G01

ZEYAD ALGHAMDI, EYAS MAJEED

G01

Multi-threaded Restaurant Simulation Report

CPCS-361 Group Project

Contents

[Introduction 2](#_Toc183880830)

[Code Explaination 2](#_Toc183880831)

[Customer.java 2](#_Toc183880832)

[CustomSemaphore.java 2](#_Toc183880833)

[Order.java 2](#_Toc183880834)

[OrderQueue.java 2](#_Toc183880835)

[Chef.java 2](#_Toc183880836)

[Waiter.java 2](#_Toc183880837)

[RestaurantSimulation.java 2](#_Toc183880838)

[CookedMeals.java 3](#_Toc183880839)

[TimeSimulation.java 3](#_Toc183880840)

[Table.java 3](#_Toc183880841)

[Synchronization 3](#_Toc183880842)

[Execution 3](#_Toc183880843)

[Test Results and Outputs 3](#_Toc183880844)

[Output for input file 1 4](#_Toc183880845)

[Output for input file 2 5](#_Toc183880846)

[Output for input file 3 6](#_Toc183880847)

[Codes 7](#_Toc183880848)

[Resturant Simulation Class (Main) 7](#_Toc183880849)

[Cutomer Class 9](#_Toc183880850)

[Chef Class 10](#_Toc183880851)

[Cooked Meals Class 11](#_Toc183880852)

[Custom Semaphore Class 12](#_Toc183880853)

[Order Class 13](#_Toc183880854)

[Order Queue Class 14](#_Toc183880855)

[Table Class 15](#_Toc183880856)

[Time Simulation Class 16](#_Toc183880857)

[Waiter Class 17](#_Toc183880858)

[Conclusion 17](#_Toc183880859)

# Introduction

This report documents the implementation of a restaurant simulation system designed to showcase multithreading and synchronization mechanisms. The simulation involves customers placing orders, chefs preparing meals, waiters serving dishes, and table management using semaphores. The system demonstrates key concepts of operating systems, such as concurrency, synchronization, and resource allocation.

# Code Explaination

### Customer.java

The Customer class represents a customer in the simulation. Customers arrive at a specific time, acquire a table using a semaphore, place an order, and wait until their meal is served. Synchronization ensures that customers only proceed when their conditions are met.

### CustomSemaphore.java

The CustomSemaphore class implements a custom semaphore mechanism to manage table availability. It allows customers to acquire and release tables in a thread-safe manner.

### Order.java

The Order class encapsulates details about a customer's order, including the customer's ID, the ordered item, and the preparation time.

### OrderQueue.java

The OrderQueue class manages a priority queue of customer orders. Orders are inserted based on their preparation times to ensure efficient processing.

### Chef.java

The Chef class represents a chef in the restaurant. Chefs process orders from the queue, prepare meals, and add them to the cooked meals queue. This is done in a separate thread.

### Waiter.java

The Waiter class represents a waiter responsible for retrieving cooked meals and serving them to the respective customers. Waiters operate in their own threads and ensure timely delivery of meals.

### RestaurantSimulation.java

This class orchestrates the entire simulation. It initializes the system with chefs, waiters, tables, and customers based on a configuration file. The simulation also tracks metrics like average wait times, total customers served, and simulation duration.

### CookedMeals.java

The CookedMeals class manages a synchronized queue of cooked meals, ensuring thread-safe retrieval and addition of meals.

### TimeSimulation.java

This class simulates the passage of time in the restaurant. It increments the simulation time every second in a separate thread.

### Table.java

The Table class represents a table in the restaurant. Each table is identified by a unique ID.

## Synchronization

Synchronization is achieved through thread-safe methods and custom semaphores. Key synchronization mechanisms include:  
- CustomSemaphore for managing table availability.  
- Synchronized blocks and methods for managing order queues and cooked meals.  
- Thread communication via wait() and notify().

# Execution

The simulation begins with initializing the system based on configuration parameters, such as the number of chefs, waiters, and tables. Threads are created for each entity, including customers, chefs, waiters, and the time simulation. Customers are served in a time-based sequence, with orders processed and meals delivered efficiently.

