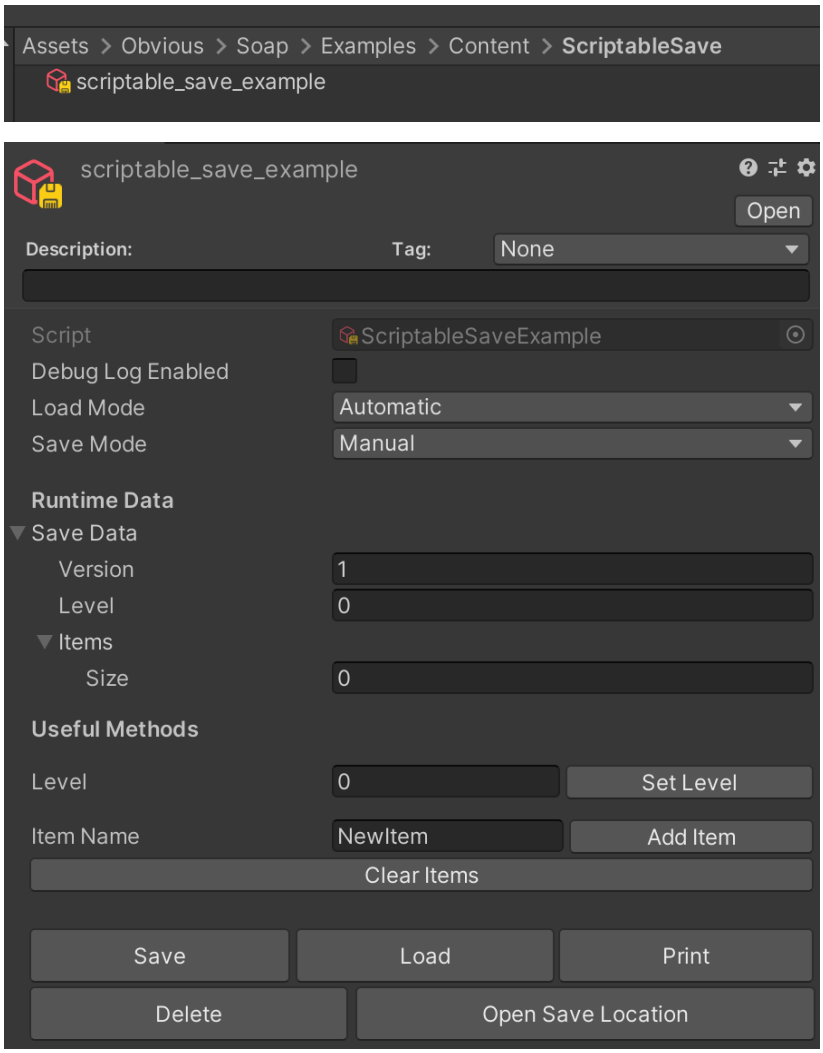
5. Scriptable Save

Table of Content

Save Data

In the project window, locate the scriptable_save_example and select it.



There is a lot to see, but actually this is a new empty save file. Let's quickly check in the code what data we are actually saving to disk. Open the script ScriptableSaveExample.cs.

```
[Serializable]
public class SaveData
{
    public int Version = 1;
    public int Level = 0;
    public List<Item> Items = new List<Item>();
}

[Serializable]
public class Item
{
    public string Id;
    public string Name;

    public Item(string name)
    {
        Id = Guid.NewGuid().ToString();
        Name = name;
    }
}
```

At the top of the file, we can see that our save data has a version number, a level, and a list of Items (which are defined just below). Note that these classes are marked [Serializable] in order to be able to convert them to JSON.

The SaveData class is the class you will customize for your game. Each game is different, so I've kept this example simple.

Useful Methods

If you scroll down in this script, you'll see the implementation of the Scriptable Save Example SO. This is also where you can customize it for your game. In my case, I added a few practical public getters and various public methods like SetLevel(), AddRandomItem(), ClearItems(), and more.

