Implementation of data structures and algorithms
Short Project 2: Lists, Stacks, Queues

Version 1.0: Initial description (Thu, Aug 29).
Due: 11:59 PM, Sun, Sep 8 2019.

Submission procedure:
* Create a folder whose name is your netid (NId).
* Place all files you are submitting in that folder.
* Use "package NId;" in all your java files.
* There is no need to submit binary files created by your IDE (such as class files).
* Include a text file named "readme.txt", that explains how to compile and run the code.
* Zip the contents into a single zip or rar file.
* If the zip file is bigger than 1 MB, you have included unnecessary files.
* Delete them and create the zip file again.
* Upload the zip or rar file on elearning.
* Submission can be revised before the deadline.
* The final submission before the deadline will be graded.
* Only one member of each team needs to submit project.
* Include the names of all team members in ALL files.

Team task:
1. Implement a bounded-sized queue BoundedQueue<T>, using arrays with the following operations:
(To avoid "generic array cannot be created" error, declare the array to be Object[] and typecast where needed to avoid type warnings.)

BoundedQueue(int size):	Constructor for queue of given size
boolean offer(T x):	add a new element x at the rear of the queue returns false if the element was not added because the queue is full
T poll():	remove and return the element at the front of the queue return null if the queue is empty
T peek():	return front element, without removing it (null if queue is empty)
int size():	return the number of elements in the queue
boolean isEmpty():	check if the queue is empty
void clear():	clear the queue (size=0)
void toArray(T[] a):	fill user supplied array with the elements of the queue, in queue order

Additional tasks (no need to submit):

2. Given two linked lists implementing sorted sets, write functions for
union, intersection, and set difference of the sets.

```
 public static<T extends Comparable<? super T>>
    void intersect(List<T> l1, List<T> l2, List<T> outList) {
            // Return elements common to l1 and l2, in sorted order.
            // outList is an empty list created by the calling
            // program and passed as a parameter.
            // Function should be efficient whether the List is
```

```
                // implemented using ArrayList or LinkedList.
                // Do not use HashSet/Map or TreeSet/Map or other complex
                // data structures.
            }

    public static<T extends Comparable<? super T>>
        void union(List<T> l1, List<T> l2, List<T> outList) {
                // Return the union of l1 and l2, in sorted order.
                // Output is a set, so it should have no duplicates.
            }

    public static<T extends Comparable<? super T>>
        void difference(List<T> l1, List<T> l2, List<T> outList) {
                // Return l1 - l2 (i.e, items in l1 that are not in l2), in sorted order.
                // Output is a set, so it should have no duplicates.
            }
```

4. Extend the "unzip" method in starter code for SP1 to "multiUnzip" method in the SinglyLinkedList class:

```
void multiUnzip(int k) {
            // Rearrange elements of a singly linked list by chaining
            // together elements that are k apart.  k=2 is the unzip
            // function in SP 1.  If the list has elements
            // 1..10 in order, after multiUnzip(3), the elements will be
            // rearranged as: 1 4 7 10 2 5 8 3 6 9.  Instead if we call
            // multiUnzip(4), the list 1..10 will become 1 5 9 2 6 10 3 7 4 8.
}
```

5. Write recursive functions for the following tasks:
   (i) reverse the order of elements of the SinglyLinkedList class,
   (ii) print the elements of the SinglyLinkedList class, in reverse order.
   Write the code and annotate it with proper loop invariants.
   Running time: O(n).

6. Implement array-based, bounded-sized stacks. Array size is specified in the constructor and is fixed. When the stack gets full, push(x) operation should return false (like Q1), and an empty stack returns null on pop().

7. Implement the Shunting Yard algorithm:
   https://en.wikipedia.org/wiki/Shunting-yard_algorithm
   for parsing arithmetic expressions using the following precedence rules (highest to the lowest).

   * Parenthesized expressions (...)
   * Unary operator: factorial (!)
   * Exponentiation (^), right associative.

* Product (*), division (/). These operators are left associative.
* Sum (+), and difference (-). These operators are left associative.

Output the equivalent expression in postfix.