Serialised Data Structures

By The Black Cat

The following features are exclusive to the Full Pack:

- Serialised Priority Queue
- Serialised Dictionary
- Serialised Ordered Dictionary
- Serialised Hash Set
- Serialised Binary Search Tree

If you haven't already, I do recommend you purchasing it, it's only $5.99 and you will gain access to all features!

Content

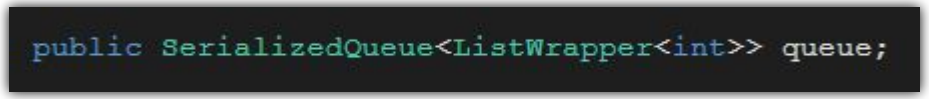
Read Before Everything!!

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.

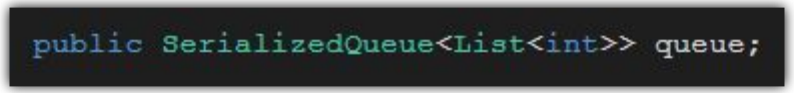
To wrap a list, use **ListWrapper<>** where the generic type is the type you want to store in the list. For instance, if you want to store **lists that store integers** in a queue, write this:



```
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!

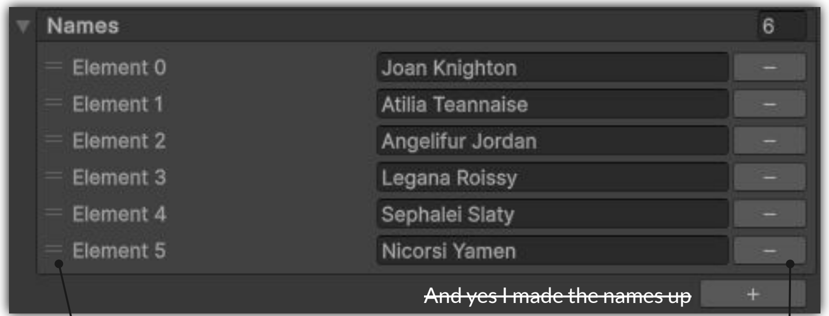
*You can manually cast a list wrapper to a generic list, or the other way around.

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.



Names	6
Element 0	Joan Knighton
Element 1	Atilia Teannaise
Element 2	Angelifur Jordan
Element 3	Legana Roissy
Element 4	Sephalei Slaty
Element 5	Nicorsi Yamen

~~And yes I made the names up~~

← Next Dequeue

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

← Last Enqueued

Will become undraggable in Play Mode

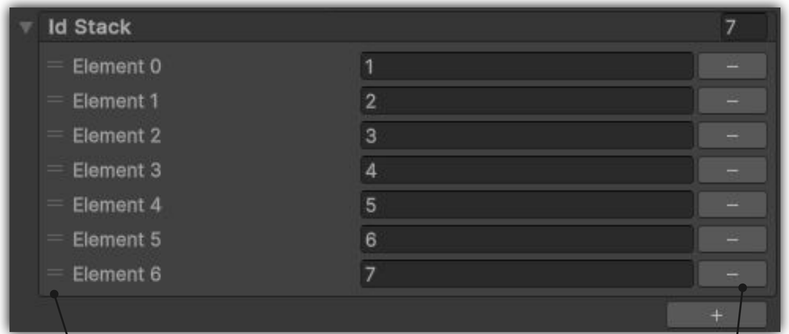
You can remove any element in Editor mode. But once you enter Play mode, you can only remove the first element (next dequeue)

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:



Id Stack	7
Element 0	1
Element 1	2
Element 2	3
Element 3	4
Element 4	5
Element 5	6
Element 6	7

Elements are pushed to the bottom and popped from there.

← Last Pushed, Next Pop

Will become undraggable in Play Mode

You can remove any element in Editor mode. But once you enter Play mode, you can only remove the last element (next pop)

Serialised Tuples

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

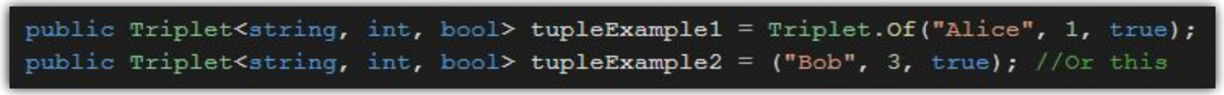
These are the names of tuples. A tuple can store up to 8 items:

Pair<T1, T2> — Stores 2 items
Triplet<T1, T2, T3> — Stores 3 items
Quartet<T1, T2, T3, T4> — Stores 4 items
Quintet<T1, T2, T3, T4, T5> — Stores 5 items
Sextet<T1, T2, T3, T4, T5, T6> — Stores 6 items
Septet<T1, T2, T3, T4, T5, T6, T7> — Stores 7 items
Octet<T1, T2, T3, T4, T5, T6, T7, T8> — Stores 8 items

These serialised tuples can be automatically converted to C# tuples, and the other way around as well.

Serialised Tuples - Example

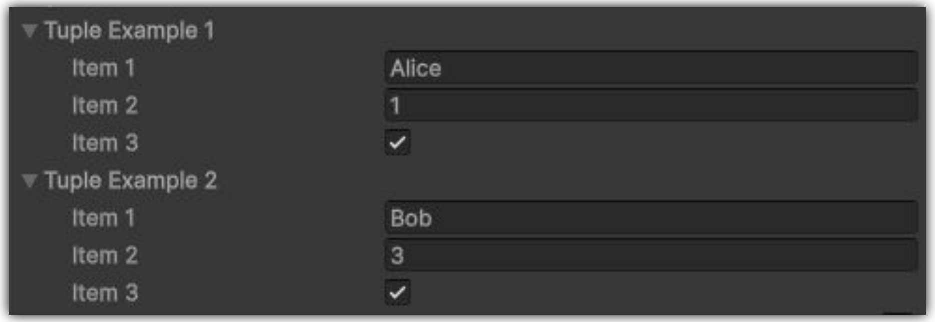
Say you want to declare a tuple with 3 items, you can create the tuple like this:



```
public Triplet<string, int, bool> tupleExample1 = Triplet.Of("Alice", 1, true);
public Triplet<string, int, bool> tupleExample2 = ("Bob", 3, true); //Or this
```

To create a tuple, call the **Of()** method from that tuple. You don't need to write the type parameters again.

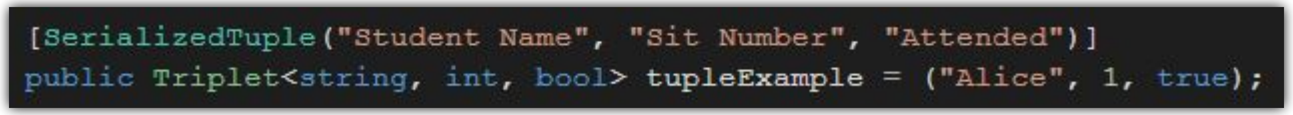
And it will be displayed in the inspector:



```
▼ Tuple Example 1
    Item 1          Alice
    Item 2          1
    Item 3          ✓
▼ Tuple Example 2
    Item 1          Bob
    Item 2          3
    Item 3          ✓
```

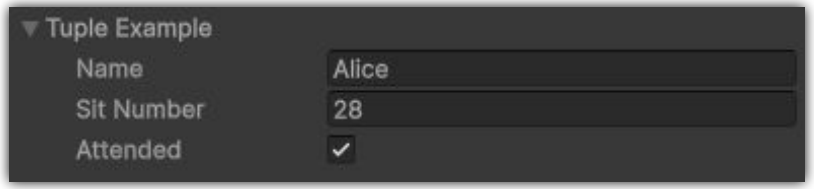
Serialised Tuples - Display Names

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("Student Name", "Sit Number", "Attended")]
public Triplet<string, int, bool> tupleExample = ("Alice", 1, true);
```

And the display names will be change ⟶



```
▼ Tuple Example
    Name          Alice
    Sit Number    28
    Attended      ✓
```

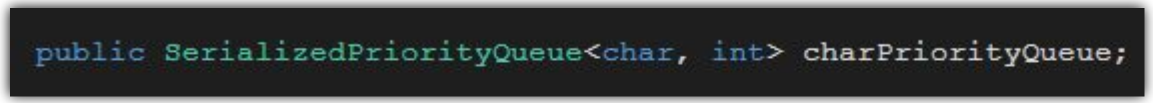
*Only works for **globally** declared tuples.

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.

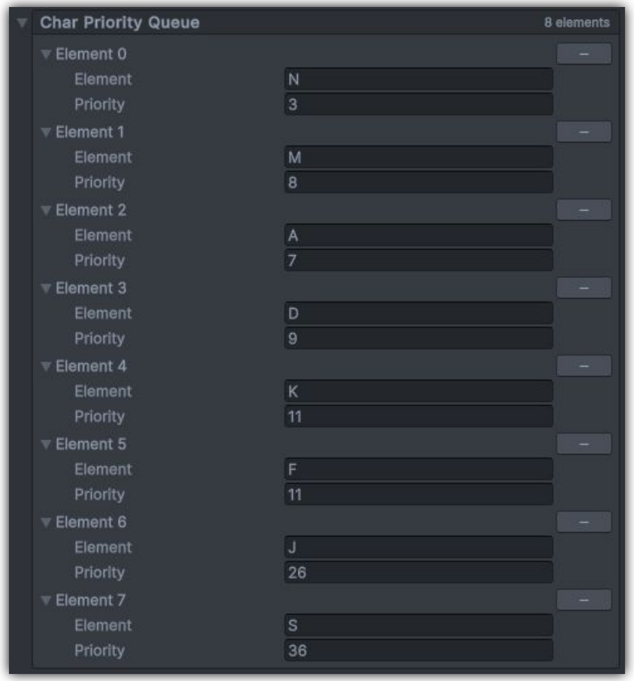


```
public SerializedPriorityQueue<char, int> charPriorityQueue;
```

