Introduction

This weeks practical asked us to do two things: create a stack from a basic interface we were told to implement, and a priority queue which would work like a regular queue when it came to store the values, but would select the element deemed to have the highest priority. All tests are in the test.java class.

Design/Implementation

Stack:

To implement the stack, there was an interface we were to implement for both the stack and the double stack. Initially I made sure that the stack worked, before moving on to implement the double stack. For the stack I read in the array from the double stack, along with a flag saying whether the array was to be read from the front or from the back. There were then a number of methods to be implemented.

Push – Which initially worked well, I used the flag to check whether it was to be read from the front or back. Then i would check if the next value was not null, if it wasn't then I would increase the size of that stack, and also change the value at that point in the stack. This was a strange method to implement, initially I had tried to make it so that the program would check for a meeting point between the two stacks, but when I tried to implement it, the two stacks being separate objects so i couldn't find a working way for them to share the same value. So in the end I removed a lot of the code I had, and added the much simpler counter, along with the check if the next value in the array was a null value. This made this work in a significantly more simple way.

Pop – This method worked quite well once I changed the implementation to make Push work with the counter. I didn't encounter many problems, and feel like this is a relatively good way to implement this method.

Top – This method works quite well, and I encountered no problems when implementing it.

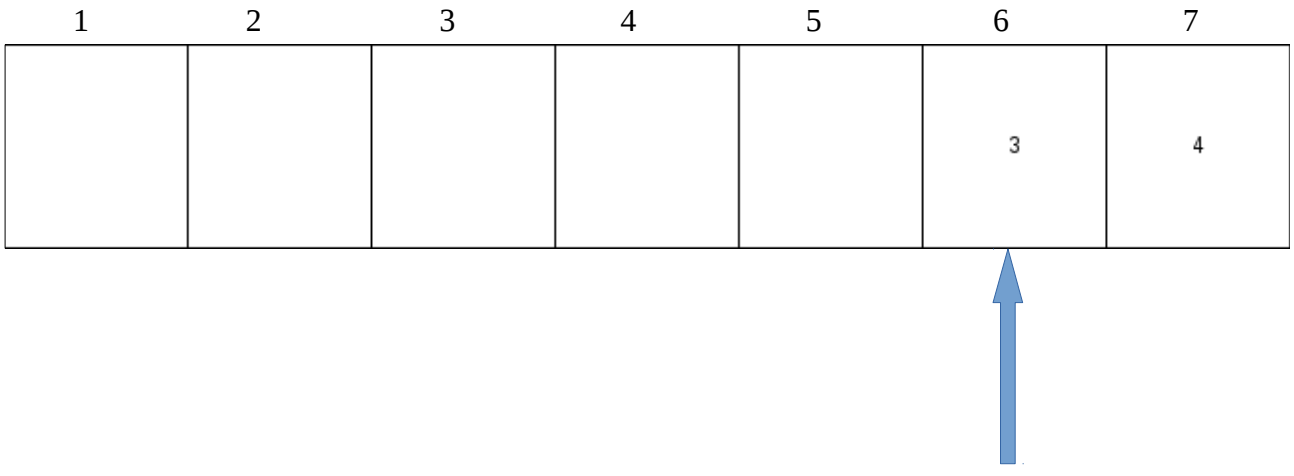
Size – Initially, I had a number of complex pieces to this method that were really quite unnecessary. Then I changed the implementation for push which made this method so much easier to implement correctly.

Is Empty – This again was relatively simple to implement once I had the counter for the size.

Clear – Initially I had made this method so that it would move through the entirety of the double stack deleting everything. Then once I had the counter, I changed it so that it would only go so far as the end of the individual stack. I thought this was a superior method, as it gives more flexibility to the user, while giving the potential for the old version to be duplicated if the user wants by simply clearing both stacks.

