Input Sequence

A Sub-module for Embedded Input Module providing classes for sequential input tracking.

Optional features and classes require Embedded Input Module.

Input Sequence is a C# file providing three new base classes for tracking player inputs and determining if a sequence of inputs have successfully been pressed in order. Additional sub-classes are also provided for use with Embedded Input Module so that developers can immediately integrate Input Sequences into their Unity Projects, however, all core classes are written to be entirely independent. Input Sequence can be completely decoupled from Embedded Input Module as a standalone module that can integrate with preferred input handlers like InputSystem, the legacy UnityEngine.Input class and Rewired.

Scroll to the end of this document to see how to create custom sequence classes using the core classes.

Current Release: Version 1.0

Overview

Full C# XML Documentation

All classes and members are documented within C#, using inheritance to trickle down to included and new sub-classes.

Class Interchangeability

AbstractInputSequence, InputSequence<T>, AbstractSequenceSet and any sub-classes can be used interchangeably when defining a Sequence or Set.

Only Indexer Members and Methods are excluded from Sets, due to technical restrictions and performance concerns.

AbstractInputSequence

An abstract class containing all vital members necessary to create and manage sequences. Does not directly contain the array of sequence inputs.

This class is used for creating and managing collections of sequences in arrays and lists.

InputSequence<T>

The base class for all Input Sequences utilising any Input Handler. This class contains an array of type **T** that represents all inputs in the sequence.

Not suitable for collections of sequences in arrays or lists.

AbstractSequenceSet

An abstract class containing all vital members necessary to create and manage a set of sequences in an array. A set can encompass any inheritor of **AbstractInputSequence**, so long as all core methods are inherited or overridden.

Embedded Input Module Classes (Optional)

Two classes inheriting from InputSequence and one from AbstractSequenceSet that are completely optional and can be disabled at the top of the script.

GamepadSequence

Takes a sequence of EmbeddedInputModule.GamepadControl values and tracks player inputs until completion.

KeyboardSequence

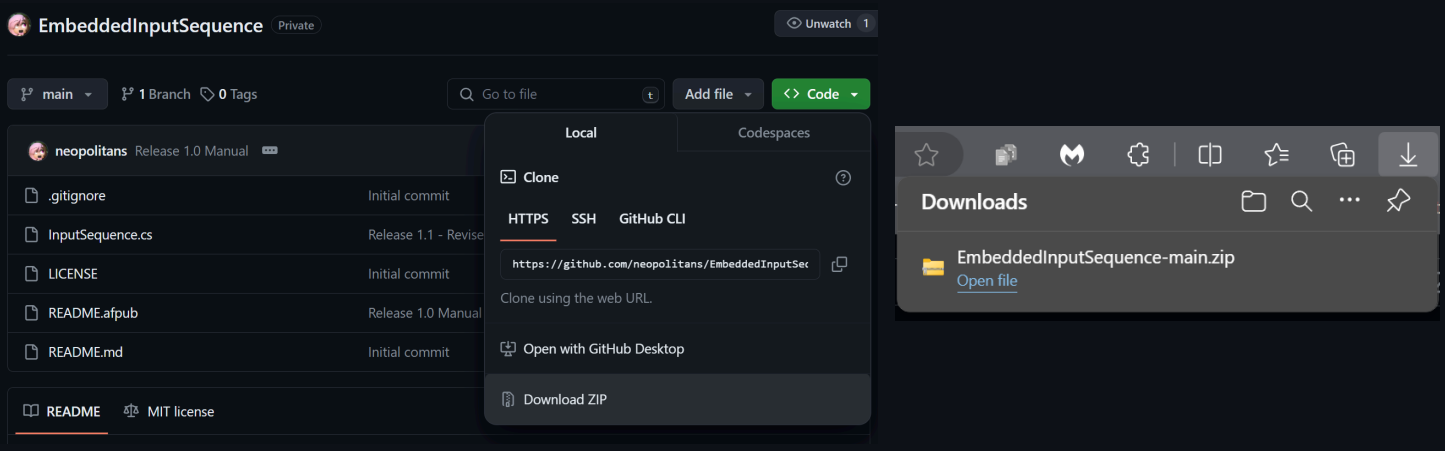
Takes a sequence of UnityEngine.KeyCode values and tracks player inputs until completion. Uses EmbeddedInputModule's methods instead UnityEngine.Input's methods for more consistent Gamepad and Mouse KeyCodes.

