جامعة بيروت العربية
BEIRUT ARAB UNIVERSITY

Restaurant Management System: An OOP and Data Structures Approach

Amjad Ayman Ziadeh, ID: 202503867

Nour Alhuda Ziadeh, ID: 202503874

CMPS347: Data Structures

Dr. Omair Faraj

Dr. Mustafa Mohammad Chehaitly

December 2, 2025

Abstract

This project presents a complete restaurant management system implemented using object-oriented programming principles and data structures. The goal of the system is to simulate and manage real-world restaurant operations, including maintaining a menu, managing customers, processing orders, prioritizing VIP customers, tracking daily revenues, and storing persistent records using text files.

To maintain consistency in implementation, the project uses Linked Lists, Stacks, and Priority Queues, with files acting as a lightweight simulated database.

The system successfully achieves all intended objectives:

- Designing an extensible OOP architecture using inheritance, composition, and interfaces.
- Implementing core data structures from scratch and integrating them within real-world components.
- Managing customers, VIPs, menu items, orders, and accounting operations.
- Simulating kitchen operations using a custom Priority Queue for VIP-based order handling.
- Ensuring data persistence by saving and loading records from files.

Overall, the project demonstrates how classical data structures can be applied effectively in a functional, domain-specific application.

Table of Contents

Introduction

1.1. Overview

Restaurants rely heavily on efficient coordination between multiple processes: order placement, meal preparation, customer management, and financial tracking. The goal of this project is to model a simplified restaurant management system that mirrors real-world workflows while emphasizing the principles of object-oriented programming and the use of fundamental data structures.

To accomplish this, the system integrates several components: menu management, customer registration, VIP handling, order processing, kitchen operations, and accounting, into a cohesive software application. Each component is carefully designed so that classes interact through well-defined relationships using composition and inheritance, with data structures enabling efficient storage and manipulation of dynamic information.

1.2. Problem Statement

Traditional restaurant workflows involve several challenges:

- Maintaining an organized menu and modifying it dynamically.
- Keeping track of regular and VIP customers with different privileges.
- Managing orders in a fair yet priority-aware manner.
- Maintaining order history for cancellations or reviews.
- Summarizing daily revenues and tracking financial data.
- Storing and retrieving data persistently across different sessions.

The problem is to design a software system that addresses all these tasks while:

1. **Implementing the required data structures manually**, and
2. **Ensuring full integration of these structures with the restaurant model.**

Logic and Processes

2.1. Logic: The system's logic replicates the workflow of a restaurant through the following processes:

1. **Customer Management**
    - Captures customer details (name, phone, VIP status, discount rates).
    - Differentiates VIP and regular customers, applying relevant discounts.
    - Saves and loads customer information from files to ensure data persistence and consistency.
2. **Menu Management**
    - Maintains a dynamic menu of items with attributes such as name, price, and category.
    - Supports adding, removing, and searching for menu items.
    - Persists menu information in files to maintain consistency across sessions.
3. **Order Management**
    - Customers place orders containing multiple menu items.
    - Orders are managed using a **priority queue**, prioritizing VIP customers.
    - Supports order cancellation via a **stack**, enabling undo operations.
    - Calculates total price, applying VIP or day-based discounts (e.g., Friday discount).
4. **Kitchen Management**
    - Serves orders from the priority queue and logs served orders.
    - Displays pending orders to manage kitchen workflow efficiently.
    - Computes daily revenue for accounting purposes.
5. **Accounting & Revenue Tracking**
    - Uses a **linked list** to store daily revenues.
    - Provides summaries of total revenue, best-selling days, and lowest-selling days.
    - Facilitates day-end closure, summing revenue and updating persisted files.
6. **Data Persistence**

- Uses text files to store customers, VIP customers, menu items, and daily revenues.
- Simulates a database to maintain consistency in implementation while avoiding complex database setups.
- Ensures that system state is preserved across program executions.

2.2. Real-Life Applications

- Automates restaurant and café operations.
- Improves customer satisfaction through organized order handling.
- Provides financial oversight with daily revenue summaries.
- Reduces manual errors in recording sales and customer data.
- Supports VIP customer programs and discount management.

General Overview of Classes, Their Connections, and Uses

The system is divided into **two major layers**:

- ❖ Data Structures Layer
- ❖ Restaurant Components Layer

3.1. Data Structures Layer

These structures are implemented from scratch and used throughout the project. Also Java Generics were used, which provide a way to define classes, interfaces, and methods that can operate on different data types without specifying the exact type upfront. By parameterizing types, Generics enable the creation of reusable and flexible components while maintaining strong type-checking.

Purpose of Using Generics

- **Code Reusability:**
  A single generic class or method can handle multiple data types, eliminating the need to write duplicate code for each type.
- **Type Safety:**
  Errors related to incompatible types are detected at compile time, reducing runtime exceptions and improving reliability.
- **Cleaner and More Readable Code:**
  Generics remove the need for explicit casting when retrieving stored elements, making the code more concise and easier to maintain.

In this project, generics are used across data structures such as Node<T>, LinkedList<T>, Stack<T>, and PriorityQueue<T> to ensure that these structures can store and manage any object type while remaining safe and efficient.

- ❖ **Node<T>**

A fundamental element holding data and a pointer to the next node.

Used internally by all linear data structures.

- ❖ **LinkedList<T>**

Stores collections such as customers, menu items, and historical revenue items.

Provides insertion, deletion, searching, and sequential access.

- **Stack<T>**

Used to track the order history so the last placed order can be canceled (LIFO).

- **PriorityQueue<T>**

Stores kitchen orders based on VIP priority and FIFO order number.
Implements sorted insertion using Comparator logic to simulate priority-aware scheduling.

3.2. Restaurant Components Layer

- **Customer & VIPCustomer**

Customer holds name, phone, and VIP status.
VIPCustomer extends Customer and adds a discount rate.
Customers are stored in a LinkedList and retrieved when orders are placed.

- **MenuItem & Menu**

MenuItem stores name, price, and category.
Menu maintains a list of MenuItems using LinkedList and allows add, remove, find, and display operations.

- **Order**

Represents a customer's order and includes:

- customer reference
- ordered items
- VIP priority value (derived from VIP status)
- order number (FIFO control)
- total price
  Used by Kitchen and Servicing.

❖ **Kitchen**

The core engine for order processing.
Uses:

- **PriorityQueue<Order>** for pending orders (VIP first)
- **LinkedList<Order>** for served orders history
  Handles serving logic, order removal, and revenue calculation.

❖ **Accounting & DailyRevenue**

Tracks the day's total revenue and stores it using LinkedList<DailyRevenue>.
Provides summary reports and persistent saving.

❖ **myUtils**

A helper class grouping file operations and summary presentation functions.
Used to simulate database storage via text files.