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

*The priority queue makes use of the min-heap data structure, the elements are **NOT** sorted.
*Anyone think I should rename the collection to **SerializedPQ** (?) Or **SerializedHeap** (??)

Serialised Priority Queue - Example

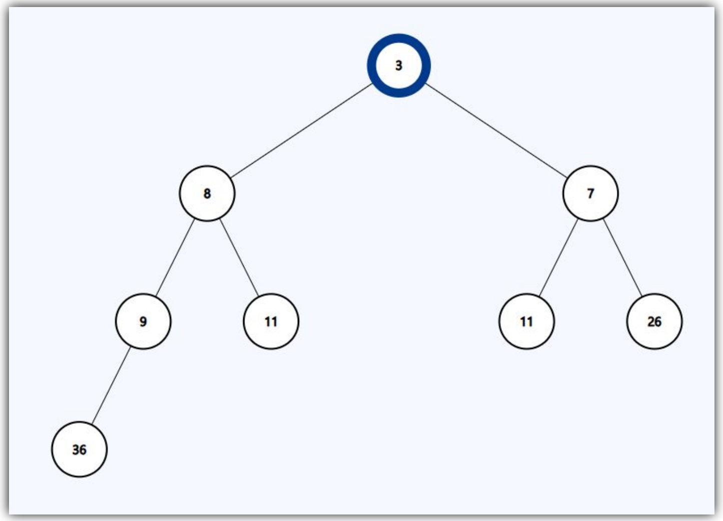


Char Priority Queue — 8 elements

	Element	Priority
Element 0	N	3
Element 1	M	8
Element 2	A	7
Element 3	D	9
Element 4	K	11
Element 5	F	11
Element 6	J	26
Element 7	S	36

← Elements are ordered by priorities

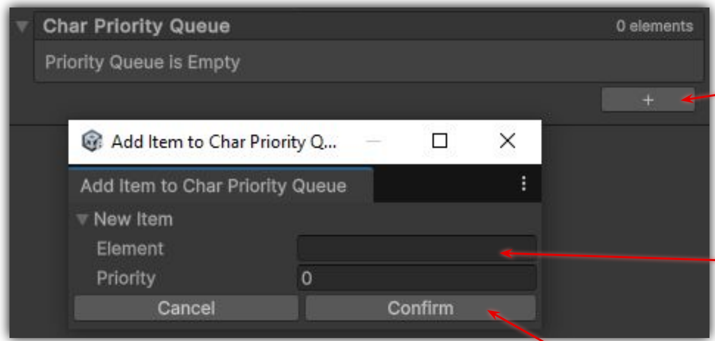
Visualisation of the queue below ↓



Serialised Priority Queue - Insert Items

Since the priority queue reorders the items by itself, adding an item to the priority queue in the inspector is a little different from other collections. When you press “**+**”, a window will show up. You can enter the content of the new item in that window, then press **Confirm** to add it to the priority queue.

*For unknown reason, this doesn’t work with **char** variables. Use a string instead.



1. Press **+**

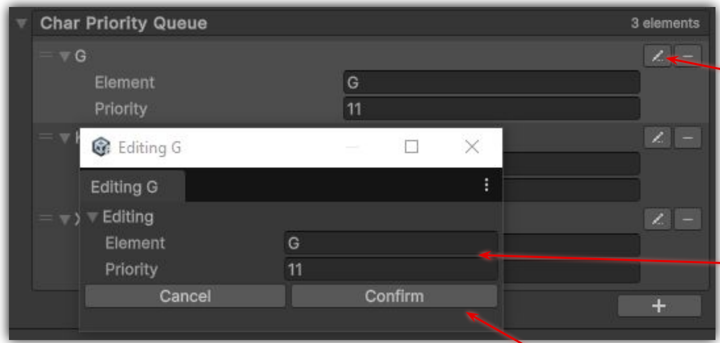
2. Enter content

3. Press **Confirm**

Serialised Priority Queue - Edit Items

Although you can directly edit items, it's recommended to press the edit button and edit the items in the window just like adding an item. Press (a little blurry here) and a window will show up. Edit the item, then press Confirm to change the item in the priority queue.

