UNIVERSITY OF WESTMINSTER

INFORMATICS INSTITUTE OF TECHNOLOGY

Informatics Institute of Technology

Department of Computing

Software Development II Coursework Report

Module	: 4COSC010C.3: Software Development II (2022)
Module Leader	: TG Deshan K Sumanthilaka
Date of submission	: 17/07/2023
Student ID	: IIT No - 20221574/ UOW No - w1999471
Student First Name	: Milni
Student Surname	: Yaddehi

"I confirm that I understand what plagiarism/collusion/contract cheating is and have read and understood the section on Assessment Offences in the Essential Information for Students. The work that I have submitted is entirely my own. Any work from other authors is duly referenced and acknowledged."

Name : Y. Milni De Silva

Student ID : 20221574

Test Cases

Array Part (Task 1)

	Test Case	Expected Result	Actual Result	Pass/Fail
1	Food Queue Initialized Correctly After the program starts, 100 or VFQ	Displays "view all queues" for all queues.	Display "view all queues"	Pass
2	View all Empty queues initialized correctly After the program starts,101 or VEQ	Displays "view all empty queues", for all queues	Displays "view all empty queues",	pass
3	Add customer "Jane" to Queue 2 102 or ACQ Enter Your choice:102. Enter Customer Name: Jane Which cashier would you like to handle your purchase? Select a Cashier number. (1, 2, or 3): 2	Display "Jane added to the queue 2 successfully".	Display "Customer Jane added to the queue 2 successfully".	Pass
	Enter your choice:102. Enter Customer Name: Peter Which cashier would you like to handle your purchase? Select a Cashier number. (1,2,3) :1	Display "Peter added to queue 1 successfully".	Display "Peter added to queue 1 successfully".	Pass
	Enter your choice:102. Enter Customer Name: John Which cashier would you like to handle your purchase? Select a Cashier number. (1,2,3) :3	Display "John added to the queue 3 successfully"	Display "John added to the queue 3 successfully"	Pass
4	Remove customer "Jane" from queue 2. 103 or RCQ Enter cashier number (1,2,3): 2. Enter customer index to remove (0,1,2,3) :0	Display "customer Jane removed from cashier 2"	Display "customer Jane removed from cashier 2"	pass
.5.	Remove Served customer from queue 104 or PCQ.	Display "customer removed from queue "	Display "customer removed from queue "	pass

			(The Customer peter removed from queue 1)	
6	View customers sorted in alphabetical order. 105 or VCS	Display "customer sorted in alphabetical order, John.	Display "customer sorted in alphabetical order, John.	pass

7	Store program Data into a file. 106 or SPD	Display "Programme Data stored successfully".	Display "Programme Data stored successfully".	Pass
8	Load program data from file 107 or LPD	Display "---Loading Data ---" Queue 3- John	Display "---Loading Data ---" Queue 3- John	Pass
9	View Remaining burgers stock 108 or STK.	Display "Remaining burger stock 45"	Display "Remaining burger stock 45"	Pass
10	Add Burgers to stock 109 or AFS. Enter the number of burgers to add: 15.	Display "15 burgers added to the stock"	Display "15 burgers added to the stock"	Pass
11	Exit the program 999 or EXT	Display "Thank you for choosing us. We'd love to see you Again. Have a Wonderful DAY!!!"	Display "Thank you for choosing us. We'd love to see you Again. Have a Wonderful DAY!!!"	pass

Class version (Task 2)

	Test Case	Expected Result	Actual Result	Pass/Fail
1	Food Queue Initialized Correctly program starts, 100 or VFQ	Display “view all queues”	Display “view all queues”	pass
2	View all Empty queues initialized correctly After the program starts,101 or VEQ	Displays "view all empty queues" for all queues	Displays "view all empty queues" for all queues	pass
3	Add customer “Jane” to Queue 2 102 or ACQ Enter Your choice:102. Enter Customer First Name: Jane Enter Customer Second Name: Dior Enter the number of burgers required:4	Displays "Jane Dior added to queue 1 successfully"	Displays "Customer Jane Dior added to queue 1 successfully"	pass
4	Enter your choice:102. Enter Customer Name: Peter Enter Customer Second Name: Stem Enter the number of burgers required:5	Displays "Peter Stem added to queue 2 successfully"	Displays "Peter Stem added to queue 2 successfully"	pass
5	Enter your choice:102. Enter Customer Name: John Enter Customer Second Name: Steve Enter the number of burgers required:11	Displays "John Steve added to queue 3 successfully"	Displays "John Steve added to queue 3 successfully"	pass
6	Remove customer “Peter Stem” from queue 2. 103 or RCQ Enter cashier number (1,2,3): 2. Enter customer index to remove (0,1,2,3) :0	Displays "Customer Peter Stem removed from cashier 2"	Displays "Customer Peter Stem removed from cashier 2"	pass

7	Remove Served customer from queue 104 or PCQ.	Displays "Customer Jane Dior removed from queue 1"	Displays "Customer Jane Dior removed from queue 1"	pass
8	View customers sorted in alphabetical order. 105 or VCS	Displays "Customers sorted in alphabetical order: John Steve"	Displays "Customers sorted in alphabetical order: John Steve"	pass
9	Store program Data into a file. 106 or SPD	Displays "Program data stored successfully"	Displays "Program data stored successfully"	pass
10	Load program data from file 107 or LPD	Displays "--- Loading Data--- Queue 3: John Steve"	Displays "--- Loading Data--- Queue 3: John Steve"	pass
11	View Remaining burgers stock 108 or STK.	Displays "Remaining burgers stock: 46"	Displays "Remaining burgers stock: 46"	pass
12	Add Burgers to stock 109 or AFS. Enter the number of burgers to add: 15.	Displays "15 burgers added to the stock" New Stock:61	Displays "15 burgers added to the stock" New Stock:61	pass
13	View income of queues 110 of IFQ	Displays the income of each cashier queue	Displays the income of each cashier queue	pass
14	View Waiting queue	Display waiting queue	Display waiting queue:(empty or customer)	pass
15	View the Status of the Queue	Display “the status of the queue”	Display “the status of the queue”	fail
16	Exit the program 999 or EXT	Displays "Thank you for choosing us. Have a Wonderful DAY!!!"	Displays "Thank you for choosing us. Have a Wonderful DAY!!!"	pass

Discussion

<<Discussion of how you chose your test cases to ensure that your tests cover all aspects of your program>>

Array Part (Task 1)

1. The constants :

Current Burger Stock: Represents the initial stock of burgers available at the burger centre. It is set to 50.

Burgers per Order: Specifies the number of burgers a customer can order in a single order. It is set to 5.

These constants are declared at the beginning of the FoodiesBurgerCenter Array class and are marked as final, indicating that their values cannot be changed once assigned. They are used throughout the code to maintain consistent values for the current burger stock and the number of burgers per order.

2. Variables:

Current_Burger_Stock: Represents the initial stock of burgers available at the burger centre.

Burgers_per_Order: Specifies the number of burgers a customer can order in a single order.

cashierQueue1, cashierQueue2, cashierQueue3: Arrays to store customer names in the queues for each cashier.

burgersAvailability: Tracks the current stock of burgers available.

3. Methods:

- **main(String[] args):** This method serves as the entry point of the program. It displays a menu of options to the user and executes different actions based on the user's choice. It utilizes a while loop to continuously prompt the user for input until they choose to exit the program.

