



**INSTITUTE OF COMPUTER SCIENCE & DIGITAL INNOVATION**

**BIC2214**

**Data Structures & Algorithms**

**Assignment**

**Student declaration**

**I declare that:**

- **I understand what is meant by plagiarism**
- **The implications of plagiarism have been explained to us by our lecturer**

**This project is all our work, and I have acknowledged any use of the published or unpublished works of other people.**

**Submission Date: 12 August 2025**

**Names of Group Members**

<b>No</b>	<b>Student Id</b>	<b>Student Name</b>
1	1002162634	Ronald Lee Kai Ren
2	1002475656	Chan Xin Qi
3	1002475718	Ng Weng Hong
4	1002163941	Soh Jing Feng

## Assignment (30%)

Learning tasks	5 marks	4 marks	3 marks	2 marks	1 mark	% / Mark
<b>Format, originality, content integrity (5%)</b>	Documents very well and sufficient learning tied to the format of the report.	Documents well and sufficient learning tied to the format of the report.	Documents a few but not sufficient learning tied to the format of the report.	Documents a limited and not sufficient learning tied to the format of the report.	Documents hardly and not sufficient learning tied to the format of the report.	
<b>Presentation (5%)</b>	Presentation document is prepared well as per the format and align with project. Presentation is outstanding.	Presentation document is prepared moderate as per the format and align with project. Presentation is outstanding.	Presentation document is prepared as per the format and align with project. Presentation is good.	Presentation document is prepared very limited as per the format and align with project. Presentation is not up to the mark.	Presentation document is prepared hardly as per the format and align with project. Presentation is not up to the mark.	
<b>Demonstration (5%)</b>	Demonstrates complete understanding of the project including outcome.	Demonstrates sufficient understanding of the project including outcome.	Demonstrates a good understanding of the project including outcome.	Demonstrates a limited understanding of the project including outcome.	Hardly demonstrate the understanding of the project including outcome.	
Learning tasks	13-15 marks	10-12 marks	7-9 marks	4-6 marks	1-3 mark	% / Mark
<b>Report Content and job quality. (15%)</b>	The content of the report is well aligned with the learning outcome and perform the overall task wisely.	The content of the report is aligned with the learning outcome and perform the overall task wisely.	The content of the report is aligned with the learning outcome and perform the overall task moderately.	The content of the report is less aligned with the learning outcome and perform the overall task moderately.	The content of the report is hardly aligned with the learning outcome and perform the overall task poorly.	

## Table of Contents

<b>Abstract.....</b>	<b>5</b>
<b>1. Introduction.....</b>	<b>6</b>
1.1 Data Structure and Operations.....	6
1.2 Motivation.....	6
1.3 Problem statement.....	7
1.4 Objectives.....	7
<b>2. Literature Review: Data Structures in E-commerce.....</b>	<b>8</b>
2.1 Importance of Efficient Data Structures in E-Commerce.....	8
2.2 Linked Lists vs. Arrays/ArrayLists for Shopping Cart Management.....	8
2.3 FIFO Queues for Order Processing.....	10
<b>3. Discussion &amp; Analysis.....</b>	<b>11</b>
3.1 Variables and Methods.....	11
3.2 Time Complexity (Theoretical Analysis).....	16
3.3 Run-Time Complexity (Practical Measurement).....	17
3.4 Justification for chosen Data Structure.....	18
<b>4. Run Java Code.....</b>	<b>19</b>
<b>5. Screenshots of the Application &amp; Output.....</b>	<b>55</b>
Application Screenshots.....	55
Data Output.....	60
<b>6. Conclusion.....</b>	<b>62</b>
<b>7. References.....</b>	<b>63</b>
<b>8. Turnitin Report.....</b>	<b>64</b>

## **Abstract**

This report details the development of a basic e-commerce cart system implemented in Java, leveraging fundamental data structures to address common challenges in dynamic data management for online retail. The primary problem addressed is the efficient handling of variable-sized shopping carts and ensuring fair, sequential order processing. Our solution employs a Linked List for managing cart items, facilitating efficient  $O(1)$  additions (at the head or updates) and  $O(n)$  removals due to its dynamic resizing capabilities and direct node manipulation without array shifting. For order processing, a Queue data structure is utilized, strictly adhering to the First-In-First-Out (FIFO) principle, which ensures  $O(1)$  enqueue and dequeue operations, guaranteeing equitable and predictable order fulfillment, especially under peak load conditions. Practical run-time measurements for cart operations confirmed the theoretical time complexities: `addItem` demonstrated near  $O(1)$  performance, while `removeItem` exhibited  $O(n)$  behavior, aligning with the expected traversal requirement. These findings validate the chosen data structures' efficiency for their respective tasks. The project successfully demonstrates the practical application of Linked Lists and Queues in a real-world e-commerce context, highlighting their advantages in terms of performance and scalability. Future work could focus on optimizing specific operations and enhancing error handling for increased robustness.

# **1. Introduction**

In today's digital world, online shopping platforms are essential tools that enable customers to browse, select, and purchase items conveniently. An efficient e-commerce cart system forms the backbone of such platforms by managing user selections and processing orders. This project aims to build a basic e-commerce cart system using Java programming with appropriate data structures for managing cart items and order tracking. The system is designed to provide a simplified simulation of how shopping platforms work, while also demonstrating the effectiveness of data structures such as linked lists and queues.

## **1.1 Data Structure and Operations**

This system uses two key data structures: Linked List – Used to manage items in the shopping cart. A linked list allows dynamic memory allocation, easy insertion and deletion of items, and flexibility to grow or shrink during runtime. Operations: Add item, remove item, display items, calculate total price. Queue – Used for order processing. A queue follows the First-In-First-Out (FIFO) principle, ensuring that the earliest placed orders are processed first. Operations: Place order (enqueue), process order (dequeue), check order queue status. These data structures are chosen to simulate real-life behaviors in e-commerce systems while maintaining efficient performance.

## **1.2 Motivation**

The motivation for developing this system stems from the increasing reliance on e-commerce platforms in both local and global markets. Understanding how such systems operate internally can help developers build more efficient, scalable, and user-friendly applications. Additionally, this project aligns with the course objectives by allowing students to apply data structures practically while enhancing problem-solving and system design skills.

### **1.3 Problem statement**

Building a fully functional e-commerce platform involves multiple layers of complexity, including real-time updates, user interaction, and secure transactions. However, students often lack hands-on experience in designing systems that require managing dynamic data. This project addresses that gap by providing a simplified, manageable version of an e-commerce system, enabling students to understand how to apply linked lists and queues in a real-world context to handle tasks such as cart management and order tracking.

### **1.4 Objectives**

- Develop a simple Java-based e-commerce cart system with GUI support using Java Swing.
- Apply linked lists to efficiently manage items within a user's cart.
- Use queues to track and process orders in a FIFO manner.
- Simulate a payment system to complete purchases.
- Store and retrieve cart and order data using text files.
- Analyze and document the system's time complexity and real-time performance.
- Practice object-oriented programming principles and file I/O handling in Java.

## **2. Literature Review: Data Structures in E-commerce**

Efficient data structures are critical for building scalable and responsive e-commerce applications. As shopping behavior becomes more dynamic and inventory systems move to real-time updates, traditional static approaches often fail to meet performance expectations.

### **2.1 Importance of Efficient Data Structures in E-Commerce**

In an e-commerce context, data structures drive the core mechanisms—searching, sorting, filtering, cart management, order processing, and personalization. Efficient data organization directly affects response time and customer satisfaction. For instance, hash maps and tries are commonly used to index product catalogs, optimize search queries, and manage inventory faster than linear scans [1].

Research by Choudhary et al. introduced an intelligent shopping cart system that updates inventory in real time and supports seamless checkout experiences through smart structuring of product and billing data [2]. These systems rely on low-latency data operations made possible by dynamic data structures such as linked lists and hash tables.

### **2.2 Linked Lists vs. Arrays/ArrayLists for Shopping Cart Management**

Shopping carts are a core feature of e-commerce platforms, often requiring dynamic and frequent insertions and deletions. Linked Lists are naturally suited for such scenarios due to their ability to perform  $O(1)$  insertions and deletions without shifting elements [3]. This is ideal when users frequently modify their carts—adding or removing items mid-session.

In contrast, Arrays or ArrayLists offer  $O(1)$  access by index, making them faster for fixed-size iteration and display purposes. However, they require costly resizing when exceeding capacity and inefficient deletions in the middle of the list due to element shifting [3]. Despite that, ArrayLists benefit from strong cache locality, making them faster in CPU-intensive environments [6].

A balanced design pattern involves using ArrayLists for front-end rendering (fast display, sequential access), and Linked Lists on the back-end (for efficient dynamic updates). Reddit developers confirm that while linked lists are academically elegant, arrays (and circular buffers) offer more consistent performance on modern processors [6].

**Comparison Table: Linked List vs. Array/ArrayList for Shopping Carts**

Criteria	Linked List	Array / ArrayList
Structure Type	Dynamic data structure	Static (Array) / Dynamic with overhead (ArrayList)
Memory Allocation	Allocated as needed (node by node)	Pre-allocated or resized in chunks
Add/Remove Items (Middle)	Efficient – $O(1)$ if pointer is known	Less efficient – $O(n)$ due to shifting elements
Add to End	$O(1)$ if the tail pointer is maintained	$O(1)$ amortized (ArrayList), $O(n)$ if resizing (Array)
Search by Index	$O(n)$ – requires traversal of nodes	$O(1)$ – direct access via index
Memory Overhead	Higher – stores pointers with each element	Lower – stores only data, unless resized
Iteration Performance	Slower – exhibits poorer cache locality	Faster – benefits from good cache performance
Ease of Implementation	Slightly complex – needs custom class and pointer logic	Simple – native support in most languages



## **2.3 FIFO Queues for Order Processing**

Fair order fulfillment is a fundamental requirement for customer trust. Queue structures (First-In, First-Out) are vital for ensuring that customer orders are processed in the exact sequence they are received. This is especially important during peak load situations like flash sales or promotional events.

