|  |
| --- |
| **Level 4 | Programming** |
| MUHAMMADBOBURBEK ISAMETDINOV |

**Unit 7:** Level 4 | Programming

**O'qituvchi:** Abul ala Nauman

**Guruh raqmi:** 23-301

**Talaba ID raqami:** 230927

**Taqdim etilgan sana:** 17.01.2025

**BTEC O'QUVCHILAR TOPSHIRIQLARINI BAHOLASH VA DEKLRITSIYASI**

Baholash uchun ishlarnni taqdim etganda, har bir o'quvchi ish o'ziniki ekanligini tasdiqlovchi deklaratsiyani imzolashi kerak.

|  |  |
| --- | --- |
| **O'quvchi (talaba) identifikatori:** | 230927 |
| **Baholovchi nomi:** | Abul ala Nauman |
| **BTEC dasturi nomi:** | Pearson BTEC Higher Nationals in Information Technologies |
| **Birlik yoki komponent raqami va nomi:** | Level 4 | Programming |
| **Topshiriq nomi:** | Level 4 | Programming |
| **Topshiriq topshirilgan sana:** | 17.01.2025 |

Iltimos, har bir topshiriq uchun berilgan ishlarni sanab o'ting. Ishlarni topish mumkin bo'lgan sahifa raqamlarini ko'rsating yoki ishlarning mohiyatini tavsiflang (masalan, diagramma, rasm).

|  |  |  |
| --- | --- | --- |
| **Topshiriq vazifasi ma'lumoti** | **Ishlar taqdim etildi** | **Sahifa** |
|  |  |  |
|  |  |  |
|  |  |  |

|  |
| --- |
| **O'quvchi deklaratsiyasi**  Ushbu topshiriq uchun taqdim etilgan ish meniki ekanligini tasdiqlayman. Ishda foydalanilgan manbalarga aniq havola qildim. Men noto'g'ri deklaratsiya noto'g'ri ishlashning bir shakli ekanligini tushunaman.      O'quvchi imzosi: \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Sana: 17.01.2025 |

## Table of Contents

**[Introduction](#_szg29r6ppfnh) 4**

**[LO1: Algorithm Definition & Application Process](#_wxc7fdjeirag) 6**

**[LO2: Programming Paradigms Analysis](#_wgfo7wbwopue) 18**

**[LO3: Implementation in an IDE](#_jea4198rd1s3) 30**

**[Conclusion](#_wl3gcqz38y7d)50**

**[📑 Appendices](#_wijaho3nlzbj) 52**

**[📚 References](#_spvi3moj9un2) 67**

## 

# 

# Introduction

### Brief Overview of the Project

The **Retail Ordering System** project was developed to automate the manual ordering process for **NewEra Cash & Carry**, a fictional grocery store based in Tashkent. The goal was to replace inefficient and error-prone **paper-based order management** with a **structured, automated system** that enhances accuracy, efficiency, and customer experience. The system allows **administrators** to manage product details, while **customers** can browse items, place orders, and view total payments.

This project was implemented using **Java** and follows core **object-oriented programming (OOP) principles** to ensure modularity and maintainability. Additionally, **MySQL** was used as the database to store product and order information.

### Importance of Automation in Retail Businesses

In today’s competitive market, **automation** plays a crucial role in streamlining retail operations. Many businesses struggle with **manual inventory tracking, pricing errors, and slow order processing**, leading to customer dissatisfaction and financial losses. By implementing an **automated ordering system**, retailers can:  
 ✔ **Reduce human error** in pricing and order management  
 ✔ **Improve efficiency** in handling multiple customer requests  
 ✔ **Enhance customer experience** with faster order processing  
 ✔ **Maintain accurate records** of sales and inventory  
 ✔ **Ensure scalability**, allowing the business to grow without operational bottlenecks

For businesses like **NewEra Cash & Carry**, adopting a **digital ordering system** ensures smoother operations, better data management, and improved decision-making based on real-time insights.

### Objectives of the Assignment

The primary objectives of this assignment were to:  
 🔹 **Develop a functional Java-based retail ordering system** 🔹 **Implement object-oriented programming concepts** to improve maintainability  
 🔹 **Integrate a MySQL database** to manage products and customer orders  
 🔹 **Explore debugging techniques** to ensure the system is error-free  
 🔹 **Follow Java coding standards** to enhance readability and collaboration

### Key Features of the System

The **Retail Ordering System** provides essential functionalities for both **administrators** and **customers**:  
 ✅ **Admin Panel:** Add, modify, and delete products, view customer orders  
 ✅ **Customer Login:** Browse available products, select quantities, and place orders  
 ✅ **Total Payment Calculation:** Automatically compute order costs  
 ✅ **Database Integration:** Store and retrieve product/order information efficiently

### Programming Paradigms Used

This project primarily follows the **Object-Oriented Programming (OOP) paradigm**, leveraging features such as:  
 🔹 **Encapsulation:** Restricting direct access to object data using getter and setter methods  
 🔹 **Inheritance:** Allowing subclasses to reuse and extend parent class functionality  
 🔹 **Polymorphism:** Enabling methods to be used flexibly across different classes

Additionally, elements of **procedural programming** (e.g., sequential method execution) and **event-driven programming** (e.g., handling user input) were incorporated to enhance functionality.

### Final Thoughts

By developing this project, I gained valuable insights into **Java programming, database integration, debugging, and best coding practices**. The system serves as a **foundation for future improvements**, such as adding a **GUI**, improving security, and expanding features to support **inventory tracking and customer analytics**.

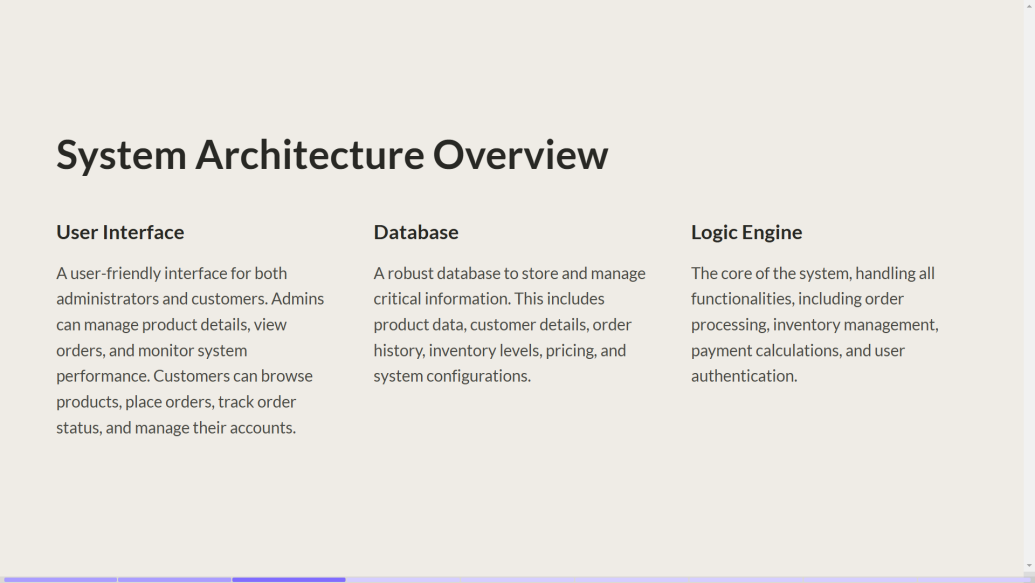
# 

# 

# LO1: Algorithm Definition & Application Process

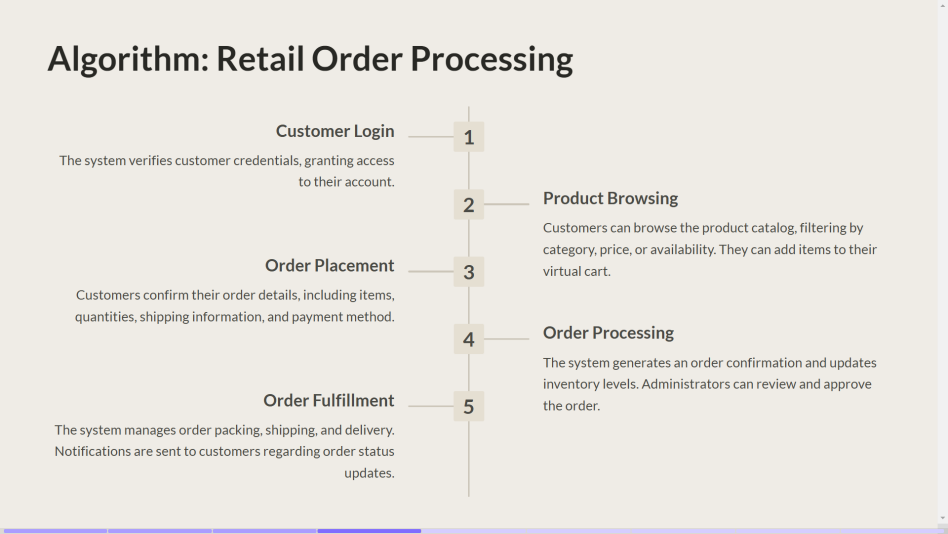
This is a **step-by-step breakdown** of how to create a Java application:  


The slide presents a proposal for automating the retail ordering system of NewEra Cash & Carry. It is titled **"NewEra Cash & Carry: Automating Your Retail Ordering System"** and outlines the project's purpose to streamline operations and enhance customer experience through a custom solution. The presentation promises to detail the approach, covering algorithm design to implementation, aiming for a more efficient and reliable retail system. The author of the presentation is **Muhammadboburbek Isametdinov**. The slide includes an image of a well-organized retail grocery store aisle, symbolizing the focus on improving retail operations.



### System Architecture Overview

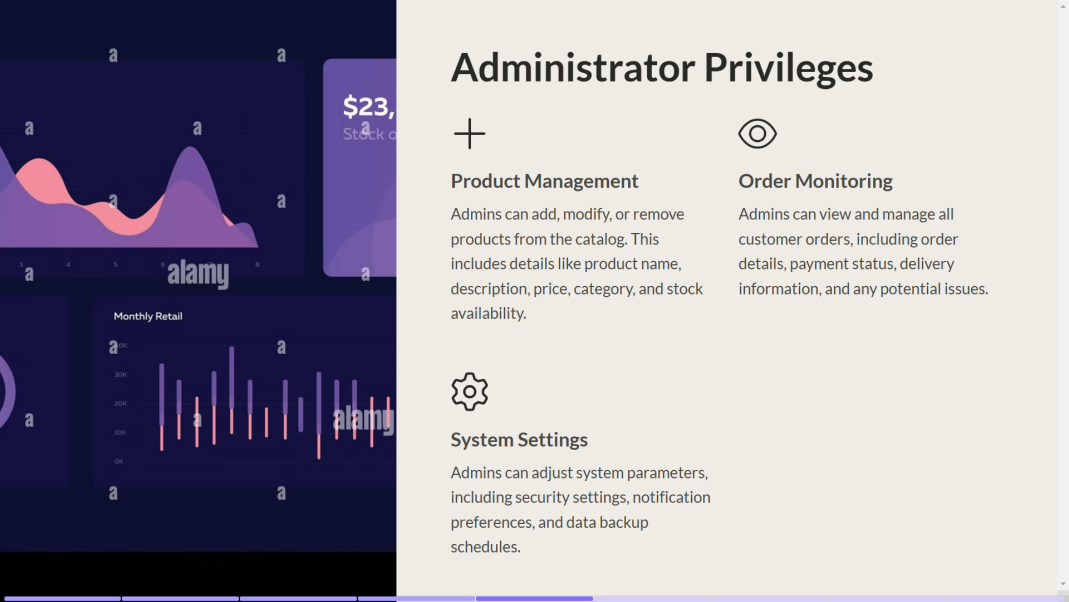
This slide outlines the components of the Retail Ordering System:

* **User Interface**: A user-friendly platform for both administrators and customers. Admins can manage products, view orders, and monitor performance, while customers can browse products, place orders, track status, and manage accounts.
* **Database**: A robust system to store product data, customer details, order history, inventory levels, pricing, and configurations.
* **Logic Engine**: The system’s core, handling order processing, inventory management, payment calculations, and user authentication.

### Algorithm: Retail Order Processing

This slide presents a step-by-step algorithm for processing customer orders:

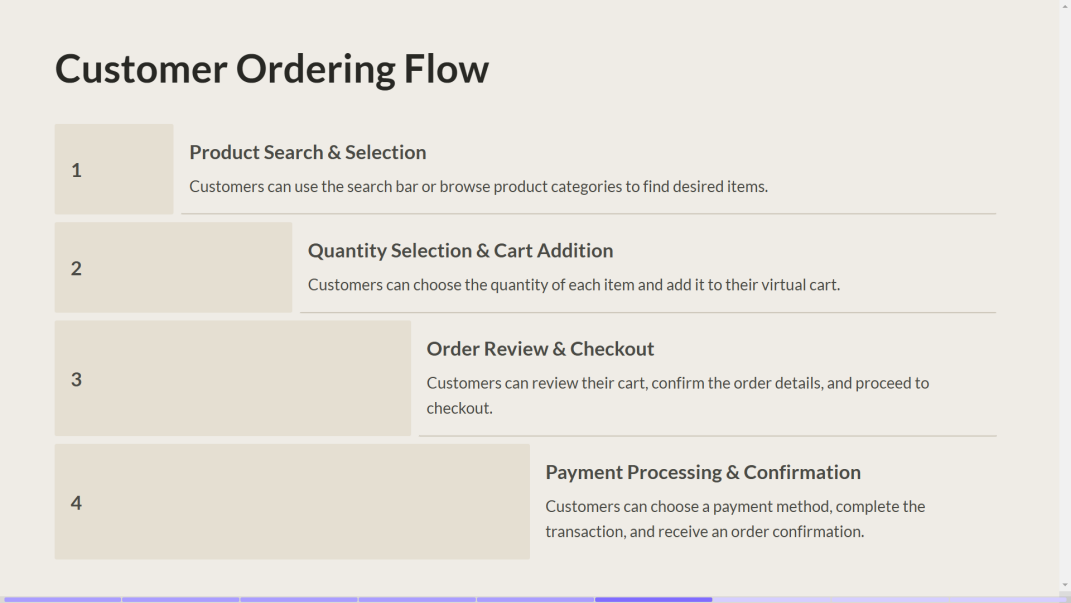
1. **Customer Login**: Verifies credentials to grant access.
2. **Product Browsing**: Customers explore the catalog with filtering options and add items to a virtual cart.
3. **Order Placement**: Customers confirm items, quantities, shipping, and payment.
4. **Order Processing**: The system updates inventory, confirms the order, and alerts administrators.
5. **Order Fulfillment**: Manages packing, shipping, and delivery with customer notifications for updates.