- **ViewAllQueues():** This method displays the queues of all cashiers along with icons indicating if the queue is empty or not. It uses the getQueueWithIcons() method to generate the queue representation with symbols.
- **isEmptyQueue(String[] queue):** This method checks if a given queue is empty by iterating through each element of the queue and checking if it is null. It returns true if all elements are null, indicating an empty queue.
- **ViewEmptyQueues():** This method displays the queues that are currently empty by calling isEmptyQueue() for each cashier queue. If a queue is empty, it prints a corresponding message.
- **getQueueWithIcons(String[] queue):** This method returns a queue with icons ('O' or 'X') indicating if a customer is present or not. It iterates through each element of the queue, assigning 'O' if the element is not null (customer present) and 'X' otherwise.
- **storeProgramData():** This method stores the program data, including queue information and current burger stock, into a file named "program_data.txt". It uses a FileWriter to write the data to the file, iterating through each cashier queue, and writing its contents. The current burger stock is also written to the file.
- **LoadData():** This method loads program data from the "program_data.txt" file. It uses a FileReader and BufferedReader to read the data line by line. It then prints the data to the console.
- **AddCustomer(Scanner scanner):** This method adds a customer to a specific cashier queue based on user input. It prompts the user for the customer's name and the desired cashier number. It checks for valid input and adds the customer to the selected cashier's queue if there is space available. It also checks the burger stock availability and displays warnings if it is low or out of stock.
- **ContainsNumberValues(String text):** This method checks if a given text contains numeric values. It iterates through each character in the text and checks if it is a digit using the Character.isDigit() method. If a digit is found, it returns true; otherwise, it returns false.
- **FindAvailableIndex(String[] queue):** This method finds an available index in the given queue array. It iterates through the array and returns the index of the first null element, indicating an available position in the queue.
- **RemoveCustomer(Scanner scanner):** This method removes a customer from a specific cashier queue based on user input. It prompts the user for the cashier number and customer index to remove. It validates the inputs and removes the customer if they exist. If the queue is empty or the inputs are invalid, appropriate messages are displayed.

- **RemoveServedCustomer():** This method removes the first served customer from the cashier queues. It checks each cashier queue sequentially, removes the first customer in a non-empty queue, and shifts the remaining customers to the left. It also reduces the burger stock by the number of burgers per order for each served customer.
- **reduceBurgerStock(int amount):** This method reduces the burger stock by the specified amount. It checks if there are enough burgers in stock and updates the stock accordingly. If the stock is empty, it displays an appropriate message.
- **shiftQueueLeft(String[] queue):** This method shifts the elements of the queue array one position to the left. It starts from the first index, assigns each element to the previous index, and sets the last index to null.
- **ViewCustomersSorted():** This method creates an array of all customers in all queues, sorts them alphabetically, and prints them out. It combines all customer names from each queue into a single array, sorts the array using the sortCustomers() method, and then prints the sorted names.
- **sortCustomers(String[] customers, int size):** This method implements a simple bubble sort algorithm to sort an array of customers in alphabetical order. It compares adjacent elements and swaps them if they are out of order. It continues this process until the array is sorted.
- **ViewRemainingStock():** This method prints out the remaining stock of burgers by accessing the burgersAvailability variable.
- **AddBurgersToStock(Scanner scanner):** This method adds burgers to the stock based on user input. It prompts the user to the number of burgers to add, updates the burgersAvailability variable, and displays the new stock level.

Overall, the program provides a menu-driven interface for managing the burger centre's queues, customer operations, and stock management. It utilizes arrays, loops, conditionals, file I/O, and helper methods to achieve its functionality.

Discussion of Class Version

1. **Constants:** Current_Burger_Stock: Represents the total number of burgers currently in stock. It is set to a specific value (50) and indicates the initial stock availability.
2. **Variables:** cashierQueue1, cashierQueue2, cashierQueue3: Instances of the FoodQueue class representing three different queues for cashiers. These variables hold the customers waiting to be served by the respective cashiers.

   burgersAvailability: Represents the remaining number of burgers in stock. It is initially set to the value of Current_Burger_Stock and gets updated as customers are served, or burgers are added to the stock.

   waitingList: An instance of the CircularQueue class representing a circular queue to hold customers on the waiting list. When all cashier queues are full, additional customers are placed in this waiting list until a cashier becomes available.

3. **Methods:** main(String[] args): The entry point of the program. It contains a while loop that displays a menu of options to the user and calls corresponding methods based on the user's choice.
   - **viewAllQueues():** Prints the status of all queues, including the cashiers and their respective customers. It displays the customers currently in each cashier queue.
   - **viewEmptyQueues():** Prints the queues that are currently empty, i.e., the cashiers with no customers waiting in their queues.
   - **storeProgramData():** Stores the program data, which includes the contents of all queues, into a file named "program_data.txt". It uses a FileWriter to write the data to the file.
   - **loadData():** Loads the program data from the "program_data.txt" file and prints it. It uses a FileReader to read the data from the file and a BufferedReader to read the lines of the file.
   - **addCustomer(Scanner scanner):** Prompts the user to enter the details of a new customer (first name, last name, and number of burgers required). It creates a Customer object and adds it to the shortest available cashier queue. If all cashier queues are full, the customer is added to the waiting list if there is space.

- **isStringOnly(String input):** Checks if a given input consists of only letters (alphabets). It is used to validate the first name and last name inputs to ensure they don't contain integers.
- **getShortestQueue():** Returns the shortest queue (with available capacity) among the cashier queues. It checks the size of each queue and returns the one with the smallest size. If multiple queues have the same smallest size, the first encountered one is returned.
- **removeCustomer(Scanner scanner):** Prompts the user to enter a cashier number (1, 2, or 3) and a customer index to remove from the corresponding cashier queue. It removes the customer at the specified index if it is valid.
- **removeServedCustomer():** Removes the first served customer from the cashier queues. It removes the customer from the first non-empty cashier queue (cashierQueue1, cashierQueue2, or cashierQueue3) and updates the burger stock accordingly. If there are customers in the waiting list, it adds the next customer to the shortest cashier queue if there is space.
- **reduceBurgerStock(int amount):** Reduces the available burger stock by the specified amount. It checks if the stock is sufficient before reducing it and prints a warning if the stock is low or empty.
- **viewCustomersSorted():** Prints all customers in alphabetical order. It creates an ArrayList of customers by combining all customers from the cashier queues and then sorts the list in alphabetical order by the customer's full name.
- **sortCustomers(ArrayList<Customer> customers):** Sorts the customers in alphabetical order. It uses the Collections.sort method and a custom comparator to compare customers based on their full names.
- **viewRemainingStock():** Prints the remaining stock of burgers.
- **addBurgersToStock(Scanner scanner):** Prompts the user to enter the number of burgers to add to the stock. It increases the burgersAvailability variable by the specified amount and prints the updated stock availability.
- **printIncome():** Calculates and prints the income of each cashier queue. It uses the getIncome method of the FoodQueue class to calculate the income based on the price of a burger (BURGER_PRICE). The income represents the total amount earned by selling burgers from each cashier queue.

These constants, variables, and methods together facilitate the management of customers, queues, stock, and income in the FoodCenterr program.

FoodQueue Class

1. Constants: capacity: Represents the maximum capacity of the food queue. It is set during the initialization of the FoodQueue object.

2. Variables: customers: An ArrayList that stores the customers in the food queue.

capacity: Represents the maximum capacity of the food queue.

3. Methods:

- **getSize():** Returns the current size of the food queue, which is the number of customers in the queue.
- **getCapacity():** Returns the maximum capacity of the food queue.
- **isEmpty():** Checks if the food queue is empty by checking if the customers list is empty. Returns true if the food queue is empty, false otherwise.
- **isFull():** Checks if the food queue is full by comparing the size of the customers list with the capacity. Returns true if the food queue is full, false otherwise.
- **getCustomers():** Returns the ArrayList of customers in the food queue.
- **getCashierNumber():** Returns the cashier number associated with the food queue. It compares the current instance of the FoodQueue with the predefined cashier queues (FoodCenterr.cashierQueue1, FoodCenterr.cashierQueue2, FoodCenterr.cashierQueue3) and returns the corresponding cashier number (1, 2, or 3). If the instance does not match any cashier queues, it returns -1.
- **addCustomer(Customer customer):** Adds a customer to the food queue if it is not full.

