# Lab Exercise 8: Queues and Stacks 2; Generics
# CS 2334

March 6, 2018

## Introduction

In this lab, you will experiment with two common data structures: stacks and queues. In particular, you will use the PriorityQueue class from the Java Collections Framework and will implement a stack and queue yourself. You will also be creating and using generic classes.

## Learning Objectives

By the end of this laboratory exercise, you should be able to:

1. Create a Stack data structure to store and retrieve objects
2. Create a Queue data structure to store and retrieve objects
3. Use the PriorityQueue class to store and retrieve objects
4. Create and use Generic Classes and Methods

## Proper Academic Conduct

This lab is to be done individually. Do not look at or discuss solutions with anyone other than the instructor or the TAs. Do not copy or look at specific solutions from the net.

# Preparation

1. Import the existing lab8 implementation into your eclipse workspace.

   (a) Download the lab8 implementation from canvas.

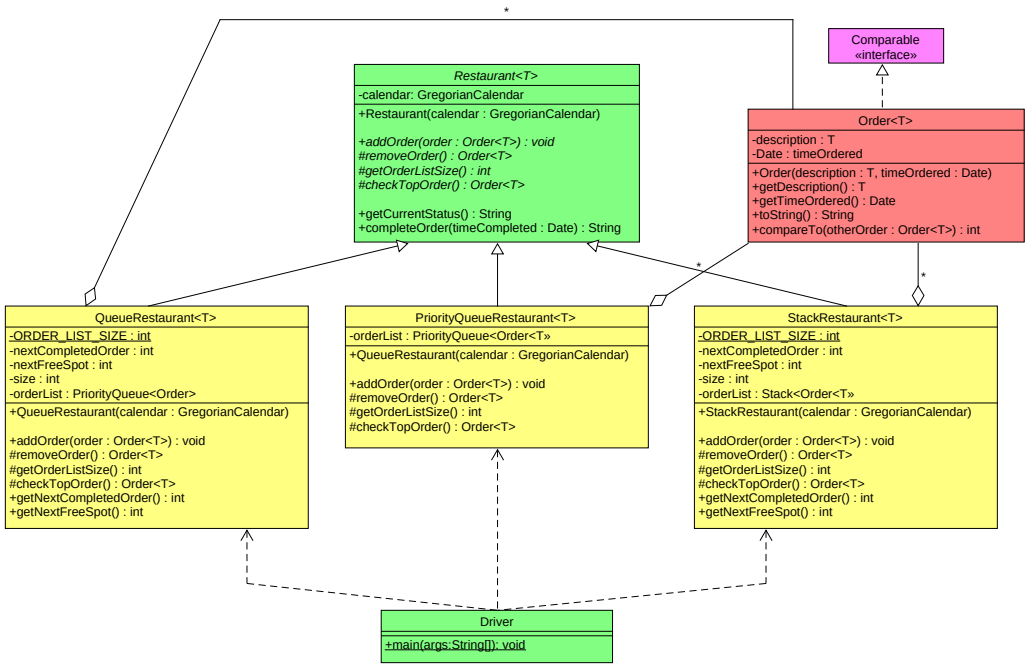
   (b) In Eclipse, select *File/Import*

   (c) Select *General/Existing projects into workspace*. Click *Next*

   (d) Select *Select archive file*. Browse to the lab7.zip file. Click *Finish*

# Class Design

Below is the UML representation of the set of classes that make up the implementation for this lab. Note that the code for PriorityQueue is fully implemented, but you are to add (java-doc style) documentation to it.




- Green colored blocks indicate that they are fully implemented.
- Yellow colored blocks indicate that they are partially implemented.
- Red colored blocks indicate that they have to be implemented by you.
- Purple colored blocks are Java standard classes/interfaces, and you should not attempted to create them.

The key classes in our project are:

- **Order**: class representing an order at a restaurant. The class has two variables: the *description* of the order (i.e. what it is) and the *timeOrdered* marking when the order was made. Some getters are made, and the toString always returns the *description*. The constructor takes in a description and a Date which the order should store. The compareTo() in order should compare Dates with the order that it is being compared with. i.e it should return this-¿date-¿compareTo(other-¿date).