MultiplatformSequence

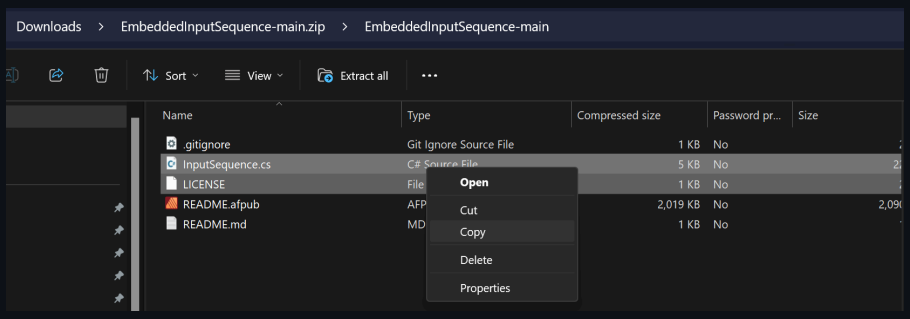
Takes any set of AbstractInputSequences and manages all of them.

Installation

[**Step 1**] Download the GitHub repository, which will be a .zip file of all files and folders contained within.

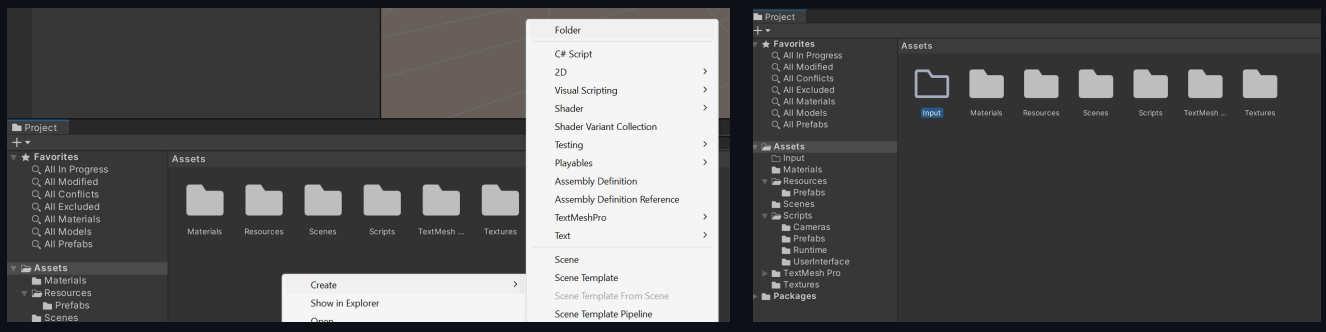


[**Step 2**] Open (or Extract) the .zip file, go into the folder then copy InputSequence.cs and the LICENSE file.