# Test Results and Outputs

Sample outputs include:  
- Customer arrival, table allocation, and order placement messages.  
- Chef preparation start and end times.  
- Waiter serving messages.  
- Simulation summary, including average wait times and total customers served.

Output for input file 1  
A screenshot of a computer

Description automatically generatedA screen shot of a computer

Description automatically generated

## Output for input file 2

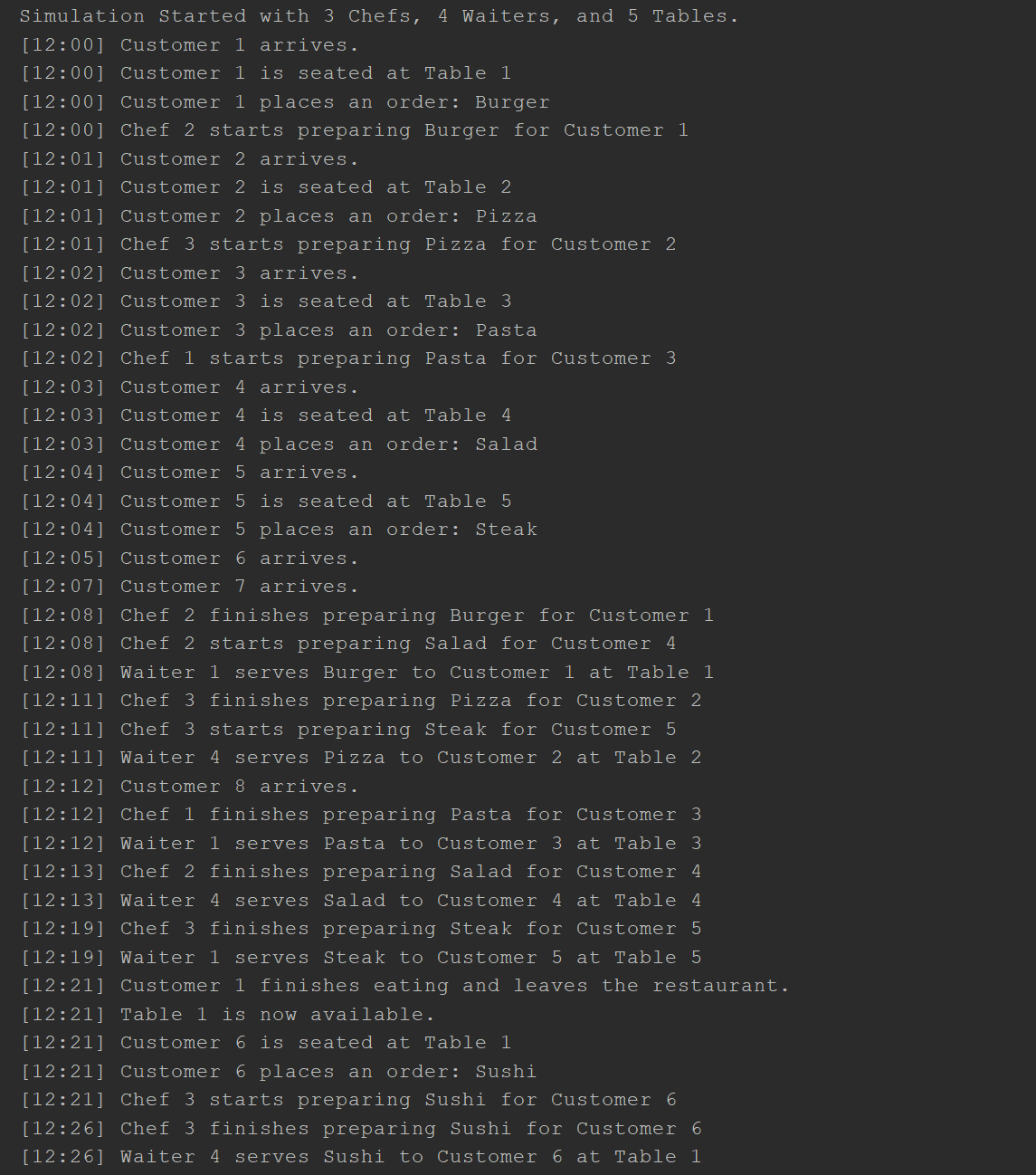
A screenshot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

## Output for input file 3



A screenshot of a computer program

Description automatically generated

# Codes

## Resturant Simulation Class (Main)

import java.io.\*;

import java.util.\*;

/\*\*

 \* Names: Zeyad Alghamdi 2237000 - Eyas Majeed   2236567

 \* Project: Multi-threaded Restaurant Simulation

 \* Compiler: JDK 18.0.1.1

 \* Hardware Configuration:

 \*   Processor: AMD Ryzen 5 4600H

 \*   RAM: 8 GB

 \*   System Type: 64-bit

 \* Operating System: Windows 11 Home

 \*/

/\*\*

 \* The RestaurantSimulation class simulates the operations of a restaurant, managing chefs,

 \* waiters, tables, and customers, and executing a time-based simulation for order preparation and serving.

 \*/

public class RestaurantSimulation {

    // Current time in the simulation, represented in minutes since midnight

    public static int currentTime;

    // Queue to manage incoming orders

    public static OrderQueue orderQueue = new OrderQueue();

    // Object to manage prepared meals

    public static CookedMeals cookedMeals = new CookedMeals();

    // Map to keep track of customer information with customer ID as the key

    public static HashMap<Integer, Customer> customers = new HashMap<>();

    // Menu map with meal names as keys and preparation times (in minutes) as values

    public static HashMap<String, Integer> menuMap = new HashMap<>();

    // Counter for the number of active customers currently in the simulation

    public static int activeCustomerCount = 0;

    // Total number of customers served during the simulation

    public static int totalCustomersServed = 0;

    // Total wait time accumulated by all customers

    public static double totalWaitTime = 0;

    // Total time taken to prepare all orders

    public static double totalOrderPreparationTime = 0;

    // Total duration of the simulation in minutes

    public static int simulationTime;

    // Start time of the simulation in minutes since midnight

    public static int simulationStartTime;

    public static void main(String[] args) throws Exception {

        // File name containing simulation configuration and data

        String inputFile1 = "restaurant\_simulation\_input3";

        // Create a BufferedReader to read the input file

        BufferedReader read = new BufferedReader(new FileReader(inputFile1));

        // Read the first line (header) and parse the configuration values

        String configLine = read.readLine();

        String[] configArray = configLine.split(" ");

        HashMap<String, Integer> configMap = new HashMap<>();

        for (String part : configArray) {

            String[] keyAndValue = part.split("=");

            configMap.put(keyAndValue[0], Integer.parseInt(keyAndValue[1]));

        }

        // Extract the number of chefs, waiters, and tables from the configuration

        int numChefs = configMap.get("NC");

        int numWaiters = configMap.get("NW");

        int numTables = configMap.get("NT");

        // Initialize and create threads for chefs

        ArrayList<Thread> chefThreads = new ArrayList<>();

        Chef[] chefs = new Chef[numChefs];

        for (int i = 0; i < numChefs; i++) {

            chefs[i] = new Chef(i + 1);

            chefThreads.add(new Thread(chefs[i]));

        }

        // Initialize and create threads for waiters

        ArrayList<Thread> waiterThreads = new ArrayList<>();

        Waiter[] waiters = new Waiter[numWaiters];

        for (int i = 0; i < numWaiters; i++) {

            waiters[i] = new Waiter(i + 1);

            waiterThreads.add(new Thread(waiters[i]));

        }

        // Initialize tables for customers

        Table[] tables = new Table[numTables];

        for (int i = 0; i < numTables; i++) {

            tables[i] = new Table(i + 1);

        }

        // Create a custom semaphore to manage table availability

        CustomSemaphore tableSemaphore = new CustomSemaphore(numTables);

        // Read and parse the menu items and their preparation times

        String menuLine = read.readLine();

        String[] menuArray = menuLine.split(" ");

        for (String meal : menuArray) {

            String[] nameAndTime = meal.split("=");

            menuMap.put(nameAndTime[0], timeToMinutes(nameAndTime[1]));

