Serializable Dictionary Pro

Rotary Heart

Documentation Version: 1.0

Email: ma.rotaryheart@gmail.com

Description:

This package contains a class, SerializableDictionary.cs, that can be inherited to be able to have a serializable dictionary. There is no limitation for what to use as Key or as Value, other than it being a serializable type. It also contains multiple example classes that shows how to setup the dictionary.

Setup:

To use simply add the SerializableDictionaryPro folder to your project (which is on the package). It doesn't need to be on root of your project, so feel free to move it wherever you want.

How to use:

This section is divided into 3 different cases for easier explanation, but for any type of implementation you will need to create a custom class, can be a nested class, that inherits from SerializableDictionary and setup the respective types for key and value that are going to be used by the dictionary.

Note that for Unity to be able to serialize this class you need to add the attribute [System.Serializable] to your class since that is the way Unity serialization works.

Contents

Data Structures Available

Currently the system has multiple data structures implemented, each of them with their unique usefulness.

1. **SerializableDictionary<TKey, TValue>**
    a. Base structure used to create the asset, provides a fully serializable dictionary.
2. **SerializableOrderedDictionary<TKey, TValue>****
    a. Same as with a dictionary, but the data is ensured to always be on the same order that it was added.
3. **SerializableSortedDictionary<TKey, TValue>****
    a. Same as with a dictionary, but the data is ensured to always be sorted.
4. **SerializableHashSet<TValue>**
    a. Does not have any key data, only values

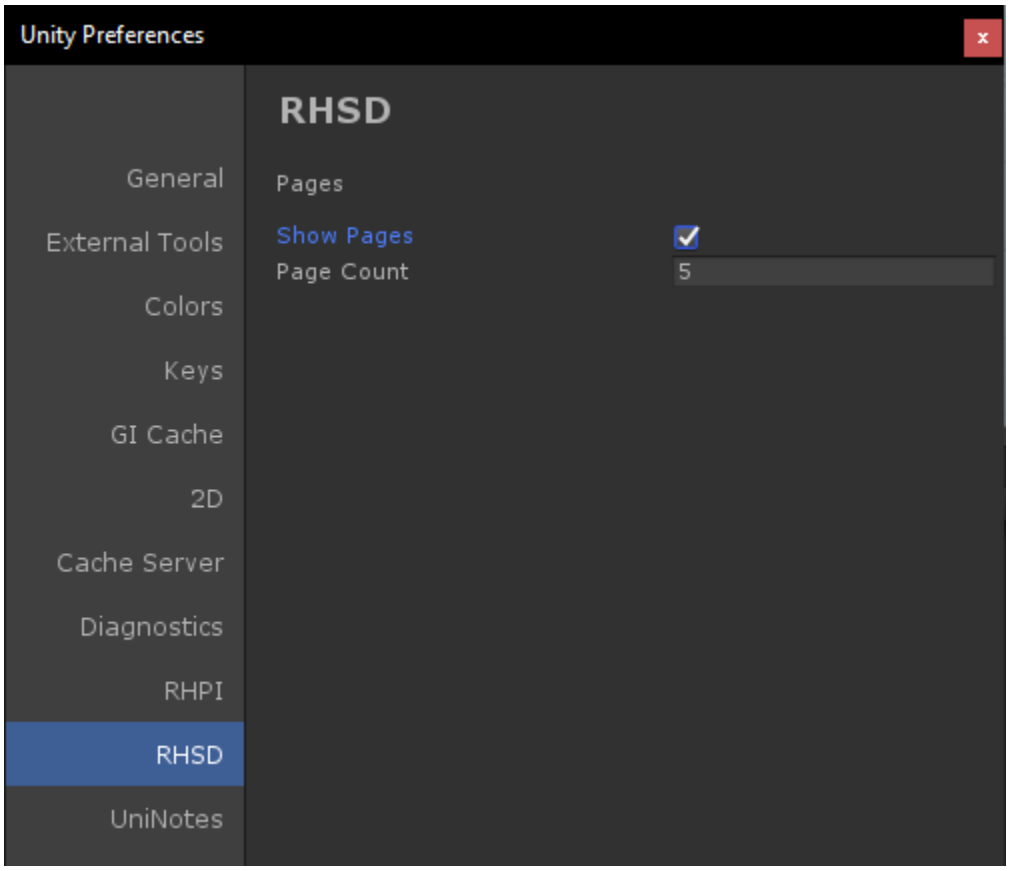
While this documentation will not explain when to use any of them, they all are fully functional with the same inspector.

