Serialised Data Structures

By The Black Cat

Content

Read Before Everything!!

This is very important. When you are using the serialised data structures, if you want to store lists, **DO NOT just store them as they are!**

What you want to do is to write a **wrapper** for the list and store the **wrapper** instead.

Luckily, you can skip the "write a wrapper" part because I have already written one for you.

To wrap a list, use **ListWrapper<>** where the generic type is the type you want to store in the list. For instance, if you want to store **lists that store integers** in a queue, write this:

```
public SerializedQueue<ListWrapper<int>> queue;
```

To access the list in the wrapper, just use ListWrapper<>.List

Do not, I repeat, **DO NOT** Just store the list like this, **it is going to complain**.

```
public SerializedQueue<List<int>> queue;
```

← NO!

*You can manually cast a list wrapper to a generic list, or the other way around.

Serialised Queue

Instead of **Queue<>**, you can use the **SerializedQueue<>** in the namespace **TheBlackCat.SerialisedDS** to serialise your queue.

```
public SerializedQueue<string> queuedNames;
```

After that, you will be able to view and edit the queue in the inspector.

← Next Dequeue

Elements are enqueued to the bottom and dequeued from the top, in this direction

← Last Enqueued

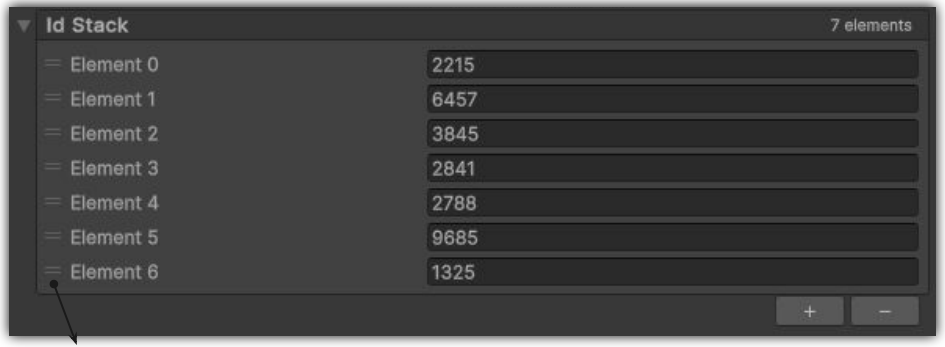
Will become undraggable in Play Mode

Serialised Stack

Instead of **Stack<>**, you can use the **SerializedStack<>** to serialise your stack.

```
public SerializedStack<int> idStack;
```

After that, you will be able to view and edit the stack in the inspector:



Will become undraggable in Play Mode

Elements are pushed to the bottom and popped from there.

← Last Pushed, Next Pop

Serialised Priority Queue - Declaration

I think?

I found that I still can't use the Priority Queue in unity, so I also implemented one myself.

You can use the **SerializedPriorityQueue<,>** where the 1st parameter is the element and the 2nd parameter is the priority.

```
public SerializedPriorityQueue<char, int> charPriorityQueue;
```

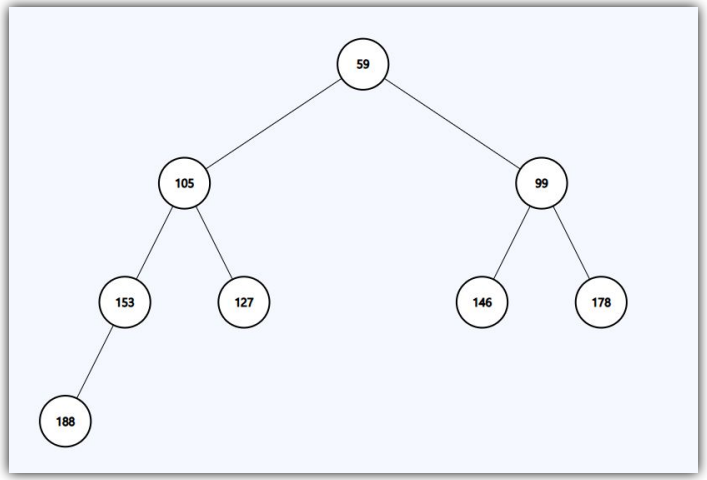
Here, the char is the element and the int determines the priority.

*The priority queue can only be modified by code. You can only view it in the inspector.
*The priority queue makes use of the min-heap data structure, the elements are **NOT** sorted.

Serialised Priority Queue - Example

← Added elements in code, elements are sorted by priorities

Visualisation of the queue below ↓



Serialised Priority Queue - Comparer

You can assign a comparer when creating a priority queue. This is useful when you are using custom class objects for the priority and you want custom comparing behaviour.