  This will look very similar to the Order class from the previous lab, but keep in mind that this is now a generic type. Instead of having a String be the description for an Order, the description can now be of any class! Which class that is is determined by the generic type (T in this case). You have seen generics before largely in the context of ArrayLists. When instantiating a list, you give the generic type for it. e.g.

```
ArrayList<String> stringList = new ArrayList<String>();
ArrayList<Integer> stringList = new ArrayList<Integer>();
```

  How does the ArrayList know what type of things it will store? This is done via generics. Now, you will be creating a generic class for Orders. Similar to the code for ArrayLists, you may have lines such as:

```
Order<String> order1 = new Order<String>("hello", new Date());
Order<Integer> order1 = new Order<Integer>(new Integer(5), new Date());
```

  You should note that the restaurant classes are also now generic. This is done to allow them to store orders of different generic types. For more information on how to handle generics, you should see the link in the hints below.

- **Restaurant**: An abstract class representing a restaurant. A Restaurant is expected to store a list of orders, and provides methods for adding and removing them. Subclasses define how orders are stored and removed. In addition, some methods are added to get additional information about the Restaurant or add public accessibility. The following methods defines how it works:

  - Constructor: takes in a GregorianCalendar and stores it.
  - addOrder(): Abstract method. Implementing classes should add the order passed in into an internal data structure. I.e. the order should be stored in some way.

- removeOrder(): Abstract method. Implementing classes should remove a stored Order from their internal data structures. This method and addOrder define the order in which Orders are completed.
- getOrderListSize(): Abstract method. Implementing classes should return an int indicating the number of orders that are stored in some internal data structure.
- checkTopOrder(): Abstract method. Implementing classes should return the order that would next be removed (upon call to removeOrder) **without** removing the order from the storage of the internal data structure. I.e. check what will next be removed without actually removing it.
- getCurrentStatus(): returns some information. In particular, returns information on the current number of orders stored and what order is next.
- completeOrder(): public method providing utility to other classes to remove the next order from the internal data structure. Also computes the time since the order was created to the given time that the order was completed and returns information based on this.

- **StackRestaurant**: An implementation class of Restaurant. A StackRestaurant stores orders in a Stack *orderList*. The spaces in which elements may be inserted is defined by *orderList*, but what is actually in the stack is defined by its start and end indices (locations inside the orderList). See the code commentary and the link in the hints section to better understand what you should do to complete this class.

  Stacks are a "Last In - First Out" or "LIFO" data structure. This means that the last element added to a stack is the first one that is removed from it. e.g. we add the elements A,B, and C to a stack in that order ("pushing" the values onto the stack). We then remove 3 elements ("popping") from the stack. They will be removed in the order c, B, A.

  The StackRestaurant completes orders in a LIFO ordering. The overridden abstract methods should reflect this.

- **QueueRestaurant**: An implementation class of Restaurant. A QueueRestaurant stores orders in a Circular Queue. The spaces in which elements may be inserted is defined by *orderList*, but what is actually in the queue is defined by its start and end indices (locations inside the orderList). See the code commentary and the link in the hints section to better understand what you should do to complete this class.

Queues are a “First In - First Out” or “FIFO” data structure. This means that the first element added to a queue is the first one that is removed from it. e.g. we add the elements A,B, and C to a queue in that order. We then remove 3 elements from the queue. They will be removed in the order that they were inserted in: A, B, C.

The QueueRestaurant completes orders in a FIFO ordering. The overridden abstract methods should reflect this.

- **PriorityQueueRestaurant**: An implementation class of Restaurant. A PriorityQueueRestaurant stores orders in a PriorityQueue *orderList*. Queues are a “First In - First Out” or “FIFO” data structure. This means that the first element added to a queue is the first one that is removed from it. PriorityQueues are a bit different, however. These sort the elements based on some ordering. The first element to be removed is determined by which element has the ”highest” priority. Because this PriorityQueue stores orders, it uses the natural ordering of Order (the compareTo) to determine how to sort the orders. Some of you noticed a bug in the previous lab as a result of this! We added a compareTo that always returned 0. Because of the internal sorting algorithm in PriorityQueue, this sometimes shuffled elements and made them not in the order they were inserted.

  e.g. we add the elements A,B, and C to a queue in that order. We sort these element by size. We say that the sizes are as follows: B ¿ A ¿ C. We then remove 3 elements from the queue. They will be removed in the order that they were inserted in: B, A, C.