        }

        // Read customer details and schedule their arrival

        ArrayList<Thread> customerThreads = new ArrayList<>();

        int leastArrivalTime = Integer.MAX\_VALUE;  // Variable to keep track of the earliest arrival time

        String line;

        while ((line = read.readLine()) != null) {

            String[] parts = line.split(" ");

            int customerID = Integer.parseInt(parts[0].split("=")[1]);

            int arrivalTime = timeToMinutes(parts[1].split("=")[1]);

            String orderItem = parts[2].split("=")[1];

            // Create a new customer and store in the map

            Customer customer = new Customer(customerID, orderItem, arrivalTime, tableSemaphore);

            customers.put(customerID, customer);

            customerThreads.add(new Thread(customer));

            // Update the earliest arrival time

            if (arrivalTime < leastArrivalTime) {

                leastArrivalTime = arrivalTime;

            }

        }

        read.close();

        // Set the current simulation time to the earliest customer arrival time

        currentTime = leastArrivalTime;

        simulationStartTime = currentTime;

        // Start the time simulation thread

        Thread timeSimulationThread = new Thread(new TimeSimulation());

        timeSimulationThread.start();

        System.out.println("Simulation Started with " + numChefs + " Chefs, " + numWaiters + " Waiters, and " + numTables + " Tables.");

        // Start all customer, chef, and waiter threads

        for (Thread thread : customerThreads) {

            activeCustomerCount++;

            thread.start();

        }

        for (Thread thread : chefThreads) {

            thread.start();

        }

        for (Thread thread : waiterThreads) {

            thread.start();

        }

        // Wait for all customer, chef, and waiter threads to finish

        for (Thread thread : customerThreads) {

            thread.join();

        }

        for (Thread thread : chefThreads) {

            thread.join();

        }

        for (Thread thread : waiterThreads) {

            thread.join();

        }

        // Wait for the time simulation thread to complete

        timeSimulationThread.join();

    }

    /\*\*

     \* Converts time in "HH:MM" format to total minutes since midnight.

     \* @param time The time string in "HH:MM" format.

     \* @return Total minutes since midnight.

     \*/

    public static int timeToMinutes(String time) {

        String[] parts = time.split(":");

        int hours = Integer.parseInt(parts[0]);

        int minutes = Integer.parseInt(parts[1]);

        return hours \* 60 + minutes;

    }

    /\*\*

     \* Converts total minutes since midnight to "HH:MM" format.

     \* @param minutes The total minutes since midnight.

     \* @return The time string in "HH:MM" format.

     \*/

    public static String minutesToTime(int minutes) {

        int hours = minutes / 60;

        int mins = minutes % 60;

        return String.format("%02d:%02d", hours, mins);

    }

    /\*\*

     \* Called when a customer has finished their visit, updating the active customer count.

     \* If no more active customers, the simulation ends.

     \*/

    public static synchronized void customerFinished() {

        activeCustomerCount--;

        totalCustomersServed++;

        if (activeCustomerCount == 0) {

            System.out.println("[End of Simulation]");

            simulationTime = currentTime - simulationStartTime;

            displaySummary();

            System.exit(0);

        }

    }

    /\*\*

     \* Displays a summary of the simulation including total customers served,

     \* average wait time, and total simulation duration.

     \*/

    public static void displaySummary() {

        System.out.println("\nSummary:");

        System.out.println("Total Customers Served: " + totalCustomersServed);

        System.out.println("Average Wait Time for Table: " + (totalWaitTime / totalCustomersServed) + " Minutes");

        System.out.println("Average Order Preparation Time: " + (totalOrderPreparationTime / totalCustomersServed) + " Minutes");

        System.out.println("Total Simulation Time: " + simulationTime + " Minutes");

    }

}

## Cutomer Class

import java.time.LocalTime;

/\*\*

 \* The Customer class represents a customer in the restaurant simulation who places an order,

 \* waits for a table, and eats the meal. This class implements the Runnable interface for

 \* running in a separate thread.

