MANISA CELAL BAYAR UNIVERSITY FACULTY OF ENGINEERING COMPUTER ENGINEERING

2024-2025 ACADEMIC YEAR
SPRING SEMESTER

Zeynep Sude BAYRAM 230316026

Ece Gül YÜKSEL 230316062

Seda Şengül ÖZKAYA 230316033

Computer Eng. Night Class

Object Oriented Programming

E-Commerce Application Project

Contents:

1. UML Diagrams

2) Order.java Class

The Order class is found in the model package and it is used to represent a customer's purchase order. It handles information about orders, including user, product, payment, quantity and date. It also includes the necessary logic to handle an order and verify both stock and payment.

1.Class Fields

```
5   public class Order {
6       private int orderId;
7       private User user;
8       private Product product;
9       private CreditCard creditCard;
10      private int quantity;
11      private Date orderDate;
```

orderId: This is the unique number for every order.
user: The person who makes the order. (Is connected to a User object)
product: What is being bought.
creditCard: The way the user is making the payment. quantity: How many product units are being ordered.
orderDate: The time when the order is first made.

2. Constructor

```
13      public Order(User user, Product product, CreditCard creditCard, int quantity) {
14          this.user = user;
15          this.product = product;
16          this.creditCard = creditCard;
17          this.quantity = quantity;
18          this.orderDate = new Date();
19      }
```

When a new Order object is made, this constructor sets it up with: The person who makes the order, the product that needs to be ordered, credit card was used for the payment, how much of the product is being produced. Uses new Date() to set the orderDate to the current time automatically.

3. Order Processing Logic

```
21      public boolean processOrder() {
22          if (creditCard.getExpDate() == null || creditCard.getExpDate().before(new Date())) {
23              System.out.println("Order failed: Credit card is invalid or expired.");
24              return false;
25          }
26          if (product.reduceStock(quantity)) {
27              user.orderProduct(product, quantity);
28              System.out.println("The order was created successfully.");
29              return true;
30          } else {
31              System.out.println("Order failed: Insufficient stock.");
32              return false;
33          }
34      }
```

This method performs key business logic and is responsible for the main business actions. Checking the Credit Card. An order will fail if the card has no expiration date or if it has already expired.
Checking if the product is in stock. Use product.reduceStock(quantity) to lower the amount of inventory. When successful, the user's order history is updated by calling user.orderProduct(...). If the stock is not enough, it prints an error message.
If the order is processed successfully, the method returns true; otherwise, it returns false.

4. **Getter and Setter Methods**

```
37      public int getOrderId() {
38          return orderId;
39      }
40
41      public void setOrderId(int orderId) {
42          this.orderId = orderId;
43      }
44
45      public User getUser() {
46          return user;
47      }
48
49      public void setUser(User user) {
50          this.user = user;
51      }
52
53      public Product getProduct() {
54          return product;
55      }
56
57      public void setProduct(Product product) {
58          this.product = product;
59      }
60
61      public CreditCard getCreditCard() {
62          return creditCard;
63      }
64
65      public void setCreditCard(CreditCard creditCard) {
66          this.creditCard = creditCard;
```

```
69      public int getQuantity() {
70          return quantity;
71      }
72
73      public void setQuantity(int quantity) {
74          this.quantity = quantity;
75      }
76
77      public Date getOrderDate() {
78          return orderDate;
79      }
80
81      public void setOrderDate(Date orderDate) {
82          this.orderDate = orderDate;
83      }
84  }
```

They are the usual getter and setter methods. They make it possible to manage who accesses and updates the class's private fields.

1) CreditCard.java Class

1.Class Fields

```
 7  public class CreditCard {
 8      private int cardId;
 9      private String cardNumber;
10      private User user;
11      private String securityCode;
12      private Date expDate;
```

cardId: each credit card is assigned a unique cardId.
cardNumber: A 16 digit String for the credit card number.
user: The person who owns the card (this also means a User object).
securityCode: The 3-digit CVV for security
expDate: The expiration date of the credit card. (java.util.Date).
These supply all of the necessary parts of a fully functional credit card.

2. Constuctor

```
14      public CreditCard(String cardNumber, User user, String securityCode, Date expDate) {
15          setCardNumber(cardNumber);
16          this.user = user;
17          setSecurityCode(securityCode);
18          this.expDate = expDate;
19      }
```

Here, the constructor starts a new credit card by setting the details given in the parameters. It checks if the card number or security code respects validation rules with setters rather than assigning value directly. By doing this, data is kept consistent and there is no chance for incorrect object creation.

3. Card Number Validation

```
33      public void setCardNumber(String cardNumber) {
34          if (cardNumber != null && cardNumber.matches("\\d{16}")) {
35              this.cardNumber = cardNumber;
36          } else {
37              System.out.println("Error: Credit card number must be a 16-digit number..");
38              this.cardNumber = null;
39          }
40      }
```

A regular expression is used in the method to guarantee the card number has exactly 16 digits (\\d{16}). If the validation process fails, an error message is written in the logs and the value is set to null. Using this way, basic security and integrity rules are enforced for each credit card number.

4. Security Code Validation

```
54      public void setSecurityCode(String securityCode) {
55          if (securityCode != null && securityCode.matches("\\d{3}")) {
56              this.securityCode = securityCode;
57          } else {
58              System.out.println("Error: Security code must be a 3-digit number.");
59              this.securityCode = null;
60          }
61      }
```

Similarly, the security code must be a **3-digit numeric string**. This simulates CVV code validation found on physical cards.

5.Getter and Setter Methods

```
21      public int getCardId() {
22          return cardId;
23      }
24
25      public void setCardId(int cardId) {
26          this.cardId = cardId;
27      }
28
29      public String getCardNumber() {
30          return cardNumber;
31      }
```

```java
    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }

    public String getSecurityCode() {
        return securityCode;
    }
    public Date getExpDate() {
        return expDate;
    }

    public void setExpDate(Date expDate) {
        this.expDate = expDate;
    }
```

They are the usual getter and setter methods. They make it possible to manage who accesses and updates the class's private fields.

6. Override

```java
    @Override
public String toString() {
    SimpleDateFormat sdf = new SimpleDateFormat("MM/yy");
    return "Card: **** **** **** " + cardNumber.substring(cardNumber.length() - 4)
            + " | Expiration: " + sdf.format(expDate);
}
```

The toString() method creates a readable string that represents the object. Shows the last 4 digits of the card, but hides the other numbers. Sets the expiration date to display in the "MM/yy" format.

2) Product.java Class

1.Class Fields

```java
public class Product {
    private int productId;
    private String productName;
    private String productColor;
    private String category;
    private int productStock;
    private double productPrice;
    private String description;
    private double productWeight;
```

productId: Each product has its own unique productId.
productName: Name of the product.
productColor: The color of the product.
category: The category of the product.

productStock: Number of items that avaiable for sale.
productPrice: Price of the products.
description: Details and a explanation of the product.
productWeight: Weight of the product.
With these fields, the system can keep and display information about each product and manage stock levels.

2. Constuctor

```
13      public Product() {}
14
15      public Product(String productName, String productColor, String category, int productStock, double productPrice, String description, double productWeight)
16          this.productName = productName;
17          this.productColor = productColor;
18          this.category = category;
19          this.productStock = productStock;
20          this.productPrice = productPrice;
21          this.description = description;
22          this.productWeight = productWeight;
23
24      }
```

The default constructor allows you to create a product object that starts with no values. The parameterized constructor sets the object with useful values. You use it when the product data has been collected and is ready to be saved.

3. Stock Management Logic

```
26      public boolean reduceStock(int quantity) {
27          if (productStock >= quantity) {
28              productStock -= quantity;
29              return true;
30          }
31          System.out.println("Insufficient stock: " + productName);
32          return false;
33      }
```

The method is applied to decrease the number of available products when a customer orders. If there are enough products in stock (productStock >= quantity), the method takes them and returns true. If the stock is not enough, it prints an error message and returns false. This method ensures that the order fulfillment process is correct by checking if the product is available.

4. Getters and Setters

```java
public double getProductWeight(){
    return productWeight;
}

public void setProductWeight(double productWeight){
    this.productWeight = productWeight;

}

public int getProductId() {
    return productId;
}

public void setProductId(int productId) {
    this.productId = productId;
}

public String getProductName() {
    return productName;
}

public void setProductName(String productName) {
    this.productName = productName;
}

public String getProductColor() {
    return productColor;
}

public void setProductColor(String productColor) {
    this.productColor = productColor;
}

public String getCategory() {
    return category;
}
```

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

public int getProductStock() {
    return productStock;
}

public void setProductStock(int productStock) {
    this.productStock = productStock;
}

public double getProductPrice() {
    return productPrice;
}

public void setProductPrice(double productPrice) {
    this.productPrice = productPrice;
}

public String getDescription() {
    return description;
}

public void setDescription(String description) {
    this.description = description;
}
}
```

3) User.java

1.Class Fields

```java
public class User {
    private int userId;
    private String username;
    private String name;
    private String surname;
    private Date birthDate;
    private String password;
    private String email;
    private String homeAdress;
    private String workAdress;
    private ArrayList<Product> orderedProducts;
    private ArrayList<Product> favorites;
    private ArrayList<CreditCard> creditCards;
```

The first set of fields (userId, username, etc.) is used to store personal and contact details. The ArrayLists are used for follow the progress of products that the user has bought. Keep their favorite things in one place. Look after their credit card accounts. This information is needed to support user interaction, customization and purchase history.

2. Constructors

```
20      public User(String username, String name, String surname, Date birthDate, String password, String email, String homeAdress, String workAdress) {
21          this.username = username;
22          this.name = name;
23          this.surname = surname;
24          this.birthDate = birthDate;
25          this.password = password;
26          this.email = email;
27          this.homeAdress = homeAdress;
28          this.workAdress = workAdress;
29          this.orderedProducts = new ArrayList<>();
30          this.favorites = new ArrayList<>();
31          this.creditCards = new ArrayList<>();
32      }
33
34      public User() {
35          orderedProducts = new ArrayList<>();
36          favorites = new ArrayList<>();
37          creditCards = new ArrayList<>();
38      }
```

The parameterized constructor is necessary when you want to create a user object with all its information. When a user object is needed with no data, the default constructor can be used (for example, during form filling or login). Both constructors set the list fields (orderedProducts, favorites and creditCards) to empty lists to avoid getting NullPointerException.

3. Favorites Management

```
40      public void addFavoriteProduct(Product product) {
41          if (!favorites.contains(product)) {
42              favorites.add(product);
43          }
44      }
```

With this method, users can add a product to their list of favorites. It makes sure not to add the same product twice by checking for its presence in the list. With this feature, users can quickly access the items they like the most.

4. Order Recording

```
46      public void orderProduct(Product product, int quantity) {
47          for (int i = 0; i < quantity; i++) {
48              orderedProducts.add(product);
49          }
50      }
```

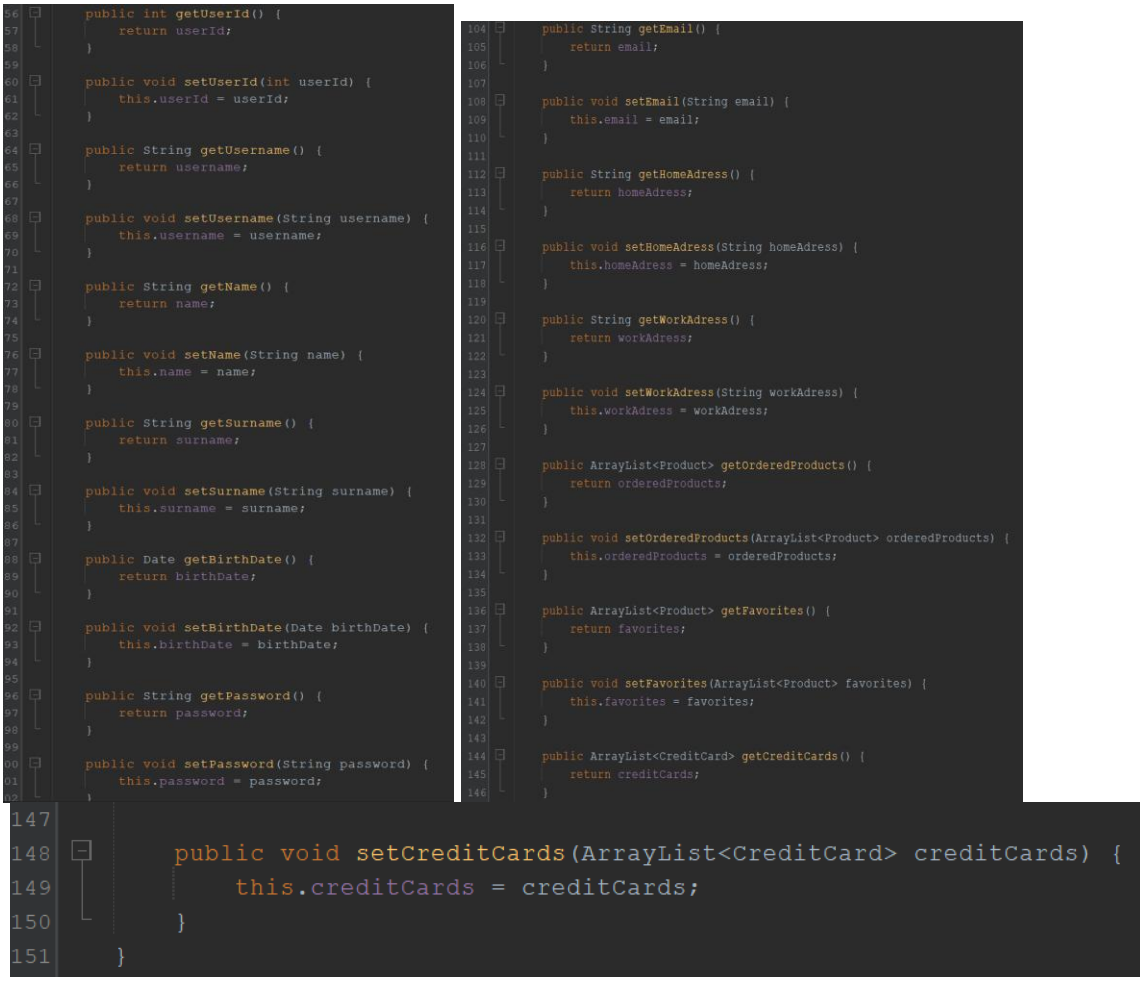
The method keeps a list of the ordered products in the user's history. It multiplies the product object quantity by the number of units to show the total number bought. This method is basic and treats every product as a single unit.

5. Credit Card Association

