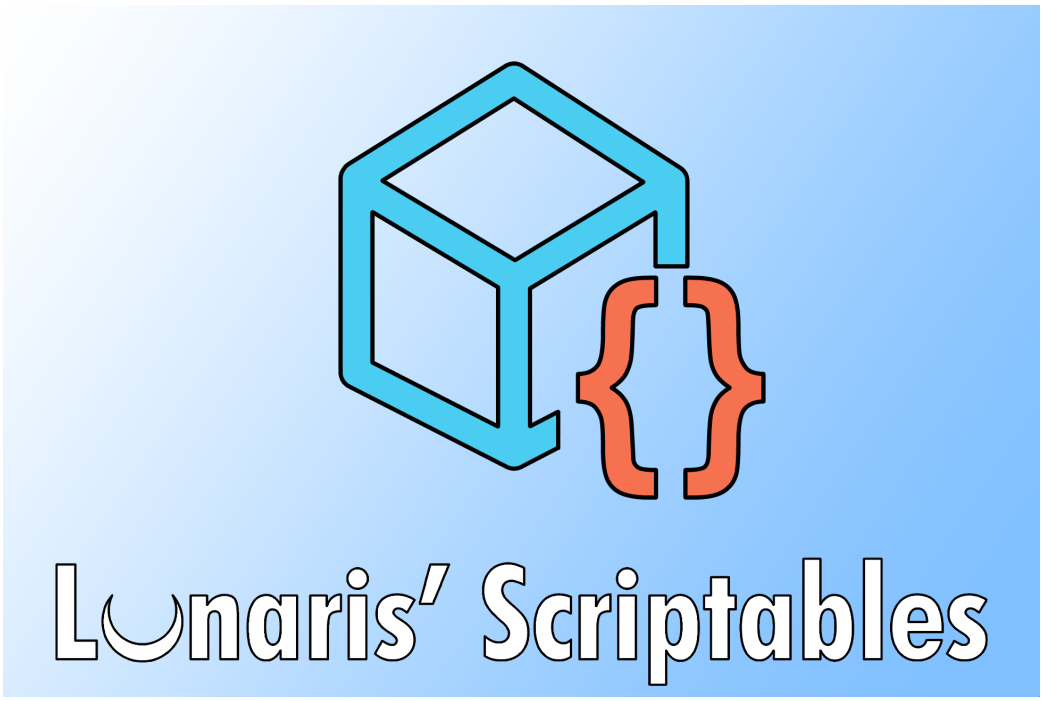
Lunaris' Scriptables Documentation



Contents

1 Overview

Lunaris' Scriptables is a complete system for scriptable object in unity, that makes it fast and easy to create and manage scriptable object with little to no coding experience, as well as making the unity project cleaner and easier to manage by using an event and data driven architecture, observer patterns and a Model View Control (MVC) likes architecture. By using these techniques Lunaris' Scriptables aims to makes your unity project:

1. Better enforce single responsibility patterns.
2. Reduce dependencies and editor serialization.
3. Less reliable on singletons and global managers.
4. More modular and independent of other system.
5. More and easier testable and debuggable at in-editor runtime.
6. Cleaner and more organized.
7. Faster to create and setup scriptable object and data structures.

Lunaris' Scripables is partly based on Ryan Hipple's 2017 Unite talk "Game Architecture with scriptable objects" (`https://www.youtube.com/watch?v=raQ3iHhE_Kk`), and can get you started if you're completely new to scriptable object and architectures using scriptable object. There is also a great blog post here: `https://unity.com/how-to/architect-game-code-scriptable-objects` which have a lot of resources and information.

A copy of the newest verison of this documentation can be found here: `https://www.jacksendary.dk/Documents/LunarisScriptablesDocumentation.pdf`

2 Getting Started

2.1 Specifications

Lunaris' Scriptables was made in Unity LTS 2020.3.1 and should work with all newer versions of the Unity editor. Older versions of unity may also work but is not officially supported. Bear in mind that individual versions of the Unity editor may suffer from bugs that may affect Lunaris' Scriptable negatively (especially editors). If this should happen please refer to `https://issuetracker.unity3d.com/` to check if it may be related to unity and when notify the developer.

All Code exists in namespace "Lunaris.Scriptables" and sub namespaces here of.

2.2 Licensing

All redistribution and/or uploading of any or all parts of Lunaris' Scriptables in public accessible domains, mediums, repositories or other ways of sharing is strictly prohibited! The code may be uploaded in private repositories where all contributors have a valid license. Please also refer to Unity's Asset Store Terms of Service and EULA `https://unity3d.com/legal/as_terms`

2.3 Installation

After purchasing Lunaris' Scriptables do the following steps to add it to your project:

1. Download Lunaris' Scriptables by either:
   (a) Go to you "My assets" or Lunaris' Scriptables asset store page and click "Open in Unity". A popup may prompt you to ask you

to open with unity. Select yes/okay. This will open the package manager in unity and should have Lunaris' Scriptables selected.

(b) Inside unity navigate to Window→Package Manager. In the "Package" drop down select "My Assets" and find Lunaris' Scriptables in the list. Using the search bar in the top right if it doesn't show us.

2. Click the "Download" button in the package manager window and wait for it to complete downloading.

3. When the download is done click the "Import" button and await the for unity to open the "import unity package" window. Make sure everything is selected and then in this window click the "Import" button.

4. Unity should now be importing Lunaris' Scriptables and it should shortly after be ready to use.

3 Lunaris' Scriptables Types

Lunaris' Scriptables comes out of the box with the posibility to generate the following types that is listed in this chapter. Each type will be introduced and briefly explained how they works and what the types can/was meant to be used for. Most of these types can be changed at run time and remember values across scenes and entering/exiting playmode. All the types Lunaris' Scriptable comes with is mostly meant to be base classes for objects/classes generated by the Lunaris' Scriptables' Code Generator which is explained with examples how works in chapter 4 Lunaris' Scriptables Code Generator, as they are generic as theirs basetype and therefore can not by it self be created as a scriptableobject. All examples in this chapter is presuming the type have been generated with a string type using the standard out of the box settings. However the type can be a object and should show up as long as Unity can serialize the type. If Unity cannot serialize the type, the default value will be null.

3.1 Variables

Variables is used to store a variable or object of any type. Variables makes it possible to create a reference to a variable and get or set its value instead of having to reference entire scripts or objects and changing values in each of them.

Properties:

Value	Gets or sets the current value of the variable object.

Figure 3.1: The editor of a Variable, in this case a String Variable. Notice that the value can be both seen and changed in the editor, even in play mode.

3.2 Observable Variables

Observable Variables is like variables used to store a variable or object of any type, but whenever its value is changed it will trigger an chain of events both before changing and after the value has changed. With Observable Variables it is possible to be reactive to changes made to the variable from other source through events. **Note** that sub variables of complex objects does not trigger the events when changed, only when the property "Value" is set/changed.

Properties:

Value	Gets or sets the current value of the variable object.
ValueChangedGameEvent	Gets the Game Event which invokes after the value is changed with additional information of the new and previous value.
ValueChangeGameEvent	Gets the Game Event which invokes after the value is changed with only the new value.
ValueChangingGameEvent	Gets the Game Event which invokes before the value is changed with additional information of the new and previous value.

Events:

OnValueChanged	Event which invokes after the value is changed with additional information of the new and previous value. NOTE: This value can NOT be serialized/shown in the editor. This event is meant for coders as an easier alternative to ValueChangedGameEvent.
OnVariableChange	Event which invokes after the value is changed. NOTE: This value can NOT be serialized/shown in the editor. This event is meant for coders as an easier alternative to ValueChangeGameEvent.
OnValueChanging	Event which invokes before the value is changed with additional information of the new and previous value. NOTE: This value can NOT be serialized/shown in the editor. This event is meant for coders as an easier alternative to ValueChangingGameEvent.
UIValueChangeEvent	Event which invokes after the value is changed. This Event is meant to be used with observable UI components.

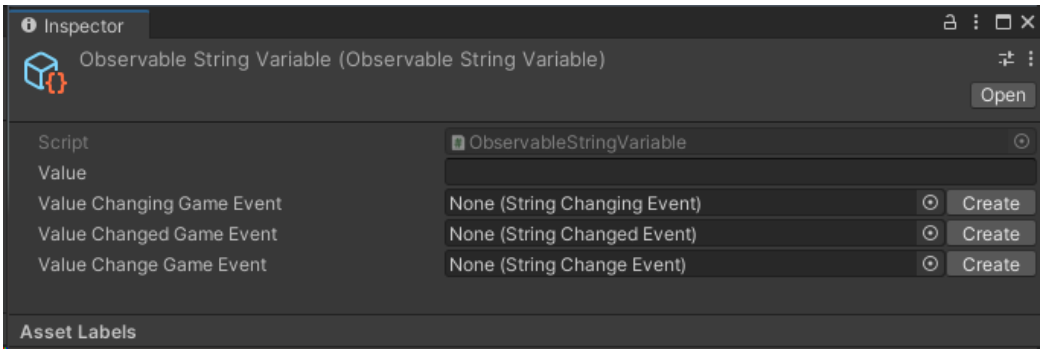


Figure 3.2: The editor of an Observable Variable, in this case a Observable String Variable. Notice that the value can be both seen and changed in the editor, as well as the game events, even in play mode.

3.3 Collection

Collections is used to store, you guessed it! a collection- or multiple variables of a certain type. Just like variables this makes it possible to reference a collection of values and get or set each of the values instead of having to reference entire scripts or objects and changing values in each of them.

Properties:

Count	Gets the number of elements contained in the collection.
Item[Int32]	Gets or sets a element in the collection at the specified index.

Methods:

Add(T)	Adds an object to the end of the collection.
Clear()	Removes all elements from the collection.
Contains(T)	Determines whether an element is in the collection.
CopyTo(T)	Copies a range of elements from the collection to a compatible one-dimensional array, starting at the beginning of the target array.
CopyTo(T, Int32)	Copies a range of elements from the collection to a compatible one-dimensional array, starting at the specified index of the target array.
Dequeue()	Removes and returns the object at the beginning of the collection.
Enqueue(T)	Adds an object to the end of the collection.
GetEnumerator()	Returns an enumerator that iterates through the collection.
IndexOf(T)	Searches for the specified object and returns the zero-based index of the first occurrence within the collection.
Insert()	Inserts an element into the collection at the specified index.
Pop()	Removes and returns the last element of the collection.
Push(T)	Adds an object to the end of the collection.
Remove(T)	Removes the first occurrence of a specific object from the collection.
RemoveAt(Int32)	Removes the element at the specified index of the collection.

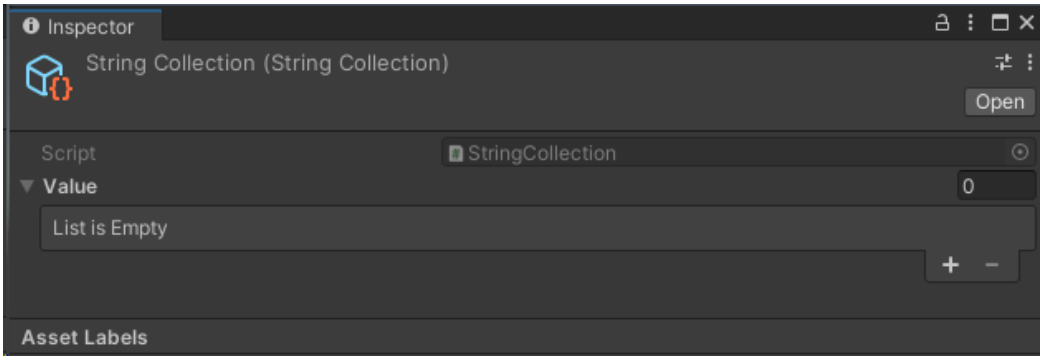


Figure 3.3: The editor of a collection, in this case a String Collection. Notice that values can be added and removes and each value can be edited individually in the editor, even in play mode.

