SortedIteration — Live Coding Workshop

Table of Contents

- Generics repetition
    - With type constraint
- `IEnumerable`
    - **State Machines**
- `IComparable`

1. Intro

1.1. Interfaces Everywhere

First of all: to realize how prevalent interfaces are I recommend you check how many the humble `int` implements — I'm sure you'll be surprised!

▶ Reveal if you are too lazy to check yourself

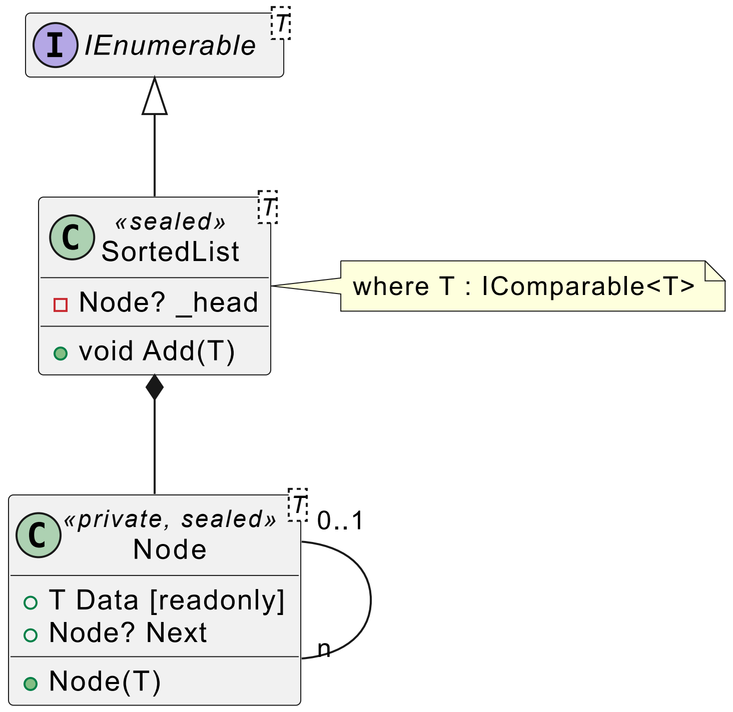
1.2. Scenario Description

Let's apply what we learnt about interfaces to a simple example.

- Implementing a class which allows its instances to be *compared* to each other
- Implementing a list which keeps items in sorted order and can be *enumerated*

As scenario, we will use the high score of a video game.

2. SortedList



2.1. Implementation

We will do the implementation in two parts:

- First, we implement a simple, linked list which inserts items in-order
  - It will, of course, be *generic*
  - Item order is checked by calling the `CompareTo` method
- Then we will enable our list to be *enumerated*
  - So that it can be used in a `foreach` for example

2.1.1. Basic list structure

In case you've forgotten how to implement a basic, generic linked list, here is a sample implementation:

SortedList

CSHARP

```csharp
public sealed class SortedList<T> 1
{
    private Node? _head; 2

    private sealed class Node 3
    {
        public Node(T data) 4
        {
            Data = data;
        }

        public T Data { get; } 4
        public Node? Next { get; set; } 5
    }
}
```

1 Define a generic class with type parameter `T`

2 Keep a reference to the *head* node

3 `Node` is an inner class, so it will implicitly have access to type parameter `T` ⇒ we don’t have to make it `Node<T>`

4 Data will be readonly

5 Next pointer can be changed

2.1.2. Insert in Order

We now have to implement the `Add` method. As stated before we will rely on the `CompareTo` method to decide at which position in the list the new node has to be added.

SortedList — Add

CSHARP

```csharp
public void Add(T data)
{
    if (_head is null) 1
    {
        _head = new Node(data);

        return;
    }

    if (data.CompareTo(_head.Data) < 0) 2
    {
        _head = new Node(data)
        {
            Next = _head
        };

        return;
    }

    var current = _head;
    while (current.Next is not null
           && data.CompareTo(current.Next.Data) > 0) 3
    {
        current = current.Next;
    }

    current.Next = new Node(data) { Next = current.Next }; 4
}
```

1 If the head is still `null` we can set the new node immediately

2 If the new value has to be ranked before the current head, we can do that

3 Otherwise, we need to seek for the proper location

4 Finally, we position the new node — in between two existing nodes or at the end

Now you have a compile error! 'Cannot resolve symbol CompareTo' — what's going on?

