CS2001 / CS2101 Week 7 Assignment: Stacks and Queues

Dharini Balasubramaniam (cs2001.staff@st-andrews.ac.uk)

Due date: Wednesday 29th October, 21:00
MMS shows practical weighting and definitive date and time for deadlines

You are expected to have read and understood all the information in this specification and any accompanying documents at least a week before the deadline. You must contact the lecturers regarding any queries well in advance of the deadline.

Objective

To reinforce your understanding of data structures in Java, particularly stacks and queues.

Learning Outcomes

By the end of this assignment you should be able to:

- implement ADTs satisfying provided interfaces, and
- construct a queue using two stacks.

Getting started

To start with, you should create a suitable assignment directory such as

```
/cs/home/<your_username>/Documents/CS2001/W07-Stacks-Queues
```

on the Linux lab clients. Note that you should replace `<your_username>` in the above with your actual username on the Linux machines. Once you have set up your assignment directory, you should decompress the `zip` file at

http://studres.cs.st-andrews.ac.uk/CS2001/Coursework/W07-Stacks-Queues/code.zip

to your assignment directory. Please note that the `zip` file contains a number of files in the `src` directory, some of which are blank or only partially implemented. All your source code should be developed with `src` as the root directory for source code. **Once you have extracted the `zip` file, and have completed some of your own implementation, take care that you don't extract the `zip` file again thereby accidentally overwriting your `src` directory (and your own implementation) with files contained in the `zip`.**

Requirements

The practical is organised into two parts. You need to attempt them in order. You will find the requirements for each sub-part in the sub-sections below. Each part involves implementing a particular ADT as outlined below. You are given code in the `code.zip` file on StudRes as mentioned above. Within the main source code directory `src`, the code is organised into the packages `common`, `impl`, `interfaces`, and `test` (and associated directory structure).

Your task is to develop an implementation of the interfaces in the `interfaces` package by writing suitable classes in the `impl` package. You should also write tests in the classes provided in the `test` package. Parts of the `impl.Factory` class have been implemented already. For this class you need to implement only the methods containing `// TODO` comments. You should use the Factory in your code where possible, as the sample test in each test class shows. Should you find some aspects of the interfaces ambiguous, you will need to make a decision as to how to implement the interface, which your tests should make clear.

As with earlier assignments, please make sure that you do not modify the ADT interfaces or package structure.

Part 1: DoubleStack

Write classes in the `impl` package to provide the functionality of two stacks that share a single array object of a specified fixed size to store stack elements. Your double stack class should implement the interface shown in Listing 1:

```
public interface IDoubleStack {
    IStack getFirstStack();
    IStack getSecondStack();
}
```

Listing 1: IDoubleStack interface

The stack interface `IStack` (for each stack contained within an `IDoubleStack` object) is as defined in lectures and shown in Listing 2 below. The interface included in the `code.zip` file mentioned above contains the Javadoc comments with further explanation.

```
public interface IStack {
    void push(Object element) throws StackOverflowException;
    Object pop() throws StackEmptyException;
    Object top() throws StackEmptyException;
    int size();
    boolean isEmpty();
    void clear();
}
```

Listing 2: IStack interface

Given an `IDoubleStack myDoubleStack` object with sufficient free space in the array that is shared between both `IStack` objects within the double stack, it should be possible to e.g. push the values 3 and 7 onto the first and second stacks within the double stack object respectively via the code shown in Listing 3.

```
myDoubleStack.getFirstStack().push(3);
myDoubleStack.getSecondStack().push(7);
```

Listing 3: Sample DoubleStack method calls

You should similarly be able to invoke `pop` and the other operations on the individual stack objects in the double stack.

The two stacks should not conflict with each other despite sharing a single array for storage, and they should each be able to make use of up to half of the single array that is shared among two stacks within a double stack.

For example, if the underlying array has length 10, then at any given point in time it should be possible for each stack to contain up to a maximum of 5 elements. It should not be possible for one to contain 4 elements and the other 6 for example. You may see this as an artificial restriction on the size, because the array has 10 spaces, however, this restriction has been made to simplify implementation in part 2. Make sure you explain and justify your design and implementation decisions in your report.

Hints:

- One way to approach this would be to store one stack at the beginning of the array, with the bottom of the stack at position `x[0]`, and the second stack at the end of the array, with the bottom of that stack at position `x[x.length-1]`.
- You will also need to write a new class that implements the `IStack` interface (compared to the example code supplied in Lectures) which is given the underlying shared array, and a flag which indicates which of the two stacks it refers to.

