Informatics Institute Of Technology

Software Development II

4COSC010C.3

Course Work – 02

Name	**-**	K.G.N.S. Dharmapriya
UoW Number	**-**	20015075
IIT Number	**-**	20221623

Table of Contents

AKNOLADGEMENT

I would like to express my sincere gratitude and appreciation to my lecturer and the other module leaders for giving me the opportunity to work on this project, which also let me conduct in-depth research and learn a lot. I owe them a great deal for their insightful advice. Those who assisted me in my endeavor have my sincere gratitude.

TABLE OF TEST-CASES

Test Case	Expected Result	Actual Result	Pass / Fail
Food Queue initialized correctly after program start, 100 or VFQ	Display ‘empty’ / ‘X’ for all queues	Display ‘empty’ for all queue	Pass
Food Queue initialized correctly after program start, 101 or VEQ	Display all empty slots	Display all empty slots	Pass
Food Queue initialized correctly after program start, 102 or ACQ	Add the customer to the first queue first row	Add the customer to the first queue first row	pass
After adding first customer, 102 or ACQ	Customer add to the second queue first row	Customer added to the first queue second row	Fail
After fixing above error, adding a customer	Add the customer to the queue which has minimum length	Add the customer to the queue which has minimum length	Pass
103 or RCQ	Remove customer from we selected specific location	Do not remove customer from our selected location	Fail
After filling all locations in queues, add customers to the waiting list	Add customers properly to the waiting list	There was an error	Fail
After remove a served customer the queue customers’ position become forward	Become forward automatically	Become forward automatically	Pass
109 or ABS	If Stock is not full, can add burgers	Display ‘Stock is full !’	Fail
After remove served customer, 110 or IFQ	can select a queue and get income of each queue	can select a queue and get income of each queue	Pass
106 or SPD	Show entered all customers	Show entered all customers	Pass
107 or LPD	Load data from text file to program	Load data from text file to program	Pass
105 or SPD	Sored entered customers list	Sored entered customers list	Pass
999 or EXT	Exit from program	Exit from program	Pass

DISCUSSION

The test cases selected for the program aimed to cover various aspects and functionalities to ensure comprehensive testing. Each test case was designed to evaluate a specific feature or scenario. For example, there were test cases to verify the correct initialization of the food queues, displaying 'empty' or 'X' for all queues. Another set of test cases focused on adding customers to the queues, checking if they were added to the correct locations, and whether the customers in the queue were shifted forward appropriately after removal. Additional test cases assessed the program's ability to handle waiting lists, detect stock limitations, calculate income for each queue, and perform file operations such as storing and loading data. The chosen test cases aimed to cover normal cases, boundary cases, error conditions, and special cases, ensuring that different aspects of the program were thoroughly tested.

