CS2001 Week 8: Stacks and Queues

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

Due date: Wednesday 6th November, 21:00
25% of Practical Mark for the Module

Objective

To reinforce understanding of stacks and queues.

Learning Outcomes

By the end of this practical you should:

- be confident in implementing simple ADTs using arrays
- understand the priority queue abstraction

Getting started

To start with, you should create a suitable assignment directory such as `CS2001/W08-StacksQueues` on the Linux lab clients. 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 and you can attempt these parts in any 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 test classes 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: Two Stacks

Write one or more 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 be able to make use of all unused space in the single array shared among two stacks within a double stack. If one stack is empty, the other should be able to occupy the entire array.

For example, if the underlying array has length 10, then at a given point in time it should be possible for one stack to contain 10 elements and the other 0, or for both stacks to contain 5 elements. It should not be possible for one to contain 5 elements and the other 6 as these would occupy more space than available in the single, shared array of size 10.

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: Priority Queue

A *Priority Queue* is an abstract data type which is like a normal queue, but in which each element has a priority associated with it. In a priority queue, an element with higher priority is dequeued prior to an element with lower priority, i.e. in *priority* order. However, among elements of equal priority, normal queue semantics (FIFO ordering) should be preserved and elements should be dequeued according to their position in the queue. Write an array-based implementation of a Priority Queue ADT with the following interface:

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

The priority of elements should be defined by the total ordering imposed by the `compareTo` method for objects enqueued in your priority queue of `Comparable` objects. If `someObject.compareTo(other)` returns a positive value, `someObject` should be treated as having higher priority than the `other` object. Similarly, for a negative return value, `someObject` should be treated as having lower higher priority than the `other` object. For a return value of zero, `someObject` should be treated as having the same priority as the `other` object. As we haven't formally covered generics yet, you may assume that at any point in time, your queue will only ever contain objects of compatible type and warnings about using the raw `Comparable` type can be disregarded. Also, the `common` package contains a `PriorityObject` class to represent prioritised objects which may help you test some aspects of your priority queue implementation more easily than when using integers (see Javadoc in the class).

Testing

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

Specifically, write JUnit tests for your double stack in the `test.ArrayDoubleStackTests` class that demonstrates that both stacks function correctly, and that their sizes are related in the expected way. Similarly, write JUnit tests in the `test.ArrayPriorityQueueTests` class to demonstrate correct operation of your priority queue.

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 ~/CS2001/W08-StacksQueues
stacscheck /cs/studres/CS2001/Practicals/W08-StacksQueues/Tests
```

assuming `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 priority 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 report with an advisory limit of around 1000 words, in PDF format, describing your design, implementation and testing. You might include diagrams to explain how your queue and stack 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 or part 2 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/`