- **removeCustomer(int index):** Removes and returns the customer at the specified index from the food queue if the index is valid.

- **isValidIndex(int index):** Checks if the provided index is valid for accessing the customers list.

- **getIncome(double burgerPrice):** Calculates and returns the total income generated by the customers in the food queue. It multiplies the number of burgers required by each customer with the specified burger price and accumulates the income.

- **getDataString(String prefix):** Generates a formatted string representation of the customers' data in the queue. It iterates over each customer and appends the prefix, customer's full name, number of burgers required, and a new line character. Returns the formatted string representation.

- **toString():** Overrides the toString() method to represent the food queue visually. It creates a string representation using "O" to represent occupied positions (customers) and "X" to represent unoccupied positions. The resulting string represents the status of each position in the queue.

  These constants, variables, and methods are used to manage the customers in the food queue, retrieve information about the queue's state, add or remove customers, calculate income, and generate formatted string representations of the queue's data.

Customer Class

1. **Variables:**
   - **firstName:** Represents the first name of the customer.
   - **lastName:** Represents the last name of the customer.
   - **burgersRequired:** Represents the number of burgers required by the customer.
2. **Methods:**
   - **Customer(String firstName, String lastName, int burgersRequired):** The constructor method initializes the Customer object with the provided first name, last name, and the number of burgers required.
   - **getFullName():** Returns the full name of the customer by concatenating the first name and last name.
   - **getBurgersRequired():** Returns the number of burgers required by the customer.

The Customer class is responsible for storing information about a customer, including their first name, last name, and the number of burgers they require. The constructor is used to initialize the customer object with the provided information. The getFullName() method returns the concatenated full name of the customer, and the getBurgersRequired() method returns the number of burgers required by the customer.

Code :

Array Part(Task 1)

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;
import java.util.Arrays;
import java.io.BufferedReader;
import java.io.FileReader;

public class FoodiesBurgerCenter_Array{
    // Define constants for current burger stock and burgers per order
    private static final int Current_Burger_Stock = 50;
    private static final int Burgers_per_Order = 5;
    // Define arrays to store customer names in the queues
    private static String[] cashierQueue1 = new String[2];
    private static String[] cashierQueue2 = new String[3];
    private static String[] cashierQueue3 = new String[5];
    // Initialize variable to track current burger stock
    private static int burgersAvailability = Current_Burger_Stock;

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        while (true) {
            // Display menu of options
            System.out.println("\t\t\t\t\t\t\t\t\t\t\t\t\t*****  FOODIES FAVE
BURGER CENTER  *****");
            System.out.println("-----------------------------------------------
-----------------------------------");
            System.out.println("\t\t\t\t\t\t\tPLEASE SELECT AN OPTION");
            System.out.println("\t\t\t\t\t\t\t100 or VFQ: view all
Queues.");
            System.out.println("\t\t\t\t\t\t\t101 or VEQ: View all Empty
Queues.");
            System.out.println("\t\t\t\t\t\t\t102 or ACQ: Add customer to a
Queue.");
            System.out.println("\t\t\t\t\t\t\t103 or RCQ: Remove a customer
from Queue.(From a specific location)");
            System.out.println("\t\t\t\t\t\t\t104 or PCQ: Remove a served
customer.");
            System.out.println("\t\t\t\t\t\t\t105 or VCS: View Customers
Sorted in alphabetical order (Do not use library sort routine)");
            System.out.println("\t\t\t\t\t\t\t106 or SPD: Store Program
Data into files");
            System.out.println("\t\t\t\t\t\t\t107 or LPD: Load Program Data
from file.");
            System.out.println("\t\t\t\t\t\t\t108 or STK: View Remaining
burgers Stock.");
            System.out.println("\t\t\t\t\t\t\t109 or AFS: Add burgers to
Stock.");
            System.out.println("\t\t\t\t\t\t\t999 or EXT: Exit the
Program.");
```

```
            System.out.println("-----------------------------------------------
----------------------------------------");
            // Prompt user for choice
            System.out.print("Enter Your Choice: ");
            // Prompt the user to enter their choice from the menu of options

            String choice = scanner.nextLine().toLowerCase();

            switch (choice) {
                case "100":
                case "vfq":
                    ViewAllQueues();
                    break;
                case "101":
                case "veq":
                    ViewEmptyQueues();
                    break;
                case "102":
                case "acq":
                    AddCustomer(scanner);
                    break;
                case "103":
                case "rcq":
                    RemoveCustomer(scanner);
                    break;
                case "104":
                case "pcq":
                    RemoveServedCustomer();
                    break;
                case "105":
                case "vcs":
                    ViewCustomersSorted();
                    break;
                case "106":
                case "spd":
                    storeProgramData();
                    break;
                case "107":
                case "lpd":
                    LoadData();
                    break;
                case "108":
                case "stk":
                    ViewRemainingStock();
                    break;
                case "109":
                case "afx":
                    AddBurgersToStock(scanner);
                    break;
                case "999":
                case "ext":
                    System.out.println("Thank you for choosing us. We'd love
to see you Again. Have a Wonderful DAY!!!");
                    return;
                default:
                    System.out.println("Invalid choice. Please try again.");
                    break;
```

```
            }
        }
    }
    // This method displays the queues of all cashiers along with icons
indicating if the queue is empty or not.
    private static void ViewAllQueues() {
        System.out.println("\t\t\t\t\t\t*****************");
        System.out.println("\t\t\t\t\t\t*   CASHIERS    *");
        System.out.println("\t\t\t\t\t\t*****************");
        System.out.println("\t\t\t\t\t\tCashier 1: " +
Arrays.toString(getQueueWithIcons(cashierQueue1)));
        System.out.println("\t\t\t\t\t\tCashier 2: " +
Arrays.toString(getQueueWithIcons(cashierQueue2)));
        System.out.println("\t\t\t\t\t\tCashier 3: " +
Arrays.toString(getQueueWithIcons(cashierQueue3)));
    }

    // This method checks if a given queue is empty or not.
    private static boolean isEmptyQueue(String[] queue) {
        // Iterate through each element in the queue.
        for (String customer : queue) {
            if (customer != null) {
                // Check if the current element is not null (indicating the
presence of a customer).
                return false;
            }
        }
        return true;
    }

    // This method displays the queues that are currently empty.
    private static void ViewEmptyQueues() {
        System.out.println("Empty Queues: ");
        if (isEmptyQueue(cashierQueue1)) {
            System.out.println("Cashier 1 is Empty");
        }
        if (isEmptyQueue(cashierQueue2)) {
            System.out.println("Cashier 2 is Empty ");
        }
        if (isEmptyQueue(cashierQueue3)) {
            System.out.println("Cashier 3 is Empty ");
        }
        if (!isEmptyQueue(cashierQueue1) && !isEmptyQueue(cashierQueue2) &&
!isEmptyQueue(cashierQueue3)) {
            System.out.println("No Empty Queues to Show... At least one
customer in each Queue.");
        }
    }

    // This method returns a queue with icons indicating if a customer is
present in the queue or not.
    // This method takes in an array of Strings called "queue" as a
parameter.
    private static String[] getQueueWithIcons(String[] queue) {
        // A new array of Strings called "queueWithSymbols" is created with
the same length as the "queue" array.
        String[] queueWithSymbols = new String[queue.length];
```

