EECS 233 Introduction to Algorithms and Data Structures
Fall 2014, Programming Assignment 1 ("P1")

Draft Due: Thu Sep 18 by midnight.
Final Due: Thu Sep 25 by midnight.
Total Points: 123 + 10 extra credit

You must submit your draft and final by the due dates above. General instructions and guidelines follow the assignment specification below.

Using feedback from P0, I'm making a few changes to simplify the submission and grading process. For P1, there will be just one assignment on Blackboard. You will submit your draft and final as different "attempts" (as blackboard calls them). We will always grade your latest submission. Follow the updated instructions at the end of the assignment.

In this assignment you implement two different abstract data types (ADTs) to store a list of numbers. You will then use one of these to implement a class to store a *set* of numbers.

1. NumList

Design and implement an abstract list data type `NumList` that can be used to represent an *ordered* sequence of double precision floating-point numbers. Provide two implementations of this ADT: a class `NumArrayList` and a class `NumLinkedList`. The first one must use an array to store the sequence and the second a singly linked list. The ADT should have the functionality described below.

- ▷ With no parameters, your constructors should each initialize an empty list. 2×2 P.
- For `NumArrayList`, a constructor with an integer argument should initialize an empty list with that capacity. 2 P.
- For `NumArrayList`, you should efficiently handle the case when there is no space to insert into the array by expanding its capacity. 5 P.
- ▷ `int size()` - return the size of the list, i.e. the number of elements, in the sequence. 2×2 P.
- For `NumArrayList`, `int capacity()` should return the capacity of the array, i.e. the number of elements it can currently hold (which is not the same as `size`). 2 P.
- ▷ `void add(double value)` - Add a new element to the end of the current sequence. 2×3 P.
- `void insert(int i, double value)` - Insert a new element before the i-th element of the sequence (using 0 for the index of the first element). For the case when the sequence has fewer than i elements, insert the new element at the end of the sequence. 2×5 P.
- `void remove(int i)` - Remove the i-th element of the sequence (using 0 for the index of the first element). For the case when the sequence has fewer than i elements, do nothing since the element to be removed does not exist already. 2×5 P.
- `boolean contains(double value)` - return `true` if the list contains `value`, `false` if it doesn't. 2×3 P.
- `double lookup(int i)` - Return the i-th element of the sequence (using 0 for the index of the first element). Raise an exception if the sequence has fewer than i+1 elements. 2×5 P.
- ▷ `boolean equals(NumList otherList)` - returns `true` if the `otherList` equals the other list. 2×3 P.
- `void removeDuplicates()` - remove duplicates of any elements in the list, preserving the list order and the position of the first of the duplicates. For example, removing the duplicates of the list [1, 2, 3, 1, 2, 4] should result in [1, 2, 3, 4]. 2×5 P.
- ▷ `String toString()` - convert the contents of the list to a string. For an empty list, it should return a null String, i.e. `""`. The elements should be formatted as they were provided, and should be separated by a single space and contain no other characters. 2×2 P.

2. NumSet

Write a class called `NumSet` that uses one of your the ADTs above to implement a set class. A set has no explicit order, but the items in a set are unique. It should have the following methods:

- Your constructor for `NumSet` should take a double array to define the set. 2 P.
- `size()` - return the number of items of the set. 2 P.
- `boolean contains(double value)` - returns `true` if the set contains `value`, `false` otherwise. 2 P.
- `NumSet intersect(S1, S2)` - return a new `NumSet` that is the intersection of sets S1 and S2. 5 P.
- `NumSet union(S1, S2)` - return a new `NumSet` that is the union of sets S1 and S2. 5 P.
- `String toString()` - convert the contents of the set to a string. The elements should be formatted as they were provided and should be separated by a single space. 3 P.
- `boolean equivalence(S1, S2)` - return `true` if S1 is equivalent to S2, i.e. if they contain the same items. (*extra credit*) +10 P.

