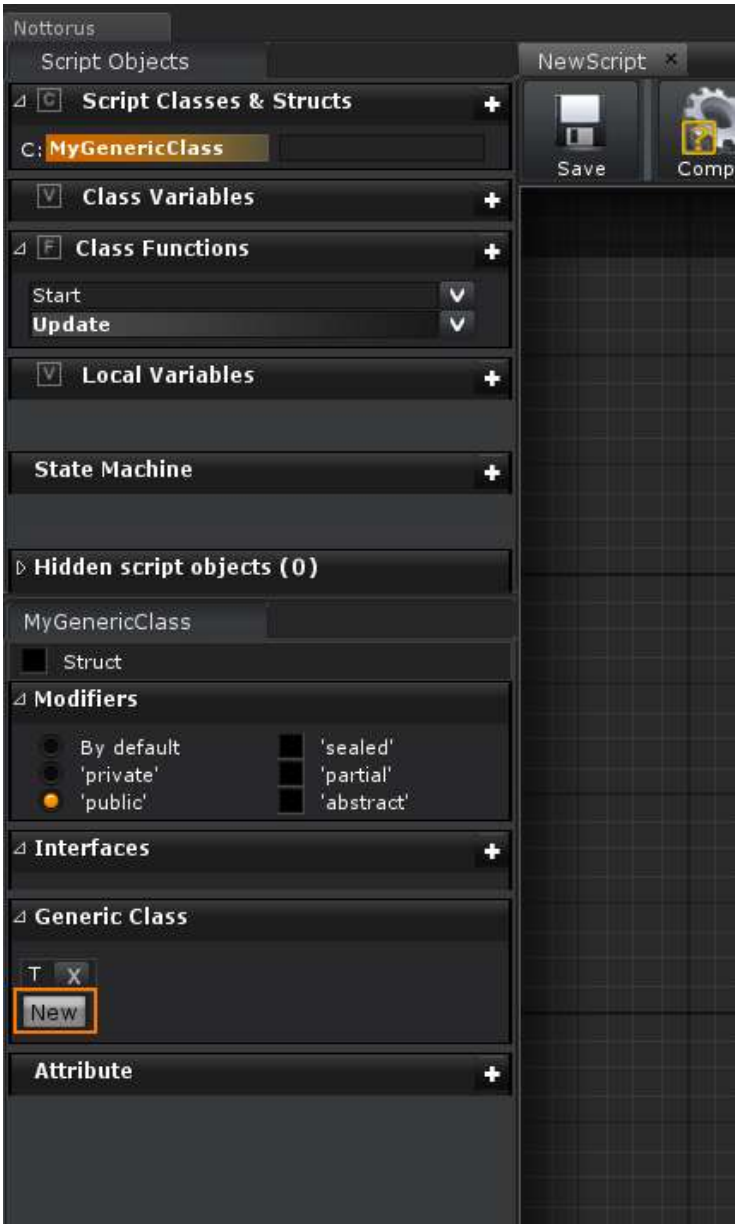
Add few new objects to collection in same way and connect it to “**Foreach loop**” node. It will iterate through all objects of collection and we can print to console all values (with default node "**Print ot console**"):



12.Creating Generics

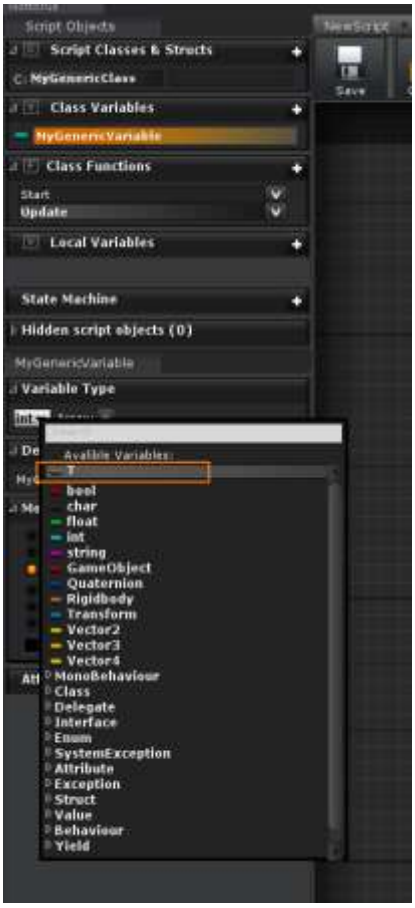
Generic classes can operate or be a holder for variables of defined type.

