CSE 11: Introduction to Programming and Computational Problem Solving: Accelerated Pace

Assignment 5

Objects and Classes

Due Date: Wednesday, May 15, 11:59 PM

Learning goals:

- Implement classes with getters and setters.
- Write unit tests using objects and instance methods.

NOTE: This programming assignment must be done individually.

Your grade will be determined by your most recent submission. If you submit to Gradescope after the deadline, it will be marked late and the late penalty will apply regardless of whether you had past submissions before the deadline.

If your code does not compile on Gradescope and the autograder fails to execute properly, you will receive an automatic zero on the autograder portion of the assignment. Make sure that you do not change the function signatures or import other packages unless otherwise specified.

Coding Style (10 points)

For this programming assignment, we will be enforcing the CSE 11 Coding Style Guidelines. These guidelines can also be found on Canvas and the class website. Please ensure to have *COMPLETE* file headers, class headers, and method headers, use descriptive variable names and proper indentation, and avoid using magic numbers.

Part 0: Getting started with the starter code (0 points)

1. If using a personal computer, then ensure your Java software development environment does not have any issues. If there are any issues, then review Assignment 1, or come to the office/lab hours before you start Assignment 5.
2. First, navigate to the `cse11` folder you created in Assignment 1 and create a new folder named `assignment5`

3. Download the starter code. You can download the starter code from Piazza → Resources → Homework → `assignment5.zip`. The starter code contains four files: `Assignment5.java`, `Ticket.java`, `Queue.java`, and `Autograder.java`. Place the starter code within the `assignment5` folder you just created
4. Compile the starter code within the `assignment5` folder. You can compile all files using the single command `javac *.java` and you should get a compiler error saying that methods in either `Ticket.java`, `Queue.java`, or Autograder.java are missing return statements and have not been implemented yet. For example:

```
Autograder.java:21: error: missing return statement
    }
    ^
Autograder.java:24: error: missing return statement
    }
    ^
Autograder.java:27: error: missing return statement
    }
    ^
Autograder.java:30: error: missing return statement
    }
    ^
```

Overview

You are tasked with developing an Autograder program that manages a tutor help queue, just like the one you use in this class when you need help during office hours to submit a support ticket. This program is the simpler version of Autograder you have been using. It has the following three components:

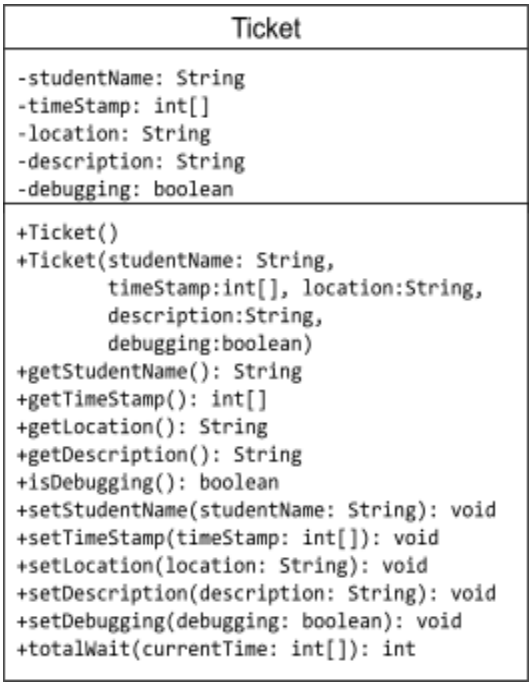
- Part 1: Class Ticket - represent properties of a support ticket
- Part 2: Class Queue - represents a queue of tickets
- Part 3: Class Autograder - represents a simplified autograder system where tickets can be submitted to the queue of a course

Part 1: Implement Ticket Class (20 points)

In this part of the assignment, you will implement a `Ticket` class that represents the main properties of a support ticket.

The Ticket Class

In `Ticket.java`, you will be implementing **2 constructors, 5 instance accessors (i.e, getter methods), 5 instance mutators (i.e., setter methods), and 1 instance method.** This class is without any great functionality of its own but acts as a template to model a 'Ticket.' Following is the UML class diagram.



