**Assignment: Stack Class**

**The Class**

Using your preferred implementation (e.g., Array, Simple Linked List, or Linked Arrays), write a Stack class and a short test program using that class. Recall that a Stack is a LIFO (Last In, First Out) object supporting methods such as Push, Pop, Peek, isEmpty, and a constructor. In this implementation, use an Integer object (as opposed to a primitive integer) as the entity being stacked; that is, the user is pushing and popping Integer objects representing primitive integers, not integer primitives themselves nor some node object used only internal to your Stack class.

**The Methods**

**Push** … Takes an Integer object as an input parameter and inserts it on the stack as the Top-Of-Stack.

**Pop** … Returns an Integer object found at the Top-Of-Stack. If the Stack is empty, Pop returns NULL.

**Peek** … Returns a reference to the Integer object at the Top-Of-Stack. If the Stack is empty, Peek returns NULL.

**Constructor** … Constructs the Stack object. As noted below, the maximum number of items ever held in the Stack will here be 100.

**IsEmpty** ... Return TRUE when the Stack is empty, FALSE otherwise.

**IsFull** … Returns TRUE when the Stack is full and can’t support any subsequent Pushes. It is implemented if an array-based implementation is chosen. It will be used prior to each Push to ensure that there are fewer than 100 items in the Stack.

**Size** … Returns the number of items currently in the Stack.

**The Test Program (the “User”)**

The test program will generate an ever increasing (i.e. incrementing by 1) set of positive integer values. Each to be constructed in an Integer object created by the test program for subsequent **Push**ing onto the Stack.

The test program will loop, with each iteration **Push**ing an ordered set of these Integer objects onto the Stack (e.g., 5,6,7, then 8,9,10,11, then 12,13, 14 and so on). The size of each set used in each iteration is a random number between 1 and 10, inclusive. Again, each set of Integer objects will consist of an ordered set of unique and increasing integer values.

With the test case – and, of course, the Stack object - keeping track of the number of items in the Stack (i.e., ongoing count of **Push**es minus **Pop**s), **Pop** a random number of items less than or equal to the number of items on the Stack.

* If the number of items **Pop**ped in an iteration is less than or equal to the number **Push**ed in the previous iteration, test that the values being **Pop**ped are as expected. (You know what to expect since the first value **Pop**ped in an iteration should equal to last value **Push**ed and they had been **Push**ed in increasing order.)
* If the number **Pop**ped exceeds the number immediately previously **Push**ed, test only the values of the number of Stacked items previously **Push**ed. (Again, you know these values.) After each Pop-And-Test, the Popped Integer object can be discarded.

Similarly, immediately prior to each **Pop** series, use **Peek** to access the first item on the Stack and verify that the first value **Pop**ped found the same integer value.

Further, after each Pop series, execute the isEmpty method, verifying that it returns FALSE when items are expected to be on the Stack and TRUE when the Stack count is expected to be zero. (This is the count expectation from the point of view of the user, the test case, not from that of the Stack object itself.)

Continue this process of Pushing and Popping for at least 20 iterations. Print number pushed and popped and comparison results with each iteration. However, unlikely, if during a set of Pushes, the number of items on the Stack were to exceed 100, Pop all of the items off of the Stack and start over. If desired, this can be done via a **makeEmpty** method.