❖ **Servicing**

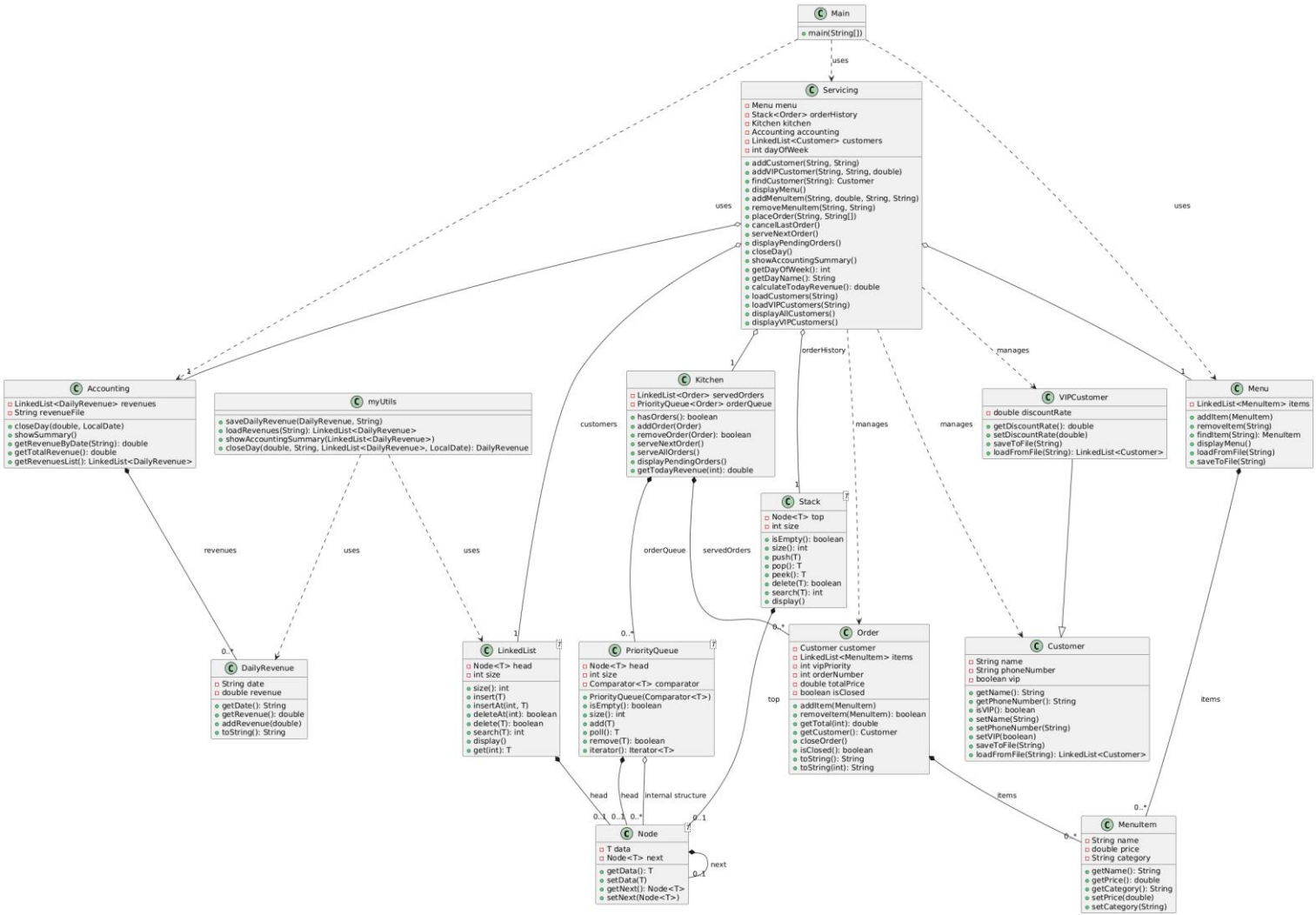
The main controller that connects all components.
Responsibilities include:

- customer and VIP registration
- menu operations
- placing/canceling orders
- invoking kitchen operations
- closing the day
  Works as the central management class the UI (Main) interacts with.

❖ **Main**

Creates the system, initializes components, and provides the user interface for testing or interaction.

Uml Diagram of the Program

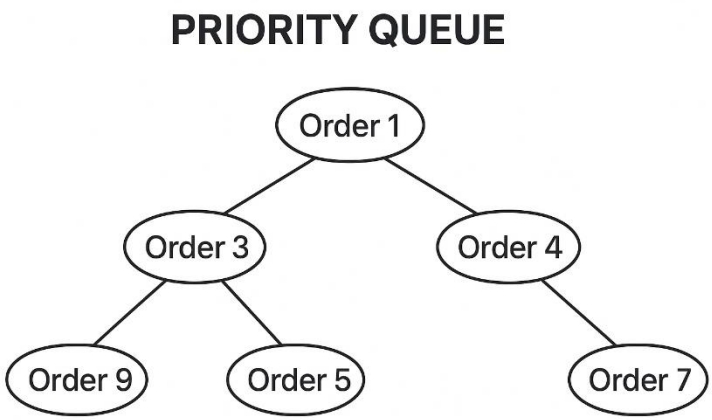


Visual Representation of Data Structures Used

The data structures implemented in this system include Linked Lists, a Priority Queue, a Stack, and basic Node-based structures. These data structures form the backbone of the application and were implemented manually to meet project requirements and avoid built-in Java collections.

5.1. Each major component, orders, menu items, customers, and revenue entries, uses these structures to enable dynamic data manipulation, efficient traversal, and predictable behavior.

- **Priority Queue (for managing active orders)**
  The Priority Queue stores **Order** objects wrapped inside Node structures. It ensures that:
  - VIP customers are always served before regular customers.
  - Among customers with equal VIP priority, orders follow **FIFO** (First-In-First-Out).
    This structure ensures fairness and efficiency during rush hours.

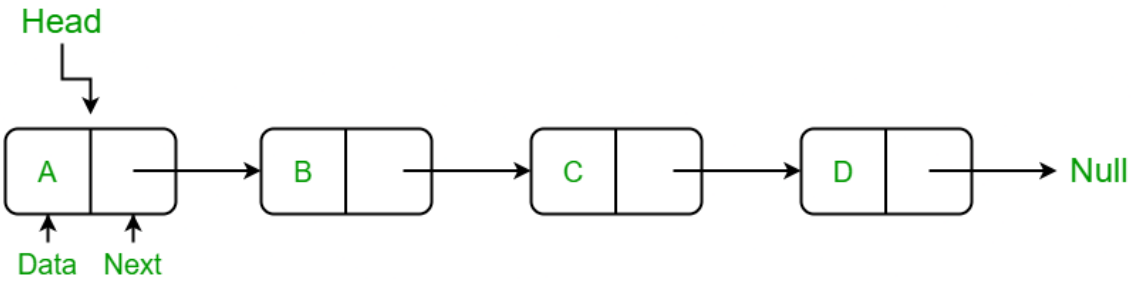


Implemented in kitchen class to ensure priority in order preparation.

- **Linked Lists (for menu, served orders, customer lists, and revenue logs)**
  Linked Lists allow:
  - Dynamic menu growth without predefined limits
  - Efficient insertion/removal of served orders
  - Storage of customer information

- Sequential access to historical revenue records

  Each list uses Node objects containing one element and a pointer to the next.



