CS 455 Lab 4: Array Lists

Fall 2020 [Bono]

Goals and Background

In this lab you're going to get practice programming with array lists by implementing a class, `Nums`, that encapsulates one (i.e., there's an `ArrayList` inside every `Nums` object) and that has a few methods to examine and manipulate it. The resulting class isn't that useful as is, but part of the point of this lab is to also practice implementing classes using incremental development, testing each subset as you create it.

Note: in this lab where we say implement so-and-so method, it means to implement it so its functionality matches the method comment already written for it (i.e., you will have to read the method comment to know exactly what the method is supposed to do), and it would also mean you are not changing the method header.

Remember to switch roles with your partner from what you were last time. For more information on lab partnerships, see Lab 2.

Reading and reference material

- Horstmann, Sections 7.1.4, 7.3, 7.7 on arrays and array lists
- Horstmann, Section 5.6 on test cases
- Horstmann, Section Special Topic 4.4 on static methods; for more about how we are using static methods here see below
- Horstmann, Appendix D: The Java Library.
  An easier-to-understand, abbreviated version of Java API Documentation. This lab uses: `java.util.ArrayList`, `java.util.Scanner`, and `java.lang.Integer`,

Exercise 1 (1 checkoff point)

Step 1. The starter code available on Vocareum is two files, `Nums.java`, and `NumsTester.java`. There is also a small data file there called `nums.in`.

The class that encapsulates our array list is called `Nums`. It stores a sequence of ints and allows us to do some computations with the sequence. When we implement a non-trivial class, we want to write and test the minimal amount of code to make sure the data was put in it correctly, before implementing any other methods. That normally means implementing (1) at least one of the constructors and (2) an accessor method or print method that would allow us to see that it got created correctly.

Question 1.1 Why can doing this save us time in the long run, over implementing the whole class at once?

For the `Nums` class we're going to implement the constructor and the `printVals` method initially, although we will be doing more than that by the end of the exercise. Since the constructor just creates an empty `Nums` object, the results of this won't be too interesting. Read on through Step 3 before you start this task.

Read the code and comments in the following files. First, here's a summary of what they contain, and some changes you will be making to them:

- `Nums.java` contains the specification for the complete `Nums` class (i.e., method comments plus method headers). All the methods are currently empty, except for minimal code so the class can be compiled as is. These are called "stub" versions of the methods. Some of the stubs have non-empty bodies because they have a non-void return type, so require a return statement to compile. Those return statements don't necessarily return the correct value for the completed version of the methods: i.e., you will probably have to change them in the course of doing this lab.

- `NumsTester.java` contains most of the code for a class that tests the `Nums` class. In particular, you are going to add code to test the `add` method as part of exercise 1, but the code to test the rest of the methods is already present. Note: unlike the usual tester programs, this one is interactive, so we can work a little more with a `Scanner`.

Step 2. Go ahead and compile and run `NumsTester` so you can see what the output looks like with the stubs version of the `Nums` class.

Step 3. Implement the `Nums` constructor and `printVals` method. This, of course, also involves creating the instance variable(s) that are private to the class. You can run `NumsTester` with this code, to make sure an empty `Nums` object gets created and printed correctly. Look at the first few lines of `main`; that's where this code gets tested.

Step 4. Write the code necessary to to test building and printing a non-empty `Nums` object. To do that, you need to implement the `readNums` method in `NumsTester` (the test code) then the `Nums` `add` method itself (more about that process in the next few paragraphs).

More about the static method, `readNums` in `NumsTester`: (for more information about static methods, see the sidebar below). We're going to test our `Nums` class with input from a file (more about that below), so `readNums` should not print out any prompts for the user. We wrote a similar program, `ScoreCounts.java` in a recent lecture (available in the Vocareum Lecture Code area under `09-03`). We'll review what's involved here:

- In your code you can test if you've reached the end-of-file (eof) by using the `Scanner` method `hasNextInt` which returns a `boolean`. (Hint: you can find some documentation for selected Java classes in Appendix D of the text.) When running the program with input from the keyboard (i.e., not from a file), you can generate an end-of-file by typing newline then `Ctrl-d` after you have entered all the desired input.