3.4 Observable Collection

Observable Collections is similar to a normal collection, but trigger events whenever the collection or a value in it is changing or changed. Similar to Observable Variables, this makes it is possible to be reactive to changes, made to the collection and variables, from other source through events. **Note** that sub variables of complex objects does not trigger the events when changed, only when the property "Value" is set/changed.

Properties:

Count	Gets the number of elements contained in the collection.
Item[Int32]	Gets or sets a element in the collection at the specified index.
CollectionChanged-GameEvent	Gets the Game Event which invokes after the collection or a value in it changed with additional information of the new and previous value, as well as affacted collection, index and how the collection/value was changed.
CollectionChanging-GameEvent	Gets the Game Event which invokes before the collection or a value in it changed with additional information of the new and previous value, as well as affacted collection, index and how the collection/value was changed.

Methods:

Add(T)	Adds an object to the end of the collection.
Clear()	Removes all elements from the collection.
Contains(T)	Determines whether an element is in the collection.
CopyTo(T)	Copies a range of elements from the collection to a compatible one-dimensional array, starting at the beginning of the target array.
CopyTo(T, Int32)	Copies a range of elements from the collection to a compatible one-dimensional array, starting at the specified index of the target array.
Dequeue()	Removes and returns the object at the beginning of the collection.
Enqueue(T)	Adds an object to the end of the collection.
GetEnumerator()	Returns an enumerator that iterates through the collection.
IndexOf(T)	Searches for the specified object and returns the zero-based index of the first occurrence within the collection.
Insert()	Inserts an element into the collection at the specified index.
Pop()	Removes and returns the last element of the collection.
Push(T)	Adds an object to the end of the collection.
Remove(T)	Removes the first occurrence of a specific object from the collection.
RemoveAt(Int32)	Removes the element at the specified index of the collection.

Events:

OnCollectionChanged	Event which invokes after the collection or a value in it changed with additional information of the new and previous value, as well as affacted collection, index and how the collection/value was changed. NOTE: This value can NOT be serialized/shown in the editor. This event is meant for coders as an easier alternative to ValueChangedGameEvent.
OnCollectionChanging	Event which invokes before the collection or a value in it changed with additional information of the new and previous value, as well as affacted collection, index and how the collection/value was changed. NOTE: This value can NOT be serialized/shown in the editor. This event is meant for coders as an easier alternative to ValueChangingGameEvent.
UIValueChangeEvent	Event which invokes after the value is changed. This Event is meant to be used with observable UI components.

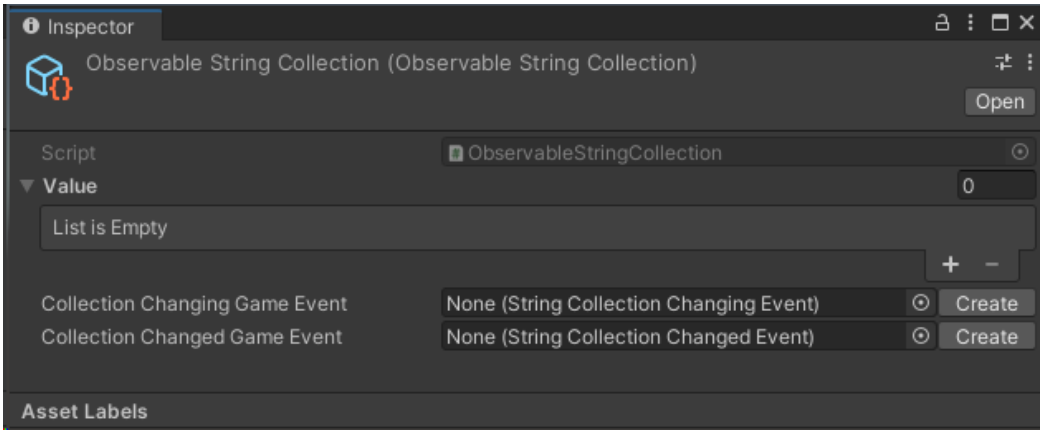


Figure 3.4: The editor of an observable collection, in this case an Observable String Collection. Notice that values can be added and removes and each value can be edited individually in the editor, as well as the game events, even in play mode.

3.5 Events

Events is based on an observer design pattern that enables a subscriber to register and receive notifications from a provider, when an event is invoked/executed. It is suitable for any scenario that requires push-based notification. Events is created as scriptableobject assets and can be attached to an event listener which may have multiple registered subscribers. Events comes in 5 different kinds; Changing, Changed, Change, CollectionChanging and CollectionChanged. Each event can easily be executed from the editor with a push of a button, with data corresponding to it, as long as that data is serializable by unity. This makes it easy to test the events in the editor and while in play mode.

3.5.1 Changing Event

Changing events is meant to be an events that triggers before a change is made to a value, and is currently used in observable variables to notify listeners that a change is about to happen, and adds arguments about what the values is currently is and what it is about to become.

Methods:

Invoke(T)	Executes all the listening event listeners "OnEventRaised" method which then executes to run the response method(s).
RegisterListener(IGameEventListener< ValueChangingEventArgs< T>>)	Adds an event listener to the event.
UnregisterListener(IGameEventListener< ValueChangingEventArgs< T>>)	Removes an event listener to the event.

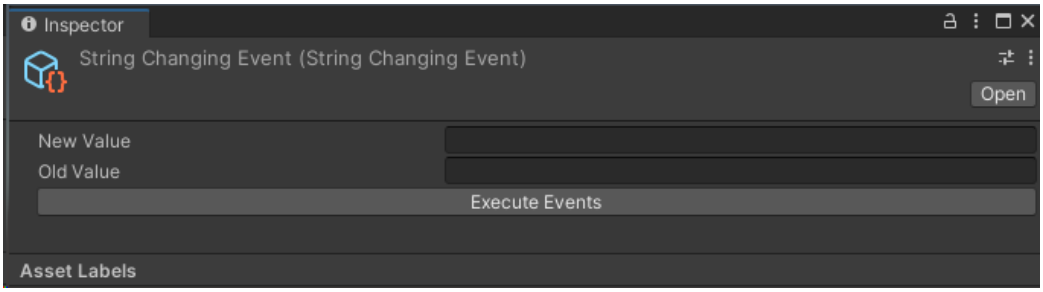


Figure 3.5: The editor of Changing Event, in this case an String Changing Event. Notice that the events can be executed by clicking the "Execute Events" button and the argument parameters can be set above the button to easily test the event. This may only work while in play mode for certain api calls.

Important: The values/data entered in the event editors is not persistent and is lost when the object is deselected.

3.5.2 Changed Event

Changed events is meant to be an events that triggers after a change is made to a value, and is currently used in observable variables to notify listeners that a change has been made to its value, and adds arguments about what the values were before and after the change.

Methods:

Invoke(T)	Executes all the listening event listeners "OnEventRaised" method which then executes to run the target method(s).
RegisterListener(IGameEventListener< ValueChangedEventArgs< T>>)	Adds an event listener to the event.
UnregisterListener(IGameEventListener< ValueChangedEventArgs< T>>)	Removes an event listener to the event.

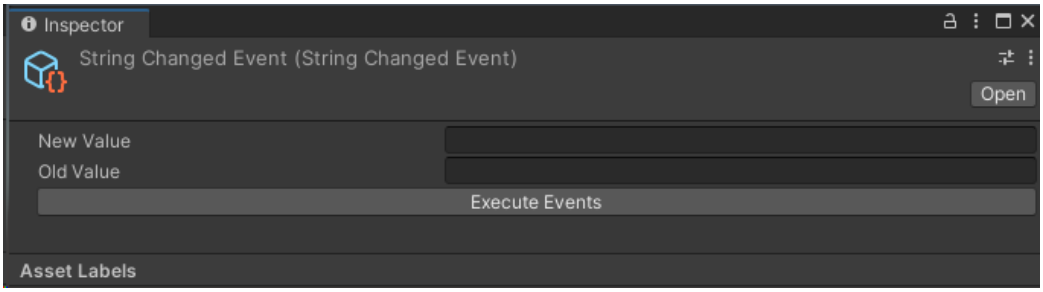


Figure 3.6: The editor of Changed Event, in this case an String Changed Event. Notice that the events can be executed by clicking the "Execute Events" button and the argument parameters can be set above the button to easily test the event. This may only work while in play mode for certain api calls.

3.5.3 Change Event

Change Events is a simplified event that is meant to be used to notify listeners about a change to a value, but only notifies listeners with the new value as a parameter. Change event is the last event to be triggered in observable variables.

Methods:

Invoke(T)	Executes all the listening event listeners "OnEventRaised" method which then executes to run the target method(s).
RegisterListener(IGameEventListener<T>)	Adds an event listener to the event.
UnregisterListener(IGameEventListener<T>)	Removes an event listener to the event.

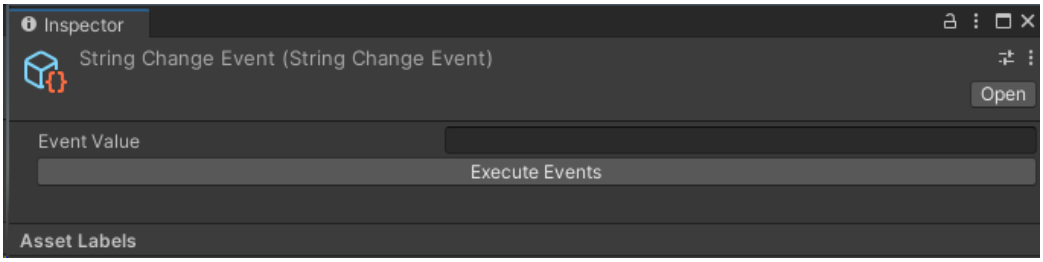


Figure 3.7: The editor of Change Event, in this case an String Change Event. Notice that the events can be executed by clicking the "Execute Events" button and the argument parameters can be set above the button to easily test the event. This may only work while in play mode for certain api calls.

3.5.4 Collection Changing Event

Collection Changing events is meant to be an events that triggers before a change is made to a collection, and is currently used in observable collections to notify listeners that a change will be made to the collection, and adds arguments about what the values were before and after the change, at what index this change is going to happen, the collection it is happening to and what sort of change is happening to the collection.

Methods:

Invoke(T)	Executes all the listening event listeners "OnEventRaised" method which then executes to run the target method(s).
RegisterListener(IGameEventListener< CollectionChangingEventArgs< T>>)	Adds an event listener to the event.
UnregisterListener(IGameEventListener< CollectionChangingEventArgs< T>>)	Removes an event listener to the event.

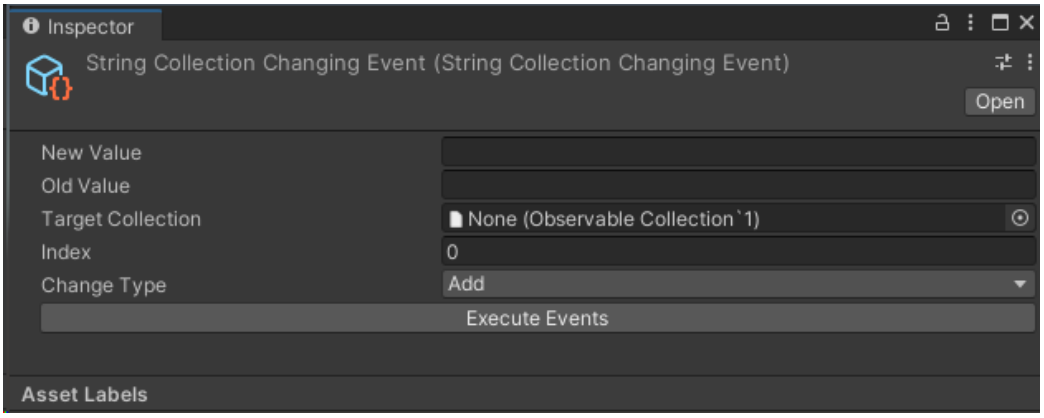


Figure 3.8: The editor of Collection Changing Event, in this case an String Collection Changing Event. Notice that the events can be executed by clicking the "Execute Events" button and the argument parameters can be set above the button to easily test the event. This may only work while in play mode for certain api calls.