```
        // A for loop is used to iterate through each element in the "queue"
array.
        for (int i = 0; i < queue.length; i++) {
            if (queue[i] == null) {
                // If the current element in the "queue" array is null, an
"X" is added to the corresponding index in the "queueWithSymbols" array.
                queueWithSymbols[i] = "X";
            } else {
                // If the current element in the "queue" array is not null,
an "O" is added to the corresponding index in the "queueWithSymbols" array.
                queueWithSymbols[i] = "O";
            }
        }
        // The "queueWithSymbols" array is returned.
        return queueWithSymbols;
    }
    // This method stores the program data to a file.
    private static void storeProgramData() {
        try {
            FileWriter writer = new FileWriter("program_data.txt");

            // Write the data for QUEUE1
            // Create a new array called queue1Array with the same length as
the cashierQueue1 array.
            String[] queue1Array = new String[cashierQueue1.length];
            // Initialize a variable called queue1Count to keep track of the
number of elements in the queue1Array.
            int queue1Count = 0;
            for (String customer : cashierQueue1) {
                // Iterate through each element in the cashierQueue1 array.
                if (customer != null) {
                    // Check if the current element is not null.
                    queue1Array[queue1Count] = customer;
                    // Assign the current element to the corresponding index
in the queue1Array.
                    queue1Count++;
                    // Increment the queue1Count variable to keep track of
the number of elements in the queue1Array.
                }
            }
            writer.write("QUEUE1:" +
Arrays.toString(Arrays.copyOf(queue1Array, queue1Count)) + "\n");

            // Write the data for QUEUE2
            String[] queue2Array = new String[cashierQueue2.length];
            int queue2Count = 0;
            for (String customer : cashierQueue2) {
                if (customer != null) {
                    queue2Array[queue2Count] = customer;
                    queue2Count++;
                }
            }
            writer.write("QUEUE2:" +
Arrays.toString(Arrays.copyOf(queue2Array, queue2Count)) + "\n");

            // Write the data for QUEUE3
            // Create a new array called queue3Array with the same length as
```

```
the cashierQueue3 array.
            String[] queue3Array = new String[cashierQueue3.length];
            // Initialize a variable called queue3Count to keep track of the
number of elements in the queue3Array.

            int queue3Count = 0;
            for (String customer : cashierQueue3) {
                // Iterate through each element in the cashierQueue3 array.
                if (customer != null) {
                    // Check if the current element is not null.
                    queue3Array[queue3Count] = customer;
                    // Assign the current element to the corresponding index
in the queue3Array.
                    queue3Count++;
                    // Increment the queue3Count variable to keep track of
the number of elements in the queue3Array.
                }
            }
            writer.write("QUEUE3:" +
Arrays.toString(Arrays.copyOf(queue3Array, queue3Count)) + "\n");

            // Write the data for current_burger_stock
            writer.write("Current Burger Stock:" + burgersAvailability +
"\n");

            // Close the FileWriter object called "writer".
            writer.close();

            // Print a message to the console indicating that the program
data has been stored successfully.
            System.out.println("Program data stored successfully.");
        } catch (IOException e) {
            // If an IOException is thrown, print an error message to the
console with the specific error message provided by the exception.
            System.out.println("Error storing program data: " +
e.getMessage());
        }
    }
    // This method is used to load data from a text file
    private static void LoadData() {
        System.out.println("---Loading Data---");
        try {
            // FileReader is used to read data from a file
            FileReader fileReader = new FileReader("program_data.txt");
            // BufferedReader is used to read data from a character input
stream
            BufferedReader bufferedReader = new BufferedReader(fileReader);
            String line;
            // Reading data line by line until the end of the file is reached
            while ((line = bufferedReader.readLine()) != null) {
                System.out.println(line);
            }
            // Closing the BufferedReader
            bufferedReader.close();
            System.out.println("Data Loaded Successfully!");
            // This catch block handles any IOException that might occur
while loading data
```

```
        } catch (IOException e) {
            // Printing an error message if an exception is caught
            System.out.println("ERROR while loading data");
        }
    }
    // This method is called "AddCustomer" and takes in a Scanner object
called "scanner" as a parameter.
    private static void AddCustomer(Scanner scanner) {
        System.out.print("Enter customer name: ");
        String customerName = scanner.nextLine();

        // Check if the customer name is valid
        if (customerName == null || customerName.isEmpty() ||
customerName.trim().isEmpty()) {
            System.out.println("Missing Customer Name. Please provide a name
before proceeding....");
            return;
        }

        // Check if the customer name contains numeric characters
        if (ContainsNumberValues(customerName)) {
            System.out.println("The customer name cannot contain INTEGERS.
Please re-enter!!");
            return;
        }

        System.out.print("Which cashier would you like to handle your
purchase? Select a Cashier number (1, 2, or 3): ");
        int preferredCashier = scanner.nextInt();
        scanner.nextLine(); // Consume the newline character

        // Select the appropriate cashier queue based on the user's choice
        String[] selectedCashier = null;
        int maxQueueSize = 0;
        switch (preferredCashier) {
            case 1:
                selectedCashier = cashierQueue1;
                maxQueueSize = 2;
                break;
            case 2:
                selectedCashier = cashierQueue2;
                maxQueueSize = 3;
                break;
            case 3:
                selectedCashier = cashierQueue3;
                maxQueueSize = 5;
                break;
            default:
                System.out.println("The cashier you have chosen is not valid.
Please choose a given cashier number");
                return;
        }

        // Check if there is an available index in the selected cashier queue
        int availableIndex = FindAvailableIndex(selectedCashier);
        if (availableIndex != -1) {
            // Add the customer to the selected cashier queue
```

```
            selectedCashier[availableIndex] = customerName;
            System.out.println("Customer " + customerName + " added to
Cashier " + preferredCashier + ".");
        } else {
            System.out.println("Cashier " + preferredCashier + " is full.
Sorry, the selected cashier is not available. Please choose another one.
Customer is not added.");
        }

        // Check if the number of available burgers is low or out of stock
        if (burgersAvailability <= 10) {
            System.out.println("Warning: Low stock! Remaining burgers: " +
burgersAvailability);
        }
        if (burgersAvailability == 0) {
            System.out.println("Unfortunately!!! We are out of stock for
burgers at the moment.");
        }
    }
    private static boolean ContainsNumberValues(String text) {
        // Iterate through each character in the text
        for (char c : text.toCharArray()) {
            // Check if the character is a digit
            if (Character.isDigit(c)) {
                return true;// If a digit is found, return true
            }
        }
        return false;
    }
    private static int FindAvailableIndex(String[] queue) {
        // Iterate through each index in the queue array
        for (int i = 0; i < queue.length; i++) {
            // Check if the current index is null
            if (queue[i] == null) {
                return i;// If an available index is found, return it
            }
        }
        return -1;
    }
    // This method is called "RemoveCustomer" and takes in a Scanner object
called "scanner" as a parameter.
    private static void RemoveCustomer(Scanner scanner) {
        System.out.print("Enter cashier number (1, 2, or 3): ");
        // The user's input is read as an integer called "cashierNumber".
        int cashierNumber = scanner.nextInt();
        scanner.nextLine(); // Consume the newline character
// A String array called "selectedCashier" is initialized to null.
        String[] selectedCashier = null;
// A switch statement is used to set the "selectedCashier" variable based on
the user's chosen cashier.
        switch (cashierNumber) {
            case 1:
                selectedCashier = cashierQueue1;
                break;
            case 2:
                selectedCashier = cashierQueue2;
                break;
```

```
            case 3:
                selectedCashier = cashierQueue3;
                break;
            default:
                System.out.println("Invalid cashier number. Customer not
removed. Please check the cashier number...");
                return;
        }