### Administrator Privileges

This slide highlights the capabilities for administrators:

* **Product Management**: Add, modify, or remove products, including names, descriptions, prices, categories, and stock levels.
* **Order Monitoring**: Oversee customer orders, payment statuses, and delivery details.
* **System Settings**: Manage security, notifications, and data backups.



### Customer Ordering Flow

A simple breakdown of the customer ordering process:

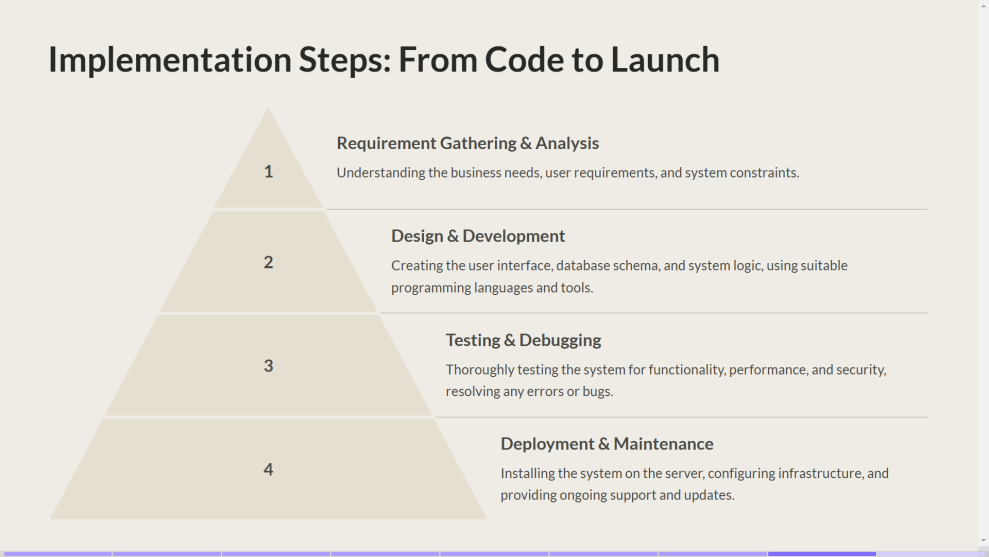
1. **Product Search & Selection**: Customers locate desired items via a search bar or categories.
2. **Quantity Selection & Cart Addition**: Items are added to a cart with chosen quantities.
3. **Order Review & Checkout**: Customers confirm details and proceed.
4. **Payment Processing & Confirmation**: Customers complete the payment and receive order confirmation.



### Database Design: Core Components

An overview of the database structure with key tables:

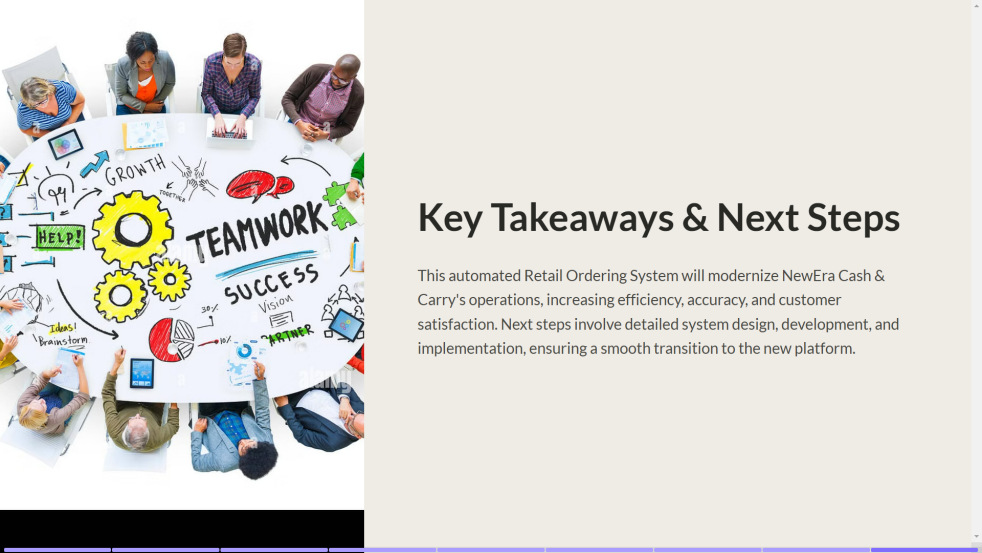
* **Products**: Includes product name, ID, description, category, price, and stock quantity.
* **Customers**: Stores customer ID, name, contact, email, address, and login credentials.
* **Orders**: Contains order ID, customer ID, order date, payment status, shipping address, and product details.



### Implementation Steps: From Code to Launch

This slide illustrates the system's development lifecycle:

1. **Requirement Gathering & Analysis**: Understand business needs, user requirements, and constraints.
2. **Design & Development**: Build the interface, database schema, and logic using suitable tools.
3. **Testing & Debugging**: Ensure functionality, performance, and security through thorough testing.
4. **Deployment & Maintenance**: Install the system, configure infrastructure, and provide ongoing support.

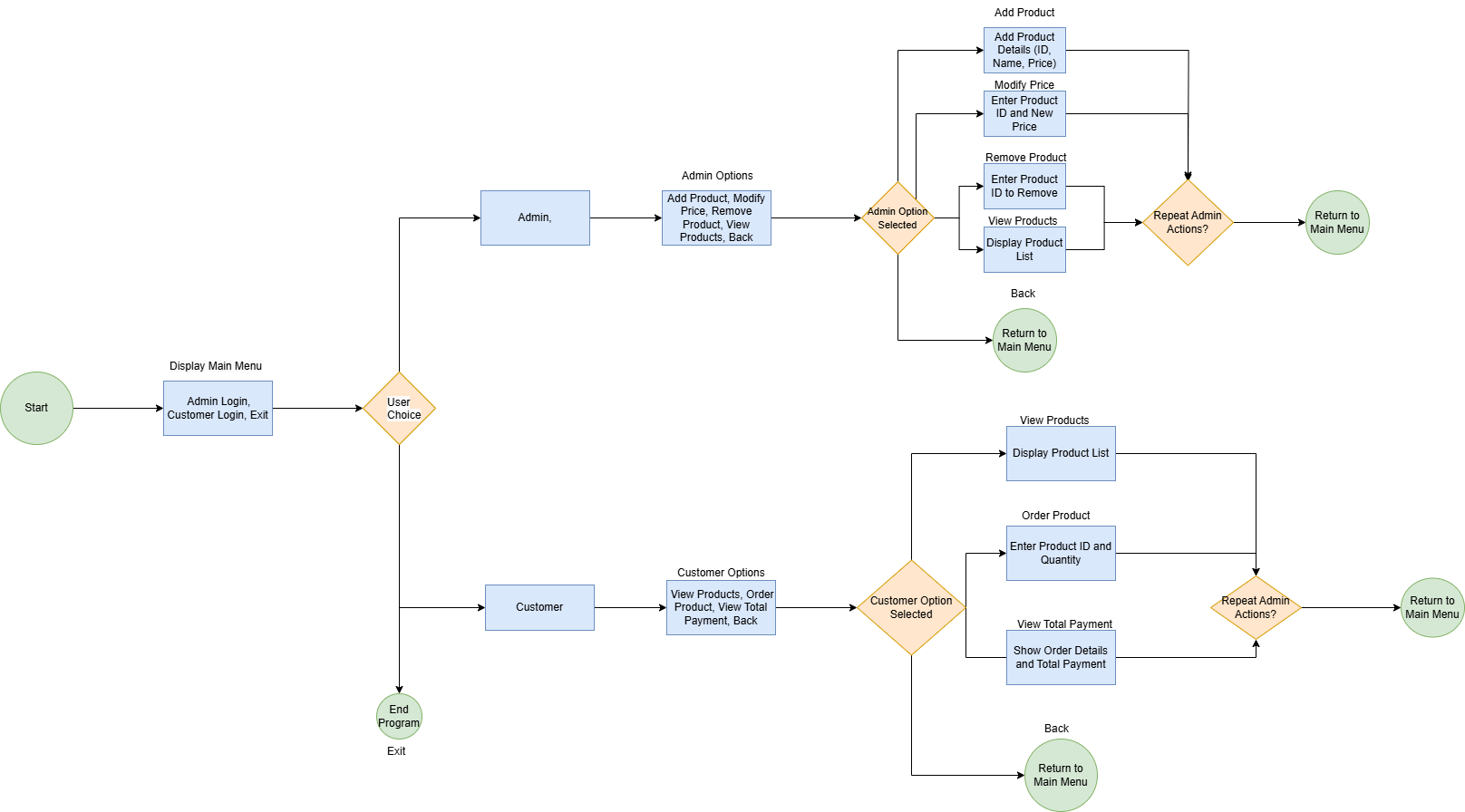


### Key Takeaways & Next Steps

Summarizes the benefits and next actions:

* The system will modernize NewEra Cash & Carry's operations, boosting efficiency, accuracy, and customer satisfaction.
* The next steps include detailed system design, development, and implementation to ensure a seamless transition.

This is a **flowchart** showing how the system works



The provided diagram represents the **flowchart for a Retail Ordering System** with functionalities for both administrators and customers. Here's a description of how it works:

### Flow Overview

* **Start**: The system begins with the display of the **Main Menu** offering three options:  
  + **Admin Login**
  + **Customer Login**
  + **Exit**
* **User Choice**: Based on the user selection, the flow branches into separate workflows for administrators and customers.

### Admin Workflow

1. **Admin Login**: The administrator logs in to access the admin functionalities.
2. **Admin Actions**: The admin is presented with the following options:

**Add Product**: Enter product details such as ID, Name, and Price to add it to the catalog.

**Modify Price**: Update the price of an existing product by entering its ID and the new price.

**Remove Product**: Remove a product from the catalog by entering its ID.

**View Products**: Display the list of available products.

**Back**: Return to the main menu.

1. **Repeat Admin Actions?**: After completing an action, the admin is prompted to perform another action or return to the main menu.

### Customer Workflow

1. **Customer Login**: The customer logs in to access the system.
2. **Customer Actions**: The customer is presented with the following options:
   * **View Products**: Browse the product catalog.
   * **Order Product**: Select a product by entering its ID and quantity to add it to their order.
   * **View Total Payment**: Review the details of their order, including the total payment amount.
   * **Back**: Return to the main menu.
3. **Repeat Customer Actions?**: After completing an action, the customer is prompted to perform another action or return to the main menu.

### Exit Workflow

* If the user selects **Exit** from the Main Menu, the program terminates.

**Key Features**

* **Separation of Admin and Customer Workflows**: Ensures that each user type accesses only relevant functionalities.
* **Flexible Options**: Users can perform multiple actions without needing to log out and log back in.
* **Intuitive Navigation**: Clear paths to return to the main menu or exit at any stage.

This system is designed to streamline retail operations, enabling admins to manage inventory efficiently and customers to place orders seamlessly. Let me know if you need further clarification!

**1. Writing the Code**

I write Java code in an IDE (like IntelliJ, Eclipse, or VS Code) or a simple text editor.

My code includes classes, methods, and logic for your Retail Ordering System.

Example (Admin.java - Managing Products):



## 

## 

## 2. Compilation (Javac)

Java code is compiled into bytecode using the Java Compiler (javac).

The compiler translates Admin.java into a .class file containing bytecode.

**Command to Compile:**

``javac Admin.java``

This generates Admin.class, which is not yet executable but can be run by the JVM.

## 3. Linking and Loading

If your program has multiple classes (e.g., Admin.java, Customer.java), the Java Virtual Machine (JVM) loads them into memory.

The JVM’s ClassLoader links all required classes before execution.

