# Implementing a Queue, starting from a SmartArray

**The goal for this exercise** is to both implement a circular Queue, as well as examine one use of object-oriented programming: using inheritance to specialize an existing class.

For this exercise, you need to create a subclass of the SmartArray class, named QueueOfInts. Your QueueOfInts class will need to support the Enqueue, Dequeue, Peek, IsEmpty methods. In order to do that, you will need to add a method to the SmartArray class – getSize.

All of the methods for both the SmartArray, and QueueOfInts classes, are described in more detail at the end of this document.

For this exercise, you are welcome to (and encouraged to) simply reuse your code for the SmartArray class from a prior exercise.

Here’s a quick, visual review of what a circular queue is. Let’s say that your queue is implemented using an array of, say, 3 elements. When the queue is created, there are zero items in the queue, and frontOfQueue and backOfQueue are both zero:



Next, let’s say you add the number 12 to the queue. Because there’s space available, you add 12 at the index specified by backOfQueue (i.e., slot 0) . Once you’re done, the queue now looks like:



After adding -99 and 7 to the queue, the queue will look like:



Note that backOfQueue has ‘wrapped around’ to be zero again, which is the essence of a **circular** queue. At this point the queue is full, so we can’t add anything more to it; let’s remove the first item in the queue (i.e., the item located at frontOfQueue). We’ll get this:



Count goes down to 2, as we expect, and frontOfQueue has been incremented in preparation for the next time Dequeue is called. What’s interesting is that while the value “12” has been returned to the caller of the Dequeue method, and while it is considered to have been removed from the queue, we leave the value 12 in slot 0. Why? Because frontOfQueue now holds the value 1, so we will not return the contents of slot 0, AND because the next value that gets added (at backOfQueue) will overwrite “12”. Thus, there’s no reason to waste time replacing it with another value.

Let’s say that we add the value 111. Since there’s now space in the queue, we’ll add 111 at the location of backOfQueue, leaving us with:



**What you need to do for this exercise:**

1. For this exercise, you need to implement the QueueOfInts class. A more detailed description of this class is included at the end of this document. This class should be found in the **Student\_Answers.cs** file in a project named something like **03\_PCE\_StudentCode**.

	1. As part of this, you will need to add the getSize method to the SmartArray class.
2. Once you’ve completed the above task, you should uncomment any commented-out tests, and then run the tests in the Test\_Queue\_SmartArray\_Basic class and the Test\_QueueOfInts class. They should all pass at this point, and if not, then you should fix your program so that those tests do pass. You can run the tests using the NUnit support in the provided starter project. You should feel free to supplement that code with your own test cases if you wish, but you are neither required nor expected to.

	1. The Test\_Queue\_SmartArray\_Basic class serves to make sure that your QueueOfInts still does everything that a basic SmartArray does. It is expected that your QueueOfInts will be able to pass these tests simply by virtue of inheriting methods from the SmartArray base class.

		1. Note that this means that your QueueOfInts class MUST inherit from the SmartArray class in order for these tests to even compile!
	2. The Test\_QueueOfInts class serves to check new, Stack-specific functionality that you’ve implemented in this exercise.

|  |
| --- |
| **SmartArray Data & Methods** |
| Data Field Name | Type | Description: |
| rgNums | Array of ints | A reference to an array of integers. |
| Note: all data fields should be marked **private** |
| Unchanged Methods: |
| These are unchanged from the prior implementation:Default constructor,GetAtIndex,PrintAllElements,Find,SetAtIndex |
| New Method Name | Returns | Description/Parameters: |
| getSize | An integer – the size, in number of elements, that the SmartArray currently has the capacity to hold. | public int getSize(){ return rgNums.Length;}☺ |
| Note: all methods should be marked **public** |

|  |
| --- |
| **QueueOfInts Data & Methods** |
| Data Field Name | Type | Description: |
| count | int | How many items are currently stored in the queue. Note that this number can (and often will) be different from the size of the array – it's entirely possible to have a queue that's using an underlying array of size 5, but that queue only has 2 (or 0, or 1, or 3, or 4) items stored in the queue. |
| frontOfQueue | int | **frontOfQueue** will be the index within the array of the FRONT (first) element in the abstract queue. If **count** isn't zero (i.e., if there's something in the queu), then this variable will hold the index of the slot in the array that contains the next item to be returned by the Dequeue method. |
| backOfQueue | int | **backOfQueue** will be the index of the NEXT space that will be used to hold an item in the queue. Therefore, this should start out at zero, meaning that the first item Enqueued will end up at slot zero. |
| Note: all data fields should be marked **private** |
| Method Name | Returns | Description/Parameters: |
| <constructor> | Nothing, by definition | Allocates the array that the QueueOfInts will use to store the integers |
| isEmpty | True, if the queue currently contains NO elements.False otherwise | Return type says it all |
| Enqueue | Returns nothing* Will throw an OverflowException if the stack runs out of space in the underlying array
 | Parameters:1. An integer that is the value to be added to the front of the queue

This method will take the value given by the parameter, and add it to the queue. Note that you are required to implement a circular queue. You are required to implement a circular queue, even if your SmartArray class automatically handles allocating extra space. *Hint:* In order for this to happen, you'll need to adjust **backOfQueue**, etc –one of the objectives of this exercise is for you to figure out how. |
| Peek | Returns the top-most item on the stack.Will throw an UnderflowException if the stack is empty, and therefore there is nothing to Peek at. | Parameters:1. None

Note that this method, unlike Pop, does NOT change the queue in any way – it only copies the front-most item into the parameter (if there is a front-most item to copy), and then returns.*Example:* Pushing the values 3, 5, 2 on to the queue, then doing a Peek, will result in the value 3 (the front item) being returned.*Hint:* Unlike Push or Pop, you will NOT need to change anything in object. |
| Dequeue | Returns the top-most item on the stack.Will throw an UnderflowException if the stack is empty, and therefore there is nothing to Pop at. | Parameters:1. None

Note that this method, unlike Peek, DOES change the queue – it copies the front-most item into the **out** parameter (if there is a front-most item to copy), and then removes that front-most item from the queue.*Hint:* In order for this to happen, you'll need to adjust **frontOfQueue**, etc – one of the objectives of this exercise is for you to figure out how. |
| Note: all methods should be marked **public** |