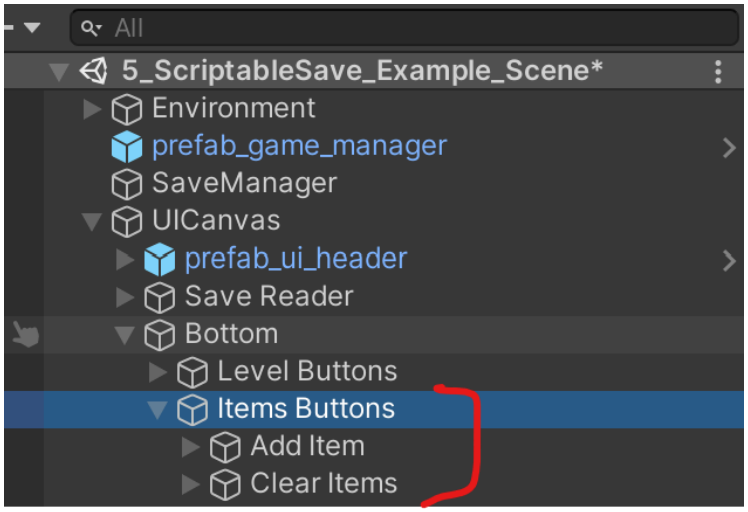
It is recommended to only allow this class to modify the _saveData directly and make all other classes use the public methods to affect the save. This way, you ensure that all modifications of the save are centralized in this

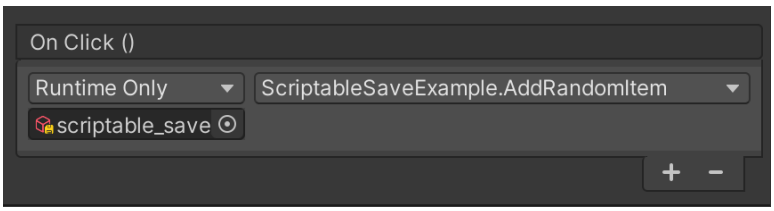
class, and you can control it. Note that after each modification, I manually call Save().

```
public void ClearItems()
{
    _saveData.Items.Clear();
    Save();
}
```

Depending on your game or save mode, you might not want to trigger a save after each modification.

Let's try now to use these methods. If you check the Items buttons in the hierarchy, you can see that each button directly references the scriptable save SO and calls a public method.



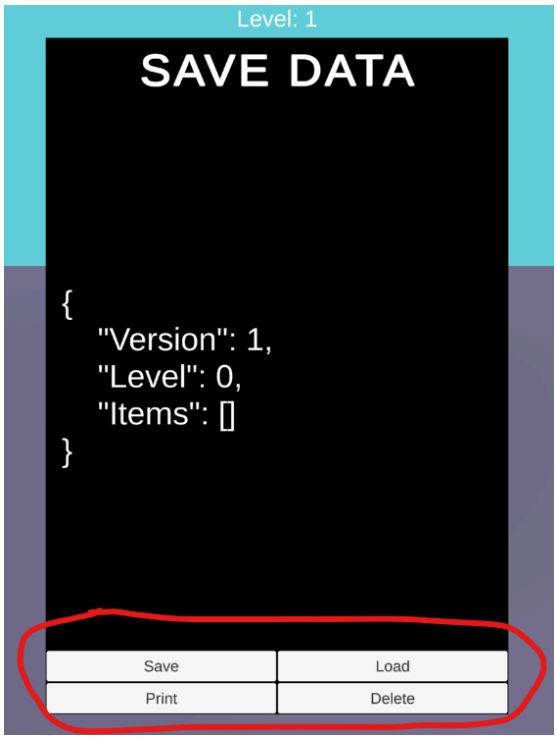


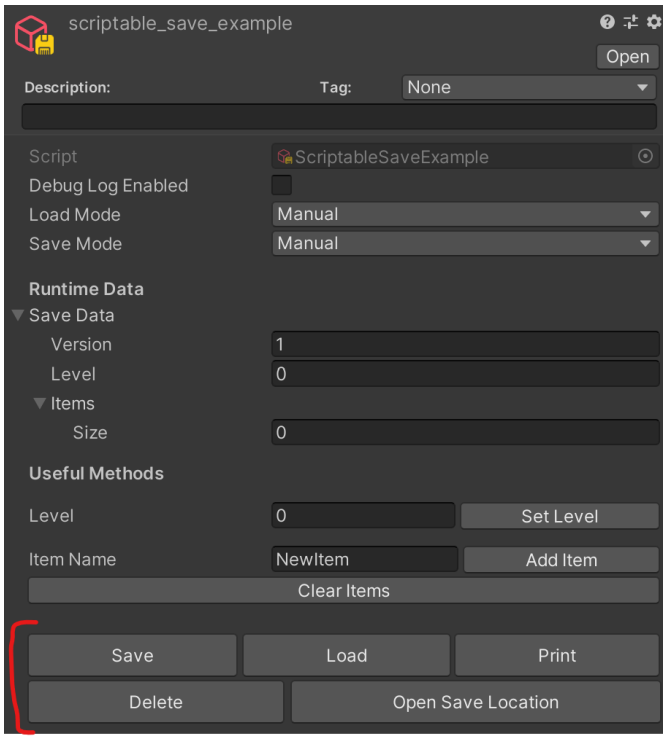
Play the game and start adding items, then clearing them. You will see that the changes are reflected in the save.