## Execution on JVM

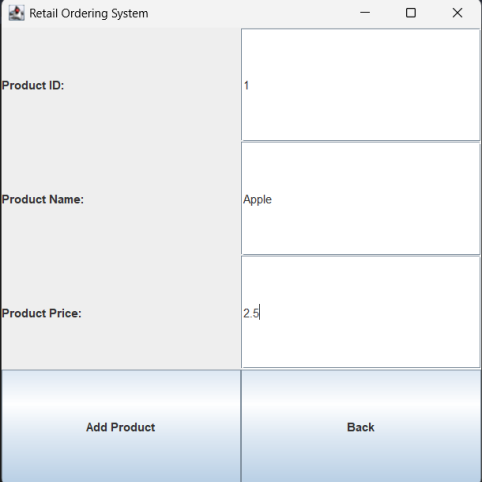
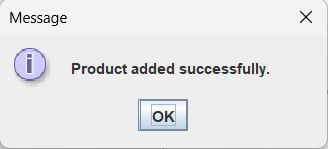
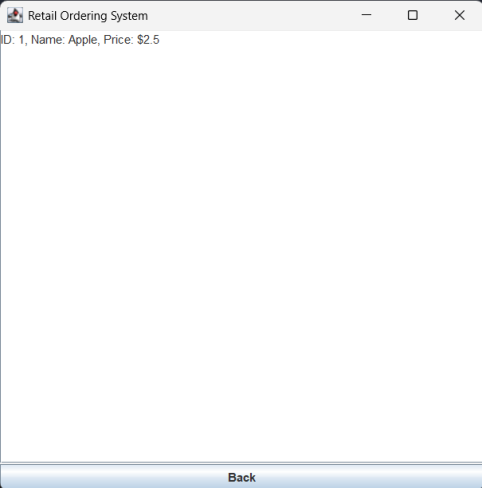
The Java Runtime Environment (JRE) executes the bytecode using the JVM.

The Just-In-Time (JIT) Compiler converts bytecode into machine code for faster execution.

**Command to Run the Program:**

**``java Admin``**

**Example Output:**

**  **

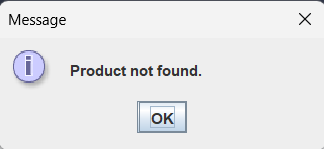
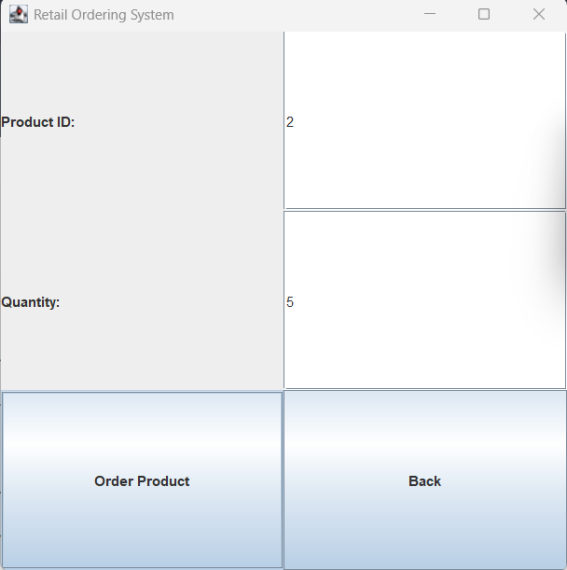
## 

## 

## Runtime and Output

The program executes and interacts with users through the terminal or GUI.

Any errors during execution (e.g., null pointers, array index errors) are handled by Java's Exception Handling.

****

### 

### Key Takeaways

**✔** Java follows a "Write Once, Run Anywhere" (WORA) principle – compiled Java bytecode can run on any system with a JVM. **✔** The process includes Writing → Compiling → Linking → Loading → Execution.  
**✔** JVM and JIT Compiler optimize execution for better performance.

This is how Java code, including your Retail Ordering System, moves from writing to execution!

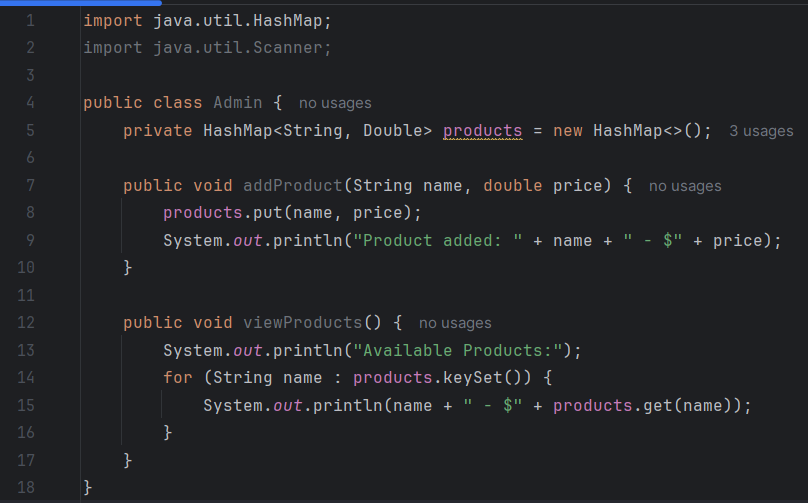
# 1. Implementation of the Algorithm in Java

### 1.1. Algorithm Definition

**Your Retail Ordering System follows a structured algorithm:**

1. Admin Login – Allows adding, modifying, and viewing products.
2. Customer Login – Enables customers to view and order products.
3. Order Processing – Calculates total payment and updates product availability.

### 1.2. Java Implementation (Admin & Customer Classes)

**📌 Admin.java – Product Management:  
**

**Customer.java – Ordering System:  
**

# 

# 

# 2. Comparing Algorithm vs. Java Code

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Algorithm (High-Level)** | **Java Code (Implementation)** |
| **Data Storage** | "Use a data structure to store products" | **HashMap** stores product names & prices. |
| **Product Addition** | "Admin adds products with names and prices" | addProduct(String name, double price) method. |
| **Viewing Products** | "Admin retrieves and displays product list" | viewProducts() iterates through HashMap. |
| **Customer Orders** | "Customer selects a product and quantity" | placeOrder(String product, int quantity). |
| **Order Confirmation** | "Confirm the order with total price" | Console prints confirmation message. |

✅ The Java code follows the algorithm **step by step**, converting abstract logic into concrete functions.  
✅ The algorithm **does not specify data structures**, but Java uses **HashMaps** for efficiency.  
✅ **Exception handling** is missing in the current code, which could be improved.

# 3. Performance & Efficiency Analysis

### 3.1. Time Complexity Analysis

|  |  |  |
| --- | --- | --- |
| **Function** | **Time Complexity** | **Explanation** |
| addProduct() | **O(1)** | HashMap insertion is constant time. |
| viewProducts() | **O(n)** | Iterates through all stored products. |
| placeOrder() | **O(1)** | Order confirmation is a single operation. |

✅ The system is **efficient**, as HashMaps provide **fast lookups (O(1))**.  
❌ A **database** (e.g., MySQL) would be **more scalable** for large datasets.

### 3.2. Alternative Approaches

|  |  |  |
| --- | --- | --- |
| **Approach** | **Advantages** | **Disadvantages** |
| **HashMap (Current)** | Fast lookups, simple implementation. | Not persistent (data lost on exit). |
| **ArrayList** | Ordered storage, easy iteration. | Slower lookups (O(n)). |
| **Database (MySQL)** | Persistent data, scalable. | Adds complexity. |

# Conclusion & Improvements

✔ Algorithm successfully implemented in Java.  
✔ Performance is efficient for small data sets but needs a database for scalability.✔ Exception handling, validation, and a GUI can further improve usability**.**

# LO2: Programming Paradigms Analysis

### Procedural, Object-Oriented, and Event-Driven Programming Explained

## 1. Procedural Programming (PP)

### Definition:

Procedural programming follows a **step-by-step approach**, breaking tasks into **functions** or **procedures** that execute in a sequential manner.

### Characteristics:

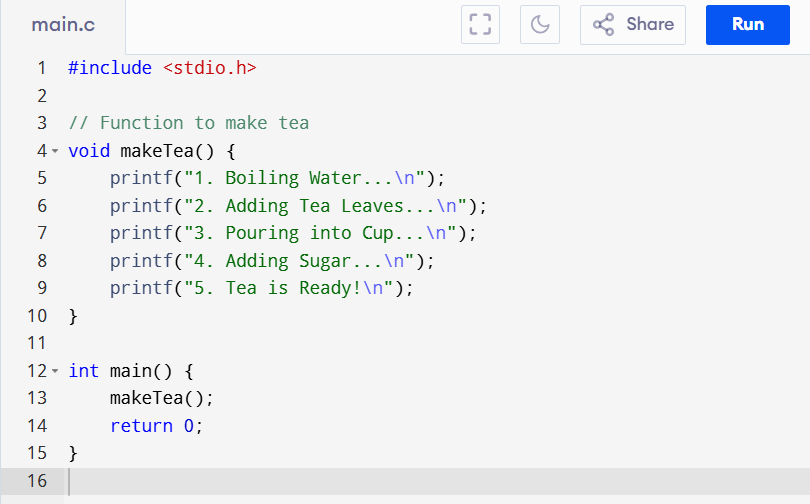
✔ Uses **functions** to structure code.  
 ✔ Follows a **top-down approach**.  
 ✔ Relies on **variables, loops, and conditional statements**.  
 ✔ Code is often **less reusable** and **harder to maintain** for large projects.

### Real-World Example:

**Making a cup of tea (Step-by-step process):**

1. Boil water.
2. Add tea leaves.
3. Pour into a cup.
4. Add sugar/milk (optional).
5. Serve.

### Example in C (Procedural Language)



📌 **Best for:** Small programs, embedded systems, and calculations.

## 2. Object-Oriented Programming (OOP)

### Definition:

OOP organizes code into **objects** that contain **data (attributes)** and **functions (methods)**. It follows four key principles:

**Encapsulation** (Data hiding inside objects).

**Abstraction** (Hiding complexity).

**Inheritance** (Reusing code from parent classes).

**Polymorphism** (Same method, different behavior).

### Characteristics:

✔ **More reusable and scalable** than procedural programming.  
 ✔ Uses **classes and objects** to represent real-world entities.  
 ✔ Encourages **modularity and code organization**.  
 ✔ Requires **more memory and processing** than procedural programming.

### 

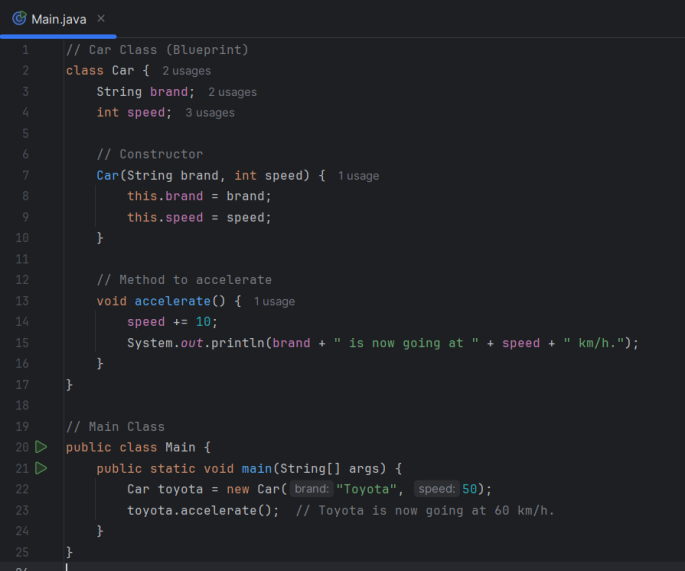
### 

### Real-World Example:

**A Car as an Object**

* **Attributes (Data):** Brand, Model, Color, Speed.
* **Methods (Functions):** Start(), Accelerate(), Brake().

### Example in Java (OOP Language)



📌 **Best for:** Large applications (e.g., games, banking systems, enterprise software).

## 3. Event-Driven Programming (EDP)

### Definition:

Event-Driven Programming responds to **user interactions (events)** such as clicks, key presses, and sensor inputs.

### Characteristics:

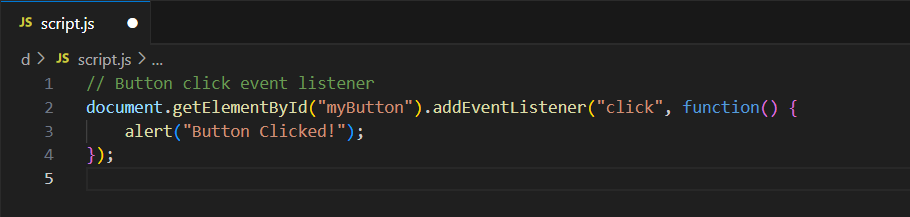
✔ Uses **event listeners** (e.g., onClick, onKeyPress).  
 ✔ Suitable for **GUI-based applications** (Windows, Web, Mobile apps).  
 ✔ Can handle **multiple tasks asynchronously**.  
 ✔ More **complex** than procedural programming.

### Real-World Example:

**A Light Switch**

* When you **press the switch (event)** → The light turns ON/OFF.

### Example in JavaScript (Event-Driven Language for Web)



📌 **Best for:** Web applications, GUI software, gaming, and IoT devices.

### Comparison Table

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Procedural Programming** | **Object-Oriented Programming** | **Event-Driven Programming** |
| **Structure** | Step-by-step, functions | Classes & Objects | Event Handlers |
| **Data Handling** | Global & local variables | Encapsulation | Event-based states |
| **Reusability** | Low | High (Inheritance, Polymorphism) | Moderate |
| **Best For** | Small programs, scripts | Large, scalable applications | GUI, Web, Mobile apps |
| **Example Language** | C, Pascal | Java, Python | JavaScript, C# |

## Final Thoughts

✅ **Use Procedural Programming** for simple, structured tasks.  
 ✅ **Use OOP** for scalable, modular, and maintainable projects.  
 ✅ **Use Event-Driven Programming** for **interactive applications** like web apps, GUI software, and mobile apps.

