Input Profile

A Sub-module for Embedded Input Module providing classes for Rebindable Input Mappings.

Optional features and classes require Embedded Input Module.

Input Profile is a C# file providing several abstract clases for defining actions and a base class for the creation of rebindable action maps for projects. Additional classes are provided for use with Embedded Input Module so developers can immediately integrate Input Profiles into their Unity Projects, however, core classes are written to be entirely independent. Input Profile can be decoupled and used as a standalone module for preferred input handlers like InputSystem and Rewired.

Current Release: Version 1.2

Overview

Full C# XML Documentation

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

InputProfile

The base class containing core members and methods for managing and retrieving rebindable actions.

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

AbstractContextualAction

An abstract class containing the core members for any virtual input and properties to determine the action's input type.
Suitable for defining a collection of actions in an array or list.

BaseContextualAction<TEnum>

An abstract class defining the input source of all actions.
Not suitable for collections of actions in an array or list.

BaseContextualButton<TEnum>

An abstract class defining the Pressed, Held and Released properties of an action bound to a button.
Not suitable for collections of actions in an array or list.

BaseContextualAxis

An abstract class defining the Vector2 value of any of the available input source axes.
Suitable for defining a collection of actions in an array or list.

Embedded Input Module Classes (Optional)

ContextualKey
A Virtual Button representing a key on a Keyboard, Mouse or Gamepad.

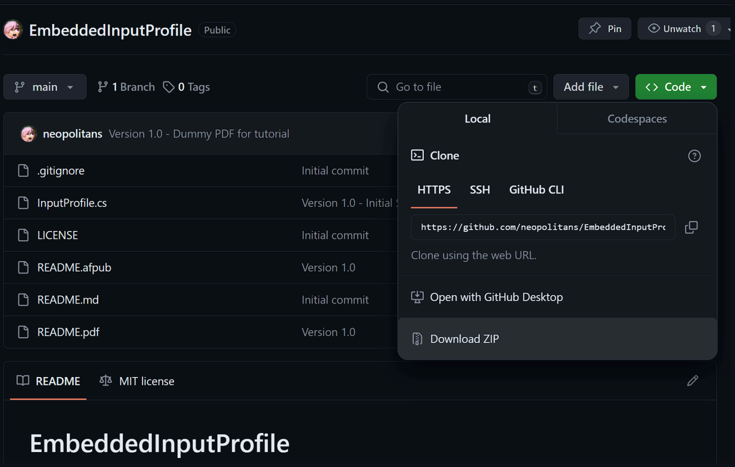
ContextualGamepad
A Virtual Button representing a control on a Gamepad.

ContextualAxis
A virtual axis that represents a multi-directional control through a Joystick, Mouse or set of 4 buttons.

PlatformInputProfile
A class containing a list of virtual inputs for player actions with an identifier for a specific platform.

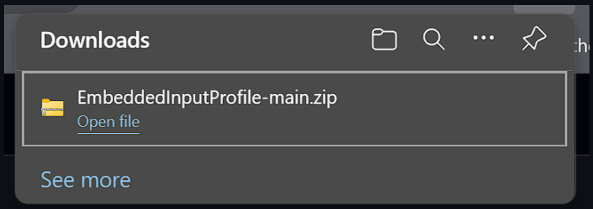
Installation

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

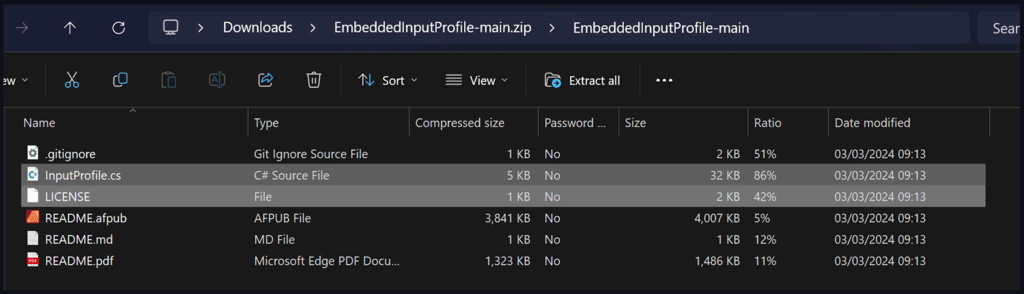


neopolitans Version 1.0 - Dummy PDF for tutorial

.gitignore	Initial commit
InputProfile.cs	Version 1.0 - Initial
LICENSE	Initial commit
README.afpub	Version 1.0
README.md	Initial commit
README.pdf	Version 1.0



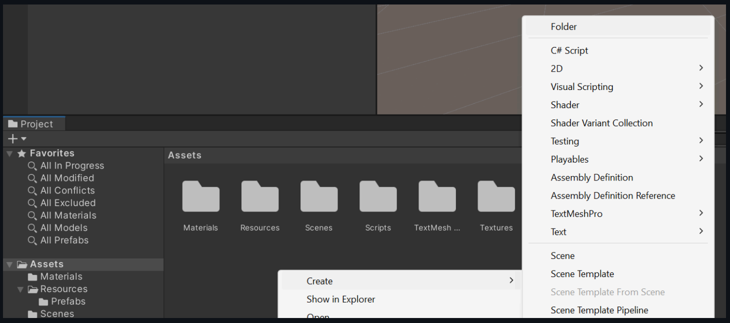
[**Step 2**] Open (or Extract) the .zip file, go into the folder then copy InputProfile.cs and the LICENSE file.

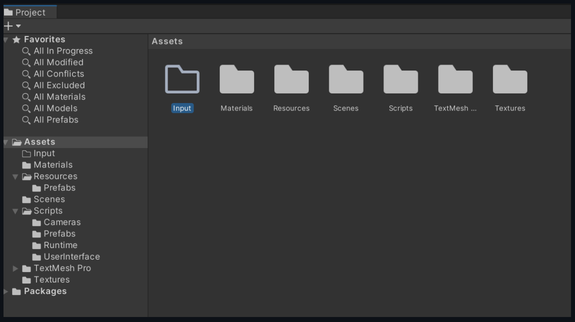


Name	Type	Compressed size	Password ...	Size	Ratio	Date modified
.gitignore	Git Ignore Source File	1 KB	No	2 KB	51%	03/03/2024 09:13
InputProfile.cs	C# Source File	5 KB	No	32 KB	86%	03/03/2024 09:13
LICENSE	File	1 KB	No	2 KB	42%	03/03/2024 09:13
README.afpub	AFPUB File	3,841 KB	No	4,007 KB	5%	03/03/2024 09:13
README.md	MD File	1 KB	No	1 KB	12%	03/03/2024 09:13
README.pdf	Microsoft Edge PDF Docu...	1,323 KB	No	1,486 KB	11%	03/03/2024 09:13

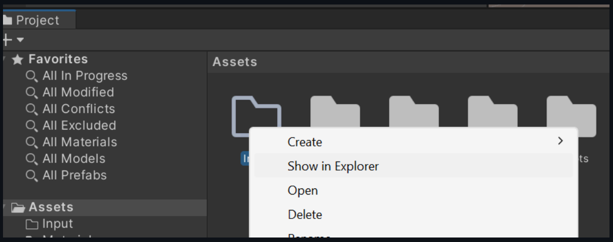
[**Step 3**] Go to the "Project" tab in Unity and create a new folder called "**Input**"

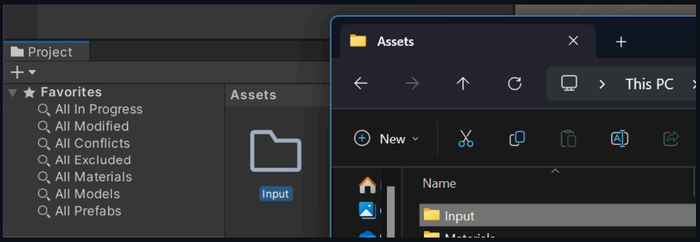
Both files **can** be copied directly into Assets or any folder with a preferred name, however some folders can be created with names which will not get included when you build your game. A detailed list can be found here.

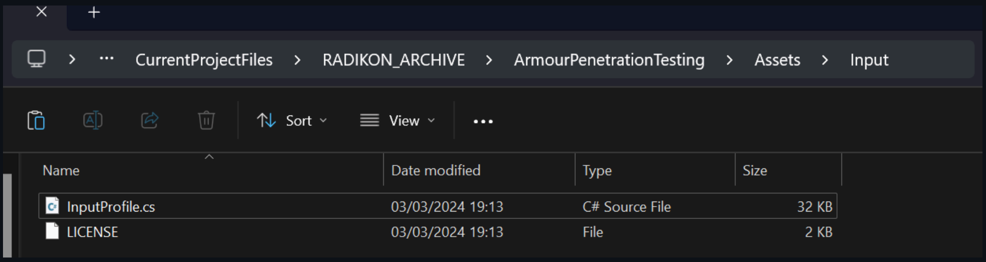




[**Step 4**] Open the new folder by highlighting it, clicking Show in Explorer and paste the copied files into the folder.





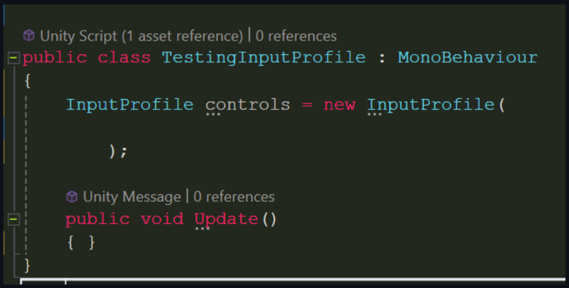


Name	Date modified	Type	Size
InputProfile.cs	03/03/2024 19:13	C# Source File	32 KB
LICENSE	03/03/2024 19:13	File	2 KB

Getting Started

Actions and Input Profiles only require a few steps for implementation. Most action classes are defined by the developer. This quick guide assumes Embedded Input Module is being used, or that the developer uses classes for a preferred input handler.

[**Step 1**] Define an Input Profile called "controls" at the top of a new or existing Script.

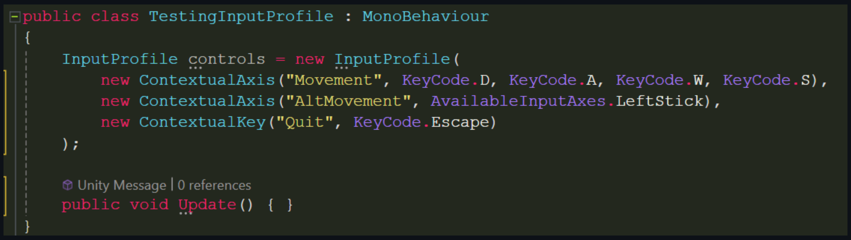


```
Unity Script (1 asset reference) | 0 references
public class TestingInputProfile : MonoBehaviour
{
    InputProfile controls = new InputProfile(

        );

    Unity Message | 0 references
    public void Update()
    { }
}
```

[**Step 2**] Create desired actions in the constructor. This guide creates three, using classes requiring Embedded Input Module:

Two Virtual Axes (**ContextualAxis**), one for four KeyCodes and one for Left Thumbstick (or MouseAxis).

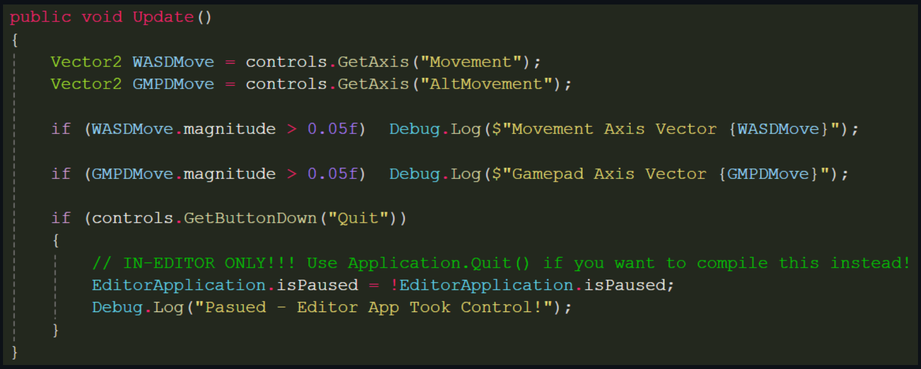
A Virtual Button (**ContextualKey**) for pausing Unity Editor or exiting the Application.



```
public class TestingInputProfile : MonoBehaviour
{
    InputProfile controls = new InputProfile(
        new ContextualAxis("Movement", KeyCode.D, KeyCode.A, KeyCode.W, KeyCode.S),
        new ContextualAxis("AltMovement", AvailableInputAxes.LeftStick),
        new ContextualKey("Quit", KeyCode.Escape)
    );

    Unity Message | 0 references
    public void Update() { }
}
```

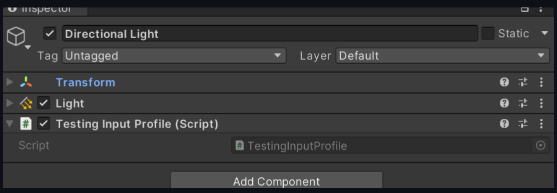
[**Step 3**] Use controls.GetAxis with an axis name, and controls.GetButtonDown inside the Update method.

GetButton, **GetButtonDown** and **GetButtonUp** return a boolean, while **GetAxis** returns a Vector2.

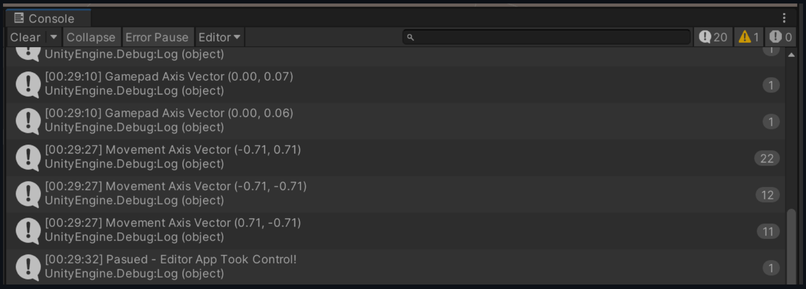


```
public void Update()
{
    Vector2 WASDMove = controls.GetAxis("Movement");
    Vector2 GMPDMove = controls.GetAxis("AltMovement");

    if (WASDMove.magnitude > 0.05f)  Debug.Log($"Movement Axis Vector {WASDMove}");

    if (GMPDMove.magnitude > 0.05f)  Debug.Log($"Gamepad Axis Vector {GMPDMove}");

    if (controls.GetButtonDown("Quit"))
    {
        // IN-EDITOR ONLY!!! Use Application.Quit() if you want to compile this instead!
        EditorApplication.isPaused = !EditorApplication.isPaused;
        Debug.Log("Pasued - Editor App Took Control!");
    }
}
```

[**Step 4**] Attach the script to a GameObject in a scene, if not already attached, and test the button inputs.