**To fully view the effectiveness of the dictionary a new example (BehaviourExample.cs) has been added. To preview it, simply add it to any GameObject on any scene and click play. You will see how the dictionaries have different orders of the data that was modified/added.

Pages System

The pages system can be useful if you have a huge dictionary, this system allows the drawer to only draw up to a specific number of entries per page. It will work the same way as any forum paging logic.

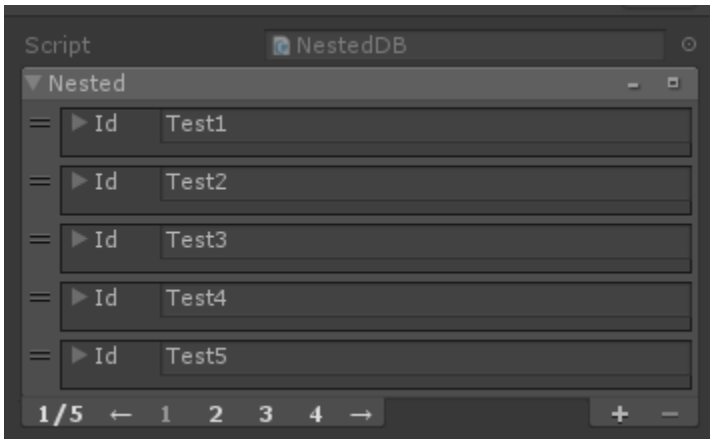
To enable the system, you need to go to the Edit > Preferences > RHSD and check the Show Pages check box.



Preferences window

Page count will be how many entries you want to draw per page, it has a minimum of 5, but it doesn't have any maximum number.

Once the system has been enabled you will be able to see the pages section on any dictionary drawer.



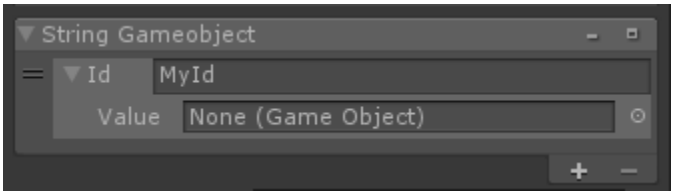
Pages section preview

By clicking on any number, you will be changing between the visible pages. This will change what is currently being drawn on the dictionary. Note that you won't be able to reorder element between different pages, for this you will have to disable the system, change the order, enable it back. Every other functionality will still work without problems, including reordering elements on the same page.

Simple Dictionary

```
[System.Serializable]
public class MyDictionary : SerializableDictionaryBase<string, GameObject> {
}
```

On this example you can see that the dictionary that we will be using has a string for key and a GameObject for value. With this now you can have a field of type MyDictionary and you will be able to modify the data from the editor.



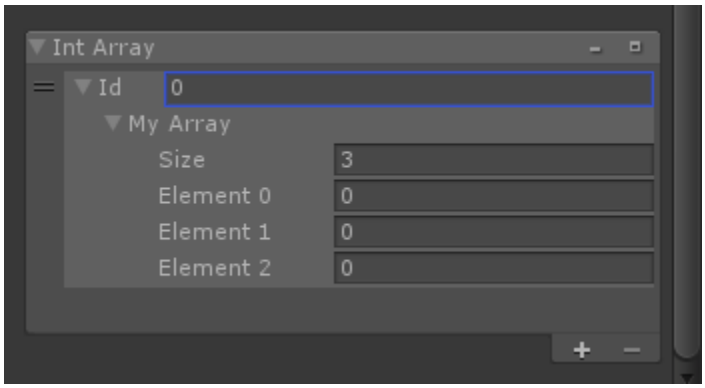
Preview of example provided

Advanced Dictionary

If the value that you want to use is either an Array or a List, you cannot use it directly into the dictionary value type because Unity doesn't serialize Array of Array or Array of List. Instead you need to wrap your Array or List with a custom class/structure. See the following example.