*For unknown reason, this doesn't work with **char** variables. Use a string instead.



G
Element: G
Priority: 11

Editing G
Editing
Element: G
Priority: 11

1. Press this pencil

2. Enter content

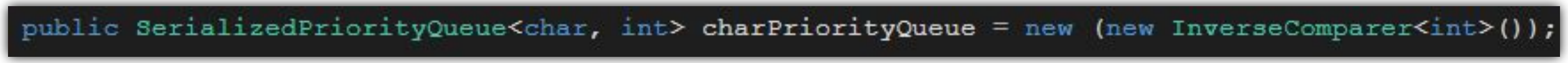
3. Press **Confirm**

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.

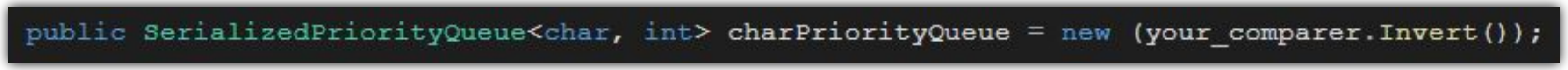
I wrote a custom comparer that inverts the comparison, the **InverseComparer<>**. The parameter is the type you use for the priority.

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



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

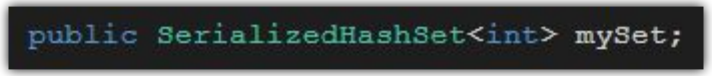
Since this priority queue is a **min-heap**, you can use this comparer to invert the min-comparison if you want a **max-heap** instead. You can pass your comparer into the inverse comparer so it will invert your custom comparer or call **Invert()** directly from the comparer.



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

Serialised Hash Set

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

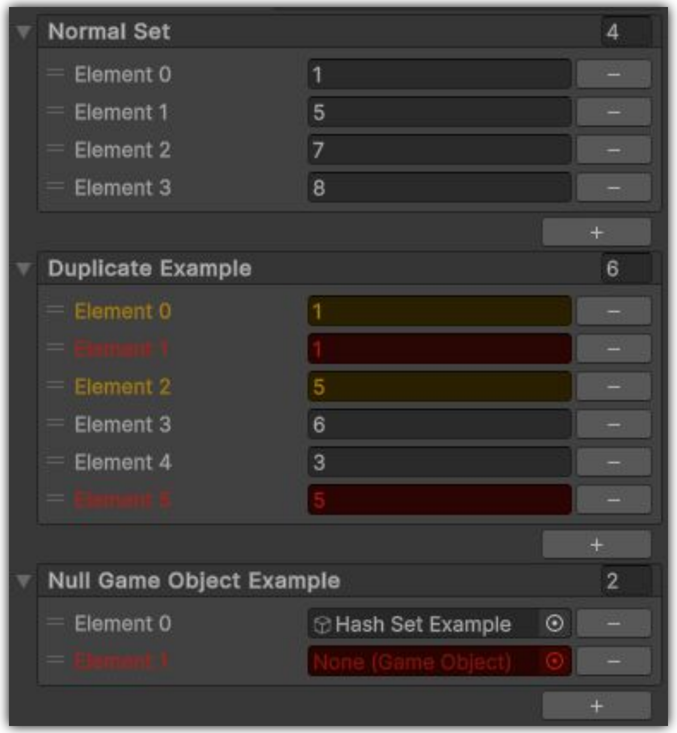


```
public SerializedHashSet<int> mySet;
```

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

It will highlight the invalid itemsfor you. **Orange** indicates it is the first occurence in the set. **Red** indicates it is a duplicate, and will be removed from the set when the game starts.

It also highlights invalid fields in **Red** like null game objects.



Normal Set 4

Element 0	1
Element 1	5
Element 2	7
Element 3	8

Duplicate Example 6

Element 0	1
Element 1	1
Element 2	5
Element 3	6
Element 4	3
Element 5	5

Null Game Object Example 2

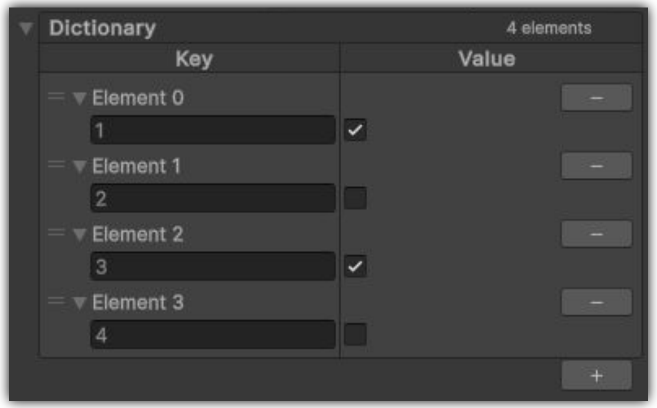
Element 0	Hash Set Example
Element 1	None (Game Object)

