CS 1331: Introduction to Object Oriented Programming

Homework 10 **Due:** Friday April 18, 2014 @ 8:00 PM

Your Last Assignment!

I promised myself I wouldn't cry... *sniff*

This assignment comes in two parts. In the first part you will implement a stack, and get a first taste of abstract data types. In the second, you will implement a few bite-size sorting algorithms. Start early! There's plenty here to work on.

Stackin' It Up

1 All The Things

From time to time in Java we need to keep track of a large number of things at once. We've already seen some means of solving this problem... arrays, perhaps? In this assignment we will develop a slightly more sophisticated way of holding a variety of things.

In particular, in this assignment we will be holding things using a collection of references to things. Arrays contain references to things, but in a very particular way. Array references are stored in a neat little row, all side-by-side. You know how many references there are. You can't have more references than you have spaces in the array. It's all very strict.

Arrays are very useful, but they have a significant limitation. Arrays have a fixed size, and you can't change it after the fact. If you want to continue adding things to an array, you will have to make a new larger one, and then copy all of your references into the new array.

In this assignment we'll be taking a very different approach. Rather than resizing arrays when they become overfull, we'll organize our references an entirely new way. The structure that we'll create will never need to be resized, because it will have no pre-defined size. The structure we'll create will be dynamic.

2 Stacks

We will be implementing a Stack data structure in this assignment. Stacks are a very basic data structure, and the first of many abstract data types for you to learn.

There are only a few things that you can do to a stack: you can put things on top and take things off the top.

- `push` is the name given to putting something on top of the stack. You can only push something onto the very top of the stack, so that it covers up everything else in the stack. The other elements are not overwritten; they are still stored in the structure, they are just farther in.
- `pop` is taking something off the stack. This has the effect of both giving you a reference to the thing on top of the stack and revealing the other stack elements that were underneath it.

What happens when you attempt to `pop` the top off of a stack, but there isn't anything there to pop off? What should a stack do when this occurs?

Someone who attempts to `pop` off of an empty stack is clearly in error. But without some means of checking whether the stack is empty, they can hardly be blamed for it. Let's take away their excuse:

- `isEmpty` is a question we can ask a stack to see whether it contains any elements. A user should check whether the stack `isEmpty` before they attempt to `pop` anything from it.

3 A Stack As a Linked Structure

In this assignment you will be required to write a Java class which behaves like the stack abstract data type. Your implementation must be a linked structure, which holds elements in a sequence of node objects. Since nodes are structures internal to the stack we are building, we will not need access to them outside of our stack class. For this reason, we recommend you implement your nodes as a private inner class.

Linked structures are formed as a network of nodes, connected together by references. The nodes we will use are very simple, and will contain only two pieces of information as instance data:

- a single element of our stack, and
- a reference to another node, considered to be this node's successor.

Nodes are just a way for us to organize the association between these two pieces of information. A node serves only as a container. The real magic happens when you string several nodes together, by hooking up each node's reference to point to another node.

Nodes can be strung together in a chain, by setting the reference of each node to point to the next node in the sequence. If we'd like to find a particular node in the chain, all we need to do is follow the references of successive nodes until we find the node we're interested in. So to keep track of all the nodes that are linked together, all we need to keep track of is the first node in the sequence! There is also no pre-determined maximum size to our

chain, too. We can add more length to the chain by hooking up more nodes, ad infinitum (or until hardware constraints demand otherwise).

Most of the details of how this is done will be left to you. Some questions/suggestions for you to consider:

- Do I have to create a node object every time I push something onto the stack?
- If I want to add an element to our stack, where should we add it? To the beginning or the end of the chain?
- What should the last node's reference point to? What thing can I store in a variable of type Node, but isn't a Node?
- If we only interact with one end of the chain, would that improve the efficiency of our implementation? (hint: yes)
- How can I remove a node from the chain without losing access to the rest of the chain? Do I need to save a reference to the rest of the chain before I remove the first element of it?
- Will visualizing the chain of references be easier if I draw a diagram for myself?
- Should I write a simple main method that tests my stack implementation? (hint: absolutely)
- How can I make my stack implementation work for whatever type of thing I want to store?

