

Introduction

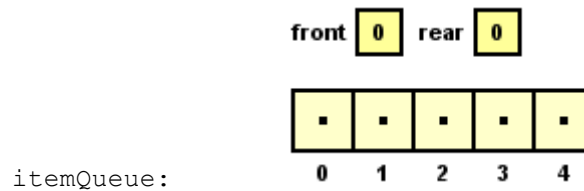
The queue is similar to the stack in that the user can only access one element from the structure at a time. The queue, however, exhibits **FIFO** (first in, first out) behavior, meaning that the remove operation (**dequeue**) will remove the element that was added before the other elements in the queue. Elements are added to the **rear** of the queue, while elements at the **front** of the queue are the first to be removed.

As an example, consider a line of customers at a store. The line of customers is the queue, and the cashier is the user of the queue. The first customer to get in line is the only customer that the cashier can check out. Other customers can get in the **rear** of the line, but the cashier can only see the next customer when the customer in the front of the line leaves.

One method of implementing a circular queue would be to use a linked chain of nodes. Though a linked implementation would not require dynamic expansion (as with an array-based implementation), it adds the overhead of having a node to hold the 'next' reference as well as a reference to the element. This lab will use an array to implement a circular queue rather than a linked implementation.

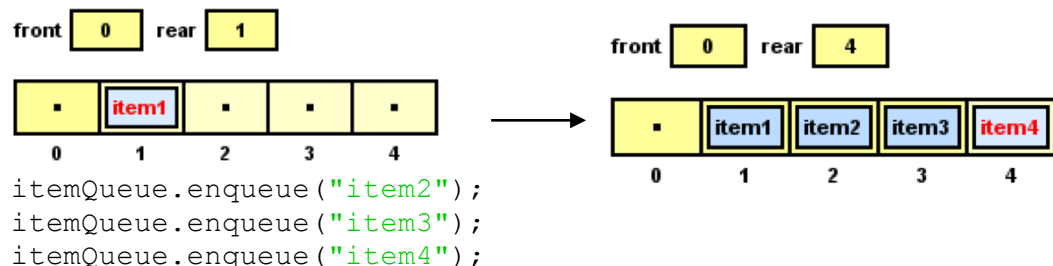
Queue Operations using an Array Implementation

In order to differentiate between a full and empty queue, one index of the array will always remain unused. The index used to track this location in the array will be called **front**; however, the actual front of the queue will be the next element. The location of the **rear** element will also be tracked with an index. The following queue is an empty queue with a capacity of 4. Note that there are 5 locations in the array, as one always remains unused.



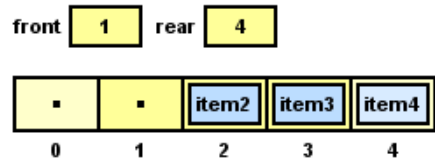
- **enqueue and dequeue:** In order to add an element to the rear of the queue (**enqueue**), the rear index must be increased by one and the element must be added to that location:

```
itemQueue.enqueue("item1");
```

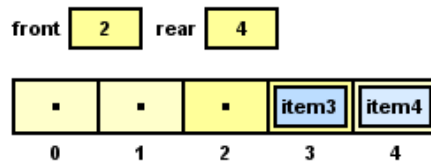


Similarly, when an item is removed from the queue (**dequeue**), the front index is moved forward and the element at the new front index is returned. The previous front item is set to null.

```
itemQueue.dequeue(); // returns "item1"
```

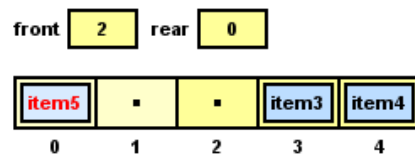


```
itemQueue.dequeue(); // returns "item2"
```