// An if statement is used to check if the "selectedCashier" array is empty.
        if (isEmptyQueue(selectedCashier)) {
            // If the "selectedCashier" array is empty, this line prints a
message to the console indicating that there are no customers to remove.
            System.out.println("Cashier " + cashierNumber + " is already
empty. No customers to remove.");
        } else {
            // If the "selectedCashier" array is not empty, this line prints
a message to the console asking the user to enter a customer index to remove.
            System.out.print("Enter customer index to remove (0 to " +
(selectedCashier.length - 1) + "): ");
            // The user's input is read as an integer called "customerIndex".
            int customerIndex = scanner.nextInt();
            scanner.nextLine(); // Consume the newline character
// An if statement is used to check if the "customerIndex" is a valid index
in the "selectedCashier" array and if there is a customer at that index.
            if (customerIndex >= 0 && customerIndex < selectedCashier.length
&& selectedCashier[customerIndex] != null) {
                // If the "customerIndex" is valid and there is a customer at
that index, the customer is removed from the "selectedCashier" array and the
removed customer's name is stored in a String called "removedCustomer".
                String removedCustomer = selectedCashier[customerIndex];
                selectedCashier[customerIndex] = null;
                // This line prints a message to the console indicating that
the customer has been removed from the chosen cashier's queue.
                System.out.println("Customer " + removedCustomer + " removed
from Cashier " + cashierNumber + ".");
            } else {
                // If the "customerIndex" is not valid or there is no
customer at that index, this line prints an error message to the console and
the method returns without removing a customer.
                System.out.println("Invalid customer index. Customer can not
be removed.");
            }
        }
    }
    // This method is called "RemoveServedCustomer" and does not take in any
parameters.
    private static void RemoveServedCustomer() {
        // Check if Cashier 1 has customers
        if (!isEmptyQueue(cashierQueue1)) {
            String removedCustomer = cashierQueue1[0]; // Get the first
customer in Cashier 1
            shiftQueueLeft(cashierQueue1); // Shift the queue to the left to
remove the served customer
            reduceBurgerStock(Burgers_per_Order); // Reduce the burger stock
by the number of burgers per order
            System.out.println("Customer " + removedCustomer + " removed from
```

```
Cashier 1.");

            // Check if the burger stock is low
            if (burgersAvailability <= 10) {
                System.out.println("Warning: Low stock! Remaining burgers: "
+ burgersAvailability);
            }

            // Check if the burger stock is empty
            if (burgersAvailability == 0) {
                System.out.println("Burger stock is empty...");
            }
        }
        // Check if Cashier 2 has customers
        else if (!isEmptyQueue(cashierQueue2)) {
            String removedCustomer = cashierQueue2[0]; // Get the first
customer in Cashier 2
            shiftQueueLeft(cashierQueue2); // Shift the queue to the left to
remove the served customer
            reduceBurgerStock(Burgers_per_Order); // Reduce the burger stock
by the number of burgers per order
            System.out.println("Customer " + removedCustomer + " removed from
Cashier 2.");

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

            if (burgersAvailability == 0) {
                System.out.println("Burger stock is empty...");
            }
        }
        // Check if Cashier 3 has customers
        else if (!isEmptyQueue(cashierQueue3)) {
            String removedCustomer = cashierQueue3[0]; // Get the first
customer in Cashier 3
            shiftQueueLeft(cashierQueue3); // Shift the queue to the left to
remove the served customer
            reduceBurgerStock(Burgers_per_Order); // Reduce the burger stock
by the number of burgers per order
            System.out.println("Customer " + removedCustomer + " removed from
Cashier 3.");

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

            if (burgersAvailability == 0) {
                System.out.println("Burger stock is empty...");
            }
        }
        else {
            System.out.println("All cashiers are empty. No customers to
remove.");
        }
```

```
    }

    private static void reduceBurgerStock(int amount) {
        // Check if there are enough burgers in stock
        if (burgersAvailability >= amount) {
            burgersAvailability -= amount; // Reduce the burger stock by the
specified amount
        } else {
            System.out.println("Cannot remove a customer. No burgers in
stock.");
        }
    }

    // This method is called "shiftQueueLeft" and takes in a String array
called "queue" as a parameter.
    // A for loop is used to iterate through each element in the "queue"
array except for the last element.
    private static void shiftQueueLeft(String[] queue) {
        for (int i = 0; i < queue.length - 1; i++) {
            // Loop through the entire array except for the last element
            queue[i] = queue[i + 1];// Shift each element one index to the
left
        }
        queue[queue.length - 1] = null; // Set the last element to null to
remove any references to the previous first element
    }
    // This method creates an array of all customers in all queues, sorts
them alphabetically, and prints them out
    private static void ViewCustomersSorted() {
        String[] allCustomers = new String[cashierQueue1.length +
cashierQueue2.length + cashierQueue3.length];
        // Create an array to hold all customers in all queues
        int index = 0;// Keep track of the number of customers added to the
array so far
        for (String customer : cashierQueue1) {// Loop through all customers
in queue 1
            if (customer != null) {// If the customer is not null (i.e.,
there is a customer in that position in the queue)
                allCustomers[index++] = customer;// Add the customer to the
array and increment the index
            }
        }
        for (String customer : cashierQueue2) {// Loop through all customers
in queue 2
            if (customer != null) {// If the customer is not null
                allCustomers[index++] = customer;// Add the customer to the
array and increment the index
            }
        }
        for (String customer : cashierQueue3) {// Loop through all customers
in queue 3
            if (customer != null) { // If the customer is not null
                allCustomers[index++] = customer;// Add the customer to the
array and increment the index
            }
        }
```

```
        if (index == 0) {// If there are no customers in any of the queues
            System.out.println("No customers in the queues.");
        } else {
            sortCustomers(allCustomers, index);// Sort the array of customers
alphabetically
            System.out.println("Customers Sorted in Alphabetical Order:");
            for (int i = 0; i < index; i++) {// Loop through all customers in
the array
                System.out.println(allCustomers[i]);
            }
        }
    }
    // This method sorts an array of customers in alphabetical order
    private static void sortCustomers(String[] customers, int size) {
        for (int i = 0; i < size - 1; i++) {// Loop through the array from
the beginning, up to the second-to-last element
            for (int j = 0; j < size - i - 1; j++) {// Loop through the array
from the beginning, up to the second-to-last element minus the number of
times we've already looped through the array
                if (customers[j].compareTo(customers[j + 1]) > 0) {// If the
current element is greater than the next element in the array
                    String temp = customers[j];// Swap the current element
with the next element in the array
                    customers[j] = customers[j + 1];
                    customers[j + 1] = temp;
                }
            }
        }
    }
    // This method prints out the remaining stock of burgers
    private static void ViewRemainingStock() {
        System.out.println("Remaining Burgers Stock: " +
burgersAvailability);
    }
    // This method adds burgers to the stock
    private static void AddBurgersToStock(Scanner scanner) {
        System.out.print("Enter the number of Burgers to Add: ");
        int burgersToAdd = scanner.nextInt();// Get the number of burgers to
add from the user
        scanner.nextLine(); // Consume the newline character

        burgersAvailability += burgersToAdd;// Add the burgers to the stock
        System.out.println("Added " + burgersToAdd + " burgers to the Stock.
New stock is : " + burgersAvailability);
        // Print out a message indicating how many burgers were added to the
stock, and what the new stock level is
    }
}
```

code:

Class Version (Task 2)