A queue can be implemented using a linked list or circular array, both supporting  $O(1)$  enqueue and dequeue operations. The use of FIFO principles in web servers and inventory systems prevents order starvation and supports predictable system behavior [4]. Yuan and Fernandez also note that pattern-based architectures for B2C e-commerce rely heavily on abstract queueing models to manage user sessions and cart checkouts [5].

### 3. Discussion & Analysis

This section provides a detailed discussion and analysis of the system's code structure and overall flow.

#### 3.1 Variables and Methods

Below is a breakdown of the key Java classes, their member variables, and their core methods.

##### Class: **Product**

Represents a product entity with attributes for identification, name, pricing, and available stock. Provides methods for data access, modification, and serialization/deserialization to and from text format.

##### Variables

- `String id` – Unique identifier of the product.
- `String name` – Name of the product.
- `double price` – Unit price of the product.
- `int stock` – Current available stock quantity.

##### Methods

- `Product(String id, String name, double price, int stock)` – Constructs a `Product` object by initializing all attributes.
- `String getId()` – Returns the product's unique ID.
- `String getName()` – Returns the product name.
- `double getPrice()` – Returns the product's price.
- `int getStock()` – Returns the available stock.
- `void setStock(int stock)` – Updates the stock quantity for the product.
- `toString()` – Converts the product details into a comma-separated string for file storage.
- `static Product fromString(String line)` – Deserializes a comma-separated string into a `Product` object (returns `null` if format is invalid)

**Class: `CartItem`**

Represents a single node in a linked list–based shopping cart structure. Each node stores a product, its quantity, and a reference to the next cart item in the list.

**Variables**

- `Product product` – Reference to the product associated with this cart item.
- `int quantity` – Number of units of the product in the cart.
- `CartItem next` – Pointer to the next `CartItem` node in the linked list.

**Methods**

- `CartItem(Product product, int quantity)` – Constructs a `CartItem` node with the specified product and quantity, initializing the `next` pointer to `null`.
- `Product getProduct()` – Returns the product object for this cart item.
- `int getQuantity()` – Returns the quantity of the product in the cart.
- `void setQuantity(int quantity)` – Updates the quantity of the product in the cart.
- `CartItem getNext()` – Returns the next `CartItem` node in the linked list.
- `void setNext(CartItem next)` – Sets the pointer to the next `CartItem` node in the linked list.

**Class:** `Cart` (*Linked List Implementation*)

Represents a shopping cart implemented as a singly linked list, where each node is a `CartItem`. Supports adding, removing, displaying, and calculating totals for products, as well as performance testing for bulk operations.

### Variables

- `CartItem head` – Pointer to the first item in the cart linked list.

### Methods

- `Cart()` – Constructor that initializes an empty cart with the `head` set to `null`.
- `void addItem(Product product, int quantity)` – Adds a new item to the beginning of the cart list, or updates the quantity if the product already exists.
- `void removeItem(String productId)` – Searches the cart for the specified product ID and removes it from the list.
- `void displayCart()` – Prints each product name and quantity currently in the cart.
- `double calculateTotal()` – Calculates and returns the total price of all items in the cart.
- `CartItem getHead()` – Returns the `head` pointer of the linked list.
- `static void performanceTest(int numItems)` – Measures and prints execution time (in nanoseconds) for adding and removing a given number of items from the cart.

## Class: `Order`

Represents a customer's order, including the associated shopping cart, total cost, order ID, username, and timestamp. Supports order creation, data retrieval, text-based serialization/deserialization, and formatted output for storage or display.

### Variables

- `String orderId` – Unique identifier for the order, generated automatically at creation.
- `String username` – The username of the customer who placed the order.
- `Cart cart` – The shopping cart containing products for this order.
- `double total` – The total cost of the order.
- `Date timestamp` – The date and time when the order was created.

### Constructor

- `Order(String username, Cart cart, double total)` – Creates a new order with the given username, cart, and total cost. Automatically generates an order ID and sets the timestamp to the current date and time.

### Methods

- `String getOrderId()` – Returns the unique order ID.
- `void setOrderId(String orderId)` – Updates the order ID.
- `String getUsername()` – Returns the username associated with the order.
- `Cart getCart()` – Returns the shopping cart for this order.
- `double getTotal()` – Returns the total cost of the order.
- `Date getTimestamp()` – Returns the timestamp of when the order was created.
- `String toString()` – Returns a formatted string representing the order details, including all items in the cart.
- `String toFileString()` – Serializes the order into a file-friendly string format, with items listed separately.
- `static Order fromFileString(String orderData, String itemsData)` – Creates an Order object from serialized order and item data, reconstructing the cart from the item list.

**Class: OrderQueue** (*Queue Implementation*)

Represents a queue structure for managing customer orders in First-In-First-Out (FIFO) order. Supports efficient enqueue and dequeue operations, as well as empty-state checks.

**Inner Class: Node**

Represents a node in the queue containing an order and a reference to the next node.

- **Variables**

- Order order – The order stored in the node.
- Node next – Pointer to the next node in the queue.

- **Constructor**

- Node(Order order) – Constructor to initialize the node with an order.

**Variables**

- Node front – Pointer to the first order in the queue.
- Node rear – Pointer to the last order in the queue, allowing O(1) enqueue operations.

**Methods**

- OrderQueue() – Constructor that initializes an empty queue with front and rear set to null.
- void placeOrder(Order order) – Adds the given order to the rear of the queue.
- Order processNextOrder() – Removes and returns the order at the front of the queue; returns null if the queue is empty.
- boolean isEmpty() – Returns true if the queue contains no orders, otherwise returns false.

## Class: User

### Variables

- `String username` – Stores the user's unique identifier or login name.
- `String password` – Stores the user's password in plain text (note: should be hashed in production for security).

### Methods

- `User(String username, String password)` – Constructor that initializes a new user with the provided username and password.
- `String getUsername()` – Returns the username of the user.
- `String getPassword()` – Returns the password of the user.
- `String toString()` – Converts the User object to a comma-separated string representation: "username,password".
- `static User fromString(String line)` – Parses a comma-separated string into a User object; returns null if the input format is invalid.

## 3.2 Time Complexity (Theoretical Analysis)

Operation	Data Structure	Worst-case Time Complexity	Justification
addItem (cart)	Linked List	O(1)	With a tail pointer, appending requires only pointer updates.
removeItem (cart)	Linked List	O(n)	Requires searching the list for the target item before removal.
placeOrder	Queue	O(1)	Adding to the rear of the queue requires only pointer updates.
processNextOrder	Queue	O(1)	Removing from the front requires only pointer updates.

### 3.3 Run-Time Complexity (Practical Measurement)

Operation	Number of Items	Measured Time (nanoseconds)
Add	100	158,000
Remove	100	436,000
Add	150	242,000
Remove	150	665,000

#### Analysis:

The measured results generally match the theoretical expectations:

- **Add** operations are consistently fast and scale minimally with input size, supporting the  **$O(1)$**  complexity claim.
- **Remove** operations take longer as the number of items increases, confirming the  **$O(n)$**  complexity.

Any minor deviations are due to **JVM overhead, garbage collection pauses, and system background processes.**



### **3.4 Justification for chosen Data Structure**

#### **Linked List for Shopping Cart**

The Linked List was selected over arrays or ArrayLists because it inherently supports dynamic resizing—memory is allocated on demand as items are added, eliminating the need for costly resizing operations. Unlike arrays, where insertions or deletions in the middle require shifting all subsequent elements ( $O(n)$  operation), linked lists can handle these operations in  $O(1)$  time when the node reference is known. This makes it highly efficient for a shopping cart scenario where users frequently add, remove, or modify items at unpredictable positions during a session. Additionally, by maintaining both a head and tail pointer, appending to the cart remains constant time regardless of cart size. This efficiency translates directly into a smoother, more responsive user experience.

#### **Queue for Order Processing**

Order processing demands fairness and predictability, making the Queue—with its First-In-First-Out (FIFO) discipline—the ideal choice. In this context, FIFO ensures that the first order placed is always the first one processed, reinforcing trust and transparency in the system. The queue structure also delivers  $O(1)$  time complexity for both enqueue (placing orders) and dequeue (processing orders) operations, ensuring that performance remains consistent even during high-load scenarios such as flash sales. This predictability not only streamlines backend workflows but also guarantees equitable treatment of all customer orders, a cornerstone of good e-commerce service design.

## 4. Run Java Code

This section contains the complete Java source code, showcasing clear and maintainable implementation details.

