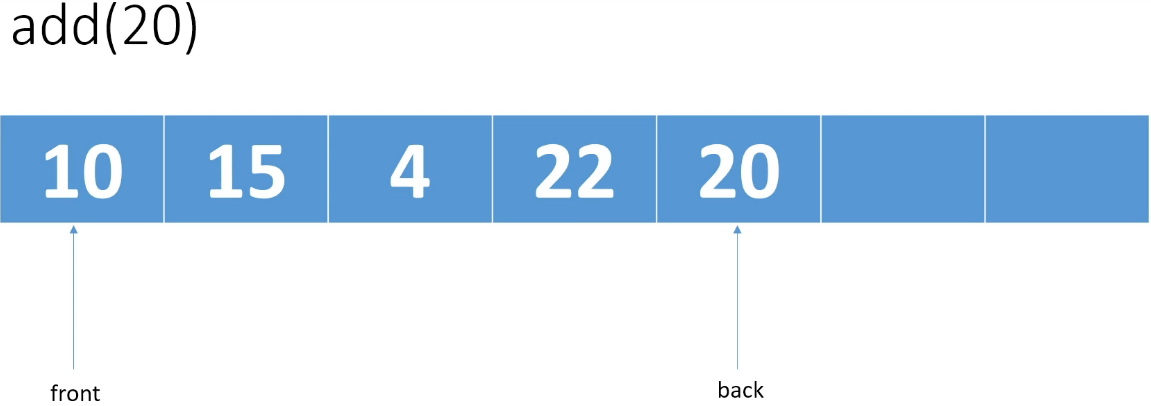
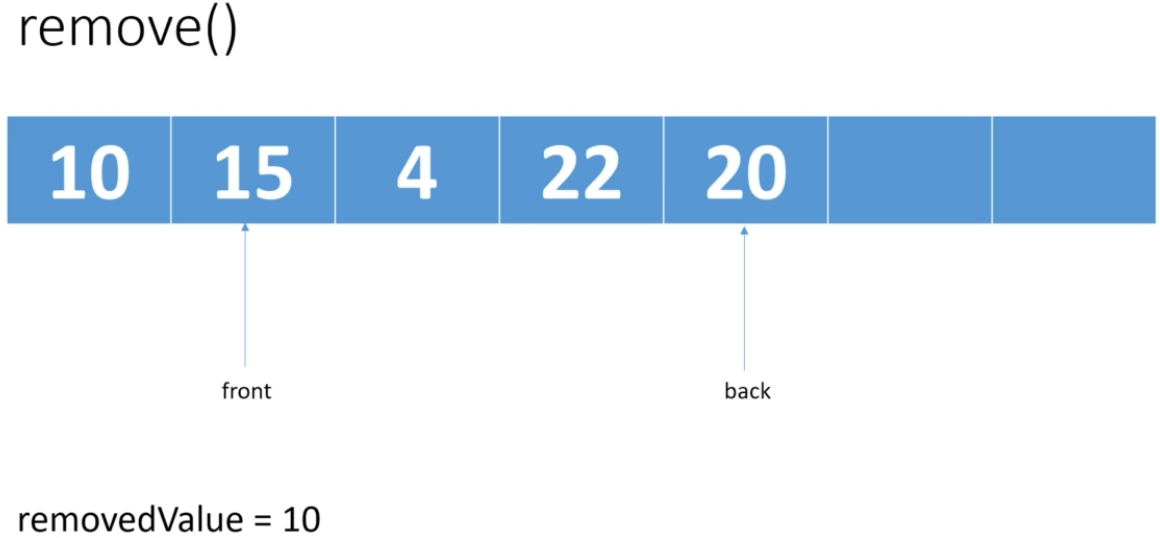
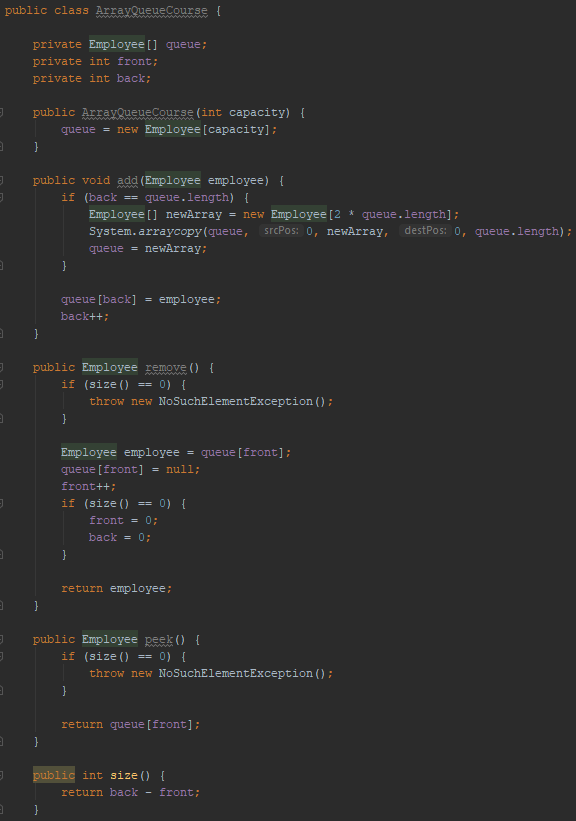
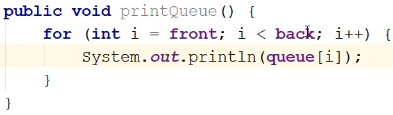
**Introduction to Queues**

