CS2001 Week 8: Stacks and Queues

Jon Lewis (jon.lewis@st-andrews.ac.uk)

Due date: Wednesday 3rd November, 21:00
MMS shows practical weighting and definitive date and time for deadlines

Objective

To reinforce understanding of stacks and queues.

Learning Outcomes

By the end of this practical you should:

- be confident in implementing ADTs
- understand how to construct a queue using two stacks

Getting started

To start with, you should create a suitable assignment directory such as `Documents/CS2001/W08-StacksQueues` on your local machine or in your home directory on the Linux lab clients when using these physically, remotely via SSH or using VSCode to connect to the lab. If you are working remotely, there are video tutorials and information available at

`https://systems.wiki.cs.st-andrews.ac.uk/index.php/Video_tutorials`
`https://systems.wiki.cs.st-andrews.ac.uk/index.php/Working_remotely`

If you are experiencing problems trying to connect to the lab you can get in touch by email with `fixit@cs.st-andrews.ac.uk`. You can also work remotely on your own machine, copying files over to your host server to run `stacscheck`, using a graphical SFTP client or secure copy client as indicated in the W01-Exercise handout.

`https://studres.cs.st-andrews.ac.uk/CS2001/Practicals/W01-Exercise/W01-Exercise.html`

If you are working on your own machine, I would remind you that we expect you to backup your work at least every 24 hours, such as by using OneDrive as indicated at

`https://info.cs.st-andrews.ac.uk/student-handbook/course-specific/sub-honours.html#Coursework`

Once you have set up your assignment directory, you should decompress the `zip` file at

`https://studres.cs.st-andrews.ac.uk/CS2001/Practicals/W08-StacksQueues/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 which you need to attempt in order. Please 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 job 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 `impl.Factory` have been implemented, for this class you only need to implement 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 in earlier practicals, please make sure that you do not modify the ADT interfaces or package structure.

Part 1: DoubleStack

Write classes in the `impl` package to help you 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 following interface:

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

The stack interface `IStack` (for each stack contained within an `IDoubleStack` object) is as defined in lectures and shown below. Please also note that the interface included in the `code.zip` file mentioned above contains the Javadoc comments for 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();
}
```

Given an `IDoubleStack myDoubleStack` object with sufficient free space in the shared array, 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

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

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, 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 the stacks 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 DoubleStack from part 1. The resulting double stack based queue should conform with the following queue interface:

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

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 stacks should 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 demonstrates that both stacks 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 practicals, 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 ~/Documents/CS2001/W08-StacksQueues
stacscheck /cs/studres/CS2001/Practicals/W08-StacksQueues/Tests
```

assuming `Documents/CS2001/W08-StacksQueues` 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 8 a `zip` file containing:

- Your entire 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 design, implementation and testing. You might include diagrams to explain how your stack and queue implementations are laid out 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 CS21001/CS2101, which can be found at:

`https://studres.cs.st-andrews.ac.uk/CS2001/Assessment/descriptors.pdf`

Assuming you have a good set of tests and a good report, you can achieve a mark of up to 11 for producing a good solution to either part 1 alone and a mark of up to 16 for producing good solutions to both parts 1 and 2. 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 at the top end.

Lateness

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof): `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`

Good Academic Practice

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

`https://www.st-andrews.ac.uk/students/rules/academicpractice/`