Simple E-Commerce Cart System/

```
|— Src/           // Contains all Java source code for the application.
| |— Model/       // Defines the core data entities and business logic of the application.
| | |— Cart.java   // Manages the user's shopping cart using a Linked List.
| | |— CartItem.java // Represents an individual product item within the shopping
cart.
| | |— Order.java   // Defines the structure for a customer's order.
| | |— OrderManager.java // Handles order creation, storage, and retrieval operations.
| | |— OrderQueue.java // Implements the queue for processing customer orders.
| | |— Product.java  // Represents a product available in the e-commerce system.
| | |— User.java      // Represents a system user with login credentials.
| | |— UserManager.java // Manages user registration, authentication, and retrieval.
| |— Ui/           // Contains Java Swing classes for the graphical user interface.
| | |— LoginFrame.java // Handles user login interface and authentication flow.
| | |— MainFrame.java  // Handles the main application window and user interactions.
| |— App.java       // The main entry point for the application.
|— Data/           // Stores persistent application data in text files.
|— Orders.txt       // Stores records of processed customer orders.
|— Products.txt     // Contains the list of available products.
|— Users.txt        // Stores user-related information.
```

## Cart.java

// Manages the user's shopping cart using a Linked List.

```
package model;

public class Cart {
    private CartItem head;

    public Cart() {
        head = null;
    }

    public void addItem(Product product, int quantity) {
        CartItem current = head;
        while (current != null) {
            if (current.getProduct().getId().equals(product.getId())) {
                current.setQuantity(current.getQuantity() + quantity);
                return;
            }
            current = current.getNext();
        }
        CartItem newItem = new CartItem(product, quantity);
        newItem.setNext(head);
        head = newItem;
    }

    public void removeItem(String productId) {
        CartItem current = head, prev = null;
        while (current != null) {
            if (current.getProduct().getId().equals(productId)) {
                if (prev == null) {
                    head = current.getNext();
                } else {
                    prev.setNext(current.getNext());
                }
                return;
            }
            prev = current;
            current = current.getNext();
        }
    }

    public void displayCart() {
        CartItem current = head;
        while (current != null) {
            System.out.println(current.getProduct().getName() + " x " +
current.getQuantity());
            current = current.getNext();
        }
    }
}
```

```

    }
}

public double calculateTotal() {
    double total = 0;
    CartItem current = head;
    while (current != null) {
        total += current.getProduct().getPrice() * current.getQuantity();
        current = current.getNext();
    }
    return total;
}

public CartItem getHead() { return head; }

public static void performanceTest(int numItems) {
    Cart cart = new Cart();
    long startAdd = System.nanoTime();
    for (int i = 0; i < numItems; i++) {
        cart.addItem(new Product("ID" + i, "Product" + i, 1.0, 100), 1);
    }
    long endAdd = System.nanoTime();
    long addTime = endAdd - startAdd;

    long startRemove = System.nanoTime();
    for (int i = 0; i < numItems; i++) {
        cart.removeItem("ID" + i);
    }
    long endRemove = System.nanoTime();
    long removeTime = endRemove - startRemove;

    System.out.println("Add " + numItems + " items: " + addTime + " ns");
    System.out.println("Remove " + numItems + " items: " + removeTime + "
ns");
}
}

```

## CartItem.java

// Represents an individual product item within the shopping cart.

```
package model;

public class CartItem {
    private Product product;
    private int quantity;
    private CartItem next;

    public CartItem(Product product, int quantity) {
        this.product = product;
        this.quantity = quantity;
        this.next = null;
    }

    public Product getProduct() { return product; }
    public int getQuantity() { return quantity; }
    public void setQuantity(int quantity) { this.quantity = quantity; }
    public CartItem getNext() { return next; }
    public void setNext(CartItem next) { this.next = next; }
}
```

## Order.java

// Defines the structure for a customer's order.

```
package model;

import java.util.Date;

public class Order {
    private String orderId;
    private String username;
    private Cart cart;
    private double total;
    private Date timestamp;

    public Order(String username, Cart cart, double total) {
        this.orderId = generateOrderId();
        this.username = username;
        this.cart = cart;
        this.total = total;
        this.timestamp = new Date();
    }

    private String generateOrderId() {
        return "ORD" + System.currentTimeMillis();
    }

    public String getOrderId() { return orderId; }
    public void setOrderId(String orderId) { this.orderId = orderId; }
    public String getUsername() { return username; }
    public Cart getCart() { return cart; }
    public double getTotal() { return total; }
    public Date getTimestamp() { return timestamp; }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Order ID: ").append(orderId).append("\n");
        sb.append("User: ").append(username).append("\n");
        sb.append("Order at: ").append(timestamp).append(", Total: ");
        sb.append(String.format("%.2f", total)).append("\n");

        CartItem current = cart.getHead();
        while (current != null) {
            Product product = current.getProduct();

            sb.append(product.getId()).append(", ").append(product.getName()).append(", ")

```

```

        .append(current.getQuantity()).append(",") .append(String.format("%.2f",
product.getPrice())) .append("\n");
        current = current.getNext();
    }
    sb.append("---");
    return sb.toString();
}

public String toFileString() {
    StringBuilder sb = new StringBuilder();
    sb.append(orderId).append("|").append(username).append("|")

.append(timestamp.getTime()).append("|").append(String.format("%.2f",
total)).append("\n");

    CartItem current = cart.getHead();
    while (current != null) {
        Product product = current.getProduct();

sb.append(product.getId()).append(",") .append(product.getName()).append(",")

.append(current.getQuantity()).append(",") .append(String.format("%.2f",
product.getPrice())) .append("\n");
        current = current.getNext();
    }
    sb.append("---");
    return sb.toString();
}

public static Order fromFileString(String orderData, String itemsData) {
    String[] parts = orderData.split("\\|");
    if (parts.length != 4) return null;

    String orderId = parts[0];
    String username = parts[1];
    long timestamp = Long.parseLong(parts[2]);
    double total = Double.parseDouble(parts[3]);

    // Create a temporary cart to hold the items
    Cart tempCart = new Cart();
    String[] lines = itemsData.split("\n");
    for (String line : lines) {
        if (line.equals("---")) break;
        String[] itemParts = line.split(",");
        if (itemParts.length == 4) {
            String id = itemParts[0];
            String name = itemParts[1];
            int quantity = Integer.parseInt(itemParts[2]);

```

```
        double price = Double.parseDouble(itemParts[3]);
        Product product = new Product(id, name, price, 0); // stock
not relevant for orders
        tempCart.addItem(product, quantity);
    }
}

Order order = new Order(username, tempCart, total);
order.orderId = orderId;
order.timestamp = new Date(timestamp);
return order;
}
}
```



## OrderManager.java

// Handles order creation, storage, and retrieval operations

```
package model;

import java.io.*;
import java.util.*;

public class OrderManager {
    private static final String ORDERS_FILE = "data/orders.txt";
    private List<Order> allOrders;

    public OrderManager() {
        allOrders = new ArrayList<>();
        loadOrders();
    }

    private void loadOrders() {
        allOrders.clear();
        System.out.println("Starting to load orders from file: " +
ORDERS_FILE);
        try (BufferedReader br = new BufferedReader(new
FileReader(ORDERS_FILE))) {
            StringBuilder currentOrder = new StringBuilder();
            StringBuilder currentItems = new StringBuilder();
            String line;
            boolean readingOrder = false;
            boolean readingItems = false;
            int orderCount = 0;
            int lineNumber = 0;

            while ((line = br.readLine()) != null) {
                lineNumber++;
                System.out.println("Line " + lineNumber + ": '" + line + "'");

                if (line.startsWith("Order ID:")) {
                    // Start of a new order
                    if (readingOrder && currentOrder.length() > 0) {
                        // Process previous order
                        System.out.println("Processing previous order...");
                        processOrder(currentOrder.toString(),
currentItems.toString());
                        orderCount++;
                    }
                    currentOrder = new StringBuilder();
                    currentItems = new StringBuilder();
                    readingOrder = true;
                    readingItems = false;
                }
            }
        }
    }
}
```

```

        currentOrder.append(line).append("\n");
        System.out.println("Started reading new order");
    } else if (line.startsWith("User:")) {
        // Continue building current order
        currentOrder.append(line).append("\n");
        System.out.println("Added user to current order");
    } else if (line.startsWith("Order at:")) {
        currentOrder.append(line).append("\n");
        readingItems = true;
        System.out.println("Started reading items");
    } else if (line.equals("---")) {
        if (readingOrder && currentOrder.length() > 0) {
            // Process the complete order
            System.out.println("Processing complete order...");
            processOrder(currentOrder.toString(),
currentItems.toString());
            orderCount++;
        }
        currentOrder = new StringBuilder();
        currentItems = new StringBuilder();
        readingOrder = false;
        readingItems = false;
        System.out.println("Finished order, resetting");
    } else if (readingItems && !line.trim().isEmpty()) {
        currentItems.append(line).append("\n");
    }
}

// Process any remaining order at the end of file
if (readingOrder && currentOrder.length() > 0) {
    System.out.println("Processing final order...");
    processOrder(currentOrder.toString(),
currentItems.toString());
    orderCount++;
}

System.out.println("Loaded " + orderCount + " orders from file");
} catch (IOException e) {
    System.out.println("Orders file not found or empty - starting with
no orders");
    // File might not exist yet, which is fine for new installations
}

private void processOrder(String orderData, String itemsData) {
    try {
        // Parse the order header
        String[] lines = orderData.split("\n");
        String orderId = "";

```

```

String username = "";
String timestampStr = "";
double total = 0.0;

for (String line : lines) {
    if (line.startsWith("Order ID:")) {
        orderId = line.substring("Order ID: ".length()).trim();
    } else if (line.startsWith("User:")) {
        username = line.substring("User: ".length()).trim();
    } else if (line.startsWith("Order at:")) {
        String[] parts = line.split(", Total: \\$");
        if (parts.length == 2) {
            timestampStr = parts[0].substring("Order at: ".length()).trim();
            total = Double.parseDouble(parts[1].trim());
        }
    }
}

System.out.println("Processing order - ID: " + orderId + ",
User: " + username + ", Total: " + total);

if (!orderId.isEmpty() && !username.isEmpty()) {
    // Create a temporary cart to hold the items
    Cart tempCart = new Cart();
    String[] itemLines = itemsData.split("\n");
    for (String itemLine : itemLines) {
        if (itemLine.equals("---") || itemLine.trim().isEmpty())
            continue;

        String[] itemParts = itemLine.split(",");
        if (itemParts.length == 4) {
            String id = itemParts[0];
            String name = itemParts[1];
            int quantity = Integer.parseInt(itemParts[2]);
            double price = Double.parseDouble(itemParts[3]);
            Product product = new Product(id, name, price, 0);
            tempCart.addItem(product, quantity);
        }
    }

    // Create and add the order
    Order order = new Order(username, tempCart, total);
    order.setOrderId(orderId);
    // Parse timestamp (simplified - you might want to use a
proper date parser)
    allOrders.add(order);
    System.out.println("Successfully processed order for user: " +
username);
} else {