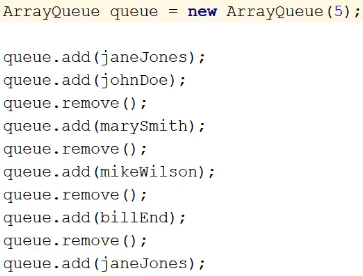
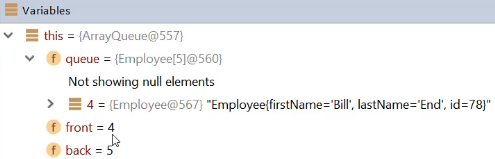
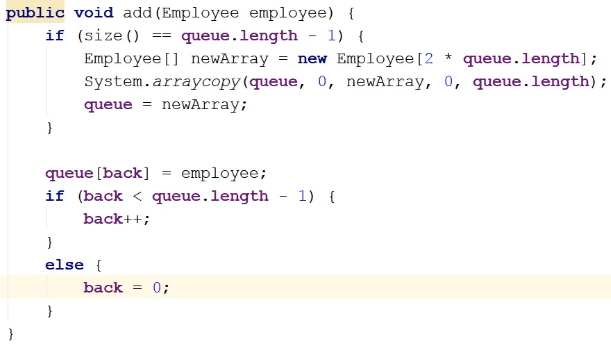
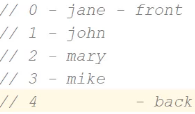
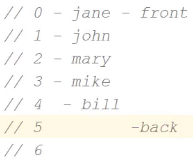
**Queues(Theory)**  
\* Queues are an **Abstract Data Type** like Stacks, **we use other data structures to implement them.**  
\* They don’t dictate how you store the data, but they do describe the way you can access the data.  
\* **FIFO** => **first in, first out**  
**1) => add** / **enqueue** => add an item to the end of the queue  
**2) => remove** / **dequeue** => remove the item at the front of the queue  
**3) => peek** => get the item at the front of the queue, but don’t remove it  
\* Items are removed from the queue in the same order that they’re added to the queue.  
\* There are 2 popular ways of implementing Queues:  
1) **Arrays**  
2) **Linked Lists**