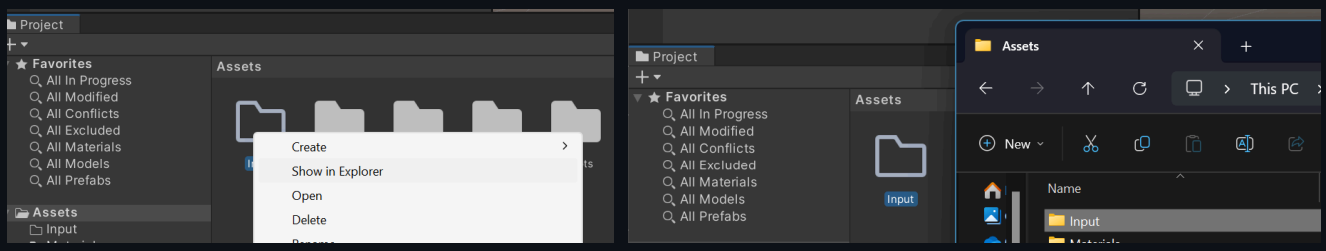


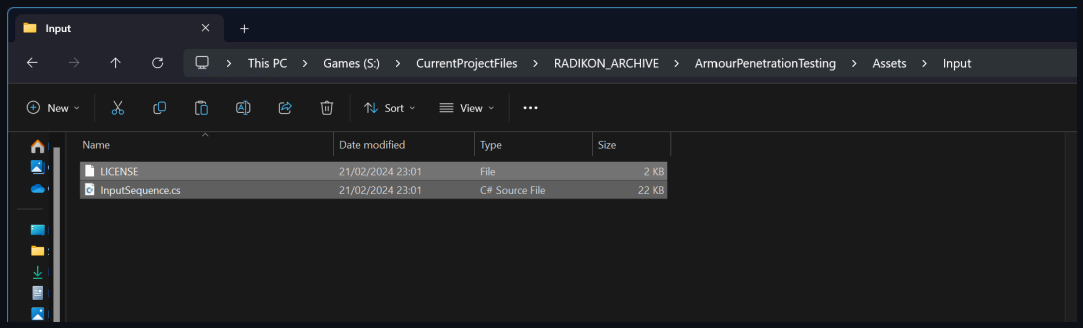
[**Step 3**] Go to the "Project" tab in Unity and create a new folder called "**Input**"

Both files **can** be copied directly into Assets or any folder with a preferred name, however some folders can be created with names which will not get included when you build your game. A detailed list can be found here.



[**Step 4**] Open the new folder by highlighting it, clicking Show in Explorer and paste the copied files into the folder.





Implementing Sequences and Sets

Input Sequences only need three steps to be implemented into a project and are interchangeable with each other. Examples provided utilise the classes that are provided for Embedded Input Module. This guide should be compatible for most, if not all, sub-classes that directly implement the same behaviour.

For any **AbstractInputSequence** or **AbstractSequenceSet**, only Step 1 changes slightly.

[**Step 1**] Define the Sequence or Set at the top of a script

Using a Single Sequence:

```
GamepadSequence SeeYouNextGameSeq = new GamepadSequence(
    GamepadControl.DPadUp, GamepadControl.DPadUp,
    GamepadControl.DPadDown, GamepadControl.DPadDown,
    GamepadControl.DPadLeft, GamepadControl.DPadRight,
    GamepadControl.DPadLeft, GamepadControl.DPadRight);
```

Initial Testing Input Sequence from Radikon War: Lone Warrior

Using a Sequence Set:

```
MultiplatformSequence SeeYouNextGameSeq = new MultiplatformSequence(
    new GamepadSequence(
        GamepadControl.DPadUp, GamepadControl.DPadUp,
        GamepadControl.DPadDown, GamepadControl.DPadDown,
        GamepadControl.DPadLeft, GamepadControl.DPadRight,
        GamepadControl.DPadLeft, GamepadControl.DPadRight),
    new KeyboardSequence(
        KeyCode.UpArrow, KeyCode.UpArrow,
        KeyCode.DownArrow, KeyCode.DownArrow,
        KeyCode.LeftArrow, KeyCode.RightArrow,
        KeyCode.LeftArrow, KeyCode.RightArrow
        )
    );
```

Final Multi-Platform Sequence Set from Radikon War: Lone Warrior

[**Step 2**] Update the Sequence or Set in an Update Method

```
/// <summary>
/// Perform any necessary updates during the Title Screen state.
/// </summary>
1 reference
void TitleScreenUpdate()
{
    /* Other Game Code here*/
    SeeYouNextGameSeq.UpdateSequence();
}
```

Excerpt from Radikon War: Lone Warrior

[**Step 3**] Check if the Sequence or Set has been completed

```
/// <summary>
/// Perform any necessary updates during the Title Screen state.
/// </summary>
1 reference
void TitleScreenUpdate()
{
    /* Other Code Here */
    SeeYouNextGameSeq.UpdateSequence();
    if (SeeYouNextGameSeq) StartCoroutine(EasterEggSeeYouNextGame());
}
```

Excerpt from Radikon War: Lone Warrior

Developers may prefer to Update the Sequence only if the sequence isn't registered as complete. This isn't done here but is a completely viable, functional ability of Input Sequence. UpdateSequence in the provided sub-classes already checks if the sequence or set is complete and returns if so.

Any class inheriting from **AbstractInputSequence** and **AbstractSequenceSet** perform optimally when the same three-step implementation is followed. However, this does not mean other handling methods are preferred. Developers are completely free to modify any of the classes to fit preferred needs and may find benefits to doing so.

Classes

Given an overview in the introductory page, Input Sequence contains three base classes, and three optional classes that use Embedded Input Module. Almost all functionality and documentation is found in the base classes. In-game functionality is heavily dependent on the Input Handler used and must be written manually. A guide for doing this is at the end.