```
import java.util.Scanner;
import java.util.ArrayList;
import java.io.FileWriter;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Collections;
import java.util.Comparator;

public class FoodCenterr {
    // Constants
    private static final int Current_Burger_Stock = 50; // Total number of
burgers in stock


    private static final int QUEUE1_CAPACITY = 2; // Capacity of Queue 1
    private static final int QUEUE2_CAPACITY = 3; // Capacity of Queue 2
    private static final int QUEUE3_CAPACITY = 5; // Capacity of Queue 3

    private static final double BURGER_PRICE = 650.00;// Price of a burger

    // Queues
    public static final FoodQueue cashierQueue1 = new
FoodQueue(QUEUE1_CAPACITY);// Queue 1 for cashier
    public static final FoodQueue cashierQueue2 = new
FoodQueue(QUEUE2_CAPACITY);// Queue 2 for cashier
    public static final FoodQueue cashierQueue3 = new
FoodQueue(QUEUE3_CAPACITY);// Queue 3 for cashier

    private static int burgersAvailability = Current_Burger_Stock;//
Remaining burgers in stock
    // Waiting List
    private static final CircularQueue<Customer> waitingList = new
CircularQueue<>(QUEUE1_CAPACITY + QUEUE2_CAPACITY + QUEUE3_CAPACITY);

    // Circular queue to hold customers on the waiting list
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("\t\t\t\t\t\t\t\t\t\t\t\t\t\t*****  FOODIES FAVE
BURGER CENTER  *****");
            System.out.println("-----------------------------------------------
---------------------------------------");
            System.out.println("\t\t\t\t\t\t\tPLEASE SELECT AN OPTION");
            System.out.println("\t\t\t\t\t\t\t100 or VFQ: view all
Queues.");
```

```
            System.out.println("\t\t\t\t\t\t\t\t101 or VEQ: View all Empty
Queues.");
            System.out.println("\t\t\t\t\t\t\t\t102 or ACQ: Add customer to a
Queue.");
            System.out.println("\t\t\t\t\t\t\t\t103 or RCQ: Remove a customer
from Queue.(From a specific location)");
            System.out.println("\t\t\t\t\t\t\t\t104 or PCQ: Remove a served
customer.");
            System.out.println("\t\t\t\t\t\t\t\t105 or VCS: View Customers
Sorted in alphabetical order (Do not use library sort routine)");
            System.out.println("\t\t\t\t\t\t\t\t106 or SPD: Store Program
Data into files");
            System.out.println("\t\t\t\t\t\t\t\t107 or LPD: Load Program Data
from file.");
            System.out.println("\t\t\t\t\t\t\t\t108 or STK: View Remaining
burgers Stock.");
            System.out.println("\t\t\t\t\t\t\t\t109 or AFS: Add burgers to
Stock.");
            System.out.println("\t\t\t\t\t\t\t\t999 or EXT: Exit the
Program.");
            System.out.println("-----------------------------------------------
---------------------------------------");
            // Prompt user for choice
            System.out.print("Enter Your Choice: ");

            String input = scanner.nextLine();

            input = input.trim().toLowerCase();

            switch (input) {
                case "100":
                case "vfq":
                    viewAllQueues();// Calls the method to view all queues
                    break;
                case "101":
                case "veq":
                    viewEmptyQueues();// Calls the method to view empty
queues
                    break;
                case "102":
                case "acq":
                    addCustomer(scanner);
                    break;
                case "103":
                case "rcq":
                    removeCustomer(scanner);
                    break;
                case "104":
                case "pcq":
                    removeServedCustomer();
                    break;
                case "105":
                case "vcs":
                    viewCustomersSorted();
                    break;
                case "106":
                case "spd":
```

```
                    storeProgramData();
                    break;
                case "107":
                case "lpd":
                    loadData();
                    break;
                case "108":
                case "stk":
                    viewRemainingStock();
                    break;
                case "109":
                case "afs":
                    addBurgersToStock(scanner);
                    break;
                case "110":
                case "ifq":
                    printIncome();
                    break;
                case "999":
                case "ext":
                    System.out.println("Thank you for choosing us. We'd love
to see you Again. Have a Wonderful DAY!!!");
                    return;
                default:
                    System.out.println("Invalid choice. Please try again.");
                    break;
            }
        }
    }

    // Prints all the queues and their current status
    private static void viewAllQueues() {
        System.out.println();
        System.out.println("\t\t\t\t\t\t*****************");
        System.out.println("\t\t\t\t\t\t*   Cashiers    *");
        System.out.println("\t\t\t\t\t\t*****************");

        System.out.println("\t\t\t\t\t\tCashier 1: " + cashierQueue1);//
Print the contents of cashierQueue1
        System.out.println("\t\t\t\t\t\tCashier 2: " + cashierQueue2);//
Print the contents of cashierQueue2
        System.out.println("\t\t\t\t\t\tCashier 3: " + cashierQueue3);//
Print the contents of cashierQueue3
        System.out.println();
        System.out.println("\t\t\t\t\t\tO-Occupied  X- Not Occupied");
    }

    private static void viewEmptyQueues() {
        System.out.println("Empty Queues:");
        // Print "Cashier 1" if cashierQueue1 is empty
        if (cashierQueue1.isEmpty()) {
            System.out.println("Cashier 1 is Empty");
        }
        // Print "Cashier 2" if cashierQueue1 is empty
        if (cashierQueue2.isEmpty()) {
            System.out.println("Cashier 2 is Empty");
        }
```

```
        // Print "Cashier 3" if cashierQueue1 is empty
        if (cashierQueue3.isEmpty()) {
            System.out.println("Cashier 3 is Empty");
        }
    }

    private static void storeProgramData() {
        try {
            FileWriter writer = new FileWriter("program_data.txt");
            writer.write(cashierQueue1.getDataString("QUEUE1:"));
            writer.write(cashierQueue2.getDataString("QUEUE2:"));
            writer.write(cashierQueue3.getDataString("QUEUE3:"));
            writer.close();
            System.out.println("Program data stored successfully.");
        } catch (IOException e) {
            System.out.println("Error storing program data: " +
e.getMessage());
        }
    }


    private static void loadData() {
        System.out.println("---Loading Data---");
        try {
            FileReader fileReader = new FileReader("program_data.txt"); //
Create a FileReader object to read from the file "program_data.txt"
            BufferedReader bufferedReader = new BufferedReader(fileReader);
// Create a BufferedReader object to read lines from the FileReader
            String line;

            while ((line = bufferedReader.readLine()) != null) {
                System.out.println(line); // Print each line read from the
file
            }

            bufferedReader.close(); // Close the BufferedReader
            System.out.println("Data Loaded Successfully!");
        } catch (IOException e) {
            System.out.println("ERROR while loading data");
        }
    }


    private static void addCustomer(Scanner scanner) {
        System.out.print("Enter customer first name: ");
        String firstName = scanner.nextLine();

        // Validate first name
        while (!isStringOnly(firstName)) {
            System.out.println("Invalid input. The customer name cannot
contain INTEGERS. Please re-enter!!");
            System.out.print("Enter customer first name: ");
            firstName = scanner.nextLine();
        }

        System.out.print("Enter customer last name: ");
        String lastName = scanner.nextLine();
```

```
        // Validate last name
        while (!isStringOnly(lastName)) {
            System.out.println("Invalid input. The customer name cannot
contain INTEGERS. Please re-enter!!");
            System.out.print("Enter customer last name: ");
            lastName = scanner.nextLine();
        }

        int burgersRequired;
        while (true) {
            System.out.print("Enter the number of burgers required: ");
            String burgersInput = scanner.nextLine();

            try {
                burgersRequired = Integer.parseInt(burgersInput);
                break;
            } catch (NumberFormatException e) {
                System.out.println("Invalid input. Please enter a valid
integer.");
            }
        }

        Customer customer = new Customer(firstName, lastName,
burgersRequired);
        FoodQueue shortestQueue = getShortestQueue();