Scriptable Save SO also comes with built-in public methods:

- Save()
- Load()
- PrintToConsole()
- Delete()
- Open File Location()

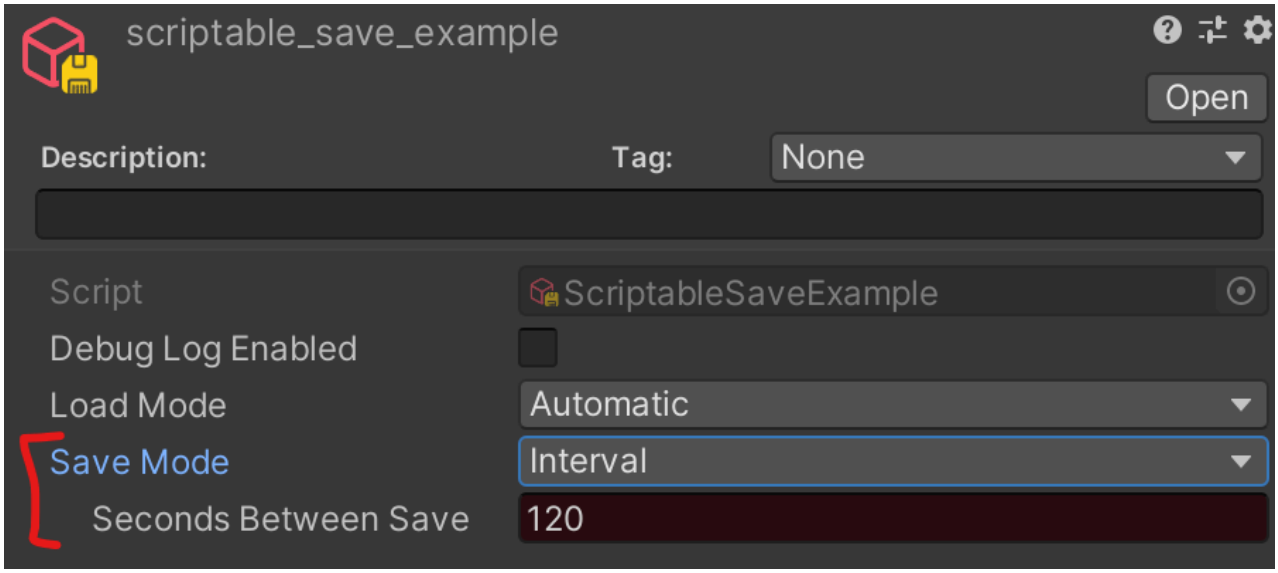
These are very useful and you can access them freely from code or from the inspector (like the buttons on the Save Reader). You can also expose them in a custom inspector to quickly access them for debug.





Save Modes

There are only two save modes: Manual and Interval. By default, Manual is selected. The reason is that it's usually safer to explicitly call the Save() method. However, in modern gaming, games often save progression automatically. If you set the save mode to Interval and define the seconds between each save, it will call the save method automatically at these intervals. You don't need to manually handle that. Feel free to try it out.

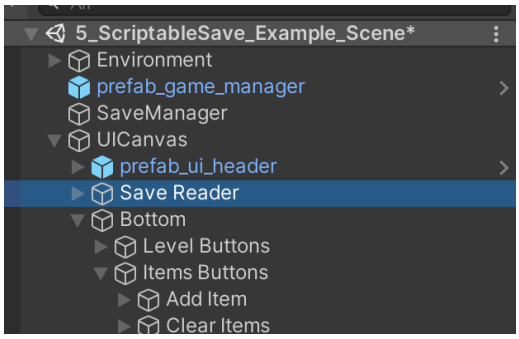


Load Modes

There are only two load modes: Automatic and Manual. By default, Automatic is selected, because loading is typically the first thing you do when your game starts.