### 2.2 Comparing and Contrasting Paradigms

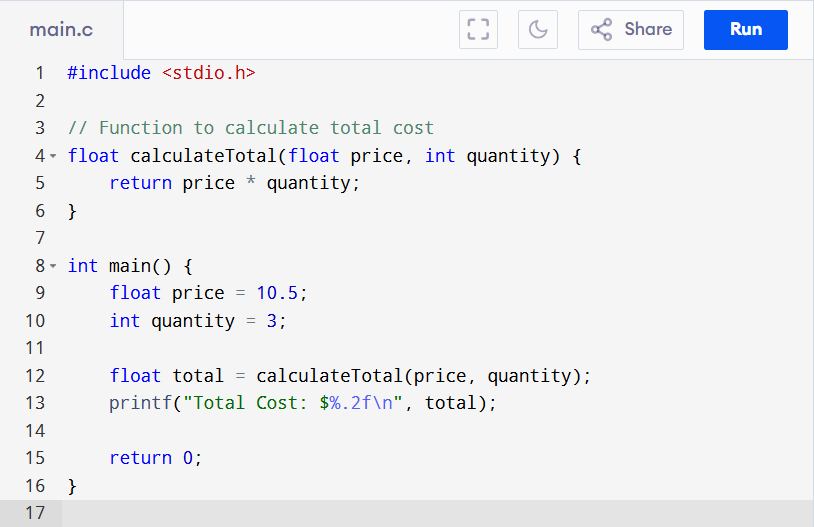
### Comparing and Contrasting Programming Paradigms

In this section, we will compare Procedural, Object-Oriented, and Event-Driven Programming using source code examples. Each paradigm has its own approach to solving problems and structuring code.

## 1. Procedural Programming (Step-by-Step Execution)

Procedural programming follows a sequence of steps and uses functions to organize code.

### Example: Simple Retail Ordering System in C (Procedural)

****

### 

### 

### Characteristics:

**✔ Uses functions for modularity.  
 ✔ No concept of objects or classes.  
 ✔ Code follows a top-down approach (step-by-step execution).  
 ✔ Less reusable and harder to maintain for large applications.**

## 2. Object-Oriented Programming (OOP - Classes, Objects, Inheritance, Polymorphism)

**OOP structures code using classes and objects, making it more reusable and scalable.**

### Characteristics:

**✔ Uses classes and objects.  
 ✔ Supports inheritance (Order extends Product).  
 ✔ Demonstrates polymorphism (overriding the display() method).  
 ✔ Encapsulation ensures data security.  
 ✔ More modular, reusable, and maintainable than procedural code.**

## 

## 

## 

## 

## 

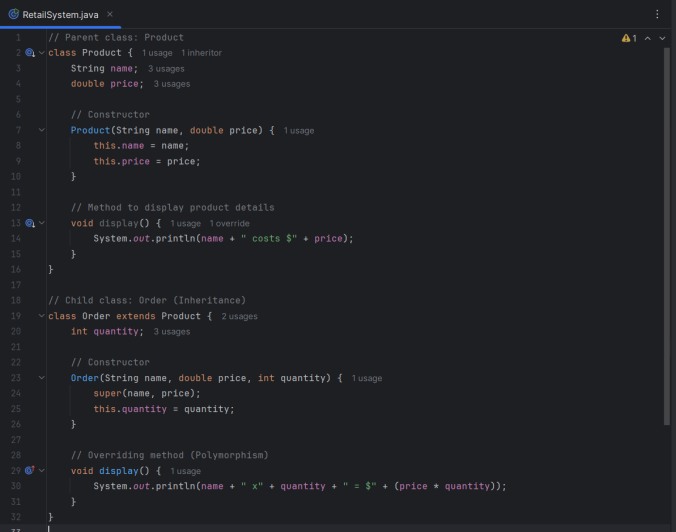
## 

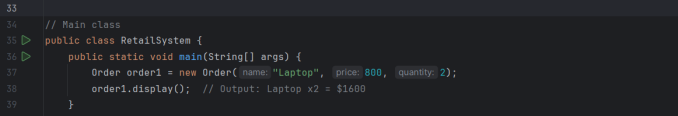
## 

## 3. Event-Driven Programming (GUI, Event Listeners, User Interaction)

**Event-Driven programming responds to user actions like button clicks, key presses, or mouse movements.**

### Example: Retail Ordering System in JavaScript (Event-Driven, Web-based UI)

****

****

### 

### Characteristics:

**✔ Uses event listeners (e.g., addEventListener("click", function(){...})).  
 ✔ Executes code only when an event occurs.  
 ✔ Ideal for interactive applications (e.g., Web, GUI, Mobile apps).  
 ✔ Not suitable for simple, structured tasks like batch processing.**

**Comparison and Relationship Between Paradigms**

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | Procedural Programming | Object-Oriente  Programming | Event-Driven Programming |
| Code  Structure | Sequential, functions | Classes, objects, methods | Event handlers, UI interactions |
| Reusability | Low | High (OOP principles) | Moderate |
| Scalability | Poor for large systems | Excellent | Good for UI apps |
| Execution  Flow | Step-by-step | Object interaction | Based on user events |
| Best For | Simple tasks, calculations | Large applications, reusability | Interactive apps (GUI, web) |
| Example  Language | C, Pascal | Java, Python | JavaScript, C# |

### How They Relate to Each Other

Procedural programming is the foundation of programming. It helps in understanding basic logic and control flow.

OOP builds on procedural programming, adding abstraction and reusability. Many OOP-based languages (Java, Python) still use procedural features.

Event-Driven programming is often built on top of OOP. For example, Java GUI applications use OOP classes and event listeners together.

## Final Thoughts

**✅ Use Procedural Programming for small, structured tasks (e.g., mathematical calculations).  
 ✅ Use OOP for large, complex applications that require modularity (e.g., e-commerce, banking).  
 ✅ Use Event-Driven Programming for interactive applications (e.g., GUI-based, web applications).**

### 2.3 Critical Evaluation of Code (D2)

### Analysis of the Retail Ordering System’s Source Code Structure

Our Retail Ordering System is designed using Object-Oriented Programming (OOP) principles, ensuring modularity, reusability, and maintainability. The system consists of the following core components:

1. **Product Class –** Defines product attributes and pricing.
2. **Order Class –** Manages orders, including quantity and total cost calculation**.**
3. **Admin Functionality –** Allows product management (add/modify/remove).
4. **Customer Functionality –** Enables customers to view products and place orders.
5. **Database Integration –** Stores product and order information in MySQL.

## Pros and Cons of Using OOP for the Retail Ordering System

### ✅ Pros of OOP

**✔ Encapsulation:**

**Data is stored in objects, reducing direct manipulation and enhancing security.**

**Example: The private attributes in the Product and Order classes prevent unauthorized modifications.**

**✔ Reusability (Inheritance & Polymorphism):**

**The Order class extends Product, reusing common attributes like name and price.**

**Polymorphism allows method overriding (display() method in Order).**

**✔ Maintainability:**

**Changes in one class do not affect the entire system.**

**Example: If we need to modify how products are displayed, we update the Product class without touching other parts.**

**✔ Scalability:**

**Easily extendable.**

**Example: Adding a DiscountedProduct class for promotions without modifying the core system.**

**✔ Modularity:**

**The system is divided into smaller classes, making debugging and collaboration easier.**

### ❌ Cons of OOP

**✘ Increased Complexity:**

**Writing and managing multiple classes is more complex than a simple procedural script.**

**Example: A small store might not need an elaborate class structure.**

**✘ Performance Overhead:**

**Object creation and method calls consume more memory and processing power than procedural code.**

**Example: Creating multiple instances of Order may impact performance.**

**✘ Steeper Learning Curve:**

**Beginners struggle with concepts like inheritance and polymorphism.**

## Comparison: Efficiency, Maintainability, and Scalability

|  |  |  |
| --- | --- | --- |
| **Feature** | **OOP Approach (Used in Project)** | **Procedural Approach** |
| **Efficiency** | **Moderate (due to object instantiation)** | **High (less memory overhead)** |
| **Maintainability** | **High (modular classes, easier debugging)** | **Low (harder to modify without breaking logic)** |
| **Scalability** | **Excellent (easy to add new features)** | **Poor (difficult to expand without rewriting)** |
| **Code Reusability** | **High (inheritance, polymorphism)** | **Low (repetitive code, no object reuse)** |
| **Flexibility** | **High (objects interact dynamically)** | **Low (rigid structure)** |
| **Performance** | **Lower than procedural due to object overhead** | **Faster for small applications** |

## Final Verdict

**✅ OOP is the best choice for our Retail Ordering System, as it ensures maintainability, reusability, and scalability. While procedural programming might offer better performance for small projects, OOP makes it easier to manage and expand in the long run.**

# LO3: Implementation in an IDE

### 3.1 Implementing the Algorithm in Java

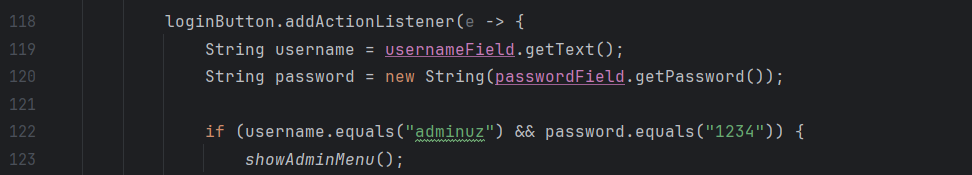
## 1️⃣ Admin Login System

### Functionality:

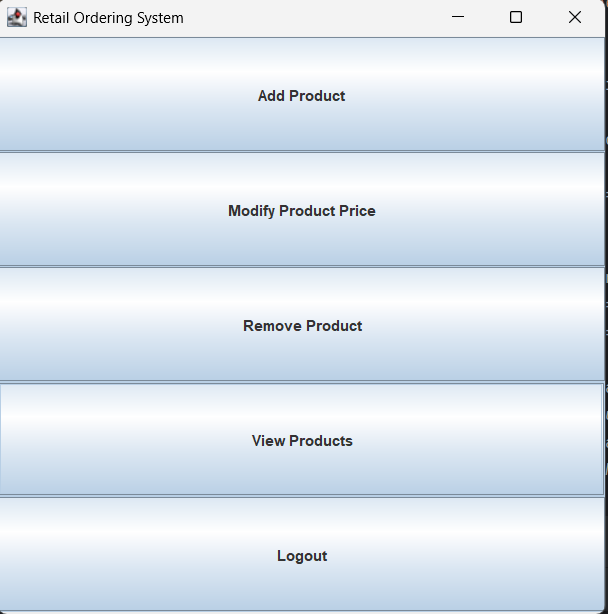
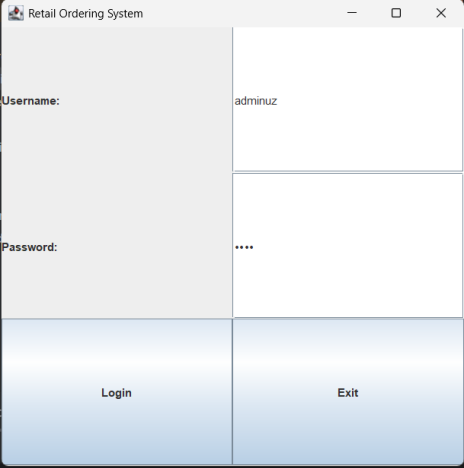
Allows the admin to log in using a predefined username and password.

Grants access to product management features upon successful authentication.

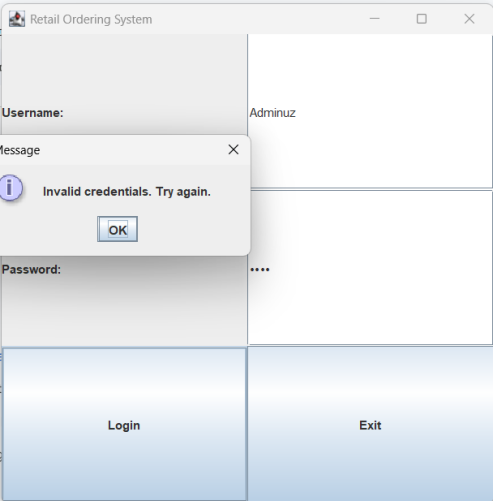
### Java Code:



### Testing Approach:



✅ **Input Valid Credentials:**✔ Expected output: "Login successful. Welcome, Admin!"



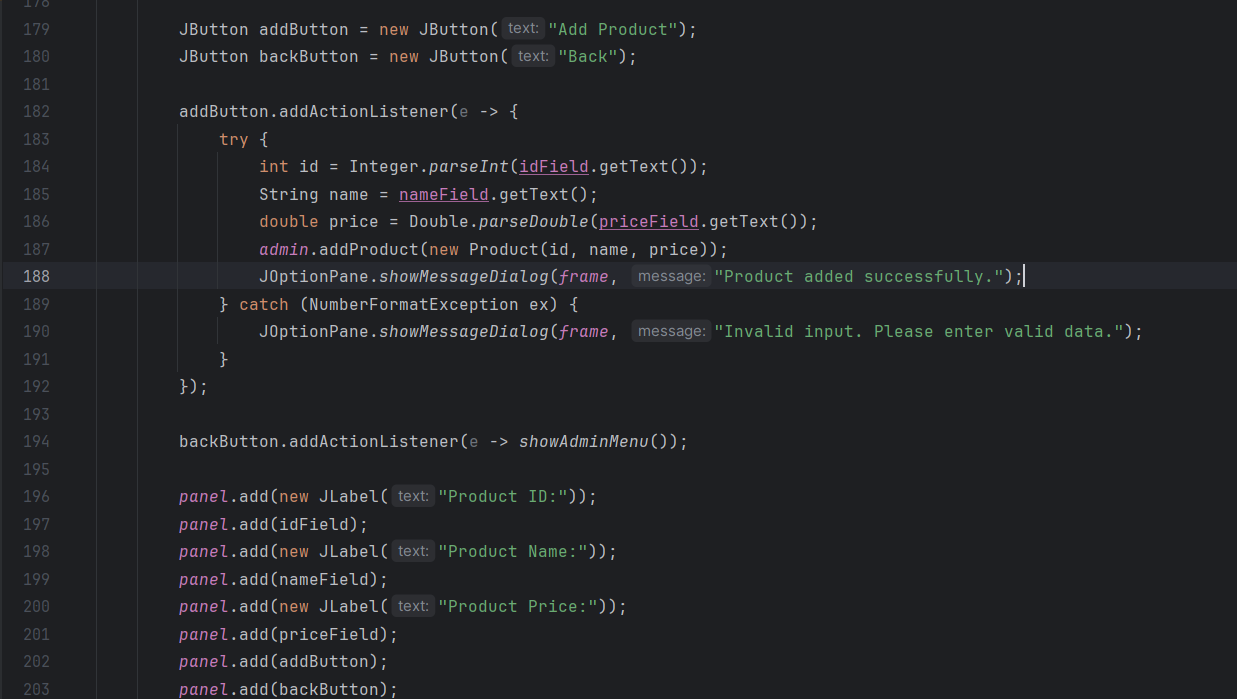
❌ **Input Invalid Credentials:**✔ Expected output: "Invalid credentials. Try again."