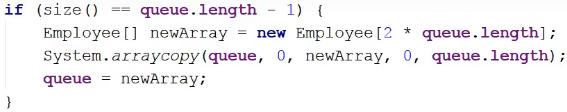
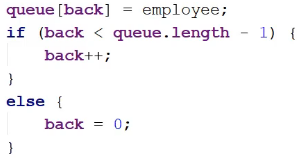
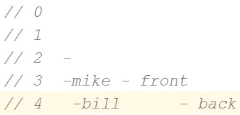
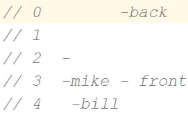
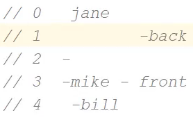
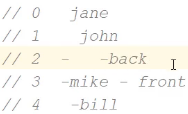
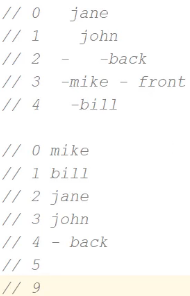


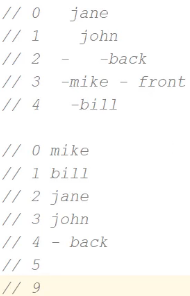


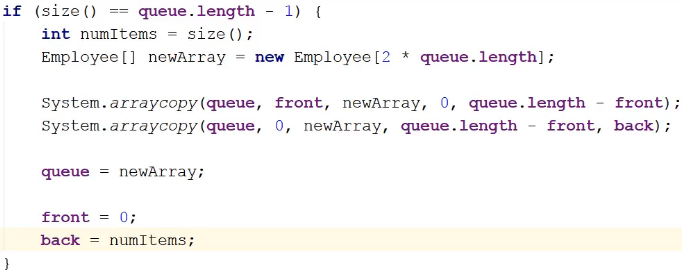
\* **Depending on the implementation, you might want to clean up the 10**.  
\* If you add an item, the back pointer will change.  
\* If you remove an item, the front pointer will change.  
\* Time Complexity would be similar to what we had with Stacks.   
=> **It’s going to depend on what you’re backing the Queue with**.  
\* If you’re backing the Queue with an **array**, then whether an operations is going to be   
**O(1) or O(n)** is going to depend on whether you have to resize the array.  
\* If you’re backing the Queue with a **Linked List**, then these 3 Queue operations will always be  
**O(1)**

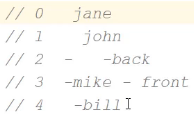
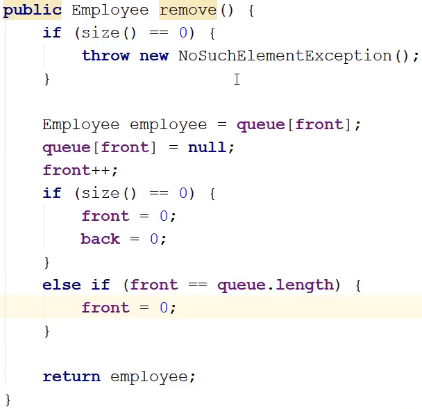
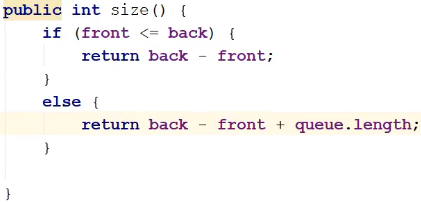
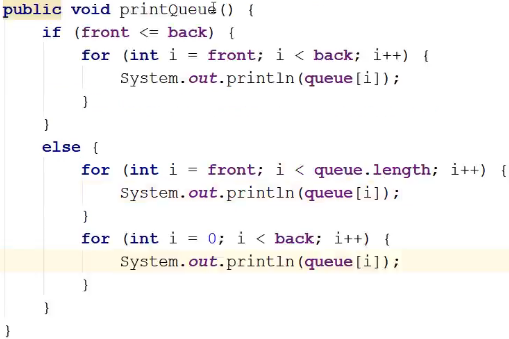
**Queues (Array Implementation)**  
**(Implemented my own version before watching this video)**  
\* I’m not using **GENERICS**, I could’ve created a queue and used generics and then the **queue** can be **used with any type of object**.  
  