```
52      public void addCreditCard(CreditCard card) {
53          creditCards.add(card);
54      }
```

The method connects a credit card to the user. You can add several cards which makes checkout more flexible.

6. Getters and Setters



```java
    public int getUserId() {
        return userId;
    }

    public void setUserId(int userId) {
        this.userId = userId;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

    public Date getBirthDate() {
        return birthDate;
    }

    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
```

```java
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getHomeAdress() {
        return homeAdress;
    }

    public void setHomeAdress(String homeAdress) {
        this.homeAdress = homeAdress;
    }

    public String getWorkAdress() {
        return workAdress;
    }

    public void setWorkAdress(String workAdress) {
        this.workAdress = workAdress;
    }

    public ArrayList<Product> getOrderedProducts() {
        return orderedProducts;
    }

    public void setOrderedProducts(ArrayList<Product> orderedProducts) {
        this.orderedProducts = orderedProducts;
    }

    public ArrayList<Product> getFavorites() {
        return favorites;
    }

    public void setFavorites(ArrayList<Product> favorites) {
        this.favorites = favorites;
    }

    public ArrayList<CreditCard> getCreditCards() {
        return creditCards;
    }
```

```java

    public void setCreditCards(ArrayList<CreditCard> creditCards) {
        this.creditCards = creditCards;
    }
}
```

4) *EcommerceTest.java*

The purpose of this class is to make sure and show the behavior of the main model classes (User, Product and CreditCard) in a simulated online shopping environment.

```java
package test;

import model.User;
import model.Product;
import model.CreditCard;
import java.util.Date;
import java.util.ArrayList;
import java.text.SimpleDateFormat;

public class EcommerceTest {
    public static void main(String[] args) {
        System.out.println("E-commerce Application Test Starting...\n");
```

1.Creating Test User

```java
        // Create test user
        User testUser = new User(
            "testuser",
            "Test",
            "User",
            new Date(),
            "password123",
            "test@example.com",
            "Home Address",
            "Work Address"
        );
        System.out.println("1. User Creation Test:");
        System.out.println("Username: " + testUser.getUsername());
        System.out.println("Email: " + testUser.getEmail());
        System.out.println("Test successful!\n");
```

Goal is to ensure that a User object can be made and accessed properly. Prints out the username and email to check that the instantiation is correct.

2.Creating Test Product

```java
        // Create test product
        Product testProduct = new Product(
            "Test Product",
            "Red",
            "Electronics",
            10,
            99.99,
            "Test description",
            10.1
        );
        System.out.println("2. Product Creation Test:");
        System.out.println("Product name: " + testProduct.getProductName());
        System.out.println("Stock: " + testProduct.getProductStock());
        System.out.println("Price: " + testProduct.getProductPrice());
        System.out.println("Color: " + testProduct.getProductColor());
        System.out.println("Category: " + testProduct.getCategory());
        System.out.println("Description: " + testProduct.getDescription());
        System.out.println("Test successful!\n");
```

The goal is to make sure Product objects start with the correct values for name, stock and category. Checks the stock, color, price and other main details.

3.Credit Card Test

```
        // Credit card addition test
        SimpleDateFormat sdf = new SimpleDateFormat("MM/yy");
        Date expDate = null;
        try {
            expDate = sdf.parse("12/25");
        } catch (Exception e) {
            System.out.println("Date conversion error: " + e.getMessage());
        }

        CreditCard testCard = new CreditCard(
            "1234567890123456",
            testUser,
            "123",
            expDate
        );
        testUser.addCreditCard(testCard);
        System.out.println("3. Credit Card Addition Test:");
        System.out.println("Number of added cards: " + testUser.getCreditCards().size());
        System.out.println("Card number: " + testUser.getCreditCards().get(0).getCardNumber());
        System.out.println("Test successful!\n");
```

Check that a credit card can be linked to a user and shown properly. Checks that the card count and numbers are correct and are not hidden here for testing.

4.Product Purchase Test

```
        System.out.println("4. Product Purchase Test:");
        int purchaseQuantity = 2;
        int initialStock = testProduct.getProductStock();
        testUser.orderProduct(testProduct, purchaseQuantity);
        System.out.println("Number of ordered products: " + testUser.getOrderedProducts().size());
        testProduct.reduceStock(purchaseQuantity);
        System.out.println("Remaining stock: " + testProduct.getProductStock());
        System.out.println("Test successful!\n");
```

The goal is to buy something and ensure that the product count in the order is right and the stock is updated. Makes sure the number of items purchased is removed from the inventory.

5. Add to Favorites Test

```
        // Add to favorites test
        System.out.println("5. Add to Favorites Test:");
        testUser.addFavoriteProduct(testProduct);
        System.out.println("Number of favorite products: " + testUser.getFavorites().size());
        System.out.println("Favorite product name: " + testUser.getFavorites().get(0).getProductName())
        System.out.println("Test successful!\n");
```

The purpose is to make sure that users are able to add products to their favorites. The list displays the number of items and the title of the product you have saved.

6. Duplicate Favorite Addition Test

```
        // Test adding the same product to favorites again
        System.out.println("6. Duplicate Favorite Addition Test:");
        testUser.addFavoriteProduct(testProduct);
        System.out.println("Number of favorite products (after duplicate addition): " + testUser.getFavorites().size());
        System.out.println("Test successful!\n");
```

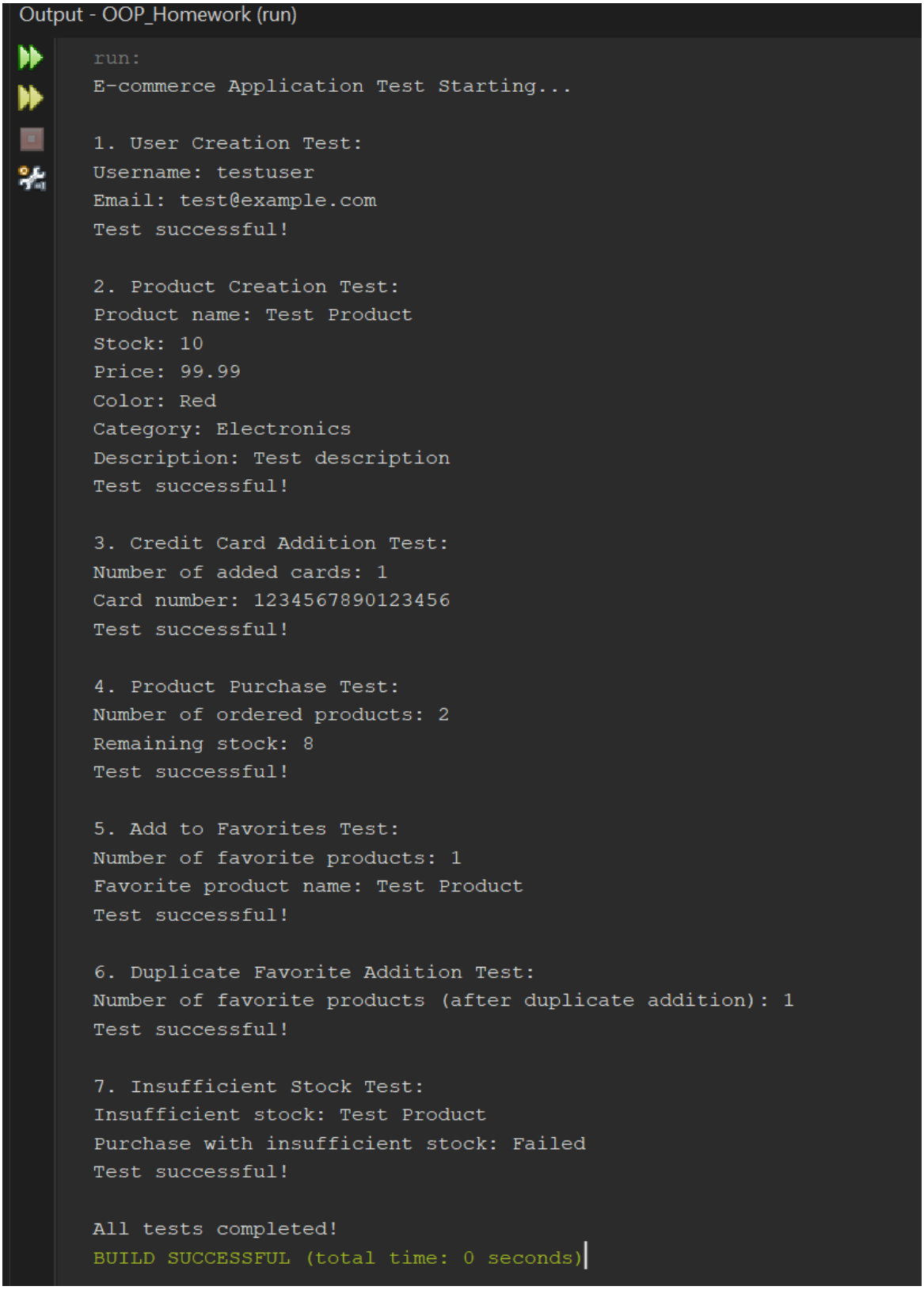
The purpose is to make sure no favorites are repeated. The list size does not increase when you try to add a product that was already there.

7. Insufficient Stock Test and Final Output

```
93              // Insufficient stock test
94              System.out.println("7. Insufficient Stock Test:");
95              int insufficientQuantity = testProduct.getProductStock() + 1;
96              boolean stockStatus = testProduct.reduceStock(insufficientQuantity);
97              System.out.println("Purchase with insufficient stock: " + (stockStatus ? "Successful" : "Failed"));
98              System.out.println("Test successful!\n");
99
100             System.out.println("All tests completed!");
101         }
102     }
```

The aim is to confirm that product inventory will not fall below zero. The function returns false and an appropriate error message is displayed then it shows that all the steps are complete and enables the tester to check the logic by looking at the console.

Total Console Output:



3) DATABASE

We use PostgreSQL pg4admin as our database.

1.User Management

This section covers how the application handles user information, such as creating accounts, finding

user details, and checking login credentials.

Add User

```
public class UserDAO {
    public boolean addUser(User user) {
        String sql = "INSERT INTO users(username, name, surname, birth_date, password, email, home_address, work_address) " +
                     "VALUES(?, ?, ?, ?, ?, ?, ?, ?)";

        try (Connection conn = DatabaseConnection.getConnection();
             PreparedStatement pstmt = conn.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)) {

            pstmt.setString(1, user.getUsername());
            pstmt.setString(2, user.getName());
            pstmt.setString(3, user.getSurname());
            pstmt.setDate(4, new java.sql.Date(user.getBirthDate().getTime()));
            pstmt.setString(5, user.getPassword());
            pstmt.setString(6, user.getEmail());
            pstmt.setString(7, user.getHomeAdress());
            pstmt.setString(8, user.getWorkAdress());

            int affectedRows = pstmt.executeUpdate();

            if (affectedRows > 0) {
                try (ResultSet rs = pstmt.getGeneratedKeys()) {
                    if (rs.next()) {
                        user.setUserId(rs.getInt(1));
                    }
                }
                return true;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }
```

Description: This code creates a new user account by saving details like username, full name, birth date, password, email, and home and work addresses into a database table called "users." When the user is added, the database assigns a unique ID to them, which this code attaches to the user's data. If the process works, it returns "true" to confirm the account was created. If something goes wrong, like a connection issue with the database, it logs the error and returns "false." This code is critical for allowing new users to sign up for the application, ensuring their information is stored securely and can be accessed later.

Get User by Username

```
    public User getUserByUsername(String username) {
        String sql = "SELECT * FROM users WHERE username = ?";
        User user = null;

        try (Connection conn = DatabaseConnection.getConnection();
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setString(1, username);
            ResultSet rs = pstmt.executeQuery();

            if (rs.next()) {
                user = new User();
                user.setUserId(rs.getInt("user_id"));
                user.setUsername(rs.getString("username"));
                user.setName(rs.getString("name"));
                user.setSurname(rs.getString("surname"));
                user.setBirthDate(rs.getDate("birth_date"));
                user.setPassword(rs.getString("password"));
                user.setEmail(rs.getString("email"));
                user.setHomeAdress(rs.getString("home_address"));
                user.setWorkAdress(rs.getString("work_address"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return user;
    }
```

Description: This code looks for a user in the database by their username. It searches the "users" table and, if it finds a match, gathers all the user's details like their ID, name, birth date, email, and addresses. It then creates a user object with this information and returns it to the application. If no user is found with that username or if there's a problem connecting to the database, it returns nothing (null). This is useful for things like displaying a user's profile page or checking if a username is already taken during registration.

Validate User