4 Generics And Exceptions

I lied. That last question isn't for you to consider. I still need to explain that to you. Because stacks are widely useful in all sorts of different situations in computing, we'll make our stack class *generic*, so that it can hold whatever reference type we want it to. The first step to making our class generic is by adding a type parameter to the class declaration:

```
public class MyStack<E> {
```

The type parameter `E` stands for whatever type of thing our stack is holding right now. Wehen we create a `MyStack`, we must specify the value this type parameter should take on, a bit like if we were passing that type to a method:

```
MyStack<String> stack = new MyStack<String>();
```

Since people using our class have to provide the type that `E` should take on, we do not have to worry about what type it is while defining stack. Between `public class MyStack<E> {` and the corresponding closing brace, the type parameter `E` refers to whatever type `MyStack` is meant to hold. If we want to declare a variable of that type, we simply say:

```
E anExampleVariable = aFunctionDeclaredToHaveReturnTypeE();
```

In addition to using generics for this assignment, we will also require you to do some basic work with `Exceptions`. If the user of our stack class attempts to pop from the stack when the stack is empty, we should throw an exception. Conveniently, there is an `Exception` class in the Java API specifically intended for this purpose, called `EmptyStackException`. You can import this class and throw it very easily.

As you may recall, however, there are two large branches of exceptions: checked and unchecked. `EmptyStackException` is one of these two. You will be required to throw the opposite type of `Exception` as `EmptyStackException`. That is, if `EmptyStackException` is a checked `Exception`, then you should throw an unchecked `Exception`, and vice versa. You will be penalized if you throw the wrong type of `Exception`! Once you deduce what variety of `Exception` `EmptyStackException` is, you should define your own `Exception` of the opposite type.

5 Stacks!

A quick run-down of what we expect regarding your stack:

- An instantiable class `MyStack`, which supports the `push`, `pop`, and `isEmpty` operations.
- `MyStack` should be a linked structure, using a simple private inner class for nodes.
- The `pop` operation should throw an `Exception` that you have defined, and that `Exception` should be checked or unchecked according to whether or not `EmptyStackException` is.

Sorting Algorithms

For this assignment you will be required to implement several sorting algorithms discussed in lecture. Each algorithm will be contained in a separate subclass of the provided `AbstractSorter` class, overriding the `sortArray()` method in that class. We will give a brief description of each algorithm, and some pseudocode.

It is important to note that each of the algorithms here are memebers of a broad category of comparison-based sorts. Correspondingly, the elements of the array must be `Comparable`. When an algorithm's pseudocode compares two elements of the array (seeing if one is larger than the other, for example), you should adapt this by calling a certain method on the `Comparable` elements of the array.

1 Bubble Sort

This is one of the simplest and most naive approaches to sorting an array. This method repeatedly sweeps over the array, comparing pairs. If it notices any two adjacent array elements that are out of order, it immediately corrects their relative positions by swapping them. It sweeps repeatedly until it finds no adjacent elements that are out of order, which only happens when the array is fully sorted.

```
swapped <- true
while swapped
    swapped <- false
    for i from 1 to length(A) - 1
        if A[i - 1] > A[i]
            swap A[i] and A[i - 1]
            swapped <- true
```

2 Insertion Sort

This algorithm proceeds by collecting a sorted subarray in the earlier indices, and inserting successive elements to this subarray. The elements are inserted into the first place from the left that an element smaller than it is found, which does put it in order relative to the rest of the sorted subarray.

```
for i from 1 to length(A) - 1
    j <- i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j <- j - 1
```

3 Selection Sort

This algorithm also builds a sorted subarray in the earlier indices, but it builds this subarray as the sorted subarray of the smallest elements of the array. It repeatedly finds the minimum element of the unsorted portion of the array, and then swaps that element with the element just after the sorted portion of the array.

```
for i from 0 to length(A) - 1
    minIndex <- findIndexOfMinimum(i, length(A) - 1)
    if minIndex is not i
        swap A[minIndex] and A[i]
```

The trick to this one is in finding the index of the minimum element. This can be done with a simple `for` loop over the specified indices, keeping track of the smallest array element seen so far.