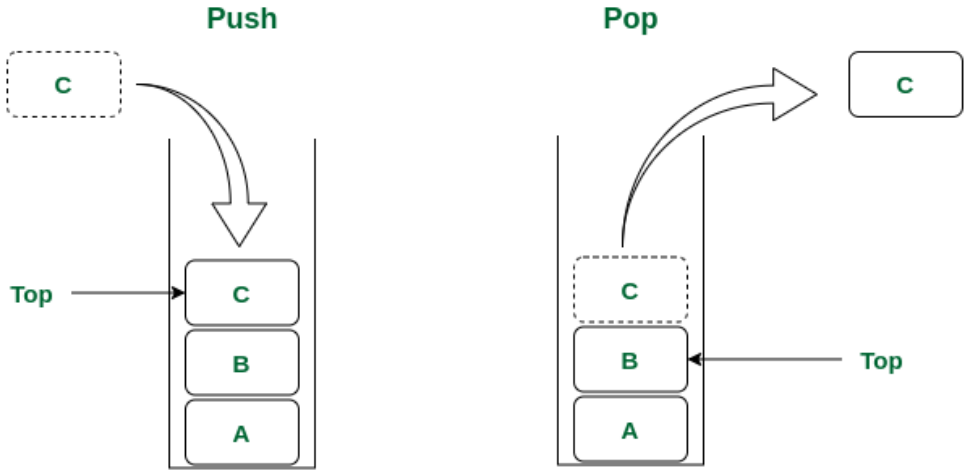
Served orders, Category, MenuItem

- **Stack (for order cancellation / undo feature)**

  The system uses a **Stack<Order>** to support the cancellation of the **most recent order placed**.

  - Every time an order is created, it is pushed onto the stack.
  - If the user chooses to cancel the last order, the system simply pops from the stack.

  This aligns with LIFO (Last-In-First-Out) behavior, making Stack the ideal structure for undo-style operations.



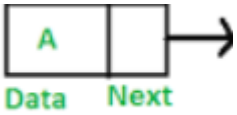
Order (for order cancellation feature)

- **Node class (foundation of all data structures)**

  The Node class is the essential building block connecting elements in:

  - LinkedList
  - Stack
  - Priority Queue

    It stores generic data and a reference to the next Node, enabling all structures to be implemented from scratch without Java library dependencies.

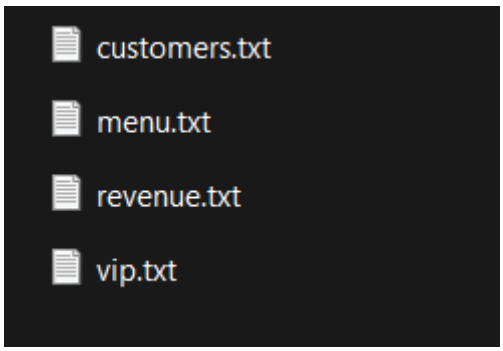


- **Files as a simulated persistent database**

  Since the project requires the use of manually implemented data structures, file storage is used to simulate a simple database:

  - Customer information
  - VIP data
  - Revenue logs

    This ensures persistence between runs while maintaining full consistency with the project's constraints.



Key Methods Highlights

Data Structures

LinkedList<T>

- insert(T data) - Adds an element at the end.
- delete(T value) - Removes the first occurrence of a value.
- search(T value) - Returns index of a value.
- get(int index) - Access element at a position.

Stack<T>

- push(T data) - Adds element on top.
- pop() - Removes and returns top element.
- peek() - Returns top element without removing.

PriorityQueue<T>

- add(T element) - Inserts element based on priority.
- poll() - Removes and returns the highest-priority element.
- remove(T element) - Deletes a specific element.

Components

Customer / VIPCustomer

- saveToFile(String path) / loadFromFile(String path) - Store/load customer data.
- VIPCustomer adds getDiscountRate() for VIP discount.

Menu / MenuItem

- addItem(MenuItem item) / removeItem(String name) - Manage menu items.
- findItem(String name) - Lookup an item.
- displayMenu() - Show all menu items.

Order

- addItem(MenuItem item) / removeItem(MenuItem item) - Modify order items.
- getTotal(int dayOfWeek) -Calculate total with discounts.

- closeOrder() - Marks the order as served.

Kitchen

- addOrder(Order o) - Add order to priority queue.
- serveNextOrder() - Serve the highest-priority order.
- serveAllOrders() - Serve all pending orders.
- displayPendingOrders() - Show orders waiting to be served.

Accounting

- closeDay(double revenue, LocalDate date) - Records daily revenue.
- showSummary() - Displays total revenue and best/worst days.

myUtils

- loadRevenues(String filePath) / saveDailyRevenue(DailyRevenue day, String filePath) - Handles file storage for revenue.

Servicing

- addCustomer(String name, String phone) / addVIPCustomer(String name, String phone, double discount) - Add customers.
- placeOrder(String customerPhone, String[] itemNames) - Place an order for a customer.
- cancelLastOrder() - Undo the last order.
- serveNextOrder() - Serve the next pending order.
- displayPendingOrders() - View pending orders.
- closeDay() - Serve all orders and update accounting.
- displayMenu() / displayAllCustomers() / displayVIPCustomers() -Utility display methods.

Main Class (Main.java**)**

- Provides the console interface for interacting with the system.
- Uses a switch-case menu to call methods from Servicing.

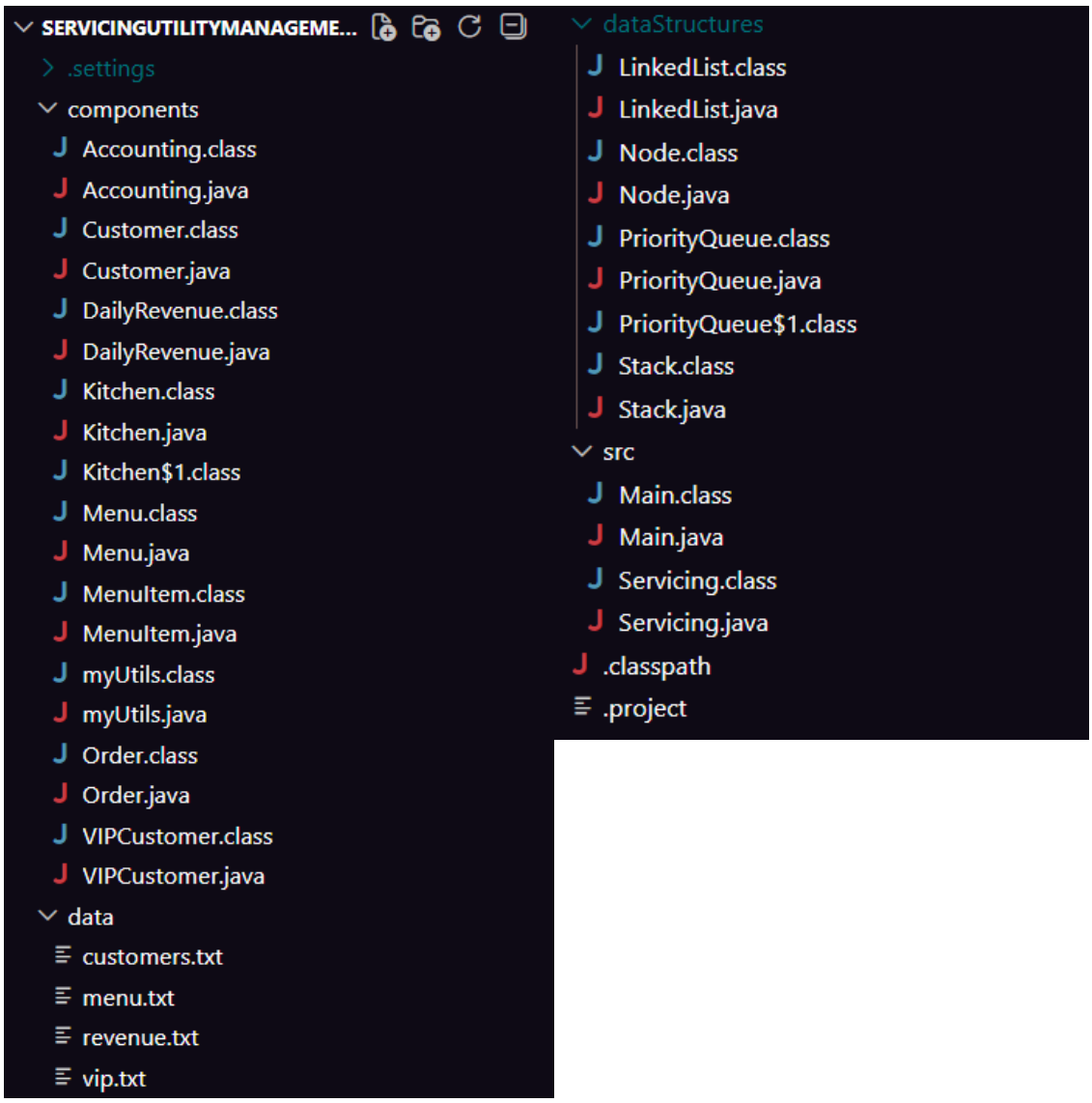
Key menu options and their actions:

CHOICE	ACTION / SERVICING METHOD CALLED
1	Add Customer → `servicing.addCustomer(name, phone)`
2	Add VIP Customer → `servicing.addVIPCustomer(name, phone, discount)`
3	Display Menu → `servicing.displayMenu()`
4	Add Menu Item → `servicing.addMenuItem(name, price, category, filePath)`
5	Remove Menu Item → `servicing.removeMenuItem(name, filePath)`
6	Place Order → `servicing.placeOrder(phone, itemNames)`
7	Serve Next Order → `servicing.serveNextOrder()`
8	Display Pending Orders → `servicing.displayPendingOrders()`
9	Display All Customers → `servicing.displayAllCustomers()`
10	Display VIP Customers → `servicing.displayVIPCustomers()`
11	Close Day → `servicing.closeDay()` (serves remaining orders, updates revenue)
12	Show Accounting Summary → `servicing.showAccountingSummary()`

- Input is read via `Scanner`.
- Runs in a **loop** until option 11 (`Close Day`) is chosen.
- Provides a **simple text-based management interface** connecting all system components.

Source Code

Folder Structures:



```java
package dataStructures;

public class Node <T> {
        private T data;
        private Node<T> next;

        public Node(T data) {
            this.data=data;
            this.next=null;
        }

        public T getData() {
            return data;
        }

        public void setData(T data) {
            this.data = data;
        }

        public Node<T> getNext() {
            return next;
        }

        public void setNext(Node<T> next) {
            this.next = next;
        }
}
```

```java
package dataStructures;

public class LinkedList<T> {

    private Node<T> head;
    private int size;

    public int getSize() {
        return size;
    }

    public LinkedList() {
        this.head = null;
        this.size = 0;
    }

    public int size() {
        return size;
    }
```

```java
    public void insert(T data) {
        Node<T> newNode = new Node<>(data);

        if (head == null) {
            head = newNode;
        } else {
            Node<T> current = head;
            while (current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(newNode);
        }
        size++;
    }

    public void insertAt(int index, T data) {
        if (index < 0 || index > size)
            throw new IndexOutOfBoundsException("Invalid index");

        Node<T> newNode = new Node<>(data);

        if (index == 0) {
            newNode.setNext(head);
            head = newNode;
        } else {
            Node<T> current = head;
            for (int i = 0; i < index - 1; i++) {
                current = current.getNext();
            }
            newNode.setNext(current.getNext());
            current.setNext(newNode);
        }
        size++;
    }

    public boolean deleteAt(int index) {
        if (index < 0 || index >= size)
            return false;

        if (index == 0) {
            head = head.getNext();
        } else {
            Node<T> current = head;
            for (int i = 0; i < index - 1; i++)
                current = current.getNext();

            current.setNext(current.getNext().getNext());
        }
```

```
        size--;
        return true;
    }

    public boolean delete(T value) {
        if (head == null)
            return false;

        if (head.getData().equals(value)) {
            head = head.getNext();
            size--;
            return true;
        }

        Node<T> current = head;
        while (current.getNext() != null &&
!current.getNext().getData().equals(value)) {
            current = current.getNext();
        }

        if (current.getNext() == null)
            return false;

        current.setNext(current.getNext().getNext());
        size--;
        return true;
    }

    public int search(T value) {
        Node<T> current = head;
        int index = 0;

        while (current != null) {
            if (current.getData().equals(value))
                return index;

            current = current.getNext();
            index++;
        }
        return -1;
    }

    public void display() {
        Node<T> current = head;
        int index = 0;
        while (current != null) {
            System.out.println(index + " -> " + current.getData());
            current = current.getNext();
```

```
            index++;
        }
    }

    public T get(int index) {
        if (index < 0 || index >= size)
            throw new IndexOutOfBoundsException("Invalid index");

        Node<T> current = head;
        for (int i = 0; i < index; i++)
            current = current.getNext();

        return current.getData();
    }
}
```

```
package dataStructures;

import java.util.Comparator;
import java.util.Iterator;

public class PriorityQueue<T> implements Iterable<T> {

    private Node<T> head;
    private int size;
    private Comparator<T> comparator;

    public PriorityQueue(Comparator<T> comparator) {
        this.comparator = comparator;
        this.head = null;
        this.size = 0;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public int size() {
        return size;
    }

    public void add(T element) {
        Node<T> newNode = new Node<>(element);

        if (head == null || comparator.compare(element, head.getData())
< 0) {
            newNode.setNext(head);
```

```
            head = newNode;
        } else {
            Node<T> current = head;
            while (current.getNext() != null &&
                   comparator.compare(element,
current.getNext().getData()) >= 0) {
                current = current.getNext();
            }
            newNode.setNext(current.getNext());
            current.setNext(newNode);
        }

        size++;
    }

    public T poll() {
        if (isEmpty())
            return null;

        T removed = head.getData();
        head = head.getNext();
        size--;
        return removed;
    }

    public boolean remove(T element) {
        if (isEmpty())
            return false;

        if (head.getData().equals(element)) {
            head = head.getNext();
            size--;
            return true;
        }

        Node<T> current = head;
        while (current.getNext() != null) {
            if (current.getNext().getData().equals(element)) {
                current.setNext(current.getNext().getNext());
                size--;
                return true;
            }
            current = current.getNext();
        }
        return false;
    }

    public Iterator<T> iterator() {
```

```
        return new Iterator<T>() {
            Node<T> curr = head;

            public boolean hasNext() {
                return curr != null;
            }

            public T next() {
                T data = curr.getData();
                curr = curr.getNext();
                return data;
            }
        };
    }
}
```

```
package dataStructures;

public class Stack<T> {

    private Node<T> top;
    private int size;

    public Stack() {
        top = null;
        size = 0;
    }

    public boolean isEmpty() {
        return top == null;
    }

    public int size() {
        return size;
    }

    public void push(T data) {
        Node<T> newNode = new Node<>(data);
        newNode.setNext(top);
        top = newNode;
        size++;
    }

    public T pop() {
        if (top == null)
            return null;
```

```
        T popped = top.getData();
        top = top.getNext();
        size--;
        return popped;
    }

    public T peek() {
        return (top == null) ? null : top.getData();
    }

    public boolean delete(T value) {
        if (top == null) return false;

        if (top.getData().equals(value)) {
            top = top.getNext();
            size--;
            return true;
        }

        Node<T> current = top;
        while (current.getNext() != null &&
!current.getNext().getData().equals(value)) {
            current = current.getNext();
        }

        if (current.getNext() == null)
            return false;

        current.setNext(current.getNext().getNext());
        size--;
        return true;
    }

    public int search(T value) {
        Node<T> current = top;
        int index = 0;
        while (current != null) {
            if (current.getData().equals(value))
                return index;

            current = current.getNext();
            index++;
        }
        return -1;
    }

    public void display() {
```

```
        Node<T> current = top;
        int index = 0;
        while (current != null) {
            System.out.println("[" + index + "] " + current.getData());
            current = current.getNext();
            index++;
        }
    }
}
```

```
package components;
import java.time.LocalDate;


import dataStructures.LinkedList;

public class Accounting {

    private LinkedList<DailyRevenue> revenues;
    private String revenueFile;

    public Accounting(String revenueFile) {