3.5.5 Collection Changed Event

Collection Changed events is meant to be an events that triggers after a change has been made to a collection, and is currently used in observable collections to notify listeners that a change has been made to the collection, and adds arguments about what the values were before and after the change, at what index this change has happened, the collection it is happening to and what sort of change has happened to the collection.

Methods:

Invoke(T)	Executes all the listening event listeners "OnEventRaised" method which then executes to run the target method(s).
RegisterListener(IGameEventListener< CollectionChangedEventArgs< T>>)	Adds an event listener to the event.
UnregisterListener(IGameEventListener< CollectionChangedEventArgs< T>>)	Removes an event listener to the event.

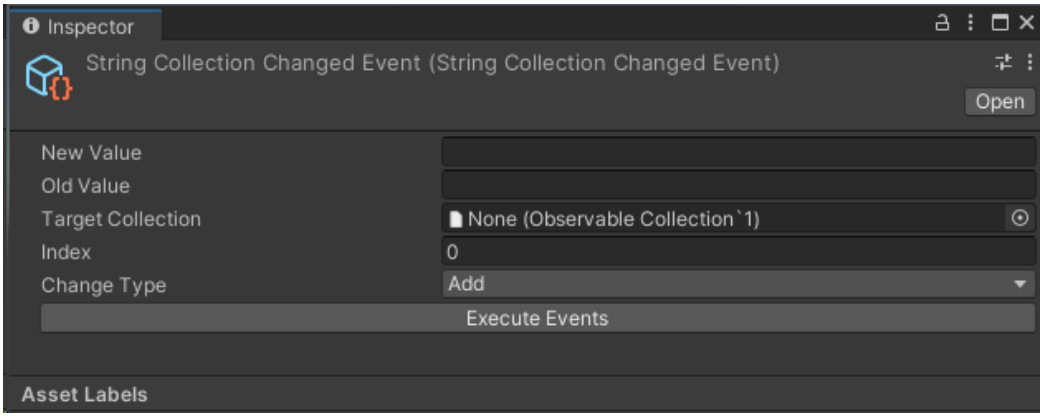


Figure 3.9: The editor of Collection Changed Event, in this case an String Collection Changed Event. Notice that the events can be executed by clicking the "Execute Events" button and the argument parameters can be set above the button to easily test the event. This may only work while in play mode for certain api calls.

3.6 Event Listeners

Event listeners is used for listening/catching event notifications when they are invoken. An evnet listener need to be of the same generic type as the event it need to listen for. Event listeners will run all response methods in order when the registered game event is invoked. When using event listeners in custom scripts you will need to call RegisterListener for it to be able to receive the events, and UnregisterListener when you want to stop it from receiving events.

Properties:

GameEvent	Gets the GameEvent that is being listened to.
Response	Gets the UnityEvent that is the response(s) to the GameEvent.

Methods:

OnEventRaised(T)	Invokes the listening Response's method(s).
RegisterListener()	Register the GameEvent to the EventListener. This is done automatically on Event Listener Behaviors OnEnable.
UnregisterListener()	Unregister the GameEvent from the EventListener. This is done automatically on Event Listener Behaviors OnDisable.

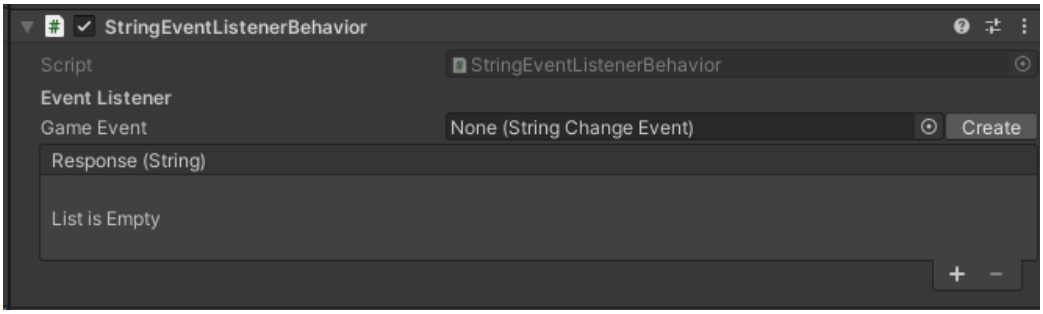


Figure 3.10: The editor of an Event Listener and an Event Listener behavior, in this case an String Event Listener and String Event Listener Behavior.

3.6.1 Event Listener Behaviors

Event Listener Behaviors is very similar to the plain Event Listener except it wraps the listener in a monobehavior that automatically calls RegisterListener and UnregisterListener when ever the gameobject/script is enabled/disabled.

3.7 Event Arguments

Event Arguments is generic classes that contains data related to their corresponding events. This data is sent as a parameter to the listening responses/events methods when an event is invoked. Each of the event arguments has their members displayed and explained in the following sections.

3.7.1 ValueChangeEventArgs

Constructors:

ValueChangeEventArgs <T>(T, T)	Initializes a new instance of ValueChangeEventArgs with the new and old values.

Properties:

NewValue	Gets or Sets the new value variable.
OldValue	Gets or Sets the old value variable.

3.7.2 ValueChangingEventArgs

Constructors:

ValueChangingEventArgs <T>(T, T)	Initializes a new instance of ValueChangingEventArgs with the new and old values.

Properties:

NewValue	Gets or Sets the new value variable. (Inherited from ValueChangeEventArgs)
OldValue	Gets or Sets the old value variable. (Inherited from ValueChangeEventArgs)

3.7.3 ValueChangedEventArgs

Constructors:

ValueChangedEventArgs <T>(T, T)	Initializes a new instance of ValueChangedEventArgs with the new and old values.

Properties:

NewValue	Gets or Sets the new value variable. (Inherited from ValueChangeEventArgs)
OldValue	Gets or Sets the old value variable. (Inherited from ValueChangeEventArgs)

3.7.4 CollectionChangeEventArgs

Constructors:

CollectionChangeEventArgs <T>(T, T, ObservableCollection <T>, long, ChangeType)	Initializes a new instance of CollectionChangeEventArgs with the new and old values.

Properties:

ChangeType	Gets or Sets the changetype which indicates how the collection was changed. (Inherited from ValueChangeEventArgs)
Collection	Gets or Sets the collection which was changed. (Inherited from ValueChangeEventArgs)
Index	Gets or Sets the index of where the change happened. (Inherited from ValueChangeEventArgs)
NewValue	Gets or Sets the old value variable. (Inherited from ValueChangeEventArgs)
OldValue	Gets or Sets the new value variable. (Inherited from ValueChangeEventArgs)

3.7.5 CollectionChangingEventArgs

Constructors:

CollectionChangingEventArgs <T>(T, T, ObservableCollection < T>, long, ChangeType)	Initializes a new instance of CollectionChangingEventArgs with the new and old values.

Properties:

ChangeType	Gets or Sets the changetype which indicates how the collection was changed. (Inherited from CollectionChangeEventArgs)
Collection	Gets or Sets the collection which was changed. (Inherited from CollectionChangeEventArgs)
Index	Gets or Sets the index of where the change happened. (Inherited from CollectionChangeEventArgs)
NewValue	Gets or Sets the old value variable. (Inherited from ValueChangeEventArgs)
OldValue	Gets or Sets the new value variable. (Inherited from ValueChangeEventArgs)

3.7.6 CollectionChangedEventArgs

Constructors:

CollectionChangedEventArgs<T>(T, T, ObservableCollection <T>, long, ChangeType)	Initializes a new instance of CollectionChangedEventArgs with the new and old values.

Properties:

ChangeType	Gets or Sets the changetype which indicates how the collection was changed. (Inherited from CollectionChangeEventArgs)
Collection	Gets or Sets the collection which was changed. (Inherited from CollectionChangeEventArgs)
Index	Gets or Sets the index of where the change happened. (Inherited from CollectionChangeEventArgs)
NewValue	Gets or Sets the old value variable. (Inherited from ValueChangeEventArgs)
OldValue	Gets or Sets the new value variable. (Inherited from ValueChangeEventArgs)

3.8 References

References, or ScriptableReferences is used to one of any specific value, scriptable variables, observable variables or converters. This makes is easier to create code that easier can meet demands of designers working in the engine, but also eliminate uncertaincy about what type to use at different times.

Constructors:

ScriptableReference<T>()	Instantiate a new instance of ScriptableReference which can be used to hold a value coming from different sources.

Fields:

referenceType	Determines the type of the variable the property Value should return.
value	A value or object of type T.
variable	A ScriptableVariable of type T.
observableVariable	A ScriptableObservableVariable of type T.
converter	A ValueConverter which convert to type T.
function	A LunarisFunction of type T.

Properties:

Value	Gets or sets the value of the reference depending on the field referenceType.

Events:

UIValueChangeEvent	Event which invokes after the value is changed. This Event is meant to be used with observable UI components.

Figure 3.11: The editor/property drawer for ScriptableReferences, in this case a StringReference. Currently it is showing an input field which accepts any string input, however clicking the button with the three dots will allow you to change between value, scriptable variables, observable variables or converters.

3.9 Converters

Converters is used to convert from one type to another. Converters makes it possible convert values, objects, scriptable variables, observable variables and even other converters into another type. Lunaris Scriptables' comes with the a few example converters, but most converters will need to be created and methods implemented to suit your specific need to convert data correctly.

Constructors:

ValueConverter<T, F>()	Class which converts a from type F to T or back.

Fields:

value	The from value which should be converted.

Properties:

Value	Gets the converted value of the converter, or sets the value of the converter by converting the set value.

Methods:

Convert(T)	Convert the value from type F to type T, and returns the converted value.
ConvertBack(F)	Convert the value from type T to type F, and returns the converted value.

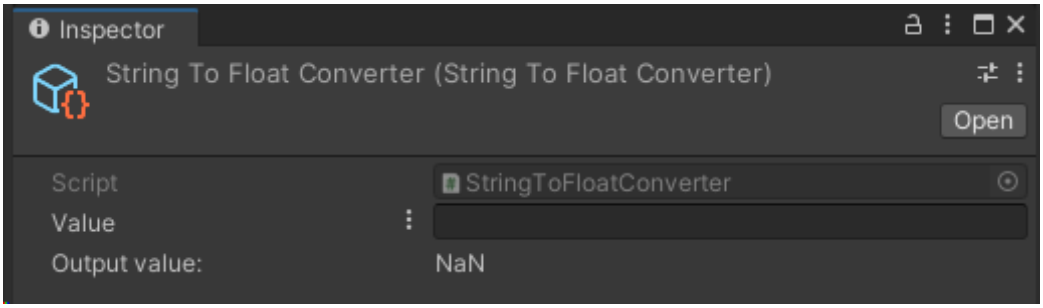


Figure 3.12: The editor of a converter, in this case a String to Float converter. Notice that the converted value can be seen in the bottom of the editor to fast check its behavior. A error help box will be displayed if the converter can not convert a value or an error occuor trying to convert the value.

Important: Please note that converters that have circular references will cause stackoverflows and potentially crash the unity editor which may result in data loss. If unity saves a change where the crash persist, please use the debug inspector/editor to remove the reference coursing issues.

3.10 Actions

Actions, or "LunarisAction" is meant to be used to make small methods with specific behavior(s) which then can be invoked by other scripts. Most actions will need to be created and methods implemented to suit your specific needs.

Constructors:

LunarisAction()	Abstract class of LunarisAction without type parameter.
LunarisAction<T>()	Abstract class of LunarisAction with one type parameter.
LunarisAction<T1, T2>()	Abstract class of LunarisAction with two type parameter.
LunarisAction<T1, T2, T3>()	Abstract class of LunarisAction with three type parameter.
LunarisAction<T1, T2, T3, T4>()	Abstract class of LunarisAction with four type parameter.

Methods:

Invoke()	Invokes the method of the action.
Invoke(T)	Invokes the method of the action with one parameter.
Invoke(T1, T2)	Invokes the method of the action with two parameters.
Invoke(T1, T2, T3)	Invokes the method of the action with three parameters.
Invoke(T1, T2, T3, T4)	Invokes the method of the action with four parameters.

