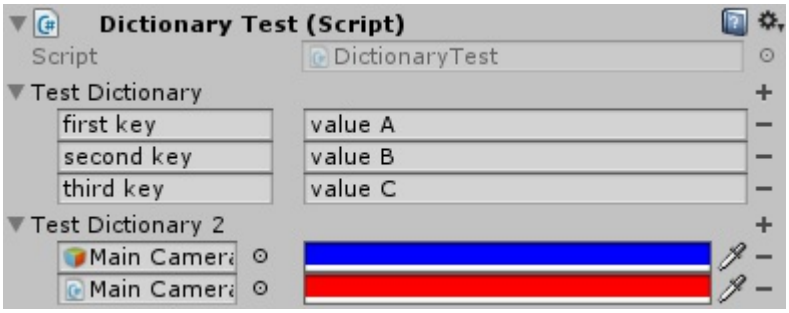
SerializableDictionary

A serializable dictionary class for Unity.

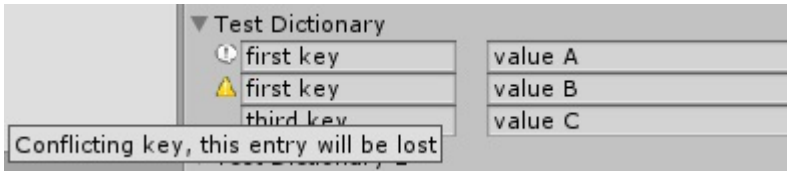
Unity cannot serialize standard dictionaries. This means that they won't show or be edited in the inspector and they won't be instantiated at startup. A classic workaround is to store the keys and values in separate arrays and construct the dictionary at startup.

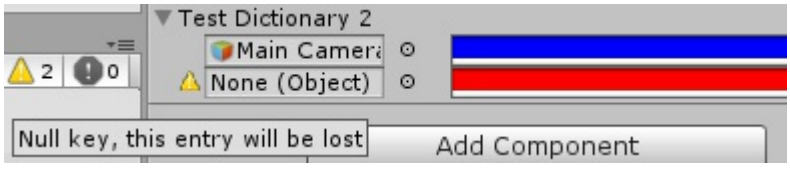
This project provides a generic dictionary class and its custom property drawer that solves this problem.



Features

- It inherits from `Dictionary<TKey, TValue>`
- It implements a `CopyFrom(IDictionary<TKey, TValue>)` method to help assign values from regular dictionaries
- You can use any serializable type by unity as key or value.
- It can be edited in the inspector without having to implement custom editors or property drawers.
- The inspector will handle invalid dictionary keys such as duplicated or `null` keys and warn the user that data loss can occur if the keys are not fixed.

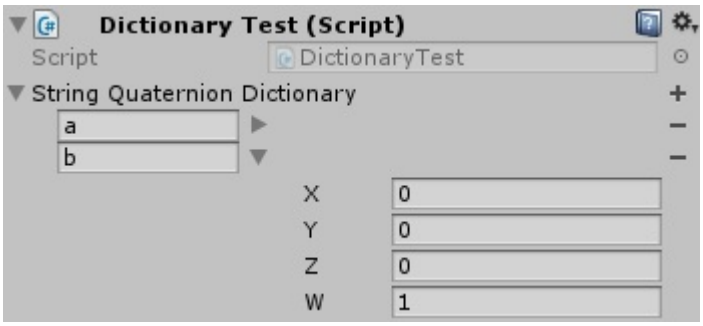




Limitations

- A non-generic derived class has to be created for each `<TKey, TValue>` combination you want to use. A `CustomPropertyDrawer` has to be declared for each of these classes.
- Types drawn with a folding arrow (such as `Quaternion` or any serializable class) used as keys or values

will have an empty label next to the arrow.



- Multiple editing of scripts using `SerializableDictionaries` in the inspector is not supported. The inspector will show the dictionaries but data loss is likely to occur.
- The conflicting key detection does not work when using `LayerMask` as key. The `LayerMask` value is changed after the `CustomPropertyDrawer` execution.

Usage

Copy these files in your project:

- `Assets/`
  - `Scripts/`
    - `SerializableDictionary.cs`
    - `UserSerializableDictionaryPropertyDrawer.cs` (optional)
  - `Editor/`
    - `SerializableDictionaryPropertyDrawer.cs`
    - `UserSerializableDictionaries.cs` (optional)

As Unity is unable to directly serialize generic types, create a derived class for each `SerializedDictionary` specialization you want.

```
[Serializable]
public class StringStringDictionary : SerializableDictionary<string, string>
{}

[Serializable]
public class MyScriptColorDictionary : SerializableDictionary<MyScript, Color>
{}
```

Declare the custom property drawer for these new types by adding the `CustomPropertyDrawer` attribute to the `SerializableDictionaryPropertyDrawer` class or of one of its derived class.

```
[CustomPropertyDrawer(typeof(StringStringDictionary))]
[CustomPropertyDrawer(typeof(MyScriptColorDictionary))]
public class AnySerializableDictionaryPropertyDrawer :
SerializableDictionaryPropertyDrawer {}
```

It is recommended to create one derived class in a separate file and add the attributes to this class instead of modifying the original `SerializableDictionaryPropertyDrawer` class. You can use the same class for all your `SerializableDictionary` specializations, there is no need to create a new one for each specialization.

Add the dictionaries to your scripts and access them directly of through a property. The dictionaries can be accessed through a property of type `IDictionary<TKey, TValue>` for better encapsulation.

```
public StringStringDictionary m_myDictionary1;

[SerializeField]
MyScriptColorDictionary m_myDictionary2;
public IDictionary<MyScript, Color> MyDictionary2
{
    get { return m_myDictionary2; }
    set { m_myDictionary2.CopyFrom (value); }
}
```

The `CopyFrom(value)` method clears the `m_myDictionary2` dictionary and adds to it each of content of the `value` dictionary, effectively copying `value` into `m_myDictionary2`.

`SerializableDictionary` has a copy constructor from `IDictionary<TKey, TValue>`. As constructors from parent classes cannot be used directly, you have to add a copy constructor to your derived classes calling the base constructor in order to use it.

```
[Serializable]
public class StringColorDictionary : SerializableDictionary<string, Color>
{
    public StringColorDictionary(IDictionary<string, Color> dict) : base(dict)
{}
}
```