        this.revenueFile = revenueFile;
        this.revenues = myUtils.loadRevenues(revenueFile);
    }

    public void closeDay(double todayRevenue, LocalDate date) {
        DailyRevenue today = myUtils.closeDay(todayRevenue,
revenueFile, revenues, date);
        System.out.println("Day closed: " + today.getDate() + ",
Revenue: $" + today.getRevenue());
    }

    public void showSummary() {
        myUtils.showAccountingSummary(revenues);
    }

    public double getRevenueByDate(String date) {
        for (int i = 0; i < revenues.size(); i++) {
            DailyRevenue day = revenues.get(i);
            if (day.getDate().equals(date)) {
                return day.getRevenue();
            }
        }
        return 0;
    }

    public double getTotalRevenue() {
```

```
        double total = 0;
        for (int i = 0; i < revenues.size(); i++) {
            total += revenues.get(i).getRevenue();
        }
        return total;
    }

    public LinkedList<DailyRevenue> getRevenuesList() {
        return revenues;
    }
}
```

```
package components;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Serializable;
import java.util.Scanner;

import dataStructures.LinkedList;

public class Customer implements Serializable {
    private String name;
    private String phoneNumber;
    private boolean vip;

    public Customer(String name, String phoneNumber) {
        this.name = name;
        this.phoneNumber = phoneNumber;
        this.vip = false;
    }

    public String getName() {
        return name;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }

    public boolean isVIP() {
        return vip;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    public void setVIP(boolean vip) {
        this.vip = vip;
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof Customer)) return false;
        Customer other = (Customer) obj;
        return this.name.equalsIgnoreCase(other.name) &&
               this.phoneNumber.equals(other.phoneNumber);
    }

    public int hashCode() {
        return (name.toLowerCase() + phoneNumber).hashCode();
    }

    public String toString() {
        return (vip ? "[VIP] " : "") + name + " (" + phoneNumber + ")";
    }

    public void saveToFile(String filePath) {
        try (FileWriter fw = new FileWriter(filePath, true)) {
            fw.write(name + "," + phoneNumber + "\n");
        } catch (IOException e) {
            System.out.println("Error saving customer to file: " +
e.getMessage());
        }
    }

    public static LinkedList<Customer> loadFromFile(String filePath) {
        LinkedList<Customer> list = new LinkedList<>();
        File file = new File(filePath);
        if (!file.exists()) return list;

        try (Scanner s = new Scanner(file)) {
            while (s.hasNextLine()) {
                String[] parts = s.nextLine().split(",");
                if (parts.length >= 2) {
                    String name = parts[0].trim();
                    String phone = parts[1].trim();
                    list.insert(new Customer(name, phone));
```

```
                }
            }
        } catch (IOException e) {
            System.out.println("Error loading customers from file: " +
e.getMessage());
        }

        return list;
    }

}
```

```
package components;

public class DailyRevenue {
    private String date;
    private double revenue;

    public DailyRevenue(String date, double revenue) {
        this.date = date;
        this.revenue = revenue;
    }

    public String getDate() {
        return date;
    }

    public double getRevenue() {
        return revenue;
    }

    public void addRevenue(double amount) {
        this.revenue += amount;
    }

    public String toString() {
        return date + "," + revenue;
    }
}
```

```
package components;
import dataStructures.LinkedList;
import dataStructures.PriorityQueue;
```

```
import java.util.Comparator;
public class Kitchen {
    private LinkedList<Order> servedOrders = new LinkedList<>();
    private PriorityQueue<Order> orderQueue;

    public Kitchen() {
        // higher vipPriority first, then fifo (orderNumber)
        Comparator<Order> vipComparator = new Comparator<Order>() {
            public int compare(Order o1, Order o2) {
                if (o1.getVipPriority() != o2.getVipPriority()) {
                    return Integer.compare(o2.getVipPriority(),
o1.getVipPriority()); // VIP first
                } else {
                    return Integer.compare(o1.getOrderNumber(),
o2.getOrderNumber()); // FIFO
                }
            }
        };

        orderQueue = new PriorityQueue<>(vipComparator);
    }

    public boolean hasOrders() {
        return !orderQueue.isEmpty();
    }

    public void addOrder(Order o) { //insert
        orderQueue.add(o);
        System.out.println("Order added: " + o.getCustomer());
    }

    public boolean removeOrder(Order order) {
        return orderQueue.remove(order);
    }


    public void serveNextOrder() {
        if (orderQueue.isEmpty()) {
            System.out.println("No pending orders to serve.");
            return;
        }

        Order order = orderQueue.poll();
        order.closeOrder();
        servedOrders.insert(order);

        System.out.println("Serving order for " + order.getCustomer());
        System.out.println(order.toString());
```

```
        System.out.println("Order served.\n");
    }

    public void serveAllOrders() {
        while (!orderQueue.isEmpty()) {
            serveNextOrder();
        }
    }

    public void displayPendingOrders() {
        if (orderQueue.isEmpty()) {
            System.out.println("No pending orders.");
            return;
        }
        System.out.println("Pending Orders- Customers are hungry- tell
the chefs to hurry up");
        for (Order o : orderQueue) {
            System.out.println(o.toString());
        }
    }

    public double getTodayRevenue(int dayOfWeek) {
        double total = 0;
        for (int i = 0; i < servedOrders.size(); i++) {
            Order o = servedOrders.get(i);
            total += o.getTotal(dayOfWeek);
        }
        return total;
    }

}
```

```
package components;
import dataStructures.LinkedList;
import java.io.*;
import java.util.Scanner;

public class Menu {
    private LinkedList<MenuItem> items = new LinkedList<>();

    public void addItem(MenuItem item) {
        items.insert(item);
    }

    public void removeItem(String name) {
        MenuItem rtool = new MenuItem(name, 0, "");
```

```
        items.delete(rtool);
    }

    public MenuItem findItem(String name) {
        MenuItem rtool = new MenuItem(name, 0, "");
        int index = items.search(rtool);
        if (index != -1) {
            return items.get(index);
        }
        return null;
    }


    public void displayMenu() {
        System.out.println("Our delicious MENU");
        items.display();
    }

    //this section deals with saving the menuItems in a file and
retrieving them, both operations dealing with items as
object(serialization)
    public void loadFromFile(String path) {
        File file = new File(path);
        if (!file.exists()) return;

        try (Scanner s = new Scanner(file)) {
            while (s.hasNextLine()) {
                String line = s.nextLine();
                String[] parts = line.split(",");
                if (parts.length == 3) {
                    String name = parts[0].trim();
                    double price = Double.parseDouble(parts[1].trim());
                    String category = parts[2].trim();
                    MenuItem q =new MenuItem(name, price, category);
                    addItem(q);
                }
            }
        } catch (Exception e) {
            System.out.println("Error reading menu file: " +
e.getMessage());
        }
    }

    public void saveToFile(String path) {
        try (PrintWriter writer = new PrintWriter(new
FileWriter(path))) {
            for (int i = 0; i < items.size(); i++) {
                MenuItem item = items.get(i);
```

```
                writer.println(item.getName() + "," + item.getPrice() +
"," + item.getCategory());
            }
        } catch (IOException e) {
            System.out.println("Error saving menu: " + e.getMessage());
        }
    }
}
```

```
package components;
import java.io.Serializable;

public class MenuItem implements Serializable {
    private String name;
    private double price;
    private String category;

    public MenuItem(String name, double price, String category) {
        this.name = name;
        this.price = price;
        this.category = category;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }

    public String getCategory() {
        return category;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public void setCategory(String category) {