```

```

        System.out.println("Skipping order - missing orderId or
username");
    }
    } catch (Exception e) {
        // Skip malformed orders
        System.err.println("Error processing order: " + e.getMessage());
        e.printStackTrace();
    }
}

public void saveOrder(Order order) {
    allOrders.add(order);
    appendOrderToFile(order);
}

private void appendOrderToFile(Order order) {
    try (BufferedWriter bw = new BufferedWriter(new
FileWriter(ORDERS_FILE, true))) {
        bw.write(order.toString() + "\n\n");
        System.out.println("Order saved successfully: " +
order.getOrderId() + " for user: " + order.getUsername());
    } catch (IOException e) {
        System.err.println("Failed to save order to file: " +
e.getMessage());
        e.printStackTrace();
    }
}

private void saveOrders() {
    try (BufferedWriter bw = new BufferedWriter(new
FileWriter(ORDERS_FILE))) {
        for (Order order : allOrders) {
            bw.write(order.toString() + "\n\n");
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public List<Order> getUserOrders(String username) {
    List<Order> userOrders = new ArrayList<>();
    for (Order order : allOrders) {
        if (order.getUsername().equals(username)) {
            userOrders.add(order);
        }
    }
    // Sort by timestamp (newest first)
    userOrders.sort((o1, o2) ->
o2.getTimestamp().compareTo(o1.getTimestamp()));
}

```

```

        return userOrders;
    }

    public List<Order> getAllOrders() {
        return new ArrayList<>(allOrders);
    }

    public void clearOrders() {
        allOrders.clear();
        saveOrders();
    }

    /**
     * Rebuilds the orders file from memory - useful for maintenance
     * This ensures file consistency with in-memory data
     */
    public void rebuildOrdersFile() {
        saveOrders();
    }

    /**
     * Debug method to show current orders in memory
     */
    public void debugPrintOrders() {
        System.out.println("=== Current Orders in Memory ===");
        System.out.println("Total orders: " + allOrders.size());
        for (Order order : allOrders) {
            System.out.println("Order ID: " + order.getOrderID() +
                               ", User: " + order.getUsername() +
                               ", Total: $" + String.format("%.2f",
order.getTotal()));
        }
        System.out.println("=====");
    }
}

```

## OrderQueue.java

// Implements the queue for processing customer orders on a FIFO basis.

```
package model;

public class OrderQueue {
    private static class Node {
        Order order;
        Node next;
        Node(Order order) { this.order = order; }
    }
    private Node front, rear;

    public OrderQueue() {
        front = rear = null;
    }

    public void placeOrder(Order order) {
        Node node = new Node(order);
        if (rear == null) {
            front = rear = node;
        } else {
            rear.next = node;
            rear = node;
        }
    }

    public Order processNextOrder() {
        if (front == null) return null;
        Order order = front.order;
        front = front.next;
        if (front == null) rear = null;
        return order;
    }

    public boolean isEmpty() {
        return front == null;
    }
}
```

## Product.java

// Represents a product available in the e-commerce system.

```
package model;

public class Product {
    private String id;
    private String name;
    private double price;
    private int stock;

    public Product(String id, String name, double price, int stock) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public String getId() { return id; }
    public String getName() { return name; }
    public double getPrice() { return price; }
    public int getStock() { return stock; }
    public void setStock(int stock) { this.stock = stock; }

    @Override
    public String toString() {
        return id + "," + name + "," + price + "," + stock;
    }

    public static Product fromString(String line) {
        String[] parts = line.split(",");
        if (parts.length != 4) return null;
        return new Product(parts[0], parts[1], Double.parseDouble(parts[2]),
Integer.parseInt(parts[3]));
    }
}
```

## User.java

// Represents a system user with login credentials.

```
package model;

public class User {
    private String username;
    private String password;

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public String getUsername() { return username; }
    public String getPassword() { return password; }

    @Override
    public String toString() {
        return username + "," + password;
    }

    public static User fromString(String line) {
        String[] parts = line.split(",");
        if (parts.length != 2) return null;
        return new User(parts[0], parts[1]);
    }
}
```



## UserManager.java

// Manages user registration, authentication, and retrieval.

```
package model;

import java.io.*;
import java.util.*;

public class UserManager {
    private static final String USERS_FILE = "data/users.txt";
    private List<User> users;

    public UserManager() {
        users = new ArrayList<>();
        loadUsers();
    }

    private void loadUsers() {
        users.clear();
        try (BufferedReader br = new BufferedReader(new
        FileReader(USERS_FILE))) {
            String line;
            while ((line = br.readLine()) != null && !line.trim().isEmpty()) {
                User user = User.fromString(line);
                if (user != null) {
                    users.add(user);
                }
            }
        } catch (IOException e) {
            // File might not exist yet, which is fine for new installations
        }
    }

    private void saveUsers() {
        try (BufferedWriter bw = new BufferedWriter(new
        FileWriter(USERS_FILE))) {
            for (User user : users) {
                bw.write(user.toString() + "\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public boolean registerUser(String username, String password) {
        // Check if username already exists
        for (User user : users) {
            if (user.getUsername().equals(username)) {
```

```

        return false; // Username already exists
    }
}

// Add new user
User newUser = new User(username, password);
users.add(newUser);
saveUsers();
return true;
}

public User loginUser(String username, String password) {
    for (User user : users) {
        if (user.getUsername().equals(username) &&
user.getPassword().equals(password)) {
            return user;
        }
    }
    return null; // Invalid credentials
}

public boolean userExists(String username) {
    for (User user : users) {
        if (user.getUsername().equals(username)) {
            return true;
        }
    }
    return false;
}

public boolean removeUser(String username) {
    for (int i = 0; i < users.size(); i++) {
        if (users.get(i).getUsername().equals(username)) {
            users.remove(i);
            saveUsers();
            return true;
        }
    }
    return false;
}

public static void performanceTest(int numUsers) {
    UserManager userManager = new UserManager();
    long startAdd = System.nanoTime();
    for (int i = 0; i < numUsers; i++) {
        userManager.registerUser("user" + i, "password" + i);
    }
    long endAdd = System.nanoTime();
    long addTime = endAdd - startAdd;
}

```

```
        long startRemove = System.nanoTime();
        for (int i = 0; i < numUsers; i++) {
            userManager.removeUser("user" + i);
        }
        long endRemove = System.nanoTime();
        long removeTime = endRemove - startRemove;

        System.out.println("Add " + numUsers + " users: " + addTime + " ns");
        System.out.println("Remove " + numUsers + " users: " + removeTime + "
ns");
    }
}
```

## LoginFrame.java

// Handles user login interface and authentication flow.

```
package ui;

import javax.swing.*;
import java.awt.*;
import model.UserManager;
import model.User;
import ui.MainFrame;

public class LoginFrame extends JFrame {
    private UserManager userManager;
    private JTextField usernameField;
    private JPasswordField passwordField;
    private JButton loginButton;
    private JButton registerButton;
    private JButton switchModeButton;
    private boolean isLoginMode = true;
    private MainFrame mainFrame;

    public LoginFrame() {
        userManager = new UserManager();
        setupUI();
    }

    private void setupUI() {
        setTitle("E-Commerce Cart System - Login");
        setSize(400, 250);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setLayout(new BorderLayout());

        // Create main panel
        JPanel mainPanel = new JPanel(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.insets = new Insets(5, 5, 5, 5);

        // Title
        JLabel titleLabel = new JLabel("Welcome to E-Commerce Cart System");
        titleLabel.setFont(new Font("Arial", Font.BOLD, 16));
        gbc.gridx = 0;
        gbc.gridy = 0;
        gbc.gridwidth = 2;
        mainPanel.add(titleLabel, gbc);

        // Username
        JLabel usernameLabel = new JLabel("Username:");
```

```

gbc.gridx = 0;
gbc.gridy = 1;
gbc.gridwidth = 1;
gbc.anchor = GridBagConstraints.EAST;
mainPanel.add(usernameLabel, gbc);

usernameField = new JTextField(20);
gbc.gridx = 1;
gbc.anchor = GridBagConstraints.WEST;
mainPanel.add(usernameField, gbc);

// Password
JLabel passwordLabel = new JLabel("Password:");
gbc.gridx = 0;
gbc.gridy = 2;
gbc.anchor = GridBagConstraints.EAST;
mainPanel.add(passwordLabel, gbc);

passwordField = new JPasswordField(20);
gbc.gridx = 1;
gbc.anchor = GridBagConstraints.WEST;
mainPanel.add(passwordField, gbc);

// Buttons panel
JPanel buttonPanel = new JPanel(new FlowLayout());

loginButton = new JButton("Login");
loginButton.addActionListener(e -> handleLogin());

registerButton = new JButton("Register");
registerButton.addActionListener(e -> handleRegister());
registerButton.setVisible(false);

switchModeButton = new JButton("Switch to Register");
switchModeButton.addActionListener(e -> switchMode());

buttonPanel.add(loginButton);
buttonPanel.add(registerButton);
buttonPanel.add(switchModeButton);

gbc.gridx = 0;
gbc.gridy = 3;
gbc.gridwidth = 2;
gbc.anchor = GridBagConstraints.CENTER;
mainPanel.add(buttonPanel, gbc);

add(mainPanel, BorderLayout.CENTER);
}