CODE

Task 01 (Array Version)

```java
import java.io.File;    // Import the File class from the java.io package
import java.io.IOException;   // Import the IOException class from the
java.io package
import java.util.*;   // Import all classes from the java.util package
import java.io.*;   // Import all classes from the java.io package

public class FoodiesFaveFoodcenter
{
    public static String[][] queues = new String[3][];  // 2D Array to
store the queues
    public static int[] maxCapacity = {2, 3, 5};  // Maximum capacity for
each queue



    private static int stock = 50;  // Initial stock of burgers

    public static Scanner userInput = new Scanner((System.in));   //user
input method

    public static void main(String[] args) {

        try{
            File file = new File("Text.txt");  // to store data create a
file
            file.createNewFile();

        }
        catch (IOException ioe){
            System.out.println();
        }

        queues[0] = new String[maxCapacity[0]];
        queues[1] = new String[maxCapacity[1]];
        queues[2] = new String[maxCapacity[2]];

        String[] queue1 = queues[0];
        String[] queue2 = queues[1];
        String[] queue3 = queues[2];

        Scanner userInput = new Scanner(System.in);
        int choice;

        do {
            displayMenu();   // Display the menu options
            choice = userInput.nextInt();
            userInput.nextLine();

            switch (choice) {
                case 100:
                    viewAllQueues(queue1,queue2,queue3);
                    break;
                case 101:
                    viewAllEmptyQueues(queue1);
                    viewAllEmptyQueues(queue2);
                    viewAllEmptyQueues(queue3);
                    break;
                case 102:
                    addCustomer(queue1,queue2,queue3);
                    break;
                case 103:
                    removeCustomer();
                    break;
                case 104:
                    removeServedCustomer();
                    break;
                case 105:
                    viewCustomersSorted();
                    break;
                case 106:
                    storeProgramData(queue1);
                    storeProgramData(queue2);
                    storeProgramData(queue3);
                    break;
                case 107:
                    loadProgramData();
                    break;
                case 108:
                    viewRemainingStock();
                    break;
                case 109:
                    addBurgersToStock();
                    break;
                case 999:
                    System.exit(999);
                    break;
                default:
                    System.out.println("Invalid choice. Please try
again.");
                    break;
            }
        } while (choice != 999);
    }

    private static void displayMenu() {
        System.out.println("\t\t*********************");
        System.out.println("\t\t* Food Center Menu *");
        System.out.println("\t\t*********************");
        System.out.println("\n\t100 or VFQ: View all Queues");
        System.out.println("\t101 or VEQ: View all Empty Queues");
        System.out.println("\t102 or ACQ: Add customer to a Queue");
        System.out.println("\t103 or RCQ: Remove a customer from a Queue");
//menu options
        System.out.println("\t104 or PCQ: Remove a served customer");
        System.out.println("\t105 or VCS: View Customers Sorted in
alphabetical order");
        System.out.println("\t106 or SPD: Store Program Data into file");
        System.out.println("\t107 or LPD: Load Program Data from file");
        System.out.println("\t108 or STK: View Remaining burgers Stock");
        System.out.println("\t109 or AFS: Add burgers to Stock");
        System.out.println("\t999 or EXT: Exit the Program");
        System.out.println("\n\t\tEnter your choice:  ");
    }

    public static void viewAllQueues(String[] queue1, String[] queue2,
String[] queue3) {
        System.out.println("****************");
        System.out.println("*   Cashiers   *");
        System.out.println("*****************");


        for (int i = 0; i < queue3.length; i++) {

            if(i<2){
                System.out.print(queue1[i] == null ? "X": "O");
            }
            if(i<3){
                System.out.print(queue2[i] == null ? "\t\tX": "\t\tO");
            }
            if(i<5){
                if (i==3||i==4){
                    System.out.print("\t\t");
                }
                System.out.print(queue3[i] == null ? "\t\tX": "\t\tO");
            }
            System.out.println();
        }
    }

    private static void viewAllEmptyQueues(String[] queue) {
        System.out.println(" Queue :");

        for (int i = 0; i < queue.length; i++) {
            if (queue[i] == null) {
                System.out.println("\t\tSlot " + (i + 1));
            }
        }
    }

    private static void addCustomer(String[] queue1, String[] queue2,
String[] queue3) {
        int queueNumber;

        System.out.println("Enter the queue number (1, 2, or 3):");
        try {
            queueNumber = userInput.nextInt(); // Read the queue number
input from the user
            userInput.nextLine(); // Move to the next line to clear the
input buffer
        } catch (InputMismatchException e) {
            System.out.println("Invalid queue number. Please enter a valid
integer."); // Print an error message for an invalid queue number
            return;
        }

        while (queueNumber < 1 || queueNumber > 3) {
            System.out.println("Invalid queue number.");
            System.out.println("Enter the queue number (1, 2, or 3):");
            try {
                queueNumber = userInput.nextInt(); // Read the queue number
input from the user
                userInput.nextLine();  // Move to the next line to clear
the input buffer
            } catch (InputMismatchException e) {
                System.out.println("Invalid queue number. Please enter a
valid integer.");
                return;
            }
        }

        System.out.println("Enter the customer name:");
        String customerName = userInput.nextLine();

        if (queueNumber == 1) {
            add(queue1, customerName);
            System.out.println(customerName + " added to queue 1
successfully!");
        } else if (queueNumber == 2) {
            add(queue2, customerName);
            System.out.println(customerName + " added to queue 2
successfully!");
        } else if (queueNumber == 3) {
            add(queue3, customerName);
            System.out.println(customerName + " added to queue 3
successfully!");
        }

        // Update stock
        stock -= 5;
        if (stock <= 10) {
            System.out.println("Warning: Low stock! Remaining stock: " +
stock + " burgers");
        }
    }

    public static void add(String[] queue, String name) {
        for (int i = 0; i < queue.length; i++) {
            if (queue[i] == null) {
                queue[i] = name;   // Add the customer to the first
available slot in the queue
                break;
            }
        }
    }

    private static void removeCustomer() {
        Scanner scanner = new Scanner(System.in);
        int queueNumber;

        System.out.println("Enter the queue number (1, 2, or 3):");
        try {
            queueNumber = Integer.parseInt(scanner.nextLine()); // Read the
queue number input from the user
        } catch (NumberFormatException e) {
            System.out.println("Invalid queue number. Please enter a valid
integer.");
            return;
        }

        if (queueNumber < 1 || queueNumber > 3) {
            System.out.println("Invalid queue number.");
            return;
        }

        String[] queue = queues[queueNumber - 1]; // Get the selected queue

        if (queue.length == 0) {
            System.out.println("Queue is already empty.");
            return;
        }

        System.out.println("Enter the customer index to remove (0 to " + 
(queue.length - 1) + "):");
        int customerIndex;
        try {
            customerIndex = Integer.parseInt(scanner.nextLine()); // Read 
the customer index input from the user
        } catch (NumberFormatException e) {
            System.out.println("Invalid customer index. Please enter a 
valid integer.");
            return;
        }

        if (customerIndex < 0 || customerIndex >= queue.length) {
            System.out.println("Invalid customer index.");
            return;
        }

        for (int i = customerIndex; i < queue.length - 1; i++) {  // Shift 
the customers to the left to remove the selected customer
            queue[i] = queue[i + 1];
        }

        queue[queue.length - 1] = null;   // Set the last element to null 
to indicate an empty slot

        System.out.println("Customer removed from Queue " + queueNumber);
    }
    private static void removeServedCustomer() {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter the queue number (1, 2, or 3):");
        int queueNumber;
        try {
            queueNumber = Integer.parseInt(scanner.nextLine());
        } catch (NumberFormatException e) {
            System.out.println("Invalid queue number. Please enter a valid 
integer.");
            return;
        }

        if (queueNumber < 1 || queueNumber > 3) {
            System.out.println("Invalid queue number.");
            return;
        }

        String[] queue = queues[queueNumber - 1];

        if (queue.length > 0) {
            System.out.println("Enter the position of the served customer 
(0 to " + (queue.length - 1) + "):");
            int position;
            try {
                position = Integer.parseInt(scanner.nextLine());
            } catch (NumberFormatException e) {
                System.out.println("Invalid position. Please enter a valid 
integer.");
                return;
            }
            if (position < 0 || position >= queue.length) {
                System.out.println("Invalid position.");
                return;
            }

            String servedCustomer = queue[position]; // Get the customer at
the specified position

            // Shift the customers to the left to remove the served
customer
            for (int i = position; i < queue.length - 1; i++) {
                queue[i] = queue[i + 1];
            }

            // Set the last element to null to indicate an empty slot
            queue[queue.length - 1] = null;

            System.out.println("Customer " + servedCustomer + " served from
Queue " + queueNumber);
        } else {
            System.out.println("No customers to serve in Queue " +
queueNumber);
        }
    }
    private static void viewCustomersSorted() {
        int totalCustomers = 0;

        for (String[] queue : queues) {
            for (String customer : queue) {
                if (customer != null) {
                    totalCustomers++; // Count the number of non-null
customers
                }
            }
        }

        String[] allCustomers = new String[totalCustomers];
        int index = 0;

        for (String[] queue : queues) {
            for (String customer : queue) {
                if (customer != null) {
                    allCustomers[index++] = customer; // Add non-null
customers to the array
                }
            }
        }

        // Sort the customer array using a simple bubble sort algorithm
        int n = allCustomers.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (allCustomers[j].compareTo(allCustomers[j + 1]) > 0) {
                    // Swap customers if they are out of order
                    String temp = allCustomers[j];
                    allCustomers[j] = allCustomers[j + 1];
                    allCustomers[j + 1] = temp;
                }
            }
        }
        System.out.println("Customers Sorted in alphabetical order:");

        for (String customer : allCustomers) {
            System.out.println(customer); // Print the sorted customers
        }
    }

    private static void storeProgramData(String[] queue) {
        try {
            FileWriter write = new FileWriter("Text.txt", true);    //
Create a FileWriter object to write data to the file
            for (int i = 0; i < queue.length; i++) {
                if (queue[i] != null) {
                    write.append(queue[i]);  // Append the customer data to
the file
                    write.append(System.lineSeparator());   // Add a new
line after each customer
                }
            }
            write.close();   // Close the FileWriter object to release
resources
        } catch (IOException ex) {      // Exception handling code can be
added here to handle any IO errors that may occur

        }
    }

    private static void loadProgramData() {
        try {
            File readFile = new File("Text.txt");   // Create a File object
to read data from the file
            Scanner reader = new Scanner(readFile);   // Create a Scanner
object to read the file
            while (reader.hasNextLine()) {   // Loop through each line in
the file
                String text = reader.nextLine();   // Read the current line
of text from the file
                System.out.println(text);  // Print the text to the console
            }
            reader.close();  // Close the Scanner object to release
resources
        } catch (IOException e) {
            System.out.println("Error File Reading");  // Handle any IO
errors that may occur
        }
    }



    private static void viewRemainingStock() {
        System.out.println("Remaining burgers in stock: " + stock); //
Print the remaining stock of burgers
    }

    private static void addBurgersToStock() {
        Scanner scanner = new Scanner(System.in);  //getting inputs
        int quantity;

        System.out.println("Enter the quantity of burgers to add:");
        quantity = scanner.nextInt();
        scanner.nextLine();
        stock += quantity;
        System.out.println(quantity + " burgers added to stock. Total 
stock: " + stock);
    }
}
```

