Workshop

Create Custom Data Structures

Overview

In this workshop we will create our own custom data structures – a custom list and a custom stack. The custom list will have similar functionality to C# lists that you've used before. Our custom list will work only with integers for now, but this will be fixed later in the course. It will have the following functionality:

- **`void Add(int element)`** - Adds the given element to the end of the list
- **`int RemoveAt(int index)`** - Removes the element at the given index
- **`bool Contains(int element)`** - Checks if the list contains the given element returns **(True or False)**
- **`void Swap(int firstIndex, int secondIndex)`** - Swaps the elements at the given indexes

Feel free to implement your own functionality or to write the methods by yourself.

The custom stack will also have similar functionality to the C# stack and again, we will make it work only with integers. Later on, we will learn how to implement it in a way that will allow us to work with any types. It will have the following functionality:

- **`void Push(int element)`** – Adds the given element to the stack
- **`int Pop()`** – Removes the last added element
- **`int Peek()`** – Returns the last element in the stack without removing it
- **`void ForEach(Action<int> action)`** – Goes through each of the elements in the stack

Feel free to implement your own functionality or to write the methods by yourself.

1. Implement the CustomList class

Details about the structure

First of all, we must make it clear how our structure should work under the provided public functionality.

- It should hold a **sequence of items in an array**.
- The structure should have **capacity** that **grows twice** when it is filled, **always starting at 2**.

The **`CustomList`** class should have the properties listed below:

- **`int[] items`** **-** An array which will hold all of our elements
- **`int Count`** – This property will return the count of items in the **collection**
- **`indexer`** **–** The Indexer will provide us with functionality to access the elements using **integer indexes**

The structure will have internal methods to make the managing of the internal collection easier.

- **`Resize`** – this method will be used to increase the internal collection's length twice
- **`Shrink`** – this method will help us to decrease the internal collection's length twice
- **`Shift`** – this method will help us to rearrange the internal collection's elements after removing one.

Implementation

Create a new public class **`CustomList`** and add a private constant field named **`InitialCapacity`** and set the value to **2.** This field is used to declare the **initial capacity** of the **internal** array. It's always a **good practice** to use **constants** instead of **magic numbers** in your classes. This approach makes the code better for **managing and reading.**

```
public class CustomList
{
    private const int InitialCapacity = 2;
```

Now let's create the initial **collection,** which is a **private array of type `int`.** To initialize the array, we will use the constructor of the class.

```
private int[] items;

public CustomList()
{
    this.items = new int[InitialCapacity];
}
```

Keep in mind that if the **internal array** has length of **4,** this doesn't mean that our collection holds 4 elements. So we need a property, which will keep the information of the actual **count** of the **elements** in the structure. This property should be updated **every** single time when we **make changes** related to the **count** of the elements like **adding** or **removing**.

```
public int Count { get; private set; }
```

It would be awesome if we can access the items in the collection without exposing the internal array. So, let's implement this functionality. It can be done in the following way:

```
public int this[int index]
{
    get
    {
        if (index >= this.Count)
        {
            throw new ArgumentOutOfRangeException();
        }
        return items[index];
    }
    set
    {
        if (index >= this.Count)
        {
            throw new ArgumentOutOfRangeException();
        }
        items[index] = value;
    }
}
```

When somebody tries to access our collection using index, the **get** accessor is **invoked** and the indexer will return the **value** on the **given index.** When someone is trying to **set** a **value to given** index the **set** accessor is **invoked**.

In both accessors we must check if the index is **less than the Count and greater or equal to zero**, because our structure **actual items count** might be **different** from the **internal array length.**

The whole class should look like this:

```
public class CustomList
{
    private const int InitialCapacity = 2;

    private int[] items;

    public CustomList()
    {
        this.items = new int[InitialCapacity];
    }

    public int Count { get; private set; }

    public int this[int index]
    {
        get
        {
            if (index >= this.Count)
            {
                throw new ArgumentOutOfRangeException();
            }
            return items[index];
        }
        set
        {
            if (index >= this.Count)
            {
                throw new ArgumentOutOfRangeException();
            }
            items[index] = value;
        }
    }
```

Implement void Add(int element) Method

It is time to create the method, which **adds** a new elements to the **end** of our collection, just like in the C# lists. It looks like an easy task, but keep in mind that if our internal array is **filled**, we have to **increase it by twice** the length it currently has and **add** the **new element**.

To make our job easier let's create a **`Resize()`** method first. The method should be used only **within** the **class** so it must be **private**.

```
private void Resize()...
```

Here is how the method should work step by step.

- Create a new array with length **twice** the current length of the internal array:

```
int[] copy = new int[this.items.Length * 2];
```

- Iterate through the items in the **internal array** and fill the **newly created array**:

```
for (int i = 0; i < this.items.Length; i++)
{
    copy[i] = this.items[i];
}
```

- Set the newly created array to the field "**items**":

```
this.items = copy;
```

The whole method should look like this:

```
private void Resize()
{
    int[] copy = new int[this.items.Length * 2];

    for (int i = 0; i < this.items.Length; i++)
    {
        copy[i] = this.items[i];
    }

    this.items = copy;

}
```

Now, we are ready to start implementing the logic behind the **Add()** method. This is how the method should work:

- Check if the **count** of the **actual items** in our **CustomList** is equal to the **Length of our collection**. If it is, this means that the internal array is **filled** and we need to use the **Resize()** method.

```
if (this.Count==this.items.Length)
{
    this.Resize();
}
```

- After we have checked that we have empty space in the internal array, we can just **add** the **new** item at the end and update the "**Count**" property:

```
this.items[this.Count] = item;
this.Count++;
```

The whole method should look like this:

```
public void Add(int item)
{
    if (this.Count==this.items.Length)
    {
        this.Resize();
    }

    this.items[this.Count] = item;
    this.Count++;
}
```

Before proceeding with the next tasks, it is a good practice, since you've done so much work, to test if everything is fine. Use the **debugger** to **test** for bugs.

Implement int RemoveAt(int index) Method

`RemoveAt()` method has the functionality to **remove an item** on the **given index** and **return the item**. Let's think about how to solve this problem by **dividing it to smaller tasks**.

- First we must check if the index is **valid** and if not, we should throw **`ArgumentOutOfRangeException`.**
- Get the item on the given index and assign it to a variable, which will be **returned** at the end.
- Set the value on the given index to the **default value of `int`.**
- Now we have an empty element and we need to **shift** the elements.
- Reduce the **count** and check if the count is **4 times smaller** than the **internal array length**. If it is we have to think about a way to **shrink** the array.
- In the end, return the variable to which we assigned the value of the requested index.

So now you already know that we need to implement the other 2 internal methods **`Shift()`** and **`Shrink()`**.

void Shift(int index)

The shift method uses a loop, which moves all the elements to the left starting from a given index.

```
private void Shift(int index)
{
    for (int i = index; i < this.Count - 1; i++)
    {
        this.items[i] = this.items[i + 1];
    }
}
```

void Shrink()

The **`Shrink()`** method is exactly the same as the **`Resize()`** method with the small difference that it will **reduce** the length twice, instead of increasing it.

```
private void Shrink()
{
    int[] copy = new int[this.items.Length / 2];
    for (int i = 0; i < this.Count; i++)
    {
        copy[i] = this.items[i];
    }
    this.items = copy;
}
```

Now we are ready to proceed with the implementation of the **`RemoveAt()`** method.

```
public int RemoveAt(int index)
```

- First, we need to validate that our index is valid. Keep in mind that the validation of the index should be verified using the **Count property.**

  ```
  if (index>=this.Count)
  {
      throw new ArgumentOutOfRangeException();
  }
  ```

- Get the value on the given index, assign it to a variable and set the value in the array on the specified index to default. Also don't forget to shift the items.

  ```
  var item = this.items[index];
  this.items[index] = default(int);
  this.Shift(index);
  ```

- Now we must reduce the Count and check if **shrinking** the array is **required**.

  ```
  this.Count--;
  if (this.Count<=this.items.Length/4)
  {
      this.Shrink();
  }
  ```

- After all, just return the element that we saved at the start.

The whole method should look like this:

```
public int RemoveAt(int index)
{
    if (index>=this.Count)
    {
        throw new ArgumentOutOfRangeException();
    }

    var item = this.items[index];
    this.items[index] = default(int);
    this.Shift(index);

    this.Count--;
    if (this.Count<=this.items.Length/4)
    {
        this.Shrink();
    }

    return item;
}
```

It is time to test out your **`CustomList`** again. If everything works fine, proceed with the next task.

Implement void Insert(int index, int item) Method

You are already familiar with this method, so let us go straight to the implementation. First of all, we will **split** the logic on **smaller tasks**:

- We have an index parameter, so we must **validate the index.**
- We must check if the array should be **resized**.
- We have to **rearrange** the items to **free the space for the required index**.
- Finally **insert** the given item on the index and **increase** the **count**.

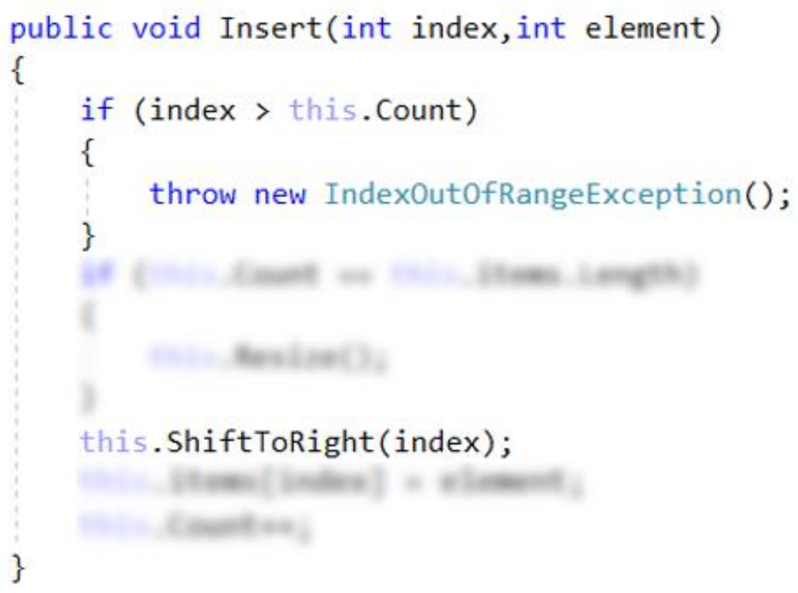
You probably have already noticed, that since we have a method to rearrange the items to the left, used to fill up the empty space when we remove an item, we must have a method to rearrange items to the right, so let's create it.

Starting from the **end of the actual elements,** this method will **copy** every single item on the **next index**. The loop will **end on the requested index**.

```
private void ShiftToRight(int index)
{
    for (int i = Count; i > index; i--)
    {
        this.items[i] = this.items[i - 1];
    }
}
```

You'll be given the **implemented method** with blurred parts, so you have to do it on your own. Follow the description above.

Hint: Inserting an item on the index equal to the **Count** (**inserting item at the end of the collection**) should be a valid operation and it should do the same as the **Add() method**.



```
public void Insert(int index,int element)
{
    if (index > this.Count)
    {
        throw new IndexOutOfRangeException();
    }
    if (this.Count == this.Items.Length)
    {
        this.Resize();
    }
    this.ShiftToRight(index);
    this.Items[index] = element;
    this.Count++;
}
```

Implement bool Contains(int element) Method

This method should check if the given element is in the collection. Return true if it is and false if it's not. It's a simple task, so you should do it all on your own.

Hint: When you are iterating through the items, use the **"Count"** property as an end condition, instead of the internal array length.

