Serialized Dictionary

Readme & User Guide

Serialized Dictionary is designed to feel native to the Unity Editor while providing some additional functionaliy to speed up frequent workflows.

Quick Start

Use the class **SerializedDictionary<,>** in the Namespace *AYellowpaper.SerializedCollections* instead of the **Dictionary<,>** class to serialize your data. Use the **SerializedDictionary** Attribute for further customization. It follows the same Unity serialization rules as other Unity types.

See the image below for example usage:

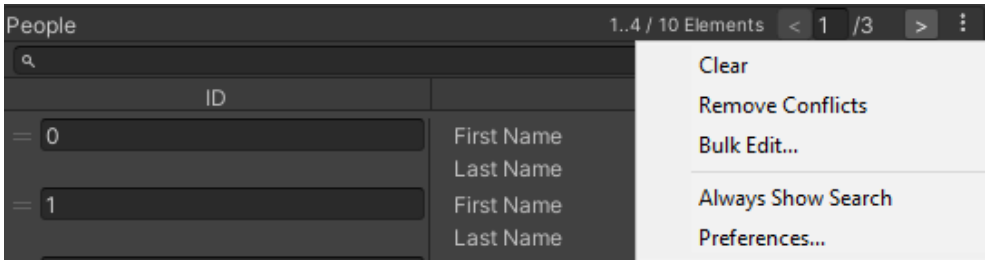
```
[SerializedDictionary("Damage Type", "Description")]
public SerializedDictionary<DamageType, string> ElementDescriptions;
```

User Guide

Serialized Dictionary will serialized any Unity serializable type, including Unity Objects like transforms and ScriptableObjects. Furthermore, it allows to serialize duplicate keys and null values. The main purpose is to avoid accidental loss of data when you decide to change code or remove objects. The following color coding exists:

- Red: the key is invalid, meaning either duplicate or null
- Yellow: there are duplicate keys, but this is the one that's used (it comes before the others)
- Blue: the key was found in the search

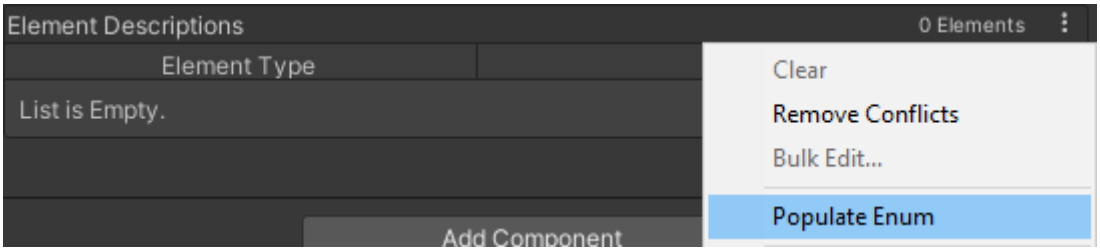
The Burger Menu in the top right is very important. It contains important options that will speed up your workflow. Most of the should be self explanatory.



Bulk Edit Operations

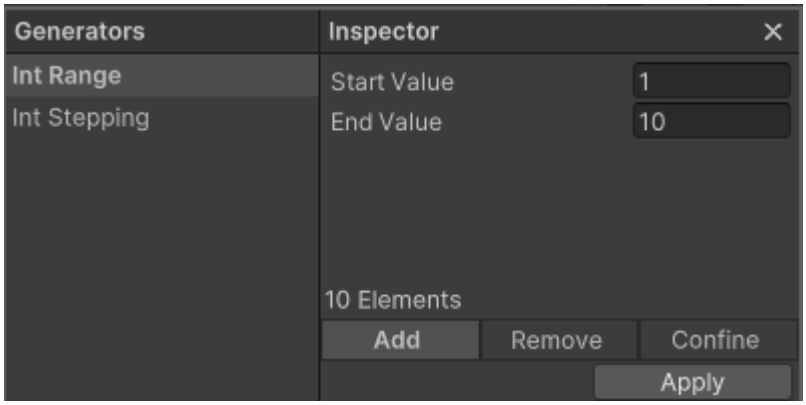
To quickly modify lots of existing entries you can use and also create custom **KeyListGenerators**. E.g. for dictionaries that contain enums as keys, there's a **KeyListGenerator** that will populate the dictionary with all values from the enum with one press of a button.

1. Select "Populate Enum" with dictionary that has enum as key



2. The dictionary is filled with all values from the enum

Furthermore, there are populators for integers, which allow for custom input fields to modify the data that will be generated.



In this case, Int Range will create keys between the range of 1 to 10. Before you Apply the generated values, you have the option to select between Add, Remove and Confine. They do the following:

- Add will add the values if they don't exist as keys yet
- Remove will remove the given values
- Cofine will add the values if they don't exist as keys yet, and remove all keys that are not contained in the list of generated values

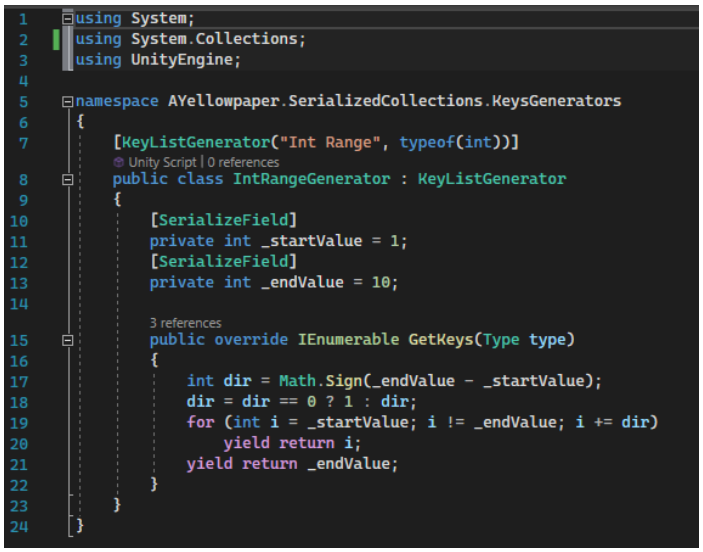
As as example, assume you have keys 5 to 15 in your dictionary, and have chosen 1 to 10 in the generator. Given the following options, the resulting keys will be as follows:

- Add will result in keys from 1 to 15, because 1 to 4 will be added

- Remove will result in keys 11 to 15, because 5 to 9 will be removed
- Confine will result in 1 to 10, because 1 to 4 will be added and 11 to 15 removed

Creating Bulk Edit Operations

Some KeyListGenerators exist for enums and ints. But you might want to add your own custom Key Generators. This is easily done by creating a new class that inherits from **KeyListGenerator** and adding the **KeyListGenerator** Attribute to it. See the image for the int generator example:



```
using System;
using System.Collections;
using UnityEngine;

namespace AYellowpaper.SerializedCollections.KeysGenerators
{
    [KeyListGenerator("Int Range", typeof(int))]
    Unity Script | 0 references
    public class IntRangeGenerator : KeyListGenerator
    {
        [SerializeField]
        private int _startValue = 1;
        [SerializeField]
        private int _endValue = 10;

        3 references
        public override IEnumerable GetKeys(Type type)
        {
            int dir = Math.Sign(_endValue - _startValue);
            dir = dir == 0 ? 1 : dir;
            for (int i = _startValue; i != _endValue; i += dir)
                yield return i;
            yield return _endValue;
        }
    }
}
```