Overview

For the fourth practical we had to develop an implementation of the ADT interfaces for a double stack and priority queue. Like the second practical, it was required we make unit tests and therefore making sense to develop the practical in a TDD process, thus implementing all the given classes after writing suitable tests. It was also required to use a singleton factory class in order to encapsulates the object instantiation process.

Design

The framework for this practical was already given to us and therefore all we had to do was implement it and fill various method bodies. I have not edited the Exception, Interface and Factory classes. All of the implemntations have an insert time o

Double Stack: Insert: O(1), Remove: O(N)

In order to make efficient use of resources, the "Stack" class utilizes all the available space in the array and while there is space in the array, either stack wont overflow, as both stacks grow in opposite direction and can occupy any unused space in the array.
The constructor in the "Stack" class, takes in an array of Object and an integer ID, both from the "DoubleStack" class. In the constructor the current index, in the array, for both the stacks is initialized and stored statically. The array is also stored statically and holds the data for both stacks. Finally, the ID is used to identify which stack is which. This is important when the methods in the "Stack" class are being called. Each method will first identify, using this ID, which stack is calling the method. Both stacks start with an index past the two extreme corners of the shared array. Stack1 starts with an index of – 1 and the first element in stack1 is pushed at index 0. Stack2 starts with an index equal to the length of the array and the first element in stack2 is pushed at index (n-1). When called, the "push" method first checks if the index of the second stack is equal to that of the first, indicating the stack is full. If so a "StackOverflowException" exception is thrown. Otherwise the element is pushed to correct position in the array.
The "pop" method first checks whether, if stack one called it, the index of stack is not -1 or if stack2 called it, the index of stack 2 is not equal to the size of the array. If so a "StackEmptyException" exception is thrown. Otherwise if stack1 called the method stack1's index is decremented, or if stack2 called the method, stack2's index is incremented and the removed element is returned by the method. When the push method is called again the removed data in the array is simply be overwritten.
The "top" method works in a similar way to the pop method, but doesn't the change in the indexes of the arrays. The other methods work using the indexes of the stack to determine the size of a stack, if the stack is empty and to clear the stack.

Priority Queue: Insert: O(1), Remove: O(N)

In the implementation of the Priority Queue, the constructor sets the max size for the queue and initializes an array of abstract class Comparable, thus meaning every time we specify the type of the items in the queue, we specify the Comparable class. The integer attribute "index" is the index of the next available location in the array.
The first method is the "enqueue" method, which first checks if the index of the queue is equal to the size of the queue, indicating the queue is full. If so a "QueueFullException" exception is thrown. Otherwise the element is added to the next free slot in the array.
The next method in the class is the "dequeue" method. This method traverses the array to find and remove the largest item. Because each element in the array is of type Comparable, the "comparteTo" method can be used to compare the elements and find the largest one. It should be noted that the largest element is defined by Comparable class. As the program traverse the array, the variable "largestElementIndex" is used to keep track of the index of the largest element we have seen so far. The largest element found in is then replaced by the element at the end of the array. Again the other methods work using the indexes of the queue to determine the size of a queue, if the queue is empty and to clear the queue.

Priority Queue Using Double Stack: Insert: O(N), Remove: O(1)

A different implementation of the priority queue ADT was possible using two stacks, accessed through the "DoubleStack" class. The "PriorityQueueUsingDoubleStack" like the "PriorityQueue" class implements the "IPriorityQueue" interface and therefore the methods are the same as those found in the "PriorityQueue" class. The main difference between the two classes is that the "enquue" method is now the more substantial method.
The first stack used is simply a temporary storage stack. The second stack is the queue and the element you pop represents the next element in the queue. Whenever the "enqueue" method is called the method checks the queue is full, if so a "QueueFullException" exception is thrown. Otherwise a check is first preformed to see if the second stack is empty, if so the element is just added to the stack. Otherwise if there are already elements in the second stack and a new element is added, the second stack pops elements, until the correct place to push the new element is found. Every popped element goes onto the temporary stack. Once the newly added element has been inserted into the second stack the program begins to pop from the first stack and push the elements back onto the second stack.