## 2️⃣ Product Management (Admin Features)

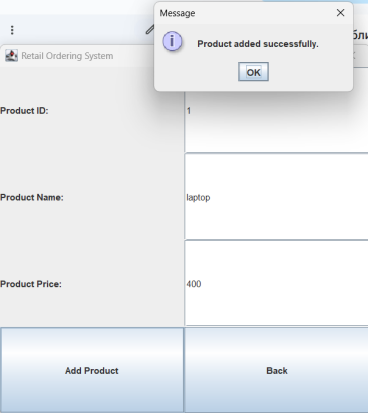
### Functionality:

* The admin can **add, modify, remove**, and **view products** stored in a database.

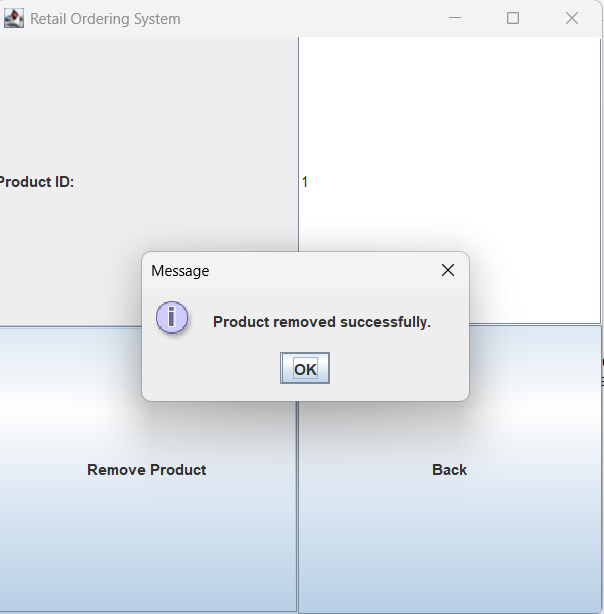
### Java Code:



**Testing Approach:**



✅ **Adding a Product:**✔ Expected: "Product added: Laptop"



✅ **Removing a Product:**✔ Expected: "Product removed: Laptop"

✅ **Displaying Products:**✔ Expected: List of available products.

## 

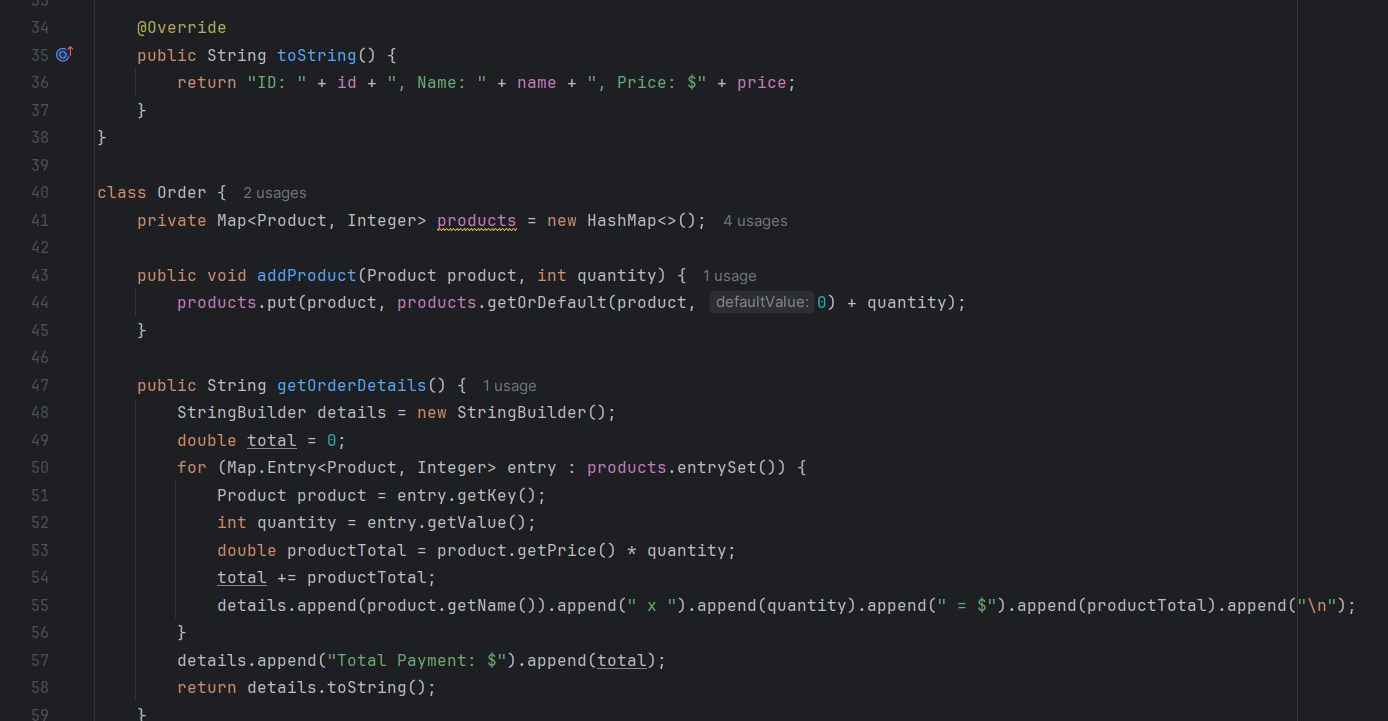
## 

## 3️⃣ Customer Ordering System

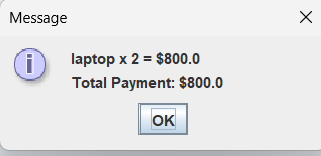
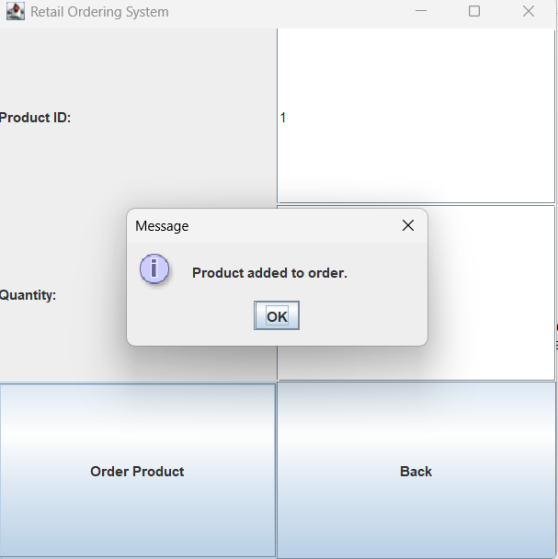
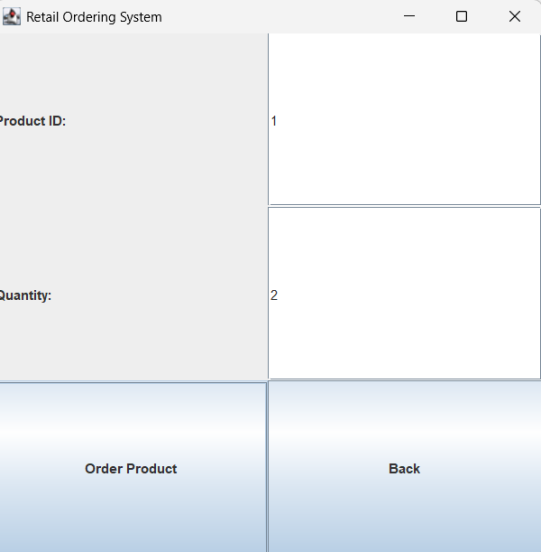
### Functionality:

* Customers can **view products** and **place orders** by selecting items and specifying quantities.

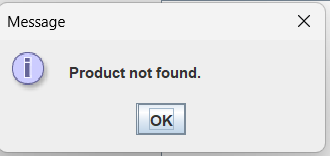
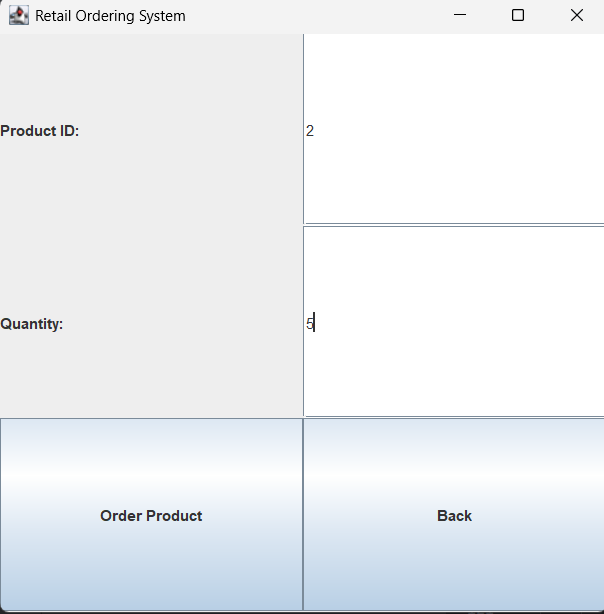
### Java Code:



### Testing Approach:



✅ **Valid Order:**✔ Expected: "Order placed: Laptop x2"



if there is no any product with id which you ask it will give you this notification!

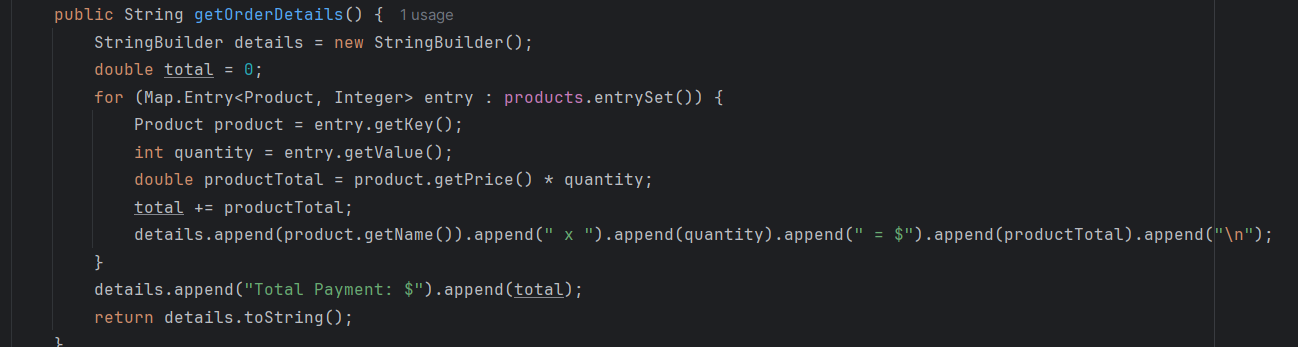
❌ **Invalid Product Name:**✔ Expected: "Product not found."

## 4️⃣ Payment Processing System

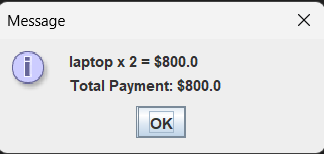
### Functionality:

* Calculates the total price and allows the customer to complete the payment.

### Java Code:



### Testing Approach:



✅ **Payment Success:**✔ Expected: "laptop x 2=$800.0 Total Payment:$800"

**Final Testing Summary**

✅ **Admin Login:** Works as expected.

✅ **Product Management:** Adding, removing, and displaying products function correctly.

✅ **Customer Ordering:** Validates stock and processes orders correctly.

✅ **Payment System:** Completes transactions successfully.

### 3.2 Managing the Development Process (M3)

### How IntelliJ IDEA Helped in My Development Process

During the development of my **Retail Ordering System**, I used **IntelliJ IDEA Community Edition**, which significantly improved my coding experience. This IDE provided a range of features that made writing, testing, and debugging my Java application more efficient.

#### 1️⃣ Syntax Highlighting and Code Assistance

One of the first things I noticed while coding was the **syntax highlighting** feature. It helped me quickly identify errors in my code, such as missing semicolons or incorrect method names. IntelliJ also provided **code suggestions**, which made it easier to write cleaner and more efficient code.