Directional Light
Tag Untagged Layer Default
Testing Input Profile (Script)
Script TestingInputProfile



```
UnityEngine.Debug:Log (object)
[00:29:10] Gamepad Axis Vector (0.00, 0.07)          1
UnityEngine.Debug:Log (object)
[00:29:10] Gamepad Axis Vector (0.00, 0.06)          1
UnityEngine.Debug:Log (object)
[00:29:27] Movement Axis Vector (-0.71, 0.71)        22
UnityEngine.Debug:Log (object)
[00:29:27] Movement Axis Vector (-0.71, -0.71)       12
UnityEngine.Debug:Log (object)
[00:29:27] Movement Axis Vector (0.71, -0.71)        11
UnityEngine.Debug:Log (object)
[00:29:32] Pasued - Editor App Took Control!         1
UnityEngine.Debug:Log (object)
```

Input Profiles are fully functional as static objects in a static class and can work inside a custom Input Manager.

Classes

New Abstract Class - AbstractContextualAction

As the core class containing members, properties and methods for any action, this is provided to facilitate collections of any Inheritng type. No input-reading logic is present, however, this is suitable for inheritance if developers are only concerned with collecting custom actions in a base InputProfile.

Members

	Description
AbstractContextualAction.label	The name that identifies the Action. Used to query the action during runtime.
AbstractContextualAction.onRebind	The delegate called when the Action is rebound during runtime.
AbstractContextualAction.isAxis	Does the type of the Action match that of a Virtual Axis?
AbstractContextualAction.isButton	Does the type of the action match that of a Virtual Button?

Methods

	Description
AbstractContextualAction.Clone	Creates a duplicate of the action. This will provide the data of the current instance to the class' constructor.

New Abstract Class - BaseContextualAction<TEnum> (TEnum is any type of enum class)

Inheritance: **AbstractContextualAction**

The first step of inheritance for most classes in this module, BaseContextualAction provides the ability to define any type of input source with an enumerator. This could store the direct source of an input or be a category defining input types.

Members

	Description
BaseContextualAction.inputSource	The source of an action's input values. An example being a KeyCode value, returning the respective button's state.

New Abstract Class - BaseContextualButton<TEnum> (TEnum is any type of enum class)

Inheritance: **BaseContextualAction<TEnum>, IDeviceInputButton**

Specifically for Keys, Gamepad Buttons or other inputs which return their state in a boolean form, BaseContextualButton contains the three common states of any virtual button. This leverages the generic typing of BaseContextualAction so that inputSource will share the type of the enumerator provided such as a KeyCode if UnityEngine.KeyCode is used.

Members

	Description
BaseContextualButton.Pressed	Whether the virtual button is pressed.
BaseContextualButton.Released	Whether the virtual button is released.
BaseContextualButton.Held	Whether the virtual button is held.

New Abstract Class - BaseContextualAxis

Inheritance: **BaseContextualAction<AvailableInputAxes>**

A type of Action requiring developers to provide logic for getting a Vector2 from a joystick, mouse or virtual axis of buttons.

Members

	Description
BaseContextualAxis.Value	A Vector2 directly provided from a joystick, mouse or virtual axis of buttons.

New Class - InputProfile

A class that contains a list of virtual inputs for player actions. This is suitable for collections in arrays and lists, if desired.

Methods

	Description
InputProfile.actions	The actions contained in the profile which can be queried.

Methods

	Description
InputProfile.GetButton	Returns true while the virtual button identified by the given name is held down.
InputProfile.GetButtonUp	Returns true during the first frame the user releases the virtual button identified by the given name.
InputProfile.GetButtonDown	Returns true during the first frame the user pressed down the virtual button identified by the given name.
InputProfile.GetAxis	Returns the value of the virtual axis identified by the given name. The values will be in the range of -1 to 1 for keyboard, joystick and mouse input devices.
InputProfile.GetAction	Returns the AbstractContextualAction identified by the given name. This is useful for rebinding controls.
InputProfile.Clone	Create a duplicate of the Input Profile by creating duplicates of each action within and assigning them to the duplicate instance.

Constructors

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

Input Profiles can also be used to create Input Managers, where a default profile exists constantly and is cloned so that there can be a rebindable control map while providing a backup control mapping to restore.

New Interface - IDeviceInputButton

An interface which requires implementing classes to provide properties for getting the Held, Pressed and Released states of a virtual button. This interface is also used to identify if an Action is a type of button.

This is a globally accessible Interface. It is marked as internal to denote it’s intended use as an internal type-casting utility.

Members

	Description
IDeviceInputButton.Pressed	Whether the virtual button is pressed.
IDeviceInputButton.Released	Whether the virtual button is released.
IDeviceInputButton.Held	Whether the virtual button is held.

New Enums

	Description
AvailableInputAxes	An enum as a list of basic supported input sources for a virtual axis. This list is incomplete and can be expanded if you do not wish to create more enumerator types. • LeftStick • RightStick • MouseAxis • ButtonAxis (4-Button Virtual Axis)

Optional Subclasses (Requires Embedded Input Module)

These classes inherit from BaseContextualButton, BaseContextualAxis and InputProfile, using a handful of features from Embedded Input Module. These can also be modified to work with other input handlers if desired.

New Class - ContextualKey

Inheritance: **BaseContextualButton<UnityEngine.KeyCode>**

A virtual button representing a key on a Keyboard, Mouse or Gamepad.

Members

	Description
ContextualKey.altInputSource	An optional alternate input source used for the virtual button's input state.

Methods

	Description
ContextualKey.Clone	Constructs a duplicate virtual button using the most detailed constructor available.

Constructors

	Description
ContextualKey(string, KeyCode)	A constructor for a virtual button that takes a label and a KeyCode as the primary input source.
ContextualKey(string, KeyCode, System.Action)	A constructor for a virtual button that takes a label, a KeyCode as the primary input source and a delegate method to bind to the callback invoked when a key is rebound.
ContextualKey(string, KeyCode, KeyCode)	A constructor for a virtual button that takes a label and two KeyCodes. The first as the primary input source and the second as the alternate input source.
ContextualKey(string, KeyCode, KeyCode, System.Action)	A constructor for a virtual button that takes a label, two KeyCodes and a delegate method to bind to the callback invoked when a key is rebound. The first KeyCode is the primary input source and the second is the alternate input source.

New Class - PlatformInputProfile

Inheritance: **InputProfile**

A class containing a list of virtual inputs for player actions with an identifier for a specific platform.

Members

	Description
PlatformInputProfile.platform	The platform that this Input Profile is created for.

Methods

	Description
PlatformInputProfile.Clone	Create a duplicate of the Input Profile by creating duplicates of each action within and assigning them to the duplicate instance.

Constructors

	Description
PlatformInputProfile(InputIconDisplayType, AbstractContextualAction[])	A constructor for PlatformInputProfile that takes a target platform and any number of AbstractContextualActions. (uses **params** to make it possible to have any amount of values)

This class does have limited use-cases and exists mostly as an example, however, developers may find some utility in associating a specific platform to a profile in a collection of profiles. Developers may also find utility in having a specific platform associated with a profile to prevent a mis-matched input profile and input device issue.

New Class - ContextualGamepadButton

Inheritance: BaseContextualButton<EmbeddedInputModule.GamepadControl>

A virtual button representing a button on a Gamepad. As gamepads have limited buttons, there is only one input source.

Methods

	Description
ContextualGamepadButton.Clone	Constructs a duplicate virtual button using the most detailed constructor available.

Constructors

	Description
ContextualGamepadButton(string, GamepadControl)	A constructor for a virtual button that takes a label and a GamepadControl as the input source.
ContextualGamepadButton(string, GamepadControl, System.Action)	A constructor for a virtual button that takes a label,a GamepadControl as the input source and a delegate method to bind to the callback invoked when a key is rebound.

New Class - ContextualAxis

Inheritance: BaseContextualAxis

A virtual axis that rerpesents a multi-directional control from a joystick, Mouse or set of 4 buttons.

Methods

	Description
ContextualAxis.positiveX	The KeyCode for positive X.
ContextualAxis.negativeX	The KeyCode for negative X.
ContextualAxis.positiveY	The KeyCode for positive Y.
ContextualAxis.negativeY	The KeyCode for negative Y.
ContextualAxis.altPositiveX	The optional alternate KeyCode for positive X.
ContextualAxis.altNegativeX	The optional alternate KeyCode for negative X.
ContextualAxis.altPositiveY	The optional alternate KeyCode for positive Y.
ContextualAxis.altNegativeY	The optional alternate KeyCode for negative Y.

Constructors

	Description
ContextualAxis(string, AvailableInputAxes)	A constructor for a virtual axis that reads from an input axis or a blank button axis.
ContextualAxis(string, KeyCode, KeyCode, KeyCode, KeyCode)	A constructor for a virtual axis that reads from a completed 4-button axis.
ContextualAxis(string, AvailableInputAxes, System.Action)	A constructor for a virtual axis that reads from an input axis or a blank button axis with a callback triggered when the action is rebound.
ContextualAxis(string, KeyCode, KeyCode, KeyCode, KeyCode, System.Action)	A constructor for a virtual axis that reads from a completed 4-button axis with a callback triggered when the action is rebound.
ContextualAxis(string, KeyCode, KeyCode, KeyCode, KeyCode, KeyCode, KeyCode, KeyCode, KeyCode)	A constructor for a virtual axis that reads from a completed 4-button axis, with 4 alternate buttons for each direction.
ContextualAxis(string, KeyCode, KeyCode, KeyCode, KeyCode, KeyCode, KeyCode, KeyCode, KeyCode, System.Action)	A constructor for a virtual axis that reads from a completed 4-button axis, with 4 alternate buttons for each direction.

For the brevity of this document 4 constructors have been omitted from the list. These use GamepadControl, however, they convert that value into a KeyCode value. Such constructors may not be of use, especially as EmbeddedInputModule.Dpad exists, however they are provided for convenience over the numerous KeyCode values for joystick buttons.

Alternate Axes are not advised for virtual axes constructed with Gamepad Controls.

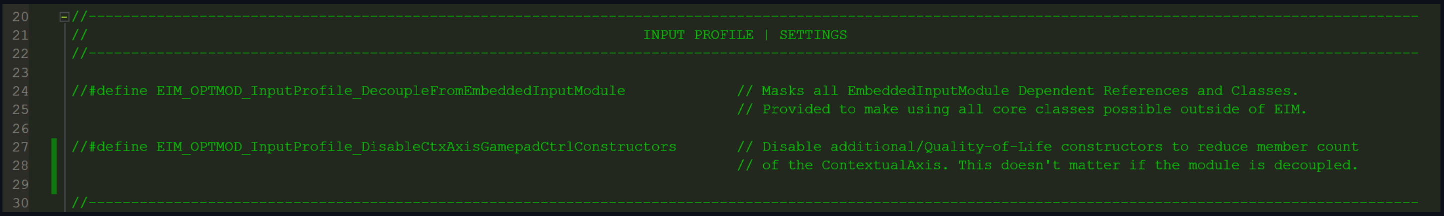
Configurable Settings

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 InputProfile.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.



```
20  //------------------------------------------------------------------------------------------------------------------------------------------------
21  //                                                   INPUT PROFILE | SETTINGS
22  //------------------------------------------------------------------------------------------------------------------------------------------------
23
24  //#define EIM_OPTMOD_InputProfile_DecoupleFromEmbeddedInputModule             // Masks all EmbeddedInputModule Dependent References and Classes.
25                                                                              // Provided to make using all core classes possible outside of EIM.
26
27  //#define EIM_OPTMOD_InputProfile_DisableCtxAxisGamepadCtrlConstructors       // Disable additional/Quality-of-Life constructors to reduce member count
28                                                                              // of the ContextualAxis. This doesn't matter if the module is decoupled.
29
30  //------------------------------------------------------------------------------------------------------------------------------------------------
```

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_InputProfile_DecoupleFromEmbeddedInputModule

Masks **ContextualKey**, **ContextualGamepadButton**, **ContextualAxis** and **PlatformInputProfile** 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 24

EIM_OPTMOD_InputProfile_DisableCtxAxisGamepadCtrlConstructors.

Disables the additional constructors provided for the optional class ContextualAxis, which take GamepadControls instead of KeyCodes. This is recommended to be enabled if the amount of constructor variations are causing issues with intellisense or if the additional constructors go unused.

Disabling these constructors may have little impact outside of compilation time as a slight micro-optimization.

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

Location: Line 27

One Last Note

Due to the names of InputActions and InputActionMaps, the naming of Actions and Action Mappings as "ContextualAction" and "InputProfile" are to prevent class overlapping and emphasise that actions are defined by their context. These can be used to work for **any** input handler.

Using a Custom Input Handler

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 other contemporary, off-the-shelf solutions to handling game input. To achieve the flexibility of Input Profile, these tutorials demonstrate the requirements and the custom classes needed for an input handler. These tutorials use legacy **UnityEngine.Input** class so that any developer can run the same code and modify them easily.

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

- An enumerator that represents an input button, axis or type **or** compatiblity with enums like UnityEngine.KeyCode
- A method or property to access whether an input is pressed, released, held or to get a Vector2 value.

Any input handler not directly compatible may need additional translation layers to convert an input device's state into a readable format for this module.

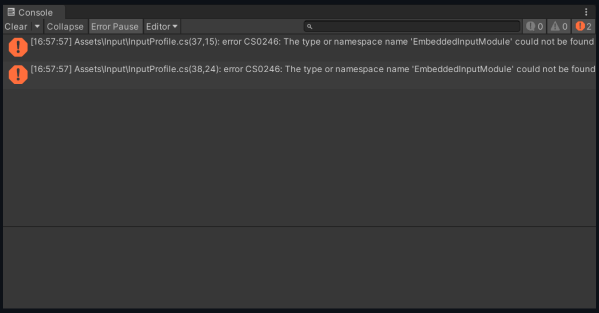
This comprehensive guide contains all core step so beginners can follow along.

Sorry for any inconvenience in advance.

Setup for All Tutorials

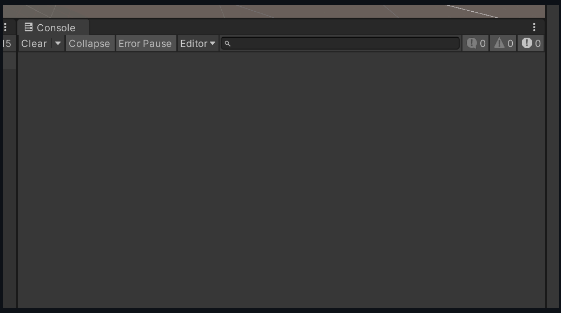
All the proceeding tutorials will start at the class creation step, however the initial setup is provided here for all subsequent tutorials. All tutorials hereon out will expect, unless stated, for this to be done first.

[**Step 1**] (All Tutorials) Enable the Decoupling from Embedded Input Module setting.