Priority Queue Using Linked List: Insert: O(N), Remove: O(1)

I have also created another class which implements a priority queue ADT, this time using a linked list. This class implements the "IPriorityQueue" interface, however I have had to add an extra method "insert". Like the "PriorityQueueUsingDoubleStack" class the most substantial method is the "enqueue" method. The "enqueue" first checks the queue is full, if so a "QueueFullException" exception is thrown. Otherwise a new instance of the Node class is initiated and set equal to the node at the front of the queue. Next the node at the front of queue is set equal to the result of the recursive method "Insert" which returns a node object. The "Insert" method takes in two parameters a node and an element, initially the node is the copy of the node at the front of queue made earlier, and the element is the element to be added to the queue. Once this method is entered another new node is created with the new element to be added. Following this the method checks if the queue is empty if so the front node is set equal to the new node. Otherwise the program will recourse using the pointer to next node and the original element as the input to method, until the correct position in the queue is found.

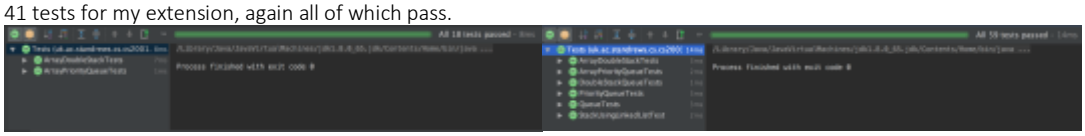
Queue Using Double Stack Insert: O(1), Remove: O(1)
The final implantation of a queue ADT I attempted was a just a regular queue using two stacks which again was possible using the "DoubleStack" class. The "QueueUsingDoubleStack" like the class implements the "IPriorityQueue" interface and therefore the methods are the same as those found in the "PriorityQueue" class. The first stack is used for "enqueue" and the second stack is stack is used for the "dequeue" operation. The "enqueue" method simply pushs an element to first stack and so like the original "PriorityQueue" class the most substantial method was the "dequeue" method. The "dequeue" method first checks if the second stack is empty. If it is every element in the input stack is popped and pushed into the second stack, effectively reversing the order of the first stack. The top element is then popped from the second stack.

Stack Using Linked List
I also decided to look at other implantations of stacks and went on to make an implantation of the stack ADT using a linked list. The "StackUsingLinkedList" like the "Stack" class implements the "IStack" interface and therefore the methods are the same as those found in the "Stack" class. The pop and push elements are quite simple in my implantation, when popping an element only a reference requires modification. When new elements are pushed a new node instance is created storing the element and pointer to the node that was at the front of the stack, the "first" attribute in the class is then updated to the new node. The links keep the stack intact and mean the stack can be completely traversed once emptied. A befit of this implementation is there are no upfront memory costs as you only consume the space required per node.

Uses of Stacks and Queues
A stack is an example of a FILO (First In Last Out) data structure in which data items can be pushed (added to the stack on top of all previous data items) or popped (taken from the top of the stack. The Deapth-First Search is an example of where a stack is used. The depth-first search goes down a path until it gets to a dead end; then we backtrack or back up (by popping a stack) to get an alternative path.
Stacks have many other applications, for example an undo or back button, where data is stored in the order it came in. Other examples where a stack is the data structure used would be parsing or a recursive function.

A queue is a FIFO data structure that behaves in much the same way as queues containing people. Data items are added to the queue at one end and removed from the other. As well as a queue being the data structure used to create buffers in memory a queue is also used when using the Breadth-First search. The Breadth-First search explores all the nearest possibilities by finding all possible successors and enqueue them to a queue.
A priority queue can be the data structure used for a computer scheduler which is responsible for storing processes to be executed by the CPU in a multi-tasking environment. The role of the scheduler is to ensure that the CPU is utilized as much as and that tasks are completed quickly in a fair order. Fixed Priority Scheduling assigns each process a priority, with high priority processes effectively jumping the queue.

Testing
Because the practical specification says that the extensions should all be tested using the same tests, I have used parameterized tests, which a feature in Junit 4. This has allowed to run the exact same set of test for all the queue classes. To do this the program need public static method that returns a Collection containing the classes to be tested, I also needed a public constructor that uses the classes.
After using the automated tester all my 3 tests passed in my basic requirement, one test failed however this is because of the use of parameterization in Junit. For my basic requirement I have written 18 different tests, all of which pass. I have then written a further 41 tests for my extension, again all of which pass.



Evaluation
After testing my program, I pleased to say it works as expected and all the tests pass, where the tests include erroneous and extreme data, attempting to break the program. I have also learnt a lot about the different implantations of the Priority Queue and Stack ADTs and have been able to implement each ADT in different ways, comparing the performance.

Conclusion
In conclusion I'm satisfied with end implantations of the ADTS, they all work as expected and have different performance characteristics. Moreover, the testing proves the implementations are able to carry out the required tasks with ease and is are to handle erroneous data without crashing making them functional and robust.