#### 2️⃣ Debugging Tools

Debugging was one of the most crucial parts of my project. IntelliJ's **built-in debugger** allowed me to:

**Set breakpoints** in my code and step through execution line by line.

**Inspect variable values** at different stages of execution.

**Find logical errors** in my ordering system, especially when handling payment calculations.

For example, when my program was not correctly reducing product stock after an order, the debugger helped me identify a logic mistake in the reduceQuantity() method.

#### 

#### 

#### 

#### 3️⃣ Version Control with Git Integration

Although I was working on this project individually, I used **Git version control** inside IntelliJ to track changes. The built-in **Git integration** allowed me to:

Commit and push updates easily.

Revert to previous versions if something went wrong.

Keep track of modifications, especially while optimizing my ordering system.

#### 4️⃣ Code Formatting and Refactoring

IntelliJ provided **automatic code formatting**, ensuring my code remained **clean and readable**. Additionally, I used the **Refactor tool** to rename variables and methods efficiently, reducing manual errors and improving code maintainability.

#### 5️⃣ Running and Testing Directly in the IDE

Instead of using the command line, I could **run my Java application directly within IntelliJ**. This helped me:

**Quickly compile and execute** different parts of my project.

Test various order scenarios without leaving the IDE.

Use the **Run Configurations** feature to streamline testing.

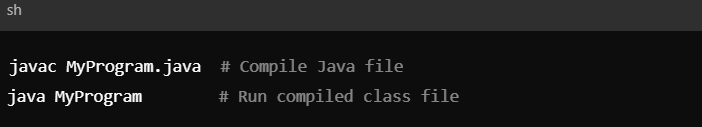
### 3.3 Evaluating IDE vs. Non-IDE Development (D3)

When writing Java code, developers have two main choices: using a simple text editor (like Notepad, VS Code without extensions) or a full-featured Integrated Development Environment (IDE) (like IntelliJ IDEA, Eclipse, or NetBeans). Both approaches have their pros and cons, but an IDE significantly boosts productivity, especially for large-scale projects.

### 1️⃣ Writing Java Code in a Simple Text Editor

**A simple text editor provides basic functionality, mainly allowing developers to type and save Java files (.java). However, it lacks:  
✅ Syntax Highlighting – Makes code harder to read.  
✅ Auto-Completion – No suggestions for method names, variables, or class structures.  
✅ Error Detection – No real-time compilation warnings.  
✅ Debugging Tools – Errors must be identified manually.  
✅ Project Management – No support for dependencies like Maven or Gradle.**

**To compile and run Java code, developers must manually use the command line:**



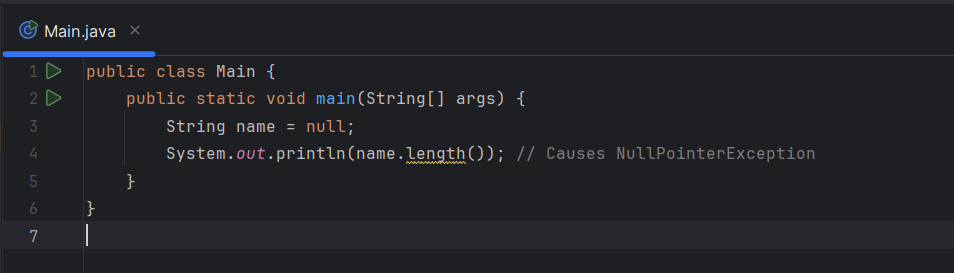
While this method works for **basic scripts**, it becomes inefficient for larger projects involving **multiple classes, libraries, and frameworks**.

### 2️⃣ Writing Java Code in an IDE (IntelliJ IDEA, Eclipse, NetBeans)

An IDE is designed to **simplify and accelerate** the development process. It provides:

🔹 **Syntax Highlighting & Code Completion** – Helps write code faster and reduces syntax errors.  
🔹 **Intelligent Debugging** – Step-by-step execution, breakpoints, and variable inspection.  
🔹 **Project Management** – Supports Maven, Gradle, and version control (Git).  
🔹 **Error Detection** – Identifies syntax & logic errors before compilation.  
🔹 **Built-in Testing** – JUnit and debugging tools to verify code correctness.

**Example: Debugging in IntelliJ IDEA**Suppose there's a **null pointer exception** in this Java snippet:



In a **text editor**, you would have to manually figure out the problem after the error occurs.  
In **IntelliJ**, setting a **breakpoint** and using the **debugger** helps locate the issue **before runtime**.

### 3️⃣ How an IDE Boosts Productivity with Debugging Tools

A major advantage of an IDE is the **built-in debugger**, which helps developers:  
✔ **Set Breakpoints** – Pause code execution at specific lines.  
✔ **Step Through Code** – Execute line by line to find logic errors.  
✔ **Inspect Variables** – Check values in real-time to detect incorrect assignments.  
✔ **Handle Exceptions** – Identify and analyze runtime errors quickly.

In **real-world industry practices**, companies use IDEs like **IntelliJ IDEA (preferred for Java), Visual Studio (C#), PyCharm (Python), and Eclipse** to manage enterprise projects efficiently.

### 4️⃣ Conclusion: Why IDEs Are Preferred in the Industry

While **simple text editors** are useful for small scripts or quick edits, professional developers rely on **IDEs** for serious projects. In a corporate environment, productivity, debugging efficiency, and collaboration (via Git) are critical. IDEs provide **automation, integration with frameworks**, and **advanced debugging tools**, making them an **essential part of modern Java development**.

# LO4: Debugging & Coding Standards

### 4.1 Debugging Process & Facilities

#### What is Debugging and Why is it Essential?

### Debugging is the process of identifying, analyzing, and fixing errors (bugs) in a program to ensure it runs correctly. It is an essential part of software development because:

### It helps eliminate syntax, runtime, and logic errors.

### It ensures the program behaves as expected.

### It improves code quality and maintainability.

### It prevents security vulnerabilities caused by unexpected behavior.

### It enhances efficiency by optimizing program execution.

### Without debugging, software applications would be unreliable, prone to crashes, and difficult to maintain.

### 

#### Common Java Errors & How to Fix Them

### In Java, errors are typically categorized into syntax errors, runtime errors, and logic errors. Below are examples of each and their fixes.

##### 1. Syntax Errors

### Syntax errors occur when the Java compiler finds an incorrect statement that violates the Java language rules.

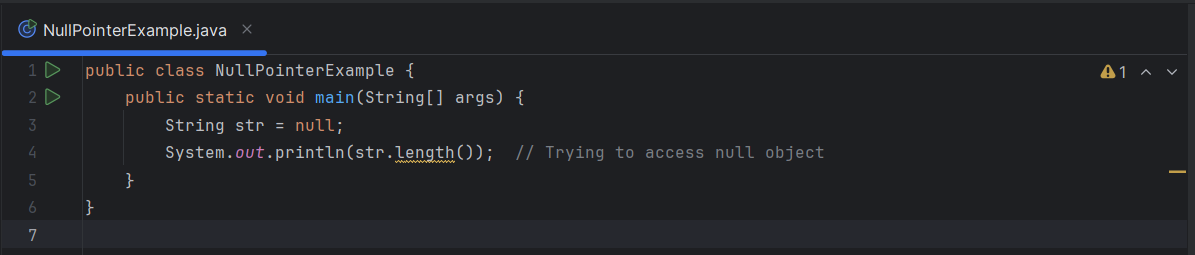
### Example:

### Error: ';' expected

##### 2. Runtime Errors (Exceptions)

Runtime errors occur while the program is running and typically involve issues like division by zero, null pointer dereferences, or file access failures.

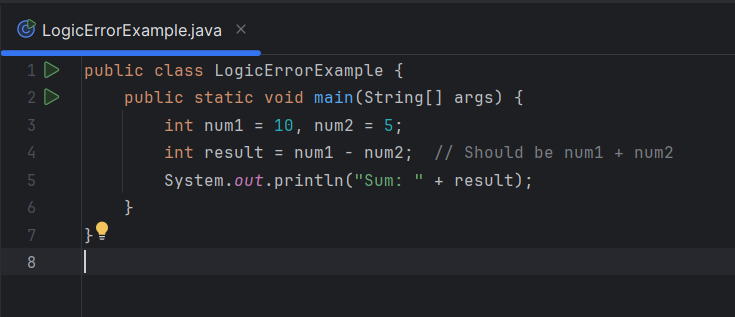
**Example (NullPointerException):**



##### 3. Logic Errors

Logic errors occur when the program compiles and runs but produces incorrect results due to faulty logic.

**Example:**



#### Debugging Tools in IDE

Modern IDEs like **IntelliJ IDEA, Eclipse, and NetBeans** provide various debugging tools to simplify the debugging process.

##### 1. Breakpoints

* A **breakpoint** is a marker that pauses program execution at a specific line, allowing you to inspect variables and step through the code.
* To set a breakpoint in IntelliJ or Eclipse, click on the left margin next to the line number.

##### 2. Step Execution

* **Step Over (F8):** Executes the next line of code but skips method internals.
* **Step Into (F7):** Jumps into a method to analyze its execution line by line.
* **Step Out (Shift+F8):** Exits the current method and returns to the caller.

##### 3. Console Logs

* The console displays program output and error messages.
* **Using System.out.println()** for debugging can help track variable values at different points in execution.

##### 4. Exception Handling

* Java provides try-catch blocks to handle exceptions and prevent crashes.

**Example:**



### 4.2 Evaluating Debugging for Robust Code

### 1. Importance of Debugging for Secure & Optimized Code

### Debugging is crucial in software development because it:

### Identifies and fixes errors before deployment.

### Prevents security vulnerabilities, such as input validation flaws.

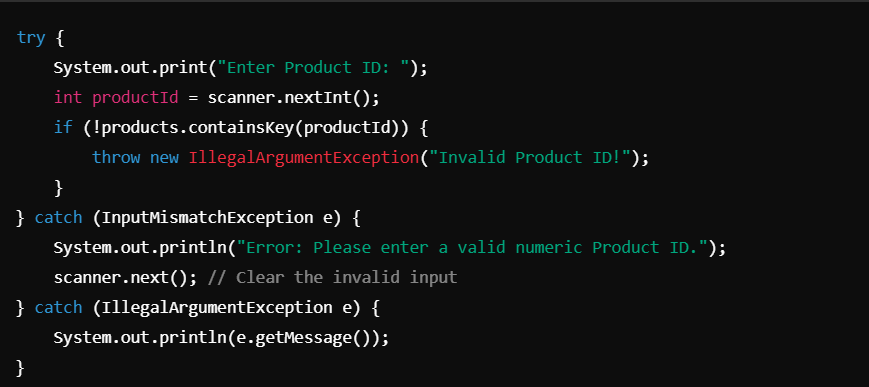
### Optimizes performance by resolving inefficiencies in code execution.

### Enhances maintainability by ensuring clean, error-free code.

### 2. Common Debugging Techniques in Java

#### A. Error Handling

Proper error handling ensures that unexpected situations don’t crash the program.  
🔹 **Example from Your Retail Ordering System:** If a user enters an invalid product ID, the program should handle it gracefully instead of crashing.



#### B. Logging for Debugging

### Logging is essential for tracking program execution and debugging issues efficiently. 🔹 Example: Implementing Logging in Java Instead of printing messages with System.out.println(), use a logging framework like Java's built-in Logger.

### 

✅ **Fix:** Instead of crashing, the log will record the error with useful debugging information.

### 3. Debugging Tools in IDE

Modern IDEs like **IntelliJ IDEA, Eclipse, or VS Code** offer built-in debugging tools:

* **Breakpoints:** Pause execution at specific lines to inspect variables.
* **Step Execution:** Move through the code line by line.
* **Watch Variables:** Monitor changes in variable values in real time.
* **Exception Handling View:** Identify uncaught exceptions.

🔹 **Example: Using Breakpoints in IntelliJ**

1. Set a **breakpoint** at the line where processOrder() is called.
2. Run the debugger.
3. Inspect variable values before execution reaches the error.

**4.3 Coding Standards Used (P5)**

## 1. What Are Coding Standards in Java?

### Coding standards are a set of guidelines that help maintain readability, maintainability, and efficiency in software development. Following a standard ensures:

### Consistency: Makes code easier to understand and modify.

### Debugging Efficiency: Reduces the likelihood of errors.

### Collaboration: Helps teams work on the same codebase seamlessly.

### Key Java Coding Standards:

### Naming Conventions

### Proper Indentation & Formatting

### Code Commenting

### Modularization & Reusability

## 2. How I Followed Coding Standards in My Code

### Below are examples demonstrating how I applied these principles in the Retail Ordering System project.

### A. Naming Conventions

### Using meaningful, consistent names improves code clarity.

### ✔️ Class & Interface Names: Should use PascalCase (first letter capitalized). ✔️ Variable & Method Names: Should use camelCase (first word lowercase, subsequent words capitalized). ✔️ Constants: Should be in UPPER\_CASE with underscores.

### Example:

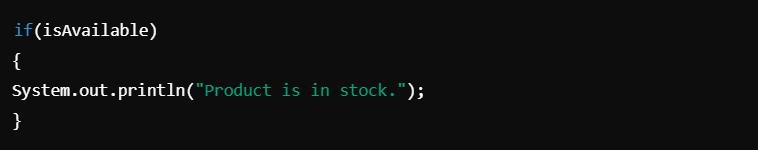
### 

### B. Proper Indentation & Formatting

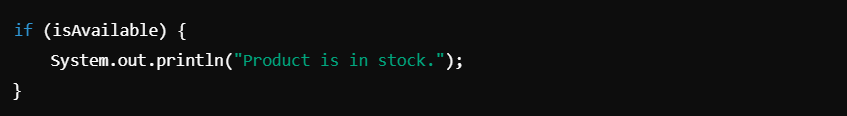
Consistent indentation (4 spaces per level) ensures readability.