New Abstract Class - AbstractInputSequence

This is the core class behind almost all of Input Sequence, containing all members, properties, operators and methods required by all inheritors. Although the sequence of inputs is absent from the abstract class' members list, it is not required for most use-cases. Inherited by **InputSequence<T>** and any sub-classes, this base class allows all inheriting classes to be collected inside Arrays and Lists for management inside a Set and other purposes.

Members

	Description
AbstractInputSequence.successfulInputs	The boolean values that are set as inputs are read sequentially.
AbstractInputSequence.sequenceCompleted	Has the sequence been completed?
AbstractInputSequence.autoResetSequence	Should the sequence automatically reset if an unsuccessful input is detected?
AbstractInputSequence.current	The current input as an index value that the sequence is listening for.
AbstractInputSequence.lastInputSuccessful	Was the last input in the sequence successfully detected?

Methods

	Description
AbstractInputSequence.UpdateSequence	Update the sequence by listening to the player's input. If the next input in the sequence is detected, the input after becomes the next input.
AbstractInputSequence.SetAsComplete	Mark the sequence as completed.
AbstractInputSequence.ResetSequence	Reset the sequence. This will reset all values in the Successful Inputs array.

Operators

	Description
AbstractInputSequence.this[int]	Indexer that reads the corresponding value from the Successful Inputs array.
AbstractInputSequence.true	A custom true operator that returns if Sequence Complete is true.
AbstractInputSequence.false	A custom false operator that returns if Sequence Complete is false.

New Class - InputSequence<T> (T is any type of enum class)

Inheritance: **AbstractInputSequence**

This is an implementation wrapper for **AbstractInputSequence** that makes the abstraction work as an object class. The generic type of InputSequence is any System.Enum (**enum**) , preferably an enumerator that represents Controller, Keyboard, Mouse or other Human Interface Device input. Hopefully even if that device may be a Guitar Hero guitar or Power Glove.

All members, methods and operators are inherited from **AbstractInputSequence**.

Added Members

	Description
InputSequence<T>.sequence	The array of controls to be correctly pressed or detected in sequential order.

Overridden Methods

	Description
InputSequence<T>.ResetSequence	(Contains a default implementation of ResetSequence)
InputSequence<T>.UpdateSequence	(An empty method to make the compiler happy)

New Abstract Class - AbstractSequenceSet

The base class which contains a collection of Input Sequences and updates all at once, looking for a completed sequence. Any sequence should inherit from **AbstractInputSequence** in order to be managed by an **AbstractSequenceSet**.

Members

	Description
AbstractInputSequence.sequences	The array of **AbstractInputSequences** managed by this set.
AbstractInputSequence.sequenceCompleted	Has any sequence been completed?

Methods

	Description
AbstractInputSequence.UpdateSequence	Update all sequences in the set. If the next input in a sequence is detected, the input after becomes the next input. Sequences may reset if Auto Reset Sequence is enabled per sequence.
AbstractInputSequence.SetAsComplete	Mark all sequences and the set as complete.
AbstractInputSequence.ResetSequence	Reset the sequence. This will reset all values in the Successful Inputs array.

Operators

	Description
AbstractInputSequence.true	A custom true operator that returns if Sequence Complete is true.
AbstractInputSequence.false	A custom false operator that returns if Sequence Complete is false.

Optional Subclasses (Requires Embedded Input Module)

These classes inherit from **InputSequence<T>** and **AbstractSequenceSet** respectively. Two of which make more extensive use of Embedded Input Module's features and compatibility, however, one only utilises IAccessibilityConfigurable for bulk configuration of the Auto Reset Sequence property in sequences.

New Class - GamepadSequence

Inheritance: **InputSequence<EmbeddedInputModule.GamepadControl>, IAccessibilityConfigurable**

A subclass that tracks a sequence of **GamepadControl** inputs using Embedded Input Module's Controller state properties, such as **EmbeddedInputModule**.LeftStickButton.

All members, methods and operators are inherited from **InputSequence** and **IAccessibilityConfigurable**.

Members

	Description
GamepadSequence.IsAccessibilityEnabled	Whether the Accessibility Feature(s) of the inheriting class are enabled.

Methods

	Description
GamepadSequence.UpdateSequence	Update the sequence by listening to the player's input. If the next input in the sequence is detected, the input after becomes the next input.
GamepadSequence.SetAccessibility	Enable or Disable the Accessibility Feature(s) of the inheriting class.