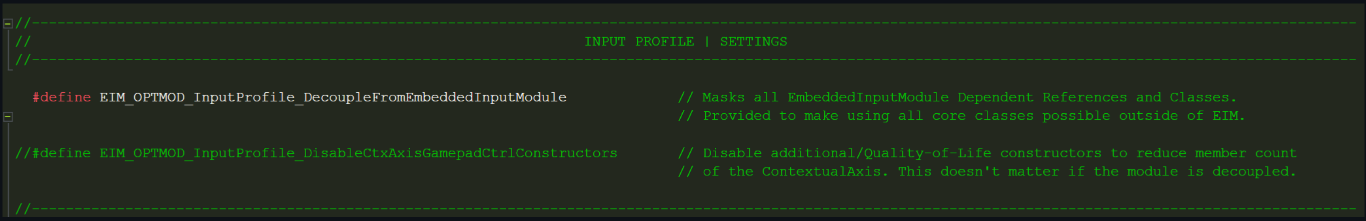


```
[16:57:57] Assets\Input\InputProfile.cs(37,15): error CS0246: The type or namespace name 'EmbeddedInputModule' could not be found
[16:57:57] Assets\Input\InputProfile.cs(38,24): error CS0246: The type or namespace name 'EmbeddedInputModule' could not be found
```

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



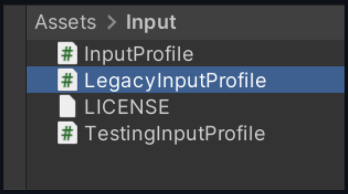
There should only be messages related to your project.

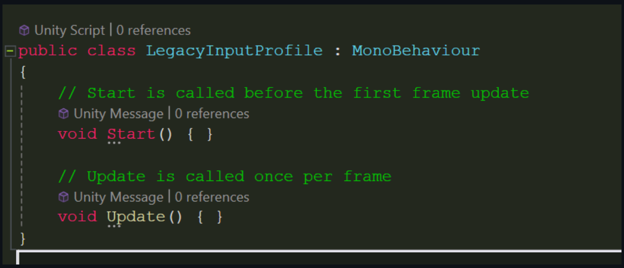


```
//----------------------------------------------------------------------------------------------------------------------------------------------------------------
//                                                          INPUT PROFILE | SETTINGS
//----------------------------------------------------------------------------------------------------------------------------------------------------------------

  #define EIM_OPTMOD_InputProfile_DecoupleFromEmbeddedInputModule            // Masks all EmbeddedInputModule Dependent References and Classes.
                                                                             // Provided to make using all core classes possible outside of EIM.

//#define EIM_OPTMOD_InputProfile_DisableCtxAxisGamepadCtrlConstructors      // Disable additional/Quality-of-Life constructors to reduce member count
                                                                             // of the ContextualAxis. This doesn't matter if the module is decoupled.

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

Settings Used For Step 1 Example

[**Step 2**] Create a new script in the folder InputProfile is stored in or use an existing Input Manager script.





```
Unity Script | 0 references
public class LegacyInputProfile : 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() { }
}
```

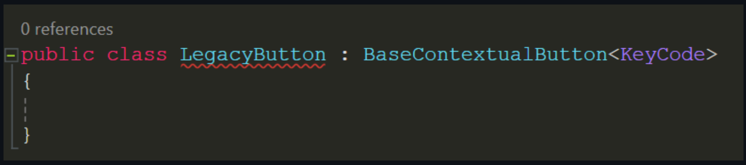
The class generated, though not namespace declarations above, should be deleted for tutorials, unless stated.

The script will be used to demonstrate various classes and the file name is only for tutorial purposes.

Creating a Custom Button

This section covers creating a class for a virtual button action that can be identified and read from by Input Profile. There are minimal additions required and any input handler that meets the minimum requirements **should** be compatible.

[**Step 1**] Create a class called "LegacyButton" that inherits from **BaseContextualButton<KeyCode>**.

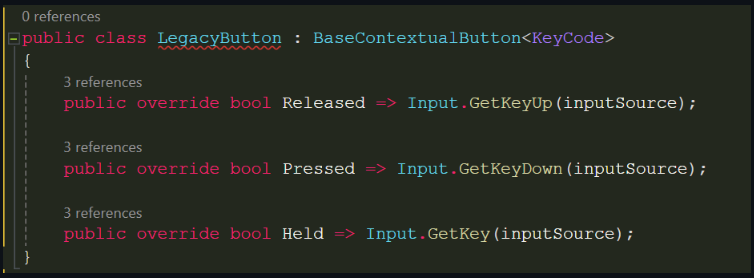


```
0 references
public class LegacyButton : BaseContextualButton<KeyCode>
{

}
```

The error highlight is because there are five missing members, added in the next steps.

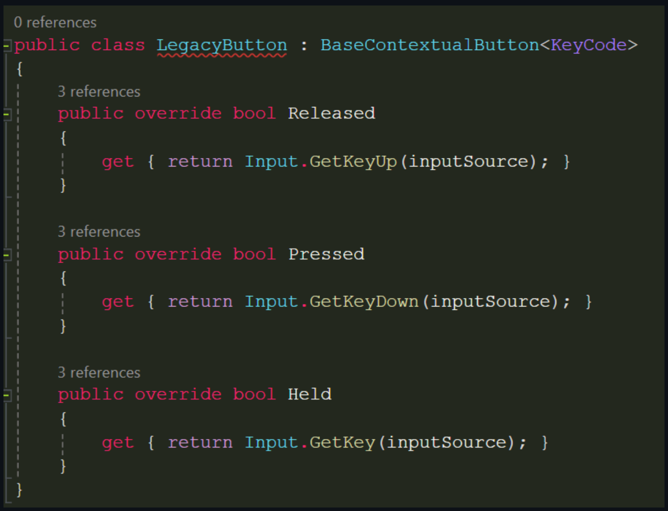
[**Step 2**] Create three **public override bool** properties called Released, Pressed and Held.
Have the return values call legacy Input class methods for "GetKeyUp", "GetKeyDown" and "GetKey".

These can be created with a lambda expression:



```
0 references
public class LegacyButton : BaseContextualButton<KeyCode>
{
    3 references
    public override bool Released => Input.GetKeyUp(inputSource);

    3 references
    public override bool Pressed => Input.GetKeyDown(inputSource);

    3 references
    public override bool Held => Input.GetKey(inputSource);
}
```

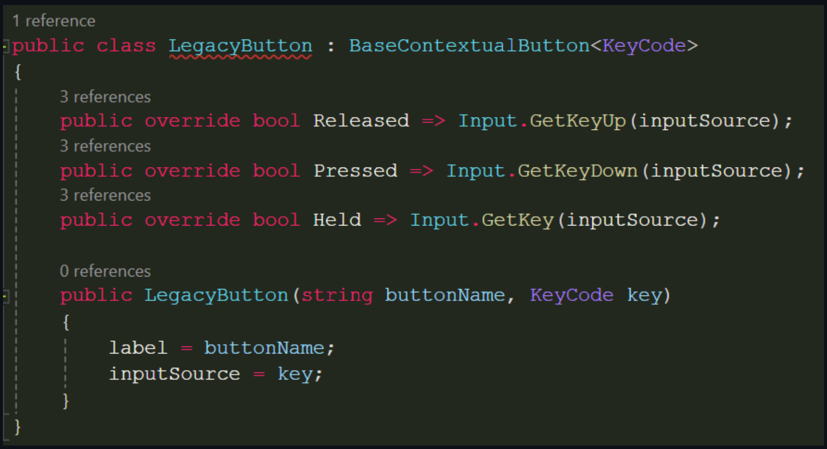
Or with a more traditional get-set accessor:



```
0 references
public class LegacyButton : BaseContextualButton<KeyCode>
{
    3 references
    public override bool Released
    {
        get { return Input.GetKeyUp(inputSource); }
    }

    3 references
    public override bool Pressed
    {
        get { return Input.GetKeyDown(inputSource); }
    }

    3 references
    public override bool Held
    {
        get { return Input.GetKey(inputSource); }
    }
}
```

The KeyCode to provide UnityEngine.Input is stored in LegacyButton.inputSource, inherited from BaseContextualAction.

[**Step 3**] Create a constructor that takes a string labelled "buttonName" and a KeyCode labelled "key".
In the constructor body, set "label" to buttonName and "inputSource" to key.

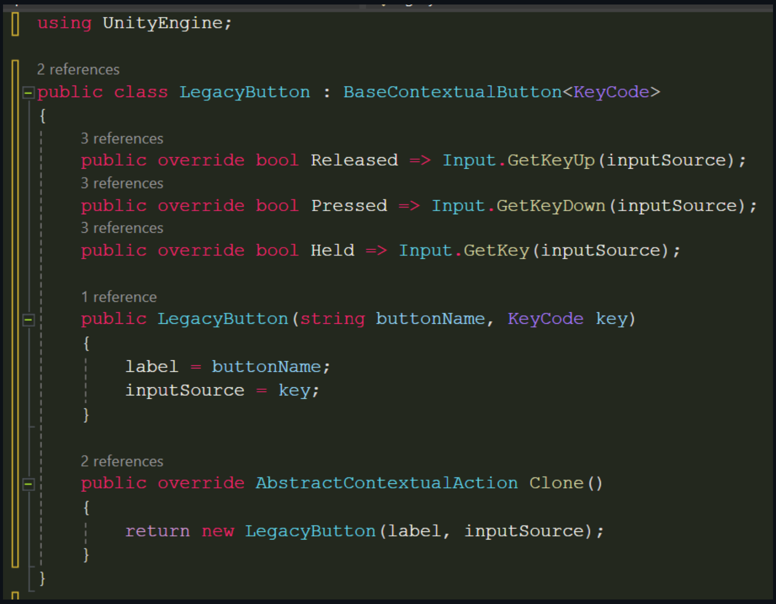


```
1 reference
public class LegacyButton : BaseContextualButton<KeyCode>
{
    3 references
    public override bool Released => Input.GetKeyUp(inputSource);
    3 references
    public override bool Pressed => Input.GetKeyDown(inputSource);
    3 references
    public override bool Held => Input.GetKey(inputSource);

    0 references
    public LegacyButton(string buttonName, KeyCode key)
    {
        label = buttonName;
        inputSource = key;
    }
}
```

LegacyButton.label is inherited from AbstractContextualAction, the first abstract class inherited by any subsequent "Action" class in this script.

[**Step 4**] Lastly, create a **public override AbstractContextualAction** method called 'Clone' below the constructor.

In the method body, return a new instance of LegacyButton, passing in the button's label and inputSource.

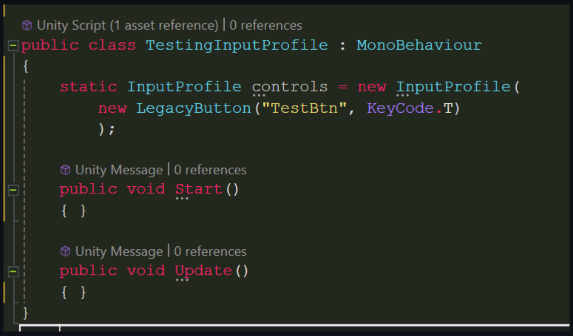


```
using UnityEngine;

2 references
public class LegacyButton : BaseContextualButton<KeyCode>
{
    3 references
    public override bool Released => Input.GetKeyUp(inputSource);
    3 references
    public override bool Pressed => Input.GetKeyDown(inputSource);
    3 references
    public override bool Held => Input.GetKey(inputSource);

    1 reference
    public LegacyButton(string buttonName, KeyCode key)
    {
        label = buttonName;
        inputSource = key;
    }

    2 references
    public override AbstractContextualAction Clone()
    {
        return new LegacyButton(label, inputSource);
    }
}
```

Testing the Custom Button

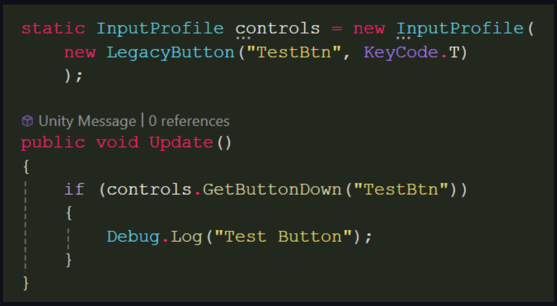
[**Step 1**] Create another new script, or open an existing one if preferred and create a new InputProfile at the top.

In that InputProfile, add a new LegacyButton with the button name of "TestBtn", bound to a preferred key.



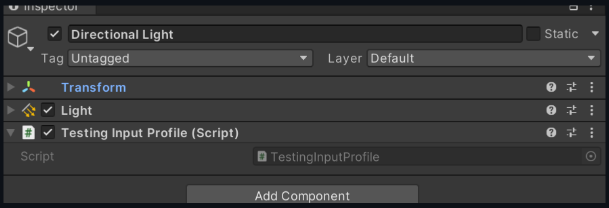
```
Unity Script (1 asset reference) | 0 references
public class TestingInputProfile : MonoBehaviour
{
    static InputProfile controls = new InputProfile(
        new LegacyButton("TestBtn", KeyCode.T)
        );

    Unity Message | 0 references
    public void Start()
    { }

    Unity Message | 0 references
    public void Update()
    { }
}
```

[**Step 2**] Check if 'TestBtn' is pressed in update using profileVariable.GetButtonDown and call Debug.Log if it has been.

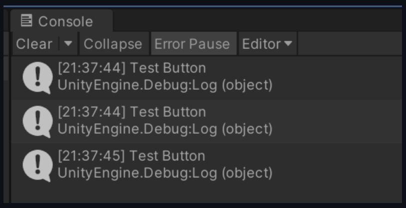


```
static InputProfile controls = new InputProfile(
    new LegacyButton("TestBtn", KeyCode.T)
    );

Unity Message | 0 references
public void Update()
{
    if (controls.GetButtonDown("TestBtn"))
    {
        Debug.Log("Test Button");
    }
}
```

[**Step 3**] Attach the script to a GameObject in a scene, if not already attached, and test the button inputs.



A GameObject with the test script added.



```
[21:37:44] Test Button
UnityEngine.Debug:Log (object)
[21:37:44] Test Button
UnityEngine.Debug:Log (object)
[21:37:45] Test Button
UnityEngine.Debug:Log (object)
```

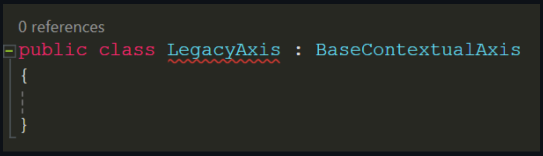
Debug Messages on Button Pressed.

Creating a Custom Axis

This section covers creating a class for a virtual axis action that can be identified and read from by Input Profile. This tutorial focuses on creating an axis that supports Mouse movement and four input keys.

As this tutorial uses **UnityEngine.Input** to demonstrate an input handler that fulfils the minimum requirements, support for Controller thumbsticks varies per controller and operating system (annoyingly). It is recommended to use an input handler that automates this process for developers' sanity.

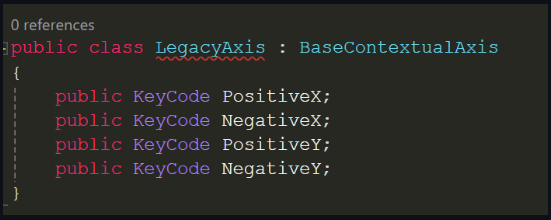
[**Step 1**] Create a class called "LegacyAxis" that inherits from **BaseContextualAxis**



```
0 references
public class LegacyAxis : BaseContextualAxis
{

}
```

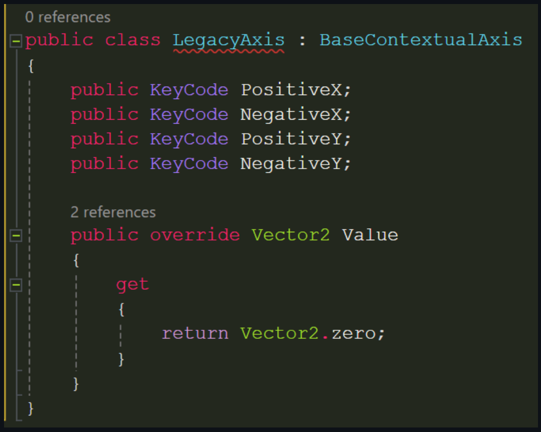
The error highlight is because there are two missing members, added in the next steps.

[**Step 2**] Create four KeyCode variables. These will be used for creating one of the two return values.