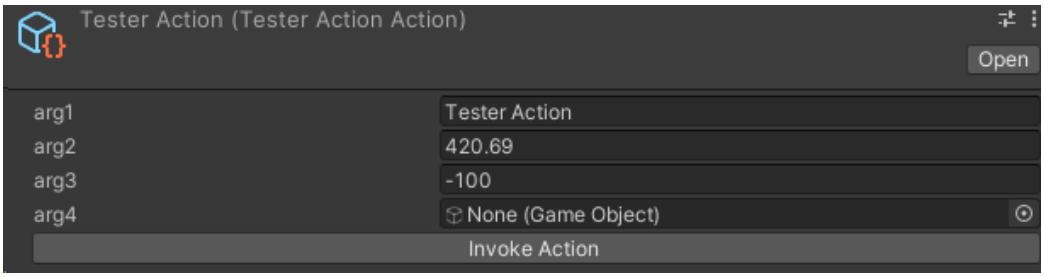


Figure 3.13: The editor of an action, in this case an action taking 4 type parameters; string, float, int and a gameobject. The parameters will differ depending on your specific implementation of an action.

3.11 Functions

Functions, or "LunarisFunction" is meant to be used to make small methods with specific behavior(s) which then can be invoked by other scripts. Most functions will need to be created and methods implemented to suit your specific needs. Unlike actions, functions always have a return type.

Constructors:

LunarisFunction<TResult>()	Abstract class of LunarisFunction without type parameter.
LunarisFunction<T, TResult>()	Abstract class of LunarisFunction with one type parameter.
LunarisFunction<T1, T2, TResult>()	Abstract class of LunarisFunction with two type parameter.
LunarisFunction<T1, T2, T3, TResult>()	Abstract class of LunarisFunction with three type parameter.
LunarisFunction<T1, T2, T3, T4, TResult>()	Abstract class of LunarisFunction with four type parameter.

Methods:

Invoke()	Invokes the method of the function and return a value of type TResult.
Invoke(T)	Invokes the method of the function with one parameter and return a value of type TResult.
Invoke(T1, T2)	Invokes the method of the function with two parameters and return a value of type TResult.
Invoke(T1, T2, T3)	Invokes the method of the function with three parameters and return a value of type TResult.
Invoke(T1, T2, T3, T4)	Invokes the method of the function with four parameters and return a value of type TResult.

3.12 Script Hooks

Script Hooks is used to avoid writing boilerplate code to raise Atoms Events or invoke actions from Unity's MonoBehavior functions. All MonoBehavior functions can be found here: `https://docs.unity3d.com/ScriptReference/MonoBehaviour.html` and Lunaris' Scriptables comes with some of the most commonly used functions.

4 Lunaris' Scriptables Code Generator

After importing Lunaris' Scriptables, a new option inside the Tools menu called "Lunaris" will show up. Navigating to this item will reveal a menu with an item called "Scriptables Code Generator". Clicking this will open up the Lunaris generator window with the posibility to fast and easy create LunarisScriptableObjects without writing a line of code. Please note that refreshing/importing assets will cause the generator window reload and **reset all data**.

Important: Whenever we are browsing/searching/choosing for primitive types we must use the .NET typenames, however the end result will be the primitive types of those. For more information please refer to this link: `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/built-in-types`. Support for search of primitive names and nullable types may be released in the future.

Before using the generators for our project we may want to change the settings of Lunaris' Scriptables to best fit our use case. All settings and what they do is briefly explained in chapter: 5 Settings

4.1 ScriptableObject Generator

When opening Lunaris' Scriptables' Code generator, we are presented with the window shown in figure 4.1. This section we will step by step see how the "ScriptableObject Generator" fast and easy can create ScriptableObjects.

In this example we want to make a simple scriptableobject called "PlayerData" which can hold two variables: Name and Health.

First enter the name "PlayerData" into the text field under the label "Scriptableobject Name". This will be the name of both the class and the generated file.

Next we also write "PlayerData" in the textfield under the label "Create

Asset Menu Path". This is the path where we can create the object asset from the asset menu. For more information please see `https://docs.unity3d.com/ScriptReference/CreateAssetMenuAttribute.html`.

Now click the "Add Variable" button twice. We should now have a window very similar to figure 4.2.

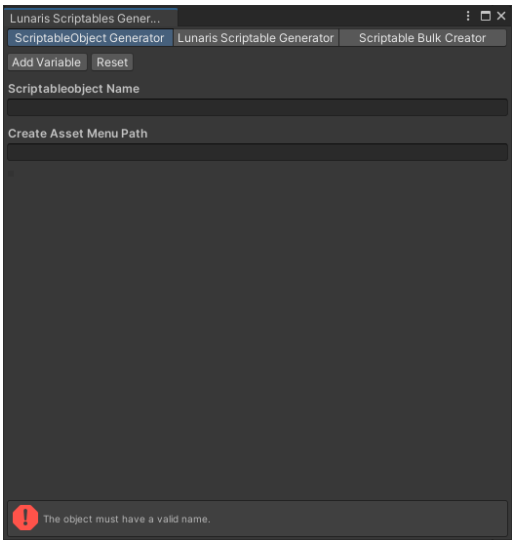


Figure 4.1: The ScriptableObject Generator window upon opening or after clicking reset.

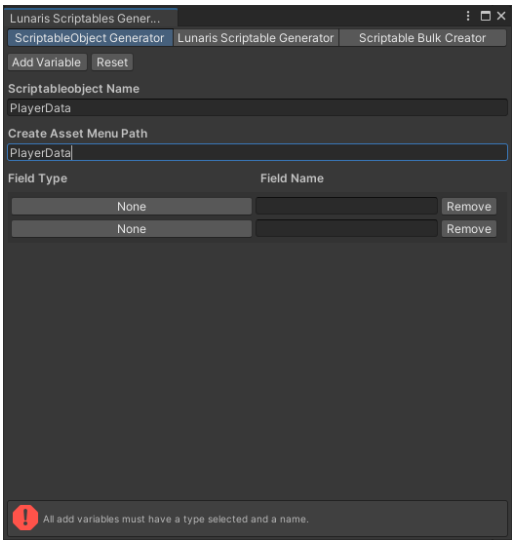


Figure 4.2: The ScriptableObject Generator window after adding name path and fields.

To specify the variable types first click the button with the text "None" under the label "Field Type". This will show a dropdown window, as seen on figure 4.3, in this window it is possible to search and select a target type for a field. Select "System.String" and "System.Single" in the two added variables, and then name them "_name" and "_health". We should now have a window which is similar to the window in figure 4.4.

Notice that in the bottom of the window is an error message if there is something which haven't been filled out correctly. In this case please refer to the message and fix the errors. The Create button should appear as soon as all information is filled in.

It is possible to remove single variables by clicking the "Remove" button to the right in the row, and we can completely reset the window back to its state shown in figure 4.1 by clicking the "Reset" button in the top besides the "Add Variables" button.

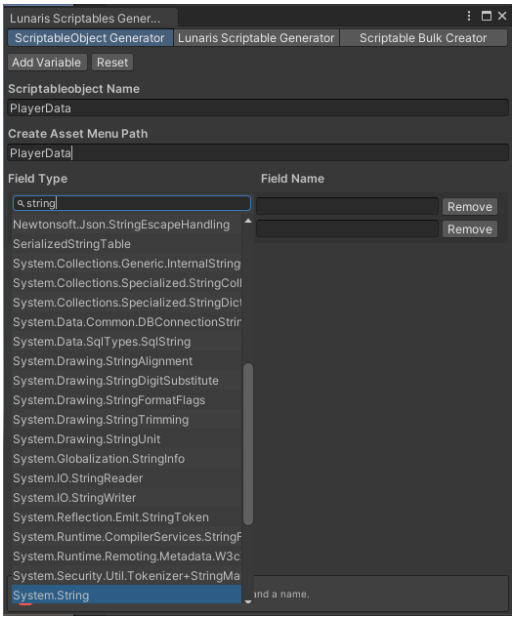


Figure 4.3: The ScriptableObject Generator window with type selector open searching for "string".

Figure 4.4: The ScriptableObject Generator window with all info filled in and ready to generate.

When ready, we can create the object by clicking the button in the bottom of the window called "Create {Scriptableobject Name} Scriptableobject" to create the code/file with the information typed in the Scriptable Generator tab. If we followed along the steps in this section we should now have a new file create at the path specified in the Lunaris' Scriptables' settings, which by default is "Assets/Generated/LunarisScriptables/ScriptableObjects". Please refer to chapter 5 Referenceschapter:settings for further detail. This file should look very similar to the code in listing 4.1.

```
1  using System;
2  using UnityEngine;
3  namespace Lunaris.Scriptables.ScriptableObjects
4  {
5      [CreateAssetMenu(menuName="PlayerData")]
6      public class PlayerData : LunarisScriptableObject
7      {
8          [SerializeField()]
9          public string _name;
10         [SerializeField()]
11         public float _health;
12     }
```

```
13 }
```

Listing 4.1: The generated code after clicking the "Create PlayerData scriptableobject" button shown in the bottom of figure 4.4

Important: Please note that the generators **DOES NOT** remove illegal characters from the files or class name such as dot (.) comma (,) question marks (?) or curly brackets ({}), but may instead throw errors, or be unable to compile the newly created class.

Important: Please note that when generating files with the same name as an already existing class may cause "error CS0101: The namespace 'namespace' already contains a definition for 'name'". If this happen you will need to delete or rename one of the generated classes.

Important: Please note that when generating files with the same path of an already existing file with the same name it will be overridden.

4.2 Lunaris Scriptable Generator

In the top of the Lunaris' Scriptables' generator window is a tabbed menu with the name "Lunaris Scriptable Generator". Clicking this tab menu item will show the window in figure 4.1. This tool makes it possible to make specialized variables, events, collections and eventlisteners. Clicking the add button twice will add two objects to the windows as shown in figure 4.6. As can be seen in figure 4.6 some options in grayed out. This is by design, as some generated items depend on the existence of others to not cause scripting errors.

Figure 4.5: The Lunaris Scriptable Generator window upon opening or after clicking reset.

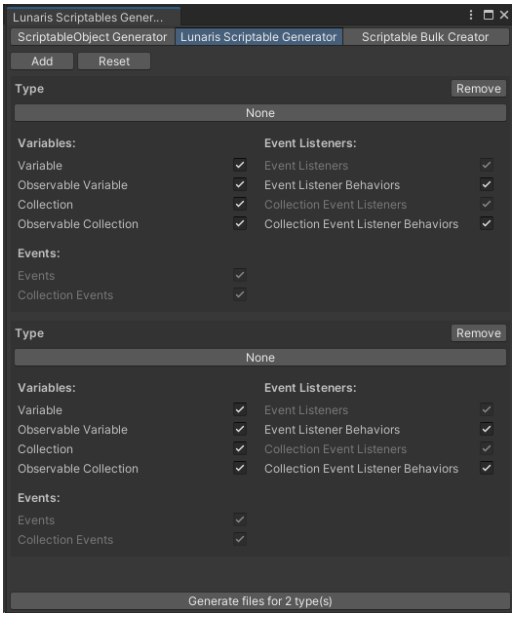


Figure 4.6: The Lunaris Scriptable Generator window after adding two types.

Next click the none buttons for the two types. A dropdown should appear as seen in figure 4.7. Choose "System.String" for the first item and "UnityEngine.Vector3" for the second item.

Now lets presume that we do not need collection types for the Vector3 type, in this case we will simply uncheck Collection, Observable Collection, Collection Event, Collection Event Listeners and Collection Event Listeners Behavior. This should leave us with a window very similar to the window in figure 4.8

It is possible to remove single types by clicking the "Remove" button to the right in the top right of each box, and we can completely reset the window back to its state shown in figure 4.5 by clicking the "Reset" button in the top besides the "Add" button.

