**Homework 1 - Stacks**

**Assignment**

**Introduction**

The purpose of this project is to implement a Stack ADT in the two most common ways, an array and a linked list. You will implement stacks for Java double numbers.

Your Stack implementations will be used 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. [Here](http://en.wikipedia.org/wiki/Backmasking) is a history of this (sometimes controversial!) practice.

**The Program**

You will write a program that reads a sound file in the .dat format (explained below) and writes another .dat sound file which is the reverse of the first. We have provided a class Reverse whose main method reads in a .dat sound file, pushes all the sound values on a stack, and then pops them all off and writes them into a new .dat sound file. We have also provided an interface DStack, which defines a stack that holds double values. Your first job is to familiarize yourself with these files.

**Implementing the Stack ADT**

You need to provide two stack implementations, one using an array and one using a linked list. They should be called ArrayStack and ListStack, respectively. They should implement the DStack interface given to you. Once you provide these implementations, Reverse should work and create backward sound files. It should take no more than a page or two of code to provide the implementations. 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. While there are convenient Java library methods for copying arrays, for this assignment, use your own loop to copy array elements manually (so you can "see" the work involved in copying).

Both ArrayStack and ListStack should throw an EmptyStackException if pop() or peek() is called when the stack is empty. To use EmptyStackException, add the following line to your file:

import java.util.EmptyStackException;

The only Java class that you should use to complete the implementations of your stacks is java.util.EmptyStackException. You should also use the length field of an array.

**Running Reverse**

The Reverse program takes 3 arguments (also known as "command-line arguments"). The first is the word array or list and specifies which implementation to use. The next two are the input and output .dat file names (you need to include the .dat extension). Running the program will depend on your system; from a command line it will look something like:

java Reverse list in.dat out.dat

In an IDE there is usually a dialog box for setting program parameters which contains a field for the program arguments. (For example, in jGRASP select Build->Run Arguments and a bar will appear at the top of the screen that allows you to type in the arguments. See Working with Eclipse below for information on setting command line parameters in Eclipse.)

To test your program, you will need a .dat file, which you can create from a .wav file as explained in the Digital Sound section. It is also useful to create short .dat files by hand to aid testing.

**Testing Your Stack Implementations**

We have provided you a client program, Reverse.java, that uses the stack implementations you will write. Getting Reverse.java to run and correctly reverse a sound file is fun and indicates that your stack implementations will compile and run. It does not, however, imply that your stack implementations have been thoroughly tested. Reverse.java just uses your stacks in one particular way: pushing a bunch of elements onto the stack and then popping them all off. There are other ways that someone use your stack implementations and other cases that are not tested by just being able to successfully listen to sound files in reverse.

Your stack implementations will be tested more generally and you should too! It is highly suggested creating another small client program of your own to help test other aspects of your code. You also might consider creating some short (e.g., 10 lines) .dat files by hand to aid testing. (See the ".dat File Format" section below for more information).

**Homework Questions**

Submit a Microsoft Word document, answering the following questions:

1. How did you test that your stack implementations were correct?
2. The file secret.wav is a backwards recording of a word or short phrase. Use sox (or another converter) and your program to reverse it, and write that as the answer to this question.
3. Your array stacks start with a small array and double in size if they become full. For a .dat file with 1 million lines, how many times would this resizing occur? What about with 1 billion lines or 1 trillion lines (assuming the computer had enough memory)? Explain your answer.
4. Suppose that, instead of a DStack interface, you were given a fully-functional FIFO Queue class. How might you implement this project (i.e., simulate a Stack) with one or more instances of a FIFO Queue? Download QueueStack.java and FIFOQueue.java and complete the push and pop operation in the QueueStack.java. The FIFOQueue class provides the operations enqueue, dequeue, isEmpty, and size. Turn in a file QueueStack.java.
5. 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?
6. Include a description of how your project goes "above and beyond" the basic requirements (if it does).
7. What did you enjoy about this assignment? What did you not enjoy? What could you have done better?
8. What else, if anything, would you like to include related to this homework?