```
0 references
public class LegacyAxis : BaseContextualAxis
{
    public KeyCode PositiveX;
    public KeyCode NegativeX;
    public KeyCode PositiveY;
    public KeyCode NegativeY;
}
```

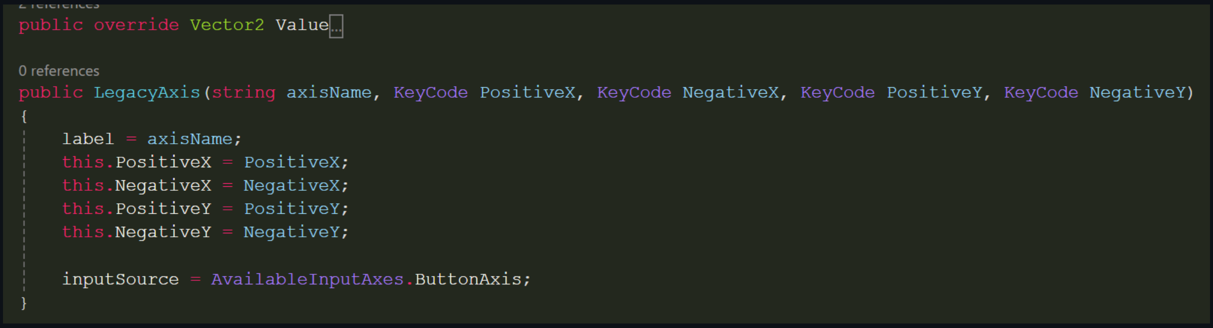
[**Step 3**] Create a **public override Vector2** property. This must have a get accessor, a lambda expression will not work. Temporarily have this return Vector2.zero to prevent extra console errors until testing.



```
0 references
public class LegacyAxis : BaseContextualAxis
{
    public KeyCode PositiveX;
    public KeyCode NegativeX;
    public KeyCode PositiveY;
    public KeyCode NegativeY;

    2 references
    public override Vector2 Value
    {
        get
        {
            return Vector2.zero;
        }
    }
}
```

[**Step 4**] Create two class constructors below the Value property.

Constructor 1 should take a string called 'axisName' and four KeyCode values. Assign these to LegacyAxis.label and the KeyCode variables created in Step 2. Assign LegacyAxis.inputSource to **AvailableInputAxes.ButtonAxis**.



```
2 references
public override Vector2 Value...

0 references
public LegacyAxis(string axisName, KeyCode PositiveX, KeyCode NegativeX, KeyCode PositiveY, KeyCode NegativeY)
{
    label = axisName;
    this.PositiveX = PositiveX;
    this.NegativeX = NegativeX;
    this.PositiveY = PositiveY;
    this.NegativeY = NegativeY;

    inputSource = AvailableInputAxes.ButtonAxis;
}
```

Constructor 2 should take a string called 'axisName' and an **AvailableInputAxes** value called "axisType".

```
public override Vector2 Value...

0 references
public LegacyAxis(string axisName, KeyCode PositiveX, KeyCode NegativeX, KeyCode PositiveY, KeyCode NegativeY)...

0 references
public LegacyAxis(string axisName, AvailableInputAxes axisType)
{
    label = axisName;
    inputSource= axisType;
}
```

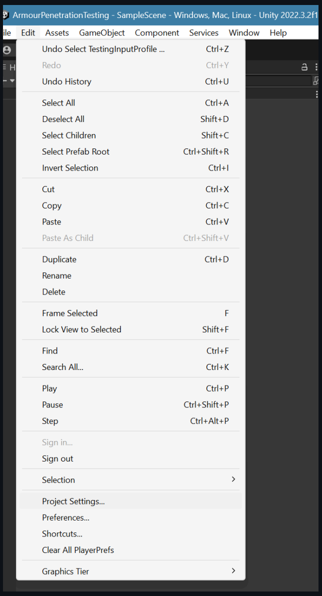
These constructors will be useful later for the Clone method as well. For now, they will give two options with the chance for expansion later.

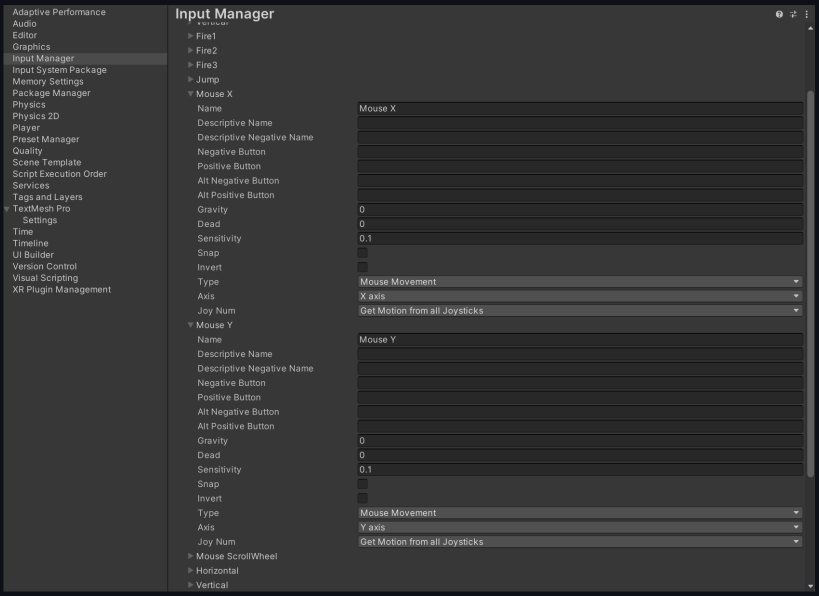
[**Step 5**] Return to the Value property and replace the temporary value with a switch statement switching on the value of LegacyAxis.inputSource. This switch statement should provide cases for default, MouseAxis and ButtonAxis.

Default should return **Vector2**.zero as an error-handling condition.

```
2 references
public override Vector2 Value
{
    get
    {
        switch (inputSource)
        {
            default: return Vector2.zero;
        }
    }
}
```

Mouse Input, however, requires two axes to be present in the legacy Input Manager.
Check if two axes called "Mouse X" and "Mouse Y" are present, or add them.





Input Manager

- Fire1
- Fire2
- Fire3
- Jump
- Mouse X

Name	Mouse X
Descriptive Name	
Descriptive Negative Name	
Negative Button	
Positive Button	
Alt Negative Button	
Alt Positive Button	
Gravity	0
Dead	0
Sensitivity	0.1
Snap	
Invert	
Type	Mouse Movement
Axis	X axis
Joy Num	Get Motion from all Joysticks

- Mouse Y

Name	Mouse Y
Descriptive Name	
Descriptive Negative Name	
Negative Button	
Positive Button	
Alt Negative Button	
Alt Positive Button	
Gravity	0
Dead	0
Sensitivity	0.1
Snap	
Invert	
Type	Mouse Movement
Axis	Y axis
Joy Num	Get Motion from all Joysticks

- Mouse ScrollWheel
- Horizontal
- Vertical

Once the axes are found or created, set a new case for **AvailableInputAxes.MouseAxis** that creates a new **Vector2** value from Input.GetAxis("Mouse X") and Input.GetAxis("Mouse Y").

```
2 references
public override Vector2 Value
{
    get
    {
        switch (inputSource)
        {
            default: return Vector2.zero;

            case AvailableInputAxes.MouseAxis:
                return new Vector2(Input.GetAxis("Mouse X"), Input.GetAxis("Mouse Y"));
        }
    }
}
```

ButtonAxis requires the most setup but thankfully, the KeyCodes and Constructor required has already been taken care of. Create a new case for **AvailableInputAxes.ButtonAxis** with two float variables at the top called "x" and "y".

This case should create a new **Vector2** at the end in a variable using the X and Y variables at the top then return that variable.

```
case AvailableInputAxes.MouseAxis:
    return new Vector2(Input.GetAxis("Mouse X"), Input.GetAxis("Mouse Y"));

case AvailableInputAxes.ButtonAxis:
    float x = 0;
    float y = 0;

    Vector2 result = new Vector2(x, y);
    return result;
```

Each button **needs** to be counted if it is pressed or held and, unfortunately, each button needs it's own if statement. An if-else statement would ignore any buttons below the one in the list that returned "true".

With these two factors, Input.GetKey and Input.GetKeyDown are required in each case, for each axis button. Positive inputs should add **1.0f**, while negative inputs should subtract **1.0f**.

```
case AvailableInputAxes.ButtonAxis:
    float x = 0;
    float y = 0;

    if (Input.GetKeyDown(PositiveX) || Input.GetKey(PositiveX))
        x += 1.0f;

    Vector2 result = new Vector2(x, y);
    return result;
```

This is the first if-statement provided, for ease of reading and easier bugfixing.

```
if (Input.GetKeyDown(PositiveX) || Input.GetKey(PositiveX))
    x += 1.0f;
if (Input.GetKeyDown(NegativeX) || Input.GetKey(NegativeX))
    x -= 1.0f;
if (Input.GetKeyDown(PositiveY) || Input.GetKey(PositiveY))
    y += 1.0f;
if (Input.GetKeyDown(NegativeY) || Input.GetKey(NegativeY))
    y -= 1.0f;
```

All if-statements provided, for clarity of use.
Apply house-rules for scripting depending on individual or company standards.

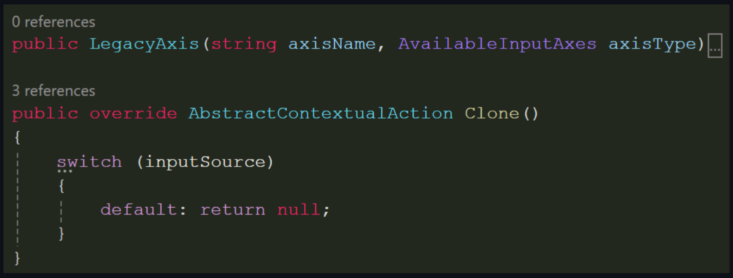
The final part of this step is to reduce memory usage slightly. Between defining the result **Vector2** and returning it, call result.Normalize().

```
Vector2 result = new Vector2(x, y);
result.Normalize();
return result;
```

Vector2.normalized would return a new vector with the normalized values of the result vector that was just created. **Vector2**.Normalize() directly modifies the result vector instead. Unless, for any reason, the original result is needed, the latter is the best course of action here.

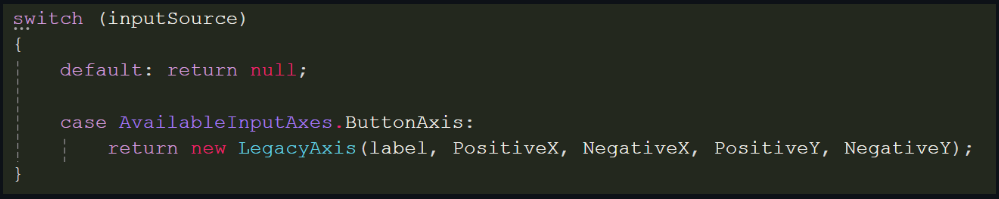
[**Step 6**] Lastly, create a **public override AbstractContextualAction** method called 'Clone' below the constructor. In the method body, there should be a switch statement switching on the value of LegacyAxis.inputSource.

Default should return **null** as an error handling condition.



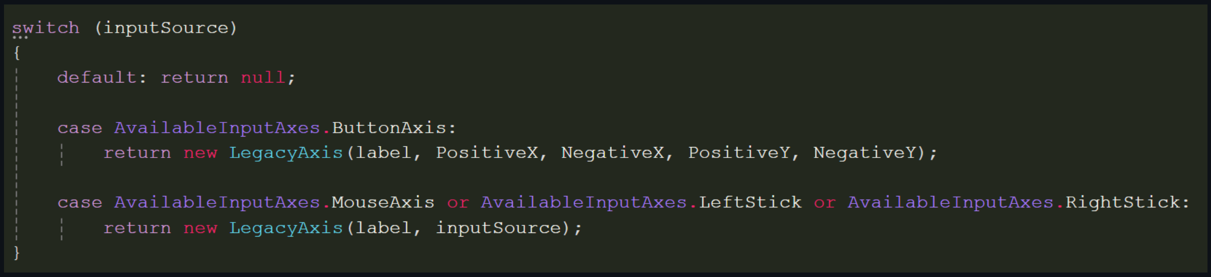
```
0 references
public LegacyAxis(string axisName, AvailableInputAxes axisType)...

3 references
public override AbstractContextualAction Clone()
{
    switch (inputSource)
    {
        default: return null;
    }
}
```

ButtonAxis should return a new LegacyAxis instance, passing in label and all KeyCode inputs.



```
switch (inputSource)
{
    default: return null;

    case AvailableInputAxes.ButtonAxis:
        return new LegacyAxis(label, PositiveX, NegativeX, PositiveY, NegativeY);
}
```

The last case should comprise of all three remaining values of **AvailableInputAxes**, as they would be constructed the same regardless of value.

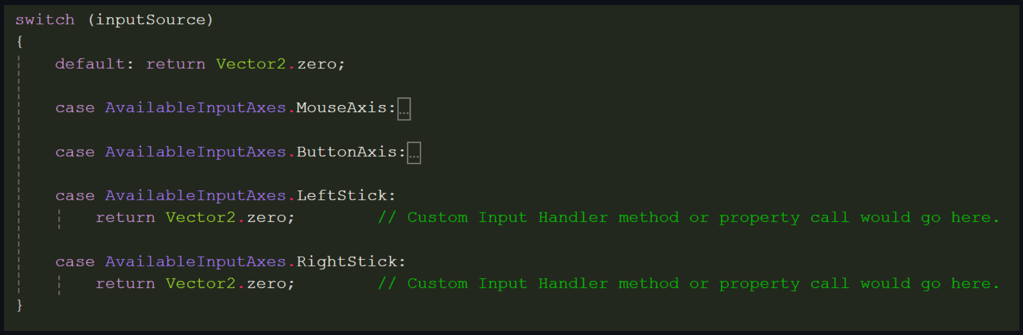


```
switch (inputSource)
{
    default: return null;

    case AvailableInputAxes.ButtonAxis:
        return new LegacyAxis(label, PositiveX, NegativeX, PositiveY, NegativeY);

    case AvailableInputAxes.MouseAxis or AvailableInputAxes.LeftStick or AvailableInputAxes.RightStick:
        return new LegacyAxis(label, inputSource);
}
```

Joystick Support?

The last case of Step 6 shows two axis values for controller thumbsticks. This **is** possible with the legacy input system, however, extremely tedious. Joystick Support, as noted by various developers, varies based on the type of controller that is being used **and** the operating system of the user. Windows, MacOS and Linux all have slight variations in how a controller is set-up when detected by Unity Engine. This step is **entirely optional** and is best used with an input handler that automatically converts thumbsticks and joysticks to Vector2 values.

In the property "Value", add two cases for **AvailableInputAxes.LeftStick** and **AvailableInputAxes.RightStick**. Have these return the corresponding Vector2 value for a left thumbstick and right thumbstick.



