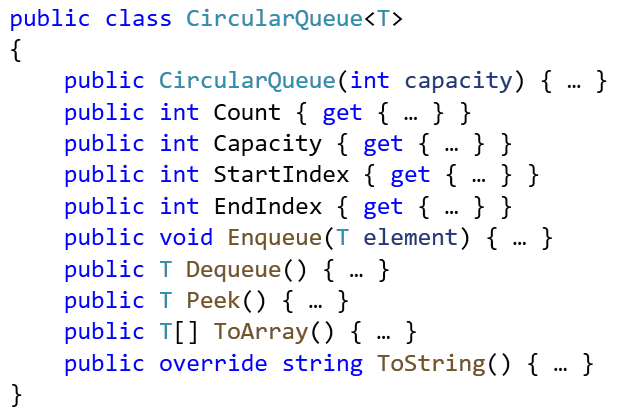
# Exercises: Unit Testing

Problems for exercises and homework for the ["C# OOP" course @ SoftUni"](https://softuni.bg/trainings/4377/csharp-oop-february-2024).

## Unit Testing the CircularQueue<T> Class

You are given a C# class CircularQueue<T>, which implements the data structure **circular queue** (learn more at <https://en.wikipedia.org/wiki/Circular_buffer>):



The circular queue internally holds its elements in **array**, where the **start** and the **end** of the queue are **array indexes**. In the beginning, the **end index** goes after the **start index**:



After several enqueue / deque operations are executed, the queue elements move right and can **cross the array border** and continue from its leftmost element. In this case the **end index** goes before the **start index**:



The **border crossing** in the circular queue implementation is a little bit tricky and should be handled with care.

### Write Unit Tests

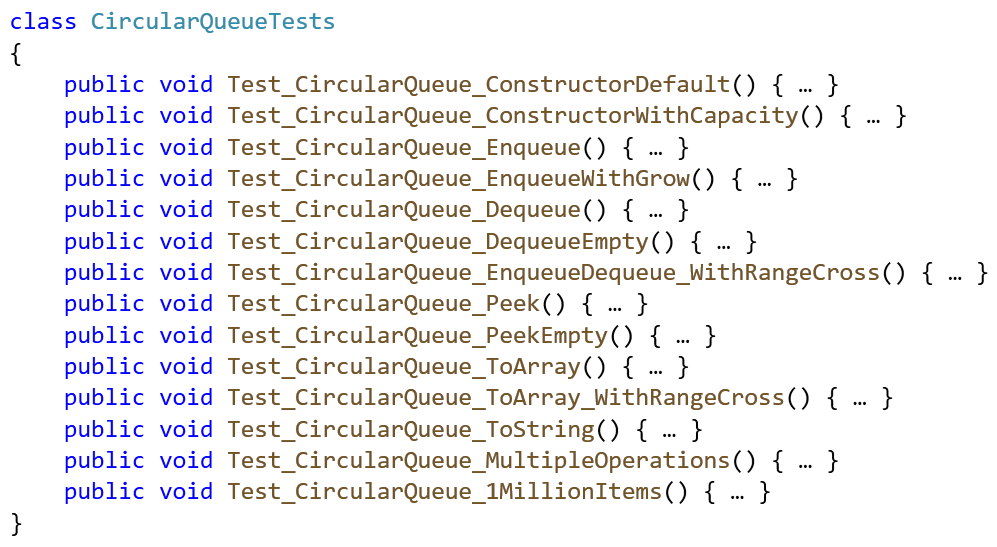
Your task is to write **unit tests** for the class CircularQueue<T>. Ensure that the **code coverage** is high and that all interesting cases are covered: test **all public methods**, test the **queue growing**, test the **range crossing** (when the end index comes before the start index) and any other special cases. Test the **private methods** indirectly, but especially designed invocations of the public methods.

### Hints

Create a Visual Studio solution, holding two projects:

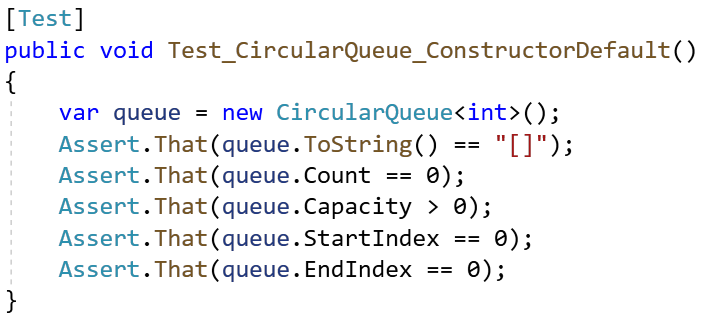
* CircularQueue – it will hold the CircularQueue<T> class, which should be unit tested
* CircularQueue.Tests – it will hold the **test classes**, which cover the circular queue functionality

