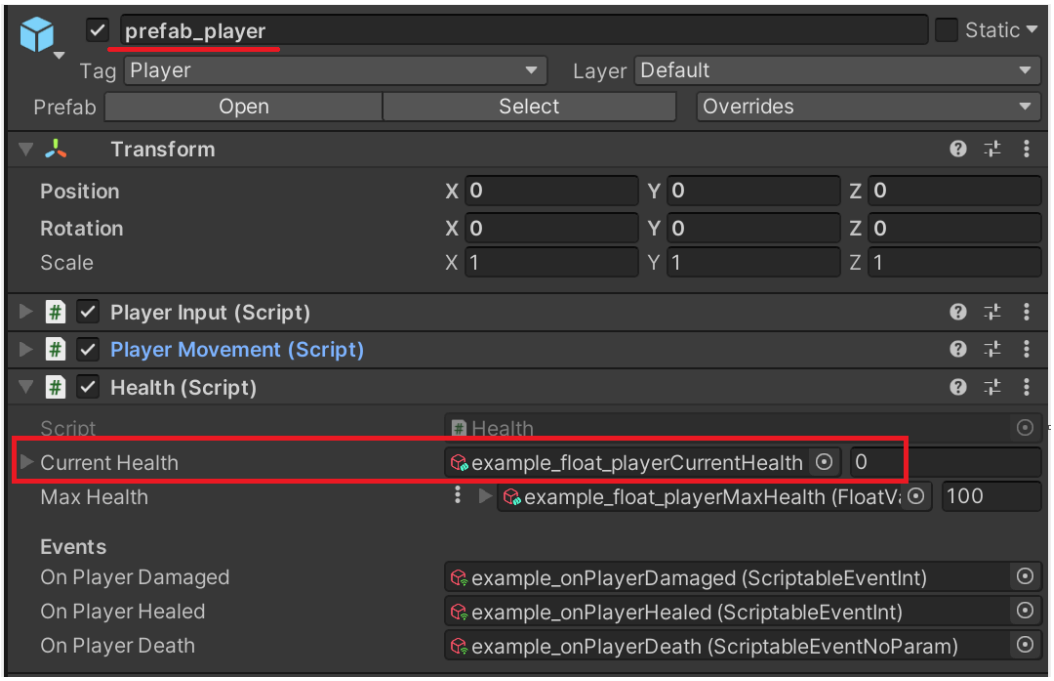
1.Scriptable Variables

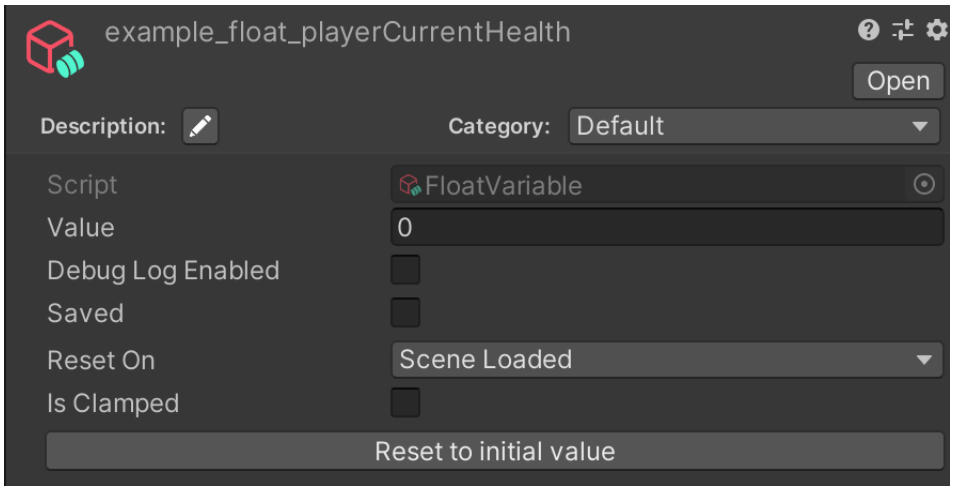
Table of Content

Direct Access

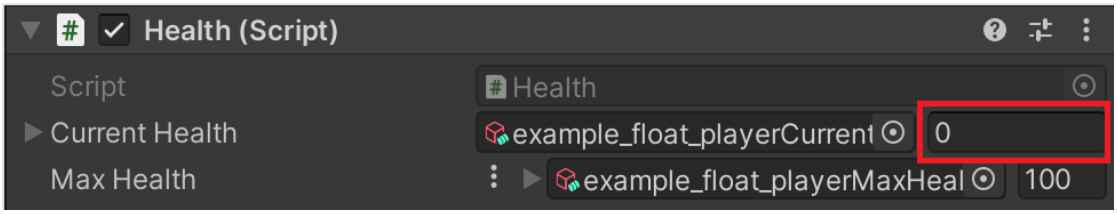
If you click on the **prefab_player** in the scene hierarchy**,** you can see that he has a **Health** component. This component has a reference to a scriptable variable float called “example_float_playerCurrentHealth”.