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:

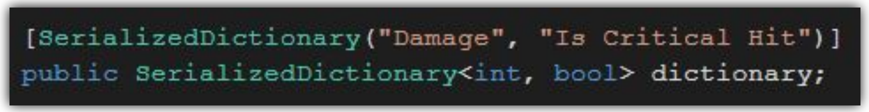


Dictionary 4 elements

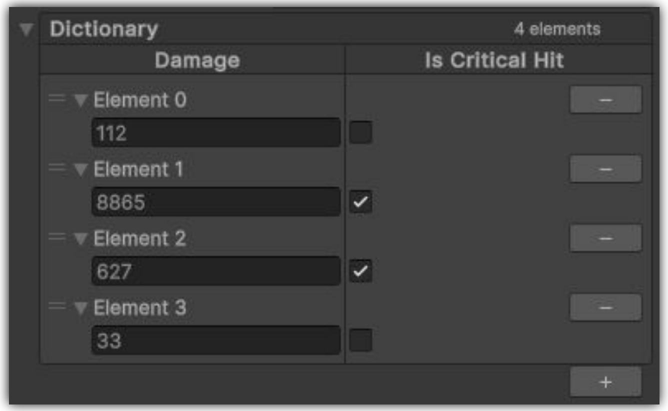
Key	Value
Element 0: 1	✓
Element 1: 2	
Element 2: 3	✓
Element 3: 4	

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.



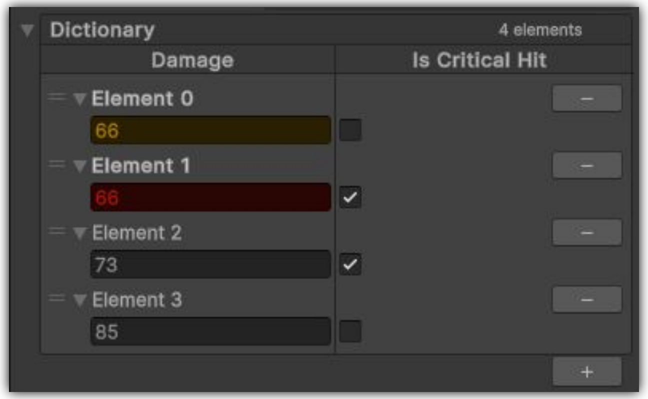
```
[SerializedDictionary("Damage", "Is Critical Hit")]
public SerializedDictionary<int, bool> dictionary;
```



← Header labels are renamed

Serialised Dictionary - Invalid Keys Warning

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

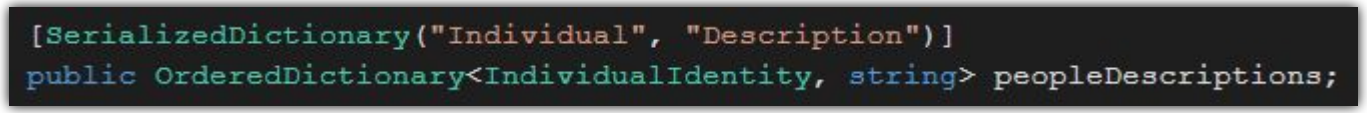


Orange indicates the key is the first occurence in the dictionary. **Red** indicates the key is a duplicate, and will be removed from the dictionary when the game starts. It also warns you about null keys in **Red**.

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, and basically just stores the keys and values in the order you added them.

Serialised Sorted Dictionary

Well Unity does support **SortedDictionary**, but it doesn't serialise it.

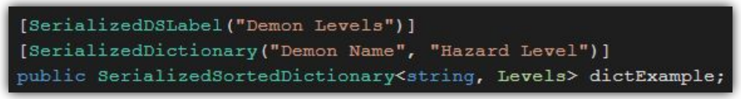
The name is a little bit too long isn't it

If you want to use a serialisable sorted dictionary, you can use my **SerializedSortedDictionary<K, V>**. Again, it reorders the item, so you need to add and edit items in a window.