You may implement the following **test cases**:



### Testing the Constructor

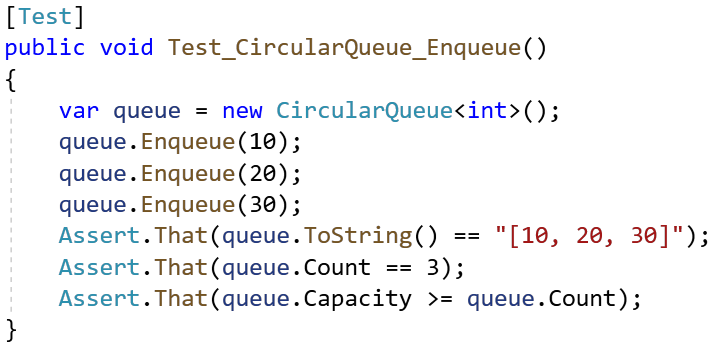
As start, we can **test the default constructor** of the CircularQueue<T> class:



Testing the **constructor with fixed initial capacity** is very similar.

### Testing Enqueue / Dequeue / Peek

Next, we can **test the enqueue method**: create a queue, enqueue few elements and check whether the elements in the queue are as expected:



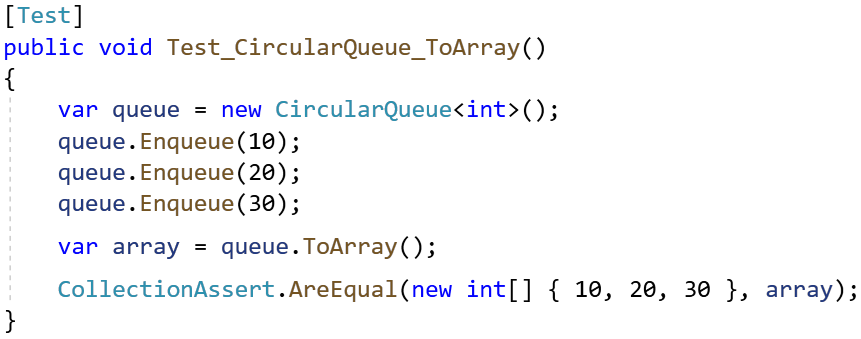
Test the **enqueue with auto-grow**: enqueue enough elements for overflow the capacity. This will cause auto-grow.

Test the **dequeue** operation: with **non-empty** and with **empty queue**.

Test the **peek** operation: with **non-empty** and with **empty queue**.

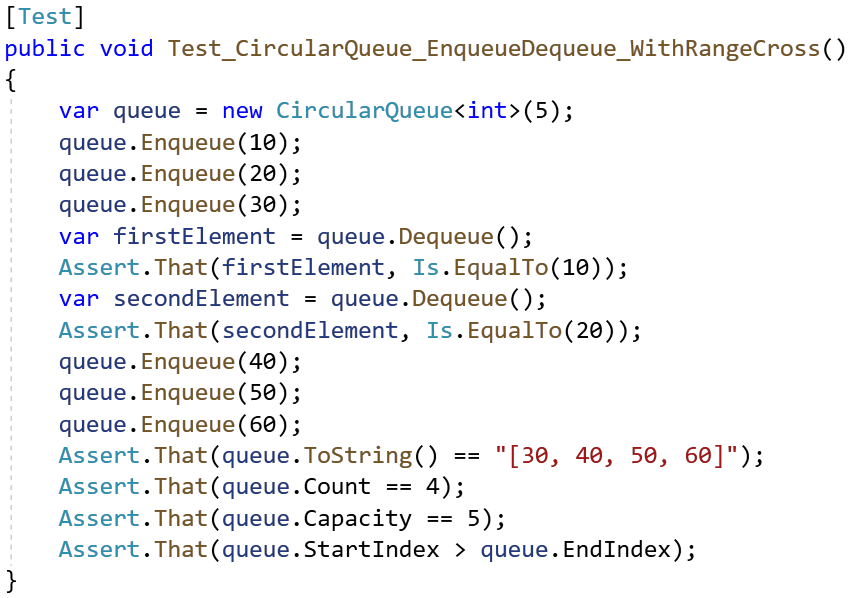
### Testing ToArray() and ToString()