Implement void Swap(int firstIndex, int secondIndex) Method

Just like the method above, we consider this an easy task for you. Of course, you have a hint. When we work with indexes, we must always check if they are less than the count, because you may end up in the situation, where the collection has 3 actual elements, while the internal array has a length of 4.

If you have good ideas to implement new functionalities, like `Find()`, `Reverse()` or overriding `ToString` methods, feel free to do it.

2. Implement the CustomStack class

Details about the structure

The implementation of a custom stack is much easier, mostly because you can only execute actions over the last index of the collection, plus you can iterate through the collection. You should be able to create it entirely on your own. The first thing you can do is have a clear vision of how you want your structure to work under the provided public functionality. For example:

- It should hold a **sequence of items in an array**.
- The structure should have a **capacity** that **grows twice** when it is filled, **always starting at 4**.

The **`CustomStack`** class should have the properties listed below:

- **`int[] items`** - An array, which will hold all of our elements.
- **`int Count`** – This property will return the count of items in the **collection.**

- **`const int InitialCapacity` –** this constant's value will be the initial capacity of the internal array.

Implementation

Create a new public class **`CustomStack`** and add a private constant field named **`InitialCapacity`** and set the value to **4.** This field is used to declare the **initial capacity** of the **internal** array. We already know that it's not a good practice to have **magic numbers** in your code. Afterwards, we are going to declare our internal array and a field for the count of elements in our collection.

```
public class CustomStack
{
    private const int initialCapacity = 4;
    private int[] items;
    private int count;
```

Of course, you have to initialize the collection. Also, set the count variable to 0. As we already know, this can be done inside the constructor of the class:

```
public CustomStack()
{
    this.count = 0;
    this.items = new int[initialCapacity];
}
```

Next, you have to add a public property **`Count`** that holds the value of the **`count`** field. This way, you will be able to get the count of items in the collection from other classes.

```
public int Count
{
    get
    {
        return this.count;
    }
}
```

Now you can proceed to the implementation of the methods, which your **`CustomStack`** is going to have. All of the functionalites described in the description are very easy to implement, so we strongly recommend for you to try to do it on your own. If you have any difficulties, you can help yourself with the code snippets below.

Implement void Push(int element) Method

This method adds an element to the end of the collection, just like the C# Stack **`Push()`** method does . This is a very easy task. Here is the code you can use, if you meet any difficutlies:

```
public void Push(int element)
{
    if(this.items.Length == this.count)
    {
        var nextItems = new int [this.items.Length * 2];
        for (int i = 0; i < this.items.Length; i++)
        {
            nextItems[i] = this.items[i];
        }
        this.items = nextItems;
    }
    this.items[this.count] = element;
    count++;
}
```

Implement int Pop() Method

The **`Pop()`** method returns the last element form the collection and removes it. The implementation is easier than the implementation of the **`RemoveAt(int index)`** and **`Remove()`** methods of the **`CustomList`**. Try to implement it on your own. Afterwards, you can look at this implementation:

```
public int Pop()
{
    if(this.items.Length == 0)
    {
        throw new InvalidOperationException("CustomStack is empty");
    }
    var lastIndex = this.count - 1;
    int last = this.items[lastIndex];
    this.count--;
    return last;
}
```

Implement int Peek() Method

The **`Peek()`** method has the same functionality as the C# Stack – it returns the last element from the collection, but it **doesn't remove** it. The only thing we need to consider is that you can't get an element from an empty collection, so you must make sure you have the proper **validation**. For sure, you will be able to implement it on your own.

Implement void ForEach(Action<object> action) Method

This method goes through every element from the collection and executes the given action. The implementation is very easy, but it requires some additional knowledge, so here is what you can do:

```
public void ForEach(Action<object> action)
{
    for (int i = 0; i < this.count; i++)
    {
        action(this.items[i]);
    }
}
```

You can add any kind of functionalities to your **`CustomStack`** and afterwards you can test how it works in your **`Main()`** method in your **`StartUp`** class.