Generic class can be created with special option “Generic class” in class details window. Press the “New” button to add the generic parameter to your class:



There are three available generic parameters that can be used, but in most cases used only one parameter. Each parameter represents a Type that can be changed (defined) when the instance of class will be created.

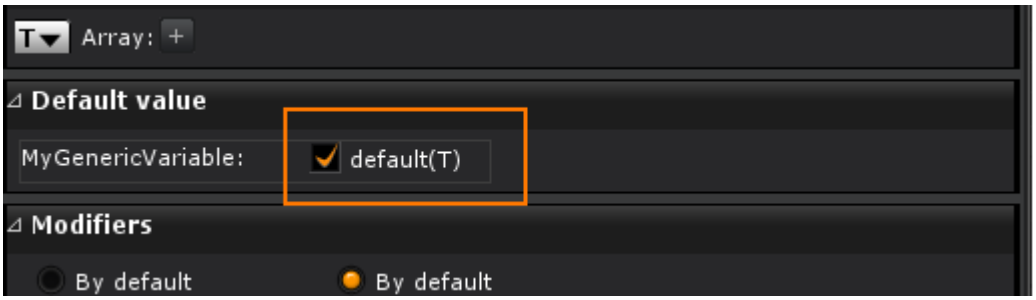
Now our class became generic class (MyGebericClass<T>) and it allow using a generic parameter inside class as Type of variable:



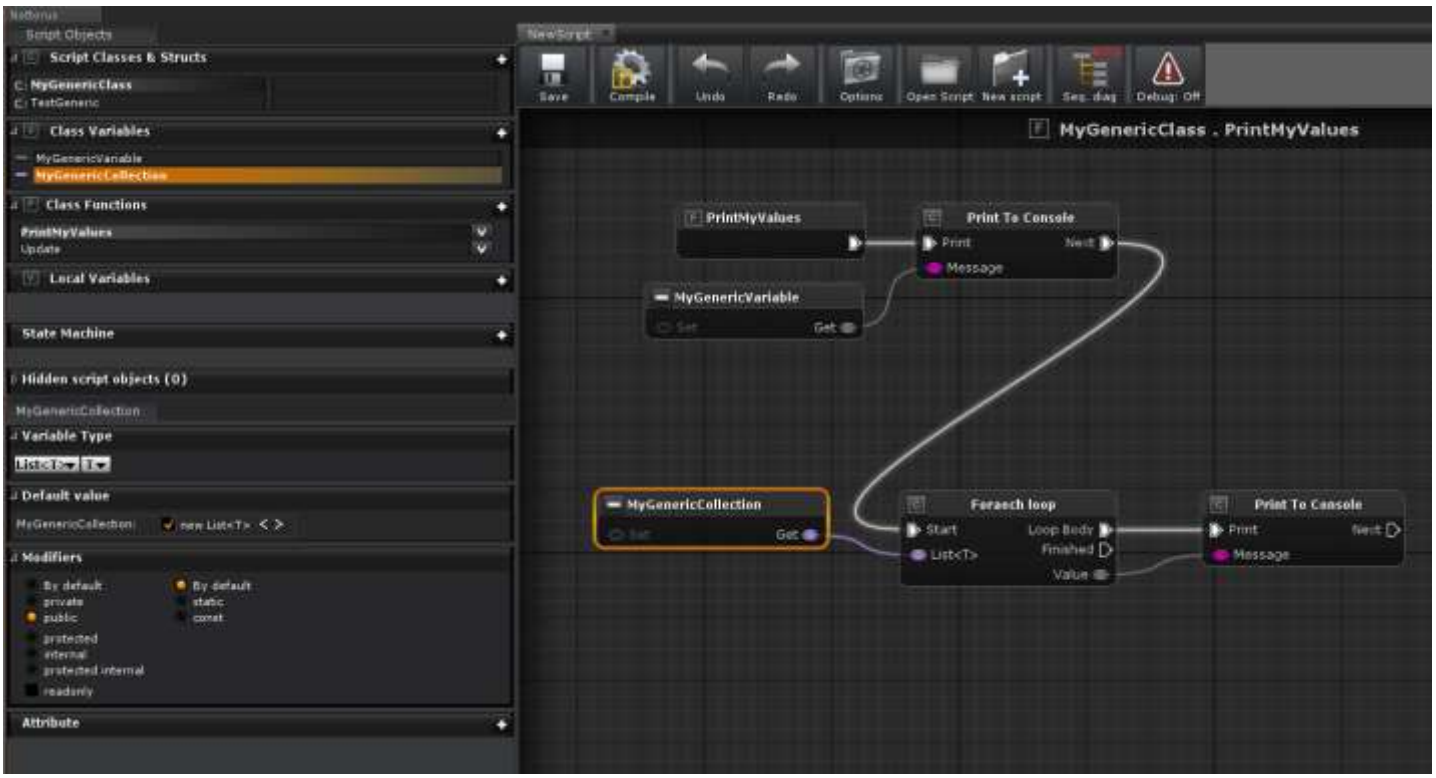
or collection of variables:

Don't forget to set "Default value" option for these variables if it required. Generic Types has only one default value (default(X)), which will set default value of this variable type. Check Default Values Table for more details.

Note: the "default" option will return null for reference types (class, enum, interface) and zero for numeric value types.

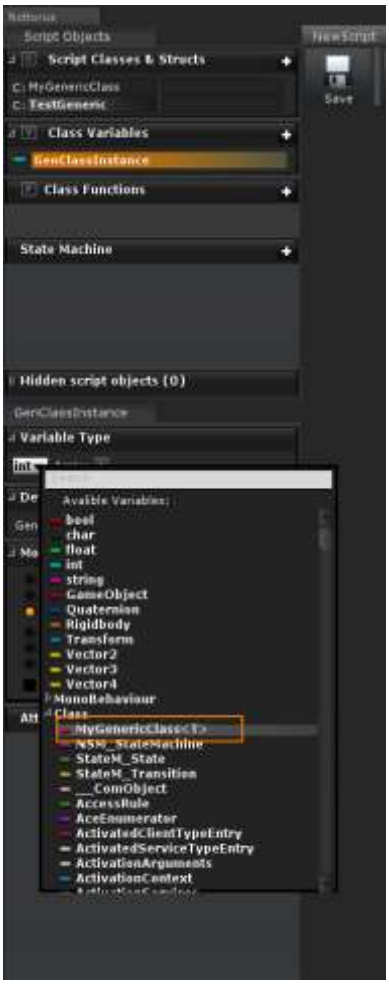


To test these variables we will add a function which will print all of their values:

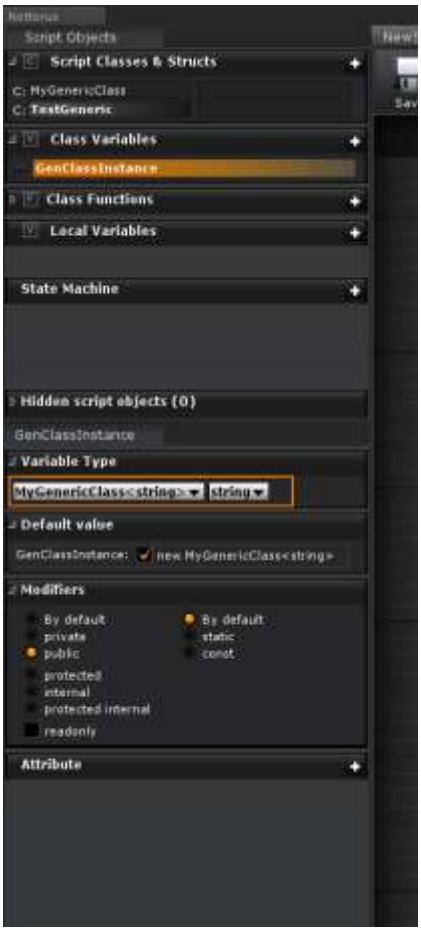


Let's test it!

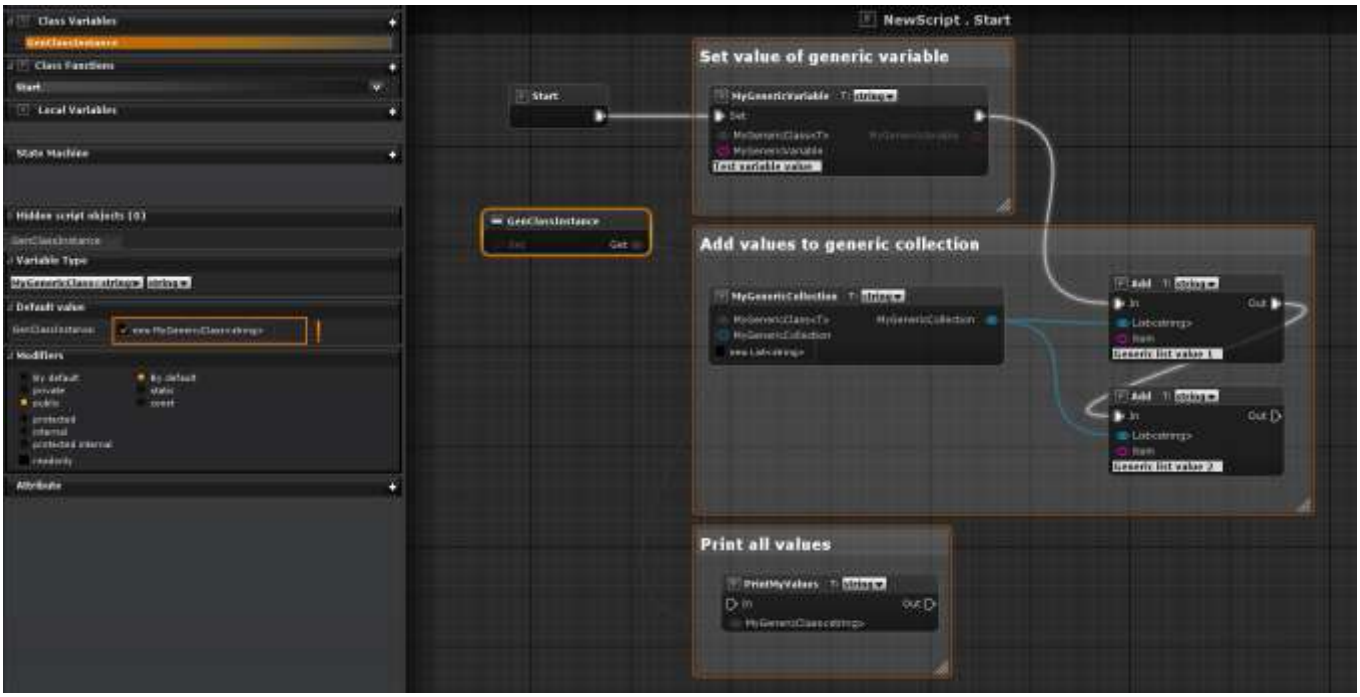
Now create the instance of our new class,



, set the type of generic parameter,



add few values to a generic collection and print all values using our test function:



Note: if the generic parameter type of these nodes was not defined automatically you can change them from dropdown list by pressing generic parameter button on nodes header (from T to string type).