        this.category = category;
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof MenuItem)) return false;
        MenuItem other = (MenuItem) obj;
```

```
        return this.name.equalsIgnoreCase(other.name);
    }

    public String toString() {
        return name + " - $" + String.format("%.2f", price) + " ("+
category +")"; //formats the price to  show the first 2 decimal digits
    }
}

package components;

import java.io.*;
import java.time.LocalDate;
import java.util.Scanner;

import dataStructures.LinkedList;

public class myUtils {

    //save daily revenue to the text file
    public static void saveDailyRevenue(DailyRevenue day, String
filePath) {
        try (FileWriter fw = new FileWriter(filePath, true)) {
            fw.write(day.toString() + "\n");
        } catch (IOException e) {
            System.out.println("Error writing revenue file: " +
e.getMessage());
        }
    }

    //load all daily revenues from file into a linked list
    public static LinkedList<DailyRevenue> loadRevenues(String
filePath) {
        LinkedList<DailyRevenue> revenues = new LinkedList<>();
        File file = new File(filePath);
        if (!file.exists()) return revenues;
        try (Scanner s = new Scanner(file)) {
            while (s.hasNextLine()) {
                String[] parts = s.nextLine().split(",");
                if (parts.length == 2) {
                    String date = parts[0].trim();
                    double revenue =
Double.parseDouble(parts[1].trim());
                    revenues.insert(new DailyRevenue(date, revenue));
                }
            }
        } catch (Exception e) {
```

```
            System.out.println("Error reading revenue file: " +
e.getMessage());
        }

        return revenues;
    }

    // get total revenue for all days, best, and worst selling day
    public static void showAccountingSummary(LinkedList<DailyRevenue>
revenues) {
        if (revenues.size() == 0) {
            System.out.println("No revenue data found.");
            return;
        }

        DailyRevenue bestDay = revenues.get(0); //use the algorithm to
find max and min from the linked list
        DailyRevenue worstDay = revenues.get(0);
        double total = 0;

        for (int i = 0; i < revenues.size(); i++) {
            DailyRevenue day = revenues.get(i);
            total += day.getRevenue();
            if (day.getRevenue() > bestDay.getRevenue()) bestDay = day;
            if (day.getRevenue() < worstDay.getRevenue()) worstDay =
day;
        }

        System.out.println("Accounting Summary");
        System.out.println("Total Revenue: $" + String.format("%.2f",
total));
        System.out.println("Best Selling Day: " + bestDay.getDate() + "
($" + bestDay.getRevenue() + ")");
        System.out.println("Lowest Selling Day: " + worstDay.getDate()
+ " ($" + worstDay.getRevenue() + ")");
    }

    // close up current day,sum revenue, add to file and add a new day
    public static DailyRevenue closeDay(double todayRevenue, String
filePath, LinkedList<DailyRevenue> revenues, LocalDate date) {
        DailyRevenue today = new DailyRevenue(date.toString(),
todayRevenue);
        revenues.insert(today);
        saveDailyRevenue(today, filePath);
        return today;
    }

}
```

```java
package components;
import dataStructures.LinkedList;

public class Order {
    private Customer customer;
    private LinkedList<MenuItem> items;
    private int vipPriority;
    private int orderNumber;
    private static int counter = 0;
    private double totalPrice;
    private boolean isClosed;

    public Order(Customer customer) {
        this.customer = customer;
        this.items = new LinkedList<>();
        this.totalPrice = 0;
        this.isClosed = false;
        this.vipPriority = customer.isVIP() ? 1 : 0;
        this.orderNumber = ++counter;
    }

    public void addItem(MenuItem item) {
        items.insert(item);
        totalPrice += item.getPrice();
    }

    public boolean removeItem(MenuItem item) {
        boolean removed = items.delete(item);
        if (removed) {
            totalPrice -= item.getPrice();
        }
        return removed;
    }

    public double getTotal(int dayOfWeek) {
        double vipDisc = 0;
        double fridayDisc = 0;

        if (customer instanceof VIPCustomer) {
            VIPCustomer vip = (VIPCustomer) customer;
            vipDisc = vip.getDiscountRate();
        }

        if (dayOfWeek == 5) {
            fridayDisc = 20;
        }
```

```
        double maxDiscount = Math.max(vipDisc, fridayDisc);

        return totalPrice * (1 - maxDiscount / 100.0);
    }

    public Customer getCustomer() {
        return customer;
    }

    public void closeOrder() {
        isClosed = true;
    }

    public boolean isClosed() {
        return isClosed;
    }

    public int getVipPriority() {
        return vipPriority;
    }

    public void setVipPriority(int vipPriority) {
        this.vipPriority = vipPriority;
    }

    public int getOrderNumber() {
        return orderNumber;
    }


    public void setClosed(boolean isClosed) {
        this.isClosed = isClosed;
    }

    public String toString(int dayOfWeek) {
        String s = "Order by " + customer + ":\n";
        for (int i = 0; i < items.size(); i++) {
            s += " - " + items.get(i) + "\n";
        }
        s += "Total: $" + String.format("%.2f", getTotal(dayOfWeek));
        return s;
    }

    public String toString() {
        String s = "Order by " + customer + ":\n";
        for (int i = 0; i < items.size(); i++) {
            s += " - " + items.get(i) + "\n";
        }
        return s;
```

```
    }
}

package components;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;

import dataStructures.LinkedList;

public class VIPCustomer extends Customer {
    private double discountRate;

    public VIPCustomer(String name,String phoneNumber, double
discountRate) {
        super(name,phoneNumber);
        setVIP(true);
        this.discountRate = discountRate;
    }

    public double getDiscountRate() {
        return discountRate;
    }

    public void setDiscountRate(double discountRate) {
        this.discountRate = discountRate;
    }

    public String toString() {
        return "[VIP " + getName() + " (" + getPhoneNumber() + ")"+ " -
" + discountRate + "% off]";
    }



    public void saveToFile(String filePath) {
        try (FileWriter fw = new FileWriter(filePath, true)) {
            fw.write(getName() + "," + getPhoneNumber() + "," +
discountRate + "\n");
        } catch (IOException e) {
            System.out.println("Error writing VIP file: " +
e.getMessage());
        }
    }

    public static LinkedList<Customer> loadFromFile(String filePath) {
```

```
        LinkedList<Customer> list = new LinkedList<>();
        File file = new File(filePath);
        if (!file.exists()) return list;

        try (Scanner s = new Scanner(file)) {
            while (s.hasNextLine()) {
                String line = s.nextLine().trim();
                if (line.isEmpty()) continue;
                String[] parts = line.split(",");
                if (parts.length == 3) {
                    String name = parts[0].trim();
                    String phone = parts[1].trim();
                    double discount =
Double.parseDouble(parts[2].trim());
                    VIPCustomer vip = new VIPCustomer(name, phone,
discount);
                    list.insert(vip);
                }
            }
        } catch (Exception e) {
            System.out.println("Error reading VIP file: " +
e.getMessage());
        }

        return list;
    }



}
```

```
package src;
import java.time.LocalDate;

import components.Accounting;
import components.Customer;
import components.DailyRevenue;
import components.Kitchen;
import components.Menu;
import components.MenuItem;
import components.Order;
import components.VIPCustomer;
import dataStructures.LinkedList;
import dataStructures.Stack;

public class Servicing {
```

```
    private Menu menu;
    Stack<Order> orderHistory = new Stack<>(); //in case an order is
cancelled
    private Kitchen kitchen;
    private Accounting accounting;
    private LinkedList<Customer> customers;
    private int dayOfWeek;

    public Servicing(Menu menu, Accounting accounting) {
        this.menu = menu;
        this.kitchen = new Kitchen();
        this.accounting = accounting;
        this.customers = new LinkedList<>();

        LinkedList<DailyRevenue> revenues =
accounting.getRevenuesList(); //set dayofweek which is used to
determine friday discount
        if (revenues.size() > 0) {
            LocalDate lastDate =
LocalDate.parse(revenues.get(revenues.size() - 1).getDate());
            this.dayOfWeek = lastDate.getDayOfWeek().getValue(); //
monday=1, tuesday=2 .. etc