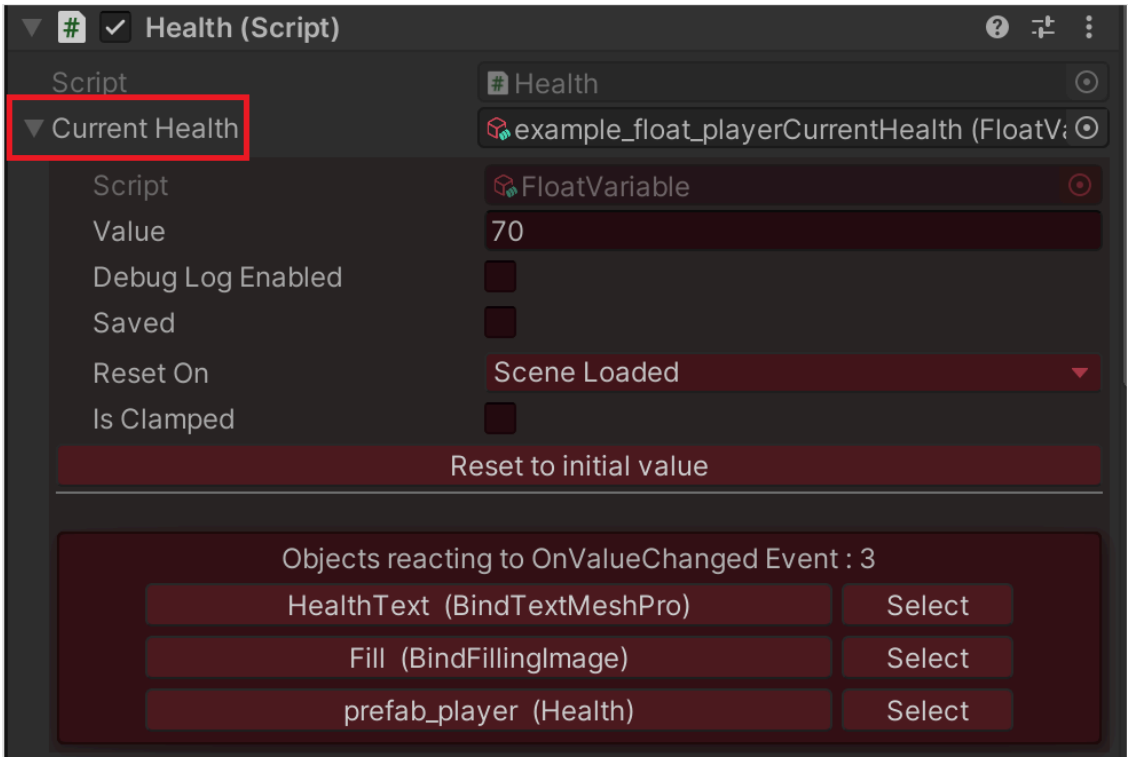
Play the game, then select the **example_float_playerCurrentHealth** variable in your project and change its value in the inspector. You can directly modify the value of the scriptable variable in the inspector window at runtime**.**



You can also directly modify the variables from the **Health** component itself:

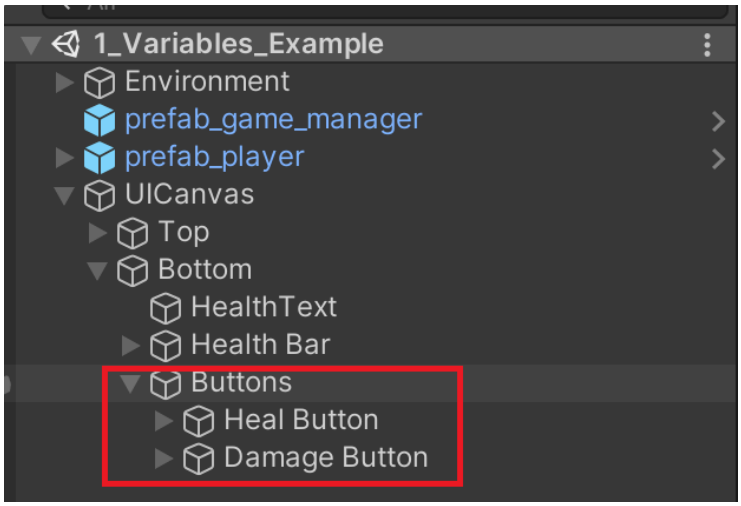


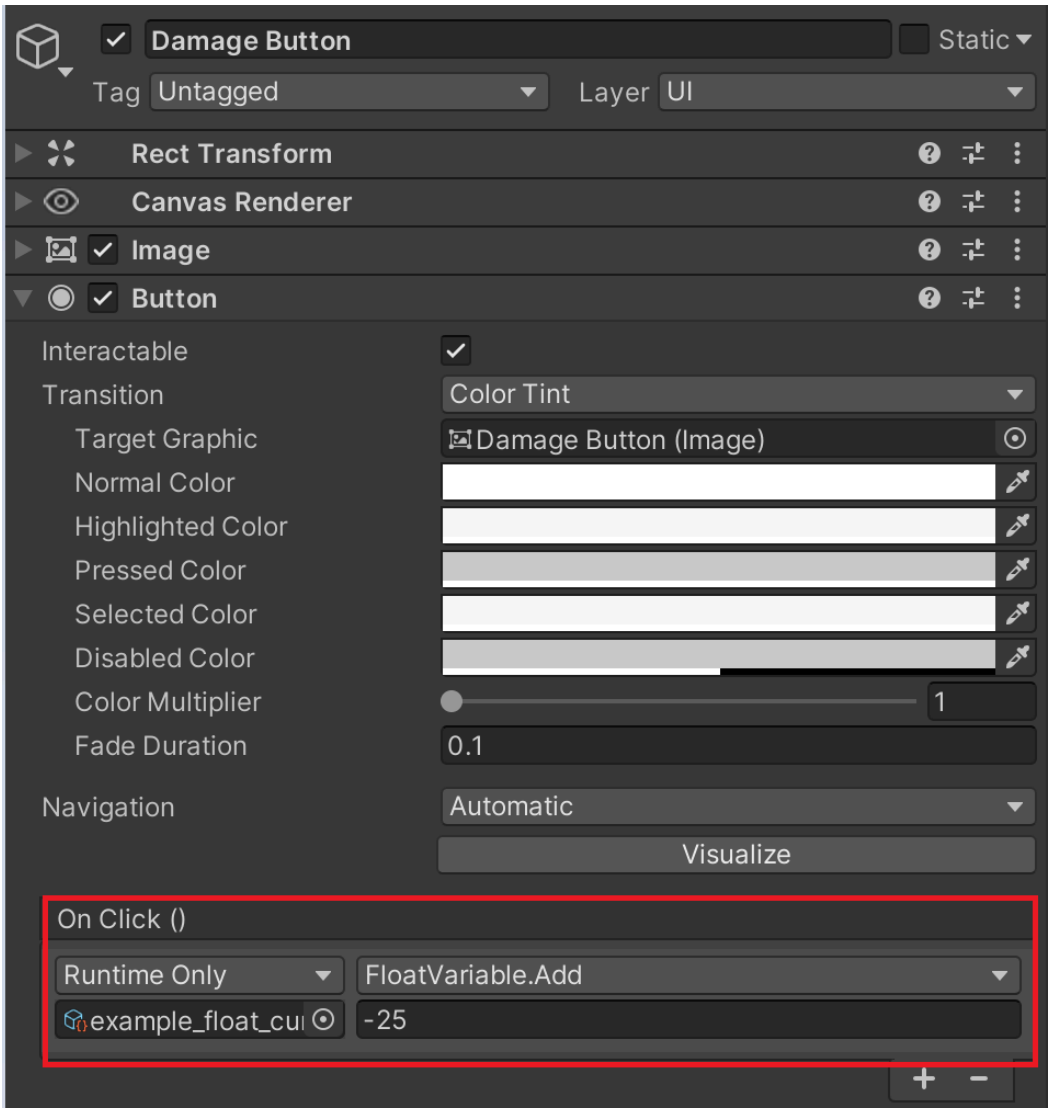
Moreover, by clicking on the small arrow you can expand the inspector showing the Scriptable Variable parameters. You can also change them at runtime:



While still in play mode, click on the **Damage** button or the **Heal** button, you can affect the health directly. Let's look at the methods tied to these buttons by finding them in the hierarchy.

UICanvas/Bottom/Buttons





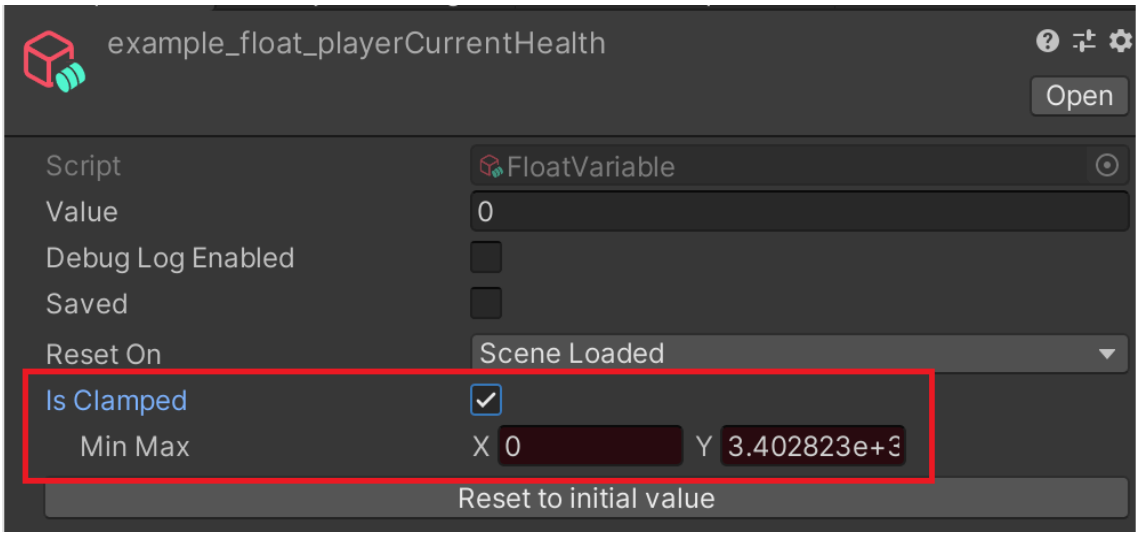
Traditionally, we would call Player->TakeDamage() or Player->Heal, but we can be more direct. Instead, we **directly access** the scriptable object variable and add a value; using the Add() public method from the editor.

It's very direct, and you don't always want that, but sometimes you do (especially when making Hyper casual games). For things like increasing the level index, or other simple things, it can be useful to access them directly.

In the Editor, ScriptableVariables automatically reset to their initial value (the value in the inspector before entering play mode) when exiting play mode.

The FloatVariable and the IntVariable have 2 extra properties:

- **IsClamped**: enable the variable to be clamped.
- **Min Max**: the minimum and maximum values that it will be clamped to.



Go ahead and set IsClamped to true on the playerCurrentHealth variable. As you can see, the minimum is set to 0. If you go into play mode and click on the **Damage button** or try to set the health manually in the inspector, you can observe that the health will not go below 0.

OnValueChanged Event

You can **bind callback methods** when the variable value changes.