Save Reader

Let's select the Save Reader in the hierarchy and inspect its code. This code is responsible for displaying the content of the save in the UI.



In Awake(), you can see that we register to all the callbacks provided by the Scriptable Save: OnSaved, OnLoaded, and OnDeleted. We want to update the text whenever the save performs an operation.

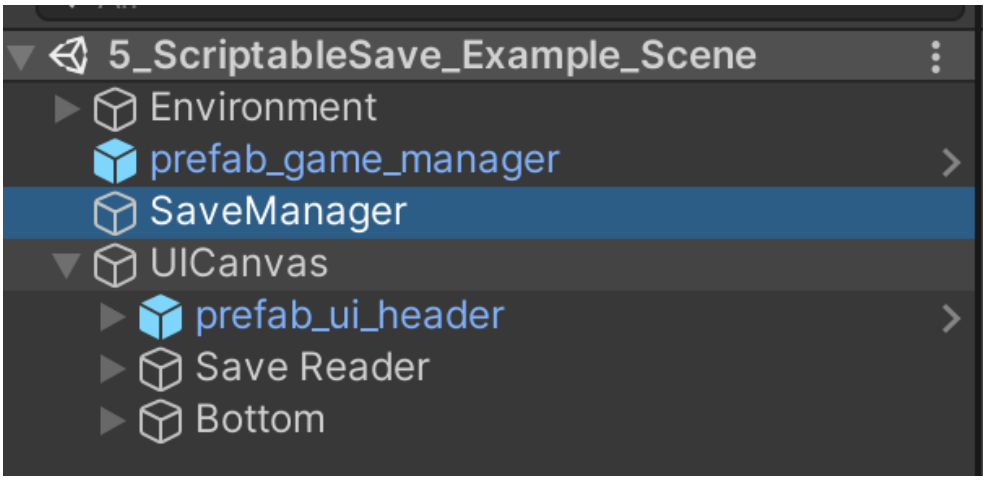
```
private void Awake()
{
    _scriptableSaveExample.OnLoaded += RefreshText;
    _scriptableSaveExample.OnSaved += RefreshText;
    _scriptableSaveExample.OnDeleted += RefreshText;

    if (_scriptableSaveExample.LoadMode == ScriptableSaveBase.ELoadMode.Automatic)
        RefreshText();
}
```

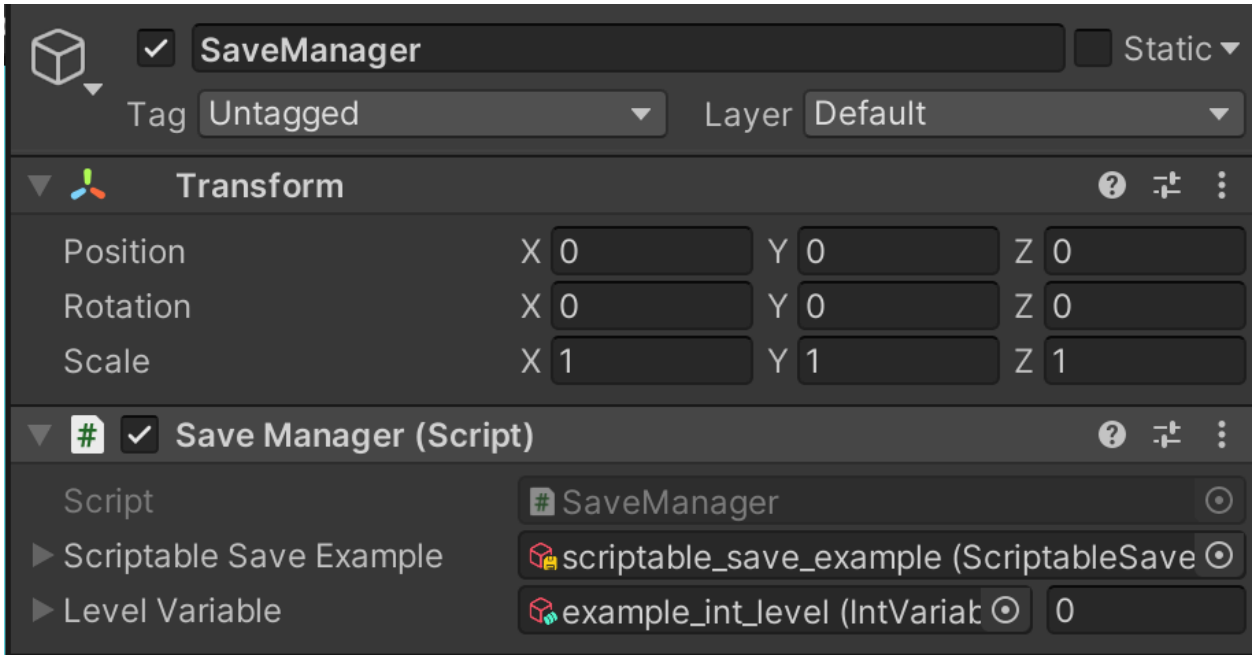
Note that if the load mode is set to automatic, Awake() will be called after the save has already been loaded. Therefore, we need to force RefreshText(). To display the save content, simply access the LastJsonString property:

```
private void RefreshText()
{
    _nameText.text = _scriptableSaveExample.LastJsonString;
}
```