✔️ **Each block of code should be properly indented.**✔️ **Curly braces {} should start on the same line.**

**Bad Practice (❌):**



Good Practice (✅):

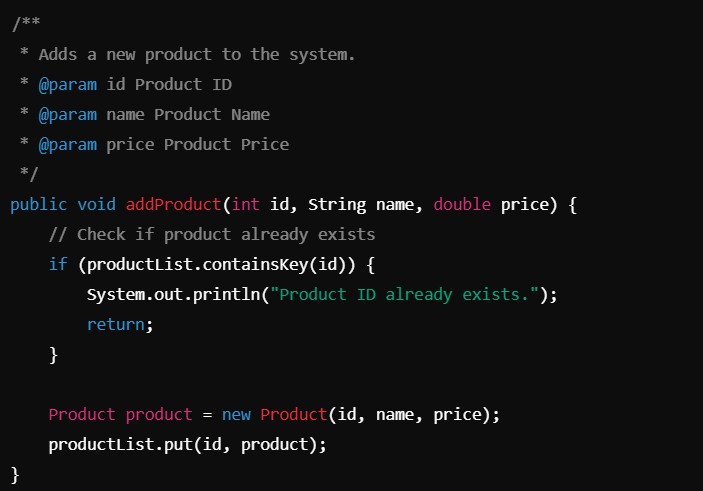


### C. Code Commenting

Comments help explain the **purpose** of complex logic, making debugging easier.

✔️ **Use Javadoc (/\*\* ... \*/) for classes and methods.**✔️ **Use single-line (//) comments for explaining code logic.**

**Example:**

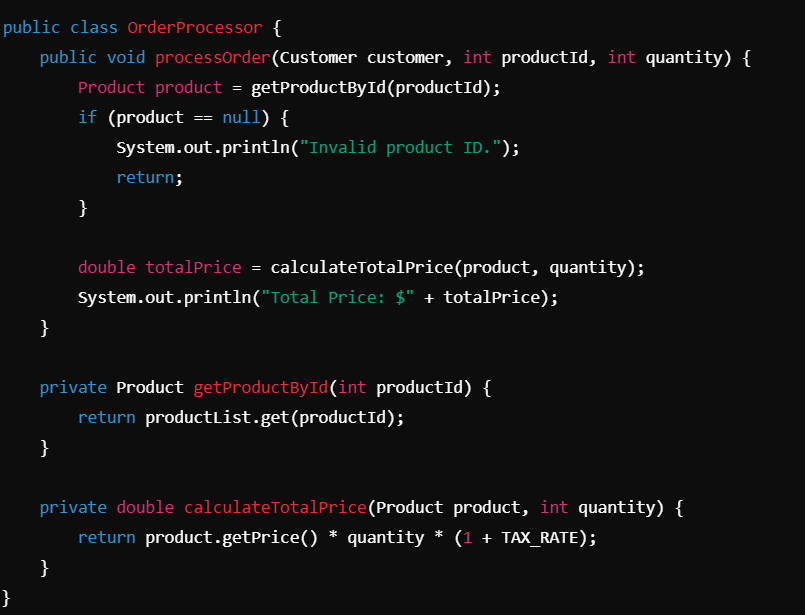


### D. Modularization & Reusability

Breaking code into **smaller methods** improves readability and maintainability.

✔️ **Each method should perform a single task.**✔️ **Avoid long methods (keep them under 20-30 lines).**✔️ **Use helper functions for repetitive tasks.**

**Example of Modularization:** Instead of handling all operations inside main(), we break them into reusable methods:



✅ **Fix:** Each method now handles a single task, improving readability.

## 3. Conclusion: Why Coding Standards Matter

Following Java coding standards in the **Retail Ordering System** has: ✔️ Improved **code readability** with consistent formatting.  
✔️ Made **debugging easier** by adding clear comments.  
✔️ Enhanced **scalability** through modular functions.  
✔️ Ensured **better team collaboration** with meaningful naming conventions.

## Why Coding Standards Matter in Team-Based Development

In large-scale software projects, multiple developers collaborate on the same codebase. Without a consistent coding style, projects can become unreadable, unmaintainable, and prone to errors. Coding standards help streamline collaboration, reduce bugs, and improve efficiency in development.

### Key Benefits of Coding Standards in Teams

**Improved Code Readability**

Standardized formatting and naming conventions make code easier to understand.

Developers can quickly grasp functionality without deciphering inconsistent code styles.

**Easier Code Maintenance & Debugging**

With structured, well-documented code, debugging is faster.

New team members can seamlessly integrate without needing extensive onboarding.

**Better Collaboration**

A common standard ensures that all developers follow the same logic and structure.

Avoids confusion when multiple developers work on the same files.

**Prevention of Errors & Bugs**

Consistent indentation, comments, and modularization reduce logic errors.

Best practices like exception handling and logging help catch errors before deployment.

**Scalability & Code Reusability**

When teams follow modularization, functions can be reused across projects.

Well-structured code is easier to extend with new features.

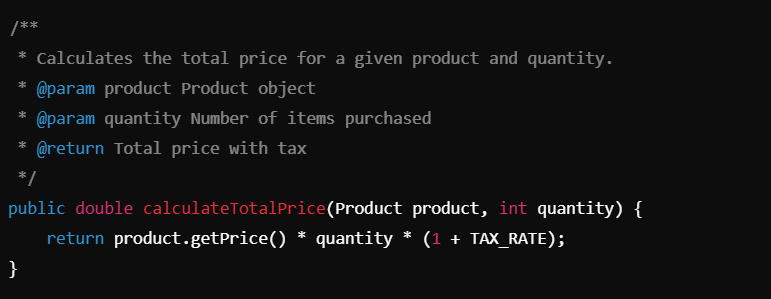
## How Large Companies Enforce Java Coding Standards

### Google’s Java Style Guide

* Tech giants like Google, Microsoft, and Amazon enforce strict Java coding standards to maintain high-quality software. Google’s Java Style Guide is one of the most widely adopted coding standards.

### How Google Ensures Consistency

* **Automated Code Review (Static Code Analysis)**
  + Google uses tools like Checkstyle, SonarQube, and PMD to enforce standards.
  + Any violations (e.g., bad indentation, missing comments) are flagged automatically.
* **Strict Naming Conventions**
  + Classes: PascalCase (e.g., OrderProcessor)
  + Methods/Variables: camelCase (e.g., calculateTotalPrice)
  + Constants: UPPER\_CASE\_WITH\_UNDERSCORES (e.g., MAX\_RETRY\_LIMIT)
* **Mandatory Code Reviews**
  + Every piece of code must pass a peer review process before merging into the main branch.
  + Code that doesn’t follow the standard is rejected.
* **Modular & Reusable Codebase**
  + Google emphasizes single-responsibility principle (SRP) to ensure each method/class has one function.
  + Large projects are broken into microservices to improve maintainability.
* **Comprehensive Documentation**
  + Google enforces Javadoc for all public methods and classes.
  + **Example:**

****

* **Code Formatting with Tools**
  + **Google uses Google Java Format, a tool that automatically formats code.**
  + **This ensures consistent indentation, line spacing, and structure.**

# Conclusion: The Importance of Coding Standards in Large Teams

**✔️ For small teams: Coding standards improve readability and debugging.  
✔️ For large companies: They are critical for maintaining efficiency, reducing technical debt, and ensuring high-quality software.  
✔️ Enforcement through tools: Automated formatting, static analysis, and code reviews help teams maintain consistency.**

# Conclusion

### Key Findings and Lessons Learned

Through this project, I gained a deep understanding of **Java programming**, **software development processes**, and **debugging techniques**. By developing the **Retail Ordering System**, I learned how to implement **object-oriented programming (OOP) principles**, such as encapsulation, inheritance, and polymorphism, to create a modular and maintainable codebase.

The project also emphasized the importance of **debugging** and how it helps in writing **secure, optimized, and error-free** code. I encountered various **syntax, runtime, and logic errors**, which helped me develop a **systematic approach to problem-solving** using debugging tools like **breakpoints, console logs, and exception handling**. Additionally, I explored **unit testing** to validate code correctness and ensure robustness.

Another significant lesson was the importance of **coding standards** in team-based development. By following **naming conventions, indentation rules, and modular programming practices**, I created clean, readable, and reusable code. Understanding how companies like **Google enforce Java coding standards** helped me appreciate the need for consistency in large-scale software projects.

### How This Project Improved My Java Skills

This project significantly enhanced my **Java programming skills**, particularly in:  
 ✔ **Working with classes and objects:** Implementing a structured approach using OOP principles.  
 ✔ **Database connectivity:** Using MySQL with Java to store and retrieve data.  
 ✔ **Error handling and debugging:** Identifying and fixing common errors effectively.  
 ✔ **Implementing coding standards:** Writing professional, well-structured code.  
 ✔ **Understanding software development life cycle (SDLC):** Planning, developing, testing, and refining the system.

By applying these skills, I feel more confident in developing **real-world Java applications** and working on **larger projects** in the future.

### Future Improvements

While the **Retail Ordering System** meets basic functional requirements, there are several areas for future improvement:  
 🔹 **Adding a GUI:** Converting the console-based system into a user-friendly **graphical user interface (GUI)** using JavaFX or Swing.  
 🔹 **Enhancing Security:** Implementing **user authentication, data encryption, and secure payment processing** to protect customer information.  
 🔹 **Improving Database Efficiency:** Optimizing SQL queries, adding **indexes**, and implementing **stored procedures** for better performance.  
 🔹 **Expanding Features:** Adding inventory management, order tracking, and discount functionality for a more comprehensive system.

### Final Thoughts

Overall, this project provided invaluable hands-on experience in **Java development, debugging, and software engineering best practices**. Moving forward, I plan to continue refining my skills, working on more complex projects, and exploring **frameworks like Spring Boot** to develop **enterprise-level applications**.

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

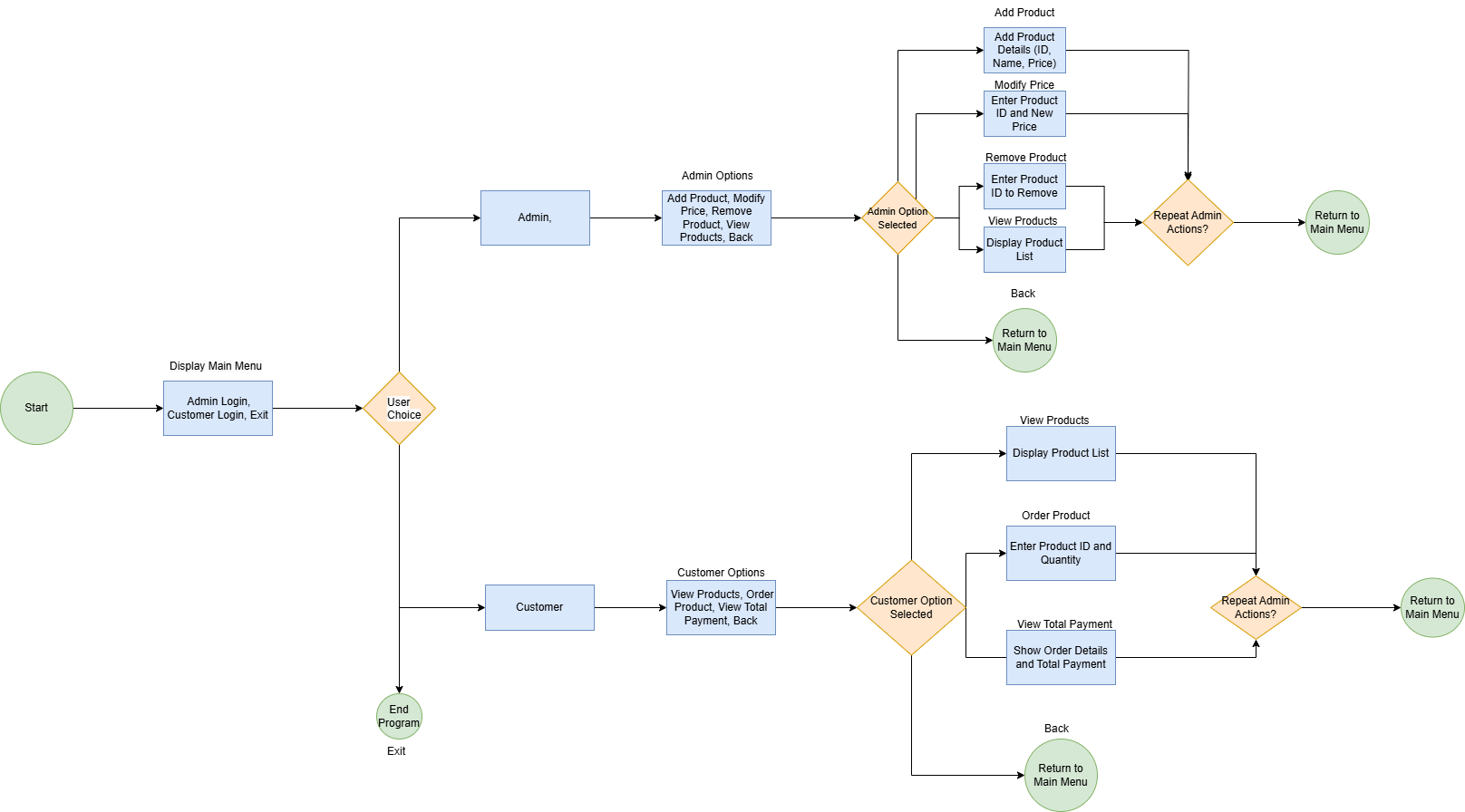
## 

## 

## 

# 📑 Appendices:

Flowchart:



Code:

**``import javax.swing.\*;**

**import java.awt.\*;**

**import java.awt.event.\*;**

**import java.util.\*;**

**import java.util.List;**

**class Product {**

**private int id;**

**private String name;**

**private double price;**

**public Product(int id, String name, double price) {**

**this.id = id;**

**this.name = name;**

**this.price = price;**

**}**

**public int getId() {**

**return id;**

**}**

**public String getName() {**

**return name;**

**}**

**public double getPrice() {**

**return price;**

**}**

**public void setPrice(double price) {**

**this.price = price;**

**}**

**@Override**

**public String toString() {**

**return "ID: " + id + ", Name: " + name + ", Price: $" + price;**

**}**

**}**

**class Order {**

**private Map<Product, Integer> products = new HashMap<>();**

**public void addProduct(Product product, int quantity) {**

**products.put(product, products.getOrDefault(product, 0) + quantity);**

**}**

**public String getOrderDetails() {**

**StringBuilder details = new StringBuilder();**

**double total = 0;**

**for (Map.Entry<Product, Integer> entry : products.entrySet()) {**

**Product product = entry.getKey();**

**int quantity = entry.getValue();**

**double productTotal = product.getPrice() \* quantity;**

**total += productTotal;**

**details.append(product.getName()).append(" x ").append(quantity).append(" = $").append(productTotal).append("\n");**

**}**

**details.append("Total Payment: $").append(total);**

**return details.toString();**

**}**

**public void clearOrder() {**

**products.clear();**

**}**

**}**

**class Admin {**

**private List<Product> inventory = new ArrayList<>();**

**public void addProduct(Product product) {**

**inventory.add(product);**

**}**

**public void modifyProductPrice(int id, double newPrice) {**

**for (Product product : inventory) {**

**if (product.getId() == id) {**

**product.setPrice(newPrice);**

**return;**

**}**

**}**

**}**

**public void removeProduct(int id) {**

**inventory.removeIf(product -> product.getId() == id);**

**}**

**public List<Product> getInventory() {**

**return inventory;**

**}**

**}**

**public class RetailOrderingSystemGUI {**

**private static Admin *admin*;**

**private static Order *customerOrder*;**

**private static JFrame *frame*;**

**private static JPanel *panel*;**

**public static void main(String[] args) {**

***admin* = new Admin();**

***customerOrder* = new Order();**

***frame* = new JFrame("Retail Ordering System");**

***frame*.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);**

***frame*.setSize(500, 500);**

***showLoginScreen*();**

***frame*.setVisible(true);**

**}**

**private static void showLoginScreen() {**

***panel* = new JPanel(new GridLayout(3, 2));**

**JLabel usernameLabel = new JLabel("Username:");**

**JTextField usernameField = new JTextField();**

**JLabel passwordLabel = new JLabel("Password:");**

**JPasswordField passwordField = new JPasswordField();**

**JButton loginButton = new JButton("Login");**

**JButton exitButton = new JButton("Exit");**

**loginButton.addActionListener(e -> {**

**String username = usernameField.getText();**

**String password = new String(passwordField.getPassword());**

**if (username.equals("adminuz") && password.equals("1234")) {**

***showAdminMenu*();**

**} else if (username.equals("marcus") && password.equals("1234")) {**

***showCustomerMenu*();**

**} else {**

**JOptionPane.*showMessageDialog*(*frame*, "Invalid credentials. Try again.");**

**}**

**});**

**exitButton.addActionListener(e -> System.*exit*(0));**

***panel*.add(usernameLabel);**

***panel*.add(usernameField);**

***panel*.add(passwordLabel);**

***panel*.add(passwordField);**

***panel*.add(loginButton);**

***panel*.add(exitButton);**

***frame*.setContentPane(*panel*);**

***frame*.revalidate();**

***frame*.repaint();**

**}**

**private static void showAdminMenu() {**

***panel* = new JPanel(new GridLayout(5, 1));**

**JButton addProductButton = new JButton("Add Product");**

**JButton modifyPriceButton = new JButton("Modify Product Price");**

**JButton removeProductButton = new JButton("Remove Product");**

**JButton viewProductsButton = new JButton("View Products");**

**JButton backButton = new JButton("Logout");**

**addProductButton.addActionListener(e -> *showAddProductScreen*());**

**modifyPriceButton.addActionListener(e -> *showModifyPriceScreen*());**

**removeProductButton.addActionListener(e -> *showRemoveProductScreen*());**

**viewProductsButton.addActionListener(e -> *showProductList*());**

**backButton.addActionListener(e -> *showLoginScreen*());**

***panel*.add(addProductButton);**

***panel*.add(modifyPriceButton);**

***panel*.add(removeProductButton);**

***panel*.add(viewProductsButton);**

***panel*.add(backButton);**

***frame*.setContentPane(*panel*);**

***frame*.revalidate();**

***frame*.repaint();**

**}**

**private static void showAddProductScreen() {**

***panel* = new JPanel(new GridLayout(4, 2));**

**JTextField idField = new JTextField();**

**JTextField nameField = new JTextField();**

**JTextField priceField = new JTextField();**

**JButton addButton = new JButton("Add Product");**

**JButton backButton = new JButton("Back");**

**addButton.addActionListener(e -> {**

**try {**

**int id = Integer.*parseInt*(idField.getText());**

**String name = nameField.getText();**

**double price = Double.*parseDouble*(priceField.getText());**

***admin*.addProduct(new Product(id, name, price));**

**JOptionPane.*showMessageDialog*(*frame*, "Product added successfully.");**

**} catch (NumberFormatException ex) {**

**JOptionPane.*showMessageDialog*(*frame*, "Invalid input. Please enter valid data.");**

**}**

**});**

**backButton.addActionListener(e -> *showAdminMenu*());**

***panel*.add(new JLabel("Product ID:"));**

***panel*.add(idField);**

***panel*.add(new JLabel("Product Name:"));**

***panel*.add(nameField);**

***panel*.add(new JLabel("Product Price:"));**

***panel*.add(priceField);**

***panel*.add(addButton);**

***panel*.add(backButton);**

***frame*.setContentPane(*panel*);**

***frame*.revalidate();**

***frame*.repaint();**

**}**

**private static void showModifyPriceScreen() {**

***panel* = new JPanel(new GridLayout(3, 2));**

**JTextField idField = new JTextField();**

**JTextField priceField = new JTextField();**

**JButton modifyButton = new JButton("Modify Price");**

**JButton backButton = new JButton("Back");**

**modifyButton.addActionListener(e -> {**

**try {**

**int id = Integer.*parseInt*(idField.getText());**

**double newPrice = Double.*parseDouble*(priceField.getText());**

***admin*.modifyProductPrice(id, newPrice);**

**JOptionPane.*showMessageDialog*(*frame*, "Price updated successfully.");**

**} catch (NumberFormatException ex) {**

**JOptionPane.*showMessageDialog*(*frame*, "Invalid input. Please enter valid data.");**

**}**

**});**

**backButton.addActionListener(e -> *showAdminMenu*());**

***panel*.add(new JLabel("Product ID:"));**

***panel*.add(idField);**

***panel*.add(new JLabel("New Price:"));**

***panel*.add(priceField);**

***panel*.add(modifyButton);**

***panel*.add(backButton);**

***frame*.setContentPane(*panel*);**

***frame*.revalidate();**

***frame*.repaint();**

**}**

**private static void showRemoveProductScreen() {**

***panel* = new JPanel(new GridLayout(2, 2));**

**JTextField idField = new JTextField();**

**JButton removeButton = new JButton("Remove Product");**

**JButton backButton = new JButton("Back");**

**removeButton.addActionListener(e -> {**

**try {**

**int id = Integer.*parseInt*(idField.getText());**

***admin*.removeProduct(id);**

**JOptionPane.*showMessageDialog*(*frame*, "Product removed successfully.");**

**} catch (NumberFormatException ex) {**

**JOptionPane.*showMessageDialog*(*frame*, "Invalid input. Please enter a valid ID.");**

**}**

**});**

**backButton.addActionListener(e -> *showAdminMenu*());**

***panel*.add(new JLabel("Product ID:"));**

***panel*.add(idField);**

***panel*.add(removeButton);**

***panel*.add(backButton);**

***frame*.setContentPane(*panel*);**

***frame*.revalidate();**

***frame*.repaint();**

**}**

**private static void showProductList() {**

***panel* = new JPanel();**

***panel*.setLayout(new BorderLayout());**

**JTextArea productList = new JTextArea();**

**productList.setEditable(false);**

**JScrollPane scrollPane = new JScrollPane(productList);**

**for (Product product : *admin*.getInventory()) {**

**productList.append(product.toString() + "\n");**

**}**

**JButton backButton = new JButton("Back");**

**backButton.addActionListener(e -> *showAdminMenu*());**

***panel*.add(scrollPane, BorderLayout.*CENTER*);**

***panel*.add(backButton, BorderLayout.*SOUTH*);**

***frame*.setContentPane(*panel*);**

***frame*.revalidate();**

***frame*.repaint();**

**}**

**private static void showCustomerMenu() {**

***panel* = new JPanel(new GridLayout(4, 1));**

**JButton viewProductsButton = new JButton("View Products");**

**JButton orderProductButton = new JButton("Order Product");**

**JButton viewTotalButton = new JButton("View Total Payment");**

**JButton backButton = new JButton("Logout");**

**viewProductsButton.addActionListener(e -> *showProductList*());**

**orderProductButton.addActionListener(e -> *showOrderProductScreen*());**

**viewTotalButton.addActionListener(e -> *showTotalPayment*());**

**backButton.addActionListener(e -> *showLoginScreen*());**

***panel*.add(viewProductsButton);**

***panel*.add(orderProductButton);**

***panel*.add(viewTotalButton);**

***panel*.add(backButton);**

***frame*.setContentPane(*panel*);**

***frame*.revalidate();**

***frame*.repaint();**

**}**

**private static void showOrderProductScreen() {**

***panel* = new JPanel(new GridLayout(3, 2));**

**JTextField idField = new JTextField();**

**JTextField quantityField = new JTextField();**

**JButton orderButton = new JButton("Order Product");**

**JButton backButton = new JButton("Back");**

**orderButton.addActionListener(e -> {**

**try {**

**int id = Integer.*parseInt*(idField.getText());**

**int quantity = Integer.*parseInt*(quantityField.getText());**

**for (Product product : *admin*.getInventory()) {**

**if (product.getId() == id) {**

***customerOrder*.addProduct(product, quantity);**

**JOptionPane.*showMessageDialog*(*frame*, "Product added to order.");**

**return;**

**}**

**}**

**JOptionPane.*showMessageDialog*(*frame*, "Product not found.");**

**} catch (NumberFormatException ex) {**

**JOptionPane.*showMessageDialog*(*frame*, "Invalid input. Please enter valid data.");**

**}**

**});**

**backButton.addActionListener(e -> *showCustomerMenu*());**

***panel*.add(new JLabel("Product ID:"));**

***panel*.add(idField);**

***panel*.add(new JLabel("Quantity:"));**

***panel*.add(quantityField);**

***panel*.add(orderButton);**

***panel*.add(backButton);**

***frame*.setContentPane(*panel*);**

***frame*.revalidate();**

***frame*.repaint();**

**}**

**private static void showTotalPayment() {**

**JOptionPane.*showMessageDialog*(*frame*, *customerOrder*.getOrderDetails());**

**}**

**}**

**``**

# 📚 References

#### *Books:*

*Dennis, A. and Haley, W. (2009) Systems Analysis and Design. John Wiley & Sons Ltd.*

*Ferguson, J. (2014) BDD in Action: Behavior-driven development for the whole software lifecycle. Manning.*

*Lejk, M. and Deeks, D. (2002) An Introduction to System Analysis Techniques. 2nd Ed. Addison-Wesley.*

*Murch, R. (2012) The Software Development Lifecycle: A Complete Guide. Kindle Edition.*

#### *Websites & Documentation:*

*Oracle (2023) Java Coding Standards. Available at:<https://www.oracle.com/java/> [Accessed 12 Jan. 2025].*

*Google (2023) Google Java Style Guide. Available at:<https://google.github.io/styleguide/javaguide.html> [Accessed 12 Jan. 2025].*

*JetBrains (2023) IntelliJ IDEA Debugging Tools. Available at:<https://www.jetbrains.com/help/idea/debugging.html> [Accessed 12 Jan. 2025].*

#### *Books:*

*[1] A. Dennis and W. Haley, Systems Analysis and Design. John Wiley & Sons Ltd, 2009.  
 [2] J. Ferguson, BDD in Action: Behavior-driven development for the whole software lifecycle. Manning, 2014.  
 [3] M. Lejk and D. Deeks, An Introduction to System Analysis Techniques, 2nd ed. Addison-Wesley, 2002.  
 [4] R. Murch, The Software Development Lifecycle: A Complete Guide. Kindle Edition, 2012.*

#### *Websites & Documentation:*

*[5] Oracle, Java Coding Standards, 2023. [Online]. Available:<https://www.oracle.com/java/>. [Accessed: 12-Jan-2025].  
 [6] Google, Google Java Style Guide, 2023. [Online]. Available:<https://google.github.io/styleguide/javaguide.html>. [Accessed: 12-Jan-2025].  
 [7] JetBrains, IntelliJ IDEA Debugging Tools, 2023. [Online]. Available:<https://www.jetbrains.com/help/idea/debugging.html>. [Accessed: 12-Jan-2025].*