CSC 325 Adv Data Structures

Assignment #2

Jump Tables

Directions and Sample Output:

For this assignment you are to make a data structure visualizer (similar to the ones I show in class if you've had me before). The idea is that you will visualize 3 data structures: stack, queue, and list. You are to use Java's built in Stack, Queue and ArrayList classes. Before I explain much more, here is the output you will be creating (user input is in **bold red**, arrows indicate what happens when that user input is submitted (by pressing enter), and green text and green horizontal line provide content and divide the drawings (and are not actual output)):

```
1. Stack
2. Queue
3. List
4. Quit
? 1
```

When launched the user is presented with a menu. Typing a "1" and hitting enter changes to display a stack (the contents of which is read from a file, which you are given)

```
|   |
|---|
| c |
|---|
| b |
|---|
| a |
|---|
1. Push
2. Pop
3. Save & Move to Queue
4. Save & Move to List
5. Quit
? 1 x
```

The user is then presented with options to modify the stack or save and move to another data structure. If "Push" is chosen, a space followed by a single character should be typed. This will be what is pushed onto the stack. The stack is then redrawn.

```
|   |
|---|
| x |
|---|
| c |
|---|
| b |
|---|
| a |
|---|
1. Push
2. Pop
3. Save & Move to Queue
4. Save & Move to List
5. Quit
? 1 y
```

Here the user is attempting to push 'y' onto the stack.

```
|   |
|---|
| y |
|---|
| x |
|---|
| c |
|---|
| b |
|---|
| a |
|---|
1. Push
2. Pop
3. Save & Move to Queue
4. Save & Move to List
5. Quit
? 2
```

Selecting "Pop" will pop from the stack and redraw. No other input from the user is required if they want to pop. Note that you should clear the screen after every action.

```
|   |
|---|
| x |
|---|
| c |
|---|
| b |
|---|
| a |
|---|
1. Push
2. Pop
3. Save & Move to Queue
4. Save & Move to List
5. Quit
? 3
```

Selecting one of the "Save & Move" options will save the stack contents back to the file. This means if the user re-launches the application or navigate back to the stack, the file will be read again and loaded into the stack.

```
| a | b | c |
1. Enqueue
2. Dequeue
3. Save & Move to Stack
4. Save & Move to List
5. Quit
? 1 !
```

Just like with the stack, when you select "Enqueue" you should give a character along with it. This may be any single character. The queue is then redrawn. Similarly, selecting 'Dequeue' will dequeue the front number.

```
| a | b | c | ! |
1. Enqueue
2. Dequeue
3. Save & Move to Stack
4. Save & Move to List
5. Quit
? 3
```

Selecting "Save & Move" saves the queue to the queue file so it can be retrieved later. Here we are jumping back to the stack.

```
|   |
|---|
| x |
|---|
| c |
|---|
| b |
|---|
| a |
|---|
1. Push
2. Pop
3. Save & Move to Queue
4. Save & Move to List
5. Quit
? 4
```

As you can see, our stack was preserved from last time (since it was saved when moving to queue and then re-read when moving back to stack). Now let's move to the list.

```
{ a, b, c,  }
1. Append
2. Remove
3. Save & Move to Stack
4. Save & Move to Queue
5. Quit
? 1 8
```

Lists are shown with curly braces and commas between numbers. Selecting 'Append' requires the character to add just like the others. Recall that single digit numbers can be considered characters as well. Selecting 'Remove' will remove the number at the end of the list. Selecting any 'Save & Move' will save the list to the list file.

```
{ a, b, c, 8,  }
1. Append
2. Remove
3. Save & Move to Stack
4. Save & Move to Queue
5. Quit
? 5
```

Selecting "Remove" will remove the number 4. But here I am showing that we can quit the application. All menus have this option. This merely exits the application and doesn't clear the screen beforehand.

A few constraints:

- Assume the user will put a single space followed by a single character if adding to the data structure is chosen (i.e. push, enqueue or append). Remember that not all menu options require more input from user, so only scan for the extra character when it is expected.
- You must read from the appropriate file every time a view switches to a particular data structure.
    - Do this in the Enter method for that data structure. Load the data structure with the file contents. This is the ONLY thing an enter method should do.
- You must write the new data back to the file every time the view switches away from a data structure.
    - Do this in the Exit method for the data structure you are leaving. This is the ONLY thing an exit method should do.
- You must use Java's generic Stack, Queue, and ArrayList classes to store data and use Java's HashMap class for your jump tables. And you must provide an appropriate generic type parameter (use

"Character" for your data) where appropriate. Any use of "raw types"[1] will lower your grade as this is considered by most developers to be bad practice.