Save Manager

Now, let's see how we can use the Scriptable Save to initialize Scriptable variables.
I created a simple script called SaveManager.cs. You can find it attached to the SaveManager object in the hierarchy.





This script has a reference to both the save and the variables it can initialize (in this case I only have one, but you can have as many as you need).

```
public class SaveManager : MonoBehaviour
{
    [SerializeField] private ScriptableSaveExample _scriptableSaveExample;
    [SerializeField] private IntVariable _levelVariable;

    private void Awake()
    {
        _scriptableSaveExample.OnLoaded += SetLevelValue;
        _scriptableSaveExample.OnSaved += SetLevelValue;
    }

    //Load on start as other classes might register to OnLoaded in their Awake
    private void Start()
    {
        if (_scriptableSaveExample.LoadMode == ScriptableSaveBase.ELoadMode.Manual)
            _scriptableSaveExample.Load();
        else
            SetLevelValue();
    }

    private void OnDestroy()
    {
        _scriptableSaveExample.OnLoaded -= SetLevelValue;
        _scriptableSaveExample.OnSaved -= SetLevelValue;
    }

    private void SetLevelValue()
    {
        _levelVariable.Value = _scriptableSaveExample.Level;
    }
}
```

Essentially, whenever the save events OnSaved or OnLoaded are called, we just need to set the values of our variables to those of the save. This way, we use the scriptable save as the source of truth. We are now free to use and reference our scriptable variable in our different components, as we used to do before.

Questions: **Why would I do this instead of using the "save" option on the scriptable variable ?**

It's up to you. For simple games, using the Player Prefs saving option of Scriptable variables might be enough. However, for more complex games or if you prefer to save all your data in a single file, this approach might be preferable.

Could I not reference the scriptable save directly to access the Level in the SaveData instead of using a Scriptable Variable ?

You could, but then you wouldn't get access to the benefits of using a scriptable variable, notably the OnValueChanged callback, the built-in compatibility with the bindings, and the simplicity of only getting a single variable instead of the whole save. Finally, restricting the scriptable save to key components gives you control over which components can access the save.

Would it be unreasonable to imagine writing back to the save when the Level Variable value changes ?

It wouldn't be! In fact, you can do this, but it's tricky. You can easily fall into a loop where OnValueChanged modifies the saves, which then calls OnValueChanged again, and so on. This is especially problematic if you save/load/delete the save at runtime. If you choose to implement this, you need to be very careful and do thorough testing, but it is indeed possible and can be convenient sometimes! I'm not showing this as an example, as I believe it can be confusing for people starting with Soap.

Save Version

You might have noticed that I added a Version int in the SaveData. This is an example, but it's a great way to detect if a save file is from an older version.

```
[Serializable]
public class SaveData
{
    public int Version = 1;
    public int Level = 0;
    public List<Item> Items = new List<Item>();
}
```

For developers who have worked on live games, when you add a new feature or refactor some elements, the save data structure might change. In these cases, you need to upgrade the old data structure to the new one. Scriptable Save provides two methods that you can override to suit your needs:

```
protected override void UpgradeData(SaveData oldData)
{
    if (_debugLogEnabled)
        Debug.Log(message: "Upgrading data from version " + oldData.Version + " t
    // Implement additional upgrade logic here
    oldData.Version = _saveData.Version;
}

protected override bool NeedsUpgrade(SaveData saveData)
{
    return saveData.Version < _saveData.Version;
}
```