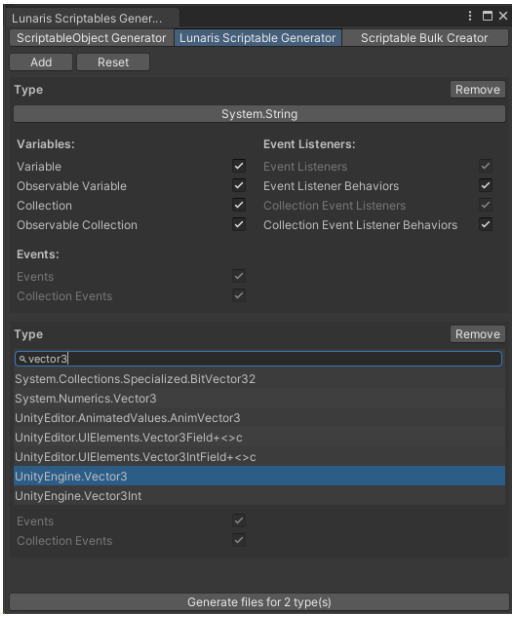


Figure 4.7: The Lunaris Scriptable Generator window with the type selector open and searching for "vector3".

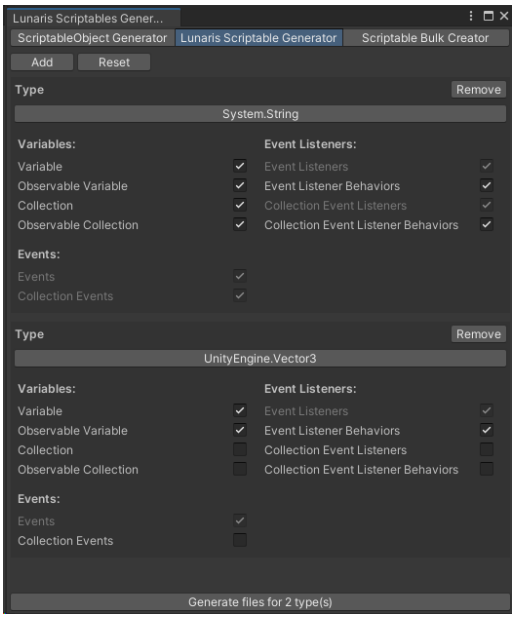


Figure 4.8: The Lunaris Scriptable Generator window with all information entered and some types deselected for the Vector3 type.

Lastly we click the "Generate files for X type(s)" in the bottom of the window. This should generate the files for the type selected and we should now have all the files shown in 4.9

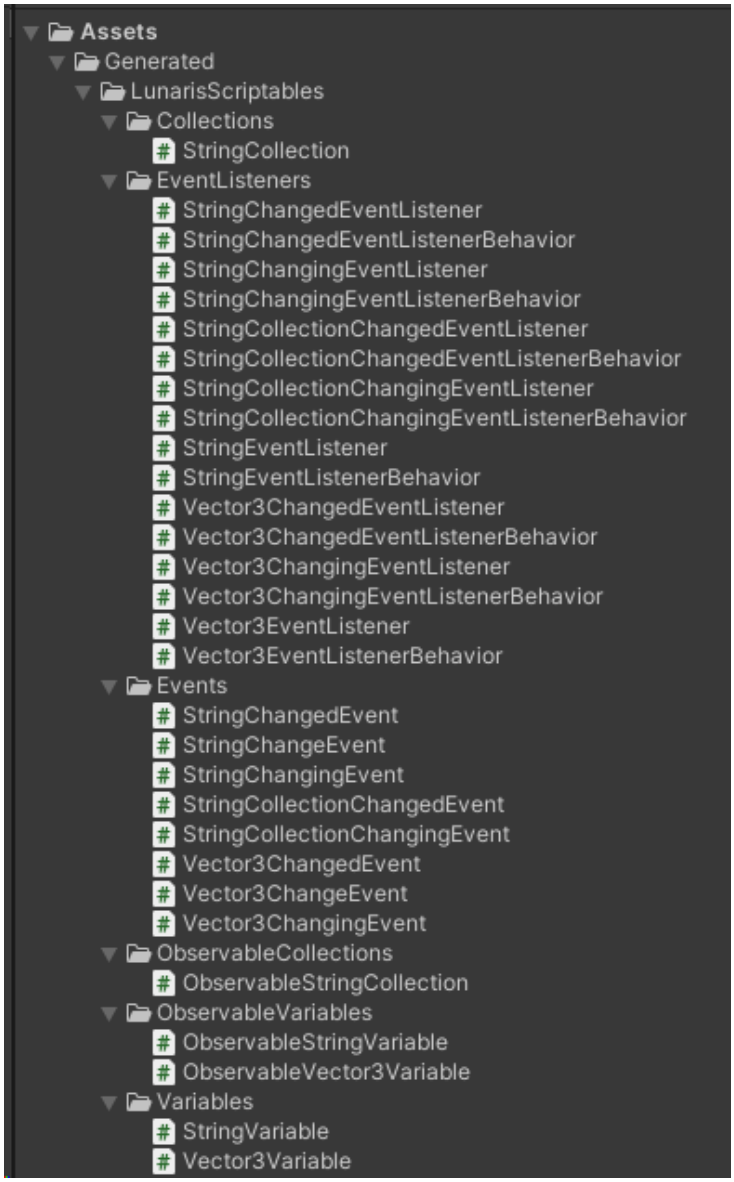


Figure 4.9: The files generated as a result of clicking the "Generate files for X Type(s)" in the Lunaris Scriptable Generator window. Note how no collections have been created for the Vector3 type.

Important: Please note that the generators **DOES NOT** remove illegal characters from the files or class name such as dot (.) comma (,) question marks (?) or curly brackets ({}), but may instead throw errors, or be unable to compile the newly created class.

Important: Please note that when generating files with the same name as an already existing class may cause "error CS0101: The namespace 'names-

pace' already contains a definition for 'name'". If this happen you will need to delete or rename one of the generated classes.

Important: Please note that when generating files with the same path of an already existing file with the same name it will be overridden.

4.3 Scriptable Bulk Creator

The last past of the Lunaris' Scriptables code generator is the Scriptable Bulk Creator. In the top click the "Scriptable Bulk Creator" tab menu to access the tool and you should be presented with a window similar to the one in figure 4.10. This tool makes it possible to fast create multiple scriptable object assets from the comfort of a single place.

To get started and to follow this guide click the "Add ScriptableObject" button in the top twice. This should now have added two rows with a type and a name column and should be similar to figure 4.11.

Figure 4.10: The Scriptable Bulk Creator window upon opening or after clicking reset.

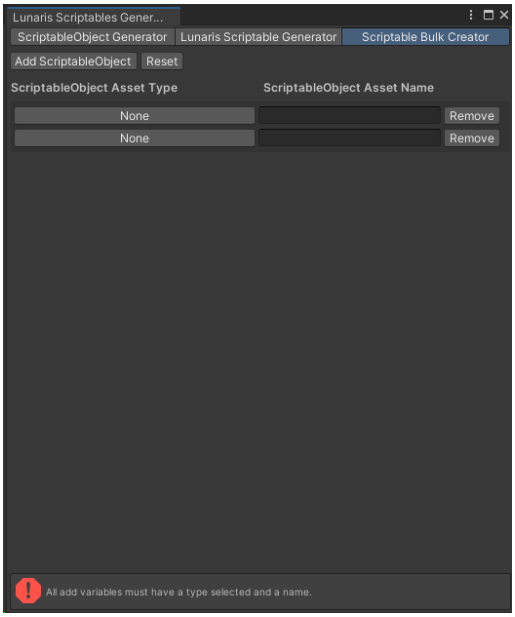


Figure 4.11: The Scriptable Bulk Creator window after adding two types.

Next click the none buttons for the two types. A dropdown should appear as seen in figure 4.12. If you've followed along the guides in both the section 4.1 ScriptableObject Generator and 4.2 Lunaris Scriptable Generator you should now in the dropdown menu be able to see "Lunaris.Scriptables.ScriptableObjects.PlayerDat

and "Lunaris.Scriptables.Variables.Vector3Variable" in the type column. Next enter "Player1Data" and "RespawnLocation" in the name columns. This should leave us with a window very similar to the one in figure 4.13.

Note that by default only scriptableobjects inheriting from "LunarisScriptableObject" is displayed in the dropdown menu. This can be changed in settings, please ref to chapter 5 Settings for an overview over all settings, and the sections 5.10 Only Show LunarisScriptableObjects in Bulk Creator and 5.11 Use Filters for Bulk Creator for settings affecting the Scriptable Bulk Creator.

It is possible to remove single variables by clicking the "Remove" button to the right in the row, and we can completely reset the window back to its state shown in figure 4.10 by clicking the "Reset" button in the top besides the "Add Variables" button.

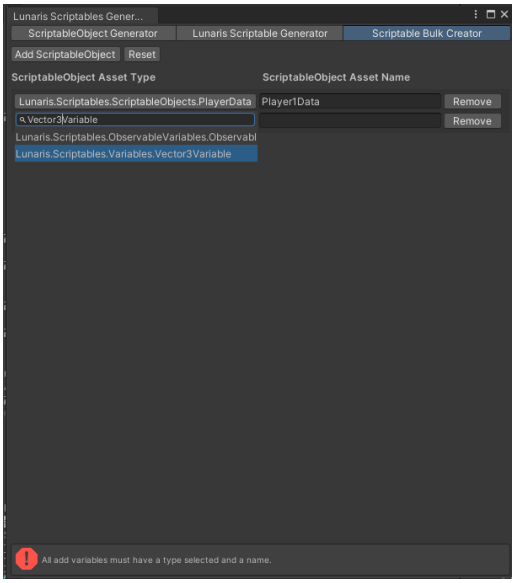


Figure 4.12

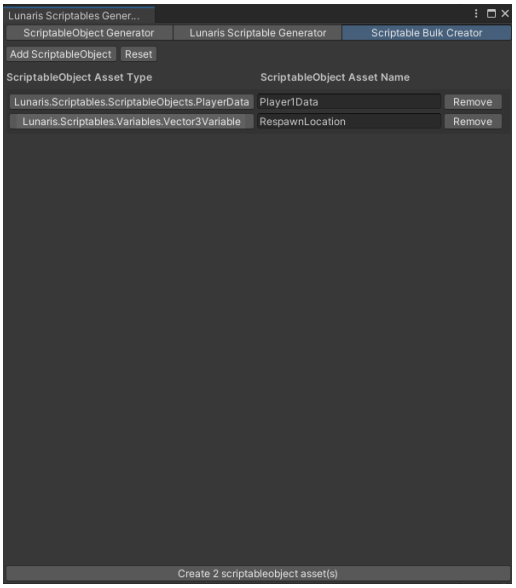


Figure 4.13

We can now create our scriptableobjects assets by clicking the "Create X scriptableobject asset(s)", which should create the files shown in figure 4.14.

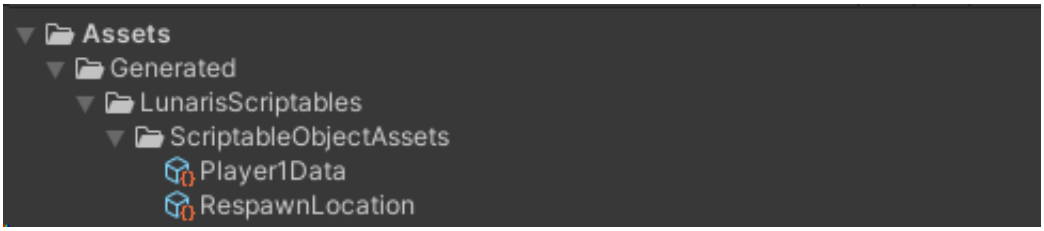


Figure 4.14

Important: Please note that the generators **DOES NOT** remove illegal characters from the file names such as comma (,) question marks (?) or curly brackets ({}), but may instead throw errors.

Important: Please note that when generating scriptableobject assets with the same path of an already existing file with the same name it will be overridden.