  
**throw new NoSuchElementException**

**Circular Queue Implementation (Part One)**  
\* In the last video I said that we could improve on the implementation of our queue.  
=> Think about the case where we add 2 employees and then we remove 1, add 1, remove 1, add 1, remove 1 and we keep alternating like that because in that case we never have more than 2 employees on the queue.  
=> And so if we created a queue of length 10 or of length 5, the queue should never need to be resized because we always have fewer items than the size of the queue.  
=> But will that be the case?  
=> Let’s demonstrate what’s wrong with our current implementation:  
  
\* Front grows by 1 with each remove().  
\* Back grows by 1 with each add().  
back = 0 1 2 3 4 5 6  
front = 0 1 2 3 4  
\* **At the end of this, front is 4, back is 6, so the back hits the condition to resize the array**.  
  
\* So even though 80% of our queue is empty, we’re doubling the length of the array and then if we keep doing this add() remove() thing, at some point we’re going to double it again.  
=> So it would be nice if we could use the empty space at the front of the array, and we can:   
**Circular Queue** => **wrap the back of the queue to the front of the array**.  
\* So rather than incrementing back to 5, we’ll wrap it to the front.  
\* **And so the front part of the queue will actually be at the end of the array and the back part will start filling up at the beginning of the array**.  
\* That way we can make use of all this empty space and our queue won’t have to be resized as often.  
  
**add() code changes**  
\* We need to change the check to resize the array - using size() and length -1.  
\* We don’t necessarily want to just increment back++ because we might want to wrap back to the front of the queue.  
  
=> If we come in here and we add Bill, we can’t wrap back to the front because that’s not the next available position. When we add Bill in here, we want the next available position to be 5 and that means that we’re going to have to resize the array.  
  
=> And so in this case the size() which is 4 == queue.length -1 and that means that the condition to resize the array is met, so we would resize the array and then Bill would be at 4 and back would be 5.  
And then the condition to back++ will be true.  
\* This is why we have to resize the queue when we’re 1 element short of the length of the array and why we have the test for back++. Because once we’ve resized the queue, we know the back will be < queue.length -1 and we can just go ahead and increment it.

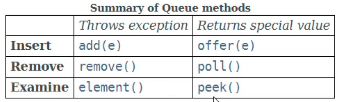
\* Now let’s take a look at the alternating case:  
  
  
  
=> We’re not going to resize because when we do the test here, we’ve only got 1 employee Mike when we came in and so we don’t need to do a resize.  
=> We’re going to add Bill into back which is currently 4.  
=> **Then the check `back < queue.length -1` fails, and so we wrap back to the front of the queue by setting back = 0**.  
 add Jane:  add John:   
=> Well now if we want to add let’s say Mary, there’s room for Mary here at the back position but we can’t increment back. We are going to meet the resize condition because we have 4 items in the queue so we’re going to resize the array. BUT if we just copy in the elements exactly as they are, that obviously won’t work because if we add Mary in here at position 2, it doesn’t matter if we ‘ve added more space at the end because we can’t increment back.  
=> **So we have to change how we’re resizing the array**.  
=> When we resize the array, we’re going to unwrap the queue, so basically we’ll copy the elements from the front of the queue to the back of the array - into the front of the resized array and then we’ll add the remaining elements in the queue to the end.  
=> And so afte we’ve resized, our front will be reset to 0 and the back will be the size() - the number of elements because that will be the next available position.  
\* So we’ll have:  
  