Task 02 & 03 (Classes version)

1 – Main Class

```
import java.io.File;              // Import the File class for file
handling
        import java.io.FileWriter;       // Import the FileWriter class for
writing to a file
        import java.io.IOException;      // Import the IOException class
for handling I/O exceptions
        import java.util.ArrayList;       // Import the ArrayList class for
storing data dynamically
        import java.util.Collections;    // Import the Collections class
for sorting
        import java.util.Scanner;         // Import the Scanner class for
user input

public class Main {
    private static final Scanner userInput = new Scanner(System.in);    //
Create a Scanner object for user input
    public static int[] maxQueueLimit = {2, 3, 5};            // Maximum
capacity of each queue

    // Create three instances of FoodQueue with the specified capacities
    public static FoodQueue queue1 = new FoodQueue(maxQueueLimit[0]);
    public static FoodQueue queue2 = new FoodQueue(maxQueueLimit[1]);
    public static FoodQueue queue3 = new FoodQueue(maxQueueLimit[2]);

    public static FoodQueue[] queues = {queue1, queue2, queue3};     //
Array to store the queues

    public static int[] income = {0, 0, 0};      // Array to store the
income of each queue

    public static int burgersInStock = 50;           // Initial stock of
burgers
    public static final int warningLimit = 10;          // Warning limit for
low stock
    public static ArrayList<Customer> waitingList = new ArrayList<>();  //
List to store customers in the waiting list

    public static void main(String[] args) {

        try {
            File file = new File("Text.txt");          // Create a file
object with the file name "Text.txt"
            file.createNewFile();          // Create a new file if it does
not exist
        } catch (IOException ioe) {
            System.out.println();
        }

        String choice;      // Variable to store the user's menu choice

        do {
            displayMenu();
            choice = userInput.nextLine();

            switch (choice) {
                case "100", "VFQ":
                    viewAllQueues();
                    break;

                case "101", "VEQ":
                    viewAllEmptyQueues();
                    break;

                case "102", "ACQ":
                    addCustomer();
                    break;

                case "103", "RCQ":
                    removeCustomer();
                    break;

                case "104", "PCQ":
                    removeServedCustomer();
                    break;

                case "105", "VCS":
                    viewCustomersSorted();
                    break;

                case "106", "SPD":
                    storeProgramData();
                    break;

                case "107", "LPD":
                    loadProgramData();
                    break;

                case "108", "STK":
                    viewRemainingStock();
                    break;

                case "109", "AFS":
                    addBurgersToStock();
                    break;

                case "110", "INC":
                    incomeOfEachQueue();
                    break;

                case "999", "EXT":
                    System.exit(0);     // Terminate the program

                default:
                    System.out.println("Invalid choice. Please try 
again.");
                    break;
            }
        } while (choice != "999" || choice != "EXT");
    }


    private static void displayMenu() {
        System.out.println("\n\t\t**********************");
        System.out.println("\t\t* Food Center Menu *");
        System.out.println("\t\t**********************");
        System.out.println("\n\t100 or VFQ: View all Queues");
        System.out.println("\t101 or VEQ: View all Empty Queues");
        System.out.println("\t102 or ACQ: Add customer to a Queue");
        System.out.println("\t103 or RCQ: Remove a customer from a Queue");
        System.out.println("\t104 or PCQ: Remove a served customer");
// Display the menu options
        System.out.println("\t105 or VCS: View Customers Sorted in
alphabetical order");
        System.out.println("\t106 or SPD: Store Program Data into file");
        System.out.println("\t107 or LPD: Load Program Data from file");
        System.out.println("\t108 or STK: View Remaining burgers Stock");
        System.out.println("\t109 or AFS: Add burgers to Stock");
        System.out.println("\t110 or INC: Get income of each queue
separately");
        System.out.println("\t999 or EXT: Exit the Program");
        System.out.print("\n\t\tEnter your choice:  ");
    }

    // View all the queues
    private static void viewAllQueues() {
        System.out.println("\n***  Cashiers  ***\n");
        System.out.println("1       2       3");
        System.out.println("__      __      __");
        int maxCapacity = Math.max(queue1.getCapacity(),
Math.max(queue2.getCapacity(), queue3.getCapacity()));

        for (int i = 0; i < maxCapacity; i++) {
            if (i < queue1.getCapacity()) {
                System.out.print(queue1.getCustomers()[i] != null ? "O" :
"X");  // Print 'O' if a customer exists, 'X' otherwise
            }
            System.out.print("\t\t");
            if (i < queue2.getCapacity()) {
                System.out.print(queue2.getCustomers()[i] != null ? "O" :
"X");
            }
            System.out.print("\t\t");
            if (i < queue3.getCapacity()) {
                System.out.print(queue3.getCustomers()[i] != null ? "O" :
"X");
            }
            System.out.println();
        }
        System.out.println("\nX - Not Occupied   O - Occupied");
    }


    private static void viewAllEmptyQueues() {
        int index = 1;
        for (FoodQueue queue : queues) {
            System.out.println("Queue " + index);
            for (int i = 0; i < queue.getCapacity(); i++) {
                if (queue.getCustomers()[i] == null) {
                    System.out.println("Slot " + (i + 1) + " : Empty");
                } else {
                    System.out.println("Slot " + (i + 1) + " : " +
queue.getCustomers()[i].getFirstName());  // If the slot has a customer,
print the customer's first name
                }
            }
            index++;
        }
    }
    // Add a customer to a queue
    private static int waitingListIndex = 0;   // Index for circular queue
implementation

    private static void addCustomer() {
        if (burgersInStock > 0) {
            System.out.print("Enter First Name: ");
            String firstName = userInput.nextLine();
            System.out.print("Enter Last Name: ");
            String lastName = userInput.nextLine();
            System.out.print("Enter Burgers Needed: ");

            try {
                int burgersNeeded = Integer.parseInt(userInput.nextLine());
// Read the number of burgers needed

                if (burgersNeeded < burgersInStock) {
                    Customer customer = new Customer(firstName, lastName,
burgersNeeded);    // Create a new customer object with the entered details

                    int minIndex = 0;       // Initialize the index of the
queue with the minimum length
                    int minLength = Integer.MAX_VALUE;     // Initialize
the minimum length of the queues

                    // Find the queue with the minimum length
                    for (int i = 0; i < queues.length; i++) {
                        int queueLength = queues[i].getQueueFilledLength();

                        if (queueLength == queues[i].getCapacity()) {
                            continue;     // Skip if the queue is already
full
                        } else if (queueLength < minLength) {
                            minLength = queueLength;
                            minIndex = i;  // Update the index of the queue
with the minimum length
                        }
                    }

                    if (minLength >= queues[minIndex].getCapacity()) {
                        System.out.println("Added customer to the Waiting
List.");
                        waitingList.add(waitingListIndex, customer);  //
Add the customer to the waiting list at the current index
                        waitingListIndex = (waitingListIndex + 1) %
maxQueueLimit.length;  // Implement circular queue for the waiting list
                    } else {
                        if (!waitingList.isEmpty()) {
                            // If the waiting list is not empty, add the
next customer from the waiting list to the selected queue
                            Customer nextCustomer =