Constructors

	Description
GamepadSequence(GamepadControl[])	A constructor for GamepadSequence that takes any number of GamepadControl values. (uses **params** to make it possible to have any amount of values)

New Class - KeyboardSequence

Inheritance: **InputSequence<UnityEngine.KeyCode>, IAccessibilityConfigurable**

A subclass that tracks a sequence of **KeyCode** inputs using **EmbeddedInputModule.GetKey()**, translating KeyCodes into the values held by the respective InputSystem Input Controls.

All members, methods and operators are inherited from **InputSequence** and **IAccessibilityConfigurable**.

Members

	Description
KeyboardSequence.IsAccessibilityEnabled	Whether the Accessibility Feature(s) of the inheriting class are enabled.

Methods

	Description
KeyboardSequence.UpdateSequence	Update the sequence by listening to the player's input. If the next input in the sequence is detected, the input after becomes the next input.
KeyboardSequence.SetAccessibility	Enable or Disable the Accessibility Feature(s) of the inheriting class.

Default Constructors

	Description
KeyboardSequence(KeyCode[])	A constructor for KeyboardSequence that takes any number of KeyCode values. (uses **params** to make it possible to have any amount of values)

Additional Constructors (CheckGameapdInputsForKeyboardSequence Enabled)

	Description
KeyboardSequence(GamepadControl[])	A constructor for KeyboardSequence that takes any number of GamepadControl values and translates them to the relative KeyCode values. (uses **params** to make it possible to have any amount of values)

New Class - MultiplatformSequence

Inheritance: **AbstractSequenceSet, IAccessibilityConfigurable**

A subclass that can contain and manage a set of **AbstractInputSequences**. This class does not explicitly need to be utilised as all core behaviour exists in AbstractSequenceSet. Any inheritor of **AbstractSequenceSet** only needs a unique constructor.

All members, methods and operators are inherited from **AbstractSequenceSet** and **IAccessibilityConfigurable**.

Members

	Description
MultiplatformSequence.IsAccessibilityEnabled	Whether the Accessibility Feature(s) of the inheriting class are enabled.

Methods

	Description
MultiplatformSequence.SetAccessibility	Enable or Disable the Accessibility Feature(s) of the inheriting class.

Constructors

	Description
MultiplatformSequence(AbstractInputSequence[])	A constructor for MultiplatformSequence that takes **AbstractInputSequences**. (uses **params** to make it possible to have any amount of values)

These optional classes also serve as examples which can be modified to fit any developer's needs. GamepadSequence and KeyboardSequence do heavily utilise Embedded Input Module and may require extensive modification. When it comes to utilising MultiplatformSequence independently, **IAccessibilityConfigurable** can be removed or implemented again to retain full functionality.

Configurable Settings

A few configurable settings exist inside Input Sequence. Each setting uses preprocessor directives within C# in order to switch between desired features and, where applicable, relevant restricted substitutes for features. Compared to the more in-depth settings in Embedded Input Module, there are only three settings for Input Sequence.

Enabling or Disabling Settings

To enable or disable each setting, open up the InputSequence.cs script from where you have it stored in your Unity Project and navigate to Line 21, which is the start of the header for the Settings section.

Within the section is a list of settings, each with their own description. The difference between enabled and disabled settings are in the colour of the text at the start of the line.

```
//#define EIM_OPTMOD_InputSequence_DecoupleFromEmbeddedInputModule          // Masks all EmbeddedInputModule Dependent References and Classes.
                                                                            // Provided to make using AbstractInputSequence<T> with any preferred input
                                                                            // handler easier.

 #define EIM_OPTMOD_InputSequence_CheckGamepadInputsForKeyboardSequence     // Check gamepad inputs for each GamepadControl value in KeyboardSequence when
                                                                            // a non-sequential input is detected during UpdateSequence().
```

To **Enable** a disabled setting, remove the two forward-slashes preceding the corresponding #define.
To **Disable** an enabled setting, add two forward-slashes preceding the corresponding #define.

List of Settings

EIM_OPTMOD_InputSequence_DisableDebugMessages

Disables the debug messages for successful inputs, unsuccessful inputs, sequence completion and the full sequence state.

It is advised to disable this setting when debugging and modifying classes that utilise Embedded Input Module to ensure proper functionality.

Default: Enabled
Location: Line 25

EIM_OPTMOD_InputSequence_DecoupleFromEmbeddedInputModule

Masks **GamepadSequence**, **KeyboardSequence** and **MultiplatformSequence** from inside the script.