Peek at the **Health.cs** class. As you can see, you can bind yourself to a variable. You don't need to constantly check in Update() for its value, you can simply subscribe to the **OnValueChanged** event that is called every time the value changes.

```
private void Start()
{
    _currentHealth.Value = _maxHealth.Value;
    _currentHealth.OnValueChanged += OnHealthChanged;
}

private void OnDestroy()
{
    _currentHealth.OnValueChanged -= OnHealthChanged;
}

private void OnHealthChanged(float newValue)
{
    var diff = newValue - _currentHealth.PreviousValue;
    if (diff<0)
    {
        OnDamaged(Mathf.Abs(diff));
    }
    else
    {
        OnHealed(diff);
    }
}
```

If you are familiar with the UniRx library, it works the same as the reactive property. In SOAP it acts as a reactive property that can also solve dependencies by being a scriptable object.

Note: do not forget to unsubscribe to the event in OnDestroy() or OnDisable(). As the scriptable variable is an asset, the event is still triggered in the inspector and can trigger your callback methods, displaying an error message.

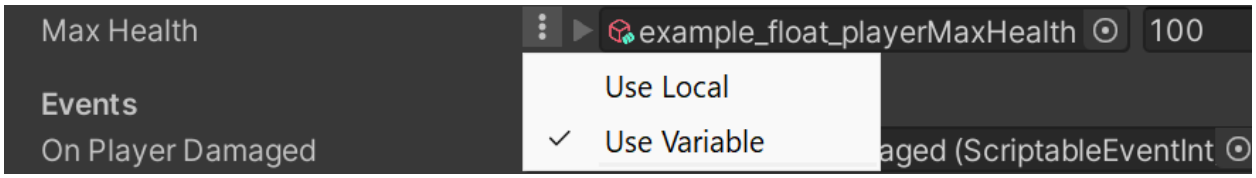
The field **previous value** of a variable can be useful in some cases, like in this example to determine whether the player has been healed or if it has taken damage.

Variable References

You might have noticed that _maxHealth's type is a **FloatReference** instead of a FloatVariable. It allows you to choose between a local value or scriptable variable.

This can be useful if you have multiple objects that share the same components, let's say different cars that share the same controller. Each car can have a reference to a FloatReference for the speed. The player car will choose a FloatVariable and the other cars can just have a local variable.

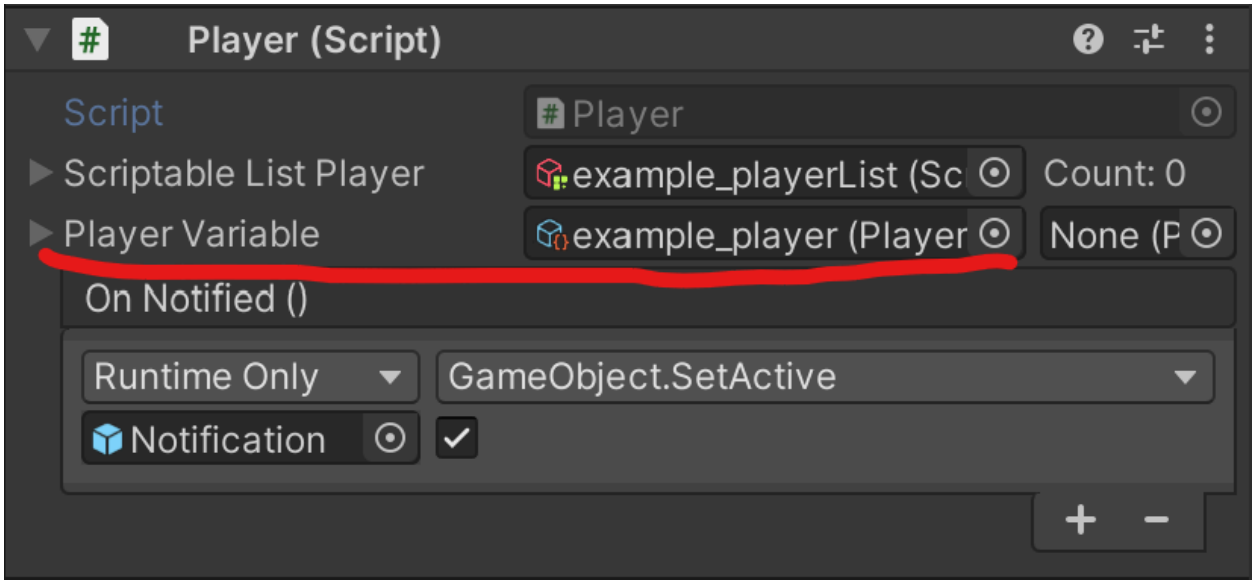
```
[SerializeField] private FloatReference _maxHealth = null;
```



All default types included in Soap have their corresponding reference scripts.

Solving Dependencies

In this scene, I want to illustrate how SOAP can help you solve dependencies with 2 simple examples. For the first example, select the "**prefab_player**" and check its **Player** component. You can see that there is a reference to a SV of type Player.