\* And as you can see, we’re in the same situation here as we were if we added 4 items and then wanted to add a 5th. Here we’ve added 4 items when we’ve wrapped the queue and we still get into the situation where we can’t increment back because if we do, it’s going to be pointing to something that’s already occupied.  
=> So when we have 1 fewer items in the array than the length of the array  
(it doesn’t matter whether the remaining free spot is at the end of the array or somewhere in the middle of the array), we still have to go through the resizing because we have the same problem of not being able to increment back++ or in the case of the end of the array, we can’t wrap back to the beginning of the array by setting back = 0.  
\* So let’s change our code to resize the array and copy the elements in such that we’ll take the elements from the front of the queue and copy them to the front of the resized array and we’ll copy the elements in so that when we’re finished, our front is reset to 0 and the elements are in the order they were added and the back will be at the end.  
\* So essentially in that resizing part we’re not just resizing the array, we’re also moving the queue, we’re unwrapping the queue potentially and moving it to the front of the array.

**Circular Queue Implementation (Part Two)**  
\* So we’re going to copy from the front to the end of the array and then from the beginning of the array till the last element in the queue - before back.  


\* 1) queue.length = 5, front = 3, so 5 - 3 = 2 items to copy.  
\* 2) start inserting at position queue.length - front which is 2 and copy `back` elements, so 2.  
\* Because the queue is wrapped at the front, `back` points to the first available position.  
\* **`back` basically always equals the number of items that have been wrapped to the front**.  
  
\* We can’t set `back = size()` because `back` is still set to the old back.  
\* So if the queue has been wrapped, it’s been reset.  
\* If the queue has not been wrapped and we’re resizing it, this still works but we’ll be doing a redundant copy here (the second one). So it’s an unnecessary copy and if wanted, we could check for that and try to figure out if the queue has been wrapped or not.  
\* I’m going to leave it as it is because **when you’re using a CircularQueue, unless you’re only ever adding stuff, and you add a whole ton of stuff before you remove it, most of the time the queue’s going to wrap at some point**.

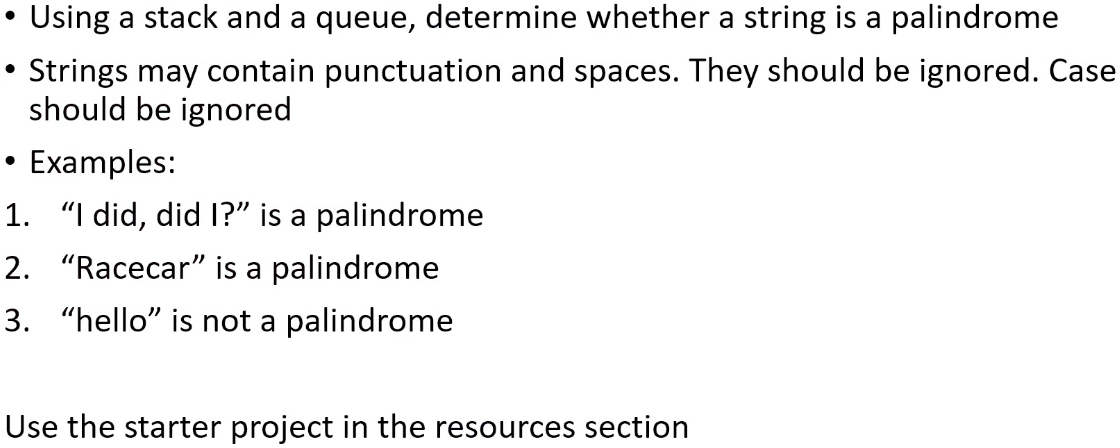
**remove()** **code change**  
\* We don’t have to make that many changes.  
\* But now we have to worry about wrapping front.  
  
=> Let’s say we remove() Mike, and so front++ gets incremented to 4 and then we remove Bill and front++ gets incremented to 5.  
=> Well that this point, we want front to wrap around to 0, to Jane.  
  