waitingList.remove(waitingListIndex);  // Get the next customer from the
waiting list
                            queues[minIndex].addCustomer(nextCustomer);  //
Add the customer to the selected queue
                            System.out.println("Added customer from waiting
list to Queue " + (minIndex + 1));
                            burgersInStock -= nextCustomer.getNobr();
                            waitingListIndex = (waitingListIndex - 1 +
maxQueueLimit.length) % maxQueueLimit.length;  // Update the waiting list
index using circular queue logic
                        } else {
                            // If the waiting list is empty, add the
customer directly to the selected queue
                            queues[minIndex].addCustomer(customer);  // Add
the customer to the selected queue
                            System.out.println("Added customer to Cashier "
+ (minIndex + 1) + " Queue.");
                            burgersInStock -= burgersNeeded;
                        }

                        if (burgersInStock <= warningLimit) {
                            System.out.println("Warning: Low stock.
Remaining stock: " + burgersInStock);
                        }
                    }
                } else {
                    System.out.println("Enter an amount below " +
burgersInStock);
                }
            } catch (NumberFormatException e) {
                System.out.println("Invalid input for the number of burgers
needed. Please enter a valid integer.");
            }
        } else {
            System.out.println("Burgers Out of Stock");
        }
    }
    private static void removeCustomer() {
        System.out.println("Enter Queue Number: ");
        int queueNumber = Integer.parseInt(userInput.nextLine());  // Read
the queue number from the user
        System.out.println("Enter Queue Index: ");
        int queueIndex = Integer.parseInt(userInput.nextLine());  // Read
the queue index from the user

        if (queueNumber > 0 && queueNumber < 4 && queueIndex > 0 &&
queueIndex <= queues[queueNumber - 1].getQueueFilledLength()) {
            // Check if the queue number and index are valid

            FoodQueue selectedQueue = queues[queueNumber - 1];  // Get the
selected queue based on the queue number
            Customer[] customers = selectedQueue.getCustomers();  // Get
the array of customers in the selected queue

            int removedCustomerBurgers = customers[queueIndex -
1].getNobr();  // Get the number of burgers of the removed customer
            burgersInStock += removedCustomerBurgers;  // Increase the
number of burgers in stock

            // Shift customers to fill the empty position caused by the
removal
            for (int i = queueIndex - 1; i <
selectedQueue.getQueueFilledLength() - 1; i++) {
                customers[i] = customers[i + 1];
            }
            customers[selectedQueue.getQueueFilledLength() - 1] = null;  //
Set the last position as null

            System.out.println("Customer Removed Successfully");
            if (!waitingList.isEmpty()) {
                // If the waiting list is not empty, add the next customer
to the selected queue
                Customer nextCustomer = waitingList.remove(0);  // Get the
next customer from the waiting list
                queues[queueNumber - 1].addCustomer(nextCustomer);  // Add
the customer to the selected queue
                System.out.println("Customer Added From Waiting List");
                burgersInStock -= nextCustomer.getNobr();
            }
        } else {
            System.out.println("Invalid Queue or Index");
        }
    }


    private static void removeServedCustomer() {
        System.out.println("Enter Queue Number: ");
        int queueNumber = Integer.parseInt(userInput.nextLine());  // Read
the queue number from the user

        if (queues[queueNumber - 1] == null)   //// Check if the selected
queue is empty
            System.out.println("Queue is Empty !");

        else if (queueNumber > 0 && queueNumber < 4) {    // If the queue
number is valid

            income[queueNumber - 1] += queues[queueNumber -
1].getCustomers()[0].getNobr() * 650;    // Increase the income of the
corresponding queue by the number of burgers served multiplied by the price
            queues[queueNumber - 1].getCustomers()[0] = null;  // Set the
first customer as null to remove the served customer
            System.out.println("Served Customer Removed Successfully");

            for (int i = 0; i < queues[queueNumber - 1].getCapacity() - 1;
i++) {   // Shift the customers to fill the empty position caused by the
removal

                queues[queueNumber - 1].getCustomers()[i] =
queues[queueNumber - 1].getCustomers()[i + 1];
            }

            queues[queueNumber - 1].getCustomers()[queues[queueNumber -
1].getCapacity() - 1] = null;  // Set the last position as null


            if (!waitingList.isEmpty()) {   // If the waiting list is not
empty, add the next customer to the selected queue

                queues[queueNumber - 1].getCustomers()[queues[queueNumber -
1].getQueueFilledLength()] = waitingList.get(0);  // Add the customer from
the waiting list to the selected queue
                System.out.println("Customer Added From Waiting List");
                burgersInStock -= waitingList.get(0).getNobr();
                waitingList.remove(0);  // Remove the customer from the
waiting list
            }
        } else {
            System.out.println("Invalid Queue number");
        }
```

```java
    }

    private static void viewCustomersSorted() {
        int queueIndex = 1;
        for (FoodQueue queue : queues) {
            System.out.println("\nQueue " + queueIndex);

            ArrayList<String> sorting = new ArrayList<>();  // Create an
ArrayList to store the customer names for sorting

            for (int i = 0; i < queue.getCustomers().length; i++) {
                if (queue.getCustomers()[i] != null) {
                    sorting.add(queue.getCustomers()[i].getFullName());  //
Add the full name of each customer to the sorting ArrayList
                }
            }

            Collections.sort(sorting);  // Sort the ArrayList in
alphabetical order

            for (int j = 0; j < sorting.size(); j++) {
                System.out.println(sorting.get(j));
            }

            queueIndex++;  // Increment the queue index
        }
    }


    private static void storeProgramData() {
        try {
            FileWriter write = new FileWriter("Text.txt", true);
// Create a FileWriter object with the file name "Text.txt"
            for (FoodQueue queue : queues) {
                for (int i = 0; i < queue.getCustomers().length; i++) {
                    if (queue.getCustomers()[i] != null) {

write.append(queue.getCustomers()[i].getFullName());  // Append the full
name of each customer to the file
                    }
                }
            }
            write.close();
// Close the FileWriter object
            System.out.println("Program Data Stored Successfully");
        } catch (IOException e) {
            System.out.println("An error occurred while storing program
data.");
            e.printStackTrace();                                          //
Print the stack trace if an exception occurs
        }
    }


    private static void loadProgramData() {
        try {
            File readFile = new File("Text.txt");  // Create a File object
with the file name "Text.txt"
            Scanner reader = new Scanner(readFile);     // Create a Scanner
object to read from the file
```

```
            while (reader.hasNextLine()) {
                String text = reader.nextLine();    // Read the next line
from the file
                System.out.println(text);    // Print the line to the
console
            }

            System.out.println("\nStored data in file");
            reader.close();    // Close the Scanner object
        } catch (IOException e) {
            System.out.println("Error File Reading");
        }
    }



    private static void viewRemainingStock() {
        System.out.println("Remaining Stock of Burgers: " +
burgersInStock);
    }


    private static void addBurgersToStock() {
        System.out.print("Enter the number of burgers to add: ");
        int burgersToAdd = Integer.parseInt(userInput.nextLine());  // Read
the number of burgers to add from the user

        burgersInStock += burgersToAdd;
        System.out.println("Burgers added to the stock.");
    }

    private static void incomeOfEachQueue() {
        for (int i = 0; i < income.length; i++) {
            System.out.println("Income of Queue " + (i + 1) + ": " +
income[i]);
        }
    }
}
```