If developers seek to use their own input handler or otherwise do not seek to use Embedded Input Module over Unity's solutions, it is advised this setting is Enabled. Developers will not have to remove any extra code with this setting enabled and may move over to Embedded Input Module later if desired.

Default: Disabled
Location: Line 29

EIM_OPTMOD_InputSequence_CheckGamepadInputsForKeyboardSequence

Enables additional checks within KeyboardSequence that go through all GamepadControl values to detect if any Gamepad Controls were detected during a call to UpdateSequence. As KeyboardSequence only checks if the AnyKeyDown property is true, this provides a more extensive check.

Also adds an additional constructor to **KeyboardSequence** that can take **GamepadControl** values.

Default: Disabled (Don't bother changing this if DecoupleFromEmbeddedInputModule is enabled.)
Location: Line 33

Creating a Custom Input Sequence

For Developers using custom input handlers, using legacy UnityEngine.Input as an example

Developers may have preferred input handlers they are familiar with. Modules designed for Embedded Input Module seek to, where possible, be fully compatible with more contemporary and off-the-shelf solutions to handling game input. However, to demonstrate the flexibility of Input Sequence, this section demonstrates the requirements of a custom input handler and how to create a custom sequence class that uses the legacy **UnityEngine.Input** class.

A Custom Input Handler must provide the following for the Controller, Keyboard, Mouse or Peripheral intended for a class:

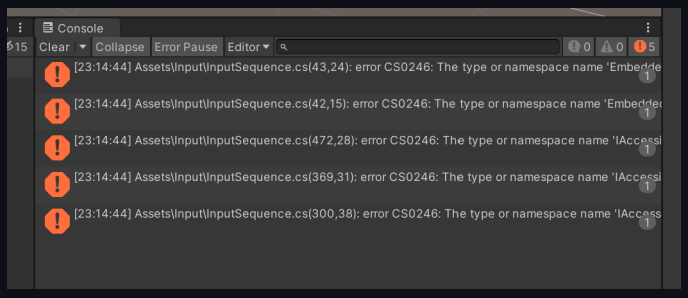
- An enumerator (that inherits from or directly is a System.Enum type)
- A method or property to access whether an input is pressed this frame (or held)

This comprehensive guide contains all core step so beginners can follow along.

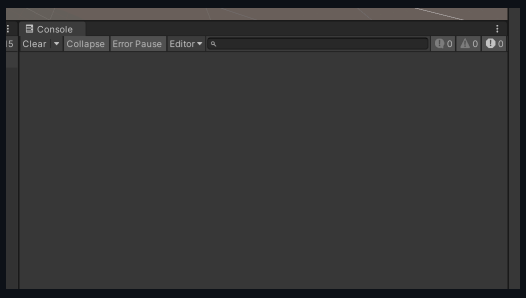
Sorry for any inconvenience in advance.

[**Step 0**] (Optional) Enable the Decoupling from Embedded Input Module setting.

If there are errors like this, then decoupling is necessary.



There should only be messages related to your project.

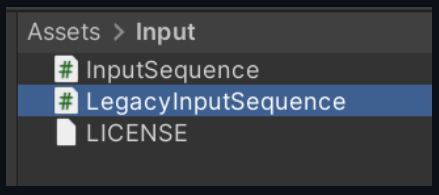



```
//-------------------------------------------------------------------------------------------------------------------------------
//                                                  INPUT SEQUENCE | SETTINGS
//-------------------------------------------------------------------------------------------------------------------------------

#define EIM_OPTMOD_InputSequence_DisableDebugMessages                   // Disables debug messages for Successful Inputs, Unsuccessful Inputs,
                                                                        // Sequenece completion, current indexes and next inputs for Dev Debugging.


#define EIM_OPTMOD_InputSequence_DecoupleFromEmbeddedInputModule        // Masks all EmbeddedInputModule Dependent References and Classes.
                                                                        // Provided to make using AbstractInputSequence<T> with any preferred input
                                                                        // handler easier.

#define EIM_OPTMOD_InputSequence_CheckGamepadInputsForKeyboardSequence  // Check gamepad inputs for each GamepadControl value in KeyboardSequence when
                                                                        // a non-sequential input is detected during UpdateSequence().

//-------------------------------------------------------------------------------------------------------------------------------
```

Settings Used For Step 1 Example

[**Step 1**] Create a new script in the installation folder (or other preferred location).