 \*/

public class Customer implements Runnable {

    // Unique ID for the customer

    private int customerID;

    // Item the customer orders from the menu

    private String orderItem;

    // Time (in minutes since midnight) the customer arrives at the restaurant

    private int arrivalTime;

    // Semaphore to manage table availability

    private CustomSemaphore tableSemaphore;

    // Table number assigned to the customer

    private int tableNumber;

    /\*\*

     \* Constructor to initialize a Customer object.

     \* @param customerID The unique ID for the customer.

     \* @param orderItem The item the customer orders.

     \* @param arrivalTime The time the customer arrives, in minutes since midnight.

     \* @param tableSemaphore The custom semaphore managing table availability.

     \* @throws Exception If there is an issue during initialization.

     \*/

    public Customer(int customerID, String orderItem, int arrivalTime, CustomSemaphore tableSemaphore) throws Exception {

        this.customerID = customerID;

        this.orderItem = orderItem;

        this.arrivalTime = arrivalTime;

        this.tableSemaphore = tableSemaphore;

    }

    /\*\*

     \* The run method defines the behavior of the customer thread.

     \* The customer waits until their arrival time, gets seated at a table,

     \* places an order, waits for the meal, eats the meal, and then leaves the restaurant.

     \*/

    public synchronized void run() {

        try {

            // Wait until the current simulation time matches the customer's arrival time

            while (getCurrentTime() < arrivalTime) {

                Thread.sleep(500); // Sleep for a short duration and then recheck

            }

            // Print a message when the customer arrives

            System.out.println("[" + RestaurantSimulation.minutesToTime(getCurrentTime()) + "] Customer " + this.customerID + " arrives.");

            // Acquire a table using the semaphore and print a message

            this.tableNumber = tableSemaphore.acquire();

            Thread.sleep(250); // Short pause to simulate time taken to get seated

            System.out.println("[" + RestaurantSimulation.minutesToTime(getCurrentTime()) + "] Customer " + this.customerID + " is seated at Table " + this.tableNumber);

            // Calculate and add the wait time to the total wait time in the simulation

            RestaurantSimulation.totalWaitTime += getCurrentTime() - getArrivalTime();

            // Place the order and add it to the order queue

            RestaurantSimulation.orderQueue.addOrder(new Order(customerID, orderItem, RestaurantSimulation.menuMap.get(orderItem)));

            System.out.println("[" + RestaurantSimulation.minutesToTime(getCurrentTime()) + "] Customer " + this.customerID + " places an order: " + this.orderItem);

            // Wait until the meal is served

            synchronized (this) {

                wait();

            }

            // Simulate eating the meal for a random duration between 10 and 20 seconds

            int randomTime = 10 + (int) (Math.random() \* 11);

            Thread.sleep(randomTime \* 1000);

            // Release the table and print messages when the customer finishes eating and leaves

            tableSemaphore.release(this.tableNumber);

            System.out.println("[" + RestaurantSimulation.minutesToTime(getCurrentTime()) + "] Customer " + this.customerID + " finishes eating and leaves the restaurant.");

            System.out.println("[" + RestaurantSimulation.minutesToTime(getCurrentTime()) + "] Table " + this.tableNumber + " is now available.");

            // Notify the simulation that the customer has finished

            RestaurantSimulation.customerFinished();

        } catch (Exception e) {

            // Print an error message if something goes wrong

            System.out.println("Something is wrong: " + e.getMessage());

        }

    }

    /\*\*

     \* Method to notify the customer when their meal is ready.

     \* Wakes up the customer thread from waiting.

     \*/

    public synchronized void receiveMeal() {

        synchronized (this) {

            notify();

        }

    }

    // Getter for the customer ID

    public int getCustomerID() {

        return customerID;

    }

    // Getter for the order item

    public String getOrderItem() {

        return orderItem;

    }

    // Getter for the arrival time

    public int getArrivalTime() {

        return arrivalTime;

    }

    // Getter for the table number

    public int getTableID() {

        return tableNumber;

    }

    /\*\*

     \* Method to get the current simulation time.

     \* @return Current time in the simulation, in minutes since midnight.

