Programming Assignment 1
The Perfect Shuffle

Due Thursday April 11, 12:00 midnight

The purpose of this assignment is threefold: to make sure everyone is up to speed with Java, to practice modularity and ADTs, and to build an ADT implementation which will be used (with minor modifications) in other programming assignments. You should therefore test your ADT carefully, even though all of its features may not be used here.

You are to write a Java program which takes as input a single positive integer \( n \) from the command line, and performs a \textit{perfect shuffle} on \( n \) cards (originally in order) until the original sorted order is reproduced. Each card is labeled with a number from 1 to \( n \). A perfect shuffle is performed by splitting the deck into a top part and a bottom part (if \( n \) is odd, the middle card goes into the top part) and then (starting with the top part) repeatedly taking the bottom card from each part and placing them on top of a new deck. Thus to perform a perfect shuffle on the deck \( 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \) (i.e. \( n = 9 \)), we begin by splitting it into a top part \( 1 \ 2 \ 3 \ 4 \ 5 \) and a bottom part \( 6 \ 7 \ 8 \ 9 \). We start the new order with 5 on the bottom, then 9, then 4, and so on until the new order is \( 1 \ 6 \ 2 \ 7 \ 3 \ 8 \ 4 \ 9 \ 5 \). (Note that if \( n \) is odd, then the first card remains in its original location after the shuffle.) Repeatedly shuffling a deck of size 9 produces the following:

Shuffles Deck Order
\begin{verbatim}
0   1  2  3  4  5  6  7  8  9
1   1  6  2  7  3  8  4  9  5
2   1  8  6  4  2  9  7  5  3
3   1  9  8  7  6  5  4  3  2
4   1  5  9  4  8  3  7  2  6
5   1  3  5  7  9  2  4  6  8
6   1  2  3  4  5  6  7  8  9
\end{verbatim}

Thus six shuffles are required to bring the cards back into their original sorted order. The output produced by your program should look similar to the table above. For output purposes, you may assume that the deck length \( n \) is less than or equal to 25, so that it fits on a single line. Shuffling a deck of size 20 would produce the following output:

Shuffles Deck Order
\begin{verbatim}
0   1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
1   11  1  12  2  13  3  14  4  15  5  16  6  17  7  18  8  19  9  20  10
2   16  1  11  6  1  17  12  7  2  18  13  8  3  19  14  9  4  20  15  10  5
3   8  16  3  11  19  6  14  1  19  17  4  12  20  7  15  2  10  18  5  13
4   4  8  12  16  20  3  7  11  15  19  2  6  10  14  18  1  5  9  13  17
5   2  4  6  8  10  12  14  16  18  20  1  3  5  7  9  11  13  15  17  19
6   1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
\end{verbatim}

\footnote{\textsuperscript{1} Thanks to David Helmbold}
Although there are many ways to shuffle cards, the purpose of this assignment is to implement a doubly linked list of integers ADT. Your program should do the following:

1. Read the number of cards from the command line.
2. Initialize a linked list (i.e. a deck) containing the \( n \) cards in sorted order, and print out the original order. This is the only time you should need to use the value \( n \).
3. Split this deck into two other decks. You can do this by alternately “dealing” from the top and bottom of the original deck, and inserting into the new decks as you go.
4. Combine these two decks by alternately “dealing” off the bottoms.
5. Print out the shuffle number and current deck state.
6. If the current deck is not in the original sorted order, continue shuffling.

Your List ADT will actually be a double ended queue with a current-position marker. Thus the set of “mathematical structures” for this ADT consists of all finite sequences of integers, where one integer may (or may not) be distinguished as the current element. This current element will be used by the client (namely your Shuffle program) to traverse the lists.

Your List module will support the following operations:

```java
// Constructors
List() // Creates an empty list
List(int data) // Creates a list containing exactly one int

// Access Methods
boolean isEmpty() // Returns true if List has no elements
boolean atFirst() // Returns true if first element is current
boolean atLast() // Returns true if last element is current
boolean offEnd() // Returns true if no element is current
int getFirst() // Returns the first element in List
int getLast() // Returns the last element in List
int getCurrent() // Returns the current element in List
int getLength() // Returns the length of the list
boolean equals(List list) // Returns true if this instance of List contains
// the same elements as the argument list

// Manipulation Procedures
void moveFirst() // Makes the first element current
void moveLast() // Makes the last element current
void moveNext() // Steps the current marker toward end of List
void movePrev() // Steps the current marker toward beginning of List
void insertBeforeFirst(int data) // Adds new element to beginning of List
void insertAfterLast(int data) // Adds new element to end of List
void insertBeforeCurrent(int data) // Inserts new element just before current
void insertAfterCurrent(int data) // Inserts new element just after current
void deleteFirst() // Deletes first element
void deleteLast() // Deletes last element
void deleteCurrent() // Deletes current element (Postcondition: offEnd() )

// Other Operations
List copy() // Returns a new List containing the same elements as this List
public String toString() // Overides Object’s toString method
public static void main(String[] args) // Test driver for the List class
```
The above operations are required for full credit, although it is not expected that all will be used by the client module (i.e. the Shuffle class).

The following operations are optional, and may come in handy in some subsequent assignment:

List concat(List list) // Returns a new List which is the concatenation of
    // this List and the argument List. This instance of
    // List and the argument List are unchanged in
    // the operation.
List cleanUp() // Deletes all repeated elements in this List, essentially
    // making it a set.
List union(List list) // Returns a new List which is the union of this List
    // and the argument List, considered as sets
List intersection(List list) // Returns a new List which is the set
    // intersection of this list and the argument
    // list

Your List class should also contain a private Node class which encapsulates one List element. This private class should contain fields containing a single int which is the data held at that Node, and references to the previous and next Nodes in the list. It should also define appropriate constructors, accessors, and manipulation procedures, including an equals method which compares the data values in two Nodes. Your List class should contain fields of type Node called first, last, and current which are references to the first, last, and current Nodes in the List.

Another option would be to design the Node class to contain a generic Object field rather than an int. This would change the signature of most of the above List methods, so that they either take or return an Object (not int). The purpose in this option is to allow Lists of anything, which may be useful in future assignments.

Your program will be structured in two files: a client module called Shuffle.java, and a List ADT module called List.java. Each file will contain one top level class, Shuffle and List respectively. Shuffle will represent decks of cards as reference variables of type List and use the above operations to perform shuffling operations. You must also submit a README file for this (and every) assignment describing the files created for the assignment, their purposes and relationships. Each file you turn in must begin with your name, Cats ID, and student ID.

Start early and ask questions if anything is unclear. It is helpful to write simple test programs to make sure you understand each part of the problem in isolation. The main method in your List class is required because it is much easier to debug your List ADT in isolation before you use it in the Shuffle class. You should first design and build your List ADT, test it thoroughly, and only then start coding your Shuffle class. Information on how to turn in your program will be posted on the webpage and discussed in class.