```
switch (inputSource)
{
    default: return Vector2.zero;

    case AvailableInputAxes.MouseAxis:...

    case AvailableInputAxes.ButtonAxis:...

    case AvailableInputAxes.LeftStick:
        return Vector2.zero;        // Custom Input Handler method or property call would go here.

    case AvailableInputAxes.RightStick:
        return Vector2.zero;        // Custom Input Handler method or property call would go here.
}
```

Alternatively, code that depends on each platform can be used to get joystick input values, though the reliability of resources varies on several factors. **Please** only do that route if you are either absolutely sure of the resource(s) you are using, and/or are extremely familiar with the input handler you are using. If you are going to attempt this with **UnityEngine.Input**, be warned that it is an extreme headache for new developers.

Testing the Custom Axis

[**Step 1**] Create another new script, or open an existing one if preferred and create a new InputProfile at the top. In that InputProfile, add two new LegacyAxis actions.

LegacyAxis 1 should be called "legacyMouse" and have it's axis set to **AvailableInputAxes.MouseAxis**.

```
public class TestingInputProfile : MonoBehaviour
{
    InputProfile controls = new InputProfile(
        new LegacyAxis("legacyMouse", AvailableInputAxes.MouseAxis)
    );

    Unity Message | 0 references
    public void Update() { }
}
```

LegacyAxis 2 should be called "legacyMove" and have four KeyCode values.

- Positive X set to **KeyCode.D**
- Negative X set to **KeyCode.A**
- Positive Y set to **KeyCode.W**
- Negative Y set to **KeyCode.S**

```
public class TestingInputProfile : MonoBehaviour
{
    InputProfile controls = new InputProfile(
        new LegacyAxis("legacyMouse", AvailableInputAxes.MouseAxis),
        new LegacyAxis("legacyMove", KeyCode.D, KeyCode.A, KeyCode.W, KeyCode.S)
    );

    Unity Message | 0 references
    public void Update() { }
}
```

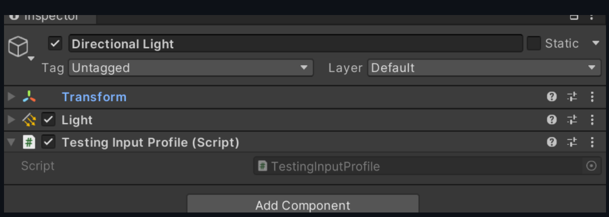
[**Step 2**] Create two **Vector2** variables in Update sharing the same names as the LegacyAxis actions just created. Set the value of both variables to be a call to the input profile's GetAxis function, passing in the axis names.

```
Unity Message | 0 references
public void Update()
{
    Vector2 legacyMouse = controls.GetAxis("legacyMouse");
    Vector2 legacyMove = controls.GetAxis("legacyMove");
}
```

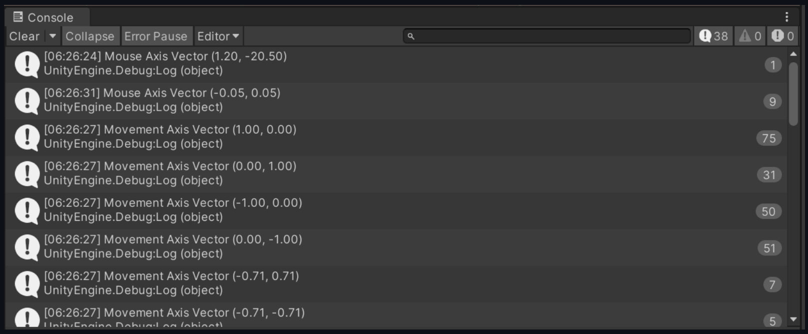
[**Step 3**] For both vectors, create an if-statement that checks if the magnitude exceeds a minimum value. This tutorial will use a minimum threshold of **0.05f** and print the axis value if that threshold is exceeded.

```
public void Update()
{
    Vector2 legacyMouse = controls.GetAxis("legacyMouse");
    Vector2 legacyMove = controls.GetAxis("legacyMove");

    if (legacyMouse.magnitude > 0.05f) Debug.Log($"Mouse Axis Vector {legacyMouse}");

    if (legacyMove.magnitude > 0.05f) Debug.Log($"Movement Axis Vector {legacyMove}");
}
```

[**Step 4**] Attach the script to a GameObject in a scene, if not already done so, and test the button and mouse axes.



Directional Light
Tag Untagged   Layer Default
Testing Input Profile (Script)
Script   TestingInputProfile



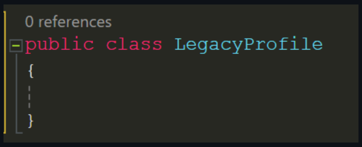
```
[06:26:24] Mouse Axis Vector (1.20, -20.50)
UnityEngine.Debug:Log (object)                      1
[06:26:31] Mouse Axis Vector (-0.05, 0.05)
UnityEngine.Debug:Log (object)                      9
[06:26:27] Movement Axis Vector (1.00, 0.00)
UnityEngine.Debug:Log (object)                     75
[06:26:27] Movement Axis Vector (0.00, 1.00)
UnityEngine.Debug:Log (object)                     31
[06:26:27] Movement Axis Vector (-1.00, 0.00)
UnityEngine.Debug:Log (object)                     50
[06:26:27] Movement Axis Vector (0.00, -1.00)
UnityEngine.Debug:Log (object)                     51
[06:26:27] Movement Axis Vector (-0.71, 0.71)
UnityEngine.Debug:Log (object)                      7
[06:26:27] Movement Axis Vector (-0.71, -0.71)
                                                    5
```

Creating a Custom Input Profile

This section covers the creation of a class similar to **InputProfile**, with features of **PlatformInputProfile** combined.

Despite these classes existing, developers may seek to add or modify them. However, if the goal is to only extend the behaviour of InputProfile then it is recommended to inherit from that class specifically.

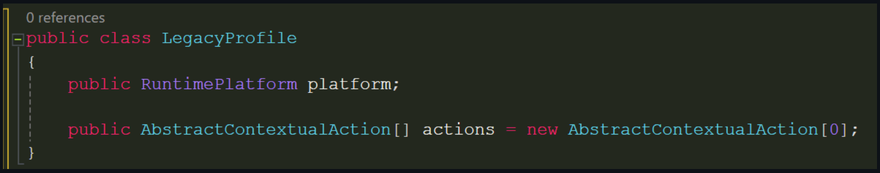
[**Step 1**] Create a class called “LegacyProfile” with no inheritance.



```
0 references
public class LegacyProfile
{

}
```

[**Step 2**] Add two variables at the top of the class:

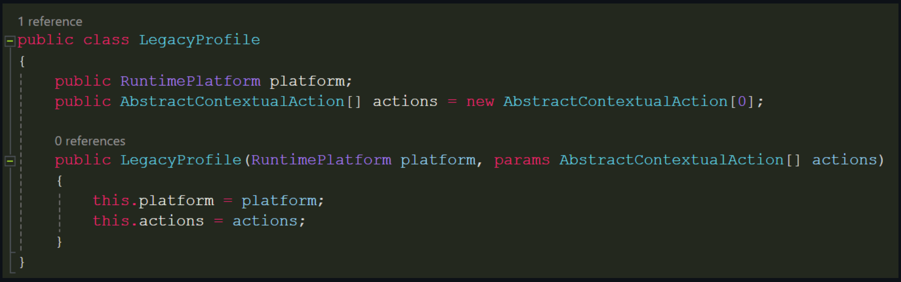
A variable array of **AbstractContextualAction** objects called “actions”, which starts with no objects.
A variable of the **UnityEngine.RuntimePlatform** type, which will be the platform the profile is intended for.



```
0 references
public class LegacyProfile
{
    public RuntimePlatform platform;

    public AbstractContextualAction[] actions = new AbstractContextualAction[0];
}
```

[**Step 3**] Create a new constructor for LegacyProfile, which takes a **RuntimePlatform** variable and an array of **AbstractContextualAction** objects. Prefix the array argument with the **params** keyword so that developers can provide any amount of actions without creating an array to store them in manually.

The **params** keyword will also make assigning to the LegacyProfile.actions array a single variable assignment.

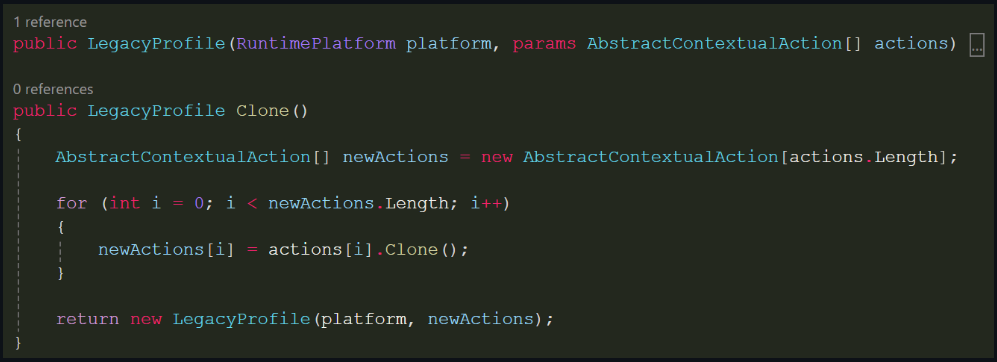


```
1 reference
public class LegacyProfile
{
    public RuntimePlatform platform;
    public AbstractContextualAction[] actions = new AbstractContextualAction[0];

    0 references
    public LegacyProfile(RuntimePlatform platform, params AbstractContextualAction[] actions)
    {
        this.platform = platform;
        this.actions = actions;
    }
}
```

[**Step 4**] Below the constructor, create a Copy method that returns a **LegacyProfile** object.

Create a new **AbstractContextualAction** array at the top of the copy method with the same length as the **LegacyProfile**.actions array, called “newActions”.

Iterate through both arrays, using **AbstractContextualAction.Clone()** to duplicate all objects of the old array and assign them to the newActions array.



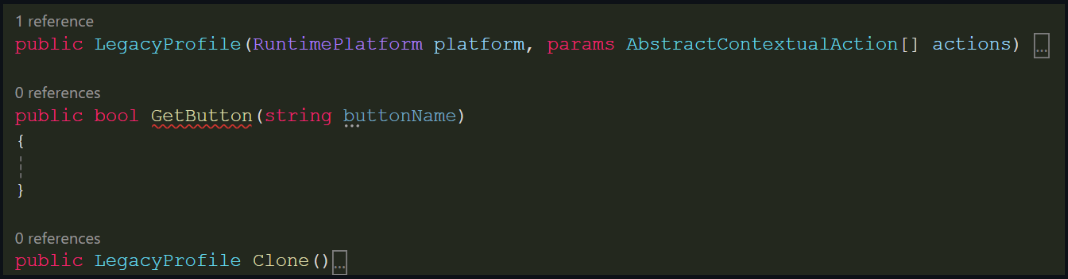
```
1 reference
public LegacyProfile(RuntimePlatform platform, params AbstractContextualAction[] actions) ...

0 references
public LegacyProfile Clone()
{
    AbstractContextualAction[] newActions = new AbstractContextualAction[actions.Length];

    for (int i = 0; i < newActions.Length; i++)
    {
        newActions[i] = actions[i].Clone();
    }

    return new LegacyProfile(platform, newActions);
}
```

When inheriting from **InputProfile**, this would return an **InputProfile** type instead.

[**Step 5**] Although all core members for the primitive information, there are no helper functions.

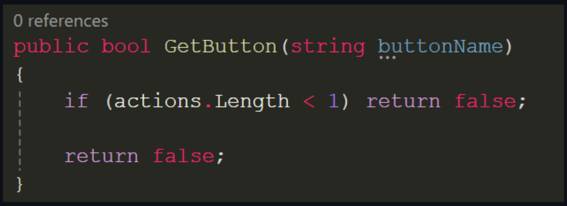
Between the constructor and the clone method, add a new method that returns a bool called "**GetButton**". This method will return if a button is currently held down.

It should only take one argument, a string called "buttonName".



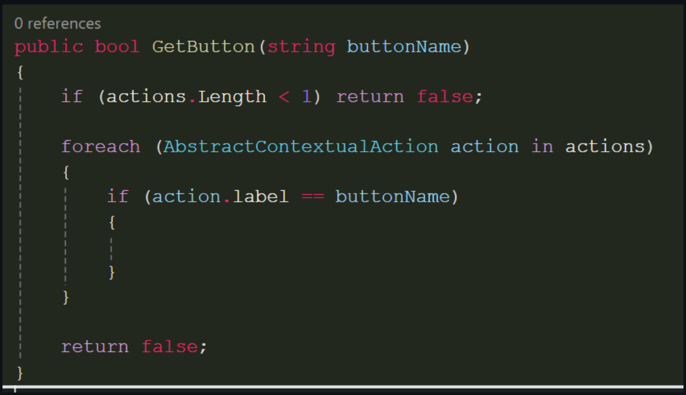
```
1 reference
public LegacyProfile(RuntimePlatform platform, params AbstractContextualAction[] actions) ...

0 references
public bool GetButton(string buttonName)
{

}

0 references
public LegacyProfile Clone()...
```

In the method body, use an if-statement at the start to check if the actions array has any members. If it has less than one, return false. At the bottom of the method, return false anyway (this will handle any fail conditions).



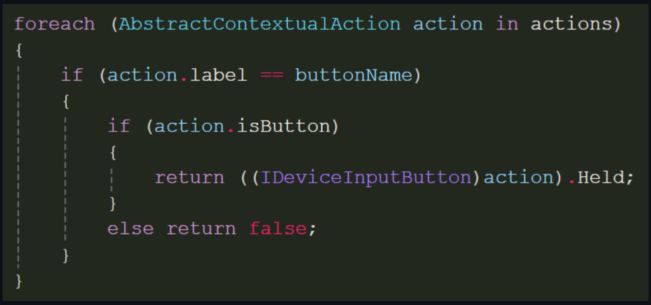
```
0 references
public bool GetButton(string buttonName)
{
    if (actions.Length < 1) return false;

    return false;
}
```

To get a 'true' condition the list of actions needs be looped through, however, **AbstractContextualAction** objects will not return any button value. Create a foreach loop that runs through the actions array, checking if each **AbstractContextualAction**.label matches the buttonName string provided.