\* Remember, when we’re doing this test, we’ve already incremented front.  
=> After removing Bill, front = 5 and our condition is true.  
=> If front was less than the queue.length, then that means there’s more space so it’s safe to have front sitting there, it’s not going to be an index out of bounds exception.  
**size()** **code change** - it’s possible that back will be less than front so that won’t work anymore.  
=> We’re going to test here whether the queue has wrapped.  
  
**printQueue()** **code change**  
=> We’re going to test here whether the queue has wrapped.  
  
\* **That’s it, we’ve now updated our queue to be a Circular Queue**.  
\* **If we add 5 items, we’re resizing when we have 4 items, so we’re resizing when there’s still 1 empty space in the array, but it’s a lot better than resizing when 80% of the array is empty**.  
\* With this implementation, we’ll only ever resize when we only have 1 space left.

**Queues and the JDK**  
\* Time Complexity for a Queue backed by an **array** will be **O(n) or O(1)** based if whether we have to **resize the array**.  
\* **Doubly Linked List** is **perfect for queues**, you add items to the tail of the list and you remove items from the head of the list, it’s perfect and all of the operations add() remove() peek() would be performed in **O(1)**.  
\* To implement it, we’d do exactly what we did when we implemented a Stack using a Linked List.  
=> We would wrap the LinkedList class with some other class that we would call a Queue let’s say and we would just expose the add(), remove() and peek() methods.  
\* It’s simple to do so we’re not going to do through it.  
\* The JDK has a **Queue interface**.  
**add() => throws an exception if it fails.  
remove() => throws an exception if it fails.  
element() => returns null if it’s empty.**

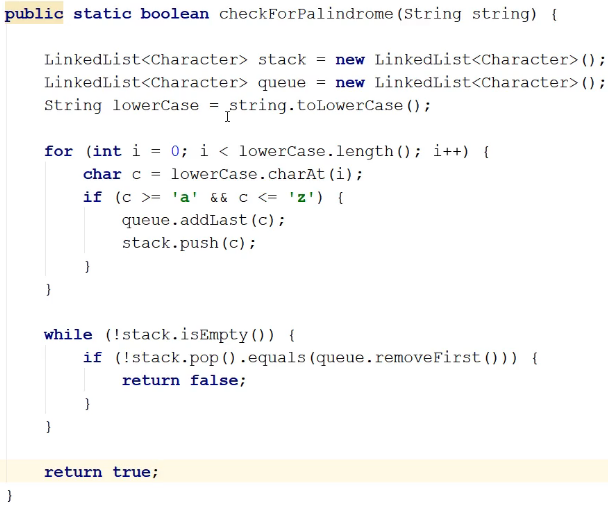
**poll() => the difference between poll() and remove() is that in the remove() case if the Queue is empty, you get an exception and in the poll() case if the Queue is empty, you just get a null.  
offer() => inserts the specified element into this queue if it is possible to do so immediatelly without violating capacity restrictions.  
peek() => throws an exception if it’s empty.**  
\* **Quite a lot of classes implement the Queue interface**.  
<https://docs.oracle.com/javase/9/docs/api/java/util/Queue.html>  
  
