Lab 11: Implementing an Interface

Introduction

In recent lectures, we discussed using arrays, classes, and interfaces. In this lab, you will utilize all of these topics to build a simple, but yet useful new class. Consider the following interface describing the methods of a simple **double ended queue (or deque)**:

```
public interface SimpleDeque
{
    public boolean isEmpty();       // Return true if the Deque is empty and
                                    // false otherwise
    public void addFront(Object X); // Add Object X at front of list
    public void addRear(Object X);  // Add Object X at rear of list
    // If array is full, add methods should do nothing

    public Object removeFront();    // Remove and return object at
                                    // front of list
    public Object removeRear();     // Remove and return object at
                                    // rear of list
    // If the deque is empty, remove methods should do nothing and
    // return null
}
```

A queue has the behavior such that items are added at the rear and removed from the front, thereby giving a First In First Out (FIFO) access to the items added and subsequently removed from the list. No other manipulations of the data are permitted (for example, we cannot add or remove anywhere in the middle). Looking at it "in reverse", we could add new items at the front of the queue and remove them from the rear. This is still providing FIFO access, but just from a different point of view. Now, consider both adding and removing items at the rear of the list (without every accessing the front). This is called stack access and gives us Last In First Out (LIFO) access to the items (the data is removed in reverse order). The same behavior occurs if we both add and remove at the front without every accessing the rear of the list.

This simple deque above is expressed as an interface rather than a class, because we are not describing the data or how it is represented — we are simply describing its access behavior. However, to actually build a working deque, we need a class that implements the interface above. For example:

```
public class MyDeque implements SimpleDeque
{
    Object[] theData;
    int numItems;

    public MyDeque(int maxItems)
    {
        theData = new Object[maxItems];
        numItems = 0;
    }
    // Implementation of the five methods of SimpleDeque, plus
    // perhaps other methods as well
}
```

NOte that the implementation above uses an array of `Object` to store the items in the deque. Since `Object` is the base class to all other Java classes, and array of `Object` can thus be used to store any Java class types (we can even store primitive values if we utilize their wrapper classes). Also note that nothing in the `SimpleDeque` interface requires an array to be used to store the data. You will see in your CS0445 course that a linked list may in fact be a better implementation than an array in this case. However, for this implementation we will use an array because it is simple and easy to understand.

Another important thing to notice about the partial implementation above is that the size of the array used is not equal to the number of items in the deque. The number of items in the deque is maintained in the separate `int` variable `numItems`. Since Java array sizes are fixed, once the array object is created, to avoid having to recreate new array objects with each add or removal, we simply allocate an array that is some reasonable size (specified by the parameter in the constructor) when we create the deque. At that time, we also set `numItems` to zero since there are no actual items in the queue. We then increment `numItems` with each additional and decrement `numItems` with each removal. This is the same maintenance technique used in the `MovieDB` class of a previous lab. As we discussed in lecture, when the array fills, we could copy the data into a new, larger array. However, in this case, to keep things simple, we will not resize the array and simply not allow any new items to be added once the array fills.

Adding or removing at the rear of the array is a relatively simple process — to add, we simply put the new object in location `numItems` and then increment `numItems`. To remove, we store the last item in a temp object, decrement `numItems` and then return the item. It is probably a good idea to also set the location back to `null` before returning.

Adding or removing at the front of the array is a bit more complicated. For this simple implementation, we will do it in the following way:

- **addFront**: move object at location 0 to `numItems - 1` over one spot to "the right" (i.e. into location 1 to `numItems`), then put the new object into the location 0 and increment `numItems`. For example, given the array below of length 9 with 6 items in it, doing an `addFront` of 25 will have the effect shown in the three lines below:

Index	0	1	2	3	4	5	6	7	8
Before addFront	50	30	10	40	20	80			
After move		50	30	10	40	20	80		
After add	25	50	30	10	40	20	80		

- **removeFront**: store the object in location 0 in a temp variable, then move objects in location 1 to `numItems - 1` over one spot to "the left" (i.e. to location 0 to `numItems - 1`). Set location `numItems - 1` to null, decrement `numItems` and return the temp object. In effect, you are doing the opposite of what is shown in the `addFront` above.

  Note that the `addFront()` and `removeFront()` methods as described above are not implemented in the most efficient manner. If you take CS0445, you will likely discuss better ways of implementing these methods. Also note the special cases for inserting into a full list (do nothing in this case) and for removing from an empty list (return `null` in this case).

Program

For this lab, you will complete the `MyDeque` class so that it works with the simple test program (`Lab11.java`). An out line of `MyDeque` (`MyDeque.java`) is provided for you to complete. Make sure you handle the special cases shown. You will also need `SimpleDeque.java`, in which the interface is defined. Download all three files onto your computer from the CourseWeb. Then complete `MyDeque.java`, and compile, and run `Lab11.java` so that it works correctly. The output of your program should be:

```
Stack adds and removes at rear
Marge Ingmar Ingrid Bertha Herb

Queue adds at rear and removes at front
0 1 2 3 4

Queue adds at front and removes at rear
0.0 2.0 4.0 6.0 8.0 10.0 12.0 14.0
```

Grading

Demonstrate that your program works correctly to your TA by running `Lab11.java` it for him/her. Also show your TA the source code of your `MyDeque.java`. For this lab, the following rubric will be used:

- (3 points) `addFront()`
- (3 points) `removeFront()`
- (2 points) `addRear()`
- (2 points) `removeRear()`

Note that your program must work and your output must match that shown above to receive credit for these items.

Due Date and Submission

Once you completed the program, you must demonstrate your program for your Lab TA. Once your TA already checked you, **DO NOT FORGET** to submit your `MyDeque.java` file to the CourseWeb under this lab by the due date.

If you do not complete the lab this week, you may finish it and submit your code to the CourseWeb before the due date. However, you need to demonstrate it to your TA at the beginning of next week's lab.

No late submission will be accepted.