**Programming**

**From Blackboard**

Download these files from Blackboard into a new directory:

* Provided source code: Reverse.java, DStack,java
* Later, when you are ready to run your program and the sox program (see below), you will want these sound files: bot.wav, secret.wav
* Homework questions provided code: QueueStack.java, FIFOQueue.java

**Implementation**

For this assignment you need to implement an interface, DStack, in two ways. The DStack interface defines a simple stack:

public interface DStack

{

public boolean isEmpty( );

public void push(double d);

public double pop( );

public double peek( );

}

An actual interface includes comments, including a description of how pop( ) and peek( ) should behave if they are called when the stack is empty. To implement this interface, write a class as follows:

public class ArrayStack implements DStack

{

public ArrayStack( )

{

// Your constructor code

}

public boolean isEmpty( )

{

// Your isEmpty( ) code

}

public void push(double d)

{

// Your push( ) code

}

// continue with the rest of the methods,

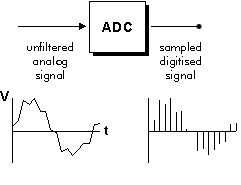
// along with any fields, etc.

}

The ListStack class should be defined similarly. You should include appropriate comments as needed. In particular, each file should begin with a comment that describes the class in the file, and includes your name and other identifying information.

**How Digital Sound Works**

We will view sound as a continuous function of time from the positive real numbers (time) to the interval [-1.0, 1.0] (amplitude). Since a computer can't "hold" a function defined on the reals, we have to approximate the function. We do this by measuring (or "*sampling* ") the sound several thousand times per second.



This process is called "Analog to Digital Conversion", or ADC. The number of times per second the sound is sampled is called the *sample rate* and is measured in *Hertz*. For example, CDs are recorded at 44100 samples per second, or 44.1kHz.

**Sox**

The only sound file format you need to know about is the .dat format described below. You don't even have to know very much about that either, as we are giving you the code that reads and writes that format. In order to play sounds you produce, you need a way to convert between the .dat format and a format that common media players (Windows Media Player, winamp, RealPlayer, etc.) understand, such as .wav. We will describe one way to do it below; however, you are free to use another converter.

sox is a command-line utility whose name stands for "SOund eXchange". It lets you 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. Note that you will need to perform this conversion to answer one of the questions.

Versions of sox are available for Windows, Mac, and Linux. You can download the one you need from [here](http://sourceforge.net/projects/sox/files/sox/) by clicking on the folder for the most recent version (or, really, any version) and then downloading the correct file for your operating system.

The Windows and Mac version are also a command-line program and work in the same way as the UNIX version described below.

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 few seconds, or your program will run out of memory.
2. Convert it to a .dat file:

sox secret.wav secret.dat

1. Manipulate it with the program you will write:

java Reverse list double secret.dat secret-revealed.dat

1. Convert it back to a .wav file:

sox secret-revealed.dat secret-revealed.wav

1. Listen to it with your favorite sound player.

That's all there is to it!

**The .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:

0.22693878 -0.0062561035

0.22696145 -0.0043945312

0.22698413 -0.0068664551

0.22700680 -0.0115661620

0.22702948 -0.0145568850

0.22705215 -0.0145416260

0.22707483 -0.0121917720

0.22709751 -0.0123901370

0.22712018 -0.0145416260

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 you shouldn't have to deal much with the .dat file yourself, as the provided Reverse.java does all the work for you. All you have to do is implement the stacks. We are explaining the format because it will be helpful for writing short files by hand for testing purposes.

**Working with Eclipse**

We encourage you to try working on your project in [Eclipse](http://www.eclipse.org/), a powerful environment for Java and several other languages. Eclipse may seem like overkill for this assignment, probably because it is! But as the projects get larger, having an integrated development environment with lots of features can come in handy, so you should consider trying it out now.

You can use Eclipse in the lab, or download it to your own computer. The [download site](http://www.eclipse.org/downloads/) offers a number of different versions: you want 'Eclipse IDE for Java Developers' (*not* the similar sounding 'Eclipse IDE for Java EE Developers').