- We have defined `SortedList` as a generic class, so at this point *any* data type coule be used for `T`, e.g. `int`, `bool`, `Potato` or `List<Tomato>`.
- How can the compiler *ensure* that all of these types have a `CompareTo` method?

  - It *cannot*, thus we get this compiler error — some types may be ok, others not ⇒ we always need a *deterministic* situation
- However, we do know which types are *guaranteed* to have a `CompareTo` method: all those which implement `IComparable`!
- ⇒ we have to tell the compiler, that only such types will be allowed as type parameter for `T` in our `SortedList`
  - That is accomplished with a *type constraint*

SortedList — Type Constraint

CSHARP

```csharp
public sealed class SortedList<T> where T : IComparable<T> 1
```

1 We specified that `SortedList` may *only* be used with (any) type which implements `IComparable`

Now it is guaranteed that all values *will* have a `CompareTo` method and the error goes away.

> This is an *important* conclusion — sit back and think carefully if you understand why this works and if you feel like you could apply such a technique yourself ⇒ ask if you have questions!

2.1.3. Allow to enumerate

The next big step is to allow our own class `SortedList` to be used in a `foreach` like so:

CSHARP

```csharp
var list = new SortedList<double>();
list.Add(3.4D);
list.Add(4.3D);

foreach(double value in list)
{
    Console.WriteLine(value);
}
```

- Remember what you learnt about `foreach`: it works on any `IEnumerable` by calling and using the `IEnumerator` for us
- ⇒ we have to implement `IEnumerable`
  a. Add it to the class definition
  b. Then have your IDE generate the required members

SortedList — Implement IEnumerable

CSHARP

```csharp
public sealed class SortedList<T> : IEnumerable<T> where T : IComparable<T>  1
...
    public IEnumerator<T> GetEnumerator() => throw new NotImplementedException();  2

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();  3
...
```

1 `IEnumerable<T>` has been added as *an implemented interface* to the class definition — mind: those come *before* the type constraint(s)

2 As we've discussed, `IEnumerable` works by using an `IEnumerator` — we'll have to implement one!

3 The interface `IEnumerable<T>` also contains a non-generic version for compatibility ⇒ best practice is to simply redirect to the generic implementation

Enumerator as State Machine

It is important to understand, that *enumerable* does not necessarily mean a (fixed size) collection. Consider it to be a 'step-wise retrieval of object from a stream until that stream is exhausted'.

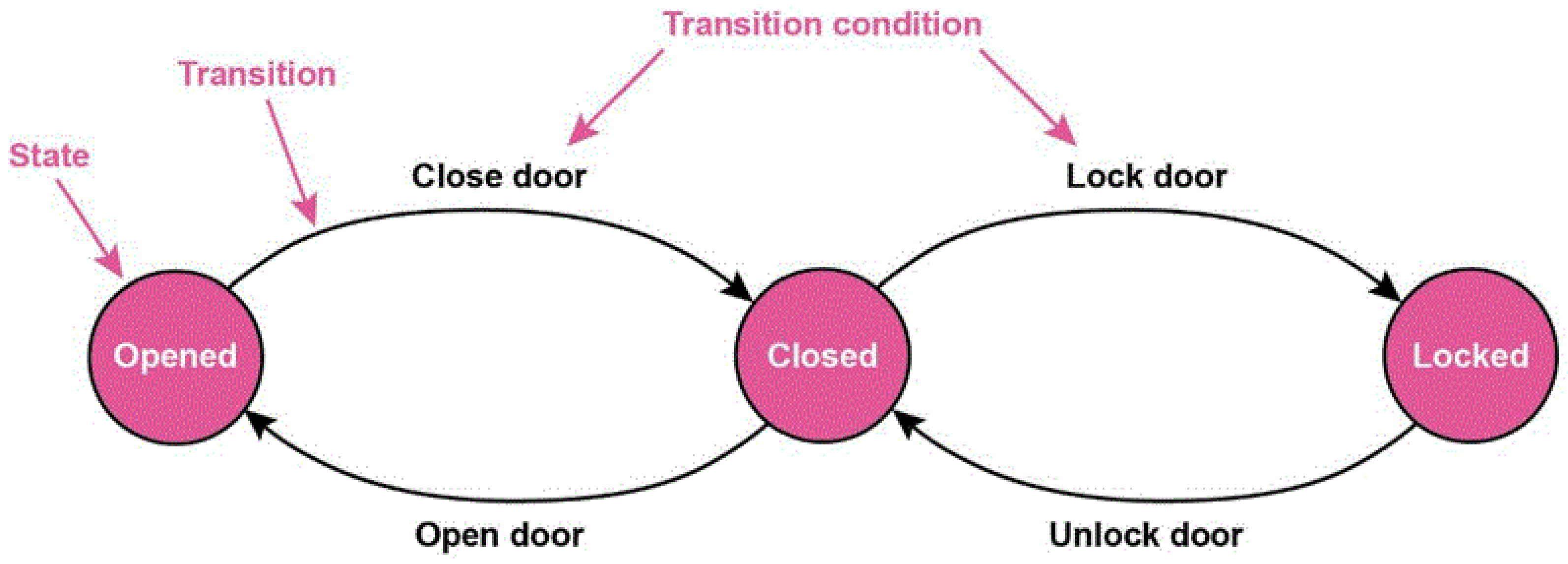
```
Imagine you are helping your uncle mount a shelf to the wall.
He asks you to hold the 5 screws he'll need and hand them to him - one by one - whenever he tells you to.
After getting the shelf into position he'll ask for the first screw and you'll hand it over.
You uncle wil need a couple of seconds to screw it in.
Then he'll ask for the second screw, which you'll hand to him.
...
You handed your uncle the fifth and final screw.
Now you are standing around, empty handed, watching him work.
Suddenly your uncle asks for another screw - but you are out already, your screw-storage is exhausted!
You have to tell him, that you don't have any more screws, the shelf will have to do with five.
```