5 Settings

Lunaris' Scriptables come with multiple settings to customize the editors and usage to better suit individuals need. All settings are available in "Edit→Project Settings..." and then in the settings window navigate to "Lunaris→Scriptables".

5.1 Automatic Create Scriptableobjects

When on ScriptableObjects is automatically created when using the "Create" button in the inspector of a Lunaris' Scriptables (Variables, ObservableVariables, Events, Collections and ObservableCollections). If off the user is presented with a save file dialog. Default this option is on.

5.2 Scriptable Asset Path

This option is only vissible when "Automatic Create Scriptableobjects" is on. This setting is a string with a relative path where auto assets created from the inspector is put. Default value is "Assets/ScriptableObjectAssets/Generated".

5.3 Prompt before overwriting files

When on a dialog popup will be displayed asking if you want to overwrite existing files with the same name of the ones being created. This dialog is only shown once and the choice is then reflected for all the files. Default this option is on.

5.4 Prompt before overwriting for all files

This option is only vissible when "Prompt before overwriting files" is on. When on a dialog popup will be displayed for every file already existing with the same name of one being created, asking if you want to overwrite. Default this option is on.

5.5 Code Generator

A Reference to a code generator asset which Lunaris' scriptables is using to create code. The default reference is located at "Assets/Lunaris/Scriptables/Generator".

5.6 Embed Lunaris Scriptables in Inspector

If on, all objects inheriting from LunarisScriptableObject has its editor shown in the inspector such that it can be edited on the referencing script rather than needed to be selected. Default this option is on.

5.7 Embed ScriptableObjects in Inspector

This option is only vissible when "Embed Lunaris Scriptables in Inspector" is on. If on, all objects inheriting from ScriptableObject has its editor shown in the inspector such that it can be edited on the referencing script rather than selecting the scriptable object asset. Default this option is off.

5.8 Show Create/Load on Lunaris Scriptables

If on, all objects inheriting from LunarisScriptableObject will display a create and if possible a load button besides its property shown in the inspector. Default this option is on.

5.9 Show Create/Load on ScriptableObjects

This option is only vissible when "Show Create/Load on Lunaris Scriptables" is on. If on, all objects inheriting from ScriptableObject will display a create

and if possible a load button besides its property shown in the inspector. Default this option is off.

5.10 Only Show LunarisScriptableObjects in Bulk Creator

If on, Only objects inheriting from LunarisScriptableObject is shown in the Scriptable bulk creator. Default this option is on.

5.11 Use Filters for Bulk Creator

This option is only vissible when "Only Show LunarisScriptableObjects in Bulk Creator" is off. When this option is on most/all built-in objects inheriting from ScriptableObject is filtered out of the bulk creator allowing only user created types to show up. Default this option is on, but is ignored/hidden when "Only Show LunarisScriptableObjects in Bulk Creator" is on.

5.12 Use Lunaris' Object Picker for abstract/generic scriptables

This option indicate whether the property drawers of abstract/generic scriptables should open unity's built-inobject picker (off), or a custom made one that better support generic types and shows derived types of the fields (on). This option is by default on.

5.13 Automatically create scriptable on single match

When this option is on, scriptableobjects will automatically be created if it is the only class matching the field type when clicking the create button. If the option is off a dropdown is always shown.

5.14 Curly Bracket Style

This option indicate whether the code generator should place curly brackets on a new line following .net/Allman standards or on the same line following

K&R standards. Default value is new line.

5.15 Indent Style

This option indicate how the code generator should indent code, using spaces or tabs. Default value is spaces.

5.16 Indent Amount

This option is only vissible when "Indent Style" is set to spaces. This option incidate how many spaces the code generator should indent code. Default value is 4.

5.17 Generator Settings

The following settings are specific for how code is generated, styles and naming conventions of output files created. Most of these settings are used with different variable across the different types that can be generated eg. "Variable Path" which indicate this is the path for variables specific.

5.17.1 Paths

Paths is a relative path to a folder/directory which will be the output path for the generated files. Please note that on some devices and operating systems you may need to have administrator privileges for the generators to be allowed to create folders. By default all paths has the value "Assets/Generated/Lunaris/Scriptables/{BaseType Name}" where "{BaseType Name}" could be "variables" for the variable type.

5.17.2 Namespaces

Namespaces lets you specify under what namespace different types are generated. By default all namespaces has the value "Lunaris.Scriptables.{BaseType Name}" where "{BaseType Name}" could be "variables" for the variable type.

5.17.3 Prefixes

Prefixes is used to add characters/text in front of the generated classes and files. Per standard the generated files and classes will have the following names as output {Perfix}{TypeName}{Surfix}. Eg. an observablevariable of type string would be called "ObservableStringVariable". Please note that the generators **DOES NOT** remove illegal characters from the files or class name such as dot (.) comma (,) question marks (?) or curly brackets ({}), but instead throw errors. For standard values please refer to the settings windows.

5.17.4 Surfixes

Surfixes is used to add characters/text in behind of the generated classes and files. Per standard the generated files and classes will have the following names as output {Perfix}{TypeName}{Surfix}. Eg. an observablevariable of type string would be called "ObservableStringVariable". Please note that the generators **DOES NOT** remove illegal characters from the files or class name such as dot (.) comma (,) question marks (?) or curly brackets ({}), but instead throw errors. For standard values please refer to the settings windows.

5.17.5 Asset Menu Path

Asset menu path is a path to where the scriptableobject will be in the Assets menu/right click menu. For more information please see `https://docs.unity3d.com/ScriptReference/CreateAssetMenuAttribute.html`. By default all asset menu paths has the value "Lunaris Scriptables/{BaseTypeName}.

5.17.6 Component Menu Path

Component menu path is a path to where monobehaviors will be in the add component menu. For more information please see `https://docs.unity3d.com/ScriptReference/AddComponentMenu.html`. By default all component menu paths has the value "Lunaris Scriptables/{BaseTypeName}.

6 Edtitor Extensions

With Lunaris' Scriptables comes some additions to the unity editor. These additions will be introduced and explained in this chapter. All examples shown in the following sections is using scripts/data created from chapter 4.2 Lunaris Scriptable Generator.

6.1 Create and Load Button for Lunaris Scriptable Objects

The first thing that you may notice is that all references to Lunaris' Scriptables in script will now show a create or a create and a load button next to them if the setting 5.8 Show Create/Load on Lunaris Scriptables is on. Further more these buttons will also be present on normal scriptableobecjts if the setting 5.8 Show Create/Load on Lunaris Scriptables is on.

The following example is presuming that the code in listing 6.1 is added to a monobehavior that is attached to a gameobject or a scriptableobject.

```
1     [SerializeField]
2     PlayerData _playerData;
```

Listing 6.1: The code lines that is used in the following examples for creating/loading assets.

Upon adding the variable to a script it should be visible in the inspector like other variables, however it should have a create button to the right like shown in Figure 6.1.

Figure 6.1: The editor of a LunarisScriptableObject when adding the code in listing 6.1 to a monobehavior that is attached to a gameobject or a scriptableobject.

Clicking the create button will automatically create a scriptableobject asset with the same name as the variable/property name (or the variable/property name of the event listener), or you will see save file dialog depending on your settings. The newly created asset will be saved to the path specified in the Lunaris' Scriptables settings or in the save dialog, and automatically be assigned to the property as can be seen in Figure 6.2.

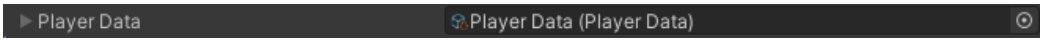


Figure 6.2: The editor of a LunarisScriptableObject after clicking the "Create" button or "load" button, if the load button is available.

When ever a variable/property with a name and type matching an existing scriptable object asset file a load button will appear as seen in Figure 6.3. Clicking this load button will add a reference to the existing asset.

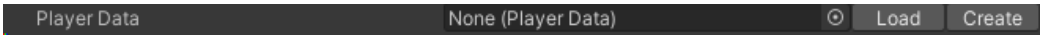


Figure 6.3: The editor of a LunarisScriptableObject with both create and load button visible.

Important: Please note that it is possible to overwrite existing files when creating new scriptable objects with the create button. This may course issues if it is a different type from the original type if it is referenced from anywhere. A prompt will always appear asking before overwriting. In this prompt it is possible to select load as well, but this will only work if the file about to be overridden is the same type as the type it is trying to be loaded into.

6.2 Embeding Lunaris Scriptable Objects and Scriptable Objects

By default Lunaris' Scriptables is able to embed all its types mentioned in chapter 3 Lunaris' Scriptables Types, and all types inheriting from LunarisScriptableObject such as those generated by using the ScriptableObject Generator, that is discussed how works in section 4.1 ScriptableObject Generator. It is further possible to embed all scriptable objects in a similar way if the option is enabled in settings.

The code in listing 6.2 is used to produce the inspector in Figure 6.4.

```
1  public class Tester : MonoBehaviour
2  {
3      [SerializeField]
4      PlayerData _playerData;
5      [SerializeField]
6      Vector3Variable vector3Val;
7      [SerializeField]
8      ObservableStringVariable observableStringRef;
9      [SerializeField]
10     StringChangeEvent stringGameEvent;
11     [SerializeField]
12     StringChangedEvent stringChangedGameEvent;
13     [SerializeField]
14     StringCollectionChangedEvent
           stringCollectionGameEvent;
15 }
```

Listing 6.2: A code snippet which produce the editor showen in Figure 6.4.

In Figure 6.4 it can be seen how it is possible to expand and collapse each embedded editor, and when it is expanded, the values of each reference can be edited and is remembered for all variables. Remember that the values/data entered in the event editors is not persistent and is lost when the object is deselected.

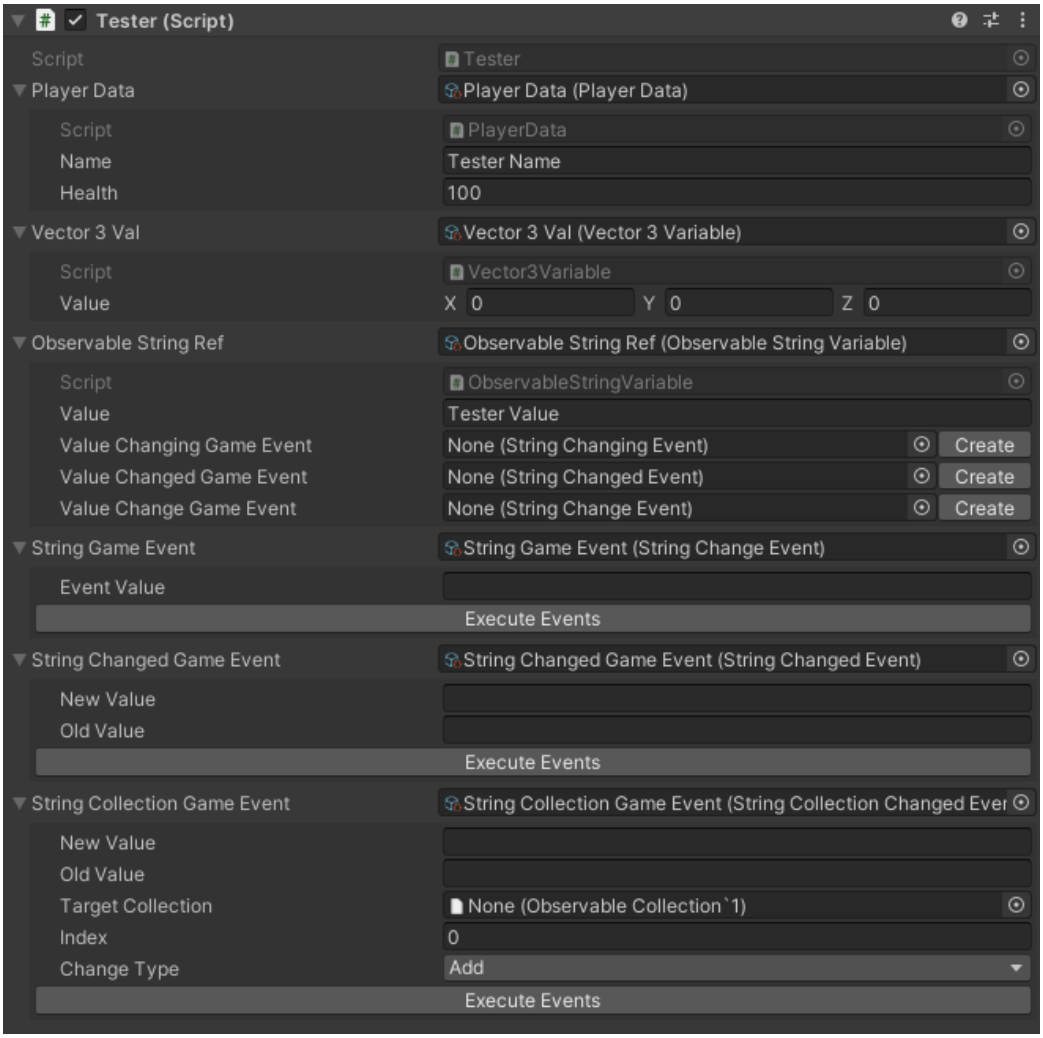


Figure 6.4

Important: Due to how embedded editors work in unity, having a lot of them expanded may cause some lag or stuttering when using the inspector.