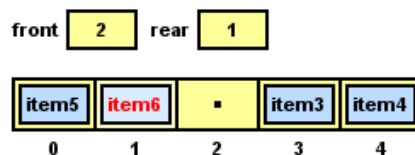


Notice now that the rear index has reached the end of the queue, but that there is still room for at the beginning. The rear index will now return to the beginning of the array to start adding elements:

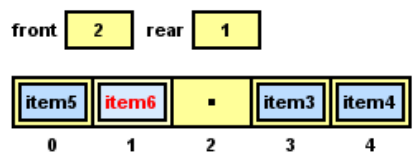
```
itemQueue.enqueue("item5");
```



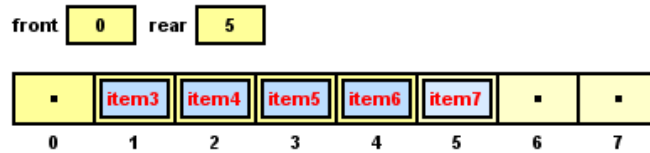
```
itemQueue.enqueue("item6");
```



The queue has now reached its maximum capacity of 4. In order to expand the array, a larger array is created, the elements are dequeued (in order) from the old queue and placed into the new array starting back at index 1. The front and rear indices must also be set to reflect the change:



```
itemQueue.enqueue("item7");
```



Because the front and rear indices must move to the beginning of the array when they reach the end, code is needed that will increment front or rear if they have not reached the end of the array and set them to 0 if the end of the array is reached:

```
rear = (rear + 1) % queue.length;  
front = (front + 1) % queue.length;
```

Explanation: The remainder after dividing a number by length will always be that number if length is greater (example: $(2 + 1) / 6 = 3/6 = 0$ with remainder 3). Therefore, rear and front will be incremented if they are below length - 1. If front or rear holds the value length - 1 (the end of the array), then adding 1 and dividing by length results in a remainder of 0. Therefore, if front or back is at length - 1 then using the code above will set the value back to 0 (the beginning of the array).

A conditional statement could also be used to reset the index to zero if it reaches the end of the array. For example:

```
rear++;  
if (rear == queue.length) {  
    rear = 0;  
}
```

or

```
rear = rear >= queue.length == 0 ? 0 : rear;
```

- **isEmpty:** One can always tell if a circular queue is empty by checking whether the front and rear indices are equal. If the two are equal, then the queue is empty.

Implementation

Create a generic class called `CircularQueue` with the following instance variables:

- `queue`: a reference to an array holding elements in the queue
- `front`: the index directly before the front (first element) of the queue
- `rear`: the index of the last element that was added to the queue
- `DEFAULT_CAPACITY`: the default capacity of the queue

Add a constructor that will initialize the internal array to a specified capacity that's provided as an argument. Recall that one location in the array stays empty, and so the size of the array will need to be 1 element larger than the specified capacity. Initialize `front` and `rear` to 0.

A second constructor takes no parameters and sets the capacity of the data array to the default value.

Add a method that will determine if the queue is empty. Recall that the queue is empty if the `front` and `rear` index the same location in the array.

Add a method that will determine if the queue is full. This method will be used internally to determine if the array needs to be expanded. The queue is full if moving the `rear` forward (enqueue) will result in the `rear` and `front` indexing the same location in the array.

When elements are added (enqueue), they are added to the the rear of the queue and the rear index is moved forward. Add an enqueue method to add an element to the queue:

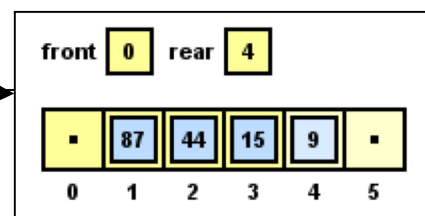
- First, you'll have to expand the internal array if the queue is full. Because there is not yet an expand method, comment the line out for now:
- To add an element to the rear of the queue, move the rear index up and add the element to the rear location in the queue.

Answer the following questions and then use the interactions pane and viewer to verify your answers.