I wrote a custom comparer that inverts the comparison, the **InverseComparer<>**. The parameter is the type you use for the priority.

To use the inverse comparer, you can instantiate a new comparer:

```
public SerializedPriorityQueue<char, int> charPriorityQueue = new (new InverseComparer<int>());
```

Since this priority queue is a **min-heap**, you can use this comparer to invert the min-comparison if you want a **max-heap** instead. You can pass your comparer into the inverse comparer so it inverts your custom comparer or call **Invert()** directly from the comparer.

```
public SerializedPriorityQueue<char, int> charPriorityQueue = new (your_comparer.Invert());
```

Serialised Tuples

It’s tricky to serialise tuples, so I created structs that act like tuples.

To use serialised tuples, use **SerializedTuple<>**. The tuple can store up to 8 values at most.

Serialised tuples can be automatically converted to c# tuples, and the other way around as well.

Serialised Tuples - Example

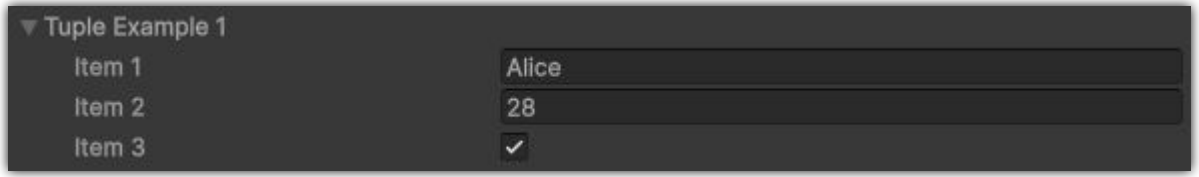
Say you want to declare a tuple with 3 items, you can create the tuple like this:

```
public SerializedTuple<string, int, bool> tupleExample1 = SerializedTuple.Of("Alice", 1, true);
public SerializedTuple<string, int, bool> tupleExample2 = ("Bob", 2, false);
```

*Use **SerializedTuple.Of()** to create tuples.

Since, like I said, a c# tuple can be cast to a serialised tuple, this approach is usually preferred due to shorter code

And it will be displayed in the inspector



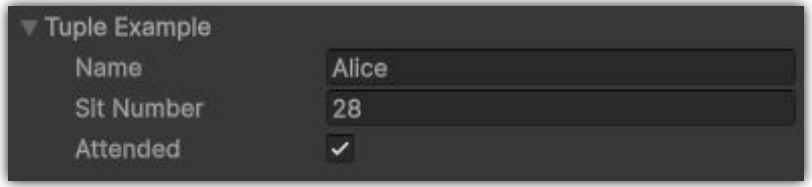
If you need to access them, use **tupleExample.Item1/item2/item3/etc.** just like how you would access items in a normal tuple.

Serialised Tuples - Display Names

If you want custom display names, you can use the **SerializedTuple** attribute. Pass in the display names for every item in the tuple (the number of arguments must be exactly the same as the number of items).

```
[SerializedTuple("Name", "Sit Number", "Attended")]
public SerializedTuple<string, int, bool> tupleExample = ("Alice", 28, true);
```

And the display names will be change →



*Only works for **globally** declared tuples.

Serialised Hash Set

Instead of **HashSet<>**, you can use the **SerializedHashSet<>** to serialise your hash set.

```
public SerializedHashSet<int> mySet;
```

After that, you will be able to view the hash set in the inspector.

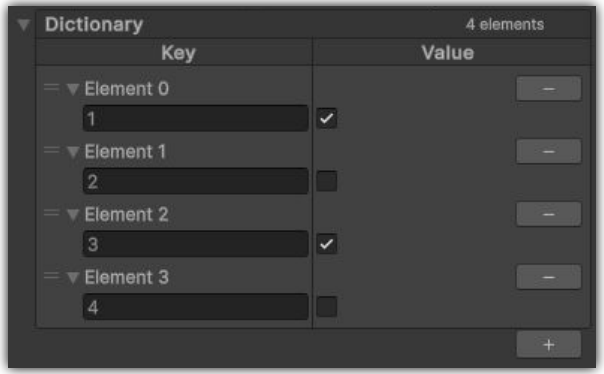
The hash set is also only modifiable through code. In the inspector, it will appear as a list that never has duplicate items.

