Restorable Queue

This project will have you implement a queue with a twist. This representation of a queue will include restorability characteristics by implementing a *Memento design pattern*.

You are to work in pairs on this project to ensure completion by the deadline.

The Queue

It is standard in Java to implement Queue data structures by implementing the Queue Interface. However, for this program, the Queue interface requires several methods that are of little interest. Instead, your `RestorableQueue<T>` will implement the following queue-based methods.

Method	Description
`public int size()`	Returns the number of elements in this collection.
`public boolean isEmpty()`	Returns true if this collection contains no elements.
`public boolean contains(Object o)`	Returns true if this collection contains the specified element.
`public Object[] toArray()`	Returns an array containing all of the elements in this collection. The first item in the list is the first item that would be removed by `dequeue` and the last item is the most recent item by `enqueue`.
`public void clear()`	Empties this queue resulting in a queue of size 0.
`public void enqueue(T item)`	Inserts the specified element into this queue.
`public T dequeue()`	Retrieves and removes the head of this queue, or throws `NoSuchElementException` if this queue is empty.
`public T peek()`	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

Your implementation will wrap around an instance of the provided `DoublyLinkedList` class and thus can be thought of as an example of an *object adapter design pattern*. This pattern is not described in the text, but here is a link with a brief description. Basically, the adapter contains an instance of the class it wraps; in our case we are wrapping `RestorableQueue` around an instance of `DoublyLinkedList`.

Restoration via the Memento Design Pattern

For some classes it is beneficial to capture and externalize an object's internal state. This externalization allows a user (*caretaker*) the ability to restore the object to a previous state at any time, if they so choose. We refer to this idea as the *memento design pattern*; please review this pattern described briefly here (although do not mimic the source code since it externalizes memento objects).

We will implement our memento using a slightly modified approach that *does not* externalize the object's state.

- The *originator* class will be the `RestorableQueue` class.
- `Memento` is a private, nested class in the `RestorableQueue`. A memento object will then store a shallow copy of the entire contents of the queue when a snapshot is taken (via `saveState`).
- We will store the saved queue states of the `RestorableQueue` class as a `Stack` of `Memento` objects.

```
public class RestorableQueue<T> implements Restorable
{
```

```
        DoublyLinkedList<T> _queue;
        Stack<Memento> _reversions;

        // ...
}
```

- The *caretaker* will be our junit test methods which initiate saving the current state and then requesting restoration.
- We want a restorable class to maintain a common interface. Therefore, our `RestorableQueue` will implement the following `Restorable` interface.

```
public interface Restorable
{
        public void saveState();
        public boolean revertState();
}
```

The `saveState` method tells the originator to save the current state of the queue. The `revertState` can then be called to bring back, in total, the last saved state. We can revert the state as long as states have been saved.

Here is a sample sequence of operations that illuminate state saving and restoration.

1. Begin with a queue containing `4, 6, 2, 10` where `4` was added to the queue most recently.
2. Call `saveState`.
3. Dequeue. (Queue contains `4, 6, 2`.)
4. Enqueue `1`. (Queue contains `1, 4, 6, 2`.)
5. Call `saveState`.
6. Clear the queue. (Queue is empty.)
7. Call `revertState`. (Queue contains `1, 4, 6, 2`.)
8. Dequeue. (Queue contains `1, 4, 6`.)
9. Call `revertState`. (The queue would now contain `4, 6, 2, 10`.)

Provided Source Code

The `DoublyLinkedList` class and `Restorable` interface have been provided.

JUnit Testing

You will implement two test classes. The first class (`QueueTest`) tests the functionality associated with the Queue. The second class (`RestorableTest`) will test the functionality associated with the restorability aspects of the queue. *You must explicitly test each method of the queue.*

No output should be produced by your tests; we are seeking only a 'green' output indication in Eclipse.

Submitting: Source Code

Your code should be well documented, including docstring comments of methods, blocks of code, and header comments in each file. Junit tests should have reasonable String messages output, if failure occurs.