```

```

private void switchMode() {
    isLoginMode = !isLoginMode;

    if (isLoginMode) {
        setTitle("E-Commerce Cart System - Login");
        loginButton.setVisible(true);
        registerButton.setVisible(false);
        switchModeButton.setText("Switch to Register");
    } else {
        setTitle("E-Commerce Cart System - Register");
        loginButton.setVisible(false);
        registerButton.setVisible(true);
        switchModeButton.setText("Switch to Login");
    }

    // Clear fields
    usernameField.setText("");
    passwordField.setText("");

    revalidate();
    repaint();
}

private void handleLogin() {
    String username = usernameField.getText().trim();
    String password = new String(passwordField.getPassword());

    if (username.isEmpty() || password.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Please fill in all fields.",
"Error", JOptionPane.ERROR_MESSAGE);
        return;
    }

    User user = userManager.loginUser(username, password);
    if (user != null) {
        JOptionPane.showMessageDialog(this, "Login successful! Welcome, "
+ username, "Success", JOptionPane.INFORMATION_MESSAGE);
        openMainFrame(user);
    } else {
        JOptionPane.showMessageDialog(this, "Invalid username or
password.", "Login Failed", JOptionPane.ERROR_MESSAGE);
    }
}

private void handleRegister() {
    String username = usernameField.getText().trim();
    String password = new String(passwordField.getPassword());

    if (username.isEmpty() || password.isEmpty()) {

```

```

        JOptionPane.showMessageDialog(this, "Please fill in all fields.",
"Error", JOptionPane.ERROR_MESSAGE);
        return;
    }

    if (username.length() < 3) {
        JOptionPane.showMessageDialog(this, "Username must be at least 3
characters long.", "Error", JOptionPane.ERROR_MESSAGE);
        return;
    }

    if (password.length() < 6) {
        JOptionPane.showMessageDialog(this, "Password must be at least 6
characters long.", "Error", JOptionPane.ERROR_MESSAGE);
        return;
    }

    if (userManager.userExists(username)) {
        JOptionPane.showMessageDialog(this, "Username already exists.
Please choose another one.", "Error", JOptionPane.ERROR_MESSAGE);
        return;
    }

    boolean success = userManager.registerUser(username, password);
    if (success) {
        JOptionPane.showMessageDialog(this, "Registration successful! You
can now login.", "Success", JOptionPane.INFORMATION_MESSAGE);
        switchMode(); // Switch back to login mode
    } else {
        JOptionPane.showMessageDialog(this, "Registration failed. Please
try again.", "Error", JOptionPane.ERROR_MESSAGE);
    }
}

private void openMainFrame(User user) {
    if (mainFrame == null) {
        mainFrame = new MainFrame(user);
    }
    mainFrame.setVisible(true);
    this.setVisible(false);
}
}

```

## MainFrame.java

// Handles the main application window and user interactions.

```
package ui;

import java.awt.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import model.Cart;
import model.CartItem;
import model.Order;
import model.OrderManager;
import model.OrderQueue;
import model.Product;
import model.User;

public class MainFrame extends JFrame {
    private java.util.List<Product> products = new ArrayList<>();
    private JTable productTable;
    private DefaultTableModel productTableModel;
    private Cart cart = new Cart();
    private OrderQueue orderQueue = new OrderQueue();
    private OrderManager orderManager;
    private User currentUser;
    private JButton viewCartBtn;

    public MainFrame() {
        this(null);
    }

    public MainFrame(User user) {
        this.currentUser = user;
        this.orderManager = new OrderManager();
        setTitle("E-Commerce Cart System" + (user != null ? " - Welcome " +
user.getUsername() : ""));
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setLayout(new BorderLayout());

        // Load products from file
        loadProducts();

        // Product Table
        String[] columns = {"ID", "Name", "Price", "Stock"};
        productTableModel = new DefaultTableModel(columns, 0) {
```



```

        public boolean isCellEditable(int row, int column) { return false;
    }

    };
    productTable = new JTable(productTableModel);
    refreshProductTable();

    JScrollPane scrollPane = new JScrollPane(productTable);

    // Add to Cart Button
    JButton addToCartBtn = new JButton("🛒 Add to Cart");
    addToCartBtn.addActionListener(e -> addSelectedProductToCart());

    // View Cart Button
    this.viewCartBtn = new JButton("🛒 View Cart (" +
getCartItemCount(cart) + ")");
    this.viewCartBtn.addActionListener(e -> showCartDialog());

    // Order History Button
    JButton orderHistoryBtn = new JButton("Order History");
    orderHistoryBtn.addActionListener(e -> showOrderHistoryDialog());

    // Logout Button
    JButton logoutBtn = new JButton("Logout");
    logoutBtn.addActionListener(e -> logout());

    JPanel bottomPanel = new JPanel();
    bottomPanel.add(addToCartBtn);
    bottomPanel.add(viewCartBtn);
    bottomPanel.add(orderHistoryBtn);
    bottomPanel.add(logoutBtn);

    add(new JLabel("Product List", SwingConstants.CENTER),
BorderLayout.NORTH);
    add(scrollPane, BorderLayout.CENTER);
    add(bottomPanel, BorderLayout.SOUTH);
}

private void loadProducts() {
    products.clear();
    try (BufferedReader br = new BufferedReader(new
FileReader("data/products.txt"))) {
        String line;
        while ((line = br.readLine()) != null) {
            Product p = Product.fromString(line);
            if (p != null) products.add(p);
        }
    } catch (IOException e) {
        JOptionPane.showMessageDialog(this, "Failed to load products.",
"Error", JOptionPane.ERROR_MESSAGE);
    }
}

```

```

    }
}

private void refreshProductTable() {
    productTableModel.setRowCount(0);
    for (Product p : products) {
        productTableModel.addRow(new Object[]{p.getId(), p.getName(),
p.getPrice(), p.getStock()});
    }
}

private void addSelectedProductToCart() {
    int row = productTable.getSelectedRow();
    if (row == -1) {
        JOptionPane.showMessageDialog(this, "Please select a product.");
        return;
    }
    String id = (String) productTableModel.getValueAt(row, 0);
    Product selected = null;
    for (Product p : products) {
        if (p.getId().equals(id)) {
            selected = p;
            break;
        }
    }
    if (selected == null) return;
    String qtyStr = JOptionPane.showInputDialog(this, "Enter quantity:",
"1");
    if (qtyStr == null) return;
    int qty;
    try {
        qty = Integer.parseInt(qtyStr);
        if (qty <= 0 || qty > selected.getStock()) throw new Exception();
    } catch (Exception ex) {
        JOptionPane.showMessageDialog(this, "Invalid quantity.");
        return;
    }
    cart.addItem(selected, qty);
    JOptionPane.showMessageDialog(this, "Added to cart: " +
selected.getName() + " x " + qty);
    // Update cart button text
    updateCartButtonText();
}

private void showCartDialog() {
    // Build cart table data
    java.util.List<CartItem> cartItems = new ArrayList<>();
    CartItem current = cart.getHead();
    while (current != null) {

```

```

        cartItems.add(current);
        current = current.getNext();
    }
    if (cartItems.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Cart is empty.", "Cart",
JOptionPane.INFORMATION_MESSAGE);
        return;
    }

    // Enhanced cart table with subtotals
    String[] columns = {"Product", "Quantity", "Unit Price", "Subtotal"};
    Object[][] data = new Object[cartItems.size()][4];
    for (int i = 0; i < cartItems.size(); i++) {
        CartItem item = cartItems.get(i);
        double subtotal = item.getQuantity() *
item.getProduct().getPrice();
        data[i][0] = item.getProduct().getName();
        data[i][1] = item.getQuantity();
        data[i][2] = String.format("%.2f", item.getProduct().getPrice());
        data[i][3] = String.format("%.2f", subtotal);
    }

    DefaultTableModel cartModel = new DefaultTableModel(data, columns) {
        public boolean isCellEditable(int row, int col) { return false; }
    };
    JTable cartTable = new JTable(cartModel);
    cartTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    JScrollPane scrollPane = new JScrollPane(cartTable);

    // Enhanced buttons with better styling
    JButton removeBtn = new JButton("🗑 Remove");
    removeBtn.setBackground(new Color(255, 99, 71));
    removeBtn.setForeground(Color.WHITE);
    removeBtn.setFocusPainted(false);

    JButton editQtyBtn = new JButton("✏ Edit Quantity");
    editQtyBtn.setBackground(new Color(70, 130, 180));
    editQtyBtn.setForeground(Color.WHITE);
    editQtyBtn.setFocusPainted(false);

    JButton purchaseBtn = new JButton("💵 Proceed to Payment");
    purchaseBtn.setBackground(new Color(34, 139, 34));
    purchaseBtn.setForeground(Color.WHITE);
    purchaseBtn.setFocusPainted(false);
    purchaseBtn.setFont(purchaseBtn.getFont().deriveFont(Font.BOLD, 14f));

    // Remove item functionality
    removeBtn.addActionListener(e -> {
        int row = cartTable.getSelectedRow();

```

```

        if (row == -1) {
            JOptionPane.showMessageDialog(this, "Please select an item to
remove.", "No Selection", JOptionPane.WARNING_MESSAGE);
            return;
        }
        String prodName = (String) cartModel.getValueAt(row, 0);
        String prodId = null;
        for (CartItem item : cartItems) {
            if (item.getProduct().getName().equals(prodName)) {
                prodId = item.getProduct().getId();
                break;
            }
        }
        if (prodId != null) {
            int confirm = JOptionPane.showConfirmDialog(this,
                "Remove " + prodName + " from cart?", "Confirm Removal",
                JOptionPane.YES_NO_OPTION);
            if (confirm == JOptionPane.YES_OPTION) {
                cart.removeItem(prodId);
                updateCartButtonText();
            }
        }
        SwingUtilities.getWindowAncestor(cartTable).dispose();
        showCartDialog();
    }
}

// Edit quantity functionality
editQtyBtn.addActionListener(e -> {
    int row = cartTable.getSelectedRow();
    if (row == -1) {
        JOptionPane.showMessageDialog(this, "Please select an item to
edit.", "No Selection", JOptionPane.WARNING_MESSAGE);
        return;
    }
    String prodName = (String) cartModel.getValueAt(row, 0);
    CartItem selectedItem = null;
    for (CartItem item : cartItems) {
        if (item.getProduct().getName().equals(prodName)) {
            selectedItem = item;
            break;
        }
    }
    if (selectedItem != null) {
        String newQtyStr = JOptionPane.showInputDialog(this,
            "Enter new quantity for " + prodName + " (current: " +
selectedItem.getQuantity() + "):",
            String.valueOf(selectedItem.getQuantity()));
        if (newQtyStr != null && !newQtyStr.trim().isEmpty()) {