Test also the ToArray() and ToString() operations, which are quite similar. Just enqueue some data, invoke ToArray() / ToString() and assert the result is as expected:



### Testing the Bounds Crossing

In the circular queue the most interesting moment is when **the end index crosses the bounds** of the internal buffer and comes behind the start index. This situation should be carefully tested. This is an example of test, which causes **range crossing** of the internals buffer bounds:



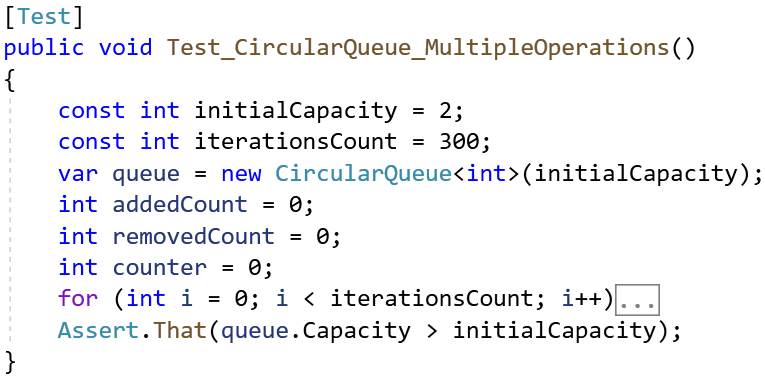
### Combined Test

To test the CircularQueue<T> class under heavy load with multiple enqueue / dequeue / peek / to string operations and multiple asserts, we can design a special test, which will repeat the following **300 times**:

* Enqueue 2 new elements
* Peek the first element
* Dequeue the first element
* Check the count of elements
* Check the elements, using ToArray() / ToString()

The above will ensure that the underlying circular buffer will auto-grow and overflow the borders multiple times. Such test will ensure with high confidence that the class behaves correctly.

This is an example how we can implement such a **complex test with multiple operations and assertions**:

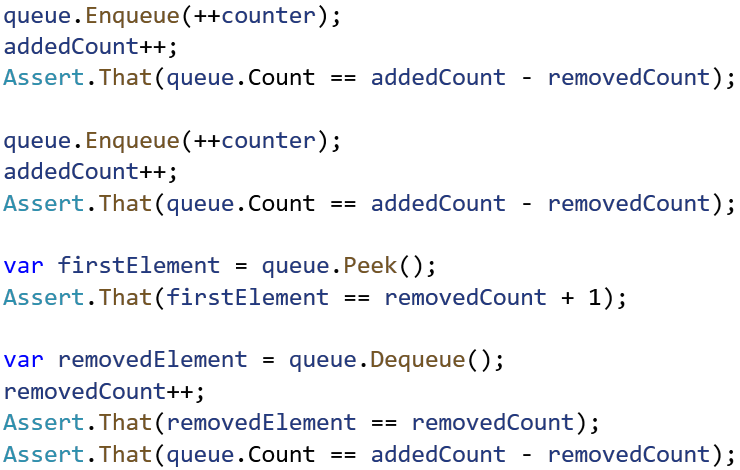


These are the **operations** and **assertions** we can perform at each iteration:

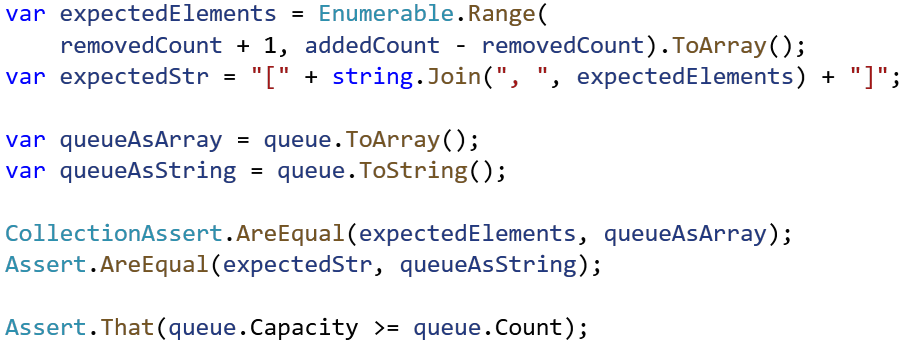
* Check the **queue size**:



* Enqueue **2 new elements**, **peek** and **remove 1 element**. This will guarantee that the circular buffer will move to the right and will cross the bounds of its underlying array at some moment:

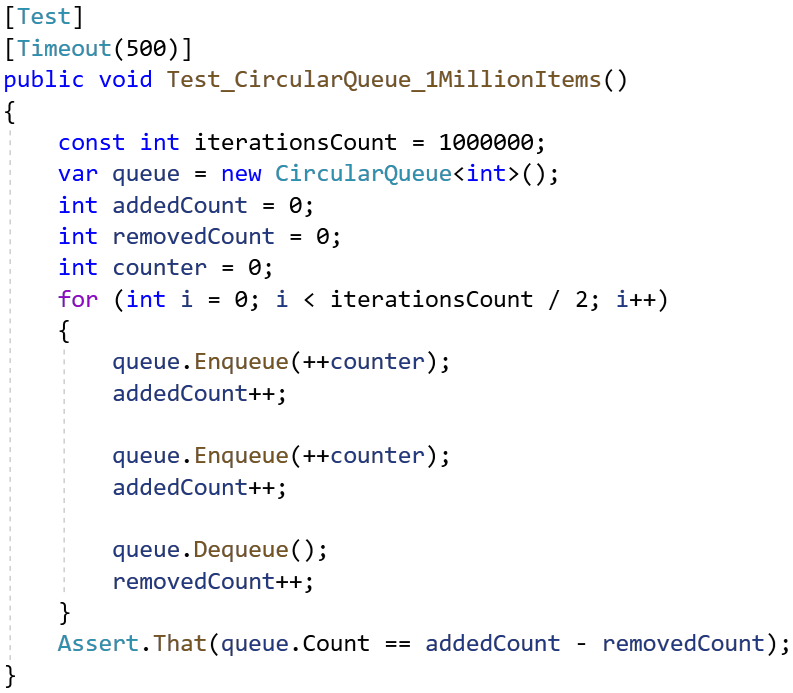


* Check the queue content after each iteration:

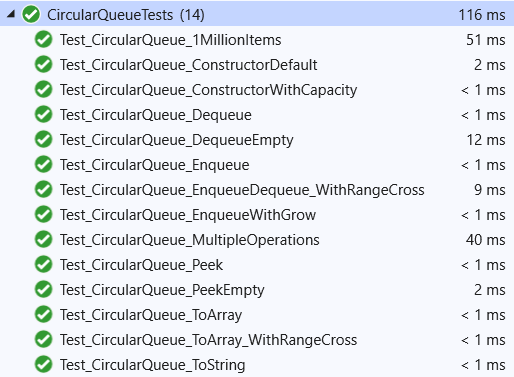


### Performance Test

Finally, we could implement a simple performance test, which **adds 1 million items** in the queue and **removes 0.5 million items** from it. It could look like this:

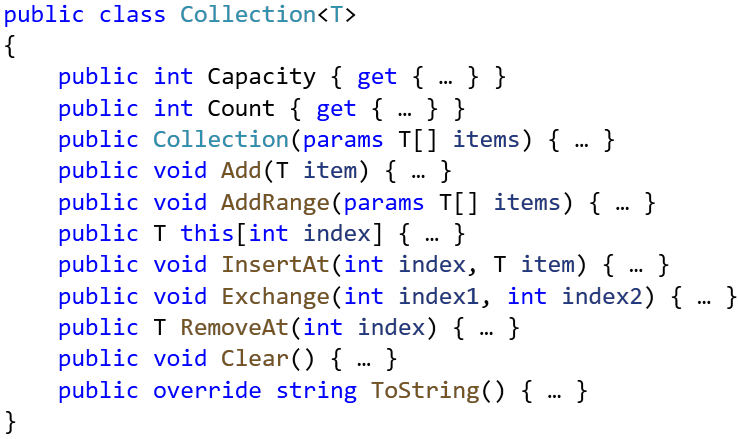


This is an example how your unit tests may look like after successful execution:



## Unit Testing the Collection<T> Class

You are given a C# class Collection<T>, which implements a generic collection, holding an indexed sequence of elements:



Write **unit tests** for the class Collection<T>. Ensure that the **code coverage** is high and that all interesting cases are covered: test **all public methods**, test the **auto growing** of the underlying array, test with valid and invalid ranges, and try to cover all other **special cases**.

### Hints

* Test **all public methods**.
* Think about all **different scenarios**, which can happen, e. g. insert at the start, insert at the end, insert at the middle, insert with auto-grow, insert at invalid position, insert into empty collection, etc.
* Ensure you test each method with **valid and invalid data**.
* Implement a **performance test** with 1 million items, with **timeout**.

You may implement the following unit tests:

