LAB 13: RESTAURANT SIMULATION

SWE 4302
Object Oriented Concepts II

Hasin Mahtab
210042174
Department of CSE
B.Sc in Software Engineering

November 23, 2023

Contents

1 Introduction

Utilizing multiple threads in a Java program to run concurrent execution of tasks is Multi-Threading. There are different ways to use threads in Java. Here we will be extending the basic Thread class to access multiple threads.

2 Problem Statement

Implement a multithreaded restaurant simulation in Java. The simulation should involve customers placing orders, chefs preparing food, waiters serving dishes, and a receptionist managing table assignments.

3 Approach

3.1 Shared Queue

We will be trying to solve this problem using a **Producer-Consumer** pattern. There are shared data structures - **BlockingQueue**, which will allow Customers, Chefs and Waiters to access the shared queues for placed-orders and the prepared-foods. The **BlockingQueue** automatically handles operations as waiting for the queue to become non-empty while retrieving an element, and wait for space to become available in the queue when storing an element.

Both OrderQueue and CookedFoodQueue implement the SharedQueue interface and then we have methods to push and pop order items of the queues. They also print out when an order is placed and when a order is being processed by a chef. We have a Menu class that randomly assigns the number of dishes a customer will order from the list of dishes available.

3.2 Table Assignment

Next we have the Table class and the Receptionist class. The Receptionist also utilizes a **ArrayBlockingQueue** which helps to assign tables to an individual customer without data corruption.

The Receptionist calls the assignTable method from within the Customer object, which ensures the customer gets the exact table that is available in the sharedTableQueue.

3.3 Orders

The Customer class has a Shared Queue of the **orderQueue** in which he places order and the chef can pop order from that queue to prepare food, this ensures a stable and concrete data flow, as the Queue uses FIFO algorithm, the first customer order starts to prepare at first. Later the customer must wait for his food to be prepared. The execution timeline is being enforced in the run() method which is inherited from the Thread class, as the Customer class extends the Thread class.

The Chef class has a Shared Queue of the **orderQueue** from which he takes orders and has a Shared Queue of the **foodQueue** in which he places foods. The main function is to take orders from the shared **orderQueue** and prepare the foods for the customer. Once an order is prepared, the chef also pushes the food to the shared **foodQueue** for the waiter to serve the food. Also prints to the console when an order is being prepared and when its done preparing.

The Waiter class also extends from the Thread class and has a Shared Queue of the **foodQueue** from which he takes foods. The main function is to take foods from the shared **foodQueue** and serves the foods to the customers. Also prints to the console when an order is being served.

3.4 Driver Code

In the Main driver code, we initialize SharedQueues, the numberOfTables, numbOfChefs, waiter and Customers to serve. We initialize the Menu items. Lastly, we need a list of threads that will run all the inidividual threads for the Chefs, Waiter, Customers and run the threads separately.

Once all the threads are started, we can use the **Thread.Join()** method to join the termination priority of the threads and make sure that the order is maintained while placing orders, preparing foods and serving them.

4 Problems Faced

- Implementing the Shared queues were challenging for the Producer-Consumer pattern.
- Implementing the Join() method needed lots of online research.