CSE 332 Winter 2013: Project 1 - Sound Blaster!

Outline

Due Dates and Turn-In

You can turn it in using the catalyst Drop-box linked on our course web page.

- Phase A: Wednesday Jan 16th, 11pm
- Phase B: Tuesday Jan 22, 11pm

Be sure to read the turn-in info before you submit!

Preliminaries

- Complete this project by yourself (i.e., without a partner). You may discuss the assignment with others in the class, but your solution must be entirely your own work.
- Go through the Collaboration Policy, Grading Policy, and Programming Guidelines before you begin working on the project. In particular, note that the write-up is worth a substantial portion of the grade.
- Download these files into a new directory:
  - You'll need these files to start out: `Reverse.java`, `DStack.java`, `GStack.java`
- The code provided for you will read in a sound file, but it does so in `.dat` format. You'll use a program called `sox` to convert wav files to `.dat`: You can find information on it here and also on how to use it here.
- Later, when you are ready to run your program and the `sox` program, you will want these sound files: `bot.wav` `secret.wav`

Introduction

The purpose of this project is to implement a Stack ADT in the two most common ways, an array and a linked list. In phase A, you'll implement stacks for Java `double` numbers. Then in phase B, you'll implement generic stacks and instantiate them with type `Double`.

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. Click here for a history of this (sometimes controversial!) practice. "But wait," you say, "CSE 143 never taught me how to work with music..." Don't worry! All the music-handling parts have been done for you.

The Assignment

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 provide you with a class `Reverse` whose main method reads in a `.dat` sound file, pushes all the sound values on a stack, then pops them all off and writes them into a new `.dat` sound file. We've also provided you interfaces `DStack` (for Phase A) and `GStack` (for Phase B), which define the stacks you will implement. Your first job is to familiarize yourself with these files.

Even though the music-reversing code will use your stack implementations in only certain ways, you should test that your stacks are correct in all cases, not just those used by the reverse program.

Phase A

For phase A, 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 backwards 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. 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;
```

Note that your solution to Phase A does not require making changes to `Reverse.java`.

Phase B

For phase B, you need to provide *two more* stack implementations, these implementing the interface `GStack<T>`. Create classes called `GArrayStack` and `GListStack` that are just like `ArrayStack` and `ListStack` except that they are generic in the type of elements kept in the stack. The simplest approach is to *copy* `ArrayStack` and `ListStack` and then make appropriate changes. Normally such copy-and-paste is poor form, but here the pedagogical purpose is to show you how little you need to change to make the code generic — and we want to grade all four stack implementations.

To use your generic stacks you will need to make some additions to `Reverse.java` by replacing the 3 occurrences of:

```
System.err.println("no support for generics yet");
System.exit(1);
```

with appropriate code. Again, the code you write will be only slightly different from non-generic code that is already there. Do not make other changes to `Reverse.java`.

Running Your Program

The `Reverse` program takes 4 arguments (a.k.a. "command-line arguments"). The first is the word `array` or `list`, and specifies which implementation to use. The second is the word `double` or `generic`; the latter is for Phase B. The next two are the input and output `.dat` file names (you need to include the `.dat` extension). From the command-line, you can run the program with something like:

```
java Reverse list double in.dat out.dat
```

To run your program in Eclipse, create a "Run Configuration" and under the "Arguments" tab put the arguments (e.g., `list double in.dat out.dat`) in the "Program arguments" box. Here are some more tips about using Eclipse with Project 1.

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 may also be useful for you to create some short `.dat` files by hand to aid testing.

Write-Up Questions

Note that the write-up questions can all be turned in for Phase B, but some of them (1-7) refer to work you did in Phase A, so you may wish to start on them early.

Turn in a report answering the following questions

1. Who and what did you find helpful for this project?
2. How did you test that your stack implementations were correct?
3. 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.
4. Other than `java.util.EmptyStackException`, did you use any classes from the Java framework or other class library? (As indicated in the programming guidelines, you will get a low score on this project if you use a library to implement your stacks.)
5. 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.
6. Instead of a `DStack` interface, pretend 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`? Write pseudocode for your `push` and `pop` operations. Refer to the Written-Homework Guidelines for instructions on writing pseudocode. Assume your Queue class provides the operations `enqueue`, `dequeue`, `isEmpty`, and `size`.
7. Why would a stack implementation using a queue, as you described in the previous problem, be worse than your array and linked-list stack implementations?
8. In the process of making your generic stack implementations from your non-generic ones, what sort of errors did you encounter and how did you resolve them?
9. In hindsight, how much did you have to understand about the code in `Reverse.java` to make the changes to use your generic stacks?
10. Include a description of how your project goes "above and beyond" the basic requirements (if it does).
11. What did you enjoy about this assignment? What did you hate? What could you have done better?
12. Anything else you would like to include?

Going Above and Beyond

The following list of suggestions are meant for you to try only if you finish the requirements early. Please be sure that your stack implementations have been thoroughly tested and you have checked your writeup carefully before attempting any of these. Recall that any extra-credit you complete is noted and kept separate in the gradebook and *may* be used to adjust your grade at the end of the quarter, as detailed in the course grading policy. *Note: please turn in all extra credit files separately, as described under 'Turn-in'.*

- (least difficult) Modify your array implementations so that when the array is 3/4 empty, the stack resizes to use an array of half the size.
- (somewhat more difficult) Implement the Karplus-Strong algorithm (to generate "reverb" sounds). A description of this algorithm is available here.
- (most difficult) Assuming that your `.dat` input file contained a single note, try writing a program which will output a single-octave scale beginning at that note. Would the notes of an "evenly spaced" scale grow logarithmically, linearly, polynomially, or according to some other function? For more information about the evenly-spaced scale, refer to this article. To earn full credit, you must start from an actual recorded note (not a synthetic one) and generate the single-octave scale. It's fine if the note durations are decreasing.

Turn-in information

- **Each file you turn in should have your name at the beginning.** All text files should have your name on the first line; your name should appear in the comments at the beginning of each source code file you turn in.
- You **must** implement the list and array stacks by hand. You may not use any classes from the Java libraries to do the work. You should not use any import statements, except for `java.util.EmptyStackException`.
- For Phase A, turn in the following files, named as follows:
  - `ArrayStack.java`
  - `ListStack.java` While it is acceptable to submit a separate file for your node class, try using an inner

class.

- For Phase B, turn in the following files, named as follows:
    - `README.txt` Answers to the Write-Up Questions.
    - `Reverse.java`
    - `GArrayStack.java`
    - `GListStack.java`
    - Any corrections or improvements to files from Phase A. If you submit new versions of files from Phase A, then also submit a file called `changes.txt` describing what changes you made. This should be a concise description (as we can compare the files if we need details).
    - Any additonal 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` or `GStack.java`. You must not change these files. Your stack implementations must implement these interfaces as provided to you.

Sound and Sox

How digital sound works, and a link for a wav to dat conversion program: here. A few tips on using sox here.

Acknowledgments

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 Ashish Sabharwal, Adrien Treuille, and Adrien's Data Structures professor, Timothy Snyder.