  
=> The first column methods throw an exception if they can’t do something.  
=> The second column methods return a specific value if they can’t do something.  
**ArrayBlockingQueue  
java.util.concurrent**<https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ArrayBlockingQueue.html>  
\* It’s a **bounded blocking queue** backed by an array.  
=> **It doesn’t resize the array**.  
=> Once created, the capacity can’t be changed.  
\* Attempts to put an element into a full queue will result in the operation blocking.  
\* If a Thread tries to **add()** an element into this queue**, it will block until some other Thread** has come along and taken an element out of the queue.  
\* The same goes for **remove()**. If the queue is empty, then you’re going to block until something is added to the queue.  
\* **It’s meant to be used when more than 1 Thread is going to be accessing the queue**.  
\* So often this type of queue is used in Producer - Consumer scenarios.  
\* Let’s say for example messages are being passed over a network from a producer to a consumer, the producer/sender adds messages to the queue and the consumer/receiver will remove messages from the queue.  
\* If the producer/sender tries to add a message to the queue and the queue is full, it’s going to block until the consumer/receiver has removed a message from the queue. And of course if the receiver goes to take a message off the queue and there’s nothing there, it’s going to block until the sender has sent another message.  
\* This Producer - Consumer scenario is perfect for FIFO behaviorhen you want the messages to be read in the order that they’re sent.  
**ConcurrentLinkedQueue**  
**java.util.concurrent**<https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html>  
=> It’s a **unbounded** **Thread-safe** queue based on linked nodes.  
=> It’s using an **efficient non-blocking algorithm**.  
\* Unlike in most collections, the **size()** method is not O(1), it’s **O(n)** - normally to get the size, the number of steps doesn’t depend on the number of items in the list because you’re usually just keeping a counter but because this implementation of the queue can be accessed by multiple Threads, determining the number of elements requires a traversal of the elements and so that not going to be a constant time operation.  
\* Just like with the List, **if you want to implement your own Queue, you might want to start with AbstractQueue** class because that will have implemented the methods for Queue and you can just override the ones that you want to customize.  
\* **LinkedList** **implements Queue interface**.

**Deque interface**  
**java.util**<https://docs.oracle.com/javase/9/docs/api/java/util/Deque.html>  
=> Deque supports insertion and removal at both ends.  
=> Deque is short for **Double Ended Queue**

\* It’s usually pronounced like “deck”.  
\* As we saw when we looked at Stacks, the JDK has   
**ArrayDeque** class  
<https://docs.oracle.com/javase/9/docs/api/java/util/ArrayDeque.html>  
=> **Resizable array implementation**.  
\* So there’s no capacity restrictions.  
\* Because it implements the Deque interface, it has methods for   
**addFirst()  
addLast()**  
=> Because with the Deque we’re allowed to add at the beginning and at the end of the queue.  
\* **LinkedList** also **implements the Deque** interface.  
=> So if you have **a situation where you want a queue that you can add and remove items from both ends**, then using **one of the classes that implement the Deque** interface would be good for you.  
\* **Queues are used way more often than Double Ended Queues**.  
\* That’s why we went into more detail about Queues.

**Queues Challenge**  
 **\* You can use classes in the JDK.  
\* Your solution should contain push() and pop() and queue methods like addLast(), removeFirst().  
\* So using your choice of Stack and Queue, go ahead and use the starter project.**

**Queues Challenge Solution**  
**(Implemented the challenge)**  
\* The key to this one is realizing that because   
Stacks are LIFO - last in, first out  
Queues are FIFO - first in, first out  
\* First we remove the punctuation and spaces.  
=> If we push everything onto our stack, then the stack will contain the original string.  
=> If we add everything to our queue, then the queue will contain the original string.  
=> Then all we have to do is loop through the stack let’s say, and and we’ll pop each character off the Stack and compare it against the top element in the Queue.  
=> And because the Queue will return the characters reading from left to right and the Stack will pop the characters in the order of right to left, we just have to - on each iteration - pull the first item of the Queue and pop the top character on the Stack and compare them.  
\* If they’re equal, then we can continue.  
\* If they’re not equal, that means we don’t have a palindrome.

\* I’m going to use a **LinkedList** from the JDK for both the Stack and the Queue.  
  
\* **It’s a shorter and more elegant solution** than the one we did in the Stacks challenge but we hadn’t learned Queues yet in the Stacks challenge and so that solution involved a little more manual work.

**Resources**  
Queue interface javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/Queue.html>  
ArrayBlockingQueue class javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ArrayBlockingQueue.html>  
ConcurrentLinkedQueue class javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html>  
Deque interface javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/Deque.html>  
ArrayDeque class javadoc  
<https://docs.oracle.com/javase/9/docs/api/java/util/ArrayDeque.html>