```
    public boolean validateUser(String username, String password) {
        String sql = "SELECT * FROM users WHERE username = ? AND password = ?";

        try (Connection conn = DatabaseConnection.getConnection();
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setString(1, username);
            pstmt.setString(2, password);
            ResultSet rs = pstmt.executeQuery();

            return rs.next();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }
}
```

Description: This code checks if a user's login details are correct by looking for a matching username and password in the "users" table. If it finds a record with both the username and password, it returns "true," meaning the user can log in. If no match is found or there's a database error, it returns "false." This code is essential for the login process, ensuring only users with valid credentials can access the system. It keeps the login secure by carefully checking both pieces of information against the database records.

2. Seller Management

This section explains how the application verifies sellers who log into the system.

Validate Seller

```
public class SellerDAO {
    public boolean validateSeller(String username, String password) {
        String sql = "SELECT * FROM sellers WHERE username = ? AND password = ?";

        try (Connection conn = DatabaseConnection.getConnection();
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setString(1, username);
            pstmt.setString(2, password);
            ResultSet rs = pstmt.executeQuery();

            return rs.next();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }
}
```

Description: This code verifies a seller's login information by checking if their username and password exist in the "sellers" table. If a matching record is found, it returns "true," allowing the seller to access the system, such as a seller dashboard for managing products. If no match is found or there's a database issue, it returns "false." This ensures that only authorized sellers can log in, keeping their accounts secure and preventing unauthorized access.

3. Product Management

This section covers how the application manages product information, such as adding new products and retrieving their details.

Add Product

```java
public boolean addProduct(Product product) {
    String sql = "INSERT INTO products(product_name, product_color, category, product_stock, product_price, description) " +
                 "VALUES(?, ?, ?, ?, ?, ?)";

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)) {

        pstmt.setString(1, product.getProductName());
        pstmt.setString(2, product.getProductColor());
        pstmt.setString(3, product.getCategory());
        pstmt.setInt(4, product.getProductStock());
        pstmt.setDouble(5, product.getProductPrice());
        pstmt.setString(6, product.getDescription());

        int affectedRows = pstmt.executeUpdate();

        if (affectedRows > 0) {
            try (ResultSet rs = pstmt.getGeneratedKeys()) {
                if (rs.next()) {
                    product.setProductId(rs.getInt(1));
                }
            }
            return true;
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return false;
}
```

Description: This code adds a new product to the "products" table in the database, saving details like the product's name, color, category (e.g., electronics or clothing), stock quantity, price, and description. After adding the product, it retrieves a unique ID assigned by the database and attaches it to the product's data. If the addition is successful, it returns "true." If there's a problem, like a database connection error, it logs the issue and returns "false." This code is crucial for sellers who need to add new items to their inventory, ensuring all product details are stored correctly for display in the application.

Get All Products

```java
public List<Product> getAllProducts() {
    String sql = "SELECT * FROM products";
    List<Product> products = new ArrayList<>();

    try (Connection conn = DatabaseConnection.getConnection();
         Statement stmt = conn.createStatement();
         ResultSet rs = stmt.executeQuery(sql)) {

        while (rs.next()) {
            Product product = new Product(
                rs.getString("product_name"),
                rs.getString("product_color"),
                rs.getString("category"),
                rs.getInt("product_stock"),
                rs.getDouble("product_price"),
                rs.getString("description"),
                rs.getDouble("product_weight")
            );
            product.setProductId(rs.getInt("product_id"));
            products.add(product);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return products;
}
```

Description: This code retrieves every product stored in the "products" table and puts them into a list. For each product, it collects details like name, color, category, stock, price, description, and weight, creating a product object for each one. The list is then returned to the application. This is used to display all available products, for example, in an online store's catalog or inventory management page. If there's a database error, it logs the issue and returns an empty list, ensuring the system remains stable.

Get Products by Category

```
public List<Product> getProductsByCategory(String category) {
    String sql = "SELECT * FROM products WHERE category = ?";
    List<Product> products = new ArrayList<>();

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setString(1, category);
        ResultSet rs = pstmt.executeQuery();

        while (rs.next()) {
            Product product = new Product(
                rs.getString("product_name"),
                rs.getString("product_color"),
                rs.getString("category"),
                rs.getInt("product_stock"),
                rs.getDouble("product_price"),
                rs.getString("description"),
                rs.getDouble("product_weight")
            );
            product.setProductId(rs.getInt("product_id"));
            products.add(product);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return products;
}
```

Description: This code finds all products that belong to a specific category, like "electronics" or "clothing," and returns them as a list. It searches the "products" table for items matching the given category, gathers their details (name, price, weight, etc.), and creates a product object for each. This list is useful for showing users products grouped by category, such as in a store's category-based browsing feature. If no products are found or there's a database issue, it logs the error and returns an empty list, keeping the system reliable.

Get Product by ID

```
public Product getProductById(int productId) {
    String sql = "SELECT * FROM products WHERE product_id = ?";
    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setInt(1, productId);
        ResultSet rs = pstmt.executeQuery();

        if (rs.next()) {
            Product product = new Product(
                rs.getString("product_name"),
                rs.getString("product_color"),
                rs.getString("category"),
                rs.getInt("product_stock"),
                rs.getDouble("product_price"),
                rs.getString("description"),
                rs.getDouble("product_weight")
            );
            product.setProductId(rs.getInt("product_id"));
            return product;
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return null;
}
```

Description: This code retrieves a single product from the "products" table using its unique ID. If the product is found, it creates a product object with details like name, color, price, and weight, and returns it. If no product matches the ID or there's a database error, it returns nothing (null). This is used when a user clicks on a specific product to view its details, such as on a product page in an online store. The code ensures accurate retrieval of product information for display.

Update Product Stock

```
    public boolean updateProductStock(int productId, int quantity) {
        String sql = "UPDATE products SET product_stock = product_stock - ? WHERE product_id = ? AND product_stock >= ?";
        try (Connection conn = DatabaseConnection.getConnection();
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setInt(1, quantity);
            pstmt.setInt(2, productId);
            pstmt.setInt(3, quantity);

            return pstmt.executeUpdate() > 0;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }
}
```

Description: This code updates a product's stock by reducing it by a specified amount, such as when a customer buys the product. It checks that there's enough stock to avoid negative values, ensuring inventory accuracy. If the update is successful, it returns "true." If there's not enough stock or a database error occurs, it logs the error and returns "false." This code is vital for keeping track of available products in the system, especially during purchases.

4. Order Management This section explains how the application manages customer orders.

Add Order

```
    public boolean addOrder(Order order) {
        String sql = "INSERT INTO orders(user_id, product_id, card_id, quantity) VALUES(?, ?, ?, ?)";

        try (Connection conn = DatabaseConnection.getConnection();
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setInt(1, order.getUser().getUserId());
            pstmt.setInt(2, order.getProduct().getProductId());
            pstmt.setInt(3, order.getCreditCard().getCardId());
            pstmt.setInt(4, order.getQuantity());

            return pstmt.executeUpdate() > 0;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }
```

Description: This code creates a new order by saving details like the user's ID, product ID, credit card ID, and the quantity ordered into the "orders" table. If the order is saved successfully, it returns "true." If there's a database issue, it logs the error and returns "false." This code is used when a customer completes a purchase, ensuring their order is recorded properly in the system for processing and tracking.

Get Orders by User

```
    public List<Order> getOrdersByUser(int userId) {
        String sql = "SELECT o.*, p.* FROM orders o " +
                     "JOIN products p ON o.product_id = p.product_id " +
                     "WHERE o.user_id = ? ORDER BY o.order_date DESC";
        List<Order> orders = new ArrayList<>();

        try (Connection conn = DatabaseConnection.getConnection();
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setInt(1, userId);
            ResultSet rs = pstmt.executeQuery();

            while (rs.next()) {
                Product product = new Product(
                    rs.getString("product_name"),
                    rs.getString("product_color"),
                    rs.getString("category"),
                    rs.getInt("product_stock"),
                    rs.getDouble("product_price"),
                    rs.getString("description"),
                    rs.getDouble("product_weight")
                );
                product.setProductId(rs.getInt("product_id"));

                Order order = new Order(null, product, null, rs.getInt("quantity"));
                order.setOrderId(rs.getInt("order_id"));
                order.setOrderDate(rs.getTimestamp("order_date"));
                orders.add(order);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return orders;
    }
}
```

Description: This code retrieves all orders placed by a specific user, sorted from the most recent to the oldest. It combines information from the "orders" and "products" tables to include details about each product in the order, such as its name, price, and weight. It creates a list of order objects and returns it to the application. This is used to show a user their order history, like in a "My Orders" section of an online store. If there's a database error, it logs the issue and returns an empty list.

5. Favorites Management This section covers how the application manages a user's favorite products.

Add Favorite

```
    public boolean addFavorite(int userId, int productId) {
        String sql = "INSERT INTO favorites(user_id, product_id) VALUES(?, ?)";

        try (Connection conn = DatabaseConnection.getConnection();
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setInt(1, userId);
            pstmt.setInt(2, productId);

            return pstmt.executeUpdate() > 0;
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }
```

Description: This code allows a user to mark a product as a favorite by adding a record to the "favorites" table with the user's ID and the product's ID. If the record is saved successfully, it returns "true." If there's a database error, it logs the issue and returns "false." This is used for features like a wishlist, where users can save products they're interested in for later purchase, making shopping more convenient.

Remove Favorite

```
public boolean removeFavorite(int userId, int productId) {
    String sql = "DELETE FROM favorites WHERE user_id = ? AND product_id = ?";

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setInt(1, userId);
        pstmt.setInt(2, productId);

        return pstmt.executeUpdate() > 0;
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return false;
}
```

Description: This code removes a product from a user's favorites list by deleting the matching record from the "favorites" table, using the user's ID and the product's ID. If the deletion is successful, it returns "true." If there's an error, like a database issue, it logs it and returns "false." This allows users to update their wishlist by removing products they no longer want to keep track of, keeping their favorites list current.

Get Favorite Product IDs

```
public List<Integer> getFavoriteProductIds(int userId) {
    String sql = "SELECT product_id FROM favorites WHERE user_id = ?";
    List<Integer> productIds = new ArrayList<>();

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setInt(1, userId);
        ResultSet rs = pstmt.executeQuery();

        while (rs.next()) {
            productIds.add(rs.getInt("product_id"));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return productIds;
}
}
```

Description: This code retrieves the IDs of all products in a user's favorites list from the "favorites" table. It searches for records matching the user's ID, collects the product IDs into a list, and returns it. This is used to display a user's wish list, showing all the products they've marked as favorites. If there's a database error, it logs the issue and returns an empty list, ensuring the system remains functional.

6. Credit Card Management

This section explains how the application handles credit card information for users.

Add Credit Card

```
public boolean addCreditCard(CreditCard card) {
    String sql = "INSERT INTO credit_cards(card_number, user_id, security_code, exp_date) " +
                 "VALUES(?, ?, ?, ?)";

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)) {

        pstmt.setString(1, card.getCardNumber());
        pstmt.setInt(2, card.getUser().getUserId());
        pstmt.setString(3, card.getSecurityCode());
        pstmt.setDate(4, new java.sql.Date(card.getExpDate().getTime()));

        int affectedRows = pstmt.executeUpdate();

        if (affectedRows > 0) {
            try (ResultSet rs = pstmt.getGeneratedKeys()) {
                if (rs.next()) {
                    card.setCardId(rs.getInt(1));
                }
            }
            return true;
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return false;
}
```

Description: This code saves a new credit card to the 'creditcards' table, including the card number and user ID, by entering the details each time.

Get Credit Cards by User

```
public List<CreditCard> getCreditCardsByUser(int userId) {
    String sql = "SELECT * FROM credit_cards WHERE user_id = ?";
    List<CreditCard> cards = new ArrayList<>();

    try (Connection conn = DatabaseConnection.getConnection();
         PreparedStatement pstmt = conn.prepareStatement(sql)) {

        pstmt.setInt(1, userId);
        ResultSet rs = pstmt.executeQuery();

        while (rs.next()) {
            User user = new User();
            user.setUserId(userId);

            CreditCard card = new CreditCard(
                rs.getString("card_number"),
                user,
                rs.getString("security_code"),
                rs.getDate("exp_date")
            );
            card.setCardId(rs.getInt("card_id"));
            cards.add(card);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return cards;
}
```

Description: This code retrieves all credit cards saved by a specific user from the 'creditcards' table. It creates a list of the user's saved cards.

Get Credit Card by ID

```
    public CreditCard getCreditCardById(int cardId) {
        String sql = "SELECT * FROM credit_cards WHERE card_id = ?";
        try (Connection conn = DatabaseConnection.getConnection();
             PreparedStatement pstmt = conn.prepareStatement(sql)) {

            pstmt.setInt(1, cardId);
            ResultSet rs = pstmt.executeQuery();

            if (rs.next()) {
                User user = new User();
                user.setUserId(rs.getInt("user_id"));

                CreditCard card = new CreditCard(
                    rs.getString("card_number"),
                    user,
                    rs.getString("security_code"),
                    rs.getDate("exp_date")
                );
                card.setCardId(rs.getInt("card_id"));
                return card;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

Description: This code finds a specific credit card by its unique ID in the 'creditcards' table. If the card is found, it returns the card details.