Each `Ticket` object, which is an instance of the `Ticket` class, contains the following fields (all are provided in the starter code; do not change any of these):

- `private String studentName`: the name of the student submitting the ticket
- `private int[] timeStamp`: an int array of size 3 that stores the time the ticket is submitted to the queue, where index 0 is hour, index 1 is minutes, and index 2 is seconds. You are required to use 24-hour time format. For example, 10 hours, 24 minutes, 36 seconds PM (i.e., after noon) is 22 hours, 24 minutes, 36 seconds in 24-hour time format.
- `private String location`: the student's location
- `private String description`: a brief description of why the student is submitting a support ticket and requesting help
- `private boolean debugging`: a boolean that denotes whether the ticket is submitted to request help debugging an issue

Notice how each member field is declared `private`. This means that the member is only visible *within* the class, not from any other class. In other words, you will need to use accessors (i.e., getter methods) and mutators (i.e., setter methods) to access and modify, respectively, these `private` members. **You must also use the `this` keyword to access member variables hidden by local variables.**

The `Ticket` class contains the following member methods:

- `public Ticket()`
  This is the no-arg constructor of the `Ticket` class. This constructor needs to set the `studentName`, `timeStamp`, `location`, `description`, and `debugging` member variables according to what is shown below:
  - `studentName` must be set to `null`
  - `timeStamp` must be an array of length **3** that contains 3 zeros, that is `{0,0,0}`
  - `location` must be set to `null`
  - `description` must be set `null`
  - `debugging` must be set to `false`

- `public Ticket(String studentName, int[] timeStamp, String location, String description, boolean debugging)`
  This is another constructor of the `Ticket` class. This constructor needs to set the `studentName`, `timeStamp`, `location`, `description`, and `debugging` member variables using the constructor parameters. For `timeStamp`, you must perform a **deep copy** of the array referred to by the constructor parameter array reference variable to the array referred to by the instance array reference variable. This means that you cannot simply set the instance variable `timeStamp` to refer to the same array being referred to by the constructor parameter. Instead, you must copy the elements of the array it refers to into the array referred to by the instance variable. You can assume that the constructor parameter `timeStamp` will always be non null.

  Remember, you must use the `this` keyword to access member variables hidden by local variables.

- Five getters/accessors -
  - `public String getStudentName()`
  - `public int[] getTimeStamp()`
  - `public String getLocation()`
  - `public String getDescription()`
  - `public boolean isDebugging()`

Each getter method must simply return the corresponding `private` data field of the `Ticket` object, except for `getTimeStamp()`.This getter method, `public int[] getTimeStamp()`, is special in that it must return a new integer array that stores the same integers referenced to by the instance variable `timeStamp`. In other words, do not simply return the same instance reference variable. Instead, you must copy the elements of the array it refers to into the array referred to by the instance variable. Again, remember you must use the `this` keyword to access member variables hidden by local variables.

- Five setters/mutators -
    - `public void setStudentName(String studentName)`
    - `public void setTimeStamp(int[] timeStamp)`
    - `public void setLocation(String location)`
    - `public void setDescription(String description)`
    - `public void setDebugging(boolean debugging)`

Each setter method must simply set the corresponding instance variable to the one provided as an argument to the method. Except, `public void setTimeStamp()` must perform a **deep copy** of the array referred to by the method parameter array reference variable to the array referred to by the instance array reference variable. This means that you cannot simply set the `timeStamp` data field to refer to the same array being referred to by the method parameter. Instead, you must copy the elements of the array it refers to into the array referred to by the instance variable. Again, remember you must use the `this` keyword to access member variables hidden by local variables.

- `public int totalWait(int[] currentTime)`

This method will take the current time in the same format as the `timeStamp` (24-hour format) and must return the total **minutes** the ticket has been in the queue since `timeStamp`. Since we want the total number of minutes, you do not need to worry about the number of seconds. You may also assume that the `currentTime` is greater than `timeStamp`, and they are within the same day (within the same 24 hours).

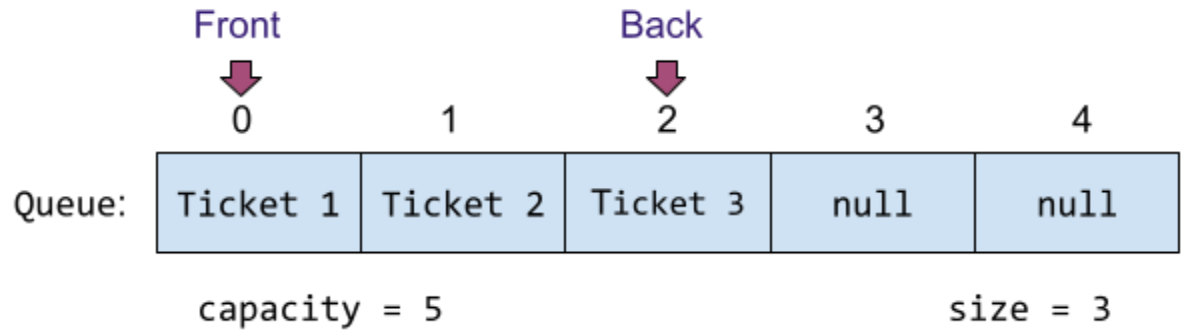
The methods that are provided in the starter code **MUST** return the correct output (if any) as that is how the Gradescope autograder will test your code.

