Scene “1. ByOdinClassicEditorOnly”

Use the standard functionality of the Odin Serializer, which give a possibility to store objects “GameSettings” in Editor only (t.1 & t.2 from aims), because for storage reference use the “ReferenceResolver” (IExternalStringReferenceResolver or IExternalGuidReferenceResolver) which based on the use of methods from AssetDatabase class (namespace UnityEditor).

Serialization did in “two steps”:

1. Serialization the “plain members” of initial object the ref to assets stored in the collection (“List<UnityEngine.Object>”)
2. The item from collection (ref to unity assets) transformed by reference resolver to “plain data” (string or GuiD)

It may be useful for some custom Editor applications (I stored the C# plain class and therefore I used the methods from SerializationUtility class, if it is required to store a ScriptableObject, you should use the UnitySerializationUtilityMod class, I describe it more detail below).

Note. Common To All to next variants:

To have a possibility resolve the reference to assets in Editor and in Build I use the next common procedure (store reference & restore object):

1. I create the dictionary Dictionary<key, value>, where the key is a string - for name of SO, the value (Type of SO) - for the ref to this SO.
2. All assets were put in special folder in Resource folder.
3. Fill dict by Resources.LoadAll<LevelSO> (“”).ToDictionary((level) => level.name, (level) => level);
4. “To store ref to SO” – I use simply get name the SO.
5. “To restore object from ref” – I use this dictionary (with some checks)

All variants below differ only by methods to combine the serialization process with the “common procedure”. Some variants are more elegant (use additional generic class and so on) and give more possibilities, other simpler to realize.

All next variants were ordered by complexity ascending

Scene “2. BySpecialSO_Unity_Odin” (“simple to realize”)

Use the special SO (like GameSettingsSpecialSO) – This is the original SO to which methods have been added, which realized the ISerializationResoverRefUnityObject interface.

This is an analogue of the classic ISerializationCallbackReceiver Unity interface. The Unity interface will be called on all serialization’s calls (“anywhere and anytime”), the proposed interface will use only when you store data. It gives a more control and less utilization of resources.

It can be used with **Unity Serializer** (w/o any use of Odin Serializer):

```
            if (data is ISerializationResoverRefUnityObject resoverRefUnityObject)
            {
                resoverRefUnityObject.BeforeSerialize();
            }
            string strJSON = JsonUtility.ToJson(data);
```

"data" the stored object. And to restore:

```
            JsonUtility.FromJsonOverwrite(str, data);
            if (data is ISerializationResoverRefUnityObject resoverRefUnityObject)
            {
                resoverRefUnityObject.AfterDeserialize ();
            }
```

Note. When we work with objects derived from UnityEngine.Object we must use at restore the "FromJsonOverwrite" (or "analogue" of it).

To store the string with ref to SO I added special member to this SO:

```
[SerializeField,HideInInspector] private string _levelSOName;
The original member will also be serialized
```

```
[SerializeField] private LevelSO _levelPlayer;
```
but its value is not used `(and will be "override" after deserialization)`

It can be used with **Odin Serializer**:

I little extend the possibilities of Serializer:

- I added the additional `NonOdinSerialized attribute (Unity Serializer not react on it)`

```
[SerializeField, NonOdinSerialized] private LevelSO _levelPlayer;
```

- And created the `SpecialSerializationPolicy – "SpecialUnity" which can react on that attribute and skip members from serilization.`

```
        SerializationContext myContext = new SerializationContext()
        {
            Config = new SerializationConfig() { SerializationPolicy =
SpecialSerializationPolicy.SpecialUnity }
        };
        if (data is ISerializationResoverRefUnityObject resoverRefUnityObject)
        {
            resoverRefUnityObject.BeforeSerialize();
        }
        UnitySerializationUtility.SerializeUnityObject(data, ref bytes, ref
unityReferences, FormatData, serializeUnityFields: true, context: myContext);
```

In this case Odin Serializer does not give any warnings and give the nicer to peoples JSON format, but this format non canonic and not compatible with standard Unity JSON.

The "result: use the Unity or Odin serializer will be the same:

It gives a possibility to store & restore SO in Editor and in Build (all tasks completed), but you must all logic to support resolving reference put to the stored objects (it's not very elegant).

Note. Common To All to next variants:

To separate the logic to support resolving reference from stored object. I created the special generic class

```
UniversalScriptableObjectStringReferenceResolver<T>  :
IExternalStringReferenceResolver where T : ScriptableObject
```

`Which derived the standard Odin interface – IExternalStringReferenceResolver`

Whereas T used as the Type of member of stored object with ref to assets (in my examples It is the LevelSO or ComplexitySO).