     \*/

    private int getCurrentTime() {

        return RestaurantSimulation.currentTime;

    }

}

## Chef Class

import java.util.\*;

/\*\*

 \* The Chef class represents a chef in the restaurant simulation.

 \* Each chef runs in a separate thread, processing orders and preparing meals.

 \*/

public class Chef implements Runnable {

    // Unique ID for the chef

    private int chefID;

    /\*\*

     \* Constructor to initialize a Chef object with a unique ID.

     \* @param chefID The unique ID for the chef.

     \*/

    public Chef(int chefID) {

        this.chefID = chefID;

    }

    /\*\*

     \* The run method defines the behavior of the chef thread.

     \* The chef retrieves orders from the queue, prepares meals, and adds them to the cooked meals queue.

     \*/

    public void run() {

        try {

            while (true) {

                // Retrieve an order from the queue

                Order order = RestaurantSimulation.orderQueue.takeOrder();

                System.out.println("[" + RestaurantSimulation.minutesToTime(getCurrentTime()) + "] Chef " + chefID +

                        " starts preparing " + order.getOrderItem() + " for Customer " + order.getCustomerID());

                // Record the preparation start time

                int startTime = getCurrentTime();

                // Simulate preparation time

                Thread.sleep(order.getOrderTime() \* 1000);

                // Update the total preparation time

                RestaurantSimulation.totalOrderPreparationTime += getCurrentTime() - startTime;

                System.out.println("[" + RestaurantSimulation.minutesToTime(getCurrentTime()) + "] Chef " + chefID +

                        " finishes preparing " + order.getOrderItem() + " for Customer " + order.getCustomerID());

                // Add the prepared meal to the cooked meals queue

                RestaurantSimulation.cookedMeals.addMeal(order);

            }

        } catch (Exception e) {

            System.out.println("An error occurred in Chef " + chefID + ": " + e.getMessage());

        }

    }

    /\*\*

     \* Retrieves the current simulation time.

     \* @return The current time in the simulation, in minutes since midnight.

     \*/

    private int getCurrentTime() {

        return RestaurantSimulation.currentTime;

    }

}

## Cooked Meals Class

import java.util.ArrayList;

/\*\*

 \* The CookedMeals class manages a queue of cooked meal orders.

 \* This class uses a CustomSemaphore to manage synchronization between chefs and waiters.

 \*/

public class CookedMeals {

    // List to store cooked meals

    private ArrayList<Order> cookedMeals;

    // Semaphore to manage the number of cooked meals available

    private CustomSemaphore mealsSemaphore;

    /\*\*

     \* Constructor to initialize the CookedMeals queue.

     \* Initializes an empty list and a semaphore with zero permits.

     \*/

    public CookedMeals() {

        cookedMeals = new ArrayList<>();

        mealsSemaphore = new CustomSemaphore(0); // Initially, no meals are available

    }

    /\*\*

     \* Retrieves and removes a cooked meal from the queue.

     \* Blocks if no meals are available until a chef adds one.

     \* @return The cooked meal order.

     \* @throws InterruptedException If the thread is interrupted while waiting.

     \*/

    public Order getMeal() throws InterruptedException {

        mealsSemaphore.acquire(); // Wait until a meal is available

        synchronized (this) {

            return cookedMeals.remove(0);

        }

    }

    /\*\*

     \* Adds a cooked meal to the queue and notifies waiters.

     \* @param meal The cooked meal to be added to the queue.

     \*/

    public void addMeal(Order meal) {

        synchronized (this) {

            cookedMeals.add(meal);

        }

        mealsSemaphore.release(meal.getCustomerID()); // Signal that a meal is available

    }

}

## Custom Semaphore Class

import java.util.LinkedList;

import java.util.Queue;

/\*\*

 \* The CustomSemaphore class is a custom implementation of a semaphore

 \* used to manage the availability of shared resources, such as tables.

 \*/

public class CustomSemaphore {

    // Number of available permits (resources)

    private int availablePermits;

    // Queue to store resource identifiers (e.g., table numbers)

    private Queue<Integer> resourceQueue;

    /\*\*

     \* Constructor to initialize the semaphore with a specific number of permits.

     \* Each permit represents a resource that can be acquired or released.