        // Add customer to the shortest queue with available capacity
        if (shortestQueue != null && shortestQueue.getSize() <
shortestQueue.getCapacity()) {
            shortestQueue.addCustomer(customer);
            System.out.println("Customer " + customer.getFullName() + " added
to Cashier " + shortestQueue.getCashierNumber() + ".");
        } else {
            // If no queue has available capacity, add customer to the
waiting list if it's not full
            if (!waitingList.isFull()) {
                waitingList.enqueue(customer);
                System.out.println("All cashiers are full. Added customer " +
customer.getFullName() + " to the waiting list.");
            } else {
                // If waiting list is also full, customer cannot be added
                System.out.println("All cashiers and waiting list are full.
Customer not added.");
            }
        }

        // Check stock availability and display warnings
        if (burgersAvailability <= 10) {
            System.out.println("Warning: Low stock! Remaining burgers: " +
burgersAvailability);
        }
        if (burgersAvailability == 0) {
            System.out.println("Unfortunately!!! We are out of stock for
burgers at the moment.");
        }
    }
```

```
    // Checks if a given input consists of only letters (alphabets)
    private static boolean isStringOnly(String input) {
        for (char c : input.toCharArray()) {
            if (!Character.isLetter(c)) {
                return false;
            }
        }
        return true;
    }


    private static FoodQueue getShortestQueue() {
        FoodQueue shortestQueue = null;

        // Check if cashierQueue1 has available capacity and assign it as the
current shortestQueue if it is
        if (cashierQueue1.getSize() < cashierQueue1.getCapacity()) {
            shortestQueue = cashierQueue1;
        }

        // Check if cashierQueue2 has available capacity and is shorter than
the current shortestQueue (if any), then assign it as the new shortestQueue
        if (cashierQueue2.getSize() < cashierQueue2.getCapacity() &&
(shortestQueue == null || cashierQueue2.getSize() < shortestQueue.getSize()))
{
            shortestQueue = cashierQueue2;
        }

        // Check if cashierQueue3 has available capacity and is shorter than
the current shortestQueue (if any), then assign it as the new shortestQueue
        if (cashierQueue3.getSize() < cashierQueue3.getCapacity() &&
(shortestQueue == null || cashierQueue3.getSize() < shortestQueue.getSize()))
{
            shortestQueue = cashierQueue3;
        }

        return shortestQueue; // Return the shortestQueue found (can be null
if all queues are full)
    }


    private static void removeCustomer(Scanner scanner) {
        System.out.print("Enter cashier number (1, 2, or 3): ");
        int cashierNumber = scanner.nextInt();
        scanner.nextLine();

        FoodQueue selectedCashier = null;
        switch (cashierNumber) {
            case 1:
                selectedCashier = cashierQueue1; // Assign the
selectedCashier as cashierQueue1
                break;
            case 2:
                selectedCashier = cashierQueue2; // Assign the
selectedCashier as cashierQueue2
                break;
```

```
            case 3:
                selectedCashier = cashierQueue3; // Assign the
selectedCashier as cashierQueue3
                break;
            default:
                System.out.println("Invalid cashier number. Customer not
removed.");
                return; // Exit the method if the cashier number is invalid
        }

        if (selectedCashier.isEmpty()) {
            System.out.println("Cashier " + cashierNumber + " is already
empty. No customers to remove.");
        } else {
            System.out.print("Enter customer index to remove (0 to " +
(selectedCashier.getSize() - 1) + "): ");
            int customerIndex = scanner.nextInt();
            scanner.nextLine();

            if (selectedCashier.isValidIndex(customerIndex)) {
                Customer removedCustomer =
selectedCashier.removeCustomer(customerIndex);
                System.out.println("Customer " +
removedCustomer.getFullName() + " removed from Cashier " + cashierNumber +
".");
            } else {
                System.out.println("Invalid customer index. Customer not
removed.");
            }
        }
    }


    private static void removeServedCustomer() {
        if (!cashierQueue1.isEmpty()) {
            // Remove customer from Cashier 1
            Customer removedCustomer = cashierQueue1.removeCustomer(0);
            // Reduce burger stock
            reduceBurgerStock(removedCustomer.getBurgersRequired());
            System.out.println("Customer " + removedCustomer.getFullName() +
" removed from Cashier 1.");
        } else if (!cashierQueue2.isEmpty()) {
            // Remove customer from Cashier 2
            Customer removedCustomer = cashierQueue2.removeCustomer(0);
            // Reduce burger stock
            reduceBurgerStock(removedCustomer.getBurgersRequired());
            System.out.println("Customer " + removedCustomer.getFullName() +
" removed from Cashier 2.");
        } else if (!cashierQueue3.isEmpty()) {
            // Remove customer from Cashier 3
            Customer removedCustomer = cashierQueue3.removeCustomer(0);
            // Reduce burger stock
            reduceBurgerStock(removedCustomer.getBurgersRequired());
            System.out.println("Customer " + removedCustomer.getFullName() +
" removed from Cashier 3.");
        } else {
            // No customers in the cashier queues
```

```
            System.out.println("All cashiers and waiting list are empty. No
customers to remove.");
            return;
        }

        // Check if there are customers in the waiting list
        if (!waitingList.isEmpty()) {
            // Get the next customer from the waiting list
            Customer nextCustomer = waitingList.dequeue();
            // Get the shortest cashier queue
            FoodQueue shortestQueue = getShortestQueue();
            if (shortestQueue != null) {
                // Add the next customer to the shortest queue
                shortestQueue.addCustomer(nextCustomer);
                System.out.println("Next customer from the waiting list, " +
nextCustomer.getFullName() + ", added to Cashier " +
shortestQueue.getCashierNumber() + ".");
            } else {
                System.out.println("All cashiers are full. Next customer from
the waiting list not added.");
            }

            // Reduce burger stock
            reduceBurgerStock(nextCustomer.getBurgersRequired());
            System.out.println("Customer " + nextCustomer.getFullName() + "
removed from the waiting list.");
        }

        // Check burger stock
        if (burgersAvailability <= 10) {
            System.out.println("Warning: Low stock! Remaining burgers: " +
burgersAvailability);
        }
        if (burgersAvailability == 0) {
            System.out.println("Unfortunately!!! We are out of stock for
burgers at the moment.");
        }
    }


    private static void reduceBurgerStock(int amount) {
        // Check if the available burger stock is greater than or equal to
the specified amount
        if (burgersAvailability >= amount) {
            // Reduce the available burger stock by the specified amount
            burgersAvailability -= amount;
        } else {
            // Print a message indicating that there are not enough burgers
in stock
            System.out.println("Cannot remove a customer. No burgers in
stock.");
        }
    }


    private static void viewCustomersSorted() {
        ArrayList<Customer> allCustomers = new ArrayList<>(); // Create an
```

```
ArrayList to store all customers

        allCustomers.addAll(cashierQueue1.getCustomers()); // Add customers
from cashierQueue1 to allCustomers
        allCustomers.addAll(cashierQueue2.getCustomers()); // Add customers
from cashierQueue2 to allCustomers
        allCustomers.addAll(cashierQueue3.getCustomers()); // Add customers
from cashierQueue3 to allCustomers

        if (allCustomers.isEmpty()) {
            System.out.println("Queue is empty."); // Print a message if
allCustomers is empty
        } else {
            sortCustomers(allCustomers); // Sort allCustomers in alphabetical
order
            System.out.println("All Customers Sorted in alphabetical
order:");
            for (Customer customer : allCustomers) {
                System.out.println(customer.getFullName()); // Print the full
name of each customer in allCustomers
            }
        }
    }


    private static void sortCustomers(ArrayList<Customer> customers) {
        // Sort the customers list using a custom comparator
        Collections.sort(customers, new Comparator<Customer>() {
            public int compare(Customer c1, Customer c2) {
                // Compare the full names of the customers and return the
result
                return c1.getFullName().compareTo(c2.getFullName());
            }
        });
    }


    private static void viewRemainingStock() {
        System.out.println("Remaining burgers stock: " +
burgersAvailability);
    }

    private static void addBurgersToStock(Scanner scanner) {
        System.out.print("Enter the number of burgers to add: ");
        int burgersToAdd = scanner.nextInt();
        scanner.nextLine();

        burgersAvailability += burgersToAdd; // Increase the available burger
stock by the specified amount
        System.out.println("Added " + burgersToAdd + " burgers to the
stock."); // Print a message indicating the number of burgers added
        System.out.println("New stock: " + burgersAvailability); // Print the
updated stock availability
    }


    private static void printIncome() {
```

```
        double income1 = cashierQueue1.getIncome(BURGER_PRICE); // Calculate
the income of cashierQueue1
        double income2 = cashierQueue2.getIncome(BURGER_PRICE); // Calculate
the income of cashierQueue2
        double income3 = cashierQueue3.getIncome(BURGER_PRICE); // Calculate
the income of cashierQueue3