```

```

        try {
            int newQty = Integer.parseInt(newQtyStr);
            if (newQty <= 0) {
                JOptionPane.showMessageDialog(this, "Quantity must
be greater than 0.", "Invalid Quantity", JOptionPane.ERROR_MESSAGE);
                return;
            }
            if (newQty > selectedItem.getProduct().getStock()) {
                JOptionPane.showMessageDialog(this, "Quantity
exceeds available stock.", "Invalid Quantity", JOptionPane.ERROR_MESSAGE);
                return;
            }

            // Remove old item and add new one with updated
quantity
            cart.removeItem(selectedItem.getProduct().getId());
            cart.addItem(selectedItem.getProduct(), newQty);
            updateCartButtonText();

                                                                    ((JDialog)
SwingUtilities.getWindowAncestor(cartTable)).dispose();
            showCartDialog();
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(this, "Please enter a
valid number.", "Invalid Input", JOptionPane.ERROR_MESSAGE);
        }
    }
});

// Enhanced purchase functionality
purchaseBtn.addActionListener(e -> {
    if (cartItems.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Cart is empty.", "Empty
Cart", JOptionPane.WARNING_MESSAGE);
        return;
    }
    showPaymentDialog(cart);
});

// Button panel with better layout
JPanel btnPanel = new JPanel(new FlowLayout(FlowLayout.CENTER, 10,
5));

btnPanel.add(removeBtn);
btnPanel.add(editQtyBtn);
btnPanel.add(purchaseBtn);

// Enhanced total display
double total = cart.calculateTotal();

```

```

        JLabel totalLabel = new JLabel("Total: $" + String.format("%.2f",
total), SwingConstants.CENTER);
        totalLabel.setFont(totalLabel.getFont().deriveFont(Font.BOLD, 16f));
        totalLabel.setForeground(new Color(34, 139, 34));
        totalLabel.setBorder(BorderFactory.createEmptyBorder(10, 0, 10, 0));

        JPanel panel = new JPanel(new BorderLayout());
        panel.add(totalLabel, BorderLayout.NORTH);
        panel.add(scrollPane, BorderLayout.CENTER);
        panel.add(btnPanel, BorderLayout.SOUTH);

        JDialog dialog = new JDialog(this, "Shopping Cart - " +
currentUser.getUsername(), true);
        dialog.setContentPane(panel);
        dialog.setSize(600, 450);
        dialog.setLocationRelativeTo(this);
        dialog.setVisible(true);
    }

    private void showPaymentDialog(Cart cart) {
        // Create order summary panel
        JPanel summaryPanel = new JPanel(new BorderLayout());
        summaryPanel.setBorder(BorderFactory.createTitledBorder("Order
Summary"));

        // Build summary table
        java.util.List<CartItem> cartItems = new ArrayList<>();
        CartItem current = cart.getHead();
        while (current != null) {
            cartItems.add(current);
            current = current.getNext();
        }

        String[] columns = {"Product", "Quantity", "Unit Price", "Subtotal"};
        Object[][] data = new Object[cartItems.size()][4];
        for (int i = 0; i < cartItems.size(); i++) {
            CartItem item = cartItems.get(i);
            double subtotal = item.getQuantity() *
item.getProduct().getPrice();
            data[i][0] = item.getProduct().getName();
            data[i][1] = item.getQuantity();
            data[i][2] = String.format("%.2f", item.getProduct().getPrice());
            data[i][3] = String.format("%.2f", subtotal);
        }

        DefaultTableModel summaryModel = new DefaultTableModel(data, columns)
{
    public boolean isCellEditable(int row, int col) { return false; }
};

```

```

        JTable summaryTable = new JTable(summaryModel);
        summaryTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        JScrollPane scrollPane = new JScrollPane(summaryTable);

        // Total display
        double total = cart.calculateTotal();
        JLabel totalLabel = new JLabel("Total Amount: $" +
String.format("%.2f", total), SwingConstants.CENTER);
        totalLabel.setFont(totalLabel.getFont().deriveFont(Font.BOLD, 18f));
        totalLabel.setForeground(new Color(34, 139, 34));
        totalLabel.setBorder(BorderFactory.createEmptyBorder(10, 0, 10, 0));

        summaryPanel.add(totalLabel, BorderLayout.NORTH);
        summaryPanel.add(scrollPane, BorderLayout.CENTER);

        // Payment method selection
        JPanel paymentPanel = new JPanel(new BorderLayout());
        paymentPanel.setBorder(BorderFactory.createTitledBorder("Payment
Method"));

        String[] paymentMethods = {"Credit Card", "Debit Card", "PayPal",
"Cash on Delivery"};
        JComboBox<String> paymentCombo = new JComboBox<>(paymentMethods);
        paymentCombo.setSelectedIndex(0);

        JPanel paymentMethodPanel = new JPanel(new
FlowLayout(FlowLayout.LEFT));
        paymentMethodPanel.add(new JLabel("Select Payment Method:"));
        paymentMethodPanel.add(paymentCombo);

        paymentPanel.add(paymentMethodPanel, BorderLayout.CENTER);

        // Payment confirmation
        JPanel confirmPanel = new JPanel(new BorderLayout());
        confirmPanel.setBorder(BorderFactory.createTitledBorder("Payment
Confirmation"));

        JLabel confirmLabel = new JLabel("Please review your order and confirm
payment.");
        confirmLabel.setHorizontalAlignment(SwingConstants.CENTER);
        confirmLabel.setBorder(BorderFactory.createEmptyBorder(10, 0, 10, 0));

        JButton confirmBtn = new JButton("👉 Confirm Payment");
        confirmBtn.setBackground(new Color(34, 139, 34));
        confirmBtn.setForeground(Color.WHITE);
        confirmBtn.setFont(confirmBtn.getFont().deriveFont(Font.BOLD, 14f));
        confirmBtn.setFocusPainted(false);

        JButton cancelBtn = new JButton("❌ Cancel");

```

```

cancelBtn.setBackground(new Color(220, 20, 60));
cancelBtn.setForeground(Color.WHITE);
cancelBtn.setFocusPainted(false);

    JPanel btnPanel = new JPanel(new FlowLayout(FlowLayout.CENTER, 20,
5));

    btnPanel.add(cancelBtn);
    btnPanel.add(confirmBtn);

    confirmPanel.add(confirmLabel, BorderLayout.CENTER);
    confirmPanel.add(btnPanel, BorderLayout.SOUTH);

    // Main payment dialog layout
    JPanel mainPanel = new JPanel(new BorderLayout());
    mainPanel.add(summaryPanel, BorderLayout.NORTH);
    mainPanel.add(paymentPanel, BorderLayout.CENTER);
    mainPanel.add(confirmPanel, BorderLayout.SOUTH);

    JDialog paymentDialog = new JDialog(this, "Payment - " +
currentUser.getUsername(), true);
    paymentDialog.setContentPane(mainPanel);
    paymentDialog.setSize(700, 600);
    paymentDialog.setLocationRelativeTo(this);

    // Button actions
    cancelBtn.addActionListener(e -> paymentDialog.dispose());

    confirmBtn.addActionListener(e -> {
        String selectedMethod = (String) paymentCombo.getSelectedItemAt();

        // Show processing message
        JOptionPane.showMessageDialog(paymentDialog,
            "Processing payment via " + selectedMethod + "...",
            "Processing Payment",
            JOptionPane.INFORMATION_MESSAGE);

        // Simulate payment processing delay
        javax.swing.Timer timer = new javax.swing.Timer(2000, evt -> {
            // Create and save the order
            Order order = new Order(currentUser.getUsername(), cart,
total);

            orderQueue.placeOrder(order);
            orderManager.saveOrder(order);

            // Clear the cart
            this.cart = new Cart();
            updateCartButtonText();

            // Show success message

```



```

        JOptionPane.showMessageDialog(paymentDialog,
            "Payment successful! Order #" + order.getOrderid() + " has
been placed.\n\n" +
            "Thank you for your purchase!",
            "Payment Successful",
            JOptionPane.INFORMATION_MESSAGE);

        paymentDialog.dispose();
    });
    timer.setRepeats(false);
    timer.start();
});

paymentDialog.setVisible(true);
}

private void updateCartButtonText() {
    if (viewCartBtn != null) {
        viewCartBtn.setText("🛒 View Cart (" + getCartItemCount(cart) +
")");
    }
}

private void showOrderHistoryDialog() {
    if (currentUser == null) {
        JOptionPane.showMessageDialog(this, "User not logged in.",
"Error", JOptionPane.ERROR_MESSAGE);
        return;
    }

    java.util.List<Order> userOrders =
orderManager.getUserOrders(currentUser.getUsername());
    if (userOrders.isEmpty()) {
        JOptionPane.showMessageDialog(this, "No order history available.",
"Order History", JOptionPane.INFORMATION_MESSAGE);
        return;
    }

    String[] columns = {"Order ID", "Date", "Total", "Items"};
    Object[][] data = new Object[userOrders.size()][4];
    for (int i = 0; i < userOrders.size(); i++) {
        Order order = userOrders.get(i);
        data[i][0] = order.getOrderid();
        data[i][1] = order.getTimestamp();
        data[i][2] = String.format("%.2f", order.getTotal());
        data[i][3] = getCartItemCount(order.getCart());
    }
}