Part 2: Implement Queue Class (40 points)

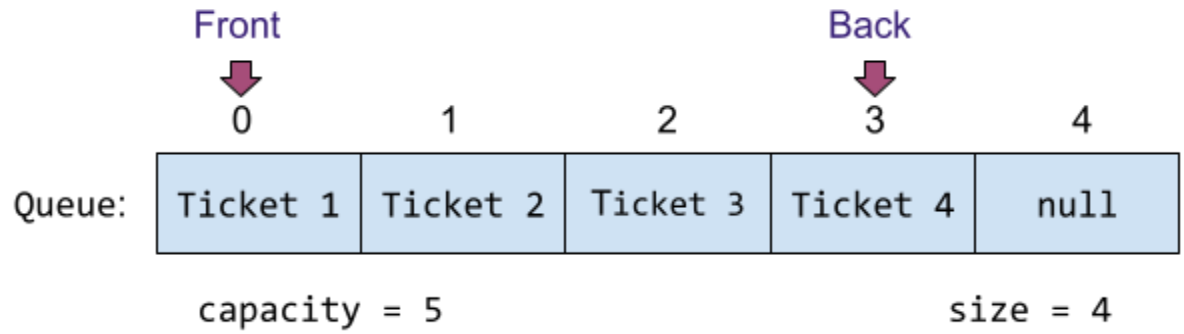
The Queue Class

The `Queue` class represents a first-in-first-out (FIFO) data structure for managing a collection of `Ticket` objects. FIFO order in this scenario means that the ticket that has an earlier timeStamp should be at the front of the queue, before other tickets.This class is implemented using an array to store objects of class `Ticket` in the explained FIFO order. To better understand FIFO in this scenarios, consider the following example:

Creating a `Queue` with `capacity` of 5, then `enqueue`ing three `Ticket` objects. Notice how the front of the `Queue` corresponds to index 0 and the back of the `Queue` is index 2.

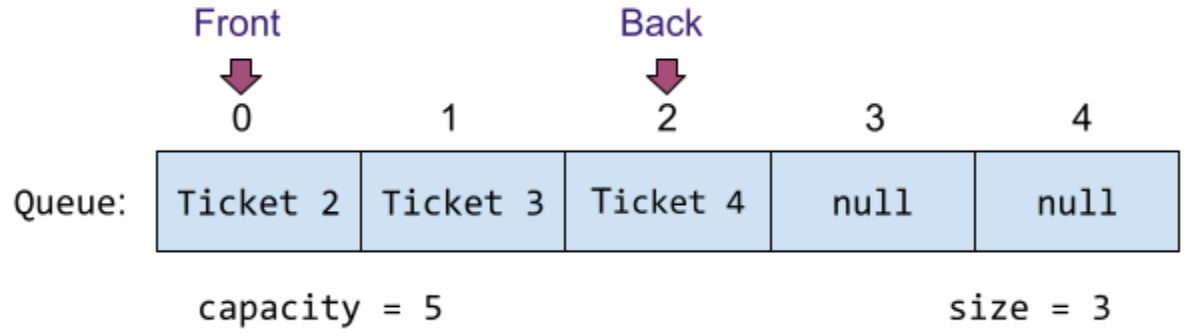


Now, if a fourth `Ticket` object is added using the `enqueue` method, notice that it is added to the **BACK** of our queue. And the `size` is updated, while `capacity` remains unchanged.



The `peek` method returns the **FRONT** `Ticket` from the `Queue` *without* deleting the `Ticket` from the `Queue`.

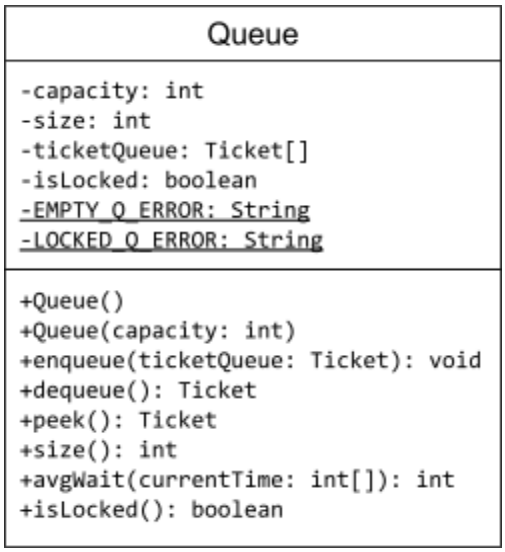
The `dequeue` method deletes the front `Ticket` from the `Queue`, shifts the remaining `Tickets` towards the front of the `Queue`, and updates the `size`. Which should look:



This is the exact same logic as you have seen on the course Autograder website.