6.3 Event Listeners and Automatic Listener Callbacks

The last major addition Lunaris' Scriptables is the ability for event listeners to automatically create callback methods in script which can be called by an event.

To automatically create callbacks, first add the code from listing 6.3 to

a monobehavior attached to a gameobject or a scriptableobject, or alternatively add an EventListenerBehavior component to a gameobject. The example on Figure 6.5 both the code from listing 6.3 is added to a script and a EventListenerBehavior is added to a gameobject.

```
1 [SerializeField]
2 StringEventListener _stringListener;
```

Listing 6.3: The code which produce the editor/inspector of the tester script shown in Figure 6.5.

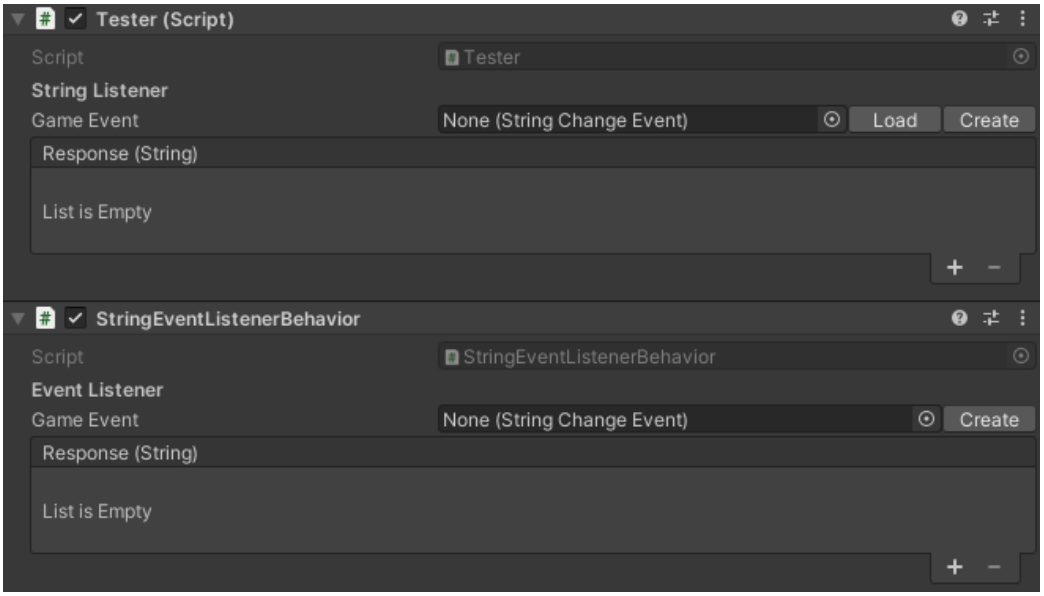


Figure 6.5: Example of an inspector with an eventlistener added to a script and an EventListenerBehavior.

Now add an event to each event listener of a matching type. Remember that you can use the "Create" button to fast and automatically create the event. You should have an event attached to the listeners and have something similar to what can be seen in Figure 6.6.

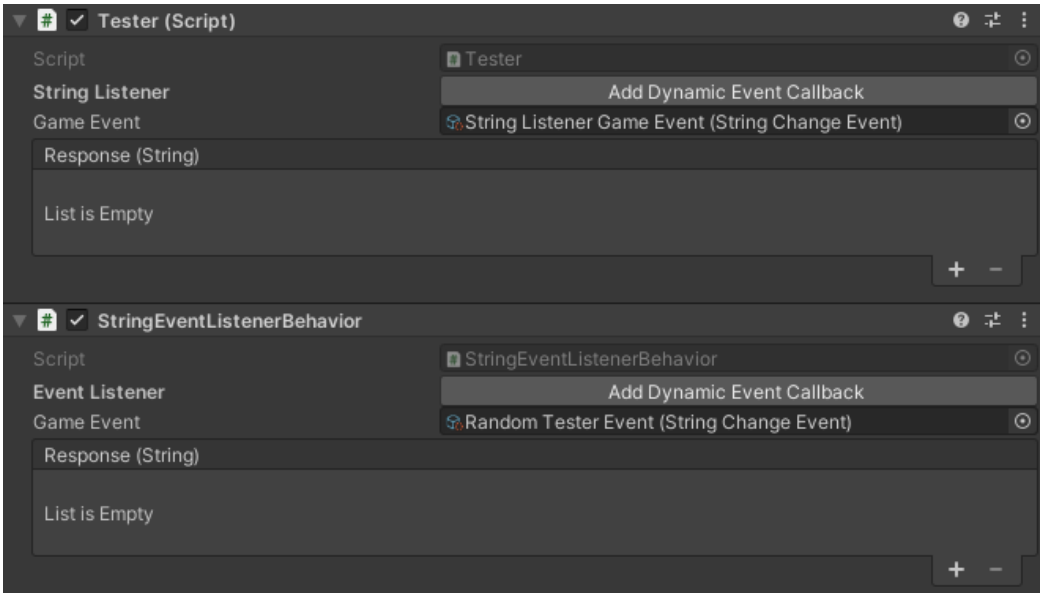


Figure 6.6: Example of an inspector with an eventlistener added to a script and an EventListenerBehavior with events attached to them.

When the an event is attached to the game event property off an event listener you will notice a button will appear which says "Add Dynamic Event Callback". When you click this button and there is only one monobehavior attached to the gameobject (besides event listener behaviors) a callback method with the name of the event will be created in the script. Clicking the "Add Dynamic Event Callback" button on both the String Listener and the Event Listener in the EventListenerBehavior will generate the code in listing 6.4, and the event listeners will automatically also add these to the response unity event list as can be seen in Figure 6.7.

```
1    public void StringListenerGameEvent(string args)
2    {
3        Debug.LogError("The method '
             StringListenerGameEvent' is not implemented.")
             ;
4    }
5
6    public void RandomTesterEvent(string args)
7    {
8        Debug.LogError("The method 'RandomTesterEvent' is
             not implemented.");
9    }
```

Listing 6.4: The code/methods produces from clicking the "Add Dynmamic Callback" button.

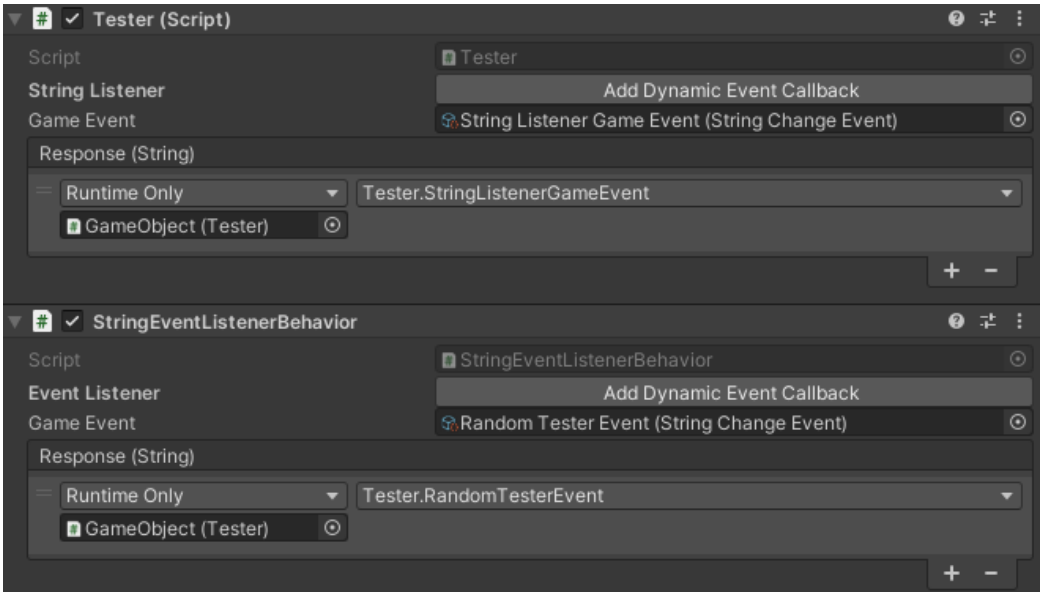


Figure 6.7: Example of the inspector after having clicked the "Add Dynamic Event Callback" for both event listeners.

Lastly if there is multiple monobehavior attached to the gameobject (besides event listener behaviors) a dropdown will appear at the button with all the monobehaviors on the object which can be seen in Figure 6.8. Choosing a script will generate the code in listing 6.4 in the selected script or add an existing method to the response unity event list.

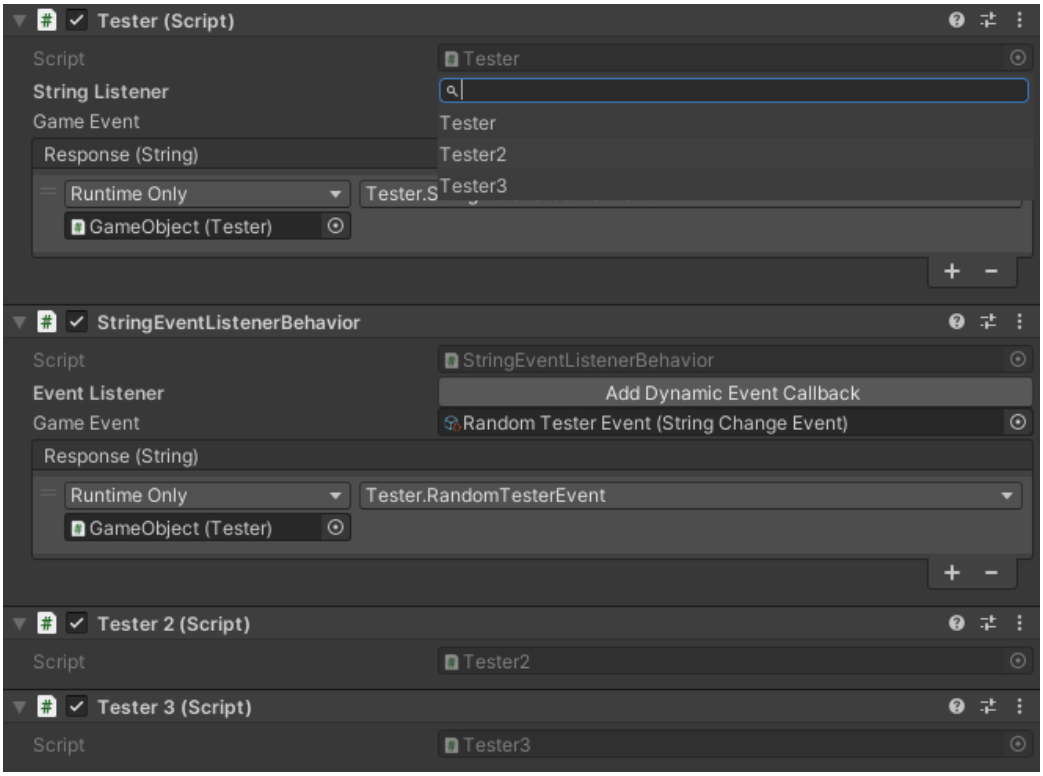


Figure 6.8: Example of the dropdown menu to select the script that should have the callback method generated.

Important: Generating the callback method may take some time and after it have been added all scripts will be reloaded/recompiled to make sure everything was added correctly.

Important: If a method already exist with the correct name and parameters, matching the event, new code is not generated but instead that method is used and added to the response unity event list instead.

Important: If there is currently script errors present in the project, the output of the callback generator may cause further errors.

7 Custom Code Generators

Lunaris' Scriptables uses a code generator to generate the code and classes, which is generated by using the tools included. It is possible to create your own generator(s) by inheriting from the abstract class "LunarisScriptableCodeGenerator" if you need to do something very specific or do not wish to use the included generator Lunaris' Scriptables comes with. All code generators inherits from ScriptableObject and will need to be created as an asset and assigned to the setting "Code Generator" which is explained in section 5.5 Code Generator.

In listing 7.1 we've implemented the LunarisScriptableCodeGenerator abstract class with all methods so it is ready to have code added to it.

```
1  [CreateAssetMenu("Code Generators/TestGenerator")]
2  public class TestGenerator :
       LunarisScriptableCodeGenerator
3  {
4      public override void
           CreateCollectionEventListenerBehaviors(Type type)
5      {
6      }
7
8      public override void CreateCollectionEventListeners(
           Type type)
9      {
10     }
11
12     public override void CreateCollectionGameEvent(Type
           type)
13     {
14     }
15
16     public override void CreateEventListenerBehaviors(
           Type type)
```

```
17      {
18      }
19
20      public override void CreateEventListeners(Type type)
21      {
22      }
23
24      public override void CreateGameEvent(Type type)
25      {
26      }
27
28      public override void CreateScriptableCollection(Type
            type)
29      {
30      }
31
32      public override void CreateScriptableObjectType(
            ScriptableObjectCratorValues creatorValues)
33      {
34      }
35
36      public override void
            CreateScriptableObservableCollection(Type type)
37      {
38      }
39
40      public override void
            CreateScriptableObservableVariable(Type type)
41      {
42      }
43
44      public override void CreateScriptableVariable(Type
            type)
45      {
46      }
47  }
```

Listing 7.1: Example of a new code without any code.

Important: The CreateGameEvent, CreateCollectionGameEvent, CreateEventListeners and CreateCollectionEventListeners must each make all scripts for both Changing, Changed and change event as these are not di-

vided into separated methods.

Important: When using the generator to create child/inherited classes of the base Lunaris' Scriptables' types it is important the generator creates code which overrides certain methods to avoid compile errors. Please refer to the base classes to find the abstract members.

Out of the box Lunaris' Scriptables comes with a code generator, which class is called "LunarisScriptableCodeDomCodeGenerator", that is using codedom. You can learn more about codedom here: `https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/dynamic-source-code-generation-and-compilation`. This generator can be extended as well to customize the resulting scripts.

The following example will show you two ways of changing the base class of observablevariables generated by the Lunaris' Scriptables' code generator. This example will also add a field to the output classes for demonstration purposes.

Important: All code generators using codedom elements MUST be placed in a folder called "Editor" to work properly.

First lets create our new base class called "ScriptableObservableVariableExtended" and add the code in listing 7.2

```
1  public abstract class
       ScriptableObservableVariableExtended<T> :
       ScriptableObservableVariable<T>
2  {
3      [SerializeField()]
4      long _id;
5
6      public long Id
7      {
8          get
9          {
10             return _id;
11         }
12         set
13         {
14             _id = value;
15         }
16     }
17 }
```

Listing 7.2: Example class for extending the base class ScriptableObservableVariable.

Next we also need to create a code generator by extending the LunarisScriptableCodeDomCodeGenerator class, which already contains all methods for generating code and creating the final files. In this example we call the new generator class "LunarisScriptableCodeDomCodeGeneratorExtended" as seen in listing 7.3. Note that CreateAssetMenu is added such that it can be created as a scriptable asset later.

```
1 [CreateAssetMenu(menuName = "CodeGenerators/
      LunarisScriptableCodeDomCodeGeneratorExtended")]
2 public class
      LunarisScriptableCodeDomCodeGeneratorExtended :
      LunarisScriptableCodeDomCodeGenerator
3 {
4 }
```

Listing 7.3: Example of a new generator class.

Now for the first example we wanna override the method "CreateScriptableObservableVariable" and change some of the generated properties of the resulting codedom element, as well as adding out own new one. Lastly, we use the already existing method of saving the scripts. The LunarisScriptableCodeDomCodeGenerator class generates all CodeCompileUnits with two items in the "Namespaces" property, the first one is references/usings and the second is the object/class it self. The reason it is done this way is due to how the final code is formatted when using codedom. If you need to add references to the output class you will need to access it like this: "CodeCompileUnitVar.Namespaces[0].Imports", and if you need to make changes to the output object it should be accessed like this: "CodeCompileUnitVar.Namespaces[1].Types" The code shown in listing 7.4 changes the base class of and output object to the class in listing 7.2, and adds a string field with a SerializeField attribute called "_guiId".

```
1 [CreateAssetMenu(menuName = "CodeGenerators/
      LunarisScriptableCodeDomCodeGeneratorExtended")]
2 public class
      LunarisScriptableCodeDomCodeGeneratorExtended :
      LunarisScriptableCodeDomCodeGenerator
3 {
4     public override void
          CreateScriptableObservableVariable(Type type)
```

```
5     {
6         CodeCompileUnit cu = new CodeCompileUnit();
7         GenerateScriptableObservableVariable(type, cu);
8         cu.Namespaces[1].Types[0].BaseTypes[0] = new
              CodeTypeReference("
              ScriptableObservableVariableExtended<" + type.
              Name + ">");
9         cu.Namespaces[1].Types[0].Members.Add(new
              CodeMemberField(typeof(string), "_guiID") {
              CustomAttributes = new
              CodeAttributeDeclarationCollection() { new
              CodeAttributeDeclaration("SerializeField") }
              });
10        SaveGeneratedFile(
              GenerateCodeStringFromComepileUnit(cu),
              LunarisScriptablesSettings.instance.
              ObservableVariablePath + "/" +
              LunarisScriptablesSettings.instance.
              ObservableVariableNamePrefix +
              CheckPrimitivesTypeName(type) +
              LunarisScriptablesSettings.instance.
              ObservableVariableNameSurfix + ".cs");
11    }
12 }
```

Listing 7.4: Example of a method which changes the base class of an obsevable variable.

The second option is very similar, but in the LunarisScriptableCodeDomCodeGenerator there is also a protercted method called "GenerateScriptableObservableVariable" which can be overridden. Unlike the first example you don't need to save the file yourself when using this method, but instead its base first and then edit the object as shown in listing 7.5 and should produce a identical result to the first example.

```
1  [CreateAssetMenu(menuName = "CodeGenerators/
       LunarisScriptableCodeDomCodeGeneratorExtended")]
2  public class
       LunarisScriptableCodeDomCodeGeneratorExtended :
       LunarisScriptableCodeDomCodeGenerator
3  {
4      protected override void
```

```
        GenerateScriptableObservableVariable(Type type,
        CodeCompileUnit cu)
5   {
6       base.GenerateScriptableObservableVariable(type,
            cu);
7       cu.Namespaces[1].Types[0].BaseTypes[0] = new
            CodeTypeReference("
            ScriptableObservableVariableExtended<" + type.
            Name + ">");
8       cu.Namespaces[1].Types[0].Members.Add(new
            CodeMemberField(typeof(string), "_guiID") {
            CustomAttributes = new
            CodeAttributeDeclarationCollection() { new
            CodeAttributeDeclaration("SerializeField") }
            });
9   }
10 }
```

Listing 7.5: Example of a method which changes the base class of an obsevable variable.

You should now be able to go to the Create menu in assets and see that the a menu called CodeGenerators is now available, and inside there you should see the "LunarisScriptableCodeDomCodeGeneratorExtended". Create this object, go to settings and assign the new generator to the code generator property field. Going to "Lunaris Scriptables Generator" under the tab "Lunaris Scriptable Generator" and generating an ObservableVariable for the "System.String" type should now generate the code in listing 7.6

```
1  [CreateAssetMenu(menuName="Lunaris Scriptables/
       Observable Variables/ObservableStringVariable")]
2  public class ObservableStringVariable :
       ScriptableObservableVariableExtended<String>
3  {
4      [SerializeField()]
5      private StringChangingEvent _valueChangingGameEvent;
6      [SerializeField()]
7      private StringChangedEvent _valueChangedGameEvent;
8      [SerializeField()]
9      private StringChangeEvent _valueChangeGameEvent;
10     [SerializeField()]
11     private string _guiID;
```

```
12      public override BaseGameEvent<string>
            ValueChangeGameEvent
13      {
14          get
15          {
16              return _valueChangeGameEvent;
17          }
18      }
19      public override BaseGameEvent<ValueChangingEventArgs
            <string>> ValueChangingGameEvent
20      {
21          get
22          {
23              return _valueChangingGameEvent;
24          }
25      }
26      public override BaseGameEvent<ValueChangedEventArgs<
            string>> ValueChangedGameEvent
27      {
28          get
29          {
30              return _valueChangedGameEvent;
31          }
32      }
33  }
```

Listing 7.6: The output code of the generator made from the code in listing 7.4 or listing 7.5.

8 Future Works

Lunaris' Scriptables have more features planned to be added in the future and we also plan on listening closely to user feedback to provide the best tools possible for our end-users. These features is mentioned in the following sections.

8.1 Planning Features

- Event reference finder - a way to see all publishers/subscribers/listeners/references to an event to fast have a overview.
- Missing reference finder and fixer.
- Observable UI which reflect changes to variables.
- Scriptable Coroutines - may become separate package.
- Rewrite editors using UI elements when stable.
- Odin compatibility path.

8.2 Known Issues

LTS 2020.3.X sometimes causes a warning "Importer(NativeFormatImporter) generated inconsistent result for asset" when creating scriptable assets from the inspector; this is most likely some left over fixes from a bug (1317257) in 2020.3.0 which have issues with SerializedProperties after inspector refresh.

Using raw generics, eg. BaseGameEvent<T> to referencing objects will in some versions of unity cause the object picker to either display no items or items which is not of the correct generic type, but rather inherits from the raw generic type. This seems to be a error in unity, and is discussed here: `https://forum.unity.com/threads/generic-scriptable-o`

bject-fields.790763/ and here https://issuetracker.unity3d.com/issues/assets-are-not-listed-in-the-object-picker-field-when-scriptableobject-is-generic. You can still easily drag and drop the correct items onto the displayed property field if the type matches. A work around is in development from unity, but only seems compatible with unity 2022.2 and newer https://forum.unity.com/threads/why-do-objectpickers-display-objects-of-incompatible-type.1290533/

When using converters it is posible to crash unity with a stackoverflow if there is a loop, that is if Converter A coverts type X to Y and then converter A uses a converter B to convert type Y back to X. Doing this will often result in a crash and data may be lost. If unity saves a change where the crash persist, please use the debug inspector/editor to remove the reference coursing issues. A work around is planned but stackoverflows can often be hard to find viable solutions to, which also keeps the desired functionality.

Currently there's a minor compatibility issue with odin inspector/serializer that prevents embedded editors to be drawn. You can fix this by firstly go to "Tool→Odin Inspector→Preferences". A new window should apear, in the side click "Editor Types" and search for "Lunaris" under the "User Types" foldout and remove the checkmark in the field called "Lunaris". Please note that the name may vary if you've changed the namespace of the generated types. Secondly you can use the "InlineEditor" attributes to draw the editor of any Lunaris' Scriptables.

Overwriting an existing file causes it to be unable to be deleted until an asset database refresh. - This is done from code when overwriting but seems to be ignored.