        System.out.println("\t\t\t\t\t\t*****************");
        System.out.println("\t\t\t\t\t\tIncome of cashiers"); // Print a
header for the income of each queue
        System.out.println("\t\t\t\t\t\t*****************");
        System.out.println("\t\t\t\t\t\tCashier 1:Rs. " + income1); // Print
the income of cashierQueue1
        System.out.println("\t\t\t\t\t\tCashier 2:Rs. " + income2); // Print
the income of cashierQueue2
        System.out.println("\t\t\t\t\t\tCashier 3:Rs. " + income3); // Print
the income of cashierQueue3
    }
}

class CircularQueue<T> {
    private Object[] elements; // Array to store the elements of the circular
queue
    private int front; // Index of the front element
    private int rear; // Index of the rear element
    private int capacity; // Maximum capacity of the circular queue
    private int size; // Current size of the circular queue

    public CircularQueue(int capacity) {
        this.capacity = capacity; // Set the maximum capacity of the circular
queue
        this.elements = new Object[capacity]; // Create an array with the
specified capacity to hold the elements
        this.front = -1; // Initialize the front index as -1
        this.rear = -1; // Initialize the rear index as -1
        this.size = 0; // Initialize the size as 0
    }


    public boolean isEmpty() {
        return size == 0; // Check if the size of the circular queue is 0
    }

    public boolean isFull() {
        return size == capacity; // Check if the size of the circular queue
is equal to its capacity
    }

    public int getSize() {
        return size; // Return the current size of the circular queue
    }

    public void enqueue(T item) {
        if (isFull()) {
            throw new IllegalStateException("CircularQueue is full."); //
Throw an exception if the circular queue is already full
        }
```

```
        if (isEmpty()) {
            front = 0; // Set the front index to 0 if the circular queue is
empty
        }

        rear = (rear + 1) % capacity; // Update the rear index to the next
position considering the circular nature
        elements[rear] = item; // Store the item at the rear index
        size++; // Increment the size of the circular queue
    }

    public T dequeue() {
        if (isEmpty()) {
            return null; // Return null if the circular queue is empty
        }

        T item = (T) elements[front]; // Get the item at the front index
        elements[front] = null; // Set the element at the front index to null

        if (front == rear) {
            front = -1; // Reset the front index to -1 if the circular queue
becomes empty after dequeueing
            rear = -1; // Reset the rear index to -1 if the circular queue
becomes empty after dequeueing
        } else {
            front = (front + 1) % capacity; // Update the front index to the
next position considering the circular nature
        }

        size--; // Decrement the size of the circular queue
        return item; // Return the dequeued item
    }
}
```

customer class

```
public class Customer {
    private String firstName; // First name of the customer
    private String lastName; // Last name of the customer
    private int burgersRequired; // Number of burgers required by the
customer

    // Constructor
    public Customer(String firstName, String lastName, int burgersRequired) {
        this.firstName = firstName; // Initialize the first name of the
customer
        this.lastName = lastName; // Initialize the last name of the customer
        this.burgersRequired = burgersRequired; // Initialize the number of
burgers required by the customer
    }
```

```
    public String getFullName() {
        return firstName + " " + lastName; // Concatenate the first name and
last name to form the full name of the customer
    }

    public int getBurgersRequired() {
        return burgersRequired; // Return the number of burgers required by
the customer
    }

}
```

FoodQueue Class

```
import java.util.ArrayList;

class FoodQueue {
    private ArrayList<Customer> customers; // ArrayList to store customers in
the food queue
    private int capacity; // Maximum capacity of the food queue

    public FoodQueue(int capacity) {
        this.capacity = capacity; // Set the maximum capacity of the food
queue
        this.customers = new ArrayList<>(capacity); // Create an ArrayList
with the specified capacity to hold the customers
    }


    public int getSize() {
        return customers.size(); // Return the current size of the food queue
    }

    public int getCapacity() {
        return capacity; // Return the maximum capacity of the food queue
    }

    public boolean isEmpty() {
        return customers.isEmpty(); // Check if the food queue is empty and
return a boolean value
    }

    public boolean isFull() {
        return customers.size() == capacity; // Check if the food queue is
full and return a boolean value
    }

    public ArrayList<Customer> getCustomers() {
        return customers; // Return the ArrayList of customers in the food
queue
    }
```

```
    public int getCashierNumber() {
        if (this == FoodCenterr.cashierQueue1) { // Check if the current
instance is the same as the cashierQueue1 instance
            return 1; // Return 1 to indicate the cashier number is 1
        } else if (this == FoodCenterr.cashierQueue2) { // Check if the
current instance is the same as the cashierQueue2 instance
            return 2; // Return 2 to indicate the cashier number is 2
        } else if (this == FoodCenterr.cashierQueue3) { // Check if the
current instance is the same as the cashierQueue3 instance
            return 3; // Return 3 to indicate the cashier number is 3
        } else {
            return -1; // Return -1 if the current instance does not match
any of the cashier queues
        }
    }


    public void addCustomer(Customer customer) {
        if (!isFull()) { // Check if the food queue is not full
            customers.add(customer); // Add the customer to the food queue
        }
    }

    public Customer removeCustomer(int index) {
        if (isValidIndex(index)) { // Check if the index is valid
            return customers.remove(index); // Remove and return the customer
at the specified index
        }
        return null; // Return null if the index is invalid
    }

    public boolean isValidIndex(int index) {
        return index >= 0 && index < customers.size(); // Check if the index
is within the valid range of indices for the customers ArrayList
    }

    public double getIncome(double burgerPrice) {
        double income = 0.0; // Variable to store the total income
        for (Customer customer : customers) { // Iterate over each customer
in the food queue
            income += customer.getBurgersRequired() * burgerPrice; //
Calculate the income by multiplying the number of burgers required by the
burger price and adding it to the total income
        }
        return income; // Return the total income
    }


    public String getDataString(String prefix) {
        // Generates a formatted string representation of the customers' data
in the queue

        StringBuilder sb = new StringBuilder();

        // Iterate over each customer in the queue
        for (Customer customer : customers) {
```

```
            // Append the prefix, customer's full name, burgers required, and
a new line character

sb.append(prefix).append(customer.getFullName()).append(":").append(customer.
getBurgersRequired()).append("\n");
        }

        // Return the final formatted string representation of the customers'
data
        return sb.toString();
    }



    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < capacity; i++) {
            if (i < customers.size()) { // Check if there is a customer at
the current position
                sb.append("O"); // Append "O" to represent an occupied
position
            } else {
                sb.append("X"); // Append "X" to represent an unoccupied
position
            }
            if (i != capacity - 1) { // Check if it's not the last position
                sb.append(" "); // Append a space after each position except
the last one
            }
        }
        return sb.toString(); // Return the resulting string representation
    }

}
```

<<END>>