# Workshop Create Custom List

## Overview

In this workshop we will create our own custom list. 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.

The custom list 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.

## 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:

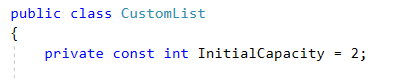
* **Int[] items -** An array which will hold all of our elements
* **Int Count** – This property will return the count of items in the **collection**
* **Indexer –** The Indexer will provide us with functionality to access the elements using **integer indexes**

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

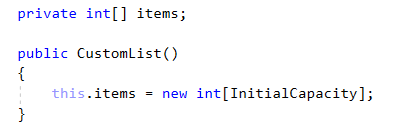
* **Resize** – this method will be used to increase the internal collection's length twice
* **Shrink** – this method will help us to decrease the internal collection's length twice
* **Shift** – this method will help us to rearrange the internal collection's elements after removing one.

### Implementation

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

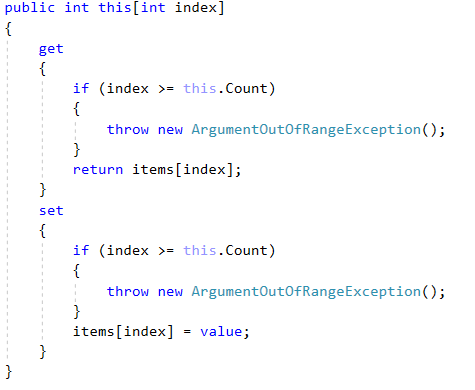


Now let's create the initial **collection** which is **private array of type int.**  
To initialize the internal array we will use the constructor of the class.



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.

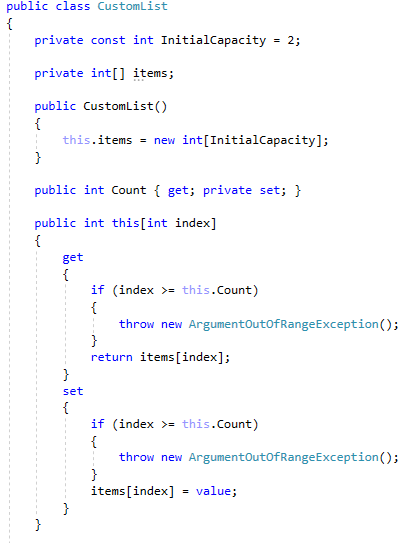


It would be awesome if we can access the items in the collection without exposing the internal array. So, let's implement this functionality it might be new to you so just add the following code to your class.

When someone is trying to access our collection using index the

When someone is trying 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 clas s should look like this.



## Implement void Add(int element) Method

It is time to create the method responsible for **adding** 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 twice** the length it currently has and add the new element.

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

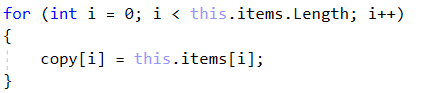


How the method should work step by step.

* Create new array with length twice the internal array.



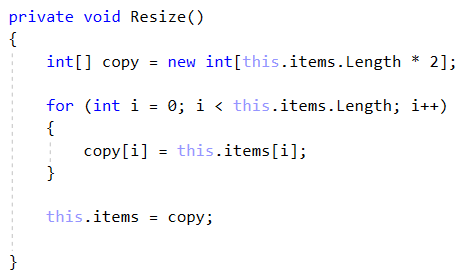
* Iterate through the items in the **internal array** and fill the **newly created array**.



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

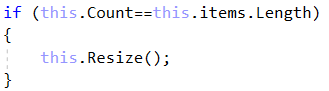


The whole method should look like this.



Now we are ready to start implementing the logic behind **Add** method.  
How the method should work is pretty simple:

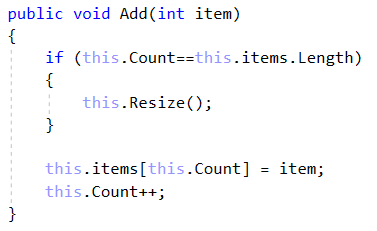
* Check if the count of the **actual items** in our **CustomList** is equal to the **Length of our collection**.  
  If it is this means that the internal array is filled up and we need to use the **Resize** method.



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



The whole method should look like this.