Please note the figures above are only one representation of the array that is used to construct the first-in-first-out queue. Your implementation may differ and as long as your methods (as

described below) return the correct values; it does not matter how the internals of your `Queue` works. Following is the UML class diagram.



In `Queue.java`, you will be implementing **2 constructors and 6 instance methods.**

A `Queue` object contains the following fields:

- `private int capacity`: An integer representing the maximum capacity of the `Queue`.
- `private int size`: An integer representing the current number of `Tickets` in the `Queue`.
- `private Ticket[] ticketQueue`: An array of type `Ticket` with size `capacity` to store `Ticket` objects.
- `private boolean isLocked`: A boolean indicating whether the `Queue` is locked (i.e., full).

Two `static` class variables are also provided and are used when printing out an error when it occurs:

- `private static final String EMPTY_Q_ERROR`: printed when the user tries to `peek()` or `dequeue()` an empty `Queue`
- `private static final String LOCKED_Q_ERROR`: printed when the user tries to `add()` a `Ticket` but the `Queue` has reached the `capacity`.

Notice how each member field is declared `private`. This means the member is only visible *within* the class, not from any other class. **You must also use the `this` keyword to access member variables hidden by local variables.**

You are free to declare any additional instance or `static` member variables; however, if you do so, you must declare them as `private`.

The `Queue` class must contain the following member methods:

- `public Queue()`
  The no-arg constructor for the `Queue` class, which initializes `capacity` to `0`, `size` to `0`, `tickets` to `null`, and `isLocked` to `false`.

- `public Queue(int capacity)`
  This constructor takes the input parameter `capacity` to initialize the maximum size of the array of `Ticket`. For example, creating an object as `new Queue(5)` must create a `Queue` whose capacity is `5`. If the parameter passed is invalid (i.e., a negative number), then initialize your class variables `capacity` to `0`. Initialize `size` to `0`, `ticketQueue` to `null`, and `isLocked` to `false`.

- `public void enqueue(Ticket ticket)`
  A method to add a `Ticket` object, `ticket`, to the back of the `Queue`. If the `Queue` is already full or is `null`, then this method must print the contents of the `LOCKED_Q_ERROR` string and return. If upon adding a new `Ticket`, the `ticketQueue` gets full (i.e., reaches capacity), the `isLocked` flag must be set to `true`.

- `public Ticket dequeue()`
  The `dequeue` method removes and returns the `Ticket` object from the front of the queue (as described above), following the First-In-First-Out (FIFO) principle. It shifts the remaining tickets towards the left (front) in the array to maintain the order after dequeuing the first ticket. It also updates the `size` of the queue.
  If `ticketQueue` is empty or is `null`, then this method must print the contents of `EMPTY_Q` and return `null`. This method should also check if `isLocked` has been set to `true` and update it upon removing the ticket.

- `public Ticket peek()`
  A method that returns the front `Ticket` without removing it from the `Queue`. Since elements in `Queue` are maintained in FIFO order, this method must return the front element of the `Queue` (as shown earlier). If `Queue` is empty or it is `null`, then this method must print the contents of `EMPTY_Q` and return `null`.

- `public int size()`
  A method that returns the current size of the `Queue` based on how many tickets are currently in the `Queue`. For example, if a `Queue` with a `capacity` of `5` enqueues three `Tickets` followed by one `dequeue()` operation, then this method must return `2`.

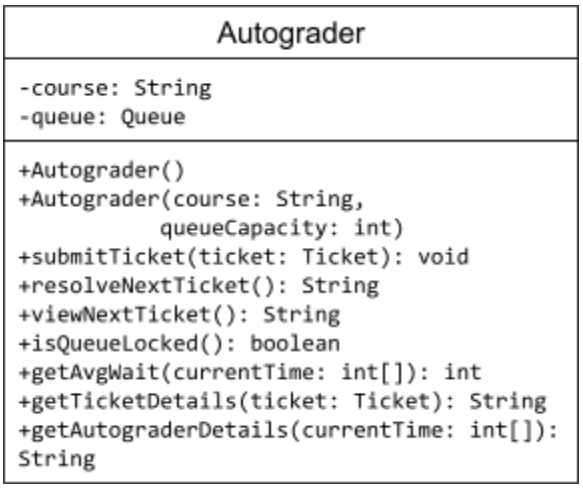
- `public int avgWait(int[] currentTime)`
  Calculates the average wait time for an incoming ticket by averaging the `totalWait()` of current `Tickets` in the `Queue`. If there are no tickets in the queue, return 0.