     \* @param permits The number of available resources.

     \*/

    public CustomSemaphore(int permits) {

        this.availablePermits = permits;

        this.resourceQueue = new LinkedList<>();

        // Initialize the resource queue with identifiers (e.g., table numbers)

        for (int i = 1; i <= permits; i++) {

            resourceQueue.add(i);

        }

    }

    /\*\*

     \* Acquires a resource from the semaphore.

     \* Blocks if no resources are available until another thread releases one.

     \* @return The identifier of the acquired resource.

     \* @throws InterruptedException If the thread is interrupted while waiting.

     \*/

    public synchronized int acquire() throws InterruptedException {

        // Wait until a resource becomes available

        while (availablePermits <= 0) {

            wait();

        }

        // Decrement the available permits and return the acquired resource

        availablePermits--;

        return resourceQueue.poll(); // Remove and return the resource

    }

    /\*\*

     \* Releases a resource back to the semaphore.

     \* Increases the number of available permits and notifies waiting threads.

     \* @param resource The identifier of the resource to be released.

     \*/

    public synchronized void release(int resource) {

        // Increment the available permits and add the resource back to the queue

        availablePermits++;

        resourceQueue.add(resource);

        // Notify one of the waiting threads

        notify();

    }

    /\*\*

     \* Gets the number of currently available permits (resources).

     \* @return The number of available permits.

     \*/

    public synchronized int getAvailablePermits() {

        return availablePermits;

    }

    /\*\*

     \* Checks whether the semaphore has available resources.

     \* @return True if there are available permits, false otherwise.

     \*/

    public synchronized boolean hasAvailablePermits() {

        return availablePermits > 0;

    }

}

## Order Class

/\*\*

 \* The Order class represents a customer's order in the restaurant simulation.

 \* Each order includes details about the customer, the item ordered, and the preparation time.

 \*/

public class Order {

    // ID of the customer who placed the order

    private int customerID;

    // The item ordered by the customer

    private String orderItem;

    // Time required to prepare the order, in seconds

    private int orderTime;

    /\*\*

     \* Constructor to initialize an Order object with the customer ID, order item, and order time.

     \* @param customerID The ID of the customer who placed the order.

     \* @param orderItem The name of the item ordered.

     \* @param orderTime The time needed to prepare the order, in seconds.

     \*/

    public Order(int customerID, String orderItem, int orderTime) {

        this.customerID = customerID;

        this.orderItem = orderItem;

        this.orderTime = orderTime;

    }

    /\*\*

     \* Getter method to retrieve the customer ID.

     \* @return The ID of the customer who placed the order.

     \*/

    public int getCustomerID() {

        return customerID;

    }

    /\*\*

     \* Getter method to retrieve the order item.

     \* @return The name of the item ordered.

     \*/

    public String getOrderItem() {

        return orderItem;

    }

    /\*\*

     \* Getter method to retrieve the order time.

     \* @return The time needed to prepare the order, in seconds.

     \*/

    public int getOrderTime() {

        return orderTime;

    }

}

## Order Queue Class

import java.util.ArrayList;

public class OrderQueue {

    private ArrayList<Order> orderQueue;

    private CustomSemaphore ordersSemaphore;

    public OrderQueue() {

        orderQueue = new ArrayList<>();

        ordersSemaphore = new CustomSemaphore(0); // Initially, no orders are available

    }

    /\*\*

     \* Adds an order to the queue in the correct position based on order time.

     \* Uses binary insertion for efficiency.

     \* @param order The order to be added.

     \*/

    public synchronized void addOrder(Order order) {

        // If the queue is empty, directly add the order

        if (orderQueue.isEmpty()) {

            orderQueue.add(order);

        } else {

            // Insert the order at the correct position to maintain the order based on preparation time

            int position = findInsertPosition(order.getOrderTime());

            orderQueue.add(position, order);

        }

        // Signal that a new order is available

        ordersSemaphore.release(order.getCustomerID());

    }

    /\*\*

     \* Retrieves and removes the first order in the queue.

     \* Blocks if the queue is empty until an order is available.

     \* @return The first order in the queue.

