24. What is a List?

Brief summary: List<T> is a strongly-typed, generic collection of objects. Lists are dynamic, which means we can add or remove the elements from them. It uses an array as the underlying collection type. As it grows, it may copy the existing array of elements to a new, larger array.

"What is a List?" may seem like a trivial question for you, and maybe you even rolled your eyes a little. You've probably used Lists thousands of times. Nevertheless, I think it's worth taking a while to understand how Lists work exactly, and what is going on under their hood.

List<T> is a strongly-typed, generic collection of objects. Lists are dynamic, which means we can add or remove the elements from them.

```
var list = new List<int> { 1,2,3 };
list.Add(4);
```

We can access List's element using an indexer, just like we do with an array:

```
list[0] = 5;
var last = list[list.Count - 1];
```

Lists provide a wide selection of built-in methods, which make them much more convenient than plain arrays. Let me show you some, but certainly not all of them:

```
list.RemoveAt(0);
list.Clear();
list.AddRange(new []{ 5, 6, 7 });
bool contains7 = list.Contains(7);
list.Prepend(10);
list.Sort();
```

Because of the dynamic nature of Lists and their convenience, they are probably the most often used collection type in C#.

All right. Let's take a look under the hood of the List. It turns out that the List is actually a fancy wrapper over an array, and all the List's data is held in a private array. Let me show you a short fragment of the List class source code:

```
[Serializable]
public class List<T> : IL
{
    private const int _de

    private T[] _items;
```

List exposes a property called Capacity. This property says what is the size of this private array held by the List. Let's see it in practice.

```
var newList = new List<int>();
Console.WriteLine(
    $"Newly created list capacity: {newList.Capacity}");

newList.Add(5);
Console.WriteLine(
    $"Capacity after adding 1 item: {newList.Capacity}");
```

Let's see what will be printed to the console:

```
Newly created list capacity: 0
Capacity after adding 1 item: 4
```

Interesting. After a new list is created, its internal array's size is 0. Then, after the element is added, the size is "changed" to 4. I've put the "changed" into quotes because, as we learned in the previous lecture, the size of an array cannot change. I will explain what happens here in a minute, but first, let's add a couple of more elements to exceed the 4 elements limit:

```
newList.AddRange(new [] {6,7,8,9});
Console.WriteLine(
    $"Capacity when 5 items inside: {newList.Capacity}");
```

```
Newly created list capacity: 0
Capacity after adding 1 item: 4
Capacity when 5 items inside: 8
```

It seems the size grew twice. Before explaining, let me just clear the list:

```
newList.Clear();
Console.WriteLine(
    $"Capacity after clearing: {newList.Capacity}");
```

```
Newly created list capacity: 0
Capacity after adding 1 item: 4
Capacity when 5 items inside: 8
Capacity after clearing: 8
```

Even if the List has been emptied, its Capacity remained as it was.

All right. Let's see what is going on.

When we insert the first element to the List, it cautiously assumes that maybe the underlying array does not need to be very large. It creates an array of size 4 and assigns it to the private _items field, which we have seen in the snippet from the source code. This array is used as long as the count of elements in the List remains smaller or equal to 4.

But once this count is exceeded, the List can no longer fit elements in the underlying array - it's simply too small. So what it needs to do is this: first, it **creates a new array**, double the size of the old one. Then, it **copies** all elements from the old array to the new array. Then it can add the new element, that previously didn't fit into the smaller array.

So as you can see, it's not like the "size of the underlying array changes". It does not, as it's not possible to resize an array in C#. A brand-new array is created, and the old array is replaced by it.

On one hand, this is pretty clever, as it allows us to use a fixed-size underlying collection to actually represent dynamic data. On the other hand, it's not great from the performance point of view. Once the resizing is needed, the List must perform this quite complex operation of allocating a new array and copying the old one into it. That's why it's quite "generous" when allocating a new array, and it

makes it double the size of the old one. If the new array would be too small it could soon need to be resized again, and we want to avoid that.

On the other hand, it may of course happen that more memory than needed is allocated. For example, if we exceed the count of 1024 elements, the List will create a new array of size 2048. But it may be the case that in our business case 1025 is the absolute limitation above which the list will never grow.

In this case, we can "help" the list a little, add tell it what count of elements it should be ready for by using the constructor parameter:

```
var listOfCapacity1050 = new List<int>(1050);
Console.WriteLine(
    $"Capacity set to: {listOfCapacity1050.Capacity}");
```

```
Capacity set to: 1050
```

We should definitely consider doing that if we know upfront what is the expected size of the List. Remember that it's not set in stone, and once it's exceeded the List will resize as normal. But we will still avoid all the resizing operations that would normally happen between 0 and 1050 elements.

```
listOfCapacity1050.AddRange(Enumerable.Range(0, 2000));
Console.WriteLine(
    $"Capacity is now: {listOfCapacity1050.Capacity}");
```

```
Capacity set to: 1050
Capacity is now: 2100
```