```
0 references
public bool GetButton(string buttonName)
{
    if (actions.Length < 1) return false;

    foreach (AbstractContextualAction action in actions)
    {
        if (action.label == buttonName)
        {

        }
    }

    return false;
}
```

Lastly, the button's held state needs to be read. However, casting straight away will not be a safe bet. Add an extra if-statement that reads **AbstractContextualAction**.isButton. If it is unsuccessful,return false (for error handling). Otherwise, wrap a cast to **IDeviceInputButton** inside parenthesis and then return **IDeviceInputButton**.Held.

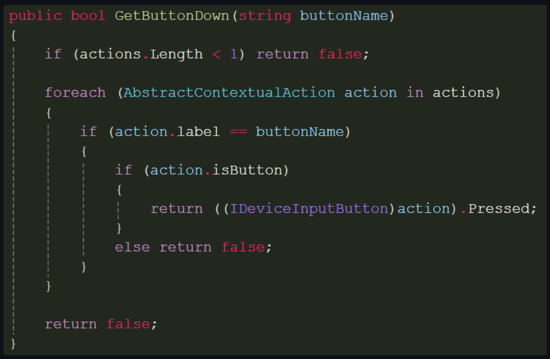


```
foreach (AbstractContextualAction action in actions)
{
    if (action.label == buttonName)
    {
        if (action.isButton)
        {
            return ((IDeviceInputButton)action).Held;
        }
        else return false;
    }
}
```

The return false, if action.isButton is false, is to prevent the loop from iterating through all actions if the action it has found through a matching name isn't the correct type.

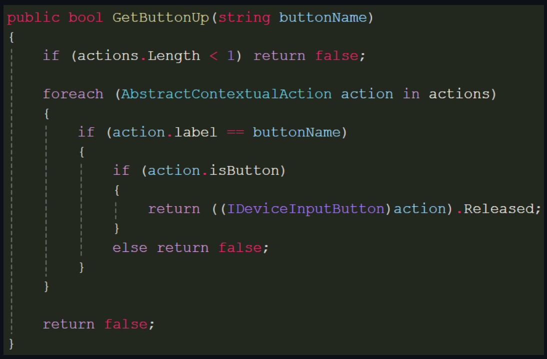
[**Step 6**] Repeat (or copy-paste) **Step 5** twice and make two changes for the next two methods.

Method 1 should be named GetButtonDown. It should return **IDeviceInputButton.Pressed** if successfully read.
Method 2 should be named GetButtonUp. It should return **IDeviceInputButton.Released** if successfully read.



```
public bool GetButtonDown(string buttonName)
{
    if (actions.Length < 1) return false;

    foreach (AbstractContextualAction action in actions)
    {
        if (action.label == buttonName)
        {
            if (action.isButton)
            {
                return ((IDeviceInputButton)action).Pressed;
            }
            else return false;
        }
    }

    return false;
}
```

Get Button Down Method

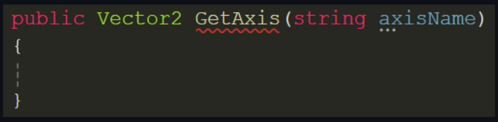


```
public bool GetButtonUp(string buttonName)
{
    if (actions.Length < 1) return false;

    foreach (AbstractContextualAction action in actions)
    {
        if (action.label == buttonName)
        {
            if (action.isButton)
            {
                return ((IDeviceInputButton)action).Released;
            }
            else return false;
        }
    }

    return false;
}
```

Get Button Up Method

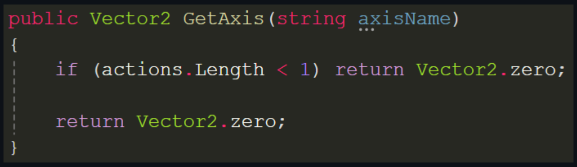
[**Step 7**] Create one last method that returns a **Vector2**, this will be called **GetAxis**.

It should only take one argument, a string called "axisName".



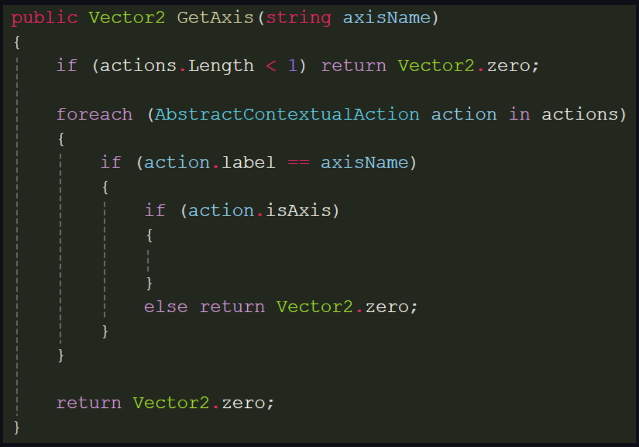
```
public Vector2 GetAxis(string axisName)
{

}
```

Add a check at the top of the method to see if there are any actions in the actions list. If there aren't any, return **Vector2.zero** as a failsafe then add a return **Vector2.zero** at the end of the method too for any fail cases.



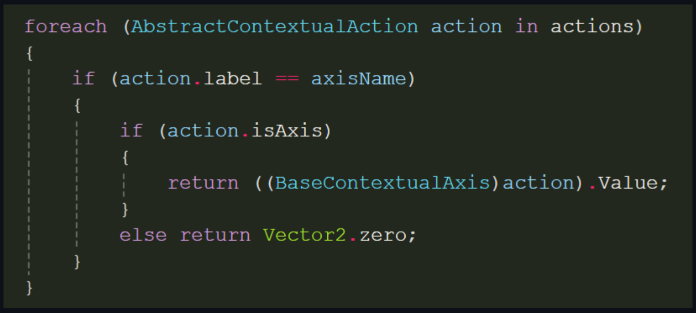
```
public Vector2 GetAxis(string axisName)
{
    if (actions.Length < 1) return Vector2.zero;

    return Vector2.zero;
}
```

Add a foreach loop that iterates over all actions between these, with the label check seen in step 5. Inside the label check, add a check that reads **AbstractContextualAction**.isAxis and return **Vector2.zero** if this is false.



```
public Vector2 GetAxis(string axisName)
{
    if (actions.Length < 1) return Vector2.zero;

    foreach (AbstractContextualAction action in actions)
    {
        if (action.label == axisName)
        {
            if (action.isAxis)
            {

            }
            else return Vector2.zero;
        }
    }

    return Vector2.zero;
}
```

Lastly, if isAxis is true, wrap a cast to **BaseContextualAxis** in parenthesis and return **BaseContextualAxis** .Value.



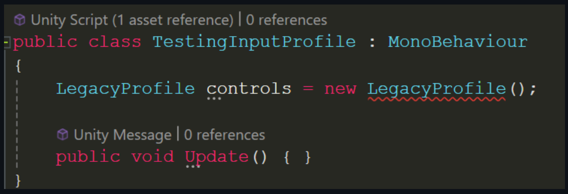
```
foreach (AbstractContextualAction action in actions)
{
    if (action.label == axisName)
    {
        if (action.isAxis)
        {
            return ((BaseContextualAxis)action).Value;
        }
        else return Vector2.zero;
    }
}
```

Testing the Custom Input Profile

For developers who have been following all class tutorials, this tutorial uses the **LegacyButton** and **LegacyAxis** classes.

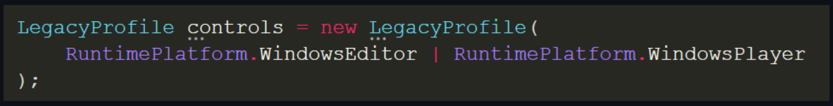
For developers who are only reading this tutorial, **ContextualButton** and **ContextualAxis** (or equivalents) for a preferred input handler will work just as well.

[**Step 1**] Create another new script, or open an existing one if preferred and create a new LegacyProfile at the top.



```
Unity Script (1 asset reference) | 0 references
public class TestingInputProfile : MonoBehaviour
{
    LegacyProfile controls = new LegacyProfile();

    Unity Message | 0 references
    public void Update() { }
}
```

[**Step 2**] Add a platform of choice, or the platform you are currently using, from **UnityEngine.RuntimePlatform** first.



```
LegacyProfile controls = new LegacyProfile(
    RuntimePlatform.WindowsEditor | RuntimePlatform.WindowsPlayer
);
```

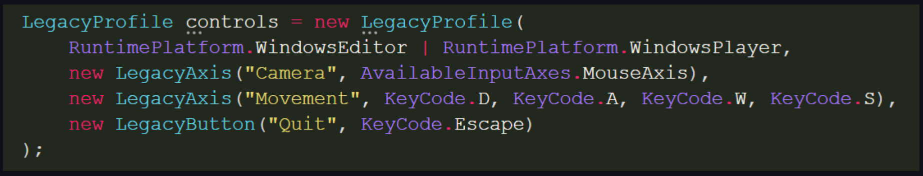
This uses the bitwise "or" operator to combine two runtime platform values.
Although the tutorial is done in Unity Editor, being able to run code in compiled buildshelps.

[**Step 3**] Add any amount of LegacyButton and LegacyAxis actions, this tutorial will create a handful for testing purposes:

A **LegacyAxis** called Camera set to **AvailableInputAxes.MouseAxis**.

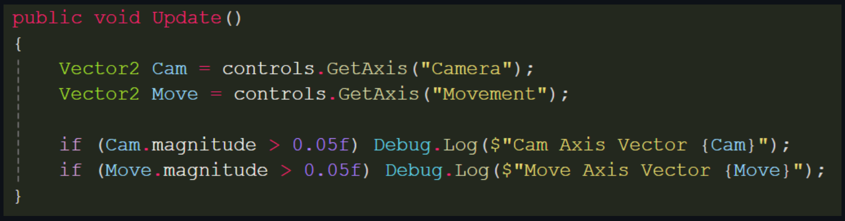
A **LegacyAxis** called "Movement" for the WASD keys, set to **AvailableInputAxes.ButtonAxis**.

A **LegacyButton** called "Quit" set to **KeyCode.Escape**.



```
LegacyProfile controls = new LegacyProfile(
    RuntimePlatform.WindowsEditor | RuntimePlatform.WindowsPlayer,
    new LegacyAxis("Camera", AvailableInputAxes.MouseAxis),
    new LegacyAxis("Movement", KeyCode.D, KeyCode.A, KeyCode.W, KeyCode.S),
    new LegacyButton("Quit", KeyCode.Escape)
);
```

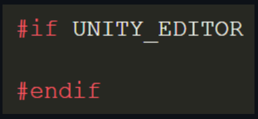
[**Step 4**] For both **LegacyAxis** objects, create an if-statement that checks if the magnitude exceeds a minimum value in the Update method. This tutorial will use a minimum threshold of **0.05f** that prints the axis value if exceeded.



```
public void Update()
{
    Vector2 Cam = controls.GetAxis("Camera");
    Vector2 Move = controls.GetAxis("Movement");

    if (Cam.magnitude > 0.05f) Debug.Log($"Cam Axis Vector {Cam}");
    if (Move.magnitude > 0.05f) Debug.Log($"Move Axis Vector {Move}");
}
```

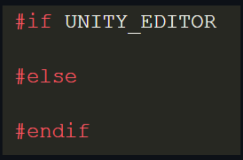
The next few steps use a C# feature called Preprocessor Directives.

This guide will be using directives that check for the presence of defined values so that the Unity Editor player pauses Itself when Quit is pressed. When the test project is built, the "Quit" button will quit the application instead. To only include code that runs in the Unity Editor, wrap the desired code within an if (or if-else) directive.



```
#if UNITY_EDITOR

#endif
```

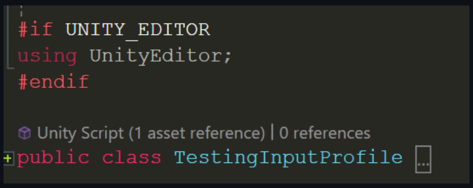
An "if" directive



```
#if UNITY_EDITOR

#else

#endif
```

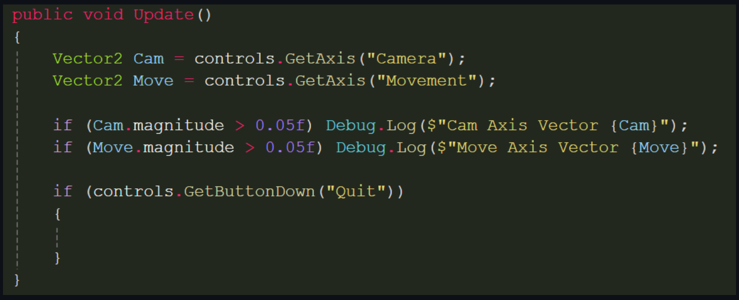
An "if-else" directive

[**Step 5**] At the top of the script, below all using declarations, add a preprocessor if-statement, which checks if **UNITY_EDITOR** is defined. Inside this, add a using directive for the UnityEditor namespace.



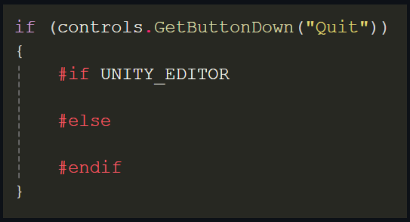
```
#if UNITY_EDITOR
using UnityEditor;
#endif

Unity Script (1 asset reference) | 0 references
public class TestingInputProfile ...
```

[**Step 6**] Below both if-statements in the Update function, add an if-statement that checks if the “Quit” action has been pressed by using the new **LegacyButton.GetButtonDown** method.



```
public void Update()
{
    Vector2 Cam = controls.GetAxis("Camera");
    Vector2 Move = controls.GetAxis("Movement");

    if (Cam.magnitude > 0.05f) Debug.Log($"Cam Axis Vector {Cam}");
    if (Move.magnitude > 0.05f) Debug.Log($"Move Axis Vector {Move}");

    if (controls.GetButtonDown("Quit"))
    {

    }
}
```

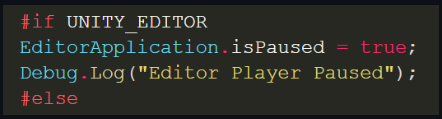
[**Step 7**] In the body of the new method, include an if-else preprocessor directive. This should also be dependent on if the **UNITY_EDITOR** value has been defined.