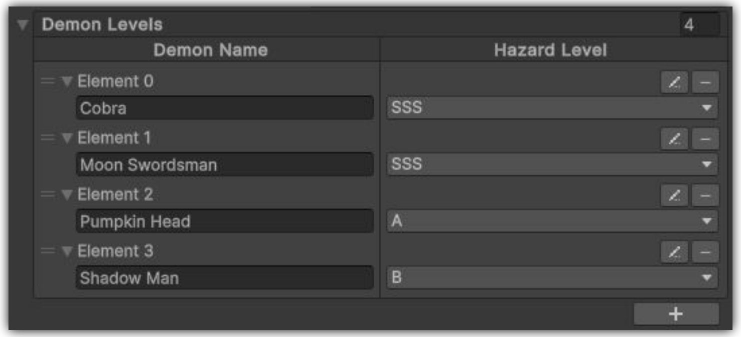
Let's use the same example for the **Dictionary**. In the normal **Dictionary**, the keys are not sorted. But in here, the keys are sorted.

And if you attempt to add a new key with that already presents in the sorted dictionary, it will just ignore you.

And if you try to edit a key to a key that already exists in the sorted dictionary, one of them will just disappear.



```
[SerializedDSLabel("Demon Levels")]
[SerializedDictionary("Demon Name", "Hazard Level")]
public SerializedSortedDictionary<string, Levels> dictExample;
```



Demon Name	Hazard Level
Element 0	
Cobra	SSS
Element 1	
Moon Swordsman	SSS
Element 2	
Pumpkin Head	A
Element 3	
Shadow Man	B

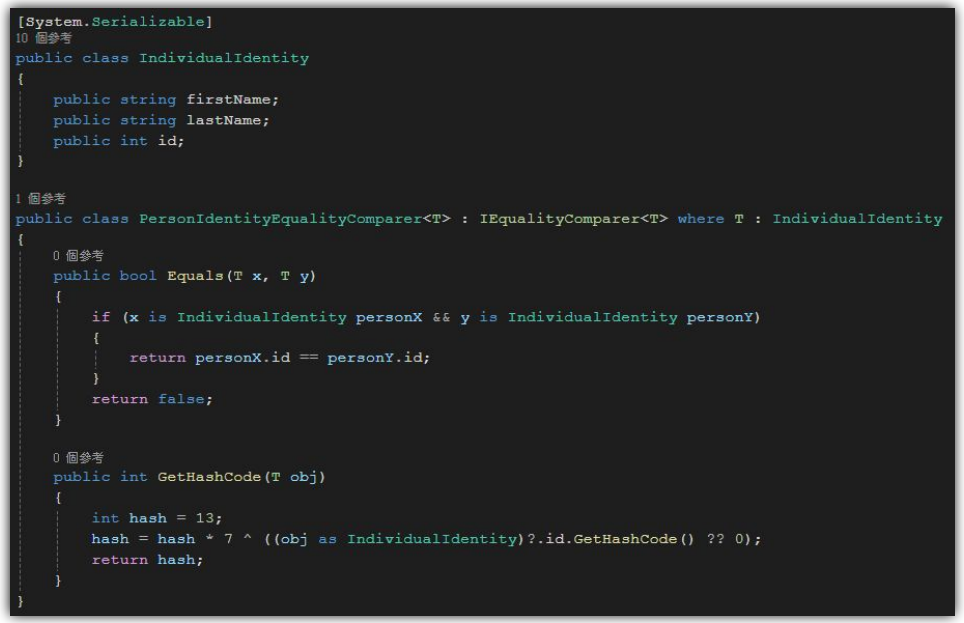
Serialised Dictionary - Equality Comparer

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.

*Equality comparer also works on hash sets.



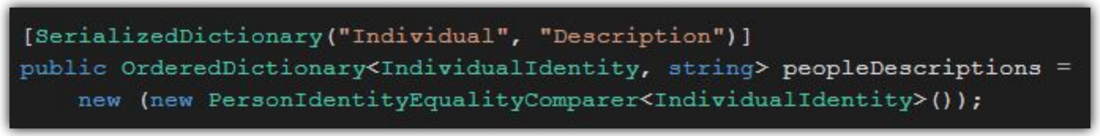
```
[System.Serializable]
10 個參考
public class IndividualIdentity
{
    public string firstName;
    public string lastName;
    public int id;
}

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;
    }
}
```