First, check out an [Eclipse tutorial](http://eclipsetutorial.sourceforge.net/totalbeginner.html). There are several available on the web.

Next, 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 "Hello World". Once you have mastered "Hello World", you might try something with multiple files, or re-doing one of your assignments from CSC 151 or CSC 152.

Some starter instructions for **opening the project in Eclipse**:

1. We have packaged the project files as an Eclipse project: SoundBlaster.zip. Download that file from Blackboard to your desktop.
2. Open Eclipse.
3. If this is your first time running 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. On the next screen, choose 'Select archive file,' and browse to find the .zip file you just downloaded. Click open. You should see the Sound Blaster! project appear with a check box next to it. Press Finish.
5. There should now be a Sound Blaster! 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 re-compile and tell you if there are compiler errors.
6. Find Reverse.java in the project and double-click. The file pops up. Now you can scroll down and hover on the little light bulb in the left margin to see what the errors in the code are.
7. Looks like we do not have classes named ListStack or ArrayStack yet - which makes sense, as those are the files you need to write for your assignment. Here is where Eclipse gets really useful. Click on the lightbulb - 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 ListStack.java (or ArrayStack.java, depending which lightbulb you clicked) with stub methods for the methods in the DStack interface, which saves you some typing.
10. At this point, we'll leave you to explore Eclipse on your own. For this assignment, you won't really need any more of Eclipse's cool features - but feel free to explore!

Some starter instructions for **running your program from Eclipse**:

When you are ready to use your stack code with the Reverse.java program, you will need to do a bit of work to pass command-line arguments to the program:

1. Right-click on the Reverse.java file in the Sound Blaster! project on the left hand side of the screen. Go to Run As -> Run Configurations.
2. Select Java Application to show Reverse. Click on the Arguments tab.
3. In the Program arguments box, put three arguments, as specified above: the type of stack, the input file and the output file. For example:

array data\_files/bot.dat data\_files/out.dat

(That should all be on one line.) In this example, data\_files is a directory in your project folder where your input and output files are located. You may create a subdirectory called data\_files in your project folder and place your .dat files there or you may find it easiest to put your .dat input and output files directly in the same directory as your .class files so that all you need to give is the file names themselves.

1. Click Run. The program should run, with output appearing in the lower right window. If something did not work (e.g., could not find your input .dat file or you gave the wrong number of arguments), then you will get back a message. As you are debugging, remember that you can look at Reverse.java to help interpret error messages.
2. To run the program again with the same arguments, press the Run button in the toolbar. If you want to run the program again with different arguments, you need to go back to the run dialog to edit what you typed previously.
3. You'll need to run Sox on your output files separately (not in Eclipse).

When you have finished the project and want to turn in ArrayStack.java, ListStack.java, and (if you do not use an inner class) ListStackNode.java, an easy way to get the individual files is to click on each one in Eclipse's left-hand window and drag to the Desktop. This copies the file from the Eclipse workspace to your desktop.

**What to Submit**

You should submit the following files:

* ArrayStack.java
* ListStack.java
* ListStackNode.java, the linked-list node for use with your ListStack class. However, you are also free to use an [inner class](http://java.sun.com/docs/books/tutorial/java/javaOO/nested.html) inside ListStack in which case you will not have a separate file.
* A Microsoft Word document containing answers to the Homework Questions..
* QueueStack.java, containing answers to Homework Question 4.
* Any additional files for the extra credit **in a zip file named** extracredit.zip. Please make sure that this zip file decompresses its contents into a folder called extracredit and *not* into a bunch of individual files.
* Do **not** turn in DStack.java, Reverse.java or FIFOQueue.java. You must not change these files. Your stack implementations must work with the code as provided to you.

**Above and Beyond**

The following list of suggestions are meant for you to try if you finish the requirements early.

* Modify your array implementations so that when the array is 3/4 empty, the stack resizes to use an array of half the size.
* Implement the Karplus-Strong algorithm (to generate "reverb" sounds). A description of this algorithm can be found on Blackboard, Karplus-Strong Algorithm.docx.