In the above scenario *you* have been an *enumerator*!

For a collection, the enumerator only hands out references, so it will not 'run out', but instead it has to *remember* which was the last position it handed out, so the next time it gets asked for an item it can provide the next position — if there is still any left.

This situation — being asked for items one-by-one, with (short or long) breaks in between — makes it necessary to remember the last position/item. So **the enumerator has to save the last *state*** it was in.

That can be *accomplished by* implementing a ***state machine***.



Since implementing state machines is a quite common task, C# has a handy feature which lets the compiler do all the work for us — nice! To use this feature you will need the **yield (https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/yield) keyword** — as an exercise work through the linked documentation (there are runnable examples as well), before we discuss it further. Did you notice how the return type could be `IEnumerable` instead of `IEnumerator` directly?

▶ State Machine Example

Implementing the Enumerator

Once you know how to have the compiler generate a state machine for you, a basic enumerator implementation is trivial:

SortedList — GetEnumerator()

CSHARP
```csharp
public IEnumerator<T> GetEnumerator()
{
    var current = _head;
    while (current is not null)
    {
        yield return current.Data;
        current = current.Next;
    }
}
```

2.2. Testing

We have successfully implemented our `SortedList`. Let's test it!

Simple Test

CSHARP

```csharp
var numbers = new SortedList<int>();  1
numbers.Add(3);  2
numbers.Add(15);  2
numbers.Add(-6);  2

foreach (int i in numbers)  3
{
    Console.WriteLine(i); // -6 3 15  4
}
```

1 `int` implements `IComparable`, so we can use it

2 Adding values in random order

3 Our `SortedList` can now be used together with `foreach`

4 Sorted ouput

And because you did so well (🙄), here's something *fancy* our list can do now — without further changes!

Fancy Test

CSHARP

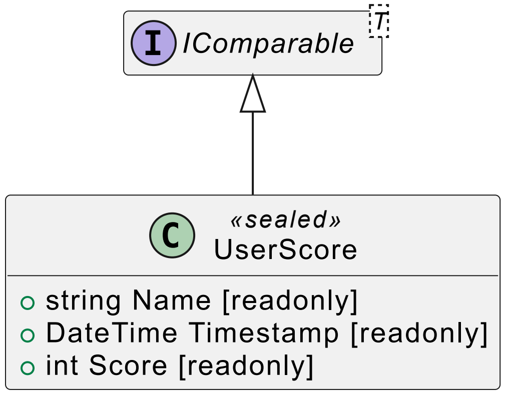
```csharp
var texts = new SortedList<string>()  1
{
    "A",
    "pizza",
    "for",
    "everyone",
    "!"
};

var sortedAndJoined = string.Join(" ", texts);  2
// "! A everyone for pizza"
```

1 *Any* class which has an `Add` method (with one item) *and* implements `IEnumerable` can implicitly be used with a collection initializer 😎

2 `string.Join` requires an `IEnumerable` as second parameter — guess which class we can pass to it now 😉

3. UserScore



Each time a user finishes a game a score is saved.