Double Stack:

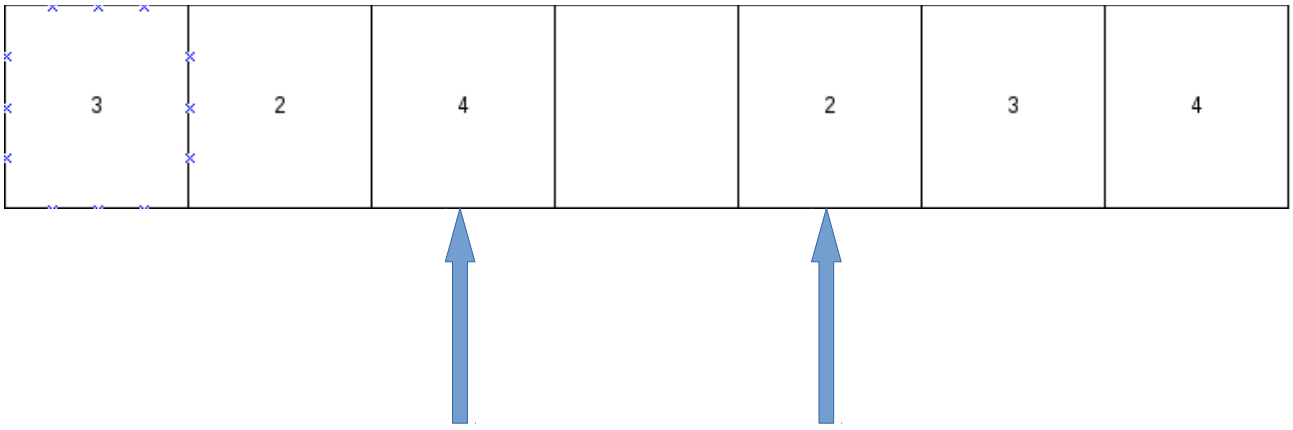
In this, I had only a few methods, but they were linked to the methods held in Stack. To implement this, I created a new array of a fixed size, which I would then populate using the two Stacks. These stacks would not be of a fixed size, but would expand until one encountered the other.



Stack one begins in position -1

This is the head of the second Stack

The Second Stack begins in the position 8 in this example



This is the current location of the Stack1 pointer to the head of the stack

This is the current location of the Stack2 pointer to the head of the stack

In the second diagram the middle blank value will be taken up by which ever stack has push called first. If another push was called regardless of which it was a stack overflow error would be thrown.

Testing:

Test Number	Reasoning	Success
1 (normal Stack)	I used this test to make sure that the double stack worked when I pushed enough to fill both stacks equally to the full size of the double stack	**Yes**
2 (normal Stack 2)	I used this test to the same effect as the one above, but in this case I pushed the stacks in the opposite order	**Yes**
3 (overload stacks)	I used this case to test if the Stack would throw a Stack Overflow Exception when it was supposed to	**Yes**
4 (overload Stacks 2)	I used this test to the same effect as the one above, but in this case I pushed the stacks in the opposite order	**Yes**
5 (push null)	I used this test to test whether or not the program would be able to process null values	**Yes**
6 (unfilled Stack)	I used this to test if the program still worked correctly even when it wasn't filled	**Yes**
7 (normal stack pop)	I used this to test if the program could correctly pop when there were values in the double stack	**Yes**
8 (pop)	I used this to test if the program would pop the value of the top element, and return it correctly	**Yes**
9 (pop fail)	I used this to test what would happen if I tried to pop on an empty stack, with the intention of throwing a Stack Empty Exception	**Yes**
10 (pop fail 2)	I used this to the same effect as the test above, but I added one value then popped twice to test if that worked again to throw the exception	**Yes**
11 (top)	I used this to test if the top method really did return the	**Yes**

	value which would be on top of the stack	
12 (clear)	I used this to test if the clear method worked by only clearing one stack	**Yes**
13 (clear 2)	I used this to test if the clear method worked when used to clear both stacks	**Yes**
14 (priority normal)	I used this to test if the priority queue worked correctly	**Yes**
15 (priority de queue)	I used this to test if the de queue method worked correctly, by checking the size and also the value which was de queued	**Yes**
16 (priority Empty)	I used this to test to see if the is Empty method worked	**Yes**
17 (priority Empty 2)	I used this to check if the is empty method worked when the queue was indeed populated by returning a false value	**Yes**
18 (priority clear)	I used this to check if the clear method worked correctly	**Yes**
19 (priority overflow)	I used this to add too many values to the queue, to see if it throws the exception it was supposed to	**Yes**

Conclusion:

I feel that this practical went relatively well. I did struggle more than I should have initially, but once I found out how I should have done the practical from the beginning, it went much more smoothly. Overall I believe that my implementation is a decent one, and again I think it has helped with my understanding of Test Driven Development. One improvement I would have made would be the introduction of a working factory method. I had implemented one, but every time I attempted to make it work it would break the tests that worked without it, so I left it out of the implementation buy will include it in the source code as an example.