  - Note that Queue is an abstract class in Java. So while the compile time type will be Queue, the runtime type will be one of the built-in Java classes that extends Queue.

Empty Data Structures:

If the user selects "Pop", "Dequeue" or "Remove" and the corresponding data structure is empty, the operation should effectively do nothing (do NOT just let the application crash). This may mean that you need to check if the data structure is empty and not run the operation if it is, or you may use different operations that won't crash the program if the data structure is empty (refer to Java documentation to see what I mean, for example there is more than one method for removing from a Queue).

```
|   |
|---|
1. Push
2. Pop
3. Save & Move to Queue
4. Save & Move to List
5. Quit
?
```

Empty stack

```
|
1. Enqueue
2. Dequeue
3. Save & Move to Stack
4. Save & Move to List
5. Quit
?
```

Empty queue

```
{  }
1. Append
2. Remove
3. Save & Move to Stack
4. Save & Move to Queue
5. Quit
?
```

Empty list

Granted this may not be the best way to draw these data structures as empty, so feel free on this one part to draw the empty data structures how you see fit (but you should draw something at least).

Finite State Machines:

You must use a finite state machine to code this application and have three jump tables, one for when we enter a state, one for when we stay in a state and one for when we exit a state. So, you will need an enum for the states, i.e.:

```
enum State {
  IDLE,
  STACK,
  QUEUE,
  LIST
}
```

And inside of your Screen class (more on this below), you will need the HashMap objects (i.e. our jump tables):

```
private HashMap<State, StateEnterExitMeth> stateEnterMeths;
private HashMap<State, StateStayMeth>      stateStayMeths;
private HashMap<State, StateEnterExitMeth> stateExitMeths;
```

Notice that we are using two different types of methods. The first, StateEnterExitMeth is for entering and exiting a state. Create this as an interface and make the invoke function take no parameters and not return anything (i.e. it should be void). The second, StateStayMeth, is for our stay methods. This should be made into a separate interface where the invoke method in this returns a boolean (but still has no parameters). The reason for this is so we can track if the user selects "Quit" in any of the menus. When this happens, the corresponding state stay method can return false. Otherwise, it should return true. To help with this, this is what the main class and main method should look like:

[1] Google search this to see what I mean if you've never heard this term before

```java
public class JumpTableMain {
  public static void main(String[] args) {
    Screen screen = new Screen();
    boolean keepRunning = true;
    while(keepRunning) {
      keepRunning = screen.doState();
    }
  }
}
```

As you can see, the "doState" function (similar to the lecture notes) should grab the current state stay method and invoke it. It should then return what that method returns. If it returns false, the loop in main exits. If it returns true, we stay in the loop and doState is called again. Essentially, we are trapped in an infinite loop until the user selects "Quit" in any of the menus.

To be clear, doState should only call the corresponding state stay method. Whereas changeState should call the appropriate exit method for the state we are leaving then the appropriate enter method for the state we are entering. The lecture notes show this as well (only without the exit methods).

Classes and Other Data Types:
For this assignment, you will need to create the following:

- An interface called `StateEnterExitMeth`
  - This contains a single method called invoke that doesn't receive any parameters and doesn't return anything.
- An interface called `StateStayMeth`
  - This contains a single method called invoke that doesn't receive any parameters but does return a boolean result.
- An enum called `State`
  - This contains all of the states our application can be in (see above).
- A class called `Screen`
  - This will be where the bulk of the application code is. This will contain the three jump lists, the data structures (ArrayList, Stack and Queue), and the current state of the application.
  - It will also contain all state methods that will draw each data structure, load and save them from/to their respective files, draw each menu, and carry out the operations for each data structure as they are selected from the menu.
- A public class called `JumpTableMain`
  - This will contain our entry point (see above).

File I/O:
You are given three files for this assignment, "stack.txt", "queue.txt", and "list.txt". The files all start with the same data, i.e. "a,b,c,". Yes, the comma at the end is purposefully there. This will make it easier for you to read the contents and not have to handle the special case of the last character not having a comma after it. When you write to the file, the trailing comma should be written. You are to read in the file using file i/o in Java. You may have to Google search this as I suspect most of you have never done this type of thing. Don't worry, Java makes it pretty easy to read and write to/from a file as there are a few classes built in to accomplish this (each with their own pros and cons, pick whichever one works for you).