        } else {
            this.dayOfWeek = 1; // default is monday
        }
    }

    public void addCustomer(String name, String phone) {
        Customer c = new Customer(name, phone);
        customers.insert(c);
        System.out.println("Customer added: " + c);
        c.saveToFile("data/customers.txt");
    }

    public void addVIPCustomer(String name, String phone, double
discount) {
        VIPCustomer vip = new VIPCustomer(name, phone, discount);
        customers.insert(vip);
        System.out.println("VIP customer added: " + vip);
        vip.saveToFile("data/vip.txt");
    }

    public Customer findCustomer(String phone) {
        for (int i = 0; i < customers.size(); i++) {
            Customer c = customers.get(i);
            if (c.getPhoneNumber().equals(phone)) return c;
        }
```

```
        return null;
    }

    public void displayMenu() {
        menu.displayMenu();
    }

    public void addMenuItem(String name, double price, String category,
String filePath) {
        MenuItem item = new MenuItem(name, price, category);
        menu.addItem(item);
        menu.saveToFile(filePath);
        System.out.println("Menu item added: " + item);
    }

    public void removeMenuItem(String name, String filePath) {
        menu.removeItem(name);
        menu.saveToFile(filePath);
        System.out.println("Menu item removed: " + name);
    }

    public void placeOrder(String customerPhone, String[] itemNames) {
        Customer c = findCustomer(customerPhone);
        if (c == null) {
            System.out.println("Customer not found!");
            return;
        }
        Order order = new Order(c);
        for (String name : itemNames) {
            MenuItem item = menu.findItem(name);
            if (item != null) order.addItem(item);
        }
        kitchen.addOrder(order);
        orderHistory.push(order);
        System.out.println("Order placed for " + c.getName());
    }

    public void cancelLastOrder() {
        if (orderHistory.isEmpty()) {
            System.out.println("No order to undo.");
            return;
        }
        Order lastOrder = orderHistory.pop();
        if(kitchen.removeOrder(lastOrder))System.out.println("Last
order cancelled for " + lastOrder.getCustomer());
        else System.out.println("An error my have occured");

    }
```

```
    public void serveNextOrder() {
        kitchen.serveNextOrder();
    }

    public void displayPendingOrders() {
        kitchen.displayPendingOrders();
    }

    public void closeDay() {
        LocalDate lastDate;
        LinkedList<DailyRevenue> revenues =
accounting.getRevenuesList();
        System.out.println("Thank you everyone for today's effort on "
+ getDayName() + "!");
        if (revenues.size() > 0) {
            lastDate = LocalDate.parse(revenues.get(revenues.size() -
1).getDate()).plusDays(1);
        } else {
            lastDate = LocalDate.now();
        }
        kitchen.serveAllOrders();
        accounting.closeDay(calculateTodayRevenue(), lastDate);
        dayOfWeek = lastDate.getDayOfWeek().getValue();
    }

    public void showAccountingSummary() {
        accounting.showSummary();
    }

    public int getDayOfWeek() {
        return dayOfWeek;
    }

    public String getDayName() {
        switch(dayOfWeek) {
            case 1: return "Monday";
            case 2: return "Tuesday";
            case 3: return "Wednesday";
            case 4: return "Thursday";
            case 5: return "Friday";
            case 6: return "Saturday";
            case 7: return "Sunday";
            default: return "Unknown";
        }
    }

    public double calculateTodayRevenue() {
```

```
        return kitchen.getTodayRevenue(dayOfWeek);
    }

    public void loadCustomers(String filePath) {
        LinkedList<Customer> list = Customer.loadFromFile(filePath);
        for (int i = 0; i < list.size(); i++) {
            customers.insert(list.get(i));
        }
    }

    public void loadVIPCustomers(String filePath) {
        LinkedList<Customer> list = VIPCustomer.loadFromFile(filePath);
        for (int i = 0; i < list.size(); i++) {
            customers.insert(list.get(i));
        }
    }

    public void displayAllCustomers() {
        System.out.println("All Customers:");
        customers.display();
    }

    public void displayVIPCustomers() {
        System.out.println("VIP Customers:");
        for (int i = 0; i < customers.getSize(); i++) {
            Customer c = customers.get(i);
            if (c instanceof VIPCustomer) {
                System.out.println(c);
            }
        }
    }
}
```

```
package src;

import java.util.Scanner;

import components.Accounting;
import components.Menu;

public class Main {

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

        Menu menu = new Menu();
```

```
        menu.loadFromFile("data/menu.txt");

        Accounting accounting = new Accounting("data/revenue.txt");

        Servicing servicing = new Servicing(menu, accounting);
        servicing.loadCustomers("data/customers.txt");
        servicing.loadVIPCustomers("data/vip.txt");


        System.out.println("~");
        System.out.println("Welcome to the Servicing Utility Management
System!!");
        System.out.println("Today is: " + servicing.getDayName());
        System.out.println("~");

        boolean running = true;
        while (running) {
            System.out.println("\nMain Menu");
            System.out.println("1- Add Customer");
            System.out.println("2- Add VIP Customer");
            System.out.println("3- Display Menu");
            System.out.println("4- Add Menu Item");
            System.out.println("5- Remove Menu Item");
            System.out.println("6- Put Order");
            System.out.println("7- Serve Next Order");
            System.out.println("8- Display Pending Orders");
            System.out.println("9- Display all customers");
            System.out.println("10- Display VIP customers");
            System.out.println("11- Close Day");
            System.out.println("12- Show Accounting Summary");
            System.out.print("Enter your choice: ");

            int choice = s.nextInt();
            s.nextLine();

            switch (choice) {
            case 1:
                System.out.print("Customer name: ");
                String name = s.nextLine();
                System.out.print("Customer phone: ");
                String phone = s.nextLine();
                servicing.addCustomer(name, phone);
                break;
            case 2:
                System.out.print("VIP name: ");
                name = s.nextLine();
                System.out.print("VIP phone: ");
                phone = s.nextLine();
```

```
                System.out.print("VIP discount (%): ");
                double discount = s.nextDouble();
                s.nextLine();
                servicing.addVIPCustomer(name, phone, discount);
                break;
            case 3:
                servicing.displayMenu();
                break;
            case 4:
                System.out.print("Item name: ");
                name = s.nextLine();
                System.out.print("Price: ");
                double price = s.nextDouble();
                s.nextLine();
                System.out.print("Category: ");
                String category = s.nextLine();
                servicing.addMenuItem(name, price,
category,"data/menu.txt");
                break;
            case 5:
                System.out.print("Item name to remove: ");
                name = s.nextLine();
                servicing.removeMenuItem(name,"data/menu.txt");
                break;
            case 6:
                System.out.print("Customer phone: ");
                phone = s.nextLine();
                System.out.print("Enter item names (comma separated):
");
                String[] items = s.nextLine().split(",");
                for (int i = 0; i < items.length; i++) items[i] =
items[i].trim();
                servicing.placeOrder(phone, items);
                break;
            case 7:
                servicing.serveNextOrder();
                break;
            case 8:
                servicing.displayPendingOrders();
                break;
            case 9:
                servicing.displayAllCustomers();
                break;
            case 10:
                servicing.displayVIPCustomers();
                break;
            case 11:
                servicing.closeDay();
```

```
                running = false;
                break;
            case 12:
                servicing.showAccountingSummary();
                break;
            default:
                System.out.println("Invalid choice!");
                break;
            }
        }
        s.close();
    }
}
```

