P08 DESSERT QUEUE

Posted on March 15, 2019 by dahl

- **SUBMIT your completed assignment by 9:59PM on Wednesday, April 10th 2019.** We will accept and grade submissions made through 9:59PM on Thursday, April 11th 2019 (HARD DEADLINE). But, NO CREDIT will be granted for any work that is submitted even one second after this hard deadline.
- **Pair Programming is NOT allowed for this assignment. You have to work on and complete this assignment INDIVIDUALLY.**

This Assignment involves implementing a queue using an array with circular indexing. Once you have developed and tested this queue implementation, you will use it to develop solutions for three different kinds of problems that involve peculiar seating and serving arrangements at a friend's monthly dinner parties.

OBJECTIVES AND GRADING CRITERIA

The main goals of this assignment include giving you practice and experience both implementing a queue and using that queue to solve some other problems. You will get practice using circular indexing and analyzing time-complexities, and you will also get more experience with test driven development.

20	Online Tests: these automated grading test results are visible upon uploading your submission. You are allowed multiple opportunities to correct the organization and functionality of your code (if necessary).
20 points	Offline Tests: these automated grading tests are run after the assignment's deadline has passed. They check for similar functionality and organizational correctness as the Online Tests. Since you will not have opportunities to make corrections after seeing the feedback from these tests, you should consider and test the correctness of your own code as thoroughly as possible.
10 points	Code Readability: human graders will review the commenting, style, and organization of your final submission. They will be checking whether it conforms to the requirements of the CS300 Course Style Guide. Since you will not have opportunities to make corrections after seeing the feedback from these graders, you should consider and review the readability of your own code as thoroughly as possible.

0. GETTING STARTED

Create a new Java8 project in Eclipse. You can name this project whatever you like, but P8 Dessert Queue is a descriptive choice. Then create a new class named QueueTests with a public static void main(String[] args) method stub. You'll be submitting your code for this assignment through gradescope. The four files that you will be submitting include: Guest.java, ServingQueue.java, QueueTests.java, and DessertSolvers.java.

1. THE GUESTS

The queue that we'll be implementing in the next step of this assignment will be used to keep track of guests at a dinner party, and the peculiar orders in which they are served. We'll start by creating a Guest class to keep track of the guests at these dinner parties. This is the type of object that our queue implementation will manage. The important properties that we'll need to track for each guest are 1) the index describing the order that they arrive to dinner in, and 2) whether that guest has any dietary restrictions. You should decide what fields (possibly a mix of instance and static) will be most appropriate to implement the following constructors and methods. You should NOT add any additional methods to this class.

```
1  public class Guest {
2    /**
3     * Resets the counting of newly constructed guest indexes, so that the next new Guest that i
4     * will be associated with the guest index of one.
5     *
6     * Note: that this will be helpful when running several tests, or solving solving several
7     * dessert simulation problems.  And that this should never be called from ServingQueue.java
8     */
9    public static void resetNextGuestIndex() {
10   }
11
12   /**
13    * Constructs a new guest with no dietary restrictions.  This guest should be associated wit
14    * index that is one larger than the previously constructed guest (or 1, if no prior guest,
15    * resetNextGuestIndex() was called more recently).
16    */
17   public Guest() {
18   }
19
20   /**
21    * Constructs a new guest with the specified dietary restrictions.  This guest should be
22    * associated with an index that is one larger than the previously constructed guest (or 1,
23    * no prior guest, or if resetNextGuestIndex() was called more recently).
24    * @param dietaryRestriction describes requirements for what this guest can and cannot eat
25    */
26   public Guest(String dietaryRestriction) {
27   }
28
29   /**
30    * Access whether this guest has any dietary restrictions or not
31    * @return true for guests constructed using new Guest(String), false otherwise
32    */
33   public boolean hasDietaryRestriction() {
34     return false;
35   }
36
37   /**
38    * The string representation of a Guest should be formatted as, for examples:
39    * #3(no dairy)
40    * for a guest with a guest index of 3 and the dietary restriction of "no dairy", or:
41    * #4
42    * for a guest with a guest index of 4 and no dietary restriction
43    * @return string representing the guest index and any dietary restriction they might have
44    * @see java.lang.Object#toString()
45    */
46   public String toString() {
47     return null;
48   }
49 }
```

2. SERVINGQUEUE IMPLEMENTATION

Next create a class called ServingQueue to implement a queue of Guest objects (this class should not use any generic type parameters). This queue must be implemented using an array with circular indexing. The capacity of the array holding these guest will NOT need to change or grow for this assignment. In order for the grading tests to ensure that your array is updated correctly, it must be defined as follows within your ServingQueue class, and the first guest added to this array must be added to index 0, the second guest should be added to index 1, etc.

```
private Guest[] array;
```

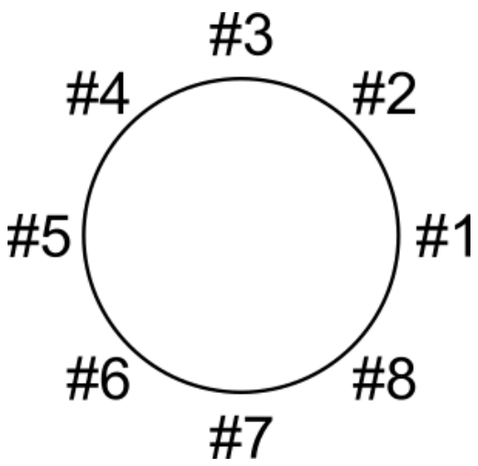
You are allowed to define additional private fields within this class, but only fields that have primitive types. The public constructor and methods that you must implement within this class include the following (any additional methods that you choose to implement must be private):

```
/**
 * Creates a new array based queue with a capacity of "seatsAtTable" guests.
 * This queue should be initialized to be empty.
 * @param seatsAtTable the size of the array holding this queue data
 */
public ServingQueue(int seatsAtTable) {
}

/**
 * Checks whether there are any guests in this serving queue.
 * @return true when this queue contains zero guests, and false otherwise.
 */
public boolean isEmpty() {
  return size == 0;
}

/**
 * Adds a single new guest to this queue (to be served after the others that
 * were previously added to the queue).
 * @param newGuest is the guest that is being added to this queue.
 * @throws IllegalStateException when called on a ServingQueue with an array that is full
 */
public void add(Guest newGuest) {
}

/**
 * Accessor for the guest that has been in this queue for the longest.  This method
 * does not add or remove any guests.
 * @return a reference to the guest that has been in this queue the longest.
 * @throws IllegalStateException when called on an empty ServingQueue
 */
public Guest peek() {
  return null;
}

/**
 * Removes the guest that has been in this queue for the longest.
 * @return a reference to the specific guest that is being removed.
 * @throws IllegalStateException when called on an empty ServingQueue
 */
public Guest remove() {
  return null;
}

/**
 * The string representation of the guests in this queue should display each
 * of the guests in this queue (using their toString() implementation), and should
 * display them in a comma separated list that is surrounded by a set of square brackets.
 * (this is similar to the formatting of java.util.ArrayList.toString()).  The order
 * that these guests are presented in should be (from left to right) the guest that has
 * been in this queue the longest, to the guest that has been in this queue the shortest.
 * When called on an empty ServingQueue, returns "[]".
 * @return string representation of the ordered guests in this queue
 * @see java.lang.Object#toString()
 */
@Override
public String toString() {
  return null;
}
```

To test your implementations of these queue methods within the class named QueueTests. Within this class, implements three or more static test methods that take no arguments, and return a

boolean value: true to indicate a passing test, and false otherwise. Also implement a main() method that calls each of these tests, and displays feedback about which tests have passed versus failed.

3. DESSERT PROBLEM



At your eccentric friend's dinner parties, guests are seated around a large round table in the order that they arrive (counter-clockwise around the table). The first guest to arrive will be served first, and the servers will then make their way counter-clockwise around the table. But instead of serving every guest in order around the table, they always skip the first waiting guest who has not yet been served, and serve the next waiting guest after them. This continues around the table until every guest has been served. So for the seating arrangement depicted on the right, the order that guests will be served in is: #1, #3, #5, #7, #2, #6, #4, #8. Notice that even after we have gone around the entire table, we continue to skip one waiting guest: notice that guest #4 is is skipped between serving guests #2 and #6. Also notice that the last guest #8 will be both skipped over and then served in the final round of serving. The reason that we are interested in this ordering, is that desserts are served starting with the last guest to receive the previous course. So when there are a total of eight guests, we'll want to arrive last so that we can be served our dessert first! And your friend's desserts are excellent! In fact this whole system of ordering was developed to thwart guests attempting to jockey for the first dessert, and was inspired by the classic (although somewhat morbid) Josephus Problem.

You will use your ServingQueue to simulate this serving of guests and to solve a few related problems. You can do this by removing guests from the queue when they are served, and by removing and then adding back to this queue guests who are skipped over and continue waiting to be served. For this purpose, create a new class called DessertSolvers. Within this class, you'll create static methods to solve each of the following problems.

