|  |  |
| --- | --- |
| **Item** | **Answer** |
| **Section** | G1 |
| **Name(s)** | Adeline Chin Wen Jie (Adeline)  Gokarn Mallika Nitin (Mallika) |
| **Brief description of project** | Our project is a restaurant simulator. The owner of Gluttons Bay Restaurant (fictional) is interested in looking at the difference between his company and the restaurant Subway, in terms of how much time it takes to service a certain number of customers.  Subway has a very linear process and attends to one customer fully at a time before moving onto the next customer. In reality Subway employs one person to execute work of both waiter and chef. For the sake of comparison, we have split the role of the waiter into waiter and chef. The process at Subway is as follows:  Gluttons Bay on the other hand is a Vegan 5 course restaurant that consists of Soup, Salad, Appetiser, Main Course and Dessert. Gluttons Bay provides seating as well, thereby requiring multiple customers to be serviced at a time. The process at Gluttons Bay is as follows:  In order to compare Subway and Gluttons Bay, a few assumptions have been made, they are as follows:   1. The food preparation overall, takes the same amount of time for chefs of both Gluttons Bay and Subway. The division of time on each part of preparation is as shown below:      1. The time taken by each customer on average overall, to decide on their food and thereafter let the waiter place the order is the same between both Gluttons Bay and Subway. The division of time on each part of ordering is as shown below:      1. The time taken by each waiter on average overall, to serve customers their food is the same between both Gluttons Bay and Subway. The division of time on each part of ordering is as shown below:      1. The time taken by each customer on average overall, to eat the food served has been *assumed* to be the same between both Gluttons Bay and Subway. *Additionally, we are looking at a high class Subway that provides seating and functions like a normal restaurant apart from the fact that they profess and follow self-service.* The division of time on each part of ordering is as shown below:      1. We are assuming that both Subway and Gluttons Bay have only one efficiently working waiter or chef at a time, that is to say, that to maximise efficiency and keep the time taken on preparation, serving etc. constant, the employees are employed in shifts that result in only 1 waiter and 1 chef being on duty at any given point in time. 2. As Gluttons Bay’s previous data shows that we cater to 15 customers on average per day, we will be using that as the upper-bound for our comparison.   One caveat of this comparison is that, on comparing the Single-threaded version and Multi-threaded version, there is no way to take into account the effect of the print statements and the time taken in switching between threads for multi-threaded. However, this is acceptable to the owner, as it can represent and account for the time taken by the waiter and chef to slack while walking and working continuously. |
| **Justification for multi-threading** | As shown in the process above, Gluttons Bay caters to multiple customers at a time and has the technology and ability to keep track of multiple orders thereby letting a single worker concentrate on doing the job dictated as the technology tells him to do it. This means that the waiter can wait on a customer and immediately after placing the order either wait on the next customer or serve the next order to the respective customer. To correctly represent the process, we would therefore require concurrency.  In subway on the other hand, a single waiter, can only remember and prepare a single order for a single customer at a given point in time. Therefore only allowing the process to run in a sequential and linear manner. |
| **Transactional integrity** | To implement concurrency in Gluttons Bay, we made use of BlockingQueue interface with implementations of ArrayBlockingQueue and LinkedBlockingDeque so as to ensure that the first order given to each actor (waiter, chef) is the one acted on.  The methods have been written with expansion in mind, in case the owner wishes to employ more than one chef and one waiter. This means that all the methods accessing the four queues are written with keeping integrity in mind given that in the future more than one thread can be accessing the queue.  One very important race condition that had to be met was on the LinkedBlockingDeque queue called customersServed. When a customer is served, by the waiter, the customer needs to ensure that the dish served is corresponding to what he/she ordered. This required the customer thread to both check the Dish, and return it to the queue if it is not meant for him/her. This required synchronization on the queue to ensure that no other customer thread is accessing the queue to remove the next dish on it.  We believe that all possible race conditions have been taken care of. |
| **Performance** | The Waiter thread has to both serve customers and take orders from customers. Having these two functions taking place concurrently, by the same thread, means that at some point the Waiter thread needs to switch between the serving and the taking of orders. Therefore, we made use of the  [**offer**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#offer(E,%20long,%20java.util.concurrent.TimeUnit))([**E**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) e, long timeout, [**TimeUnit**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TimeUnit.html) unit) and [**poll**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#poll(long,%20java.util.concurrent.TimeUnit))(long timeout, [**TimeUnit**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TimeUnit.html) unit) methods that made the Waiter thread wait for an update in the queue, that is the next order to be served/ taken for exactly 1 second each before switching to check the other queue. |
| **Evidence of exploration** | To implement concurrency in Gluttons Bay, as mentioned above, we made use of BlockingQueue interface with implementations of ArrayBlockingQueue and LinkedBlockingDeque. The BlockingQueue interface integrates the wait() and notifyAll() methods of a Thread thereby allowing us to not have to code extra lines for these two methods.  To implement ExecutorPool, we wanted to use the best implementation possible, therefore we did some extra research on the types of ExecutorPools available. Details of the same can be found in document titled Exploration.  We also researched on various ways that dependant threads can be killed, (more details available in the document titled Exploration). However we were unsuccessful in finding an implementation that would work in our case. To tackle this on line 164 of the Main class, we made use of System.exit(0); to allow the program to cleanly exit after completing all execution.  Lastly, we also researched some extra topics like CountDownLatch to see if there was a possibility of including it in our project. |
| **Innovation** | Why is your project innovative? |
| **Adherence to coding conventions & good practices** | Is there anything noteworthy that you want to mention? E.g. usage of a well-known multi-threading “best-practice”, or OO design pattern etc. |
| **References/Acknowledgement** | We used third party code from github user heidtJJ (Jared Heidt) as our starting point and built on it  The code to his project is in the folder titled reference.  We have also used the StopWatch |