7. Database Connection This section explains how the application connects to the database.

Get Connection

```
public class DatabaseConnection {
    private static final String URL = "jdbc:postgresql://localhost:5432/postgres";
    private static final String USER = "postgres";
    private static final String PASSWORD = "oop123";

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USER, PASSWORD);
    }
}
```

Description: This code sets up a connection to the PostgreSQL database using a specific address, username, and password. It's used by all other parts of the application to access the database, ensuring that all operations (like adding users or products) can communicate with the database properly. If the connection fails, it logs an error, helping developers identify and fix the issue. This code is the foundation for all database interactions in the application.

Summary

This report shows how the application's code manages data for users, sellers, products, orders, favorites, and credit cards. Each piece of code is designed to be secure, reliable, and easy to use, with clear error handling to keep the system running smoothly. The explanations provided make it easy to understand how each part works and why it's important for the application.

Database Table Description (SQL Script)

Users Table

```sql
-- Users table
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    name VARCHAR(50) NOT NULL,
    surname VARCHAR(50) NOT NULL,
    birth_date DATE NOT NULL,
    password VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    home_address TEXT NOT NULL,
    work_address TEXT
);
```

The users table stores information about customers who use the platform. Each user has a unique user_id, along with credentials and personal details such as name, surname, date of birth, email, password, and addresses. Both home and work addresses are supported, though work address is optional.

Credit_Cards Table

```sql
-- Credit cards table
CREATE TABLE credit_cards (
    card_id SERIAL PRIMARY KEY,
    card_number VARCHAR(16) NOT NULL,
    user_id INTEGER REFERENCES users(user_id) ON DELETE CASCADE,
    security_code VARCHAR(3) NOT NULL,
    exp_date DATE NOT NULL
);
```

The credit_cards table stores users' credit card information securely. Each card is linked to a specific user via user_id. The table includes essential fields such as card_number, security_code, and exp_date. This table supports multiple cards per user.

Products Table

```sql
-- Products table
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    product_name VARCHAR(100) NOT NULL,
    product_color VARCHAR(50) NOT NULL,
    category VARCHAR(50) NOT NULL,
    product_stock INTEGER NOT NULL,
    product_price DOUBLE PRECISION NOT NULL,
    description TEXT,
    product_weight DOUBLE PRECISION NOT NULL
);
```

The products table contains data about items available for purchase. It includes attributes such as product name, color, stock level, price, description, and weight. The table also references the categories table via category_id to link each product to its corresponding category, promoting data normalization and consistency.

Favorites Table

```sql
-- Favorites table
CREATE TABLE favorites (
    favorite_id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(user_id) ON DELETE CASCADE,
    product_id INTEGER REFERENCES products(product_id) ON DELETE CASCADE,
    UNIQUE(user_id, product_id)
);
```

The favorites table tracks the products that users have marked as favorites. It forms a many-to-many relationship between users and products. Each user can have multiple favorite products, but each combination of user_id and product_id is unique to prevent duplication.

Orders Table

```sql
-- Orders table
CREATE TABLE orders (
    order_id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(user_id) ON DELETE SET NULL,
    product_id INTEGER REFERENCES products(product_id) ON DELETE SET NULL,
    card_id INTEGER REFERENCES credit_cards(card_id) ON DELETE SET NULL,
    quantity INTEGER NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

The orders table logs all purchases made on the platform. It references the users, products, and credit_cards tables to associate each order with the customer, the product, and the payment method. It includes order-specific data like quantity and the timestamp of the order (order_date), which defaults to the current time.

Sellers Table

```sql
-- Sellers table
CREATE TABLE sellers (
    seller_id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    password VARCHAR(100) NOT NULL,
    company_name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
);
```

The sellers table stores data about vendors who offer products on the platform. Each seller has a unique username, email, and associated company name. Authentication details like password are also included for seller login purposes.

4)User Interface

1. **LoginFrame.java**

This class is a Swing-based user interface that forms the login screen of the e-commerce application. Login credentials provided by users or sellers are verified through the UserDAO and SellerDAO classes. Upon successful login, the user is redirected to the appropriate panel; in case of failure, the user is notified. It also provides basic functionalities such as navigating to the registration screen and toggling password visibility.

```
public class LoginFrame extends JFrame {
    private JTextField usernameField;
    private JPasswordField passwordField;
    private JButton loginButton, registerButton, sellerLoginButton;
    private JButton togglePasswordButton;
    private boolean isPasswordVisible = false;
```

In this section, all the components to be used in the interface (text boxes, buttons, and status variable) are defined. The items that will receive user login information and manage user interactions are specified here.

```
    public LoginFrame() {
        setTitle( title:"Welcome to E-Commerce");
        setSize( width:420, height:420);
        setDefaultCloseOperation( operation:JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo( c:null);
```

The title, size, closing behavior, and position of the window are set. The interface opens in a fixed size in the middle of the screen.

```
        Color backgroundColor = new Color( r:250, g:250, b:250);
        Color fieldColor = Color.WHITE;
        Color labelColor = new Color( r:60, g:60, b:60);
        Color buttonColor = new Color( r:33, g:150, b:243);
        Color buttonTextColor = Color.WHITE;
        Font font = new Font( name:"Segoe UI", style:Font.PLAIN, size:15);
        Font titleFont = new Font( name:"Segoe UI", style:Font.BOLD, size:22);
```

Interface design of color and Font settings are made. Light tones and readable fonts have been preferred for a modern and simple look.

```
        JPanel mainPanel = new JPanel();
        mainPanel.setBackground( bg:backgroundColor);
        mainPanel.setLayout(new BoxLayout( target:mainPanel, axis:BoxLayout.Y_AXIS));
        mainPanel.setBorder( border:BorderFactory.createEmptyBorder( top:30, left:40, bottom:30, right:40));
```

The main panel on which all components will be placed is defined and vertical (from top to bottom) alignment is made. The inner margins of the panel are also determined here.

```
        JLabel titleLabel = new JLabel( text:"Login to Your Account");
        titleLabel.setFont( font:titleFont);
        titleLabel.setForeground(new Color( r:40, g:40, b:40));
        titleLabel.setAlignmentX( alignmentX:Component.CENTER_ALIGNMENT);
        titleLabel.setBorder( border:BorderFactory.createEmptyBorder( top:0, left:0, bottom:20, right:0));
        mainPanel.add( comp:titleLabel);
```

The title tag is added and aligned to the center to show the user which screen they are on. Provides visual hierarchy.

```
JLabel userLabel = new JLabel( text:"Username:");
userLabel.setForeground( fg:labelColor);
userLabel.setFont(font);
userLabel.setAlignmentX( alignmentX:Component.CENTER_ALIGNMENT); // ORTALA
mainPanel.add( comp:userLabel);

usernameField = new JTextField();
usernameField.setFont( f:font);
usernameField.setBackground( bg:fieldColor);
usernameField.setMaximumSize(new Dimension( width:Integer.MAX_VALUE, height:36));
usernameField.setBorder( border:BorderFactory.createLineBorder(new Color( r:200, g:200, b:200)));
mainPanel.add( comp:usernameField);
mainPanel.add( comp:Box.createRigidArea(new Dimension( width:0, height:15)));
```

The field in which the user will enter the user name information and the label belonging to it are defined. The text box can expand horizontally.

```
JLabel passLabel = new JLabel( text:"Password:");
passLabel.setForeground( fg:labelColor);
passLabel.setFont(font);
passLabel.setAlignmentX( alignmentX:Component.CENTER_ALIGNMENT); // ORTALA
mainPanel.add( comp:passLabel);
```

A descriptive label is created for the area where the user will enter the password and is aligned to the center.

```
JPanel passwordPanel = new JPanel(new BorderLayout());
passwordPanel.setMaximumSize(new Dimension( width:Integer.MAX_VALUE, height:36));
passwordPanel.setBackground( bg:backgroundColor);
```

A horizontally aligned panel (BorderLayout) is created to place the password field and button side by side. The background color is selected to match the general theme.

```
togglePasswordButton = new JButton( text:"👁");
togglePasswordButton.setPreferredSize(new Dimension( width:50, height:36));
togglePasswordButton.setFocusPainted( b:false);
togglePasswordButton.setBackground( bg:buttonColor);
togglePasswordButton.setForeground( fg:Color.WHITE);
togglePasswordButton.setBorder( border:null);
```

A text box is defined for password entry. Characters are displayed in a hidden manner. Font, background and border are determined for visual harmony.

```
togglePasswordButton.addActionListener(e -> {
    isPasswordVisible = !isPasswordVisible;
    passwordField.setEchoChar(isPasswordVisible ? (char) 0 : '*');
    togglePasswordButton.setText(isPasswordVisible ? "🙈" : "👁");
});
```

Clicking the button changes whether the password is hidden or not. The icon is also updated to provide visual feedback to the user.

```
passwordPanel.add( comp:passwordField, constraints:BorderLayout.CENTER);
passwordPanel.add( comp:togglePasswordButton, constraints:BorderLayout.EAST);
mainPanel.add( comp:passwordPanel);
mainPanel.add( comp:Box.createRigidArea(new Dimension( width:0, height:25)));
```

The password field and the eye button are placed horizontally in the same panel. Then this panel is added to the main panel. A 25 pixel gap is left between the components to provide distance.

```
JPanel buttonPanel = new JPanel();
buttonPanel.setLayout(new GridLayout( rows:3, cols:1, hgap:10, vgap:10));
buttonPanel.setOpaque( isOpaque:false);
```

A panel is created where the login, registration and seller login buttons will be placed. With GridLayout, the buttons are arranged vertically and a 10 pixel gap is left between them. The panel is set to transparent.

```
loginButton = new JButton( text:"Login");
registerButton = new JButton( text:"Sign Up");
sellerLoginButton = new JButton( text:"Seller Login");
```

Three separate buttons are created for user login, registration and vendor login operations.

```
JButton[] buttons = {loginButton, registerButton, sellerLoginButton};
for (JButton button : buttons) {
    button.setBackground( bg:buttonColor);
    button.setForeground( fg:buttonTextColor);
    button.setFocusPainted( b:false);
    button.setFont(font);
    button.setPreferredSize(new Dimension( width:150, height:35));
    buttonPanel.add( comp:button);
}
```

The buttons are given a common color, font, and size. Finally, they are all added to the buttonPanel.

```
mainPanel.add( comp:buttonPanel);
add( comp:mainPanel);
```

Buttons are added to the main panel and then this main panel is placed inside the window (JFrame).

```
        loginButton.addActionListener(e -> {
            String username = usernameField.getText();
            String password = new String( value: passwordField.getPassword());

            UserDAO userDAO = new UserDAO();
            if (userDAO.validateUser(username, password)) {
                JOptionPane.showMessageDialog( parentComponent: LoginFrame.this,  message: "Login successful");
                new UserDashboard(username).setVisible( b: true);
                dispose();
            } else {
                JOptionPane.showMessageDialog( parentComponent: LoginFrame.this,  message: "Invalid username or password",  title: "Error",
                        messageType: JOptionPane.ERROR_MESSAGE);
            }
        });
```

When the user clicks the “Login” button, the login information is verified using the UserDAO class. If successful, the UserDashboard screen opens. If unsuccessful, a warning message is displayed to the user.

```
        sellerLoginButton.addActionListener(e -> {
            String username = usernameField.getText();
            String password = new String( value: passwordField.getPassword());

            SellerDAO sellerDAO = new SellerDAO();
            if (sellerDAO.validateSeller(username, password)) {
                JOptionPane.showMessageDialog( parentComponent: LoginFrame.this,  message: "Seller login successful");
                new SellerDashboard().setVisible( b: true);
                dispose();
            } else {
                JOptionPane.showMessageDialog( parentComponent: LoginFrame.this,  message: "Invalid seller information!",  title: "Error",
                        messageType: JOptionPane.ERROR_MESSAGE);
            }
        });
```

When the “Seller Login” button is clicked, the seller information is checked with Seller O. If it is correct, the SellerDashboard screen opens, if it is incorrect, an error message is displayed.

```
        registerButton.addActionListener(e -> new RegistrationFrame().setVisible( b: true));
    }
}
```

When the user clicks the “Sign Up” button, a new registration window (RegistrationFrame) opens.

2. Main.java

This code is where the application first runs. When the program is started, it first checks whether the PostgreSQL database driver is loaded. Then, the login screen, LoginFrame, is opened and shown to the user. This class is used to initialize the interface.

```
package ui;

import ui.LoginFrame;
import javax.swing.*;

public class Main {
    public static void main(String[] args) {
        try {
            Class.forName( className: "org.postgresql.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println( x: "PostgreSQL JDBC Driver not found.");
            e.printStackTrace();
            return;
        }
```

An attempt is made to load the PostgreSQL driver, which is required for database connections to work. If it is not found, an error message is printed and the program stops working.

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

The Swing interface is started. The interface is created securely thanks to invokeLater. Then the login screen (LoginFrame) is displayed to the user.

3. RegistrationFrame.java

This code creates the registration screen that allows users to register to the system. Data such as username, password, name, surname, date of birth, e-mail and address information are collected from the user. After the form is completed, the information is saved to the database. The interface has a simple form structure and is created with Swing components. After successful registration, the system gives an information message.