- To test your code with data from a file (rather than from keyboard input), you can use Unix input redirection so that `System.in` is linked to a file instead of the keyboard. Look at the contents of the file `nums.in`. Run your program on it using the following command.

```
java NumsTester < nums.in
```

Note: running the program this way it will no longer wait for input from the keyboard.

Once you finish `readNums` and `add`, it will be the first time you'll be able to test that a non-empty `Nums` object gets created and printed correctly. Once you've completed the implementation your program should print out the list of numbers entered, in the exact format described in the `printVals` comments (that includes no space after the last number printed). Note: `NumsTester` will still not get correct results for the tests of the other two `Nums` methods: `minVal` and `valuesGT`, since we haven't implemented those methods yet.

Step 5. Create at least two other test files to test boundary cases of the code you have written. Call them `test1.in` and `test2.in`. (And, of course, fix your program as necessary if it doesn't work properly on the boundary cases.)

To get checked-off, show the TA the answer to question 1.1, your working `NumsTester` subset program, running on regular and boundary cases; also show him or her the source code you wrote for this exercise, and your test cases.

Exercise 2 (1 checkoff point)

Save your results from exercise 1 by creating a subdirectory `ex1` and making a copy of `Nums.java` and `NumsTester.java` there. (Hint: You can make the copy in one command using a wildcard.)

For Exercise 2 keep working with the the version of the files that's in your `work` (i.e., home) directory.

Read the comments for the `minVal` method of the `Nums` class so you know what it's supposed to do.

Question 2.1 Write down the expected results of calling `minVal` on each of the `Nums` objects that would result from the input in: `nums.in`, `test1.in`, and `test2.in`.

Complete the `minVal` method of the `Nums` class. Once you test your code, answer the question below.

Question 2.2 From just looking at the results of a call to `minVal`, we can't always tell whether the `Nums` object we called it on was empty or non-empty. Write down two examples of non-empty `Nums` objects such that they return the same value as for an empty `Nums` object.

To get checked-off, show the TA your working `minVal` code, including the source code; show it running on your various test cases from exercise 1, plus additional ones if necessary; and show him or her the answers to your questions.

Exercise 3 (1 checkoff point)

Back up what you have so far by making a copy of your source files in a new `ex2` directory (like you did at the start of the previous exercise). Continue working with the code in your `work`

directory this part of the lab.

Read the comments for `valuesGT` method of the `Nums` class so you know what it's supposed to do. Note: `valuesGT` is short for "values greater than".

Then look at the hard-coded test cases for `valuesGT` in `NumsTester`. The method is called from the `testFilter` method in `NumsTester`. (`testFilter` just packages up the computation and output for a particular test of `valuesGT`, so we don't have to keep repeating that code to do multiple tests.) These test cases are designed to go with the data in `nums.in`.

Question 3.1 Write down the expected results of the hard-coded tests of `valuesGT` from `NumsTester` when run on: `nums.in`, `test1.in`, and `test2.in`.

Question 3.2 Create an additional test case in a new file `test3.in` that explicitly tests a boundary case of `valuesGT` when run with on the hard-coded tests of `valuesGT` from `NumsTester`. Add the expected results of of this new test case to the list of expected results you gave in Question 3.1

Implement the `valuesGT` method of the `Nums` class. Check that your actual results match the expected results when you run it on each of the given test files. Remember, sometimes a mismatch is because of a mistake in the expected results.

To get checked-off, show the TA your working `valuesGT` code, including the source code; show it running on all four of the test input files plus additional ones if necessary; and show him or her the answers to the questions.

What are `static` methods? You've been introduced to static methods in context of calling them in the Java library, such as in the `Math` class (described in Special Topic 4.4 of the textbook), and we've been using `static` in our `main` headers even though we didn't really know why. In this lab we have a few more static method definitions, namely `readNums` and `testFilter` in `NumsTester`, so this is a good opportunity to look at this in more detail.

