CMSC 206: Data Structures
Lab 8: Stacks

During last class, we saw how a linked list can implement a stack. In this lab, you will use an `ArrayList` to implement a stack in two different ways and do performance measurements to see how design choices can impact performance.

Part I.

1. Make a new project for this lab. Copy the `SingleLinkedList`, `StackInt`, `KWArrayList`, `PostfixCalc`, and `LeftoverNumbersException` classes into the project. Recall that you might need to refresh your project (by right-clicking on the project folder and choosing *Refresh*) after adding the new files.

2. Rename *KWArrayList.java* to *KWArrayListEnd.java* (by right-clicking the file and then going into the *Refactor* submenu). Make a copy of this file named *KWArrayListStart.java*. (Explore the right-click menus to figure this out.)

3. Edit the `KWArrayListEnd` class to implement `StackInt<E>`. This will cause `KWArrayListEnd` not to compile. Add overrides of the four abstract methods inherited from `StackInt`. These methods should implement a stack by adding and removing elements at the *end* of the `KWArrayListEnd`. (For example, the `push` method should call the one-parameter `add` method.) All four methods are very short.

4. Modify the `PostfixCalc` to use `KWArrayListEnd` instead of `SingleLinkedList`. (This involves a small change to line 47 only; nothing else needs to be touched.) Confirm that it still works by testing your calculator with several postfix expressions.

5. Edit the `KWArrayListStart` class to implement `StackInt<E>`. Override the four inherited abstract methods by adding and removing elements at the *start* of the `KWArrayListStart`.

6. Modify the `PostfixCalc` to use `KWArrayListStart`. Confirm that the calculator still works.

Part II.

Now, we will collect performance information by measuring how long certain tasks take. First, we need to write the tasks.

7. Make a new class `PerformanceTests`. Write a method according to the following description:

```
/** Tests how long it takes to push then pop elements from a stack.
 *  Pushes 'num' elements onto the stack, then pops 'num' elements from
 *  the stack.
 *  @param st The stack to push and pop
 *  @param num The number of pushes and pops
 *  @return The number of nanoseconds taken to perform the pushes and
 *          pops
 */
public static long pushThenPop(StackInt<Integer> st, int num)
```

To write this method, you will need a way of measuring the passage of time. Using the `System.nanoTime` method works well. (`System` is in the `java.lang` package.) You will have to subtract two times to get the amount of time elapsed.

8. Write a `main` method that creates a stack and calls `pushThenPop` with different lengths, ranging from 1,000 to 50,000 in increments of 1,000, printing out the amount of time taken each time. (It may be easier to understand the output in milliseconds instead of nanoseconds. How would you do this conversion?) Try each stack implementation (you have 3 of them); how does the output differ? Write some comments in the file recording your observations. Note that changing out the stack implementation should require a change in only one place in your code.

9. Write a `pushPopInterleaved` method that's just like `pushThenPop`, but it pushes an element and immediately pops it (instead of pushing all the elements and then popping all the elements). Do performance tests with this method. Write down any observations in a comment.

10. Submit on Gradescope.