Serialised Dictionary - Equality Comparer

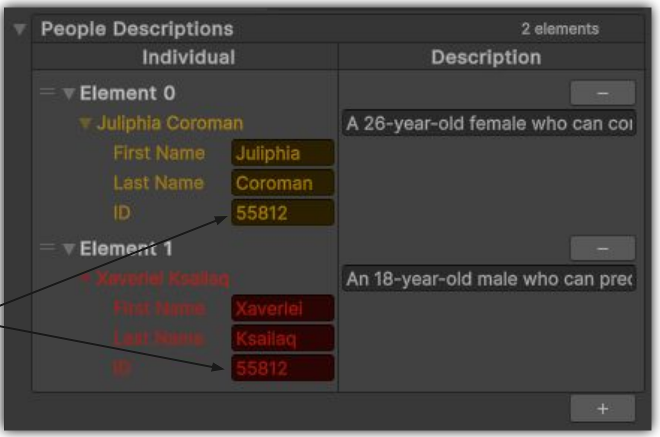
Then I pass the comparer into the ordered dictionary like this:



```
[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.



People Descriptions 2 elements

Individual	Description
Element 0	
Juliphia Coroman	A 26-year-old female who can co
First Name: Juliphia	
Last Name: Coroman	
ID: 55812	
Element 1	
Xaverlei Ksailaq	An 18-year-old male who can pre
First Name: Xaverlei	
Last Name: Ksailaq	
ID: 55812	

Different names, but same IDs

Serialised Tree

I have implemented a serialisable binary search tree — **Red Black Tree**, more specifically. This is useful when you want to have a collection that is always sorted.

There is a built-in C# collection that uses a binary search tree. That is the **SortedSet<>**. How is the sorted set differ from my binary search tree, you may ask?

1. The sorted set doesn't allow duplicates. If you attempt to add a duplicate, it doesn't do anything. Mine allows duplicates, and the duplicates are stored in the same node.
2. BSTs usually don't support **indexing**. I have implemented a way so you can access items by index efficiently.
3. And the most importantly, mine is **serialisable**.

Serialised Tree - Declaration

To declare a serialised BST, you can use **TreeSet<>**.

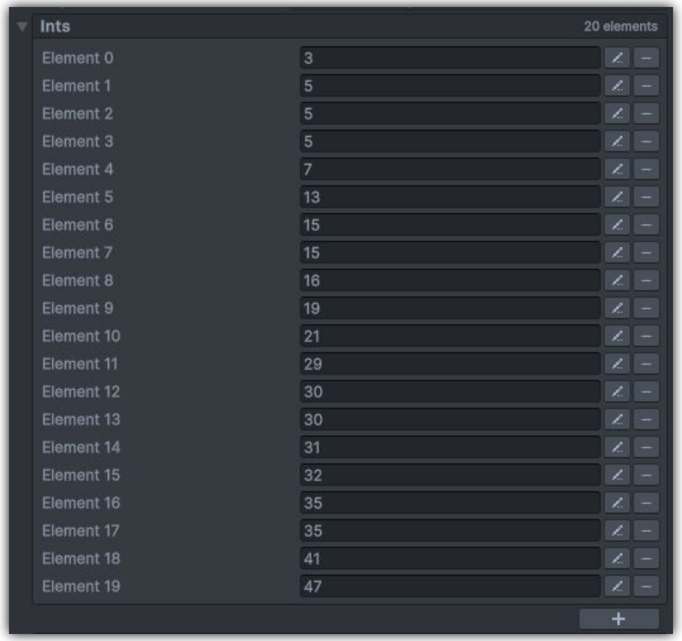
```
public TreeSet<int> ints = new TreeSet<int>();
```

In the inspector, the items will appear in sorted order→

You can also use your custom comparer for the tree.

Same as priority queue, the tree reorders the items. When adding a new item, press “**+**”, enter content in the window, then press **Confirm** to add it to the tree.

To edit an item, press the little pencil button and edit the item in the window, then press **Confirm**.



Ints	20 elements
Element 0	3
Element 1	5
Element 2	5
Element 3	5
Element 4	7
Element 5	13
Element 6	15
Element 7	15
Element 8	16
Element 9	19
Element 10	21
Element 11	29
Element 12	30
Element 13	30
Element 14	31
Element 15	32
Element 16	35
Element 17	35
Element 18	41
Element 19	47

List<T> Extension - Shuffle

You can finally shuffle the list without needing to implement the function yourself!

To simply shuffle the list, just call **List<T>.Shuffle()** and it will shuffle the entire list.

If you want to start at a specific index, simply pass the index into the function, like **List<T>.Shuffle(3)** and it will shuffle the list from index 3 to the end of the list.

If you want to just shuffle a specific amount of items, pass the number into the function as well. For instance, **List<T>.Shuffle(3, 6)** will shuffle items at indices 3, 4, 5, 6, 7 and 8.

List<T> Extension - Get Random Item

You can retrieve a random item from the list by calling **List<T>.GetRandom()**. This function will return a random item from the entire list.

Same as **Shuffle()**, you can specify the starting index and the range. For instance, **List<T>.GetRandom(3)** returns a random item within index 3 to the end of the list. **List<T>.GetRandom(3, 6)** returns a random item between index 3 and index 8 (included).

You can also retrieve multiple random items. It requires 2 more arguments, one is the count of item you want to retrieve, and one is a boolean indicating whether you allow duplicates or not (default to true).

Careful not to mix this up with GetRandom()

For example, **List<T>.GetRandoms(5)** returns 5 items randomly picked from the entire list, allowing duplicates. To disallow duplicates, simply pass false. **List<T>.GetRandoms(3, 3, 6, false)** returns 3 random items between index 3 to index 8, without repeating items, etc.

List<T> Extensions - Check Sorted

You can check if a list is sorted with just one function call now.

To check if a list is sorted in ascending order, use **List<T>.SortedInAscending()**.

To check if a list is sorted in descending order, use **List<T>.SortedInDescending()**.

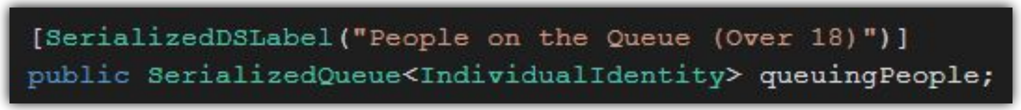
If your list is storing custom data and you have a custom comparing method for them, you can pass your comparer to the function. It will then use your comparer to compare the items. Otherwise, it uses the default comparer.

Attribute - Collection Display Names

Not only can you name the dictionary and the tuples, you can name any type of serialisable collections in this asset as well.

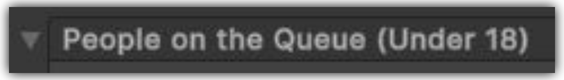
You can use the Attribute **SerializedDSLabel**, simply provide a string as the name.

For example, maybe your serialised queue is named “queuingPeople” in code, but you want to have a different display name in the inspector. Maybe you want to add brackets!