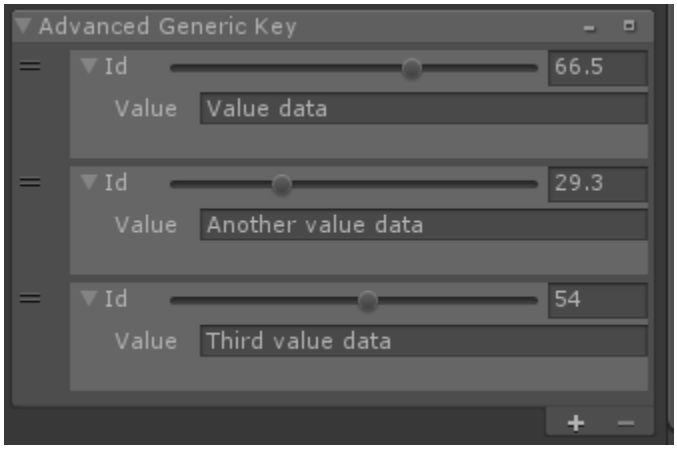
```
[System.Serializable]
public class Int_IntArray : SerializableDictionaryBase<int, MyClass> { }

[System.Serializable]
public class MyClass
{
        public int[] myArray;
}

[SerializeField]
private Int_IntArray _intArray;
```

This way unity can serialize your array and the system will work without any problems.



DrawKeyAsProperty

The system comes with an advanced attribute that can be used to allow keys to use any custom drawer, but to be able to use this we need to let the compiler know how to compare this advanced key. We will be using the example provided on the package to explain how to make it work.



For this example, we are using the attribute Range to allow the key to be drawn with the built-in range field. For this we need to setup the code as following:

```
//Include the needed attribute to the class

[System.Serializable]
public class AdvancedGenericClass
{
   [Range(0, 100)]
   public float value;

   //This equals function is added for convenience, it is not required
   public bool Equals(AdvancedGenericClass other)
   {
      if (ReferenceEquals(null, other)) return false;
      if (ReferenceEquals(this, other)) return true;
      return other.value == value;
   }
```

```
    //Override the Equals function to include any logic that we need to make sure that the keys are the same. This is what will be used when working with the dictionary
    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;
        if (obj.GetType() != typeof(AdvancedGenericClass)) return false;
        return Equals((AdvancedGenericClass)obj);
    }

    //Finally override the GetHashCode and include any kind of HashCode check
    public override int GetHashCode()
    {
        unchecked
        {
            return value.GetHashCode();
        }
    }
}

[System.Serializable]
public class AdvanGeneric_String :
SerializableDictionaryBase<AdvancedGenericClass, string> { };

[SerializeField, DrawKeyAsProperty]
private AdvanGeneric_String _advancedGenericKey;
```

As you can see to make the system ignore the Generic type and draw it as a normal property, we need to use the attribute DrawKeyAsProperty on the dictionary field.

Then we need any field that can be serialized by Unity inside the key class. The system will automatically detect the first serializable field and use it, if nothing is found it will fallback on drawing it as a regular generic key.

Now for the dictionary to work correctly we need to override the functions ***Equals*** and ***GetHashCode*** so that the comparison of keys can be executed correctly. Inside this function we will implement any logic that we want to be used to identify that they keys are the same.

Note that this step is required otherwise the keys will be compared by reference and not by the value set on the inspector, which means that unless you use a direct reference of the key it will always return false (the key not found on the dictionary).

DefaultKey

This attribute allows to specify what value should be used for the default new entry key in the dictionary.

```
[SerializeField, DefaultKey(5623)]
private Int_String _intString;
```

With the above code any new entry that is going to be added to this dictionary will have a default key of 5623. Note that you can still change the value inside the add element section below, this will only provide a starting value.