A better name for a static method would be a class method; it's is inside the scope of a particular class, but it is not associated with an individual instance of the class (i.e., object). As we've seen, from outside the enclosing class they are called with the class name instead of an object name (e.g., `Math.round(...)`). When we call a static (or non-static) method from another method in the same class, you don't have to use the dot notation (e.g., calling `testFilter(...)` from `main`). They are also analogous to regular functions in languages like C/C++, Python, and Matlab; in contrast, in Java all functions must appear inside classes.

The methods in `NumsTester` are also static because they are helper methods for a static method (namely, `main`). In Java, `main` is always static because it is called (by the run-time system) with no associated instance of its enclosing class (i.e., no `NumsTester` object, in this case). In virtually all the Java programs we've seen and will write all the variables used by `main` will be local variables (or the `args` parameter, which we'll

get to later in the semester). Generally static methods cannot access instance variables of their class, because there is no instance involved, and thus classes with only static methods, such as `NumsTester`, don't have instance variables at all. The two other methods in `NumsTester` are helper functions for `main` to reduce the complexity of `main` and the amount of code we have to write overall. Any data they use are passed via explicit parameters.

To summarize, to make a procedure-oriented design in Java you make a main class that has a static main method, plus helper methods that are static methods in that same class using parameter passing and return values to communicate information between these functions. Our test programs, including `NumsTester`, use such a design.

Exercise 4 (1 checkoff point)

In this exercise you are just going to answer some more questions about the code you wrote.

You added coded that used `hasNextInt()` as part of the `readNums` method you implemented as part of Exercise 1. `hasNextInt` will return false on more than just end-of-file.

Question 4.1 Give two examples of other possible things we could type in to cause this program to terminate (i.e., besides `Ctrl-d`). Try each of these on your program.

Look at the second line of code in the `testFilter` method in `NumsTester` which contains two "." (dot) operators:

```
nums.valuesGT(threshold).printVals();
```

Question 4.2 Write down an equivalent statement with more parentheses to show the grouping (i.e., associativity) of the "." operators, that is, to show what part of the expression gets evaluated first. Hint: if you think about it, only one grouping makes sense.

Question 4.3 What is the type of the innermost sub-expression you put parentheses around (i.e., the one that gets evaluated first)?

Question 4.4 Write a sequence of statements equivalent to the whole statement in question, but that uses a local variable to store intermediate results.

Our `Nums` class uses an array list. We could have used an array instead. (Note: Sections 7.7.4 and 7.7.8 of the text may be helpful in answering the the following two questions related to this.)

Question 4.5 What's the disadvantage of using an ArrayList compared to an array in the `Nums` class?

Question 4.6 What's the advantage of using an ArrayList compared to an array in the `Nums` class?

Question 4.7 Many programs involve arrays (or equivalently, array lists), but we don't always need to store all the values when we are processing a sequence of numbers. For each of the following tasks related to this lab say whether an array is necessary to complete the task, and briefly describe why or why not. We're interested in this because we could use a lot less memory if we don't have to save all the numbers.

To put it another way, do we need to save all the numbers to do the task? (why or why not)

1. Reading in a sequence of numbers and printing them all out.
2. Reading in a sequence of numbers and printing out the minimum value in the sequence.
3. Reading in a sequence of numbers, printing them out, and printing out the minimum value in the sequence.
4. Reading in a threshold value, a sequence of numbers, and printing out the numbers from the sequence that are above the threshold value.
5. Same task, as the previous one, but with the specification that you you have to read in the sequence of numbers before you read in the threshold.
6. Reading in a sequence of numbers, and generating the exact output produced by `NumsTester.java` about that data.

Checkoff for DEN students

Make sure you put your name and NetID in all the files you submit.

When you click the `Submit` button, it will be looking for and compiling (for source code) the files
`README`, `test1.in`, `test2.in`, `test3.in`, `Nums.java`, `NumsTester.java`, `ex1/Nums.java`, `ex1/NumsTester.java`, `ex2/Nums.java`, and `ex2/NumsTester.java`. As usual, the lab is due by 11:59pm on Sunday Pacific Time at the end of the week for this lab.