```
public class RegistrationFrame extends JFrame {
    private JTextField usernameField, nameField, surnameField, emailField, homeAddressField, workAddressField;
    private JPasswordField passwordField;
    private JComboBox<Integer> dayCombo, monthCombo, yearCombo;
    private JButton registerButton;
```

All fields to be used in the form are defined here. The text boxes to be filled by the user, the date of birth selection boxes and the registration button are determined.

```
    public RegistrationFrame() {
        setTitle( title:"User Registration");
        setSize( width:500,  height:400);
        setDefaultCloseOperation( operation:JFrame.DISPOSE_ON_CLOSE);
        setLocationRelativeTo( c:null);
```

The title, size and closing behavior of the recording screen are set. The window opens in the middle of the screen and when closed, only this window is closed.

```
        JPanel panel = new JPanel(new GridLayout( rows:9,  cols:2,  hgap:10,  vgap:10));
        panel.setBorder( border:BorderFactory.createEmptyBorder( top:20,  left:20,  bottom:20,  right:20));
```

To properly place the form fields, a GridLayout with 9 rows and 2 columns is used. Each component has 10 pixels of space between them.

```
        panel.add(new JLabel( text:"User name:"));
        usernameField = new JTextField();
        panel.add( comp:usernameField);

        panel.add(new JLabel( text:"Password:"));
        passwordField = new JPasswordField();
        panel.add( comp:passwordField);

        panel.add(new JLabel( text:"Name:"));
        nameField = new JTextField();
        panel.add( comp:nameField);

        panel.add(new JLabel( text:"Surname:"));
        surnameField = new JTextField();
        panel.add( comp:surnameField);
```

Fields where the user will enter their username, password, first name and last name are added with their labels.

```
        panel.add(new JLabel( text:"Date of birth:"));
        JPanel datePanel = new JPanel(new FlowLayout( align:FlowLayout.LEFT));
```

A three-part selection (day, month, year) will be presented for the date of birth. These fields are laid out horizontally within a single panel.

```
        dayCombo = new JComboBox<>();
        for (int i = 1; i <= 31; i++) {
            dayCombo.addItem( item:i);
        }
        datePanel.add( comp:dayCombo);

        monthCombo = new JComboBox<>();
        for (int i = 1; i <= 12; i++) {
            monthCombo.addItem( item:i);
        }
        datePanel.add( comp:monthCombo);

        yearCombo = new JComboBox<>();
        for (int i = 1900; i <= 2023; i++) {
            yearCombo.addItem( item:i);
        }
        yearCombo.setSelectedItem( anObject:2000);
        datePanel.add( comp:yearCombo);

        panel.add( comp:datePanel);
```

In this block, three separate combo boxes (JComboBox) are created for the user to select the date of birth:
• Day box (1–31),
• Month box (1–12),
• Year box (1900–2023, 2000 is selected by default).
It allows the user to easily select the date of birth. In the panel called datePanel, these boxes are arranged horizontally and finally added to the main panel (panel).

```
        panel.add(new JLabel( text:"Email:"));
        emailField = new JTextField();
        panel.add( comp:emailField);
```

A label and text box are created for the user to enter their email address. This field of the form collects email information from the user.

```
        panel.add(new JLabel( text:"Home Address:"));
        homeAddressField = new JTextField();
        panel.add( comp:homeAddressField);

        panel.add(new JLabel( text:"Work Address:"));
        workAddressField = new JTextField();
        panel.add( comp:workAddressField);
```

Two separate text boxes are defined for the user's home and work addresses. Address information is taken from these fields.

```
registerButton = new JButton( text:"Sign Up");
panel.add( comp:registerButton);
```

It is the button that allows the user to send the information in the form and start the registration process. It is made visible by adding it to the panel.

```
registerButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        String username = usernameField.getText();
        String password = new String( value:passwordField.getPassword());
        String name = nameField.getText();
        String surname = surnameField.getText();
        int day = (int) dayCombo.getSelectedItem();
        int month = (int) monthCombo.getSelectedItem();
        int year = (int) yearCombo.getSelectedItem();
        String email = emailField.getText();
        String homeAddress = homeAddressField.getText();
        String workAddress = workAddressField.getText();
```

The operations that will be performed when the button is clicked are defined here. All information entered by the user is read from the relevant fields and stored temporarily.

```
        @SuppressWarnings("deprecation")
        Date birthDate = new Date(year - 1900, month - 1, date:day);
```

The selected day, month and year information is combined and converted to a Java Date object. The Date class is a legacy structure, so the warning is suppressed.

```
        User user = new User(username, name, surname, birthDate, password, email, homeAdress:homeAddress, workAdress:workAddress);
```

With all the information retrieved, a User object is created. This object contains data representing the user and is ready to be sent to the database.

```
        UserDAO userDAO = new UserDAO();
        if (userDAO.addUser(user)) {
            JOptionPane.showMessageDialog( parentComponent:RegistrationFrame.this, message:"Registration successful!");
            dispose();
        } else {
            JOptionPane.showMessageDialog( parentComponent:RegistrationFrame.this, message:"An error occurred while registering!", title:"Error",
                    messageType:JOptionPane.ERROR_MESSAGE);
        }
    }
});
```

The UserDAO class is used to attempt to save user information to the database. If successful, an information message is displayed and the window closes. If an error occurs, the user is informed with a warning window.

```
        add( comp:panel);
    }
}
```

The panel containing all form elements is added to the JFrame to complete the interface.

4. SellerDashboard.java

This code creates a management panel where sellers can add products to the system. Product information (name, color, category, stock, price, description, weight) is received from the user and saved to the database. It also offers the opportunity to end the session with the "Log Outt" option in the menu bar. It is a simple and functional interface created with Swing components.

```
public class SellerDashboard extends JFrame {
    private JTextField nameField, colorField, categoryField, stockField, priceField, productWeightField;
    private JTextArea descriptionArea;
    private JButton addProductButton;
```

The SellerDashboard class is derived from the JFrame class and represents the seller interface.

```
    public SellerDashboard() {
        setTitle( title:"Seller Dashboard");
        setSize( width:600, height:500);
        setDefaultCloseOperation( operation:JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo( c:null);
```

The necessary form fields for product information and an add button are defined.

```
        JMenuBar menuBar = new JMenuBar();
        JMenu sellerMenu = new JMenu( s:"Account");
        JMenuItem logoutItem = new JMenuItem( text:"Log out");

        logoutItem.addActionListener(e -> logout());
        sellerMenu.add( menuItem:logoutItem);
        menuBar.add( c:sellerMenu);
        setJMenuBar( menubar:menuBar);
```

The panel title, size and opening in the middle of the screen are set. When the window is closed, the application is

completely closed.

```
JPanel mainPanel = new JPanel(new BorderLayout( hgap:10,  vgap:10));
mainPanel.setBorder( border:BorderFactory.createEmptyBorder( top:20,  left:20,  bottom:20,  right:20));

JPanel formPanel = new JPanel(new GridLayout( rows:8,  cols:2,  hgap:10,  vgap:10));
```

A menu named "Account" is added to the top menu bar and a "Log Out" option is added. When clicked, the logout() method is run.
A form field is created in the main panel with border margins and a GridLayout inside. It is designed to have a label and an input box in each row.

```
formPanel.add(new JLabel( text:"Product Name:"));
nameField = new JTextField();
formPanel.add( comp:nameField);

formPanel.add(new JLabel( text:"Colour:"));
colorField = new JTextField();
formPanel.add( comp:colorField);

formPanel.add(new JLabel( text:"Category:"));
categoryField = new JTextField();
formPanel.add( comp:categoryField);

formPanel.add(new JLabel( text:"Stock:"));
stockField = new JTextField();
formPanel.add( comp:stockField);

formPanel.add(new JLabel( text:"Price:"));
priceField = new JTextField();
formPanel.add( comp:priceField);

formPanel.add(new JLabel( text:"Weight:"));
productWeightField = new JTextField();
formPanel.add( comp:productWeightField);
```

In this section, labels and text boxes have been created for the seller to enter basic information about the product: product name, color, category, stock quantity, price and weight.

```
formPanel.add(new JLabel( text:"Description:"));
descriptionArea = new JTextArea( rows:3,  columns:20);
JScrollPane scrollPane = new JScrollPane( view:descriptionArea);
formPanel.add( comp:scrollPane);
```

A multi-line text area is defined where a description of the product can be entered. It is made scrollable with JScrollPane.

```
mainPanel.add( comp:formPanel,  constraints:BorderLayout.CENTER);
```

The product entry form is placed in the middle of the main panel.

```
JPanel buttonPanel = new JPanel(new FlowLayout( align:FlowLayout.CENTER,  hgap:10,  vgap:10));
addProductButton = new JButton( text:"Add Product");
buttonPanel.add( comp:addProductButton);

JButton logoutButton = new JButton( text:"Log Out");
logoutButton.addActionListener(e -> logout());
buttonPanel.add( comp:logoutButton);

mainPanel.add( comp:buttonPanel,  constraints:BorderLayout.SOUTH);
```

There are two buttons at the bottom: one starts the product adding process, the other logs out.

```
addProductButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            String name = nameField.getText();
            String color = colorField.getText();
            String category = categoryField.getText();
            int stock = Integer.parseInt( s:stockField.getText());
            double price = Double.parseDouble( s:priceField.getText());
            double weight = Double.parseDouble( s:productWeightField.getText());
            String description = descriptionArea.getText();

            Product product = new Product( productName:name,  productColor:color, category,  productStock:stock,  productPrice:price, description,  productWeight:weight);
            product.setProductWeight( productWeight:weight);

            ProductDAO productDAO = new ProductDAO();
            if (productDAO.addProduct(product)) {
                JOptionPane.showMessageDialog( parentComponent:SellerDashboard.this,  message:"Product added successfully!");
                clearFields();
            } else {
                JOptionPane.showMessageDialog( parentComponent:SellerDashboard.this,  message:"An error occurred while adding the product!",  title:"Error",
                        messageType:JOptionPane.ERROR_MESSAGE);
            }
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog( parentComponent:SellerDashboard.this,  message:"Geçersiz sayı formatı!",  title:"Hata",
                    messageType:JOptionPane.ERROR_MESSAGE);
        }
    }
});
```

The logoutButton is also connected to the logout() method.
When the “Add Product” button is clicked, the entered information is read, the Product object is created and saved to the database with ProductDAO. If numeric fields are entered incorrectly, the user is warned. After a successful transaction, the form is cleared.

```
add( comp:mainPanel);
}
```

The mainPanel, which contains all the components, is added to the JFrame (main window) and the interface is literally

placed on the screen.

```
    private void logout() {
        int confirm = JOptionPane.showConfirmDialog(
            parentComponent: this,
            message: "Are you sure you want to log out?",
            title: "Exit Confirmation",
            optionType: JOptionPane.YES_NO_OPTION
        );

        if (confirm == JOptionPane.YES_OPTION) {
            dispose();
            new LoginFrame().setVisible( b: true);
        }
    }
```

When the user clicks the “Log Out” button, a confirmation box appears. If “Yes” is selected, the screen closes and the LoginFrame (login screen) opens again.

```
    private void clearFields() {
        nameField.setText( t: "");
        colorField.setText( t: "");
        categoryField.setText( t: "");
        stockField.setText( t: "");
        priceField.setText( t: "");
        productWeightField.setText( t: "");
        descriptionArea.setText( t: "");
    }
}
```

It ensures that the form fields are cleared after a new product is added. So the user is presented with a blank form when adding a new product.

5. UserDashboard.java

This class is a user panel where the logged-in user can view products in the e-commerce system, place orders, add products to their favorites, and manage card information. The interface is divided into tabs using JTabbedPane. Orders, products, favorites, and cards are presented to the user in separate tabs.

```
public class UserDashboard extends JFrame {
    private String username;
    private User currentUser;
    private JTabbedPane tabbedPane;
```

Username and user information are stored here. A JTabbedPane is created because all content will be displayed with tabs.

```
    private ProductDAO productDAO = new ProductDAO();
    private OrderDAO orderDAO = new OrderDAO();
    private CreditCardDAO creditCardDAO = new CreditCardDAO();
    private FavoriteDAO favoriteDAO = new FavoriteDAO();
```

DAO objects are defined to manage database transactions for products, orders, credit cards, and favorite products.

```
    public UserDashboard(String username) {
        this.username = username;
        this.currentUser = new UserDAO().getUserByUsername(username);

        setTitle("User Panel - " + username);
        setSize( width: 800,  height: 600);
        setDefaultCloseOperation( operation: JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo( c: null);
```

The information of the logged-in user is retrieved from the database using UserDAO. The title, size and opening of the window in the middle of the screen are provided. With this structure, a personal panel is presented to the user.

```
        JMenuBar menuBar = new JMenuBar();
        JMenu userMenu = new JMenu( s: "Account");
        JMenuItem logoutItem = new JMenuItem( text: "Log off");
        logoutItem.addActionListener(e -> logout());
        userMenu.add( menuItem: logoutItem);
        menuBar.add( c: userMenu);
        setJMenuBar( menubar: menuBar);
```

A menu bar is created for the user to log out. Under the "Account" menu, there is a "Log off" option. If clicked, the logout() method is called and the user session is closed.

```
        tabbedPane = new JTabbedPane();

        JPanel productsPanel = createProductsPanel();
        tabbedPane.addTab( title: "Products",  component: productsPanel);

        JPanel favoritesPanel = createFavoritesPanel();
        tabbedPane.addTab( title: "Favourites",  component: favoritesPanel);

        JPanel ordersPanel = createOrdersPanel();
        tabbedPane.addTab( title: "Orders",  component: ordersPanel);

        JPanel cardsPanel = createCreditCardsPanel();
        tabbedPane.addTab( title: "Credit Cards",  component: cardsPanel);

        add( comp: tabbedPane);
    }
```

Using JTabbedPane, 4 different tabs are added to the user panel: Products, Favorites, Orders and Credit Cards. (The Cart tab is in the comments.) Each tab contains a JPanel for its own function.