3.1. VARIABLE NUMBER OF GUESTS SKIPPED

Here is the signature for the first static method that you will implement in the DessertSolvers class:

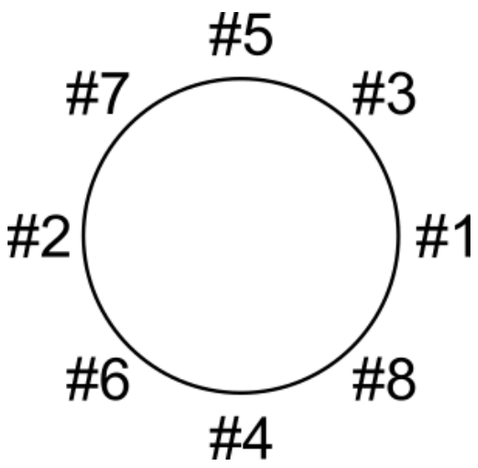
```
1 public static Guest firstDessertVariableSkips(int numberOfGuests, int guestsSkipped)
```

Start by implementing this method to solve the problem discussed above (temporarily ignoring the guestsSkipped parameter to begin). This method should take the number of guests at a party, create as many guests with an index starting at one, simulate the serving of those guests, and then return a reference to the guest that is served last. As we saw in the example above, firstDessertVariableSkips(8,1) should therefore return the last guest to arrive, or guest #8.

Now modify your implementation to account for different numbers of guestsSkipped. The original problem involved skipping a single waiting guest (guestsSkipped = 1) between any two guests that are served. This parameter should allow us to simulate skipping any (non-negative) number of guests. Using the table of eight guests above as an example, the order that these guests should be served in when guestsSkipped is three is: #1, #5, #2, #7, #6, #8, #4, #3.

If numberOfGuests is not positive or if the guestsSkipped is negative, your method should throw an IllegalArgumentException describing which of these problems was encountered.

3.2. VARIABLE NUMBER OF COURSES SERVED



For this variation of the original problem, we'll assume that one waiting guest is always skipped (rather than the variable number considered in the previous problem). However, we'll now consider meals consisting of a variable number of courses (whereas we previously only considered 2 courses: an entree followed by dessert).

Each course is served at a different table, and the order that guests are seated at each subsequent table is based on the order that they were served at the previous table. So when there are eight guests, the order that those guests are seated at the second table can be seen in the diagram to the right. And as we have previous discussed, the first guest to be served at this second table will be guest #8. So the complete order of guests being served at this second table is: #8, #3, #7, #6, #1, #2, #5, #4. And this will be the order that these guests are seated at a third table, assuming there are as many courses. For parties of eight guests and three courses, we'll want to be the fourth guest to arrive: in order to be served dessert first!

We'll simulate the serving of multiple courses in a new method with the following signature:

```
1 public static Guest firstDessertVariableCourses(int numberOfGuests, int coursesServed)
```

This method should take the number of guests at a party, create as many guests with an index starting at one, simulate the serving of those guests through the specified number of courses, and then return a reference to the guest that is served last in the second to last course (since this is the guest who will be served dessert first). You'll need to use a second queue to keep track of the order that guests are seated at each subsequent table in your solution. If coursesServed is 1, then you can simply return a guest #1 since they will be served their only course (dessert) first.

If either numberOfGuests or coursesServed is not positive, your method should throw an IllegalArgumentException describing which of these problems was encountered.

SUBMISSION

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the course style guide, you should submit your final work through gradescope.com. The four files that you must submit include: Guest.java, ServingQueue.java, QueueTests.java, and DessertSolvers.java. Your score for this assignment will be based on your "active" submission made prior to the hard deadline of **9:59PM on Thursday, April 11th**.

EXTRA CHALLENGES

Here are some suggestions for interesting ways to extend this simulation, after you have completed, backed up, and submitted the graded portion of this assignment. **No extra credit will be awarded for implementing these features**, but they should provide you with some valuable practice and experience.

- Create a new method in DessertSolvers that takes an array of guests as input, which makes use of the dietary restrictions that some guests may have. For example, you could imagine that there are two meals prepared: one for guests with dietary restrictions and one for guests without. Whenever a guest is served, the server checks whether the next waiting guest (the one that will be skipped) has any dietary restrictions. They bring out a corresponding meal and skip that waiting guest as usual. However, they may need to skip over additional guests if they brought out a meal that does not match the following guests dietary restrictions.
- Try changing your variable course number solver as follows. Have the servers alternate the order around the table that they progress when serving guests from one course to the next. For example, the first course can still be served in counter-clockwise order, but the second course should then be served in clockwise order, the third in counter-clockwise order, and so on... Could you implement this with the help of a stack? Or is there a change to your serving queue class that could help with this?
- There are some nice mathematical properties of this problem that can be used to implement more efficient solutions to some of these problems. Feel free to explore this wikipedia page or do addition google searches to learn more about such approaches. And of course, you may try implementing them to help test whether they really produce the same results as your previous implementations.