### Portfolio Function Title: DoublyLinkedList

Version number: 1

### Function Description:

Allows for a user to build doubly linked list of objects. Once a linked list is built a user can add or remove an object to in any position in the linked list including after a reference object that is already in the list. Users can also get the length of the list, check if it is empty and check if the list contains a certain object.

### Author:

Nathan Mankovich

### Date Written/Last Modified:

04-27-2015 / 04-30-2015

### How to use the Function:

To use the DoublyLinkedList program just instantiate a DoublyLinkedList<obj> with an object of your choice and then you can use the different methods in your java code.

The program is stored in the portfolio.jar file, put that file in the same folder as YourProgram.java.

Open a Windows command window and enter:

To compile the program with YourProgram.java enter:

javac -cp .;portfolio.jar YourProgram.java

pause

Then, to run the program with the provided sample-mailing-list.csv enter:

java -cp .;portfolio.jar YourProgram

pause

User Interface: Java code and command line only.

### How the Function works:

The function relies on the Node class that is an inner class of DoublyLinkedList. This class creates nodes which are populated with an object (called data), a next Node (called next) and a previous node (called prev). To link two nodes together the first node’s “.next” must be equal to the second node and the second node’s “.prev” must equal the first node. What this does is point the first node’s “.next” and the second node’s pointer to the same location. It also points the second node’s “.prev” and the first node’s pointer to the same location.

The “head” of the list is the pointer to the first node in the linked list. So, to add an element to the front of the list the head must point to a new node that contains the added object and the new first node’s “.next” must point to the old first node. Also, the old first node’s “.prev” must point to the new frist node.

The “tail” of the list is the pointer to the last node in the linked list. So, to add an element to the end of the list the tail must point to a new node that contains the added object who’s “.prev” must point to the node that used to be the last node. Also, the old last node’s “.next” must point to the new last node.

Note -- the semantics in terms of where exactly each node is stored in memory depends on the language used (ex: Java is different than C).

If an object is added to the list the integer variable called length is incremented by one. Conversely, if an object is removed from the list length is decremented by one.

If the list is empty, then the head of the list is null.

### Supported Methods (including Inputs, Outputs, Features and Results by method):

* AddToFront: adds an element to the head
  + Input: object to be added
  + Output: none
* AddToBack: adds an element at the end
  + Input: object to be added
  + Output: none
* AddAfter: adds an element after the input reference element
  + Input:
    - the object to be added
    - the reference object that the added object is to be put after
  + Output: none
* InsertAtPostion: adds an element at the input position
  + Input:
    - object to be added
    - integer position where it is to be added (the first element is zero)
  + Output: none
  + Bonus feature: If the position is less than or equal to halfway through the linked list, it is accessed using forward pointers otherwise it is accessed using previous pointers.
* RemoveFirst: removes the first element in the list
  + Input: none
  + Output: the removed object
* RemoveLast: removes the last element in the list (accessed using uses previous pointers!)
  + Input: none
  + Output: the removed object
* RemoveCertain: removes the input element from the list
  + Input: the object to remove
  + Output: the removed object
* GetFirst: gets the first element in the list
  + Input: none
  + Output: the first object in the list
* GetLast: gets the last element in the list (accessed using previous pointers!)
  + Input: none
  + Output: the last object in the list
* GetAtPosition: gets the element at the input position
  + Input: an integer of an object in the list (the first element is zero)
  + Output: the object at location of the input integer
* GetLength: gets the list length which a class variable that increments anytime an element is added to the list and decrements when an element is removed from the list
  + Input: none
  + Output: the length of the list
* CheckIfEmpty: checks if the head is null
  + Input: none
  + Output: a Boolean, true if empty
* CheckIfContains: steps through the list to see if the input list element is in the list
  + Input: object
  + Output: a Boolean, true if the list contains the input object

Most of the functions for the doubly linked list are the same as the singly linked list.

### Known problems and limitations:

If the any removing or getting methods is used on an empty list, the program will return a NoSuchElementException. This can be fixed by always checking if the list is empty before running any of these processes.