```
private JPanel createProductsPanel() {
    JPanel panel = new JPanel(new BorderLayout());
```

The main panel is created for the tab. BorderLayout is used because the contents will be divided vertically into three.

```
JComboBox<String> categoryCombo = new JComboBox<>(new String[]{"All", "Electronics", "Clothing", "Books", "Sports", "Home"});
JButton filterButton = new JButton( text:"Filter");
JPanel filterPanel = new JPanel(new FlowLayout( align:FlowLayout.LEFT));
filterPanel.add(new JLabel( text:"Category:"));
filterPanel.add( comp:categoryCombo);
filterPanel.add( comp:filterButton);
panel.add( comp:filterPanel,  constraints:BorderLayout.NORTH);
```

A drop-down menu (JComboBox) and a filter button are added to filter products by category. All these components are placed in the NORTH position.

```
DefaultListModel<Product> productListModel = new DefaultListModel<>();
JList<Product> productList = new JList<>( dataModel:productListModel);
productList.setFixedCellHeight( height:60);
productList.setCellRenderer(new ProductListRenderer());
JScrollPane scrollPane = new JScrollPane( view:productList);
panel.add( comp:scrollPane,  constraints:BorderLayout.CENTER);
```

A JList component is defined to list products. A ProductListRenderer is assigned for a special visual layout and is enclosed in a JScrollPane for scrolling support.

```
JPanel buttonPanel = new JPanel(new FlowLayout( align:FlowLayout.LEFT));
JButton orderButton = new JButton( text:"Order");
JButton addFavoriteButton = new JButton( text:"Add to favourites");
JButton logoutButton = new JButton( text:"Log off");

buttonPanel.add( comp:orderButton);
buttonPanel.add( comp:addFavoriteButton);
buttonPanel.add( comp:logoutButton);
panel.add( comp:buttonPanel,  constraints:BorderLayout.SOUTH);
```

Three buttons are created for ordering, adding to favorites, and exiting. All buttons are aligned in the SOUTH section.

```
filterButton.addActionListener(e -> {
    String category = (String) categoryCombo.getSelectedItem();
    List<Product> products = category.equals( anObject:"All") ?
        productDAO.getAllProducts() : productDAO.getProductsByCategory(category);

    productListModel.clear();
    products.forEach(productListModel::addElement);
});
```

When the filter button is clicked, products are brought to the selected category and the list is updated.

```
orderButton.addActionListener(e -> {
    Product selected = productList.getSelectedValue();
    if (selected == null) {
        JOptionPane.showMessageDialog( parentComponent:this,  message:"Please select a product");
        return;
    }

    List<CreditCard> cards = creditCardDAO.getCreditCardsByUser( userId:currentUser.getUserId());
    if (cards.isEmpty()) {
        JOptionPane.showMessageDialog( parentComponent:this,  message:"You must add a credit card before ordering.");
        return;
    }

    CreditCard selectedCard = (CreditCard) JOptionPane.showInputDialog(
        parentComponent:this,  message:"Select Credit Card:",  title:"Card Selection",
        messageType:JOptionPane.QUESTION_MESSAGE,  icon:null,
        selectionValues:cards.toArray(),  initialSelectionValue:cards.get( index:0));

    if (selectedCard != null) {
        Order order = new Order( user:currentUser,  product:selected,  creditCard:selectedCard,  quantity:1);
        if (orderDAO.addOrder(order) && productDAO.updateProductStock( productId:selected.getProductId(),  quantity:1)) {
            JOptionPane.showMessageDialog( parentComponent:this,  message:"Order created successfully.");
            filterButton.doClick(); // Listeyi yenile
        } else {
            JOptionPane.showMessageDialog( parentComponent:this,  message:"Order could not be created.",  title:"Error",  messageType:JOptionPane.ERROR_MESSAGE);
        }
    }
});
```

This process is started when the user clicks the “Order” button. First, it is checked whether the user has selected a product from the list. If the product is not selected, the user is warned and the process is canceled.
Then, in order for the order to be completed, it is checked whether the user has at least one credit card registered in the system. If there is no registered card, the order process is stopped.
If there is a card, the user is asked to select one of these cards (via a pop-up window). If the card selection is successful, the Order object is created and saved to the database via OrderDAO.
At the same time, the stock of the ordered product is reduced by 1 (updateProductStock). If the transactions are successful, the user is shown the message “Order created successfully” and the product list is updated via the filter button. If the order or stock update process fails, the user is given an error message.

```
addFavoriteButton.addActionListener(e -> {
 Product selected = productList.getSelectedValue();
if (selected == null) {
    JOptionPane.showMessageDialog( parentComponent:this,  message:"Please select a product.");
return;
}

if (favoriteDAO.addFavorite( userId:currentUser.getUserId(),  productId:selected.getProductId())) {
    JOptionPane.showMessageDialog( parentComponent:this,  message:"Product added to favorites.");
    refreshFavoritesTab();
} else {
    JOptionPane.showMessageDialog( parentComponent:this,  message:"This product is already in your favorites!",  title:"Warning",  messageType:JOptionPane.WARNING_MESSAGE);
}
});
```

This block is executed when the user clicks on the “Add to favorites” button. First, it is checked whether the user has selected a product from the list. If no product is selected, the user is warned and the process is terminated.
If a product is selected, this product is sent to the FavoriteDAO.addFavorite() method to be added to the favorites. This process is performed with the user ID and product ID. If the product has not been added to the favorites before, a success message is displayed and the favorites tab is updated by calling the refreshFavoritesTab() method. In this way, the user can immediately see the added product in the favorites.
However, if the product is already in the favorites list, the system does not allow it to be added again and a warning message (“This product is already in your favorites!”) is displayed to inform the user. This is a control mechanism that improves the user experience.

```
logoutButton.addActionListener(e -> logout());
```

When the “Log off” button is clicked, the logout process is triggered.

```
filterButton.doClick();

return panel;
}
```

When the panel is first created, the filter is automatically run to list all products and the panel is returned.

```
private JPanel createFavoritesPanel() {
    JPanel panel = new JPanel(new BorderLayout());
    DefaultListModel<Product> model = new DefaultListModel<>();
    JList<Product> favoritesList = new JList<>( dataModel: model);
    favoritesList.setCellRenderer(new ProductListRenderer());
```

A JPanel is created to display favorite products. In this panel, products are displayed in a specially visualized JList. ProductListRenderer is used for the appearance.

```
favoriteDAO.getFavoriteProductIds( userId: currentUser.getUserId()).forEach(id -> {
    Product p = productDAO.getProductById( productId: id);
    if (p != null) model.addElement( element: p);
});
```

The IDs of the user's favorite products are retrieved from the database and each product is retrieved in detail with ProductDAO. The retrieved products are added to the list (model) and displayed on the screen.

```
JScrollPane scrollPane = new JScrollPane( view: favoritesList);
panel.add( comp: scrollPane,  constraints: BorderLayout.CENTER);
```

The product list is enclosed in a panel with a scroll bar so that the user can easily browse the list even if there are many favorites.

```
JPanel buttonPanel = new JPanel(new FlowLayout( align: FlowLayout.LEFT));
JButton removeButton = new JButton( text: "Remove from Favorites");
JButton orderButton = new JButton( text: "Order");
```

Two buttons are added so that the user can delete products from the favorite list and place an order: one for “Remove from Favorite” and the other for “Place Order”.

```
removeButton.addActionListener(e -> {
    Product selected = favoritesList.getSelectedValue();
    if (selected != null && favoriteDAO.removeFavorite( userId: currentUser.getUserId(),  productId: selected.getProductId())) {
        model.removeElement( obj: selected);
        JOptionPane.showMessageDialog( parentComponent: this,  message: "Product removed from favorites");
    }
});
```

If the user selects a favorite product and clicks the "Remove from Favorites" button, the product is removed from the database and instantly deleted from the list. After the successful operation, the user is shown an information message.

```
orderButton.addActionListener(e -> {
    Product selected = favoritesList.getSelectedValue();
    if (selected != null) {
        List<CreditCard> cards = creditCardDAO.getCreditCardsByUser( userId: currentUser.getUserId());
        if (cards.isEmpty()) {
            JOptionPane.showMessageDialog( parentComponent: this,  message: "You must add a credit card before ordering.");
            return;
        }

        CreditCard selectedCard = (CreditCard) JOptionPane.showInputDialog(
                parentComponent: this,  message: "Select Credit Card:",  title: "Card Selection",
                messageType: JOptionPane.QUESTION_MESSAGE,  icon: null,
                selectionValues: cards.toArray(),  initialSelectionValue: cards.get( index: 0));

        if (selectedCard != null) {
            Order order = new Order( user: currentUser,  product: selected,  creditCard: selectedCard,  quantity: 1);
            if (orderDAO.addOrder(order) && productDAO.updateProductStock( productId: selected.getProductId(),  quantity: 1)) {
                JOptionPane.showMessageDialog( parentComponent: this,  message: "Order created successfully");
```

This code is executed when the user clicks on the “Order” button to order a product while in the favorites tab. First, it is checked whether a product is selected from the list. If it is not selected, the user is warned. If the product is selected, the

user’s registered credit cards are retrieved from the database. If there is no card, the order process is stopped and the user is informed. If there is, a window is opened for the user to select a card. When the card is selected, an order object is created and added to the database. At the same time, the product’s stock information is decreased by 1. If all operations are successful, the user is shown a success message, otherwise an error message is given.

```
                    model.clear();
                    favoriteDAO.getFavoriteProductIds( userId: currentUser.getUserId()).forEach(id -> {
                        Product p = productDAO.getProductById( productId: id);
                        if (p != null) model.addElement( element: p);
                    });
                } else {
                    JOptionPane.showMessageDialog( parentComponent: this,  message: "Order could not be created",  title: "Error",  messageType: JOptionPane.ERROR_MESSAGE);
                }
            }
        }
    });
```

After the order, the list is cleared and reloaded to keep the favorite list up to date. In this way, the user always sees the current favorite list. If the order or stock update process fails, the user is shown an error message.

```
        buttonPanel.add( comp: removeButton);
        buttonPanel.add( comp: orderButton);
        panel.add( comp: buttonPanel,  constraints: BorderLayout.SOUTH);

        return panel;
    }
```

The buttons are placed at the bottom of the panel (SOUTH). When the method is completed, the panel for this tab is returned.

```
    private JPanel createOrdersPanel() {
        JPanel panel = new JPanel(new BorderLayout());
        DefaultListModel<String> model = new DefaultListModel<>();
        JList<String> ordersList = new JList<>( dataModel: model);
```

A panel and JList are created to show the order history. Orders will be added to the list in text format.

```
        orderDAO.getOrdersByUser( userId: currentUser.getUserId()).forEach(order -> {
            model.addElement( element: String.format( format: "Order #%d - %s - %s",
                args: order.getOrderId(),
                args: order.getProduct().getProductName(),
                args: order.getOrderDate()));
        });
```

All orders of the logged in user are received via orderDAO. Each order is added to the list by formatting it with the order number, product name and date.

```
        JScrollPane scrollPane = new JScrollPane( view: ordersList);
        panel.add( comp: scrollPane,  constraints: BorderLayout.CENTER);

        return panel;
    }
```

The list is enclosed in a scrollable JScrollPane and placed in the center of the panel, allowing for easy navigation of long lists. The created order panel is returned to be added to the tab.

```
    private JPanel createCreditCardsPanel() {
        JPanel panel = new JPanel(new BorderLayout());
        DefaultListModel<CreditCard> model = new DefaultListModel<>();
        JList<CreditCard> cardsList = new JList<>( dataModel: model);
```

This tab is used to list the user's registered credit cards and add new cards. Cards are kept in a list (JList).

```
        creditCardDAO.getCreditCardsByUser( userId: currentUser.getUserId()).forEach(model::addElement);
```

All credit cards registered in the system are retrieved from the database and added to the list.

```
        cardsList.setCellRenderer(new DefaultListCellRenderer() {
            @Override
            public Component getListCellRendererComponent(JList<?> list, Object value, int index,
                                                          boolean isSelected, boolean cellHasFocus) {
                super.getListCellRendererComponent(list, value, index, isSelected, cellHasFocus);
                if (value instanceof CreditCard) {
                    CreditCard card = (CreditCard) value;
                    setText( text: String.format( format: "**** **** **** %s - %s",
                        args: card.getCardNumber().substring( beginIndex: 12),
                        args: card.getExpDate()));
                }
                return this;
            }
        });
```

The cards are specially formatted: only the last 4 digits and the expiration date are displayed, ensuring a secure and user-friendly display.

```
        JScrollPane scrollPane = new JScrollPane( view: cardsList);
        panel.add( comp: scrollPane,  constraints: BorderLayout.CENTER);
```

The card list is placed in the center of the panel with scrolling support.

```
        JPanel buttonPanel = new JPanel(new FlowLayout( align: FlowLayout.LEFT));
        JButton addButton = new JButton( text: "Add Credit Card");
```

A button is placed at the bottom so that the user can add a new card.