4 Check Yourself Before You Wreck Yourself

You should test each of your `Sorter` classes! This should be self-evident to you by now. Do not wait until the last minute to start testing your code! As soon as you've finished writing each `Sorter`, you should write a `public static void main(String[] args)` to make sure it works how you think it does. We do not require tests in your final submission (so if you'd like to write your tests in a separate class, then feel free), but you should not take that as permission to submit untested code.

5 Extra Credit: Merge Sort

Merge sort is a bit more complicated than the other algorithms we've just discussed. It is worth the effort, however. You can implement merge sort for this assignment for optional extra credit.

We won't give pseudocode for this particular algorithm. There are several variants of merge sort (some of which are more complicated than others), all of which can be found easily online. Depending on how well you implement the algorithm, we will award up to 20 points of extra credit for implementing merge sort.

To give you a sense of what we will consider "well implemented": if you implement an in-place version of merge-sort (which is one of the harder variants) then you are fairly likely to be given the full 20 points. Well-written code for an easier variant can still receive the full 20 points.

Deliverables

1. A generic stack class, implemented using linked nodes. For ease of grading it should be called `MyStack.java`.
2. An `Exception` that your stack throws when the user pops from an empty `MyStack`. It should be the opposite type of `EmptyStackException`.
3. `BubbleSorter.java`
4. `InsertionSorter.java`
5. `SelectionSorter.java`
6. **Extra credit:** `MergeSorter.java`
7. Please also submit the `AbstractSorter.java` source file so that your submission compiles and runs out-of-the-box.

Javadocs

We are going to have you do Javadocs again for this assignment, as always. The samples from the last HW writeup are below for your reference.

```
import java.util.Scanner;

/**
* This class represents a Dog object.
* @author Ethan Shernan
* @version 1.0
*/
public class Dog {
....
}

/**
* This method takes in two ints and returns their sum
* @param a, b
* @return their sum
*/
public int add(int a, int b)){
...
}
```

Checkstyle

Just in case you forget how to do Checkstyle, here are the instructions again!

You have been provided with a Checkstyle Jar file and the CS 1331 Checkstyle configuration file (`cs1331-checkstyle.xml`). We do this to make sure you are following the proper style guidelines as posted in the Style Guide under T-Square resources. Please refer to these guidelines for any help as you are writing your programs.

We will run checkstyle on all the Java source files you submit. You can run Checkstyle with the command below (in the directory containing all your Java source files):

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. You can easily count Checkstyle errors by using the command below and subtracting 2 from the number printed (which is how we will deduct points). For example:

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java | wc -l
2
```

Also means no errors.

If you have any questions about a Checkstyle error, please let a TA know! They should be pretty self-explanatory and you should be able to fix them easily. We will be taking points off for checkstyle on this assignment, so please make sure you run it!

Turn-in Procedure

Submit all of the Java source files you created to T-Square. Do not submit any compiled bytecode (`.class` files). When you're ready, double-check that you have submitted and not just saved a draft.

Please remember to run your code with Checkstyle!

Verify the Success of Your Submission to T-Square
Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.
2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.
3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.
4. Recompile and test those exact files.
5. This helps guard against a few things.
   (a) It helps insure that you turn in the correct files.
   (b) It helps you realize if you omit a file or files.[1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)

[1] Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is 8PM Friday. Do not wait until the last minute!

(c) Helps find last minute causes of files not compiling and/or running.