```
if (controls.GetButtonDown("Quit"))
{
    #if UNITY_EDITOR

    #else

    #endif
}
```

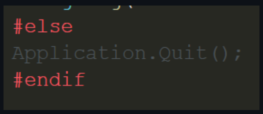
[**Step 8**] Two separate behaviours need to take place:

For the body of the if statement, set **EditorApplication.isPaused** to true. This will pause the Editor Player. Add a call to **Debug.Log** that prints out a message saying the Editor Player has been paused.



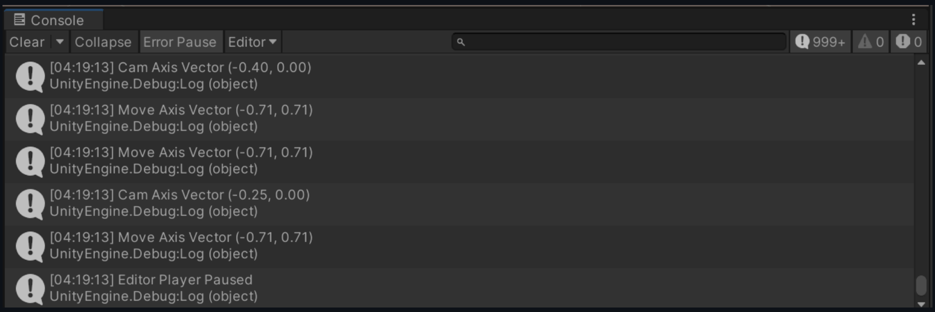
```
#if UNITY_EDITOR
EditorApplication.isPaused = true;
Debug.Log("Editor Player Paused");
#else
```

For the body of the else statement, call the **Application.Quit** method.
This is greyed out because **UNITY_EDITOR** is always defined when developing with Unity Editor.



```
#else
Application.Quit();
#endif
```

[**Step 9**] Test the new controls and see if they print as expected. Your Developer Console should look like this:



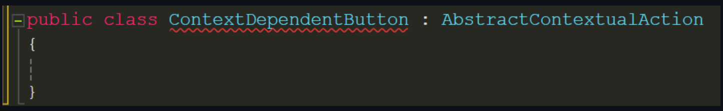
```
[04:19:13] Cam Axis Vector (-0.40, 0.00)
UnityEngine.Debug:Log (object)
[04:19:13] Move Axis Vector (-0.71, 0.71)
UnityEngine.Debug:Log (object)
[04:19:13] Move Axis Vector (-0.71, 0.71)
UnityEngine.Debug:Log (object)
[04:19:13] Cam Axis Vector (-0.25, 0.00)
UnityEngine.Debug:Log (object)
[04:19:13] Move Axis Vector (-0.71, 0.71)
UnityEngine.Debug:Log (object)
[04:19:13] Editor Player Paused
UnityEngine.Debug:Log (object)
```

Creating a Custom Action

This section covers creating a class for an action from start to finish, inheriting from **AbstractContextualAction** directly and building up all necessary functionality from scratch. This is so testing can still work with InputProfile or LegacyProfile. Determining the type of an object is covered as well, so that developers can include properties like isAxis and isButton found in the **AbstractContextualAction** class.

Specifically, this will show the prcoess of creating a button that returns true for multiple behaviours, especially based on whether behaviour altering keys, such as Control, Alt or similar are active.

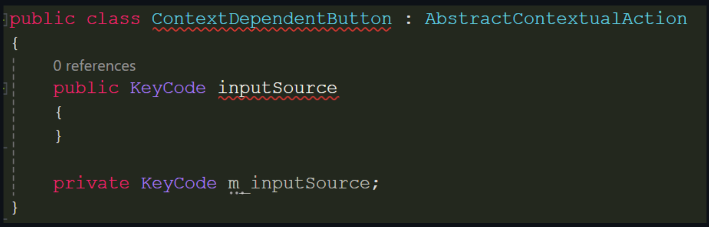
[**Step 1**] Create a class called "ContextDependentButton" that inherits from **AbstractContextualAction**.



```
public class ContextDependentButton : AbstractContextualAction
{

}
```

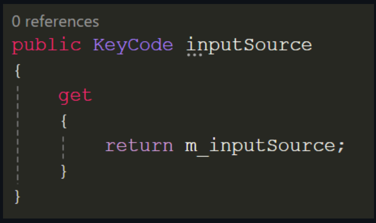
The error highlight is because the Clone method needs implementing, which is done last.

[**Step 2**] Create a KeyCode property called inputSource, as well as a KeyCode variable called m_inputSource.
The property should be public, with no accessors yet, while the variable should remain a private variable.



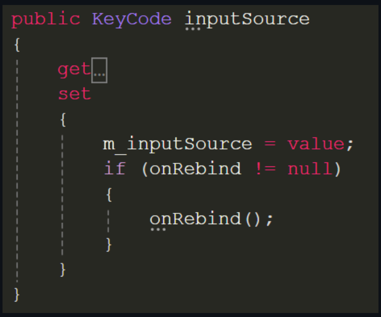
```
public class ContextDependentButton : AbstractContextualAction
{
    0 references
    public KeyCode inputSource
    {
    }

    private KeyCode m_inputSource;
}
```

Create a Get-Accessor that returns m_inputSource.



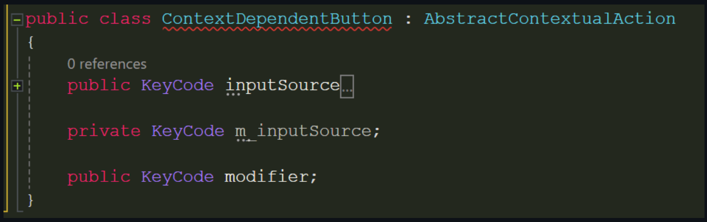
```
0 references
public KeyCode inputSource
{
    get
    {
        return m_inputSource;
    }
}
```

Create a set accessor within the property, which sets the variable m_inputSource to the value provided.
Add an if-statement to check if there's an onRebind callback and invoke it if there is one.



```
public KeyCode inputSource
{
    get...
    set
    {
        m_inputSource = value;
        if (onRebind != null)
        {
            onRebind();
        }
    }
}
```

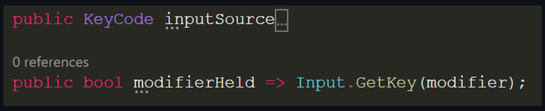
[**Step 3**] Add a public variable, called "modifier". This key, when pressed, changes the context of the button's state.



```
public class ContextDependentButton : AbstractContextualAction
{
    0 references
    public KeyCode inputSource...

    private KeyCode m_inputSource;

    public KeyCode modifier;
}
```

[**Step 4**] Create a public property that returns a boolean,called “modifierHeld”. This only needs to see if the key is held.

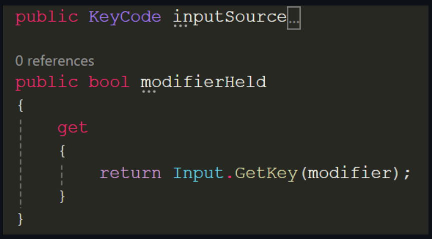
With a lambda expression:



```
public KeyCode inputSource…

0 references
public bool modifierHeld => Input.GetKey(modifier);
```

(Used for this tutorial)

With a traditional get-set accessor:

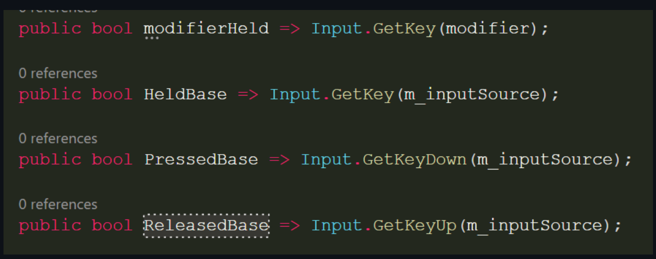


```
public KeyCode inputSource…

0 references
public bool modifierHeld
{
    get
    {
        return Input.GetKey(modifier);
    }
}
```

[**Step 5**] There needs to be six public properties that only return booleans, using your preferred method (see above).

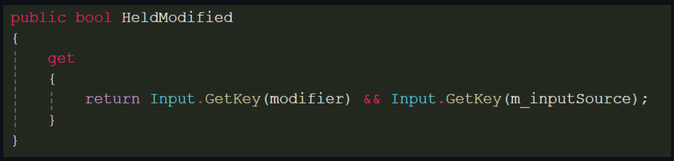
Create the first three properties:

- One called “HeldBase” that returns the value of **Input.GetKey**, using m_inputSource.
- Another called “PressedBase” that returns the value of **Input.GetKeyDown**, using m_inputSource.
- A last one called “ReleasedBase” that returns the value of **Input.GetKeyUp**, using m_inputSource.



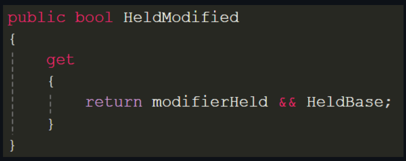
```
public bool modifierHeld => Input.GetKey(modifier);

0 references
public bool HeldBase => Input.GetKey(m_inputSource);

0 references
public bool PressedBase => Input.GetKeyDown(m_inputSource);

0 references
public bool ReleasedBase => Input.GetKeyUp(m_inputSource);
```

The next three properties depend on preference, so the two main ways will be presented and the preferred method can be chosen. Developers are encouraged to choose their preference, however this tutorial will use the second method for brevity.



```
public bool HeldModified
{
    get
    {
        return Input.GetKey(modifier) && Input.GetKey(m_inputSource);
    }
}
```

Method 1: Directly call both the checks for the modifier and input source keys. This means control passes through fewer steps before returning, however, the code is longer and a bit harder to read.

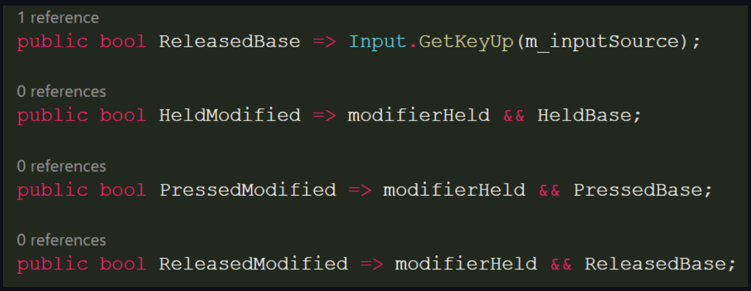


```
public bool HeldModified
{
    get
    {
        return modifierHeld && HeldBase;
    }
}
```

Method 2: Read modifierHeld and HeldBase. This means control passes through an extra step twice before returning, however the code is much more descriptive and easier to read.

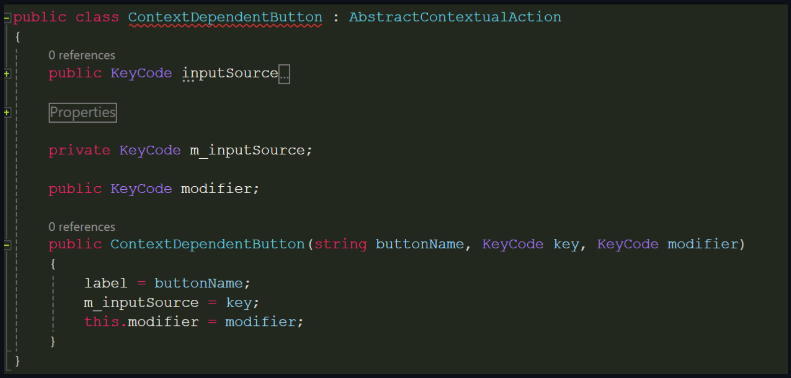
Create these next three properties with your preferred method. These are the last properties required.

- “HeldModified” which returns if the input source key **and** modifier key are held down.
- “PressedModified” which returns if the input source key is pressed **and** the modifier key is held down.
- “ReleasedModified” which returns if the input source key is released **and** the modifier key is held down.



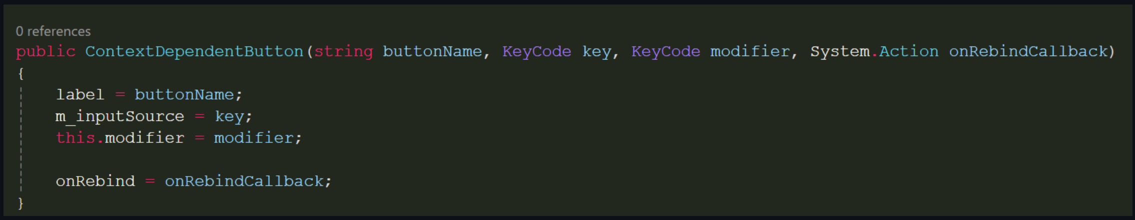
```
1 reference
public bool ReleasedBase => Input.GetKeyUp(m_inputSource);

0 references
public bool HeldModified => modifierHeld && HeldBase;

0 references
public bool PressedModified => modifierHeld && PressedBase;

0 references
public bool ReleasedModified => modifierHeld && ReleasedBase;
```

[**Step 6**] Create a constructor at the end of the class that takes a string called buttonName and two KeyCodes. The first KeyCode should be called “key”, while the second should be called “modifier”. Set the value of label, m_inputSource and modifier to the arguments passed through the constructor.



```
public class ContextDependentButton : AbstractContextualAction
{
    0 references
    public KeyCode inputSource...

    Properties

    private KeyCode m_inputSource;

    public KeyCode modifier;

    0 references
    public ContextDependentButton(string buttonName, KeyCode key, KeyCode modifier)
    {
        label = buttonName;
        m_inputSource = key;
        this.modifier = modifier;
    }
}
```

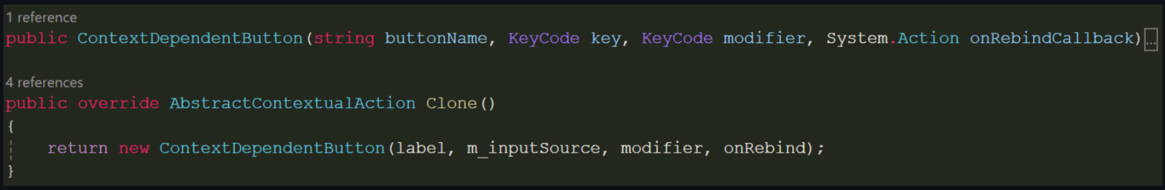
[**Step 7**] Create one more constructor with the same parameters, however, add a System.Action at the end. This should be called “onRebindCallback”, which will be the callback triggered if the keybind changes. The same values should be set as above, with onRebind being assigned the new value.



```
0 references
public ContextDependentButton(string buttonName, KeyCode key, KeyCode modifier, System.Action onRebindCallback)
{
    label = buttonName;
    m_inputSource = key;
    this.modifier = modifier;

    onRebind = onRebindCallback;
}
```

OnRebind is covered in this section as prior sections focused on inheritance and extension. This section covers all facets.

[**Step 8**] Lastly, create a **public override AbstractContextualAction** method called ‘Clone’ below the constructor. This method only needs to call the second constructor, rather than the first one. This is how most of the Clone methods in the optional classes work, as those constructors provide the most data to the new duplicate.

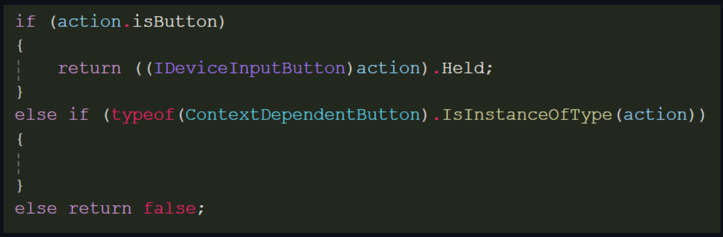


```
1 reference
public ContextDependentButton(string buttonName, KeyCode key, KeyCode modifier, System.Action onRebindCallback)...