- `public boolean isLocked()`
  A method that simply returns the current flag(value) of `isLocked` variable of the `Queue` object.

The methods that are provided in the starter code **MUST** return the correct output (if any) as that is how the Gradescope autograder will test your code.

Part 3: Implement Autograder Class (20 points)

The `Autograder` class acts as a mediator between the user or other components of the system and the queue of tickets. It provides methods to interact with the queue, including accepting new tickets, resolving tickets, and viewing tickets without dequeuing. Following is the UML class diagram.



An `Autograder` object contains two fields:

- `private String course`: the name of the course for which the queue is managed
- `private Queue queue`: An instance of the `Queue` class used to manage the queue of tickets

The `Autograder` class must contain the following member methods:

- `public Autograder()`
  The no-arg constructor to initialize the `Autograder` object with an empty `queue` and course set to `null`.

- `public Autograder(String course, int queueCapacity)`
  This constructor of `Autograder` takes in the name of a `course` and its `queueCapacity`, and creates the `Autograder` object for it.

- `public void submitTicket(Ticket ticket)`
  Accepts a new `Ticket` object and enqueues it into the `queue` of this course.

- `public String resolveNextTicket()`
  Resolves the next ticket in the `queue` by dequeuing it and returning a string with the `Ticket` details. You must use the predefined method `getTicketDetails(Ticket ticket)` to return the ticket details.

- `public String viewNextTicket()`
  Returns the details of the next ticket in the `queue` without dequeuing it, as a `String`. You must use the predefined method `getTicketDetails(Ticket ticket)` to return the ticket details.

- `public boolean isQueueLocked()`
  Checks if the queue is full (i.e., has reached its capacity).

- `public int getAvgWait(int[] currentTime)`
  This method takes in `currentTime` in 24-hour format (explained earlier) and returns the average wait time of the queue.

This class has two more methods, `getTicketDetails(Ticket ticket)` and `getAutograderDetails(int[] currentTime)` which are already implemented for you. Please DO NOT change anything in the provided code or the Gradescope will fail. You must use the `getTicketDetails` method to return the `Ticket` details in `viewNextTicket()` and `resolveNextTicket()`.

Part 4: Compile, Run, and UnitTest Your Code (10 points)

Just like in previous assignments, **in this part of the assignment, you need to implement your own test cases in the method called** `unitTests` **in the** `Assignment5.java` **class.**

In the starter code, two test cases for `Ticket`, one simple test case for `Queue`, and one simple test case for `Autograder` have already been implemented for you. You can regard each of these as an example to implement other cases though you will need to test more to get full credit. To get full credit on this section, you **must** follow the instructions below.

For `Ticket`, you must create **three** additional test cases that test either the getter/setter methods, constructors, `totalWait()`, or a combination of these.

For `Queue`, you must create **three** additional test cases. Each test case must have all the following:

1. Invoke either one or both of the constructors
2. Have at least 3 enqueue operations
3. Have at least 3 dequeue operations
4. Test either the `peek()`, `size()`, `avgWait()`, or `isLocked()` method

For `Autograder`, you must create **three** additional test cases. Each test case must have all the following:

1. Invoke the constructor
2. invoke **each** of the 5 instance methods you implemented at least once

You cannot make all your test cases follow the same structure. For example, you cannot make all your test cases add three tickets then delete three tickets for both `Queue` and `Autograder` . Your best option is to test a **random combination** of the methods to see if it gives you the right output as that is what the autograder will do.

Submission

You're almost there! Please follow the instructions below carefully and use the **exact submission format**. Because we will use scripts to grade, **you may receive a zero** if you do not follow the same submission format.

1. Open Gradescope and login. Then, select this course → PA5.
2. Click the DRAG & DROP section and directly select the required files `Ticket.java`, `Queue.java`, `Autograder.java`, and `Assignment5.java`. Drag & drop is fine. Please make sure you **DO NOT** submit a zip, just the three files in one Gradescope submission. Make sure the names of the files are correct.
3. You can resubmit unlimited times before the due date. Your score will depend on your final (most recent) submission, even if your former submissions have higher scores.
4. Your submission should look like the below screenshot.