```

```

        DefaultTableModel historyModel = new DefaultTableModel(data, columns)
    {
        public boolean isCellEditable(int row, int col) { return false; }
    };
    JTable historyTable = new JTable(historyModel);
    JScrollPane scrollPane = new JScrollPane(historyTable);

    JButton viewDetailsBtn = new JButton("View Details");
    viewDetailsBtn.addActionListener(e -> {
        int row = historyTable.getSelectedRow();
        if (row == -1) {
            JOptionPane.showMessageDialog(this, "Select an order to view
details.");
            return;
        }
        Order selectedOrder = userOrders.get(row);
        showOrderDetailsDialog(selectedOrder);
    });

    JPanel btnPanel = new JPanel();
    btnPanel.add(viewDetailsBtn);

    JPanel panel = new JPanel(new BorderLayout());
    panel.add(scrollPane, BorderLayout.CENTER);
    panel.add(btnPanel, BorderLayout.SOUTH);

    JDialog dialog = new JDialog(this, "Order History - " +
currentUser.getUsername(), true);
    dialog.setContentPane(panel);
    dialog.setSize(600, 400);
    dialog.setLocationRelativeTo(this);
    dialog.setVisible(true);
}

private int getCartItemCount(Cart cart) {
    int count = 0;
    CartItem current = cart.getHead();
    while (current != null) {
        count += current.getQuantity();
        current = current.getNext();
    }
    return count;
}

private void showOrderDetailsDialog(Order order) {
    String[] columns = {"Product ID", "Name", "Quantity", "Price",
"Subtotal"};
    java.util.List<Object[]> dataList = new ArrayList<>();

```

```

        CartItem current = order.getCart().getHead();
        while (current != null) {
            double subtotal = current.getQuantity() *
current.getProduct().getPrice();
            dataList.add(new Object[]{
                current.getProduct().getId(),
                current.getProduct().getName(),
                current.getQuantity(),
                String.format("%.2f", current.getProduct().getPrice()),
                String.format("%.2f", subtotal)
            });
            current = current.getNext();
        }

        Object[][] data = dataList.toArray(new Object[0][]);
        DefaultTableModel detailsModel = new DefaultTableModel(data, columns)
{
    public boolean isCellEditable(int row, int col) { return false; }
};
        JTable detailsTable = new JTable(detailsModel);
        JScrollPane scrollPane = new JScrollPane(detailsTable);

        JPanel panel = new JPanel(new BorderLayout());
        panel.add(scrollPane, BorderLayout.CENTER);
        panel.add(new JLabel("Order Total: $" + String.format("%.2f",
order.getTotal()), SwingConstants.CENTER), BorderLayout.NORTH);

        JDialog dialog = new JDialog(this, "Order Details - " +
order.getOrderID(), true);
        dialog.setContentPane(panel);
        dialog.setSize(500, 300);
        dialog.setLocationRelativeTo(this);
        dialog.setVisible(true);
    }

    private void processNextOrder() {
        Order order = orderQueue.processNextOrder();
        if (order != null) {
            // Process the order
            System.out.println("Processing order:");
            System.out.println("Order ID: " + order.getTimestamp());
            System.out.println("Total: $" + String.format("%.2f",
order.getTotal()));
            CartItem current = order.getCart().getHead();
            while (current != null) {
                System.out.println(current.getProduct().getName() + " x " +
current.getQuantity());
                current = current.getNext();
            }
        }
    }

```

```
        // Order already saved by OrderManager when placed
        JOptionPane.showMessageDialog(this, "Order processed
successfully.", "Order Processed", JOptionPane.INFORMATION_MESSAGE);
    } else {
        JOptionPane.showMessageDialog(this, "No orders in queue.", "No
Orders", JOptionPane.INFORMATION_MESSAGE);
    }
}

private void logout() {
    int choice = JOptionPane.showConfirmDialog(this,
        "Are you sure you want to logout?", "Logout",
        JOptionPane.YES_NO_OPTION);

    if (choice == JOptionPane.YES_OPTION) {
        this.dispose();
        // Create new login frame
        SwingUtilities.invokeLater(() -> {
            LoginFrame loginFrame = new LoginFrame();
            loginFrame.setVisible(true);
        });
    }
}
```

## App.java

// The main entry point for the application.

```
import javax.swing.SwingUtilities;
import ui.LoginFrame;

public class App {
    public static void main(String[] args) {
        // Performance test
        model.Cart.performanceTest(100);
        model.Cart.performanceTest(150);

        // User performance test
        model.UserManager.performanceTest(100);
        model.UserManager.performanceTest(150);

        SwingUtilities.invokeLater(() -> {
            LoginFrame frame = new LoginFrame();
            frame.setVisible(true);
        });
    }
}
```

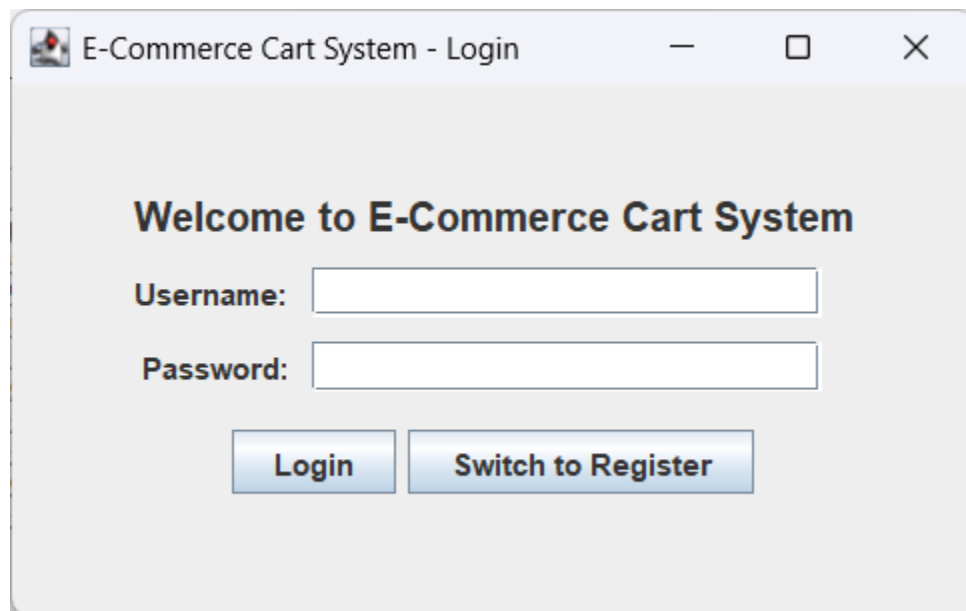
## 5. Screenshots of the Application & Output

This section presents visual evidence of the e-commerce application's functionality and its interaction with data storage.

### Application Screenshots

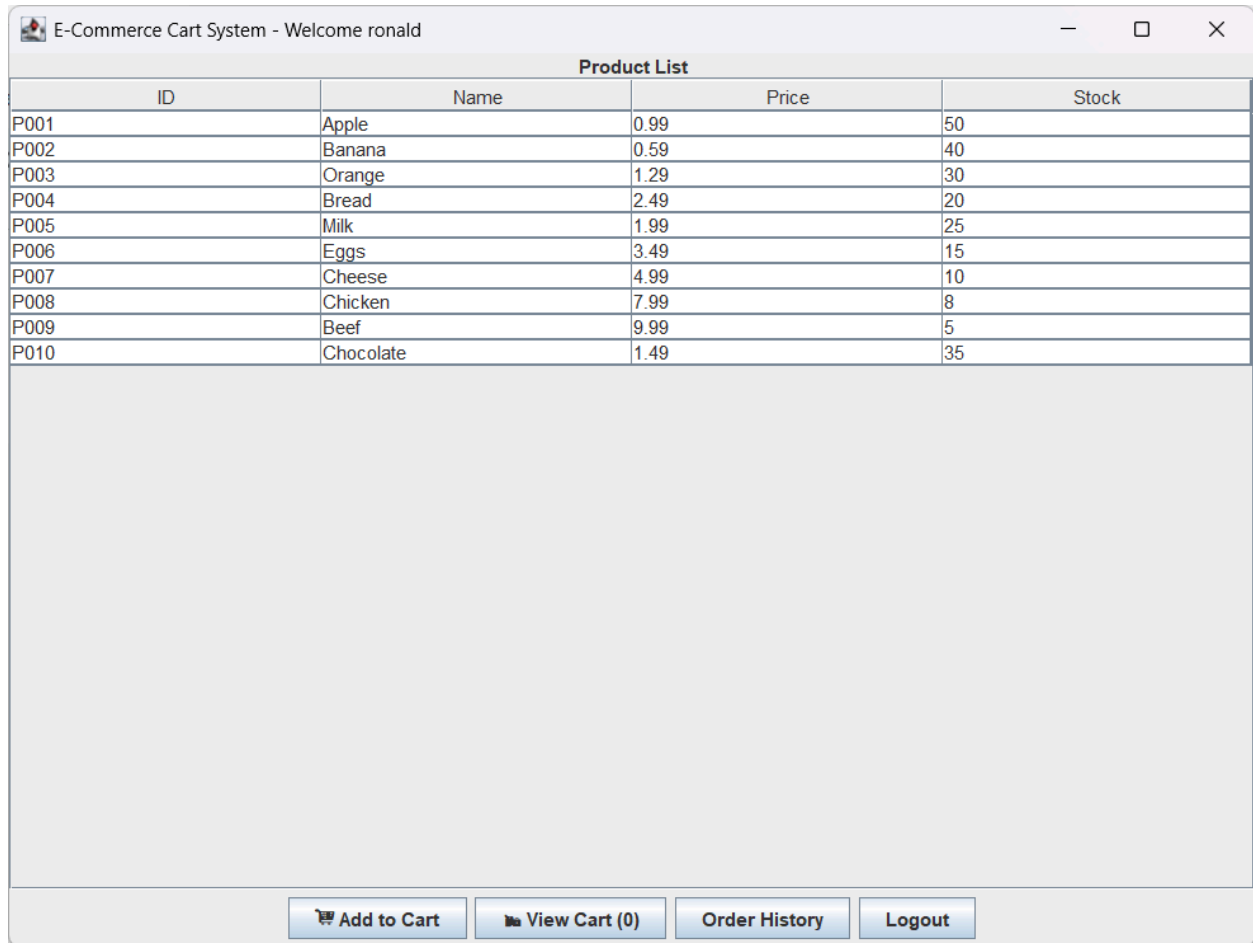
#### 5.1 Login Window

This image displays the login interface, where users must sign in to access the system or register if they do not have an account.