2 – Customer Class

```
public class Customer {
    private String firstName;  //Declaration of three public instance
variables
    private String lastName;
    private int nobr;

    public Customer(String firstName, String lastName, int nobr) {
        this.firstName = firstName;  //Assigning the values of the
constructor parameters to the corresponding instance variables using the
this keyword
        this.lastName = lastName;
        this.nobr = nobr;
    }

    public String getLastName() {
        return lastName;
    }  //A getter method that returns the last name of the customer
    public String getFirstName() {
        return firstName;
    }  // getter method that returns the first name of the customer

    public int getNobr() {
        return nobr;
    }  //A getter method that returns the number of burgers needed by the
customer.

    public String getFullName() {
        return firstName + " " + lastName;
    }   //A method that returns the full name of the customer by
concatenating the first name and last name with a space in between

}
```

3 – Food Queue Class

```
public class FoodQueue {
    private int capacity;                  // Maximum capacity of the queue
    private Customer[] customerObjects;    // Array to store Customer
objects in the queue

    public FoodQueue(int capacity) {     // Constructor that initializes the
FoodQueue with the given capacity
        this.capacity = capacity;
        customerObjects = new Customer[capacity];
    }

    public int getCapacity() {             // Getter method to retrieve the
maximum capacity of the queue
        return this.capacity;
    }

    public Customer[] getCustomers() {    // Getter method to retrieve the
Customer array in the queue
        return this.customerObjects;
    }

    public int getQueueFilledLength() {   // Method to get the filled length
of the queue (number of non-null elements)
        int notNullIndexes = 0;
        for (int i = 0; i < customerObjects.length; i++) {
            if (customerObjects[i] != null) {
                notNullIndexes++;
            }
        }
        return notNullIndexes;
    }

    public void addCustomer(Customer customer) {   // Method to add a
Customer object to the queue
        for (int i = 0; i < customerObjects.length; i++) {
            if (customerObjects[i] == null) {
                customerObjects[i] = customer;
                break;
            }
```

```
                }
        }
}
```

Task 04 (Java FX)