     \* @throws InterruptedException If the thread is interrupted while waiting.

     \*/

    public Order takeOrder() throws InterruptedException {

        ordersSemaphore.acquire(); // Wait until an order is available

        synchronized (this) {

            return orderQueue.remove(0);

        }

    }

    /\*\*

     \* Checks if the queue is empty.

     \* @return True if the queue is empty, false otherwise.

     \*/

    public synchronized boolean isEmpty() {

        return orderQueue.isEmpty();

    }

    /\*\*

     \* Helper method to find the correct position to insert an order based on its order time.

     \* Performs a linear search to find the appropriate position.

     \* @param orderTime The order time of the new order.

     \* @return The position to insert the new order.

     \*/

    private int findInsertPosition(int orderTime) {

        for (int i = 0; i < orderQueue.size(); i++) {

            if (orderTime < orderQueue.get(i).getOrderTime()) {

                return i;

            }

        }

        return orderQueue.size(); // Insert at the end if no smaller orderTime is found

    }

}

## Table Class

/\*\*

 \* The Table class represents a table in the restaurant.

 \* Each table has a unique ID and is used to seat customers.

 \*/

public class Table {

    // Unique ID for the table

    private int tableID;

    /\*\*

     \* Constructor to initialize a Table object with a specific ID.

     \* @param tableID The unique ID for the table.

     \*/

    public Table(int tableID) {

        this.tableID = tableID;

    }

    /\*\*

     \* Getter method to retrieve the table ID.

     \* @return The ID of the table.

     \*/

    public int getTableID() {

        return tableID;

    }

}

## Time Simulation Class

/\*\*

 \* The TimeSimulation class is responsible for advancing the simulation time.

 \* It runs in a separate thread and increments the simulation time every second.

 \*/

public class TimeSimulation implements Runnable {

    /\*\*

     \* The run method defines the behavior of the time simulation thread.

     \* It continuously increments the current time in the simulation every second.

     \*/

    public void run() {

        try {

            while (true) {

                // Sleep for 1000 milliseconds (1 second) to simulate time passing

                Thread.sleep(1000);

                // Increment the current time in the simulation

                RestaurantSimulation.currentTime++;

            }

        } catch (InterruptedException e) {

            // Handle interruptions by printing the stack trace

            e.printStackTrace();

        }

    }

}

## Waiter Class

/\*\*

 \* The Waiter class represents a waiter in the restaurant simulation.

 \* Each waiter runs in a separate thread, serving meals to customers.

 \*/

public class Waiter implements Runnable {

    // Unique ID for the waiter

    private int waiterID;

    /\*\*

     \* Constructor to initialize a Waiter object with a unique ID.

     \* @param waiterID The unique ID for the waiter.

     \*/

    public Waiter(int waiterID) {

        this.waiterID = waiterID;

    }

    /\*\*

     \* The run method defines the behavior of the waiter thread.

     \* The waiter retrieves cooked meals from the queue and serves them to customers.

     \*/

    public void run() {

        try {

            while (true) {

                // Retrieve a cooked meal from the queue

                Order meal = RestaurantSimulation.cookedMeals.getMeal();

                // Find the customer associated with the order

                Customer customer = RestaurantSimulation.customers.get(meal.getCustomerID());

                // Notify the customer that their meal has been served

                customer.receiveMeal();

                System.out.println("[" + RestaurantSimulation.minutesToTime(getCurrentTime()) + "] Waiter " + waiterID +

                        " serves " + meal.getOrderItem() + " to Customer " + customer.getCustomerID() + " at Table " + customer.getTableID());

            }

        } catch (Exception e) {

            System.out.println("An error occurred in Waiter " + waiterID + ": " + e.getMessage());

        }

    }

    /\*\*

     \* Retrieves the current simulation time.

     \* @return The current time in the simulation, in minutes since midnight.

     \*/

    private int getCurrentTime() {

        return RestaurantSimulation.currentTime;

    }

}

# Conclusion

This simulation successfully demonstrates the use of multithreading and synchronization to model a real-world system. The project highlights the importance of efficient resource management and thread communication in achieving concurrency.