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

## Implement int RemoveAt(int index) Method

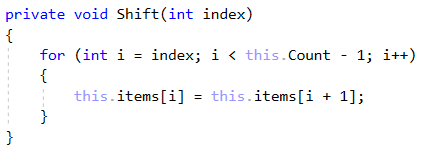
**RemoveAt** method have the functionality to **remove item** on the **given index** and **return the item**. Let's think about how to solve this problem **dividing it to smaller tasks**.

* First we must check if the index is **valid**, if it's not 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 **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.
* After all 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**.

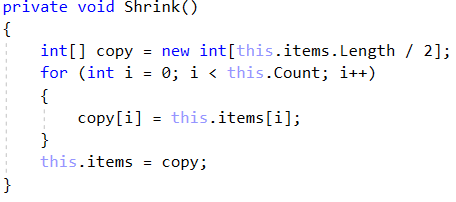
### void Shift(int index)

The shift method is a simple for cycle which moves all the elements to the left starting from a given index



### void Shrink()

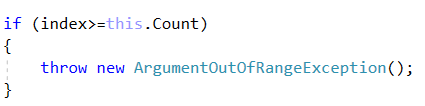
The **shrink** method is exactly the same as the **Resize** method with the small difference that we will **reduce** the length twice instead of increasing it.



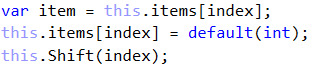
Now we are ready to proceed with the **RemoveAt** method implementation.



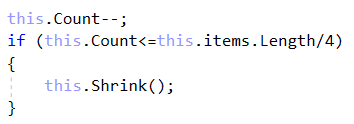
* First, we need to validate that our index is valid. Keep in mind the index validation should be verified using the **Count property.**



* 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.

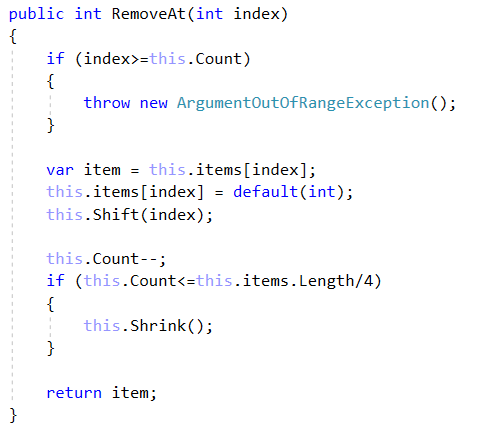


* Now we must reduce the Count and check if **shrinking** the array is **required**.



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

The whole method should look like this



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

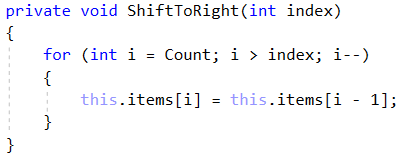
You are already familiar with this method so let us head straight to the implementation.

First of all we will **split** the logic on **smaller** **tasks** so it would be easier for us to know what we need to do.

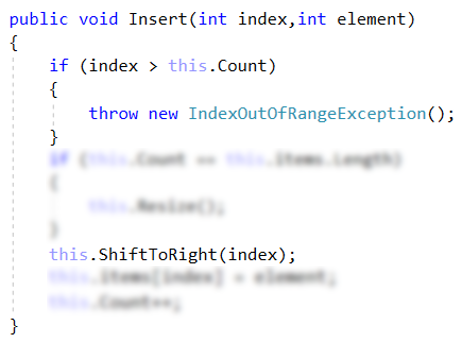
* We have index parameter, so we must **validate the index**
* We must check if array **Resize** is **required**
* We have to **rearrange** the items to **free the space on the required index**.
* Finally **insert** the given item on the index and **increase** the **Count**.

You probably already noticed that since we have a method to rearrange the items to left, used to fill up the empty space when we remove item, we must have method to rearrange items to 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 cycle will **end on the requested index as parameter**.



You'll be given the **implemented method** with blurred parts, so you have to do it on your own. Follow the description above and you won't face a problem.  
**Hint**: Inserting an item on the index equal to the **Count** (**inserting item at the end of the collection**) should be valid operation, and it should do the same as the **Add Method**.



## 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.

Tip: Iterating through the items use the "Count" property as end condition, instead of the internal array length.

## Implement void Swap (int firstIndex, int secondIndex) Method

Just like the method above we consider this as easy task for you, so you are all alone again.   
Of course, you have a tip.

When we work with indexes we always must check if they are less than 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.

This is all we must do for now to our custom list. If you have good ideas to implement new functionality, like Find, Reverse or overriding ToString methods, feel free to do it.