CSE 373, Winter 2010
Programming Project #1: SoundBlaster (50 points)
Due Thursday, January 13, 2011, 10:00 PM

This assignment focuses on implementing a Stack ADT in two ways, with an array and a linked list. Turn in three files named `ArrayStack.java`, `LinkedStack.java`, and `README.txt` (this will contain your answers to the assignment questions as described below). Your Stack implementations should implement the `DoubleStack.java` interface. Your `LinkedStack.java` should use the `LinkedStackNode.java`. You will also need the `SoundBlaster.java`, `bot.wav`, `bot.dat`, and `secret.wav` files to test your Stack implementations. All of these files can be found on the course website.

This assignment involves using your Stack implementations to do sound manipulation, namely reversing a sound clip. This process, called "backmasking", was used by musicians including the Beatles, Jimi Hendrix, and Ozzy Ozbourne, although it seems to have fallen out of favor in recent years. You do not need to know how to handle music files. You will use the given `SoundBlaster.java` and your Stack implementations to reverse human-readable, text-based, sound files (i.e. the files that end in `.dat`). Once you have used `SoundBlaster.java` to produce a `.dat` file that contains the reversed sound clip, you can use the `sox` program to convert your `.dat` file into a `.wav` file that can be played on your computer by common media players (Windows Media Player, winamp, RealPlayer, etc.). We discuss how to use the `sox` program later in the specification.

Step 1: Eclipse

In addition to learning how to implement an ADT, one of the goals of this assignment is to give you an opportunity to gain experience working in Eclipse. Eclipse is a very powerful environment for Java so we encourage you to try working on your project in Eclipse (http://www.eclipse.org/). Eclipse is so powerful that it may seem like overkill for this assignment, but as the projects get larger, having an integrated development environment with lots of features will come in handy, so you should consider trying it out now.

You can use Eclipse in the lab, or download it to your personal machine. The download site (http://www.eclipse.org/downloads/) offers a number of different versions; you'll want "Eclipse IDE for Java Developers" (NOT the similar sounding "Eclipse IDE for Jave EE Developers").

After finding a way to run Eclipse, check out an Eclipse tutorial to ease you into the environment (e.g., http://www.cs.washington.edu/education/courses/cse143/11wi/eclipse-tutorial/). There are many more off of the course website and on the web.

We encourage you to write and run a simple "Hello World" program using Eclipse before starting on the project. This is just a program that does nothing more than print out the string "Hello World". Once you have mastered "Hello World", you might try something with multiple files, or re-doing one of your projects from cse143.

Here are some starter instructions for opening the project in Eclipse.

1. Download the file `373_11wi_p1_soundblaster.zip` from the course website to your desktop. This is an archive of an Eclipse project to get you started.
2. Open Eclipse.
3. If this is the first time you've run Eclipse, it will ask you to choose a location for a workspace. The default location is probably fine.
4. Go to File →Import. Choose General → Existing Projects Into Workspace. Choose "Select archive file" and find the zip file you just downloaded. You should see the project appear with a check box next to it. Press Finish.

5. There should now be a SoundBlaster project in the left-hand window. There should be little red Xs on some of the files and folders – this means that the code has errors in it. As you edit the code, Eclipse will automatically rebuild and tell you if you've made an error.
6. Find `SoundBlaster.java` under the src → (default package) area of the project and double-click. The file pops up. Now you can scroll down and hover on little light bulb in the left margin to see what the errors in the code are.
7. It appears that we don't have classes named `LinkedStack` or `ArrayStack` yet - which makes sense, as those are the files you need to write for your assignment. Here's where Eclipse gets really useful. Click on the light bulb – Eclipse pops up a list of possible remedies.
8. We want to create a new Java class in a file, so choose the appropriate option. A dialog window pops up with various options – click Finish.
9. Eclipse auto-generates the file `LinkedStack.java` (or `ArrayStack.java`, depending which light bulb you clicked) with stub methods for all of the methods in the `DoubleStack` interface.
10. You can now continue on with Step 2.