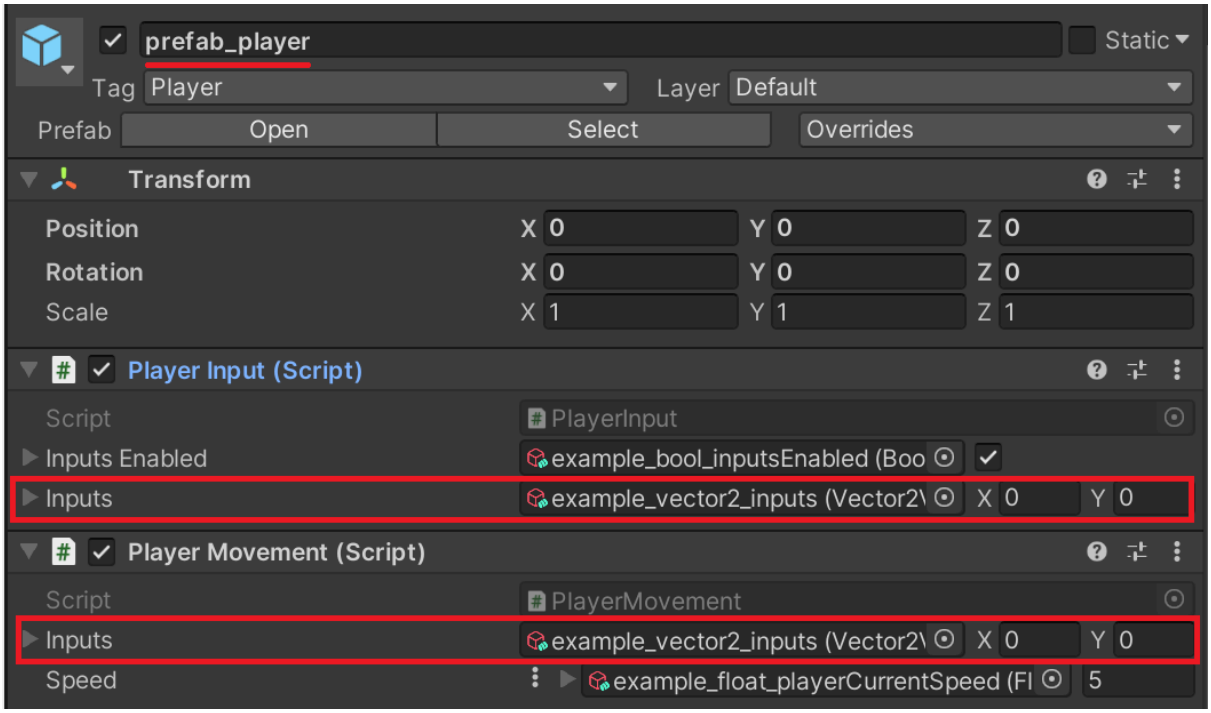


If you open the code, you will see that the only thing this script does is to assign the value of the SV in Awake(). Ignore the Scriptable List logic for now. Now that the player value is set, we can access it from any other class simply by referencing the SV of type player (example_player). To see how this is done, select the **PlayerNotifier** in the scene hierarchy.

You notice that it has a reference to our example_player variable. Now if you open the code, you can see that we can access our player very easily. This is nice, no need for singletons or other spaghetti code. Also, you can even reference this variable in prefabs, or scriptable objects as the SV example_player is an asset! This can be very powerful (for example when implementing abilities that require the player) and clean. I cover this in the youtube tutorials, so if you want to know more, I highly recommend watching those.

The second example, if you click on the “**prefab_player**”, you can look at the **Movement Controller** and the **Input Detector**.

If you press play, you can move the player with WASD / Arrow keys.



Here, we are **solving the dependency** of the movement controller with a scriptable variable. Input Detector **writes** to example_vector2_inputs (ScriptableVariableVector2) and Movement Controller **reads** from example_vector2_inputs.

Those 2 classes **don’t know each other** and the only thing they need is a scriptable object to write/read data. You can imagine how this can be useful in other cases as well. You can have entire parts of your game **independent from another one**, relying only on scriptable objects for communication.

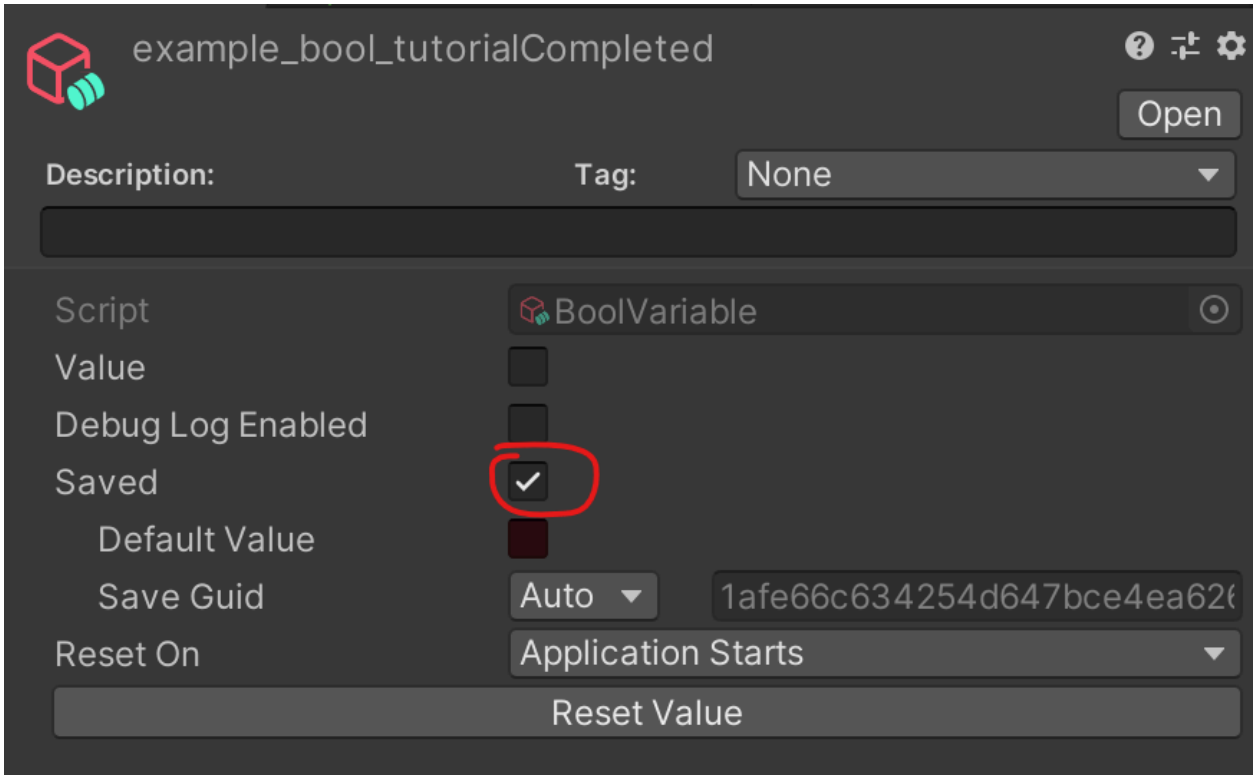
This is particularly useful when working across different scenes. For example, your UI elements can be in their own scene and reference Scriptable variables without needing to access anything specific to a scene.