Header Comments

Your program must use the following standard comment at the top of *each source code file*. Copy and paste this comment and modify it accordingly into these files.

```
/**
 * Write a succinct, meaningful description of the class here. You should avoid wordiness
 * and redundancy. If necessary, additional paragraphs should be preceded by <p>,
 * the html tag for a new paragraph.
 *
 * <p>Bugs: (a list of bugs and / or other problems)
 *
 * @author <your name>
 * @date   <date of completion>
 */
```

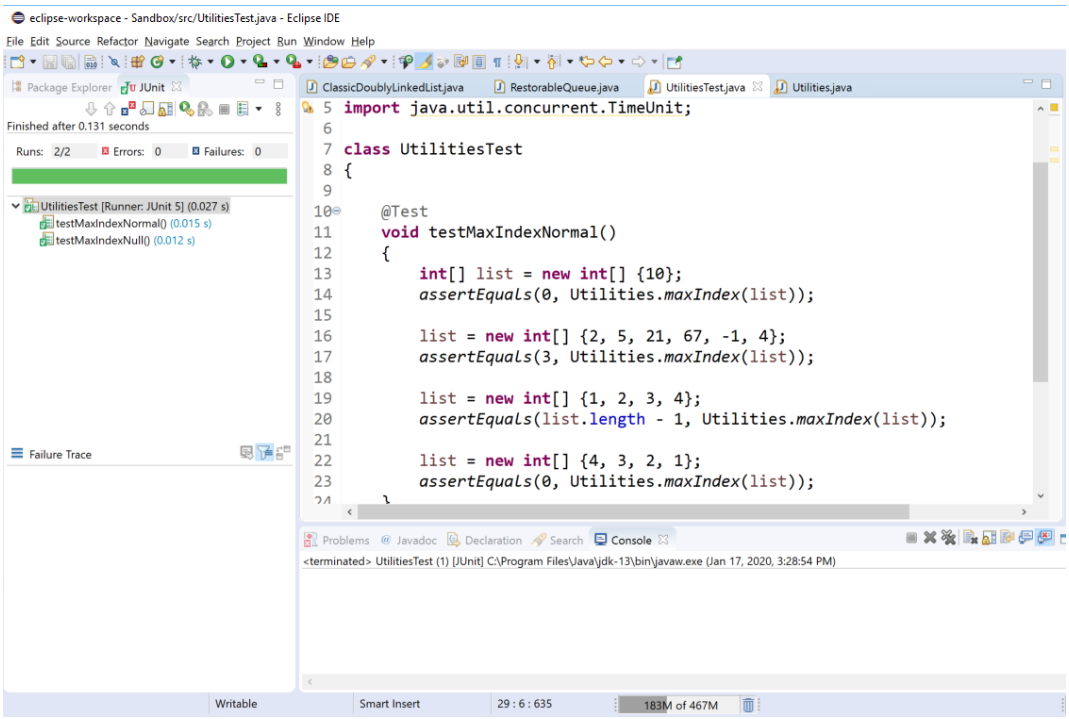
Inline Comments

Comment your code with a *reasonable amount of comments* throughout the program. Each method should have a comment that includes information about input, output, overall operation of the function, as well as any limitations that might raise exceptions; Javadoc comments are ideal. Each *block* of code (3-4 or more lines in sequence) in a function should be commented.

It is ***prohibited*** to use *long* comments to the right of lines of source code; attempt 80 character-wide text in source code files.

Submitting: Proof of Program Execution

Execute your code and take a screenshot of the associated junit window and output console (with no output). Place these screenshots into a word processing document (Word, OpenOffice, GoogleDocs, etc.). If multiple screenshots are necessary, label each clearly. Please make sure to crop and enlarge the screenshots so that the picture and / or text is clear (and doesn't strain my old eyes). For example,



Create a PDF of this document and call it `evidence.pdf`.

Source Code Files

Please submit only 3 source code files: `RestorableQueue`, `QueueTest`, and `RestorableTest`.

Final Submission File

In a folder named `lab`, place (1) the source code files and (2) `evidence.pdf`. Zip folder `lab` and label that zip file as `lab.zip`. This zip file is to be submitted via Moodle.

Please be reminded that following instructions explicitly and submitting well-formatted documents (with consistent fonts and typeset equations) in an important part of professionalism.

Only one person in your pair needs to submit the final product.