1 – Main Class

```
package com.example.task_04;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.stage.Stage;

import java.io.File;               // Import the File class for file
handling
import java.io.FileWriter;        // Import the FileWriter class for writing
to a file
import java.io.IOException;       // Import the IOException class for
handling I/O exceptions
import java.util.ArrayList;        // Import the ArrayList class for storing
data dynamically
import java.util.Collections;     // Import the Collections class for
sorting
import java.util.Scanner;          // Import the Scanner class for user
input
import com.example.task_04.HelloApplication;

public class Main {
    private static final Scanner userInput = new Scanner(System.in);    //
Create a Scanner object for user input
    public static int[] maxQueueLimit = {2, 3, 5};            // Maximum
capacity of each queue

    // Create three instances of FoodQueue with the specified capacities
    public static FoodQueue queue1 = new FoodQueue(maxQueueLimit[0]);
    public static FoodQueue queue2 = new FoodQueue(maxQueueLimit[1]);
    public static FoodQueue queue3 = new FoodQueue(maxQueueLimit[2]);

    public static FoodQueue[] queues = {queue1, queue2, queue3};     //
Array to store the queues

    public static int[] income = {0, 0, 0};      // Array to store the
income of each queue

    public static int burgersInStock = 50;           // Initial stock of
burgers
    public static final int warningLimit = 10;         // Warning limit for
low stock
    public static ArrayList<Customer> waitingList = new ArrayList<>();  //
List to store customers in the waiting list


    private static volatile boolean javaFXLaunched = false;

    public static void userInterface(Class<? extends Application>
applicationClass) {
        if (!javaFXLaunched) {
            Platform.setImplicitExit(false);
            new Thread(() -> Application.launch(applicationClass)).start();
            javaFXLaunched = true;
        } else {
            Platform.runLater(() -> {
                try {
                    Application application =
applicationClass.newInstance();
                    Stage primaryStage = new Stage();
                    application.start(primaryStage);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            });
        }
    }

    public static void main(String[] args) {

        try {
            File file = new File("Text.txt");        // Create a file
object with the file name "Text.txt"
            file.createNewFile();        // Create a new file if it does
not exist
        } catch (IOException ioe) {
            System.out.println();
        }

        String choice;     // Variable to store the user's menu choice

        do {
            displayMenu();
            choice = userInput.nextLine();

            switch (choice) {
                case "100", "VFQ":
                    viewAllQueues();
                    break;

                case "101", "VEQ":
                    viewAllEmptyQueues();
                    break;

                case "102", "ACQ":
                    addCustomer();
                    break;

                case "103", "RCQ":
                    removeCustomer();
                    break;

                case "104", "PCQ":
                    removeServedCustomer();
                    break;

                case "105", "VCS":
                    viewCustomersSorted();
                    break;

                case "106", "SPD":
                    storeProgramData();
                    break;

                case "107", "LPD":
                    loadProgramData();
                    break;

                case "108", "STK":
                    viewRemainingStock();
                    break;

                case "109", "AFS":
                    addBurgersToStock();
                    break;

                case "110", "INC":
                    incomeOfEachQueue();
                    break;

                case "112" , "GUI":
                    userInterface(HelloApplication.class);
                    System.out.println("\tG U I loaded ...... ");


                case "999", "EXT":
                    System.exit(0);    // Terminate the program

                default:
                    System.out.println("Invalid choice. Please try
again.");
                    break;
            }
        } while (choice != "999" || choice != "EXT");
    }


    private static void displayMenu() {
        System.out.println("\n\t\t*********************");
        System.out.println("\t\t* Food Center Menu *");
        System.out.println("\t\t*********************");
        System.out.println("\n\t100 or VFQ: View all Queues");
        System.out.println("\t101 or VEQ: View all Empty Queues");
        System.out.println("\t102 or ACQ: Add customer to a Queue");
        System.out.println("\t103 or RCQ: Remove a customer from a Queue");
        System.out.println("\t104 or PCQ: Remove a served customer");
// Display the menu options
        System.out.println("\t105 or VCS: View Customers Sorted in
alphabetical order");
        System.out.println("\t106 or SPD: Store Program Data into file");
        System.out.println("\t107 or LPD: Load Program Data from file");
        System.out.println("\t108 or STK: View Remaining burgers Stock");
        System.out.println("\t109 or AFS: Add burgers to Stock");
        System.out.println("\t110 or INC: Get income of each queue
separately");
        System.out.println("\t112 or GUI: View Grafical User Interface");
        System.out.println("\t999 or EXT: Exit the Program");
        System.out.print("\n\tEnter your choice:  ");
    }

    // View all the queues
    private static void viewAllQueues() {
        System.out.println("\n***  Cashiers  ***\n");
        System.out.println("1       2       3");
        System.out.println("__      __      __");
        int maxCapacity = Math.max(queue1.getCapacity(),
Math.max(queue2.getCapacity(), queue3.getCapacity()));
```

```
        for (int i = 0; i < maxCapacity; i++) {
            if (i < queue1.getCapacity()) {
                System.out.print(queue1.getCustomers()[i] != null ? "O" :
"X");  // Print 'O' if a customer exists, 'X' otherwise
            }
            System.out.print("\t\t");
            if (i < queue2.getCapacity()) {
                System.out.print(queue2.getCustomers()[i] != null ? "O" :
"X");
            }
            System.out.print("\t\t");
            if (i < queue3.getCapacity()) {
                System.out.print(queue3.getCustomers()[i] != null ? "O" :
"X");
            }
            System.out.println();
        }
        System.out.println("\nX - Not Occupied   O - Occupied");
    }


    private static void viewAllEmptyQueues() {
        int index = 1;
        for (FoodQueue queue : queues) {
            System.out.println("Queue " + index);
            for (int i = 0; i < queue.getCapacity(); i++) {
                if (queue.getCustomers()[i] == null) {
                    System.out.println("Slot " + (i + 1) + " : Empty");
                } else {
                    System.out.println("Slot " + (i + 1) + " : " +
queue.getCustomers()[i].getFirstName());  // If the slot has a customer,
print the customer's first name
                }
            }
            index++;
        }
    }



    private static int waitingListIndex = 0;   // Index for circular queue
implementation

    private static void addCustomer() {
        if (burgersInStock > 0) {
            System.out.print("Enter First Name: ");
            String firstName = userInput.nextLine();
            System.out.print("Enter Last Name: ");
            String lastName = userInput.nextLine();
            System.out.print("Enter Burgers Needed: ");
            int burgersNeeded = Integer.parseInt(userInput.nextLine());
// Read the number of burgers needed

            if (burgersNeeded < 1)
                System.out.println("You can not add zero or minus burgers
!");

            else if (burgersNeeded < burgersInStock) {
                Customer customer = new Customer(firstName, lastName,
burgersNeeded);    // Create a new customer object with the entered details
```

```
                int minIndex = 0;         // Initialize the index of the
queue with the minimum length
                int minLength = Integer.MAX_VALUE;      // Initialize the
minimum length of the queues

                // Find the queue with the minimum length
                for (int i = 0; i < queues.length; i++) {
                    int queueLength = queues[i].getQueueFilledLength();

                    if (queueLength == queues[i].getCapacity()) {
                        continue;      // Skip if the queue is already full
                    } else if (queueLength < minLength) {
                        minLength = queueLength;
                        minIndex = i;   // Update the index of the queue
with the minimum length
                    }
                }

                if (minLength >= queues[minIndex].getCapacity()) {
                    // If the minimum length is equal to the capacity, add