Step 2: Implementing the Stack ADT

The assignment asks you to write two implementations of a Stack. Both implementations should implement the `DoubleStack` interface. The `DoubleStack` interface contains the following methods:

- `public void clear()`
  This methods removes/deletes all elements from the Stack.
- `public boolean isEmpty()`
  This method should return `true` when the Stack contains no elements, or `false` otherwise.
- `public double peek()`
  This method should return the double value that is on the top of the Stack without deleting it from the Stack. If this method is called on an empty Stack, an `EmptyStackException` should be thrown.
- `public double pop()`
  This method deletes the double value that is on the top of the Stack and returns it. If this method is called on an empty Stack, an `EmptyStackException` should be thrown.
- `public void push(double value)`
  This method inserts a double at the top of the Stack.
- `public String toString()`
  This method returns a `String` representation of the Stack. The `String` should begin with a `[`, end with a `]`, and each element should be separated by a comma and appear in *reverse order* from which they were pushed onto the Stack. For example, if we have a `DoubleStack` called `stack`, and the following method calls were made: `stack.push(1.12121212)`, `stack.push(2.34343434)`, and `stack.push(3.56565656)`, a subsequent call to `toString` should return `"[3.565657, 2.343434, 1.121212]"`. In other words, the values in the String start at the top of the Stack and end at the bottom of the Stack. If the Stack is empty, `"[]"` should be returned. Additionally, the values in the `String` should be rounded to six digits past the decimal. To do this, use `Math.round` as described below. You may **NOT** use `Arrays.toString` to write this method.

To get your `toString` to contain the double values stored in the Stack rounded to six digits past the decimal, use `Math.round` as follows:

```
double d1 = 3.141592121;
d1 = Math.round(d1 * 1000000.0) / 1000000.0;
double d2 = 1.6666667;
```

```
d2 = Math.round(d2 * 1000000.0) / 1000000.0;
double d3 = 1.0;
d3 = Math.round(d3 * 1000000.0) / 1000000.0;
System.out.print("d1 = " + d1 + "; d2 = " + d2 + "; d3 = " + d3);
```

The preceding code produces the following output: `d1 = 3.141592; d2 = 1.666667; d3 = 1.0`

For the array-based implementation of the `DoubleStack` interface, your array implementation should start with a small array (say, 10 elements) and *resize* to use an array *twice as large* whenever the array becomes full, copying over the elements in the smaller array. You should do this by using the `Arrays.copyOf` method found in `java.util.Arrays`.

For the linked list-based implementation of the `DoubleStack` interface, you should use the instructor provided `LinkedStackNode.java` class to store the nodes in your `LinkedStackNode.java`.

The only Java classes that you may use to complete the implementations of your Stacks are `java.util.EmptyStackException` and `java.util.Arrays`. The only method that you can use in `java.util.Arrays` is the `copyOf` method to grow your `ArrayStack` when it becomes full.

Step 3: Testing Your Stacks

You can test your Stack implementations by running the `SoundBlaster.java` program. `SoundBlaster.java` reads in a `.dat` sound file that is specified at the console, puts the sound values on a stack, pops them off in reverse order, and puts these reversed values in a the second `.dat` sound file that is specified on the console. Additionally, the user must tell `SoundBlaster.java` whether they want to use the `ArrayStack` implementation or the `LinkedStack` implementation. `SoundBlaster.java` will consider any input entered beginning with an `'a'` as choosing the `ArrayStack` implementation, any input beginning with an `'l'` as choosing the `LinkedStack` implementation, and will continue prompting if you enter input beginning with any other character.

Below is an example log of execution from the program; user input is in bold.

```
Welcome to SoundBlaster!!
Input file (.dat) to reverse: bot.dat
Output file (.dat) to write backmask to: out.dat
ArrayStack or LinkedStack? ArrayStack
223410 samples in file
```

Note: in order for `SoundBlaster.java` to find your input file, your input file must be directly under your project directory. For example, my project is named `373_11wi_p1_soundblaster` and so there is a `373_11wi_p1_soundblaster` folder in my Eclipse workspace directory. The log of execution above works because `bot.dat` can be found directly under the `373_11wi_p1_soundblaster` folder.

Once you have run `SoundBlaster.java`, you can convert your output file (e.g., `out.dat` in the above execution) to a `.wav` file to play your reversed sound clip. See the Sox section of the specification below for more information about how to convert between a `.dat` file and a `.wav` file.

`SoundBlaster.java` is not a great testing program; it does not call all of the methods in your Stacks or may not call them in a very exhaustive way that tests all cases and combinations. Therefore we highly suggest creating another small client program of your own to help **test** other aspects of your Stack implementations behavior.

You also might consider creating some short `.dat` files by hand to aid testing (see the ".dat File Format" section of the write-up below for more information).

Step 4: Write-Up

In addition to the code that you turn in, answer the following questions in a file called `README.txt`.

1. How did you test your code?

2. The file `secret.wav` (found off of the course website) is a backwards recording of a word or short phrase from a famous movie. Use `sox` (or another converter) and your program to reverse it, and write that as the answer to this question.
3. Let's pretend that, instead of a `DoubleStack` interface, you were given a fully functional FIFO `Queue` class that provides the operations `enqueue`, `dequeue`, `isEmpty`, and `size`. How might you implement this project (i.e., simulate a `Stack`) with one or more instances of a FIFO `Queue`? Write pseudocode for your `push` and `pop` operations. Refer to the Written Homework Guidelines found off of the "Homework" page of the course website for instructions on writing pseudocode.
4. In the previous question, what trade-offs did you notice between a `Queue` implementation of a `Stack` and your original array-based implementation? Which implementation would you choose, and why?
5. What did you enjoy about this assignment? What did you hate? What could you have done better?
6. Any other information you would like to include.