Output (Main output and Test runs)

```
Main Menu
1- Add Customer
2- Add VIP Customer
2- Add VIP Customer
3- Display Menu
4- Add Menu Item
3- Display Menu
4- Add Menu Item
5- Remove Menu Item
4- Add Menu Item
5- Remove Menu Item
6- Put Order
5- Remove Menu Item
6- Put Order
6- Put Order
7- Serve Next Order
8- Display Pending Orders
9- Display all customers
10- Display VIP customers
11- Close Day
12- Show Accounting Summary
Enter your choice: 1
Customer name: amjad
Customer phone: 123456
Customer added: amjad (123456)
```

```
Enter your choice: 3
Our delicious MENU
0 -> fattoush - $15.00 (salads)
1 -> crispy chicken - $20.00 (main dish)
2 -> tabboulie - $7.00 (salads)
```

```
Enter your choice: 6
Customer phone: 123456
Enter item names (comma separated): fattoush, cripsy chicken
Order added: amjad (123456)
Order placed for amjad
```

```
Enter your choice: 8
Pending Orders- Customers are hungry- tell the chefs to hurry up
Order by amjad (123456):
 - fattoush - $15.00 (salads)
 - crispy chicken - $20.00 (main dish)
```

```
Enter your choice: 7
Serving order for amjad (123456)
Order by amjad (123456):
 - fattoush - $15.00 (salads)
 - crispy chicken - $20.00 (main dish)

Order served.
```

```
Enter your choice: 9
All Customers:
0 -> Alice (12345)
1 -> Bob (67890)
2 -> Diana (24680)
3 -> Ethan (13579)
4 -> [VIP Charlie (55555) - 15.0% off]
5 -> [VIP Fiona (98765) - 20.0% off]
6 -> [VIP George (11223) - 10.0% off]
7 -> amjad (123456)
```

```
Enter your choice: 10
VIP Customers:
[VIP Charlie (55555) - 15.0% off]
[VIP Fiona (98765) - 20.0% off]
[VIP George (11223) - 10.0% off]
```

```
Enter your choice: 12
Accounting Summary
Total Revenue: $41.00
Best Selling Day: 2025-11-09 ($25.0)
Lowest Selling Day: 2025-11-11 ($0.0)
```

```
Enter your choice: 11
Thank you everyone for today's effort on Saturday!
Day closed: 2025-11-16, Revenue: $50.0
```

Updated file sample:

```
Alice,12345
Bob,67890
Diana,24680
Ethan,13579
amjad,123456
```

```
2025-11-09,25.0
2025-11-10,16.0
2025-11-11,0.0
2025-11-12,0.0
2025-11-13,0.0
2025-11-14,0.0
2025-11-15,0.0
2025-11-16,50.0
```

```
fattoush,15.0,salads
crispy chicken,20.0,main dish
tabboulie,7.0,salads
```

Conclusion

The purpose of this restaurant management system is to provide a useful, comprehensive software solution for automating a food service company's everyday operations. It lets owners and managers handle important activities like keeping a digital menu, tracking regular and VIP clients, processing and prioritizing orders, managing kitchen operations, and providing daily sales data, all inside a single, structured platform.

The system is a helpful tool for restaurants, cafés, and similar enterprises wishing to simplify their management procedures since it increases operational efficiency, enables improved customer service, and offers clear financial supervision by decreasing human labor and inefficiencies.

References

https://www.w3schools.com/java/java_generics.asp

https://www.w3schools.com/java/java_iterator.asp