the customer to the waiting list
                    System.out.println("Added customer to the Waiting
List.");
                    waitingList.add(waitingListIndex, customer);  // Add
the customer to the waiting list at the current index
                    waitingListIndex = (waitingListIndex + 1) %
maxQueueLimit.length;  // Implement circular queue for the waiting list
                } else {
                    if (!waitingList.isEmpty()) {
                        // If the waiting list is not empty, add the next
customer from the waiting list to the selected queue
                        Customer nextCustomer =
waitingList.remove(waitingListIndex);  // Get the next customer from the
waiting list
                        queues[minIndex].addCustomer(nextCustomer);  // Add
the customer to the selected queue
                        System.out.println("Added customer from waiting
list to Queue " + (minIndex + 1));
                        burgersInStock -= nextCustomer.getNobr();
                        waitingListIndex = (waitingListIndex - 1 +
maxQueueLimit.length) % maxQueueLimit.length;  // Update the waiting list
index using circular queue logic
                    } else {
                        // If the waiting list is empty, add the customer
directly to the selected queue
                        queues[minIndex].addCustomer(customer);  // Add the
customer to the selected queue
                        System.out.println("Added customer to Cashier " +
(minIndex + 1) + " Queue.");
                        burgersInStock -= burgersNeeded;
                    }

                    if (burgersInStock <= warningLimit) {
                        System.out.println("Warning: Low stock. Remaining
stock: " + burgersInStock);
                    }
                }
            } else {
                System.out.println("Enter an amount below " +
burgersInStock);
```

```
            }
        } else {
            System.out.println("Burgers Out of Stock");
        }
    }

    private static void removeCustomer() {
        System.out.println("Enter Queue Number: ");
        int queueNumber = Integer.parseInt(userInput.nextLine());  // Read
the queue number from the user
        System.out.println("Enter Queue Index: ");
        int queueIndex = Integer.parseInt(userInput.nextLine());  // Read
the queue index from the user

        if (queueNumber > 0 && queueNumber < 4 && queueIndex > 0 &&
queueIndex <= queues[queueNumber - 1].getQueueFilledLength()) {
            // Check if the queue number and index are valid

            FoodQueue selectedQueue = queues[queueNumber - 1];  // Get the
selected queue based on the queue number
            Customer[] customers = selectedQueue.getCustomers();  // Get
the array of customers in the selected queue

            int removedCustomerBurgers = customers[queueIndex -
1].getNobr();  // Get the number of burgers of the removed customer
            burgersInStock += removedCustomerBurgers;  // Increase the
number of burgers in stock

            // Shift customers to fill the empty position caused by the
removal
            for (int i = queueIndex - 1; i <
selectedQueue.getQueueFilledLength() - 1; i++) {
                customers[i] = customers[i + 1];
            }
            customers[selectedQueue.getQueueFilledLength() - 1] = null;  //
Set the last position as null

            System.out.println("Customer Removed Successfully");

            if (!waitingList.isEmpty()) {
                // If the waiting list is not empty, add the next customer
to the selected queue
                Customer nextCustomer = waitingList.remove(0);  // Get the
next customer from the waiting list
                queues[queueNumber - 1].addCustomer(nextCustomer);  // Add
the customer to the selected queue
                System.out.println("Customer Added From Waiting List");
                burgersInStock -= nextCustomer.getNobr();
            }
        } else {
            System.out.println("Invalid Queue or Index");
        }
    }


    private static void removeServedCustomer() {
        System.out.println("Enter Queue Number: ");
        int queueNumber = Integer.parseInt(userInput.nextLine());  // Read
the queue number from the user

        if (queues[queueNumber - 1] == null)   //// Check if the selected
```

```
queue is empty
            System.out.println("Queue is Empty !");

        else if (queueNumber > 0 && queueNumber < 4) {    // If the queue
number is valid

            income[queueNumber - 1] += queues[queueNumber -
1].getCustomers()[0].getNobr() * 650;    // Increase the income of the
corresponding queue by the number of burgers served multiplied by the price
            queues[queueNumber - 1].getCustomers()[0] = null;  // Set the
first customer as null to remove the served customer
            System.out.println("Served Customer Removed Successfully");

            for (int i = 0; i < queues[queueNumber - 1].getCapacity() - 1;
i++) {   // Shift the customers to fill the empty position caused by the
removal

                queues[queueNumber - 1].getCustomers()[i] =
queues[queueNumber - 1].getCustomers()[i + 1];
            }

            queues[queueNumber - 1].getCustomers()[queues[queueNumber -
1].getCapacity() - 1] = null;  // Set the last position as null


            if (!waitingList.isEmpty()) {   // If the waiting list is not
empty, add the next customer to the selected queue

                queues[queueNumber - 1].getCustomers()[queues[queueNumber -
1].getQueueFilledLength()] = waitingList.get(0);  // Add the customer from
the waiting list to the selected queue
                System.out.println("Customer Added From Waiting List");
                burgersInStock -= waitingList.get(0).getNobr();
                waitingList.remove(0);  // Remove the customer from the
waiting list
            }
        } else {
            System.out.println("Invalid Queue number");
        }
    }

    private static void viewCustomersSorted() {
        int queueIndex = 1;
        for (FoodQueue queue : queues) {
            System.out.println("\nQueue " + queueIndex);

            ArrayList<String> sorting = new ArrayList<>();  // Create an
ArrayList to store the customer names for sorting

            for (int i = 0; i < queue.getCustomers().length; i++) {
                if (queue.getCustomers()[i] != null) {
                    sorting.add(queue.getCustomers()[i].getFullName());  //
Add the full name of each customer to the sorting ArrayList
                }
            }

            Collections.sort(sorting);  // Sort the ArrayList in
alphabetical order

            for (int j = 0; j < sorting.size(); j++) {
                System.out.println(sorting.get(j));
            }

            queueIndex++;  // Increment the queue index
        }
    }


    private static void storeProgramData() {
        try {
            FileWriter write = new FileWriter("Text.txt", true);
// Create a FileWriter object with the file name "Text.txt"
            for (FoodQueue queue : queues) {
                for (int i = 0; i < queue.getCustomers().length; i++) {
                    if (queue.getCustomers()[i] != null) {

write.append(queue.getCustomers()[i].getFullName());  // Append the full
name of each customer to the file
                    }
                }
            }
            write.close();
// Close the FileWriter object
            System.out.println("Program Data Stored Successfully");
        } catch (IOException e) {
            System.out.println("An error occurred while storing program
data.");
            e.printStackTrace();                                          //
Print the stack trace if an exception occurs
        }
    }


    private static void loadProgramData() {
        try {
            File readFile = new File("Text.txt");  // Create a File object
with the file name "Text.txt"
            Scanner reader = new Scanner(readFile);     // Create a Scanner
object to read from the file

            while (reader.hasNextLine()) {
                String text = reader.nextLine();    // Read the next line
from the file
                System.out.println(text);    // Print the line to the
console
            }

            System.out.println("\nStored data in file");
            reader.close();    // Close the Scanner object
        } catch (IOException e) {
            System.out.println("Error File Reading");
        }
    }



    private static void viewRemainingStock() {
        System.out.println("Remaining Stock of Burgers: " +
burgersInStock);
    }
```

```java
    private static void addBurgersToStock() {
        System.out.print("Enter the number of burgers to add: ");
        int burgersToAdd = Integer.parseInt(userInput.nextLine());  // Read
the number of burgers to add from the user

        burgersInStock += burgersToAdd;
        System.out.println("Burgers added to the stock.");
    }

    private static void incomeOfEachQueue() {
        for (int i = 0; i < income.length; i++) {
            System.out.println("Income of Queue " + (i + 1) + ": " +
income[i]);
        }
    }


}
```