3. Other grading considerations

We will also take into account the following when grading your assignment. Because many these considerations are subjective, one TA will be responsible for assigning the grades in each of these areas to help ensure consistency.

a) *Design* (5 P.) You should define an interface for the ADT operations, and provide two implementations. You do not need to use generics. Since this is a relatively simple assignment, there isn't that much to design. Design also covers your code efficiency and choice of implementation, e.g. how you chose to implement remove duplications.

b) *Style* (10 P.) Style considerations include:[1]

   - Variable name choices. Are they descriptive and do they convey information about their purpose? Note for simple concepts, it perfectly acceptable, even preferable to use, for example, `i` for an index or `n` for an integer.
   - Expressions and statements. Are they clear? Do they make the meaning as transparent as possible? Well-written code should not need much commenting, if any. Is it formatted in a way that aides readability?
   - Consistency and Idiom. Is formatting consistent? Are common operations consistent with programming idiom? (the textbook, for example)
   - Comments. Do they aid the reader in understanding the code? Comments that restate what is already clear the code are redundant and not helpful. Nor are comments that are not consistent with the code. For more complex methods, the comment should state the running time of the method (or running times if there are different cases).

c) *Testing* (10 P.) Your code should include JUnit tests to check that the code is correct and performs as specified. Your tests should be well thought-out with clear reasoning behind them, which you should explain in your comments. Having a large number of tests that are essentially equivalent will receive deductions. Remember that the whole goal of testing is to make the process of implementation and debugging faster and more efficient. So don't waste your time writing a bunch of unnecessary tests. Here are some guidelines from the *Practice of Programming*:

   - Test as you write the code. This goes along with writing incrementally. Once you implement a part of the assignment, write a test to check its correctness.
   - Test code at its boundaries. This is also referred to as testing "corner cases." Do conditions branch in the right way? Do loops execute the correct number of times? Does it work for an empty input? A single input? An exactly full array? Etc.

[1]The style and testing guidelines are from Chapters 1 and 6, respectively, of *The Practice of Programming* by Kernighan and Pike.

- Test systematically. This goes along with testing as you write the code. Have you checked that every line is correct? Do you have test code that executes every line and condition? Test the simple parts first, especially those parts you're going to reuse. You want to know that they're working when you use them. If you don't, debugging is much more difficult.
- Test automatically. Have test suite that automatically runs all the tests. JUnit helps provides this functionality.
- Stress test. Does your code work on large inputs? Random inputs? It's easy to make implicit assumptions that aren't always valid. Stress testing can help uncover these.

The same book also provides an amusing quote from the famous computer scientist and Case alum Don Knuth (the "father" of the analysis of algorithms among many other things) on how he writes test code:

> I get into the meanest, nastiest frame of mind that I can imagine, and write the nastiest [testing] code I can think of; then I turn around and embed that into even nastier constructions that are almost obscene.

Notes:

1. **Using built-in Java classes, such as ArrayList and LinkedList, is not allowed.** The point here is to write the classes from scratch.
2. Your ADT must implement the exact names specified above, including the case. We will test your code with our own test functions. We cannot make any modifications to your code for grading purposes.

General Instructions and Guidelines

You must submit your draft and final via blackboard by the due dates listed at the top of the assignment. Late assignments cannot be accepted, so start early and use the feedback from the draft to get help if you need it.

Turning in your code. You should submit a zipped folder of your java source and JUnit tests files via Blackboard. Follow these instructions:

- Make a separate submission folder with the following name: `EECS233_abc123_P1` where `abc123` is replaced by your own Case Network ID.
- The folder should contain your java source code, your JUnit tests, and any additional files (e.g. if you have a README.txt).
- Your JUnit filenames should have the form `ClassNameTest.java`.
- Do *not* include the java `.class` files. They take up too much space, and we will compile your files ourselves.
- Your submission folder should not contain subfolders.
- Compress your submission folder using the `.zip` format (do not use `.rar` or some other format)
- The compressed filename should be the same as the folder name, but with the `.zip` extension.
- Submit this file via Blackboard.