```
        addButton.addActionListener(e -> {
            JTextField cardNumberField = new JTextField( columns:16);
            JTextField securityCodeField = new JTextField( columns:3);
            JTextField expMonthField = new JTextField( columns:2);
            JTextField expYearField = new JTextField( columns:4);

            JPanel inputPanel = new JPanel(new GridLayout( rows:4,  cols:2));
            inputPanel.add(new JLabel( text:"Credit Card Number:"));
            inputPanel.add( comp:cardNumberField);
            inputPanel.add(new JLabel( text:"Security Code:"));
            inputPanel.add( comp:securityCodeField);
            inputPanel.add(new JLabel( text:"Expiration Month:"));
            inputPanel.add( comp:expMonthField);
            inputPanel.add(new JLabel( text:"Expiration Year:"));
            inputPanel.add( comp:expYearField);

            int result = JOptionPane.showConfirmDialog(
                parentComponent:this,  message:inputPanel,  title:"New Credit Card",
                optionType:JOptionPane.OK_CANCEL_OPTION,  messageType:JOptionPane.PLAIN_MESSAGE);
```

A form opens where the user can enter the card number, security code and expiration date information. The form is presented to the user in the form of a dialog box.

```
            if (result == JOptionPane.OK_OPTION) {
                try {
                    String cardNumber = cardNumberField.getText().replaceAll( regex:"\\s+",  replacement:"");
                    String securityCode = securityCodeField.getText();
                    int month = Integer.parseInt( s:expMonthField.getText());
                    int year = Integer.parseInt( s:expYearField.getText());

                    if (cardNumber.length() != 16 || !cardNumber.matches( regex:"\\d+")) {
                        JOptionPane.showMessageDialog( parentComponent:this,  message:"Invalid card number",  title:"Error",  messageType:JOptionPane.ERROR_MESSAGE);
                        return;
                    }

                    if (securityCode.length() != 3 || !securityCode.matches( regex:"\\d+")) {
                        JOptionPane.showMessageDialog( parentComponent:this,  message:"Invalid security code",  title:"Error",  messageType:JOptionPane.ERROR_MESSAGE);
                        return;
                    }

                    @SuppressWarnings("deprecation")
                    Date expDate = new Date(year - 1900, month - 1,  date:1);

                    CreditCard card = new CreditCard(cardNumber,  user:currentUser, securityCode, expDate);
                    if (creditCardDAO.addCreditCard(card)) {
                        model.addElement( element:card);
                        JOptionPane.showMessageDialog( parentComponent:this,  message:"Card added successfully");
                    }
                } catch (NumberFormatException ex) {
                    JOptionPane.showMessageDialog( parentComponent:this,  message:"Invalid date format!",  title:"Error",  messageType:JOptionPane.ERROR_MESSAGE);
                }
            }
        });
```

This code block is executed when the user fills out the credit card addition form and presses the “OK” button. First, the data received from the user, such as the card number and security code, are cleaned and verified. The card number must be 16 digits and consist of only numbers, and the security code must be 3 digits. If there is an error in the entries, the transaction is stopped and the user is warned.
If the data is entered in the correct format, the expiration date is taken as month/year and a CreditCard object is created. This card is added to the database via CreditCardDAO. If the transaction is successful, the user is notified and the card is added to the list. If an error is made in the numerical data entry, the system catches it and displays an appropriate error message to the user. In this way, both data integrity and user experience are preserved.

```
        buttonPanel.add( comp:addButton);
        panel.add( comp:buttonPanel,  constraints:BorderLayout.SOUTH);

        return panel;
    }
```

The “Add Credit Card” button is added to the previously created buttonPanel. This panel is placed at the bottom of the entire credit card list (BorderLayout.SOUTH). Thus, the user can add cards whenever they want. Finally, the panel containing all the components is returned and made ready to be added to the tab structure.

```
    private void logout() {
        int confirm = JOptionPane.showConfirmDialog(
            parentComponent:this,  message:"Are you sure you want to log out?",
            title:"Exit Confirmation",  optionType:JOptionPane.YES_NO_OPTION);

        if (confirm == JOptionPane.YES_OPTION) {
            dispose();
            new LoginFrame().setVisible( b:true);
        }
    }
```

This method is called when the user selects the “Log off” option. First, the user is asked to log out. A confirmation box is displayed on the screen with JOptionPane. If the user selects the “Yes” option, the current user panel (dispose()) is closed and the login screen (LoginFrame) is made visible again. This structure allows the user to log out and re-login to the system safely.

```
    private class ProductListRenderer extends JPanel implements ListCellRenderer<Product> {
    private JLabel nameLabel;
    private JLabel detailsLabel;
    private JLabel priceLabel;
    private JLabel stockLabel;
    private JPanel infoPanel;
```

This custom class determines how each item in the product list will appear. JLabel components display the product name, category, price, and availability.

```
    public ProductListRenderer() {
        setLayout(new BorderLayout( hgap:10,  vgap:5));
        setBorder( border:BorderFactory.createEmptyBorder( top:5,  left:10,  bottom:5,  right:10));
```

List items are arranged horizontally. Left and right margins are left to prevent content from sticking together.

```
        nameLabel = new JLabel();
        nameLabel.setFont(new Font( name:"SansSerif",  style:Font.BOLD,  size:14));

        detailsLabel = new JLabel();
        detailsLabel.setFont(new Font( name:"SansSerif",  style:Font.PLAIN,  size:12));
        detailsLabel.setForeground( fg:Color.GRAY);
        priceLabel = new JLabel();
        priceLabel.setFont(new Font( name:"SansSerif",  style:Font.BOLD,  size:14));
        priceLabel.setForeground(new Color( r:0,  g:100,  b:0));

        stockLabel = new JLabel();
        stockLabel.setFont(new Font( name:"SansSerif",  style:Font.PLAIN,  size:12));
```

Price is displayed prominently in dark green. A separate label is used for stock availability.

```
        infoPanel = new JPanel(new GridLayout( rows:2,  cols:1));
        infoPanel.add( comp:nameLabel);
        infoPanel.add( comp:detailsLabel);

        add( comp:infoPanel,  constraints:BorderLayout.WEST);
        add( comp:priceLabel,  constraints:BorderLayout.CENTER);
        add( comp:stockLabel,  constraints:BorderLayout.EAST);
    }
```

The product name and category are aligned on the left, the price in the middle, and the stock status on the right. This structure provides the user with easy and organized information.

```
    @Override
    public Component getListCellRendererComponent(JList<? extends Product> list, Product product,
                                                  int index, boolean isSelected, boolean cellHasFocus) {

        nameLabel.setText( text:product.getProductName());
        detailsLabel.setText( text:product.getCategory());
        priceLabel.setText( text:String.format( format:"₺%.2f",  args:product.getProductPrice()));

        if (product.getProductStock() > 10) {
            stockLabel.setText( text:"In Stock");
            stockLabel.setForeground(new Color( r:0,  g:120,  b:0));
        } else if (product.getProductStock() > 0) {
            stockLabel.setText("Low Stock (" + product.getProductStock() + ")");
            stockLabel.setForeground(new Color( r:200,  g:120,  b:0));
        } else {
            stockLabel.setText( text:"Out of Stock");
            stockLabel.setForeground( fg:Color.RED);
        }
```

Message and color are determined according to the product stock amount: Green: In stock. Orange: Low stock. Red: Out of stock.

```
        if (isSelected) {
            setBackground( bg:list.getSelectionBackground());
            setForeground( fg:list.getSelectionForeground());
            nameLabel.setForeground( fg:list.getSelectionForeground());
            priceLabel.setForeground( fg:list.getSelectionForeground());
        } else {
            setBackground( bg:list.getBackground());
            setForeground( fg:list.getForeground());
            nameLabel.setForeground( fg:list.getForeground());
            priceLabel.setForeground(new Color( r:0,  g:100,  b:0));
        }
```

When the user selects a product, the background and text color of that row changes, thus highlighting the selected item.

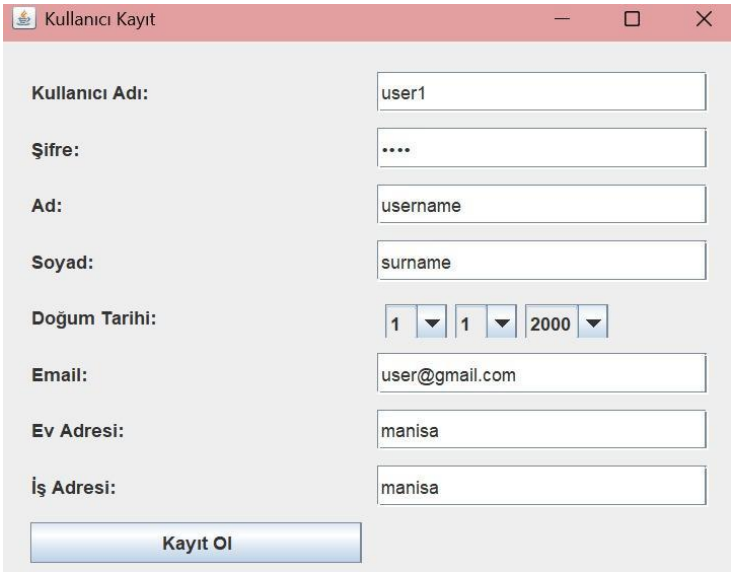
```
        setBorder( border:BorderFactory.createCompoundBorder(
            outsideBorder:BorderFactory.createMatteBorder( top:0,  left:0,  bottom:1,  right:0, new Color( r:220,  g:220,  b:220)),
            insideBorder:BorderFactory.createEmptyBorder( top:8,  left:10,  bottom:8,  right:10)
        ));