3.1. Implementation

UserScore - properties

CSHARP

```csharp
public sealed class UserScore : IComparable<UserScore>  1
{
    public UserScore(string name, DateTime timestamp, int score)  2
    {
        Name = name;
        Timestamp = timestamp;
        Score = score;
    }

    public string Name { get; }
    public DateTime Timestamp { get; }
    public int Score { get; }
}
```

1 As specified we'll have to implement all *members* of the `IComparable<T>` interface

2 Three properties with different data types, all readonly

We begin by implementing the `CompareTo` method. It has to follow these rules:

1. Highest score is ranked first
2. If two scores are equal, the earlier date is ranked first
3. If the date is *also* equal sort order is defined by alphanumerical comparison of the username

Remember what the expected return value is (by convention):

- \$0\$ — objects are equal
- \$<0\$ — this object is ranked before
  - ⇒ other object is ranked after
- \$>0\$ — this object is ranked after
  - ⇒ other object is ranked before

We used the generic version of the interface (`IComparable<UserScore>`), so the `CompareTo` method expects a `UserScore?` object, and not the non-generic `object?`

UserScore - `CompareTo`

CSHARP

```csharp
public int CompareTo(UserScore? other)
{
    if (ReferenceEquals(this, other))  1
    {
        return 0;
    }

    if (ReferenceEquals(null, other))  2
    {
        return 1;
    }

    int scoreComparison = Score.CompareTo(other.Score) * -1;  3

    if (scoreComparison != 0)  4
    {
        return scoreComparison;
    }

    int timestampComparison = Timestamp.CompareTo(other.Timestamp);  5

    return timestampComparison != 0  6
        ? timestampComparison  6
        : string.Compare(Name, other.Name, StringComparison.Ordinal);  7
}
```

1 First, we check if the other object has the *exact same* reference ⇒ it is exactly the same object, we can return `0`

2 If the other object is null — and we are not, otherwise this instance method could not be executing — we rank ourselves first

3 We use the built-in `CompareTo` of `int` — **but** we have to invert the result, because *higher* values should be ranked first

4 If the score is *not* equal we can return

5 We use the built-in `CompareTo` of `DateTime` which will get us the result for the *earlier* values ⇒ that's what we want

6 If dates are *not* equal we can return

7 As a final measure we use the built-in `Compare` method of `string` — it is a little different (static, allows definition of comparison options), but has the same (<0, 0, >0) result which we can use

As you know, when implementing `IComparable` one should always also override the *equality comparator operators*. That can be accomplished rather easy by using the `Default` instance of the default `Comparer<T,T>`, which will use our `CompareTo` method automatically.

UserScore - Comparators

CSHARP

```csharp
public static bool operator <(UserScore? left, UserScore? right) =>  1
    Comparer<UserScore>.Default.Compare(left, right) < 0;  2

public static bool operator >(UserScore? left, UserScore? right) =>  3
    Comparer<UserScore>.Default.Compare(left, right) > 0;  2

public static bool operator <=(UserScore? left, UserScore? right) =>  4
    Comparer<UserScore>.Default.Compare(left, right) <= 0;  2

public static bool operator >=(UserScore? left, UserScore? right) =>  5
    Comparer<UserScore>.Default.Compare(left, right) >= 0;  2
```

1 \$<\$ operator

2 Will use our own `CompareTo` method internally

3 \$>\$ operator

4 \$<=\$ operator

5 \$>=\$ operator

> Have you every thought about someone having to implement every little \$<\$ etc. operator you've been so casually using? In fact, we implemented our operators (above) by using operators someone else implemented 😉

3.2. Testing

Now let's bring our two pieces together and create a list of scores. We will add scores unordered and then iterate the list to see if it prints in the correct order.

Full Test

CSHARP

```csharp
var highscore = new SortedList<UserScore>();
highscore.Add(new("Susi", new(2022, 12, 15, 21, 30, 17), 180));
highscore.Add(new("Hansi", new(2023, 01, 08, 11, 12, 09), 202));
highscore.Add(new("Franzi", new(2022, 12, 16, 01, 04, 52), 180));
highscore.Add(new("Moni", new(2023, 03, 30, 14, 28, 43), 224));
highscore.Add(new("Susi2", new(2022, 12, 15, 21, 30, 17), 180));

foreach (var score in highscore)
{
    Console.WriteLine(score.Name);
}
```

And the output, according to our sorting rules, will be:

```
Moni
Hansi
Susi
Susi2
Franzi
```