Draft. The draft assignment allows you to check your code for correctness against our own suite of JUnit tests. This will be the same test suite used for the final version.[2] In general, there will be set of tests for each required component. The draft also helps ensure that you get started early, which is vital for completing programming assignments on time. It is in your best interest that your draft be as complete possible. This will allow you to benefit from feedback from the full test suite and give you time to identify problems and fix them or get help.

- **The Draft is required**. The draft is not graded, but *it is required.* To receive a grade for the final version, you must turn in a draft that implements, at a minimum, the *fundamental* items.
- **Fundamental items**. Assignment items marked with "▷" are *fundamental*. These should be simple and straightforward to implement and represent the minimal functionality we need to test your code. We will provide you with our JUnit tests for these items, so that you can check the correctness of your code yourself. Needless to say, do not modify them, but if you find errors or potential issues, please bring them to our attention as soon as possible. You can use these JUnit tests as templates to write your own test suite for the rest of the assignment.
- **Use exact method names**. Use the exact method names specified, including the character case. This is required to check your code. Please be aware that we cannot make modifications to your code for grading purposes. Also, do not declare your source as a package. This isn't necessary and interferes with the test suite.
- **Complete skeleton methods are required**. In order for the JUnit tests to compile, you need to write skeletons for *all* the required methods. These can be empty, i.e. without any code, but they must have properly typed arguments and return values. If you do not this, we will not be able to test your code.
- **Ensure your code compiles**. You must ensure that your code is in a stable state and compiles by the draft due date. Our test suite script will automatically run after the deadline using the version you have labeled *draft*. We will only run the test suites once on your code, so do not ask for your it to be regraded. You should be writing your own JUnit test suites as you code to check correctness.

Version Control. You should always write your code incrementally, with a short write-test-debug cycle. Once you have a component written (which could be just a single method), write JUnit tests that check that your implementation is correct. Since we cannot grade what we cannot compile, you need to use some kind of version control to ensure that you have a saved working version once each step is completed. This could be as simple as copying the directory

[2]Although we reserve the right to correct errors or shortcomings in the test suites when we find them. If this happens, we will try to be fair, so that students are not led to have a false sense of completion. I will even consider awarding bonus points to those who identify useful test cases that we have overlooked.

or using a sophisticated version control system like "git". Anything is fine, as long as you always maintain a working copy of your code that you can turn in (or revert to if you introduce a bug that you can't track down).

Use an Automated Backup System. Everyone should be using a backup system that backs up your files automatically and seamlessly, such as Dropbox or Google Drive, among many others. They are free services for small space requirements, so there is no excuse anymore for loosing your work to a hard drive failures or accidentally deleted files. If you haven't done so already, set up a such a system now.

Final Version. Like the draft, make sure your code compiles and is in a stable state (i.e. your best working version) before the deadline. We will run the test suite on the version you have labeled *final* following the deadline.

General Strategies. It is helpful to work through the logic of the implementation for the problems in the assignment on a piece of paper, *before* you write any code. This can include things like data structure design, implementation, and small examples you use to work through the steps of your algorithms. You do not need to turn this in, but it can be helpful for grading in you include a README.txt file or a pdf scan of your notes that explains your design choices to the grader. Your code comments should describe clearly your high-level logic, design choices, and the running time and/or memory complexity of your algorithms.

Cheating and Plagiarism. It should go without saying that you should solve the problems and write the code on your own. You can consult the book and the internet for java questions, but do not look for solutions to the problems given in the assignments. You are encouraged to get help from your classmates regarding basic programming issues, debugging problems, features of Eclipse, etc., but like I mentioned in class, be careful about giving (or receiving) too much help. Not only might it spoil someone's learning efforts, it may appear as plagiarism. To ensure fairness, we will run software to check for cheating, so please do not copy code. Also keep in mind that your goal should be to develop the ability to think through problems, find solutions, and implement them in code. Some exam questions will be designed to test to what extent you have achieved this using examples drawn from the assignments.