Saving Variables

If you enter play mode, and press the button "**End Tutorial**", you will see that the **Tutorial** object will disable itself.

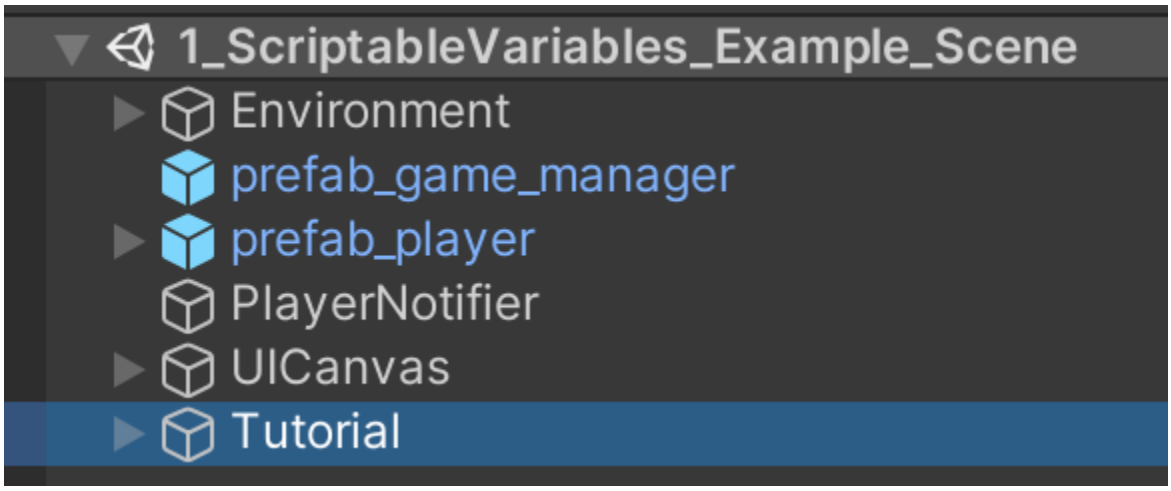
Now exit play mode and play again. You can see that the changes have persisted. It's because the **example_bool_tutorialCompleted** variable state has been saved to Player Prefs. To enable saving on your variable, select the variable and enable the property "**Saved**". All scriptable variables available in the package can be saved to Player Prefs **with a single click.** (Except the GameObjectVariable).

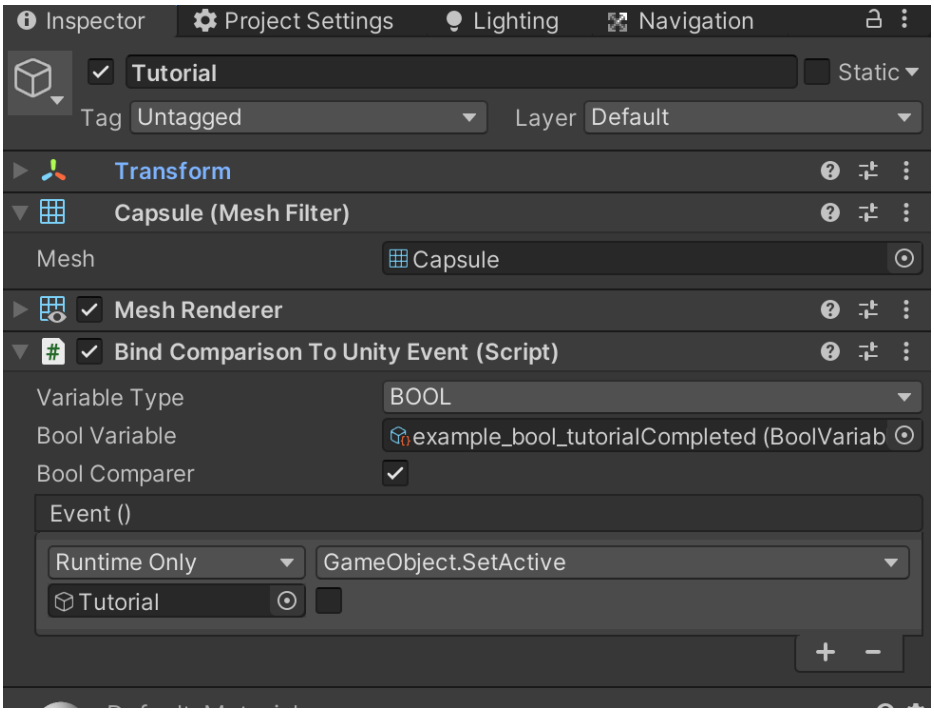
When loaded for the first time, the variable will be set to the **Default Value**.