4 references
public override AbstractContextualAction Clone()
{
    return new ContextDependentButton(label, m_inputSource, modifier, onRebind);
}
```

Identifying a Context Dependent Button

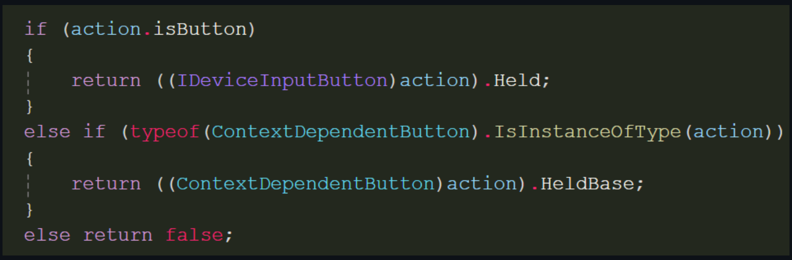
This tutorial modifies the tutorial-made **LegacyProfile** class. **Modify other classes at your own risk.**

[**Step 1**] Go to the GetButton method in either the default or your own Input Profile class. Between the if-else check for whether the action is a button, add an else-if case. Start with a **typeof** keyword that gets an object representing **ContextDependentButton**, then append the call to **IsInstanceOfType** to the end. The required object should be the action being iterated through.



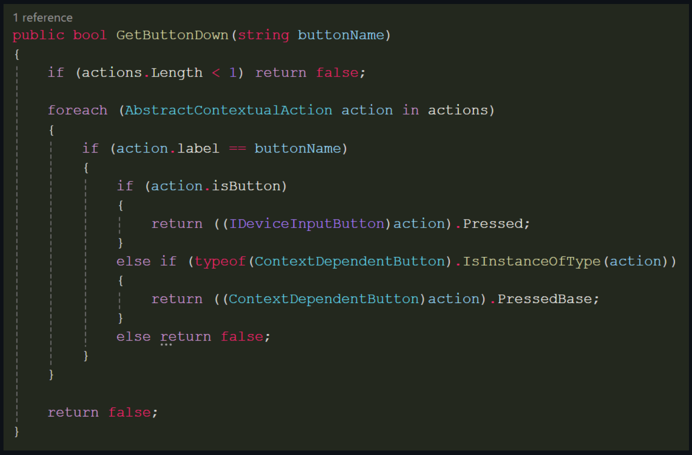
```
if (action.isButton)
{
    return ((IDeviceInputButton)action).Held;
}
else if (typeof(ContextDependentButton).IsInstanceOfType(action))
{

}
else return false;
```

[**Step 2**] Cast the action to the **ContextDependentButton** type and wrap that in parenthesis.
Then return the value of **HeldBase** from the wrapped type-cast.

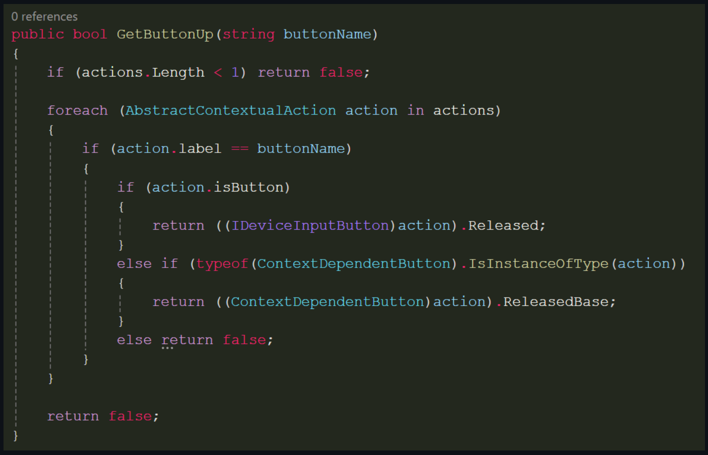


```
if (action.isButton)
{
    return ((IDeviceInputButton)action).Held;
}
else if (typeof(ContextDependentButton).IsInstanceOfType(action))
{
    return ((ContextDependentButton)action).HeldBase;
}
else return false;
```

[**Step 3**] Add this same else-if statement to **GetButtonDown** and **GetButtonUp**. For Both methods, change the returned values to be those of **PressedBase** and **ReleasedBase** respectively.



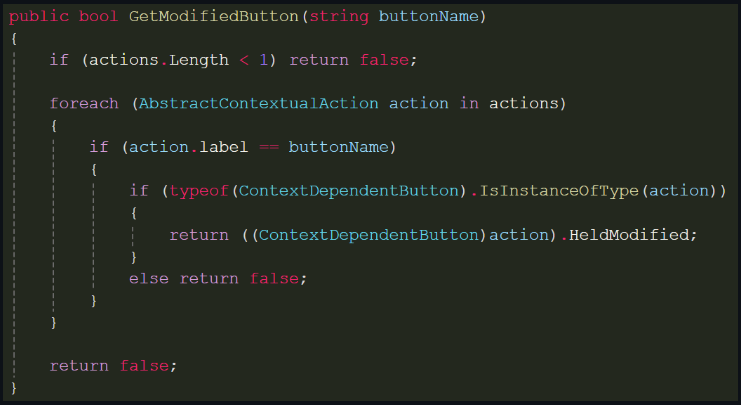
```
1 reference
public bool GetButtonDown(string buttonName)
{
    if (actions.Length < 1) return false;

    foreach (AbstractContextualAction action in actions)
    {
        if (action.label == buttonName)
        {
            if (action.isButton)
            {
                return ((IDeviceInputButton)action).Pressed;
            }
            else if (typeof(ContextDependentButton).IsInstanceOfType(action))
            {
                return ((ContextDependentButton)action).PressedBase;
            }
            else return false;
        }
    }

    return false;
}
```



```
0 references
public bool GetButtonUp(string buttonName)
{
    if (actions.Length < 1) return false;

    foreach (AbstractContextualAction action in actions)
    {
        if (action.label == buttonName)
        {
            if (action.isButton)
            {
                return ((IDeviceInputButton)action).Released;
            }
            else if (typeof(ContextDependentButton).IsInstanceOfType(action))
            {
                return ((ContextDependentButton)action).ReleasedBase;
            }
            else return false;
        }
    }

    return false;
}
```

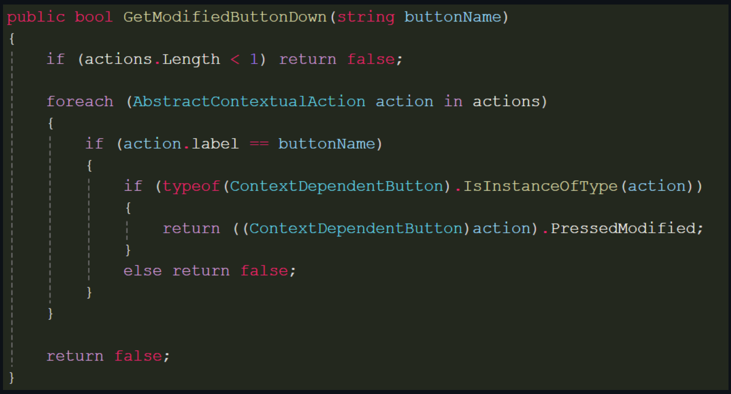
[**Step 4**] Copy the button methods and rename them as "**GetModifiedButton**", "**GetModifiedButtonDown**" and "**GetModifiedButtonUp**".

Then Inside these methods, remove the else-if statement and change the returned properties. HeldBase should be **HeldModified** instead, PressedBase turns to **PressedModified** and ReleasedBase turns to **ReleasedModified**.



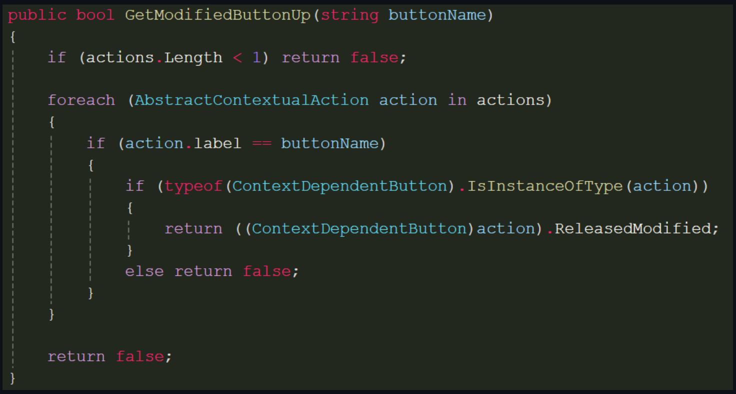
```
public bool GetModifiedButton(string buttonName)
{
    if (actions.Length < 1) return false;

    foreach (AbstractContextualAction action in actions)
    {
        if (action.label == buttonName)
        {
            if (typeof(ContextDependentButton).IsInstanceOfType(action))
            {
                return ((ContextDependentButton)action).HeldModified;
            }
            else return false;
        }
    }

    return false;
}
```

GetModifiedButton - Use HeldModified



```
public bool GetModifiedButtonDown(string buttonName)
{
    if (actions.Length < 1) return false;

    foreach (AbstractContextualAction action in actions)
    {
        if (action.label == buttonName)
        {
            if (typeof(ContextDependentButton).IsInstanceOfType(action))
            {
                return ((ContextDependentButton)action).PressedModified;
            }
            else return false;
        }
    }

    return false;
}
```

GetModifiedButtonDown - Use PressedModified



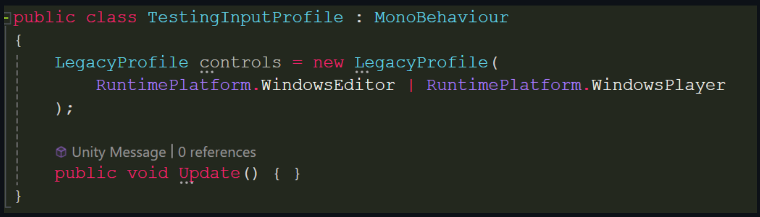
```
public bool GetModifiedButtonUp(string buttonName)
{
    if (actions.Length < 1) return false;

    foreach (AbstractContextualAction action in actions)
    {
        if (action.label == buttonName)
        {
            if (typeof(ContextDependentButton).IsInstanceOfType(action))
            {
                return ((ContextDependentButton)action).ReleasedModified;
            }
            else return false;
        }
    }

    return false;
}
```

GetModifiedButtonUp - Use ReleasedModified

Testing the Custom Action

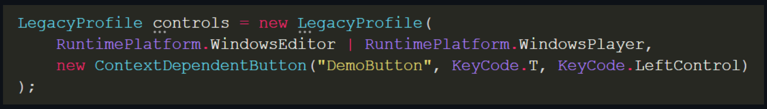
[**Step 1**] Create a new script, or open an existing one and create an variable of the modifed input profile class at the top.

If you have also modfied LegacyProfile, include the desired or tutorial RuntimePlatform variable.



```
public class TestingInputProfile : MonoBehaviour
{
    LegacyProfile controls = new LegacyProfile(
        RuntimePlatform.WindowsEditor | RuntimePlatform.WindowsPlayer
    );

    Unity Message | 0 references
    public void Update() { }
}
```

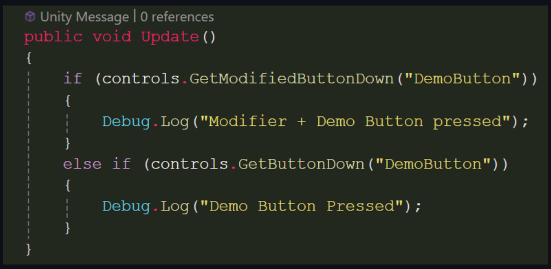
[**Step 2**] Create a new ContextDependentButton in the profile called “DemoButton” with a key and modifier of your choice. This tutorial will use **T** as the main key, with the Left Control Key as the modifier.



```
LegacyProfile controls = new LegacyProfile(
    RuntimePlatform.WindowsEditor | RuntimePlatform.WindowsPlayer,
    new ContextDependentButton("DemoButton", KeyCode.T, KeyCode.LeftControl)
);
```

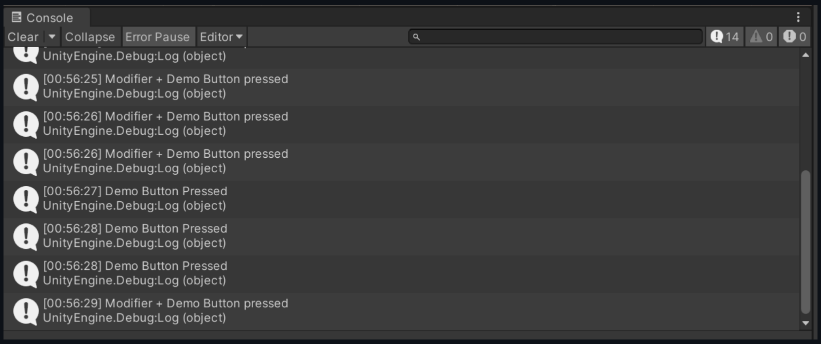
[**Step 3**] In the update method, create an if statement with two cases. Both should print out a message to the console.

The first case should try read DemoControl’s PressedModified state with **GetModifiedButtonDown**.
The second case should try read DemoControl’s PressedBase state with **GetButtonDown**.



```
Unity Message | 0 references
public void Update()
{
    if (controls.GetModifiedButtonDown("DemoButton"))
    {
        Debug.Log("Modifier + Demo Button pressed");
    }
    else if (controls.GetButtonDown("DemoButton"))
    {
        Debug.Log("Demo Button Pressed");
    }
}
```

[**Step 4**] Test the new control and modifier. Your Developer Console should look like this:



```
UnityEngine.Debug:Log (object)
[00:56:25] Modifier + Demo Button pressed
UnityEngine.Debug:Log (object)
[00:56:26] Modifier + Demo Button pressed
UnityEngine.Debug:Log (object)
[00:56:26] Modifier + Demo Button pressed
UnityEngine.Debug:Log (object)
[00:56:27] Demo Button Pressed
UnityEngine.Debug:Log (object)
[00:56:28] Demo Button Pressed
UnityEngine.Debug:Log (object)
[00:56:28] Demo Button Pressed
UnityEngine.Debug:Log (object)
[00:56:29] Modifier + Demo Button pressed
UnityEngine.Debug:Log (object)
```

Future Content - A Temporary Notice

This Readme is still, unfortuantely, unfinished. It has taken the better part of two consistent weeks work and there are some final nuances to cover. I do not like leaving work unfinished, but here’s what’s explicitly missing and why.

- LegacyProfile doesn’t currently also have a tutorial step for implementing GetAction, this was caused by the method being added when it noticed that rebinding an action was significantly harder for the end-user during the creation of the Custom Action tutorial.
- **Creating a Custom Action** currently doesn’t cover key rebindings, this was due to the API changes necessary and the necessity to start rewriting parts of the manual to fit neatly onto paged sections was a headache.

These pages will be done as soon as it is possible, however, obligations in real life have come up and it is with deep regret that this module be open-sourced without these sections yet. This has been the longest readme I’ve written at 26 pages.