2 – Customer Class

```java
package com.example.task_04;

public class Customer {
    public String firstName;  //Declaration of three public instance
variables
    public String lastName;
    public int nobr;

    public Customer(String firstName, String lastName, int nobr) { //The
constructor of the Customer class
        this.firstName = firstName;  //Assigning the values of the
constructor parameters to the corresponding instance variables using the
this keyword
        this.lastName = lastName;
        this.nobr = nobr;
    }

    public String getLastName() {
        return lastName;
    }  //A getter method that returns the last name of the customer

    public String getFirstName() {
        return firstName;
    }  // getter method that returns the first name of the customer

    public int getNobr() {
        return nobr;
    }  //A getter method that returns the number of burgers needed by the
customer.

    public String getFullName(){
        return firstName+" "+lastName;
    }   //A method that returns the full name of the customer by
concatenating the first name and last name with a space in between

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }  // A setter method to set the last name of the customer.

    public void setFirstName(String firstName) { //A setter method to set
the first name of the customer
        this.firstName = firstName;
    }

    public void setNobr(int nobr) {
        this.nobr = nobr;
    }  //A setter method to set the number of burgers needed by the
customer
}
```

3 – Food Queue Class

```
package com.example.task_04;

public class FoodQueue {
    private int capacity;                // Maximum capacity of the queue
    private Customer[] customerObjects;    // Array to store Customer
objects in the queue

    public FoodQueue(int capacity) {     // Constructor that initializes the
FoodQueue with the given capacity
        this.capacity = capacity;
        customerObjects = new Customer[capacity];
    }

    public int getCapacity() {            // Getter method to retrieve the
maximum capacity of the queue
        return this.capacity;
    }

    public Customer[] getCustomers() {    // Getter method to retrieve the
Customer array in the queue
        return this.customerObjects;
    }

    public int getQueueFilledLength() {   // Method to get the filled length
of the queue (number of non-null elements)
        int notNullIndexes = 0;
        for (int i = 0; i < customerObjects.length; i++) {
            if (customerObjects[i] != null) {
                notNullIndexes++;
            }
        }
        return notNullIndexes;
    }

    public void addCustomer(Customer customer) {  // Method to add a
Customer object to the queue
        for (int i = 0; i < customerObjects.length; i++) {
            if (customerObjects[i] == null) {
                customerObjects[i] = customer;
                break;
            }
        }
    }
}
```

4 – Hello Application Class

```
package com.example.task_04;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.stage.Stage;

import java.io.IOException;
public class HelloApplication extends Application {


    @Override

    public void start(Stage stage) throws IOException {
        FXMLLoader fxmlLoader = new
FXMLLoader(HelloApplication.class.getResource("hello-view.fxml"));
        Scene scene = new Scene(fxmlLoader.load(), 600, 400);
        stage.setResizable(false);
        stage.setTitle("Foodie Fave Queue Management System");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch();
    }
}
```

5 – Hello Controller Class

```
package com.example.task_04;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

import java.io.IOException;

public class HelloController {
    @FXML
    private Stage stage;
    private Scene scene;
    private Parent root;

    public HelloController() {

    }
    @FXML
    public void customersDetails(ActionEvent event) throws IOException {

    }
}
```

6 – Hello-View FXML

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Label?>
<?import javafx.scene.effect.Glow?>
<?import javafx.scene.image.Image?>
<?import javafx.scene.image.ImageView?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.text.Font?>

<AnchorPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-
Infinity" minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0"
style="-fx-background-image: #B2BEB5;" xmlns="http://javafx.com/javafx/19"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="com.example.task_04.HelloController">
   <children>
      <ImageView fitHeight="407.0" fitWidth="600.0">
         <image>
            <Image url="@../../../../../../../../02.png" />
         </image>
      </ImageView>
      <Label opacity="0.94" prefHeight="59.0" prefWidth="563.0" style="-fx-
background-color: #7393B3;" text="    Foodie Fave  ..."
textAlignment="CENTER" textFill="#050505">
         <font>
            <Font name="Franklin Gothic Demi Cond" size="40.0" />
         </font>
      </Label>
      <Label layoutX="260.0" layoutY="-8.0" prefHeight="75.0"
prefWidth="294.0" text="Queue management system" textAlignment="CENTER"
textFill="#4e0a0a">
         <font>
            <Font name="Gill Sans MT Condensed" size="37.0" />
         </font>
      </Label>
      <Label layoutY="59.0" prefHeight="37.0" prefWidth="335.0" style="-fx-
background-color: #899499;" text="   Customers Deatails  ------"
textAlignment="RIGHT" textFill="#1e0000" AnchorPane.bottomAnchor="304.0"
AnchorPane.rightAnchor="265.0" AnchorPane.topAnchor="59.0">
         <font>
            <Font name="Arial Rounded MT Bold" size="22.0" />
         </font>
      </Label>
      <Label layoutY="96.0" opacity="0.89" prefHeight="54.0"
prefWidth="335.0" style="-fx-background-color: #818589;" text=" Cusromes'
Names &amp; Burgers Required" textFill="WHITE">
         <font>
            <Font name="System Bold Italic" size="18.0" />
         </font>
      </Label>
   </children>
   <effect>
      <Glow />
   </effect>
</AnchorPane>
```