Scene "3.ByOdinPlainClass" (nicer but not ideal)

It is work good but only with C# plain class with ref to assets (e.g. in my case it was a `GameSettings objects)`, because it uses the methods from `SerializationUtility class:`

```
        SerializationContext serContext = new SerializationContext()
        {
            StringReferenceResolver = new
UniversalScriptableObjectStringReferenceResolver<ComplexitySO>(),
        };
        byte[] bytes = SerializationUtility.SerializeValue(data, FormatData,
serContext);
        File.WriteAllBytes(filePath, bytes);
```

And

```
        DeserializationContext desContext = new DeserializationContext()
        {
            StringReferenceResolver = new
UniversalScriptableObjectStringReferenceResolver<ComplexitySO>(),
        };
        byte[] bytes = File.ReadAllBytes(filePath);
        return SerializationUtility.DeserializeValue<T>(bytes, FormatData,
desContext);
```

which does not support the correct restore (deserialization of UnityObjects like "FromJsonOverwrite")

Therefore, it realizes all tasks, the stored objects contain only data (except "testing code" in my case).

Scene "4.ByOdinUnityObject" (may be stop on that)

In this case as object to store was used the `GameSettingsSimpleSO (SO with only data w/o any special code to support serialization). Odin serializer as I told give a possibility to serializer the UnityObject, for that exist the special methods in UnitySerializationUtility class. But initially this class was created to support serialization any different UnityObject in Editor (Odin Inspector) include different Prefab mode and so on (It is a very hard task, and I didn't understand all aspects of that and limitation, but as you know the possibilities of Odin Inspector is Great).`

Therefore "for stability and good management" (and may be another reasons) they support only the resolvers of IExternalIndexReferenceResolver Type, which use as storage for reference only internal List<UnityObjects >, which exist only in current session of Editor.

Note (from Odin doc for `UnitySerializationUtility class`) “Note that setting the IndexReferenceResolver on contexts passed into methods on this class will have no effect, as it will always be set to a UnityReferenceResolver.”

Therefore, I only made this variant only for case to store the UnityObject with ref to SO (like `GameSettingsSimpleSO, other type assets I did not test and not thought about it, all code under Apache license and you can continue to make it better).`

I made additional modifications to original Odin Serializer code:

- I created class `UnitySerializationUtilityMod and put it in the OdinSerializer namespace and added to OdinSerializer Assembly definition.`
- I created the analogue of methods `SerializeUnityObject and DeserializeUnityObject from original of UnitySerializationUtility class`
- The main different is medicated methods doesn’t use the UnityReferenceResolver and can work with custom “ReferenceResolver” (IExternalStringReferenceResolver or IExternalGuidReferenceResolver)
- All other left without changes except I was must to copy to my class the `ICache GetCachedUnityWriter and GetCachedUnityReader from original class because thesy have a private access modificator and original class (UnitySerializationUtility) is static (to minimize changes to original code).`

As a result, I can use to serialize the next code:

```
        SerializationContext serContext = new SerializationContext()
        {
            StringReferenceResolver = new
UniversalScriptableObjectStringReferenceResolver<LevelSO>(),
        };
        byte[] bytes = default;
        UnitySerializationUtilityMod.SerializeUnityObject(data, ref bytes, FormatData,
serializeUnityFields: true, context: serContext);
```

And for deserialize the next:

```
        DeserializationContext desContext = new DeserializationContext()
        {
            StringReferenceResolver = new
UniversalScriptableObjectStringReferenceResolver<LevelSO>(),
        };
        UnitySerializationUtilityMod.DeserializeUnityObject(data, ref bytes,
FormatData, desContext);
```

It works like Variant 3 but can correctly deserialize the UnityObjects (like “FromJsonOverwrite”).

Scene “5.ByOdinByPhaseCompexClass” (additional possibilities)

In this example for store, I use the GameSettingsComplex object, which contains data members and many (😉 only two) members with ref to asset. For simplicity was take the C# plain class and only two members, member can be all with ref and the C# plain can replaced by SO.

This variant uses the same two-phase serialization like in the variant 1 (I didn’t remember why I decided take that variant as base this variant, not took the “one phase variant”), but main point that for resolving reference to UnityObject it is use the chain of resolvers (like in Binary Serialization):

```
    private static string FileName((string fileBase, string fileExt) fileName, int 
num) => fileName.fileBase + num + fileName.fileExt;

    public static void SaveStrRef((string fileBase, string fileExt) fileName, 
List<Object> unityReferences)
    {
        SerializationContext serContext = new SerializationContext()
        {
            StringReferenceResolver = new 
UniversalScriptableObjectStringReferenceResolver<ComplexitySO>()
            {
                NextResolver = new 
UniversalScriptableObjectStringReferenceResolver<LevelSO>()
            },
        };
        for (int i = 0; i < unityReferences.Count; i++)
        {
            byte[] bytes = Serialize(unityReferences[i], serContext);
            File.WriteAllBytes(FileName(fileName, i), bytes);
        }
    }
```

Every resolved reference will be store in the separated file because the methods on second phase receive the List<Objects> from first phase. The Deserialization also must did in two phase in backward order: initially call the

```
public static void LoadStrRef((string fileBase, string fileExt) fileName, List<Object> 
unityReferences, bool clearUnityReferences = true)
```

and only after that the call to

```
 _gameSettingsComplex = OdinCallsExample.LoadTwoSteps <GameSettingsComplex>
(FilePathMainFile, unityReferences);
```

To combine the restored ref with other members in one object.

In this example the serialized data consists of 3 files (one main file from first phase and two from second phase).

Files:

The install.txt describe installation of the environment and OdinSerializer (not important info).