Serialised Dictionary

Dictionaries are also serialisable in this tool. Instead of **Dictionary<,>**, you can use **SerializedDictionary<,>** to serialise your dictionaries.

```
public SerializedDictionary<int, bool> dictionary;
```

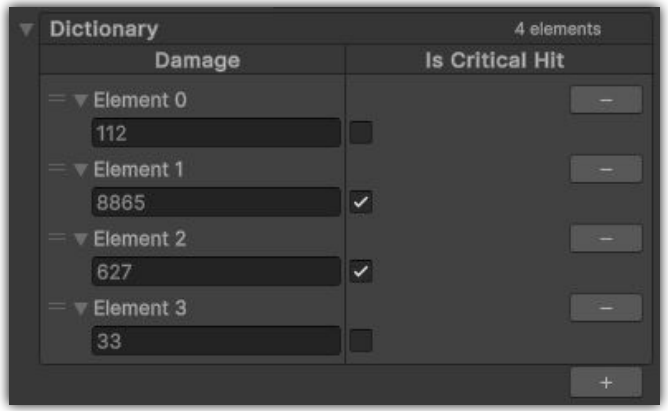
After that, you will be able to view and edit the dictionary in the inspector:



Serialised Dictionary - Header Labels

You can rename the labels in the header customly as well. Add the **SerializedDictionary** attribute and provide 2 strings, left for the key and right for the value.

```
[SerializedDictionary("Damage", "Is Critical Hit")]
public SerializedDictionary<int, bool> dictionary;
```



← Header labels are renamed

Serialised Dictionary - Duplicate Keys Warning

If there are duplicate keys in a dictionary, it will warn you in the inspector.

Orange indicates it is the first occurence in the dictionary. Red indicates the key is a duplicate, and will not be added into the dictionary when the game starts.

Ordered Dictionary

Another type of dictionary which Unity doesn't have support for yet — type-safe Ordered Dictionary.

The ordered dictionary currently available is type-unsafe, so I made a type-safe one. You can use it by declaring an **OrderedDictionary<,>**.

```
[SerializedDictionary("Individual", "Description")]
public OrderedDictionary<IndividualIdentity, string> peopleDescriptions;
```

This dictionary uses the same interface as the original serialised dictionary in the inspector.

Serialised Dictionary - Equality Comparer

Custom equality comparer also works on serialised dictionaries.

If you have a custom comparer, just pass that into the serialised dictionaries on declaration.

For example, I have a class **IndividualIdentity**, and I want it to compare equality through an int attribute in the class.

```
[System.Serializable]
10 個參考
public class IndividualIdentity
{
    public string firstName;
    public string lastName;
    public int id;
}

1 個參考
public class PersonIdentityEqualityComparer<T> : IEqualityComparer<T> where T : IndividualIdentity
{
    0 個參考
    public bool Equals(T x, T y)
    {
        if (x is IndividualIdentity personX && y is IndividualIdentity personY)
        {
            return personX.id == personY.id;
        }
        return false;
    }

    0 個參考
    public int GetHashCode(T obj)
    {
        int hash = 13;
        hash = hash * 7 ^ ((obj as IndividualIdentity)?.id.GetHashCode() ?? 0);
        return hash;
    }
}
```

*Equality comparer also works on hash sets.

Serialised Dictionary - Equality Comparer

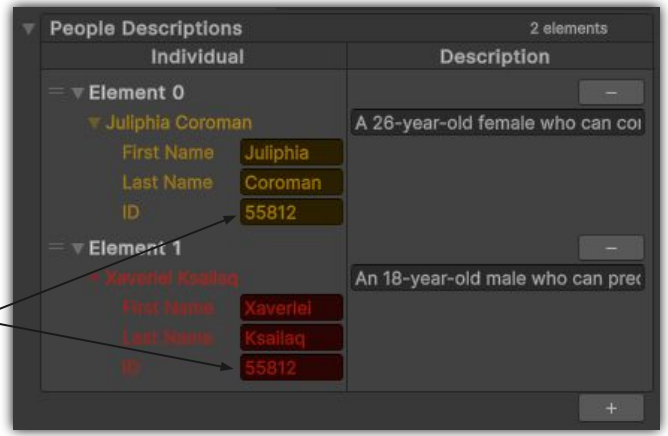
Then I pass the comparer into the ordered dictionary like this:

```
[SerializedDictionary("Individual", "Description")]
public OrderedDictionary<IndividualIdentity, string> peopleDescriptions =
    new (new PersonIdentityEqualityComparer<IndividualIdentity>());
```

Normally, no duplicate warnings will occur because the object references are different.

But since the comparer compares the int id instead, people with the same id will be considered equal.

Different names, but same IDs



Huge thanks for noticing my assets, huger thanks for even reading until here.

To learn more about each collection, you can refer to this **document**.

This asset is not perfect. Any suggestions, bug reports, reviews and advice will be highly appreciated.

If you have any problems, please contect me through:
Email: heinokchow314@gmail.com
Github: https://github.com/The-best-cat/Unity_asset-Serialised_data_structures

Good luck with your projects, and have a nice day.