Remember, when you enter a state, you read from the corresponding file and fill the data structure. When you exit from a state, you save the data structure contents back to the file. The file is only touched during entering and exiting a state! Also, because you read from the file every time you enter a state, I should be able to modify the file contents directly, then re-enter a state and see the new file contents in that data structure. For example, if I go to the queue, clear it out with multiple dequeue operations, then go to list, I should see an empty queue.txt file. If I then manually enter into the file "1,2,3," and save the file, then go back to the queue state, I expect to see "1 2 3" on the queue.

Coding Style:
I won't be checking for good coding style in this assignment. However, that doesn't mean that it isn't important! You might not lose points for bad style but do note that just getting an application to work correctly is not everything. It must also look good! Remember that in industry, you rarely work alone. If your code is sloppy and doesn't use best practices, your application will be a nightmare to maintain and update, and fellow developers will not enjoy working with you. For this reason, it is important to not have any hard-coded literals (e.g. the file names should be constants in the class that you refer to) and you should absolutely create helper functions that are called from your state methods. For example, a function to grab the character to add to the data structure from the user input, a function to clear the screen, a function to load a data structure from its corresponding file, save a data structure's content to its corresponding file, drawing the data structure to the screen, drawing the menu for that data structure, and any other helper methods you think you will need. Remember the D.R.Y. principal (i.e. "Don't Repeat Yourself"). If you find yourself writing the same logic twice or more, you should consider making a helper function for this. This can also help to make things such as redrawing the data structure after a "push", "pop", "enqueue", "dequeue", "append" or "remove" a lot easier. Again, while you won't lose points for not considering these things, points aren't everything! These lessons are just as important with or without point deductions.

You should also be putting thought into the order of your methods. Similar methods should be next to each other. Personally, I put extra blank lines between groups of methods. For example, I put all of my StateEnter methods (StateEnterIdle, StateEnterStack, StateEnterQueue, and StateEnterList) with no blank lines between them but then put a blank line before the next group. In other words:

```
/// ENTER
private void StateEnterIdle() {}
private void StateEnterStack() {}
private void StateEnterQueue() {}
private void StateEnterList() {}

/// STAY
private boolean StateStayIdle() {}
private boolean StateStayStack() {}
private boolean StateStayQueue() {}
private boolean StateStayList() {}

/// EXIT
private void StateExitIdle() {}
private void StateExitStack() {}
private void StateExitQueue() {}
private void StateExitList() {}
```

I then collapse them by pressing alt+2 (on the numpad above the letters) in Notepad++. You could also put a single blank line between functions in a group and put two blank lines between groups. However you want your code to look, as long as it is consistent and easy to find things.

Important Note:
Read everything in this document carefully. If you still don't understand, read it again. If you are still unclear about aspects of this assignment, ask me. <u>Failure to follow any directions will result in heavy point penalties</u>. If you go rogue and decide to do your own thing, even if it works you will still likely not get an A on this assignment. I have been very clear in this assignment and in the lecture notes so the only way you will get this wrong will be if you don't read carefully and don't follow directions.

Bonus:
You are given a mysterious file called Bonus.class. For this assignment to run correctly, you will need to compile and run your code on a linux-like terminal such as Git Bash. Make sure the Bonus.class file is in the same directory as your JumpTableMain.java file. Add a call to the static method “check” from the Bonus class after drawing the menu to the screen. Do this for stack, queue and list. Send the function the appropriate data structure. For example, right after drawing the stack menu, add the following code:

```
    Bonus.check(stack);
```

This assumes your Stack variable is called “stack” (change it to whatever your Stack variable is called if not). Right after drawing the queue menu, you should have: `Bonus.check(queue);` And right after drawing the array list menu, you should have: `Bonus.check(list);` To reveal the bonus, just think of what you want and write it into the data structure 😉

For submission:
Submit only your "JumpTableMain.java" file (everything should be in a single file) to Moodle.

Rubric:

#	ITEM	POINTS
1	Stack works	8
2	Queue works	8
3	ArrayList works	8
4	Uses Java’s built in data structures	5
5	Starting menu is correct	2
6	User can quit from any menu	2
7	Output is formatted as sample output shows	2
8	File I/O works correctly	5
9	State machine done correctly	5
10	State enter, stay and exit methods are correct	5
	TOTAL	**50**

Penalties

#	PENALTIES	POINTS
1	Doesn’t compile	-50%
2	Doesn’t execute once compiled (i.e. it crashes)	-25%
3	Late up to 1 day	-25%
4	Late up to 2 days	-50%
5	Late after 2 days	-100%