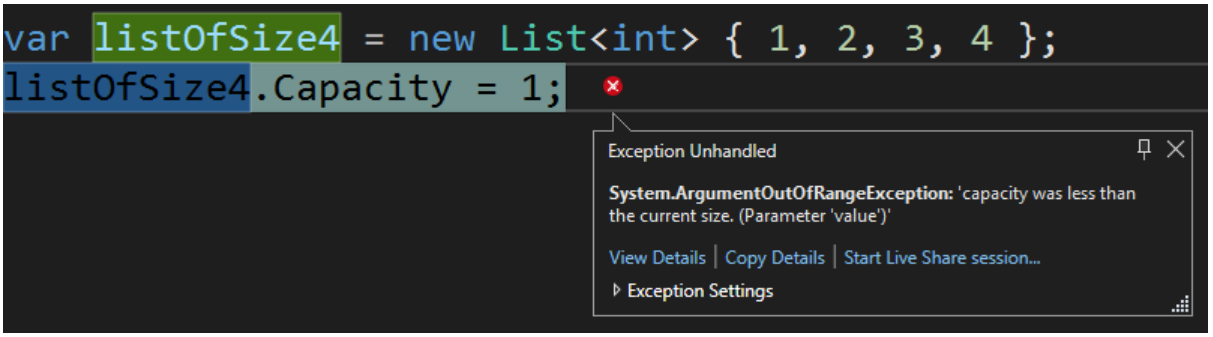
Because the operation of resizing is so heavy, the List doesn't reduce its size once elements are removed. It rather wants to assume that the allocated space will be needed sooner or later since it has already been needed once.

If we have good reasons to think that the larger space needed was a one-time thing, we can either set the Capacity to smaller manually or call the TrimExcess method, which will set the Capacity to the actual count of elements in the List.

```
var listOfCapacity1000= new List<int>(1000);
listOfCapacity1000.Add(5);
listOfCapacity1000.TrimExcess();
Console.WriteLine(
    $"Capacity after TrimExcess: {listOfCapacity1000.Capacity}");
```

```
Capacity after TrimExcess: 1
```

Remember that when setting the Capacity manually we can't make it smaller than the actual count of elements in the List. Otherwise, an exception will be thrown:



All right. We now know how things work under the hood of the List, and that in the end all data is held in an array. We must be aware that this has more implications than only resizing the array once its capacity is exceeded. Let's consider the following line:

```
var someList = new List<int> { 1, 2, 3, 5, 6 };
someList.Insert(3, 4);
```

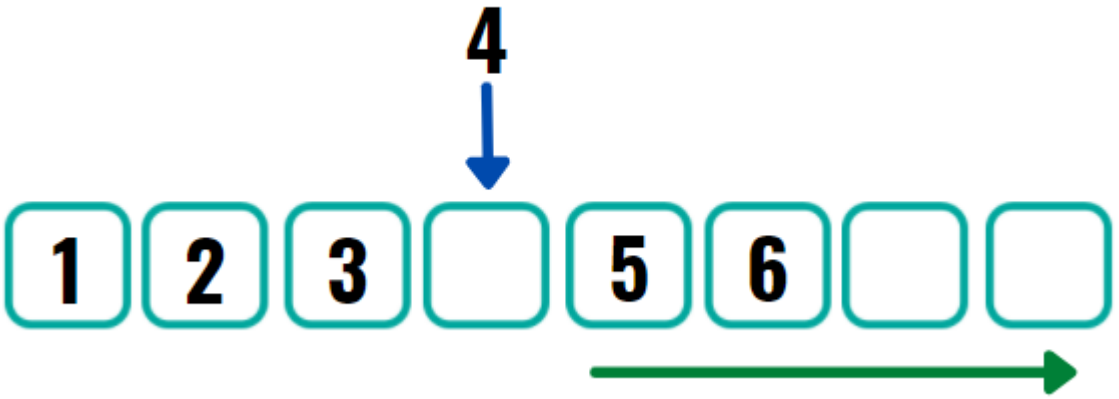
The **Insert** method takes two parameters - the index at which the new value will be placed in the List, and the value to be inserted. In this case, 4 will be inserted between 3 and 5.

Innocent as this operation seems, we must consider what happens with the array that is used underneath. Before the Insert operation, the array looks like this:

Please be aware that the last 3 elements are actually zeros (the default for int type that this Lists stores). The List knows they are not part of the represented data, because it remembers the count of elements it stores, and knows that anything

after the index Count-1 is not the actual data but the spare space that can be occupied by some new values later.

All right, back to the Insert method. We can't simply set the element at index 3 to value 4, because we would overwrite the 5. We must move all elements after the given index one index forward, to make room for the new value:



This is again impacting the performance. Worst case, we may need to move each element in the list (if we insert at index 0). This means the performance of the Insert operation is O(n), which means the count of operations will linearly grow with the collection's size.

The Insert method was just an example to show you that an operation that does something that the underlying array does not support - like in this case, inserting the element in the middle of the data - will always impact the performance, as the data in the array must be rearranged. This is something to be aware of when working with Lists, especially large ones, as the performance impact may be noticeable.

To summarize - Lists are great when it comes to representing collections that are dynamic. They give us a lot of useful methods, making working with them simple and efficient. But we must be aware that adding a feature of dynamic size to the fixed-sized collection like array comes with a cost, and this cost is performance.

Bonus questions:

- "**Why it is a good idea to set the Capacity of the List in the constructor if we know the expected count of elements upfront?**"
  Because this way we will avoid the performance-costly operation of copying the underlying array into a new, larger one, which happens when we exceed the count of 4, 8, 16… elements.

- "**What's the time complexity of the Insert method from the List class?**"
  The Insert method needs to move some of the elements of the underlying array forward, to make room for the new element. In the worst-case scenario, when we insert an element at the beginning of the List, we will need to move all existing elements. This means the complexity of this operation is O(N).