# Lab 8: Implementation Steps

Start from the class files that are provided in lab8.zip.

1. The **Driver**, **Restaurant**, and *PriorityQueue* classes have been fully implemented and **their code should not be modified**. You do however need to **add java-doc style documentation to the PriorityQueue class.**

2. Finish the class **StackRestaurant**. Be sure to properly update indices in the stack.

3. Finish the class **QueueRestaurant**. Be sure to properly update indices in the stack.

4. Implement the **Order** class.

5. Implement JUnit tests to thoroughly test all classes and methods you created/implemented.

    - We have given a **RestaurantTest.java** for reference, which you can make use of for creating other tests. This test class should already cover all of *Order* and QueueRestaurant. You may add to this test class.
    - You need to convince yourself that everything is working properly
    - Make sure that you cover all of the cases within the methods while creating your tests. Keep in mind that we have our own tests that we will use for grading.

# Hints

- See the documentation for PriorityQueue. Keep in mind the methods for a Queue (poll, peek, add):

  https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html

- See the documentation for Date (be wary of the methods listed as "deprecated"):

  https://docs.oracle.com/javase/8/docs/api/java/util/Date.html

- Stacks: https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

- Circular Queues:

  https://www.geeksforgeeks.org/circular-queue-set-1-introduction-array-implementation/

- Java Generic Classes:

  https://docs.oracle.com/javase/tutorial/java/generics/types.html

# Example Output

Below is an example output of the full program. The details of your interaction will vary (we do not test the output of Driver).

```
Please choose a restaurant option:
1. [enter] an order.
2. [complete] an order.
3. [check] the next order to be completed.
4. [quit]
enter
Please enter an order description and an order time (comma separated) with the ↩
    following format:
<description>,<time as a long>
hello,1000
Please choose a restaurant option:
1. [enter] an order.
2. [complete] an order.
3. [check] the next order to be completed.
4. [quit]
check
For the stack restaurant:
hello
For the queue restaurant:
hello
For the priority queue restaurant:
hello
Please choose a restaurant option:
1. [enter] an order.
2. [complete] an order.
3. [check] the next order to be completed.
4. [quit]
enter
Please enter an order description and an order time (comma separated) with the ↩
    following format:
<description>,<time as a long>
test
Please enter an appropriate order!
Please choose a restaurant option:
1. [enter] an order.
2. [complete] an order.
3. [check] the next order to be completed.
4. [quit]
100
Please choose a restaurant option:
1. [enter] an order.
2. [complete] an order.
3. [check] the next order to be completed.
4. [quit]
enter
Please enter an order description and an order time (comma separated) with the ↩
    following format:
<description>,<time as a long>
test,100
Please choose a restaurant option:
```

```
1. [enter] an order.
2. [complete] an order.
3. [check] the next order to be completed.
4. [quit]
etner
Please choose a restaurant option:
1. [enter] an order.
2. [complete] an order.
3. [check] the next order to be completed.
4. [quit]
enter
Please enter an order description and an order time (comma separated) with the ↩
    following format:
<description>,<time as a long>
test2,500
Please choose a restaurant option:
1. [enter] an order.
2. [complete] an order.
3. [check] the next order to be completed.
4. [quit]
check
For the stack restaurant:
test2
For the queue restaurant:
hello
For the priority queue restaurant:
test
Please choose a restaurant option:
1. [enter] an order.
2. [complete] an order.
3. [check] the next order to be completed.
4. [quit]
complete
Please enter the time of completion as a long:
10000
The completion for the stack restaurant:
It tooks 0 hours, 0 minutes, and 10 seconds to complete the following order: test2

The completion for the queue restaurant:
It tooks 0 hours, 0 minutes, and 9 seconds to complete the following order: hello

The completion for the priority queue restaurant:
It tooks 0 hours, 0 minutes, and 10 seconds to complete the following order: test

Please choose a restaurant option:
1. [enter] an order.
2. [complete] an order.
3. [check] the next order to be completed.
4. [quit]
check
For the stack restaurant:
test
For the queue restaurant:
test
For the priority queue restaurant:
test2
Please choose a restaurant option:
1. [enter] an order.
```

```
2. [complete] an order.
3. [check] the next order to be completed.
4. [quit]
quit
```