The key used for Player Prefs is the Save Guid of the ScriptableVariable. This Guid is generated using the path of the asset. If Auto is selected, the Guid is generated automatically. If you change it to Manual, you can override the Guid.

You can use the fact that a variable has been saved in combination with a Binding script. If you select the **Tutorial** in the scene, you can see that its component BindComparisonToUnityEvent will disable the object if the condition is true. Bindings are explained in the next scene.



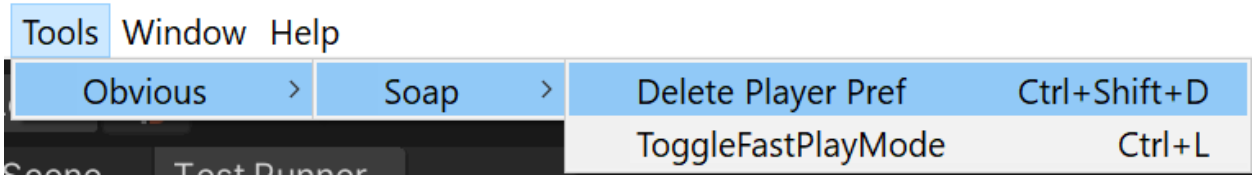


The player position is also saved, check the **PositionSaver.cs** on the **prefab_player** in the scene.

Saving variables like this is useful for things like level index, player score, coins, tutorials completed, etc. For a more complex and bigger save system, please check the 5_ScriptableSave_Example_Scene and its documentation.

Clearing the save

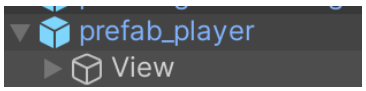
If you want to **clear the variables save** (deleting all Player Prefs), use the shortcut (CTRL+Shift+D) or the menu item:



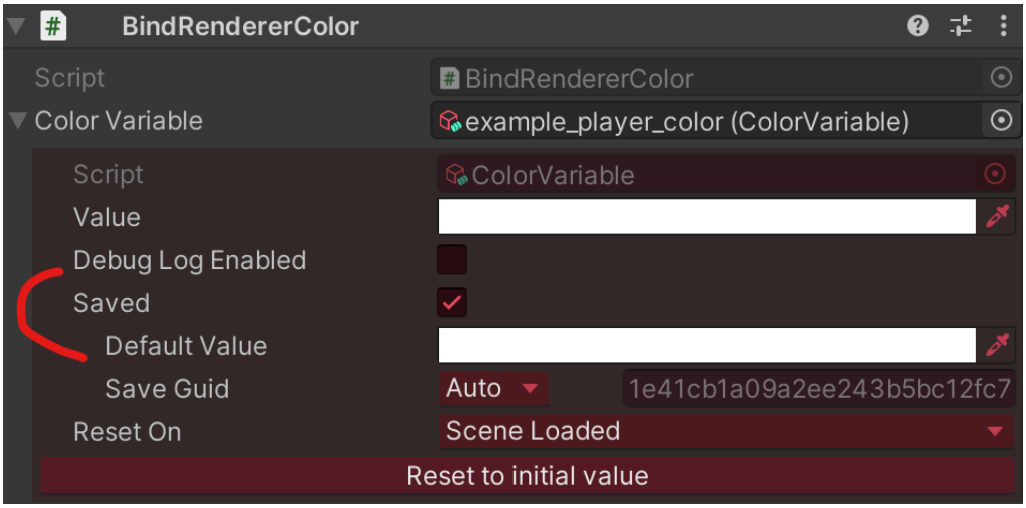
Save Exercise

You can set a random color to the player by clicking on the "Random Color" button on the bottom of the screen at runtime. Currently, the color is not saved. **Can you find how to save the player color?**

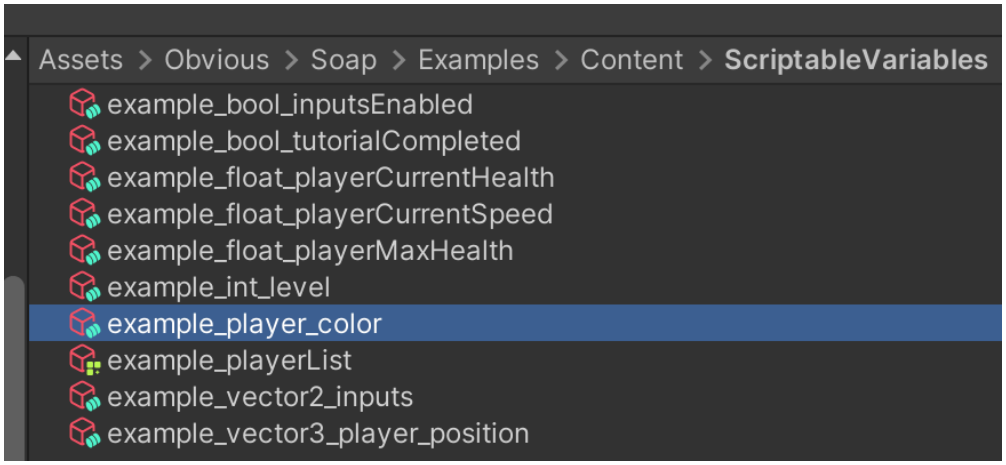
Solution 1: find the **BindRendererColor.cs** component on the prefab_player ->View.