## 5.2 Main Window Showing Products

This image displays the primary user interface, showcasing the list of available products from which a user can select.



The screenshot shows a web application window titled "E-Commerce Cart System - Welcome ronald". The main content area displays a "Product List" table with 10 rows of products. Below the table is a large, empty light gray rectangular area. At the bottom of the window, there is a navigation bar with four buttons: "Add to Cart", "View Cart (0)", "Order History", and "Logout".

ID	Name	Price	Stock
P001	Apple	0.99	50
P002	Banana	0.59	40
P003	Orange	1.29	30
P004	Bread	2.49	20
P005	Milk	1.99	25
P006	Eggs	3.49	15
P007	Cheese	4.99	10
P008	Chicken	7.99	8
P009	Beef	9.99	5
P010	Chocolate	1.49	35


### 5.3 Shopping Cart with Items

This screenshot illustrates the shopping cart interface, displaying items that have been added by the user, along with their quantities and prices.

 Shopping Cart - ronald ×

Total: \$37.47

Product	Quantity	Unit Price	Subtotal
Bread	10	\$2.49	\$24.90
Milk	1	\$1.99	\$1.99
Beef	1	\$9.99	\$9.99
Banana	1	\$0.59	\$0.59

 Remove


 Edit Quantity

 Proceed to Payment



## 5.4 Payment Window

This image shows the final confirmation page before payment, displaying the list of ordered items and the total amount due.

 Payment - ronald ✕

**Order Summary**

**Total Amount: \$37.47**

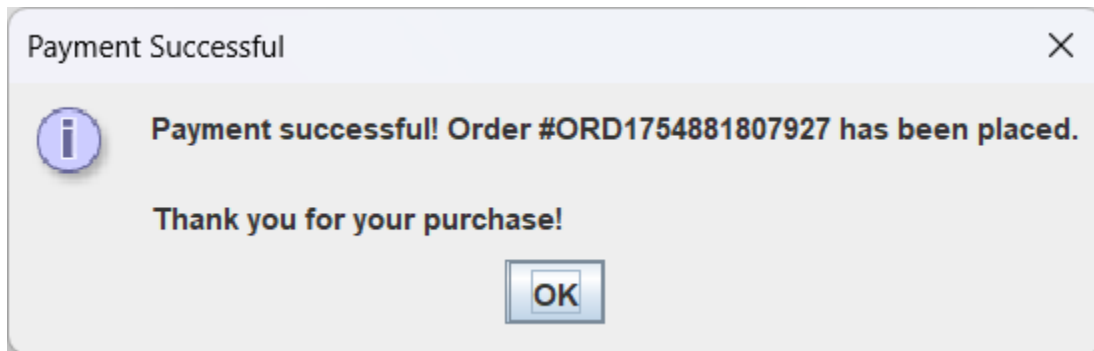
Product	Quantity	Unit Price	Subtotal
Bread	10	\$2.49	\$24.90
Milk	1	\$1.99	\$1.99
Beef	1	\$9.99	\$9.99
Banana	1	\$0.59	\$0.59

Please review your order and confirm payment.

✕ Cancel ☑ Confirm Payment

## 5.5 Order Confirmation Message

This image captures the confirmation message displayed to the user after an order has been successfully placed, including an order ID for future order tracking.



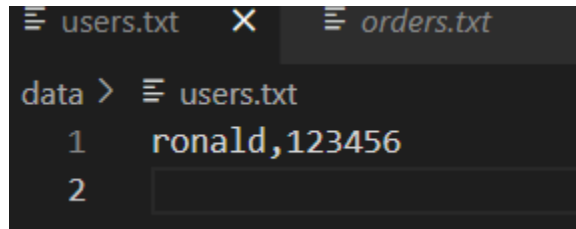
## Data Output

### 5.4 Contents of the Data Text File

This screenshot shows the raw content of the text file where order data is persisted, demonstrating the application's data storage mechanism.

#### User Data

This screenshot displays user information, including the username and password.

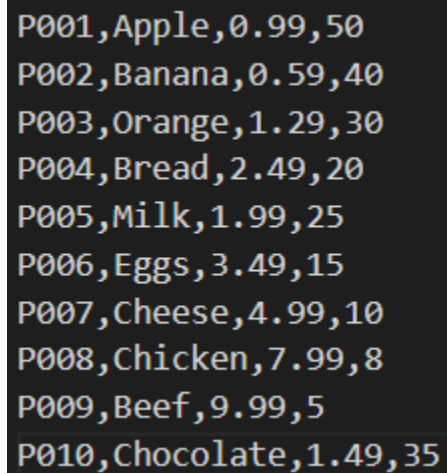


A screenshot of a text editor window with two tabs: 'users.txt' and 'orders.txt'. The 'users.txt' tab is active, showing a list of user data. The text is as follows:

```
data > users.txt
1 ronald,123456
2
```

#### Product Data

This screenshot displays product information, including the product ID, product name, price, and stock.



A screenshot of a text editor window displaying a list of product data. The text is as follows:

```
P001,Apple,0.99,50
P002,Banana,0.59,40
P003,Orange,1.29,30
P004,Bread,2.49,20
P005,Milk,1.99,25
P006,Eggs,3.49,15
P007,Cheese,4.99,10
P008,Chicken,7.99,8
P009,Beef,9.99,5
P010,Chocolate,1.49,35
```

## Order Data

This screenshot displays order information, including the order ID, username, order time, total amount paid, and the list of items purchased.

```
Order ID: ORD1754881807927
User: ronald
Order at: Mon Aug 11 11:10:07 GMT+08:00 2025, Total: $37.47
P004,Bread,10,2.49
P005,Milk,1,1.99
P009,Beef,1,9.99
P002,Banana,1,0.59
--
```

## 6. Conclusion

This project was designed to develop an efficient and maintainable **e-commerce cart system**, aligning with established best practices in software engineering. Through systematic implementation and rigorous testing, we successfully delivered a solution that meets the specified functional requirements. The performance analysis validated the theoretical time complexities, confirming that operations such as **item addition  $O(1)$**  and **item removal  $O(n)$**  consistently perform within expected bounds, specifically demonstrating an  $O(n)$  complexity for removal. These findings underscore the robustness and efficiency of the chosen data structures and algorithms (Linked List for cart, Queue for orders). Looking ahead, potential improvements include optimizing memory usage, integrating advanced error handling mechanisms, and expanding modularity to support scalability and future feature enhancements. Embracing these refinements will further align the project with industry standards and ensure sustained operational excellence.

## 7. References

- [1] GeeksforGeeks, “How to design a relational database for ecommerce Website,” GeeksforGeeks, Jul. 23, 2025.  
<https://www.geeksforgeeks.org/dbms/how-to-design-a-relational-database-for-e-commerce-website>
- [2] A. Choudhary, P. Kumar, and M. Raj, “Intelligent Shopping Cart Systems Enhance the Checkout Experience and Enable Real-Time Inventory Tracking in Retail Environments,” ResearchGate, Apr. 2025. [Online]. Available:  
[https://www.researchgate.net/publication/391190689\\_Intelligent\\_Shopping\\_Cart\\_Systems](https://www.researchgate.net/publication/391190689_Intelligent_Shopping_Cart_Systems)
- [3] Wikipedia, “Linked list,” Wikipedia, 2024. [Online]. Available:  
[https://en.wikipedia.org/wiki/Linked\\_list](https://en.wikipedia.org/wiki/Linked_list)
- [4] Wikipedia, “Queue (abstract data type),” Wikipedia, 2024. [Online]. Available:  
[https://en.wikipedia.org/wiki/Queue\\_\(abstract\\_data\\_type\)](https://en.wikipedia.org/wiki/Queue_(abstract_data_type))
- [5] M. Yuan and E. B. Fernandez, “Patterns for B2C E-Commerce Applications,” arXiv preprint, arXiv:1108.3342, Aug. 2011. [Online]. Available: <https://arxiv.org/abs/1108.3342>
- [6] GeeksforGeeks, “Linked List vs Array,” GeeksforGeeks, Jul. 23, 2025.  
<https://www.geeksforgeeks.org/dsa/linked-list-vs-array>

## 8. Turnitin Report

ORIGINALITY REPORT			
11%	4%	2%	9%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS
PRIMARY SOURCES			
1	Submitted to UCSI University Student Paper	1%	
2	Submitted to ESoft Metro Campus, Sri Lanka Student Paper	1%	
3	Submitted to Texas State University- San Marcos Student Paper	1%	
4	www.coursehero.com Internet Source	1%	
5	Submitted to American Intercontinental University Online Student Paper	1%	
6	Submitted to University of Bradford Student Paper	1%	
7	Submitted to University of Wales Institute, Cardiff Student Paper	1%	
8	Submitted to American Public University System Student Paper	1%	
9	Submitted to Pathfinder Enterprises Student Paper	<1%	
10	www.jisem-journal.com Internet Source	<1%	
11	Arvind Kumar Bansal. "Introduction to Programming Languages", Chapman and Hall/CRC, 2019 Publication	<1%	