```
[SerializedDSLabel("People on the Queue (Over 18)")]
public SerializedQueue<IndividualIdentity> queuingPeople;
```

And it will be displayed as:



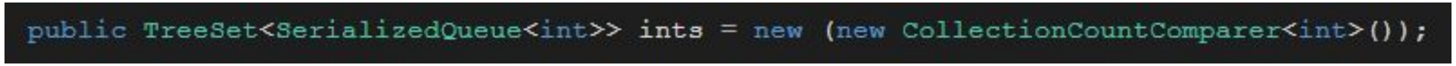
People on the Queue (Under 18)

Comparer - Collection Count

I wrote a comparer that can compare the number of elements in 2 collections. It can be any collection, as long as they are inherited from the **ICollection** interface.

You can instantiate a new **CollectionCountComparer<T>** where T is the type of value stored in the collection.

For instance, if you need to compare queues that store integers, let’s say you are storing them in the **TreeSet<SerializedQueue<int>>**:



```
public TreeSet<SerializedQueue<int>> ints = new (new CollectionCountComparer<int>());
```

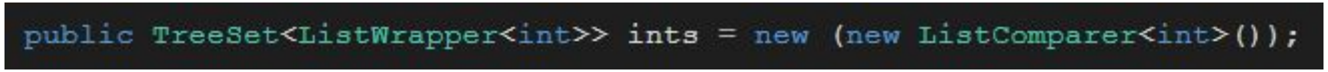
Note that you only need to write **int** in the comparer generic type parameter.

The collection with smaller Count (fewer elements stored) will be considered smaller.

Comparer - List Content

I also wrote a comparer that compare items in lists. This comparer is only usable by **List<>** and **ListWrapper<>** since it needs to access every item in both collections.

Let's say you are storing lists of integers in the **TreeSet<ListWrapper<int>>**:



```
public TreeSet<ListWrapper<int>> ints = new (new ListComparer<int>());
```

Just like **CollectionCountComparer**, you only need to write **int** in the generic type parameter.

The comparer will compare the 1st item in both lists, the list with the smaller item will be immediately considered smaller. If the items are equal, then it carries on to the next item.

If the comparer reaches the end of a list and still haven't decided which is smaller, then the list with more items is considered greater. Otherwise, both lists are equal.

Comparer - Extension

I implemented this just for the convenience of my development, but you can also use them.

If you happen to need to use **Compare()** from a comparer, you can use the following methods instead:

IComparer<T>.LessThan(T x, T y);
IComparer<T>.LessThanOrEqualTo(T x, T y);
IComparer<T>.EqualTo(T x, T y);
IComparer<T>.GreaterThanOrEqualTo(T x, T y);
IComparer<T>.GreaterThan(T x, T y);

The names are quite self-explanitory. Instead of **Compare() <=> 0**, the methods above is way easier to use and understand, I think. Be careful not to fix up **Equals()** and **EqualTo()**, though.

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.