Part 2: DoubleStackQueue

In this part of the practical, you are going to implement a Queue using two stacks, i.e. using your double stack from part 1. The resulting double stack based queue should conform with the queue interface in Listing 4:

```
public interface IQueue {
    void enqueue(Object element) throws QueueFullException;
    Object dequeue() throws QueueEmptyException;
    int size();
    boolean isEmpty();
    void clear();
}
```

Listing 4: IQueue interface

You should write a `DoubleStackQueue` class to provide the functionality above and will have to think about the attributes that you will need. In essence, you need to think about how a queue may be implemented using the two stacks in a `DoubleStack`. Some hints are given below to get you started. However, you will also have to think about how you will deal with the cases when the stacks are empty or full, how this does or doesn't relate to the queue being empty or full, and what size the individual stacks in the double stack should each have with respect to the queue size. Explanations and justifications of design and implementation are always important in your report.

Hints:

- One way to do this is to use one of the stacks for input (i.e. when enqueueing elements) and the other stack for output (when dequeueing elements).
- When enqueueing, you can push to the input stack.
- When dequeueing, you should return an element by popping from the output stack if it isn't empty. If the output stack is empty, you should first pop all the elements from the input stack and push them onto the output stack and then subsequently return an element by popping from the output stack.

Testing

Write JUnit tests to test your `ArrayDoubleStack` and `DoubleStackQueue` implementations considering normal, edge, and exceptional cases.

Specifically, write JUnit tests for your double stack in the `TestArrayDoubleStack` class that demonstrate that both stacks in your DoubleStack implementation function correctly. Similarly, write JUnit tests in the `TestDoubleStackQueue` class to demonstrate correct operation of your DoubleStack-based queue implementation.

Please make sure that the auto checker can run your tests in the `test` package prior to submitting.

Running the Automated Checker

Similarly to earlier assignments, you can run the automated checking system on your program by opening a terminal window connected to the Linux lab clients/servers and execute the following commands:

```
cd /cs/home/$(whoami)/Documents/CS2001/W07-Stacks-Queues
stacscheck /cs/studres/CS2001/Coursework/W07-Stacks-Queues/Tests
```

assuming `/cs/home/<your_username>/Documents/CS2001/W07-Stacks-Queues` is your assignment directory. This will run the JUnit test classes in the `test` package on your ADT implementation(s). A test is included in the test classes to check that your Factory can create non-null double stack and queue objects. The final test `TestQ CheckStyle` runs Checkstyle over your source code using the *Kirby Style* as usual.

`https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/programming-style.html`

Deliverables

Hand in via MMS, by the deadline of 9pm on Wednesday of Week 7 a `zip` file containing:

- Your assignment directory with all the source code for your ADT implementations, your tests and any dependencies (i.e. any other files that are needed to compile and run your code).
- A PDF report describing your solution. It is expected that your report will contain the following sections:
  - Overview
  - Design and implementation
  - Testing
  - Critical evaluation
  - References (if relevant)

  You might include diagrams to explain how your stack and queue implementations are structured and operate, and refer to and explain these diagrams in your main text. You may also wish to look at the report writing guidelines in the Student Handbook at

  `https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/writing-guidance.html`

Marking Guidance

The submission will be marked according to the mark descriptors used for CS2001/CS2101, which can be found at:

https://studres.cs.st-andrews.ac.uk/CS2001/0-General/descriptors.pdf

A very good attempt in an object-oriented fashion achieving almost all required functionality, together with a clear report showing a good level of understanding, can achieve a mark of 14 - 16. This means you should produce very good, re-usable code with very good method decomposition and provide a very good set of tests with clear explanations and justifications of design and implementation decisions in your report. To achieve a mark of 17 or above, you will need to implement all required functionality with a comprehensive set of test cases, testing all aspects of your design and covering any cases. Quality and clarity of design, implementation, testing, and your report are key to obtaining top marks.

Lateness

The standard penalty for late submission applies (Scheme A: 1 mark per 24 hour period, or part thereof): https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

Good Academic Practice

As usual, I would remind you to ensure you are following the relevant guidelines on good academic practice as outlined at

https://www.st-andrews.ac.uk/education/handbook/good-academic-practice/ and
https://info.cs.st-andrews.ac.uk/student-handbook/academic/gap.html