Sox

The only sound file format you need to know about is the `.dat` format. You don't even have to know very much about that either, as we're giving you the code that reads and writes that format (i.e. `SoundBlaster.java`). In order to play the reversed sound clips produced by `SoundBlaster.java` and your Stack implementations, you need a way to convert the `.dat` file into a format that common media players (Windows Media Player, winamp, RealPlayer, etc.) understand. You will need to perform this conversion to answer one of the write-up questions. We'll describe one way to do it below; however, you're free to use any converter you can find.

`sox` is a UNIX command-line utility whose name stands for "SOund eXchange". It allows you to convert between many different sound formats including `.wav`, `.au`, etc. In particular, `sox` allows you to convert to and from `.dat` sound files. `.dat` files are useful because they are human-readable, text-based, sound files.

There is a windows version of `sox` available, and the source archive is known to compile and work on OS X 10.4. You can download versions from the project page at SourceForge (http://sox.sourceforge.net/). The windows version is also a command-line program and works in the same way as the UNIX version described below. See the course webpage for some hints on using it.

The general strategy for using `sox` is as follows.

1. Take a `.wav` sound file of your choosing (e.g. `secret.wav`). This sound shouldn't be longer than a couple seconds, or your program will run out of memory.
2. Convert it to a `.dat` file: `sox secret.wav secret.dat`
3. Manipulate it by running `SoundBlaster.java`.
4. Convert it back to a `.wav` file: `sox secret-revealed.dat secret-revealed.wav` (this assumes the output file you saved the reversed sound clip to in step 3 is `secret-revealed.dat)`
5. Listen to it! (Use your favorite sound player.)

`.dat` File Format

The `.dat` file format starts with one line describing the sample rate of the sound file. *This line is required.* The rest of the file is composed of two columns of numbers. The first column consists of the time (measured in seconds) when the sample was recorded, and the second column contains the value of the sample, between -1.0 and 1.0. This is the beginning of a sample `.dat` file. Notice that the numbers in the first column increase by 1/44100 each step. This is because the sample rate is 44.1kHz.

```
; Sample Rate 44100

0                    0
2.2675737e-05        0
4.5351474e-05        0
6.8027211e-05        0
9.0702948e-05        0
0.00011337868        0
0.00013605442        0
0.00015873016        0
0.00018140590        0
0.00020408163        0
```

Here is the same file, a little deeper on:

```
0.22714286          -0.0144958500
0.22716553          -0.0147705080
0.22718821          -0.0157012940
0.22721088          -0.0129547120
0.22723356          -0.0127105710
0.22725624          -0.0181579590
0.22727891          -0.0191497800
0.22730159          -0.0145721440
0.22732426          -0.0122375490
0.22734694          -0.0124359130
0.22736961          -0.0108184810
```

Note that for this assignment, you shouldn't have to deal much with the `.dat` file yourself, as the provided `SoundBlaster.java` reads and outputs these files for you. All you have to do is implement the stacks. We are explaining the format because it will be helpful for you if you want to write a short file by hand to run, to verify if your program works.

Style Guidelines and Grading

A large part of your grade will come from appropriately utilizing arrays and linked lists to implement your `ArrayStack.java` and `LinkedStack.java` classes. There will be significant deductions if you use Java collections to implement these classes. Redundancy is another major grading focus; some methods are similar in behavior or based off of each other's behavior. You should avoid repeated logic as much as possible. Your class may have other methods besides those specified, but any other methods you add should be `private`.

You should follow good general style guidelines such as: making fields `private` and avoiding unnecessary fields; appropriately using control structures like loops and `if/else`; properly using indentation, good variable names and types; and not having any lines of code longer than 100 characters.

Comment your code descriptively in your own words at the top of your class, each method, and on complex sections of your code. Comments should explain each method's behavior, parameters, return, and exceptions. The files provided to you use "doc comments" format used by Javadoc, but you do not have to do this. For reference, both our `ArrayStack.java` and `LinkedStack.java` classes are around 90 lines long including comments and blank lines.

Acknowledgements

Like many assignments, this has been passed down to us through the vaporous mists of time. Among all our fore-bearers, we would especially like to thank Ruth Anderson, Tyler Robison, Ashish Sabharwal, and Adrien Treuille and his data structures professor, Timothy Snyder.