# Final Steps

1. Generate Javadoc using Eclipse.
   - Select *Project/Generate Javadoc...*
   - Make sure that your project (and all classes within it) is selected
   - Select *Private* visibility
   - Use the default destination folder
   - Click *Finish*.
2. Open the *lab8/doc/index.html* file using your favorite web browser or Eclipse (double clicking in the package explorer will open the web page). Check to make sure that that all of your classes are listed and that all of your documented methods have the necessary documentation.
3. If you complete the above instructions during lab, you may have your implementation checked by one of the TAs.

# Submission Instructions

Before submission, finish testing your program by executing your unit tests. If your program passes all tests and your classes are covered completely by your test classes, then you are ready to attempt a submission. Here are the details:

- All required components (source code and compiled documentation) are due at 11:59pm on Friday, March 9. **Submission must be done through the Web-Cat server.**
- Submit the assignment plan to canvas by the end of your lab session. Submit the completed plan with updated time tracking to canvas when you have finished your lab.

- Use the same submission process as you used in lab 4. You must submit your implementation to the *Lab 8: Queues and Stacks 2; Generics* area on the Web-Cat server.

# Rubric

The project will be graded out of 100 points. The distribution is as follows:

**Correctness/Testing: 45 points**

The Web-Cat server will grade this automatically upon submission. Your code will be compiled against a set of tests (called *Unit Tests*). These unit tests will not be visible to you, but the Web-Cat server will inform you as to which tests your code passed/failed. This grade component is a product of the fraction of **our tests** that your code passes and the fraction of **your code** that is covered by *your tests*. In other words, your submission must perform well on both metrics in order to receive a reasonable grade.

**Style/Coding: 20 points**

The Web-Cat server will grade this automatically upon submission. Every violation of the *Program Formatting* standard described in Lab 1 will result in a subtraction of a small number of points (usually two points). Looking at your submission report on the Web-Cat server, you will be able to see a notation for each violation that describes the nature of the problem and the number of subtracted points.

**Design/Readability: 35 points**

This element will be assessed by a grader (typically sometime after the lab deadline). Any *errors* in your program will be noted in the code stored on the Web-Cat server, and two points will be deducted for each. Possible errors include:

- Non-descriptive or inappropriate project- or method-level documentation (up to 10 points)
- Missing or inappropriate inline documentation (2 points per violation; up to 10 points)
- Inappropriate choice of variable or method names (2 points per violation; up to 10 points)
- Inefficient implementation of an algorithm (minor errors: 2 points each; up to 10 points)
- Incorrect implementation of an algorithm (minor errors: 2 points each; up to 10 points)

If you do not submit compiled Javadoc for your project, 5 points will be deducted from this part of your score.

Note that the grader may also give *warnings* or other feedback. Although no points will be deducted, the issues should be addressed in future submissions(where points may be deducted).

**Bonus: up to 5 points**

You will earn one bonus point for every two hours that your assignment is submitted early.

**Penalties: up to 100 points**

You will lose ten points for every minute that your assignment is submitted late. For a submission to be considered *on time*, it must arrive at the server by the designated minute (and zero seconds). For a deadline of 9:00, a submission that arrives at 9:00:01 is considered late (in this context, it is one minute late).

After 15 submissions to Web-Cat, you will be penalized one point for every additional submission.

For labs, the server will continue to accept submissions for three days after the deadline. In these cases, you will still have the benefit of the automatic feedback. However, beyond ten minutes late, you will receive a score of zero.

The grader will make their best effort to select the submission that yields the highest score.