- If the initial capacity is set to 5, what will be the size of the internal array?
- Suppose that 2 elements 87 & 44 are added to the queue. Where will they be placed in the array?
- What will be the value of `front` and `rear` after the above 2 enqueue operations? What about after 3 enqueue operations?

In the interactions pane, add 4 elements to `q1`. Open a viewer on `q1` after the first line is executed.

```
CircularQueue<Integer> q1 = new CircularQueue<>();  
q1.enqueue(87);  
q1.enqueue(44);  
q1.enqueue(15);  
q1.enqueue(9);
```



Create a dequeue method that will remove and return an element from the queue.

- The method should throw an `NoSuchElementException` (you'll need to import the `java.util` package) if the queue is empty.
- If not, move the `front` index forward to the front element and create a reference to the element at that location.

- Delete the reference at the front index (1 before the next front element) and return the previous front element.


Create an expand method to expand the internal array's size.

- Create a new array with increased size increased by a factor of two.
- Dequeue all of the current elements add them to the new array beginning at index 1.
- Set queue to reference the new array and update the front and rear pointers.

Important: Find your enqueue method and make sure to uncomment the expand method invocation before continuing.

Create the following main method with the break point at the specified location. Answer the following questions and then use the viewer to confirm your answers:

- When does the queue first become full (does not need to expand, but has the max number of elements)?
- During which line of code will the array exceed the capacity and need to expand?
- What is the placement of elements in the old array and the new array once the queue has expanded? HINT: you'll have to step into the expand method and look at both the current queue and newQueue.
- What will be printed to standard output once the program is complete?



```
public static void main(String[] args) {  
    CircularQueue<String> q = new CircularQueue<String>(3);  
    q.enqueue("a");  
    q.enqueue("b");  
    q.enqueue("c");  
    System.out.println(q.dequeue());  
    q.enqueue("d");  
    q.enqueue("e");  
    System.out.println(q.dequeue());  
    System.out.println(q.dequeue());  
    System.out.println(q.dequeue());  
    System.out.println(q.dequeue());  
    System.out.println("Empty? " + q.isEmpty());  
}
```

Ensure that your output is as follows when you run your program:

```

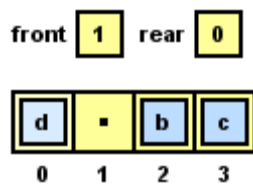
----jGRASP exec: java CircularQueue

a
b
c
d
e
Empty? true

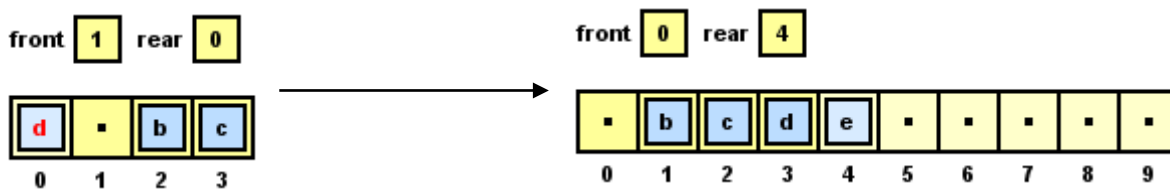
----jGRASP: operation complete.

```

The array initially becomes full after "c" is added. One element is then taken out, but the queue becomes full again when "d" is added:



Your array should have expanded when the string "e" was added to the queue



Deliverable: Create a new program called QueueClient with a main method that performs the following steps:

1. Create a CircularQueue object with an initial capacity of 4.
2. Add the following String objects to the queue:
"Red", "Yellow", "Green", "Blue", "Purple", "Orange", "White"
3. Remove 2 elements from the queue.
4. Add the following element to the queue "Grey".
5. Create a loop that removes and prints all elements from the queue.

Answer the following questions:

- If you wanted to create a size() method to return the number of elements in the queue, what would you return?

Show both the CircularQueue class and the QueueClient class to your TA. Execute the QueueClient class to show the output to your TA.