LegacyInputSequence is used here, however any name that isn't InputSequence should work fine.




```
Unity Script | 0 references
public class LegacyInputSequence : MonoBehaviour
{
    // Start is called before the first frame update
    Unity Message | 0 references
    void Start()...

    // Update is called once per frame
    Unity Message | 0 references
    void Update()...
}
```

[**Step 2**] Change **MonoBehaviour** to **InputSequence<KeyCode>** and remove the pre-generated methods.

```
0 references
public class LegacyInputSequence : InputSequence<KeyCode>
{

}
```

[**Step 3**] Create a new method called **UpdateSequence()** with the override keyword, this will contain the only custom logic required to maintain the input sequence.

```
0 references
public class LegacyInputSequence : InputSequence<KeyCode>
{
    5 references
    public override void UpdateSequence()
    {
        base.UpdateSequence();
    }
}
```

[**Step 4**] Add a check to see if the sequence is already complete before returning, followed by a variable that contains the state of the current key in the sequence.

As sequence is an array of KeyCode values, sequence[current] will return the KeyCode at the current index.

```
0 references
public class LegacyInputSequence : InputSequence<KeyCode>
{
    4 references
    public override void UpdateSequence()
    {
        // Has this sequence been completed? If so, exit.
        if (sequenceComplete) { return; }

        // Store whether the current key being checked in the sequence is pressed down or not.
        bool CurrentKey = Input.GetKeyDown(sequence[current]);
    }
}
```

[**Step 5**] Add an if statement based on the result of that current key.

Inside, set the value at successfulInputs[**current**] to true.

Then check if adding 1 to **current** will take the value over the length of the sequence.

If it doesn't, the value of **current** should increment by 1.

Otherwise, call **SetAsComplete()** to complete the sequence.

```
// Store whether the current key being checked in the sequence is pressed down or not.
bool CurrentKey = Input.GetKeyDown(sequence[current]);

// If an input is successful, set the current index to true.
//      If that input doesn't complete the sequence, go to the next index.
//      Otherwise, mark the sequence as complete.
if (CurrentKey)
{
    successfulInputs[current] = true;

    if (current + 1 < sequence.Length) { current++; }
    else SetAsComplete();
}
```

[**Step 6**] Append an else statement which checks if Input.anyKeyDown is true.

If it is, call **ResetSequence()** to reset the sequence.

```
// If an input is successful, set the current index to true.
//      If that input doesn't complete the sequence, go to the next index.
//      Otherwise, mark the sequence as complete.

// If an input isn't successful, check if there have been any other inputs.
//      If there have been, reset the sequence.

if (CurrentKey)
{
    successfulInputs[current] = true;

    if (current + 1 < sequence.Length) { current++; }
    else SetAsComplete();
}
else
{
    if (Input.anyKeyDown) { ResetSequence(); }
}
```

[**Step 7**] Create a new constructor for the class that takes an array of KeyCode values.

Set **sequence** to be that array.

Set **successfulInputs** to be a new array of boolean values equal to the length of **sequence**.

This example uses the 'params' keyword, which avoids the need to explicitly create an array by hand.

```
3 references
public class LegacyInputSequence : InputSequence<KeyCode>
{
    1 reference
    public LegacyInputSequence(params KeyCode[] keys)
    {
        // Set sequence to be the array of keys.
        sequence = keys;

        // Create a new array of bools of the same size as sequence.
        successfulInputs = new bool[sequence.Length];
    }

    4 references
    public override void UpdateSequence()...
}
```

Developers may wish to add various comments throughout the script to indicate the current status of sequences, some example debug messages can be found in the provided base classes and sub-classes.

Testing the New Class

[**Step 1**] In an existing or new script, define a variable at the top that is of the Sequence Class just created.

Use any preferred set of inputs. The directional inputs of the Konami Code are used here as it's well known.

This example used 'LegacyInputSequence', so that name is used here.

```
Unity Script | 0 references
public class TestingLIS : MonoBehaviour
{
    LegacyInputSequence legacy =
        new LegacyInputSequence(
            KeyCode.UpArrow, KeyCode.UpArrow,
            KeyCode.DownArrow, KeyCode.DownArrow,
            KeyCode.LeftArrow, KeyCode.RightArrow,
            KeyCode.LeftArrow, KeyCode.RightArrow
        );

    Unity Message | 0 references
    void Start() { }

    Unity Message | 0 references
    void Update()
    {

    }
}
```

[**Step 2**] Call Update Sequence in the Update() method of the testing script.

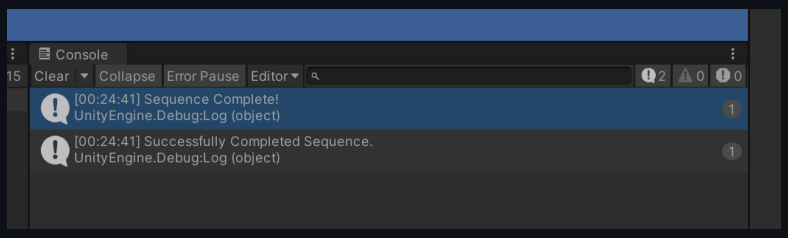
Then check if the variable is equal to true. This works because InputSequence has a custom true/false operator.

Lastly, if true, display a debug message and reset the sequence.

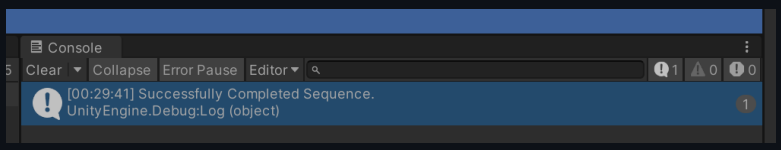
```
Unity Message | 0 references
void Update()
{
    // Keep updating the sequence.
    legacy.UpdateSequence();

    // If the sequence is complete, display a success message
    // then reset the message.
    if (legacy) {

        Debug.Log("Successfully Completed Sequence.");
        legacy.ResetSequence();
    }
}
```

[**Step 3**] Check to see if both scripts are saved, then run your project and perform the sequence.

If Debug Messages are enabled, two or more debug messages should appear:



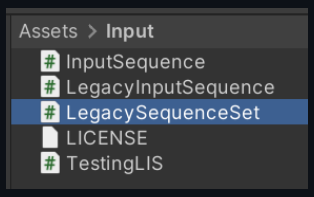
Otherwise only one debug message will appear:



Creating a Custom Sequence Set

For Developers using built-in or custom sequences.

[**Step 1**] Create a new script in the installation folder (or other preferred location) and remove the pre-generated methods.




```
using UnityEngine;

Unity Script | 0 references
public class LegacySequenceSet : MonoBehaviour
{

}
```

[**Step 2**] Change **MonoBehaviour** to **AbstractSequenceSet**.

```
0 references
public class LegacySequenceSet : AbstractSequenceSet
{

}
```

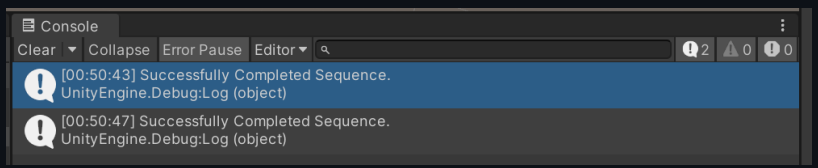
[**Step 3**] Create a new constructor for the class that takes an array of **AbstractInputSequence** objects.

Set **sequences** to be that array.

```
1 reference
public class LegacySequenceSet : AbstractSequenceSet
{
    0 references
    public LegacySequenceSet(params AbstractInputSequence[] inputSequences)
    {
        sequences = inputSequences;
    }
}
```

[**Step 4**] (Optional) If using the testing script from the previous section or a new script, define a variable at the top that is of the Sequence Set just created and test both sequences using the prior steps. The results should be identical.

```
LegacySequenceSet legacy = new LegacySequenceSet(
    new LegacyInputSequence(
        KeyCode.UpArrow, KeyCode.UpArrow,
        KeyCode.DownArrow, KeyCode.DownArrow,
        KeyCode.LeftArrow, KeyCode.RightArrow,
        KeyCode.LeftArrow, KeyCode.RightArrow
    ),
    new LegacyInputSequence(
        KeyCode.W, KeyCode.W,
        KeyCode.S, KeyCode.S,
        KeyCode.A, KeyCode.D,
        KeyCode.A, KeyCode.D
    ));
```



As **AbstractSequenceSet** is near identical to **AbstractInputSequence**, yet doesn't require logic specific to any input handler, it was able to be contained entirely within the abstract class. This allows any developer to use, modify and extend sets easily.