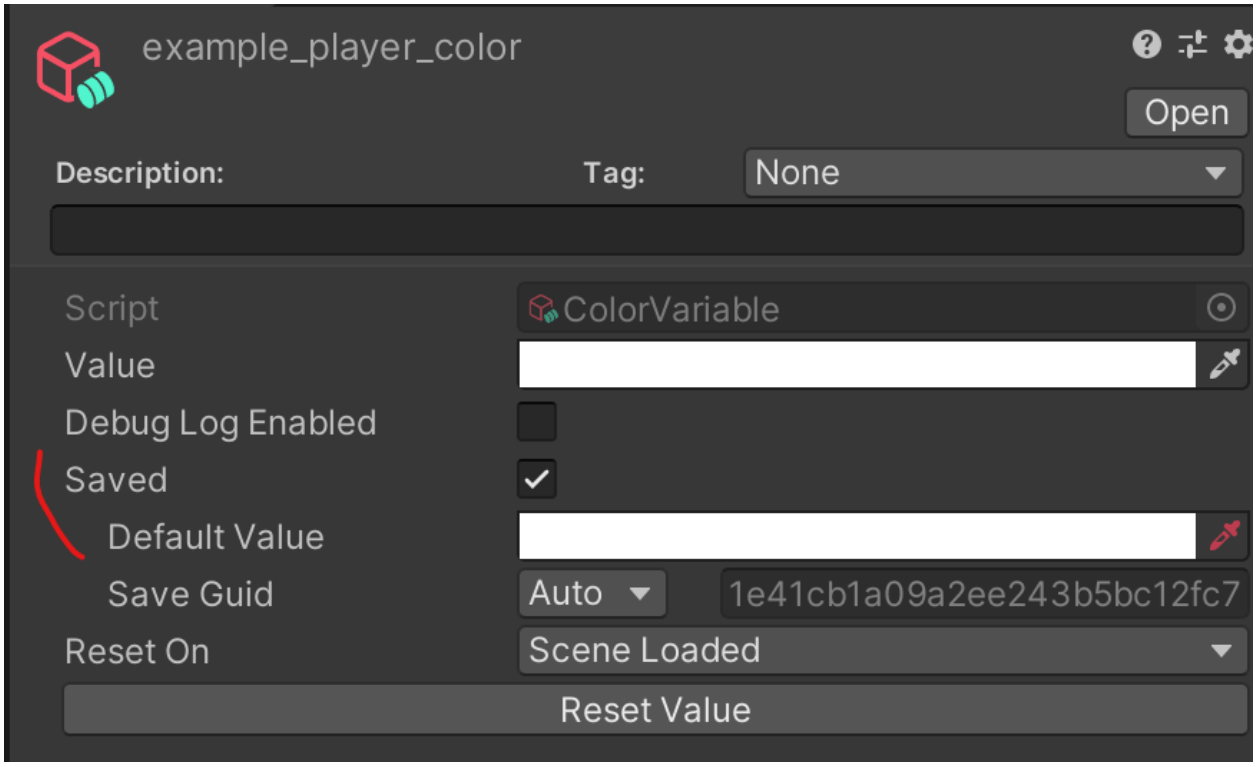


Then, expand the Scriptable Variable, enable the "Saved" property and set the default color to white.



Solution 2: find the scriptable variable **example_player_color** and enable the "Saved" property.

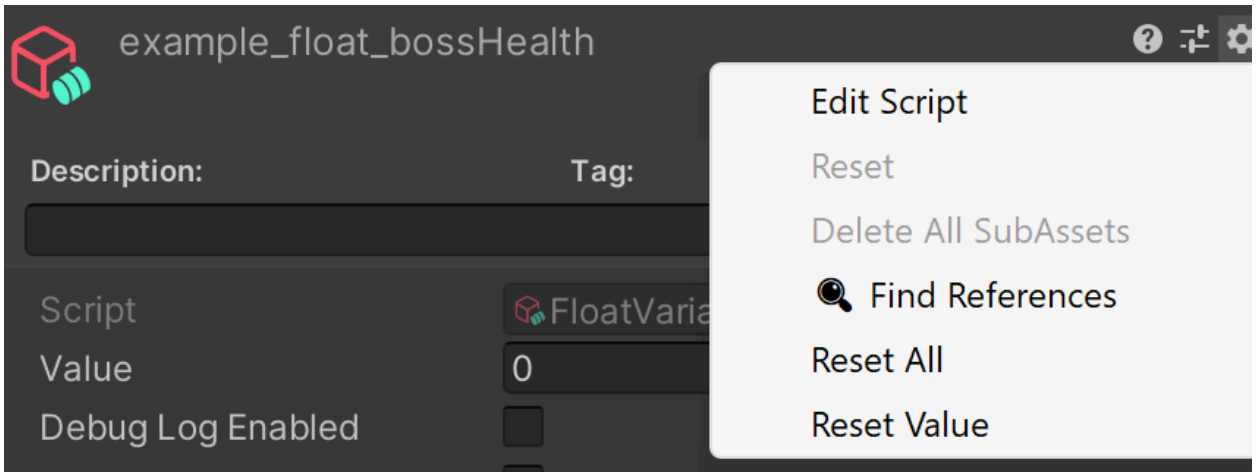




Note: when you create a new scriptable variable type, you can override the Save and Load method, and have custom save/load for them using Player Prefs 😊. Check the ColorVariable or Vector2Variable to have a concrete example.

Menu Context Options

You can easily Reset the Scriptable Variable to its default or to its initial value using the settings menu options.



Reset All: reset to the default values

Reset Value: sets the value property to the initial value (the value defined in the inspector before entering play mode).

Find References: opens a pop up telling you which asset in the scene in the project references this particular variable.