        return this;
    }
}
```

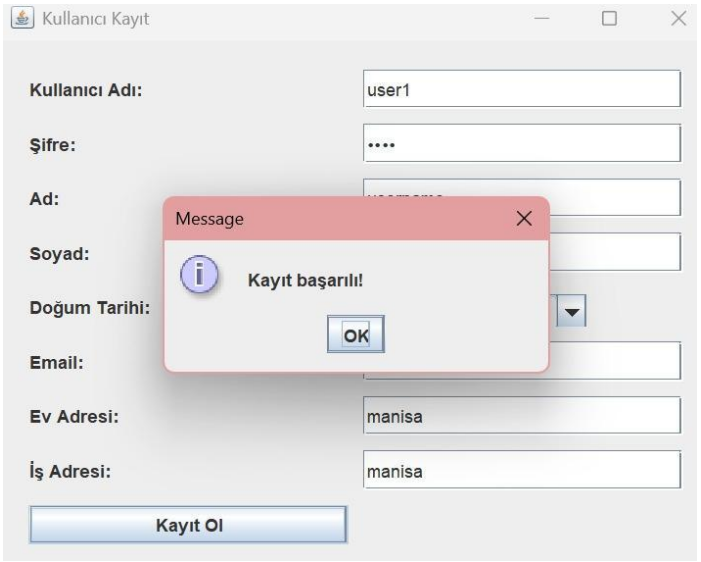
A thin gray line is drawn under the list items to provide a separating appearance. Contents are displayed regularly without being compressed with top and bottom spaces.

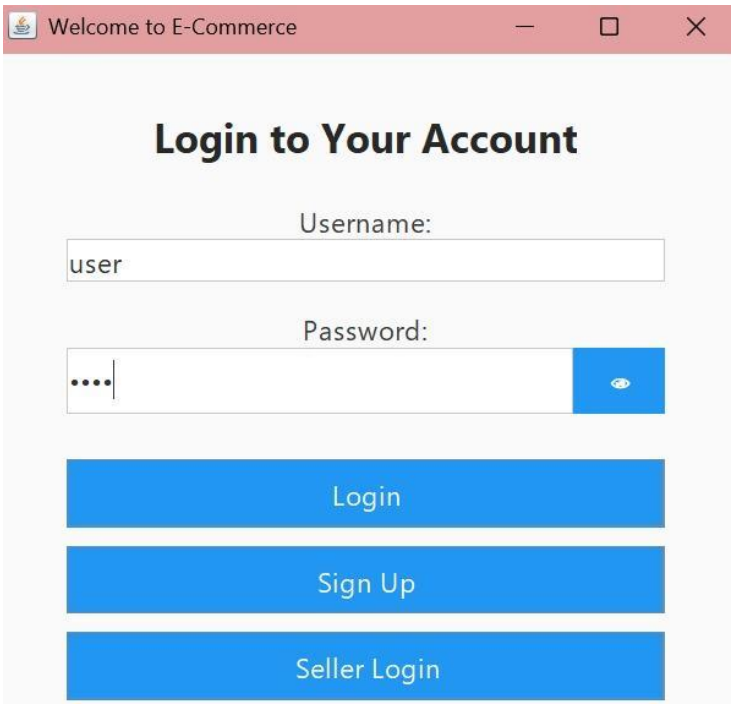
```
    private void refreshFavoritesTab() {
    JPanel favoritesPanel = createFavoritesPanel();
    tabbedPane.remove( index:1); // Favoriler sekmesinin indeksi
    tabbedPane.insertTab( title:"Favourites",  icon:null,  component:favoritesPanel,  tip:null,  index:1);
    tabbedPane.setSelectedIndex( index:1); // Kullanıcıyı favoriler sekmesinde tut
    }
}
```

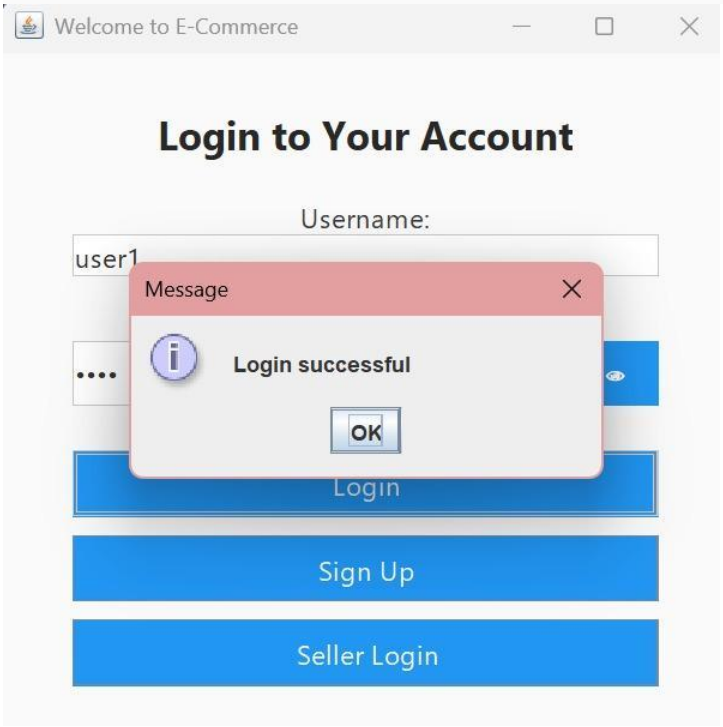
This method is called when a change (addition or deletion) is made to the favorite products. First, the createFavoritesPanel() method is re-run to create a new panel containing the current favorite products. The current favorites tab (index 1) is removed from the tabs and the updated panel is re-added to the same location. Finally, the user remains in this new tab (setSelectedIndex(1)), so the current data is visible without switching between tabs. This approach provides dynamic content updates without disrupting the user experience.

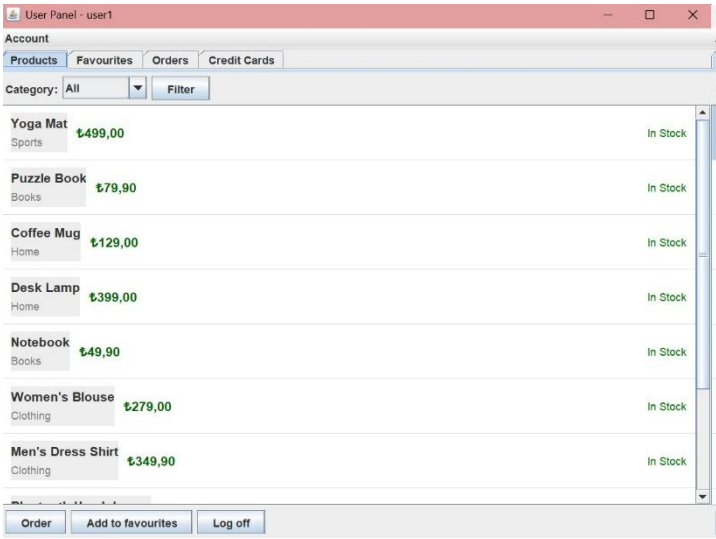
5)Application Screenshots

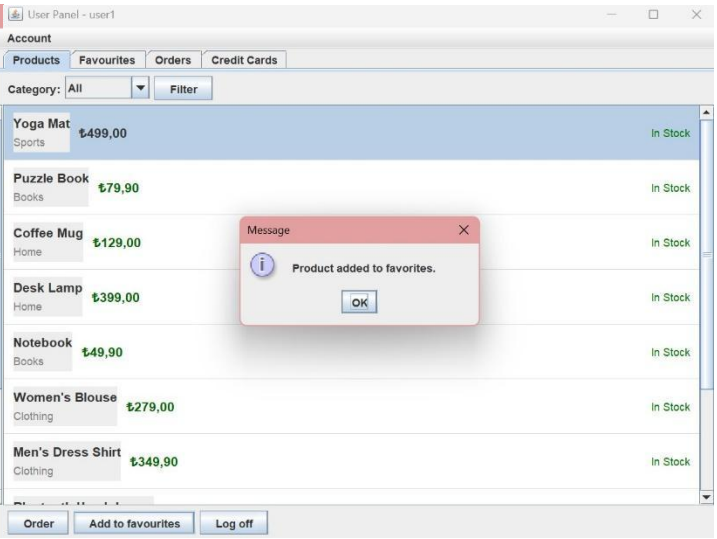


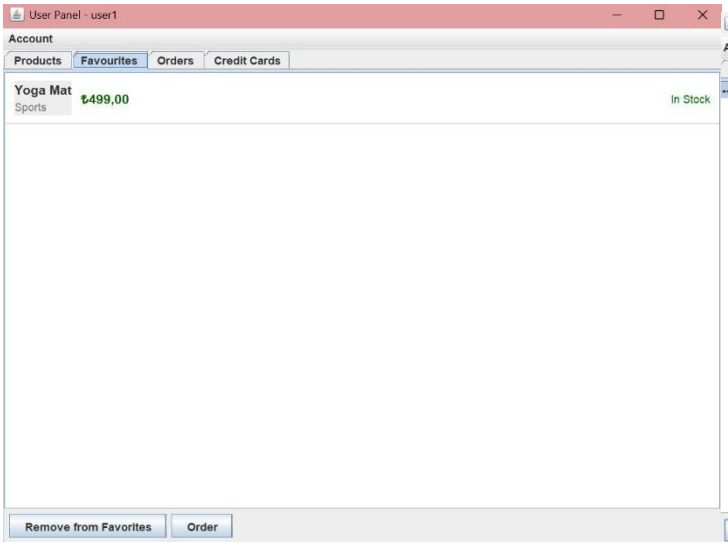


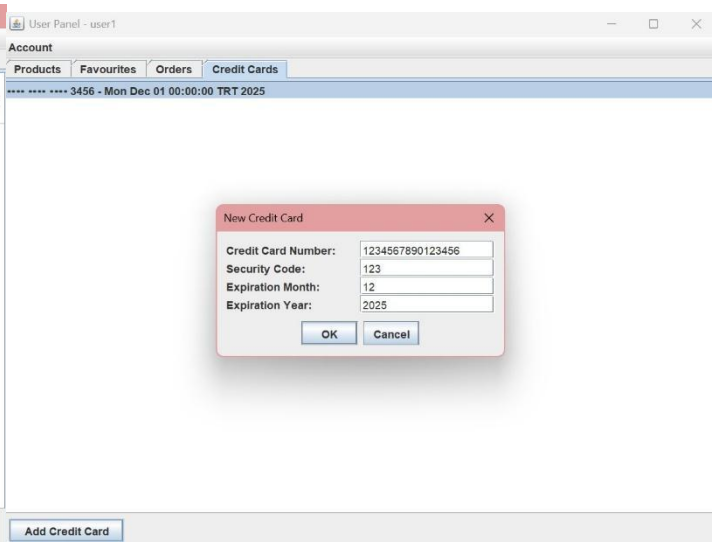

Welcome to E-Commerce
Login to Your Account
Username:
user
Password:
Login
Sign Up
Seller Login

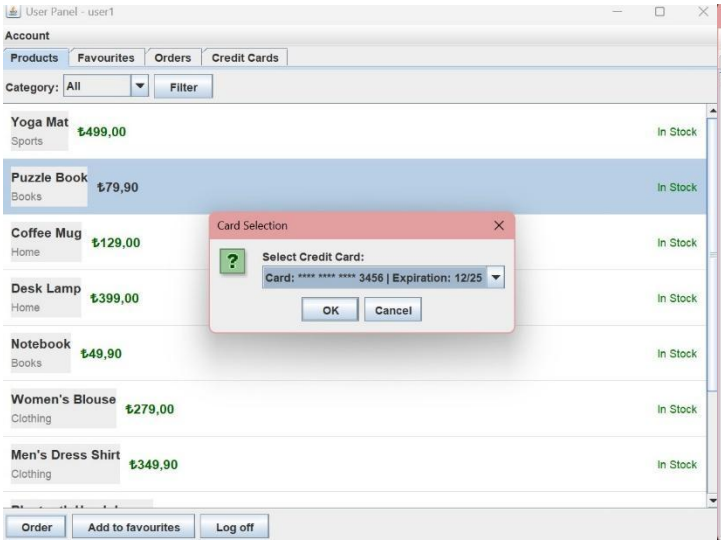

Welcome to E-Commerce
Login to Your Account
Username:
user1
Message
Login successful
OK
Login
Sign Up
Seller Login

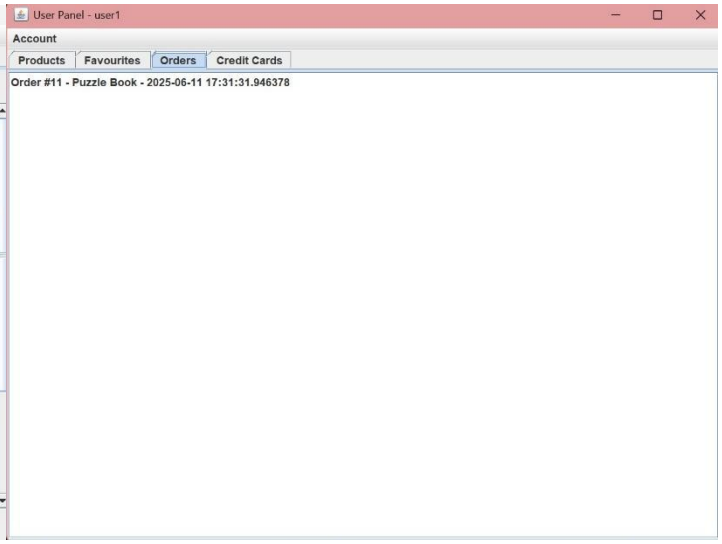

User Panel - user1
Account
Products
Favourites
Orders
Credit Cards
Category: All
Filter
Yoga Mat
Sports
₺499,00
In Stock
Puzzle Book
Books
₺79,90
In Stock
Coffee Mug
Home
₺129,00
In Stock
Desk Lamp
Home
₺399,00
In Stock
Notebook
Books
₺49,90
In Stock
Women's Blouse
Clothing
₺279,00
In Stock
Men's Dress Shirt
Clothing
₺349,90
In Stock
Order
Add to favourites
Log off

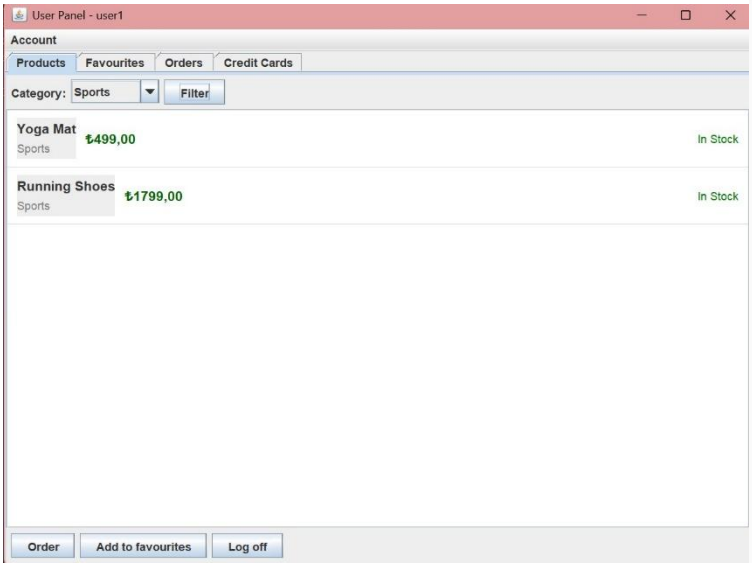

User Panel - user1
Account
Products
Favourites
Orders
Credit Cards
Category: All
Filter
Yoga Mat
Sports
₺499,00
In Stock
Puzzle Book
Books
₺79,90
In Stock
Coffee Mug
Home
₺129,00
In Stock
Desk Lamp
Home
₺399,00
In Stock
Notebook
Books
₺49,90
In Stock
Women's Blouse
Clothing
₺279,00
In Stock
Men's Dress Shirt
Clothing
₺349,90
In Stock
Message
Product added to favorites.
OK
Order
Add to favourites
Log off

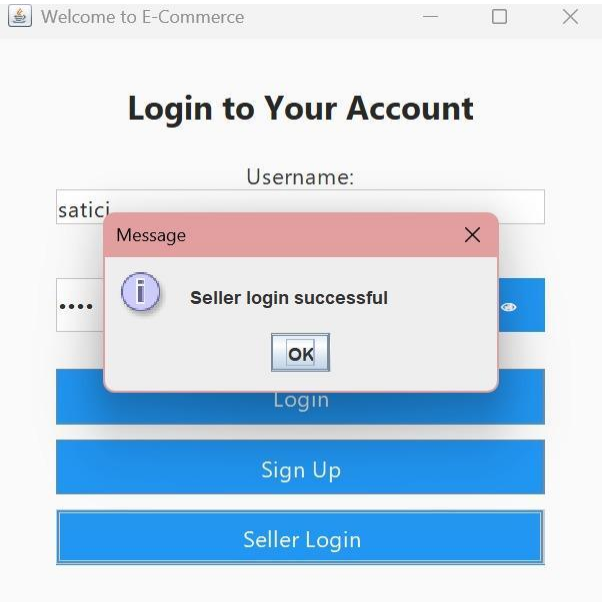

User Panel - user1
Account
Products
Favourites
Orders
Credit Cards
Yoga Mat
Sports
₺499,00
In Stock
Remove from Favorites
Order

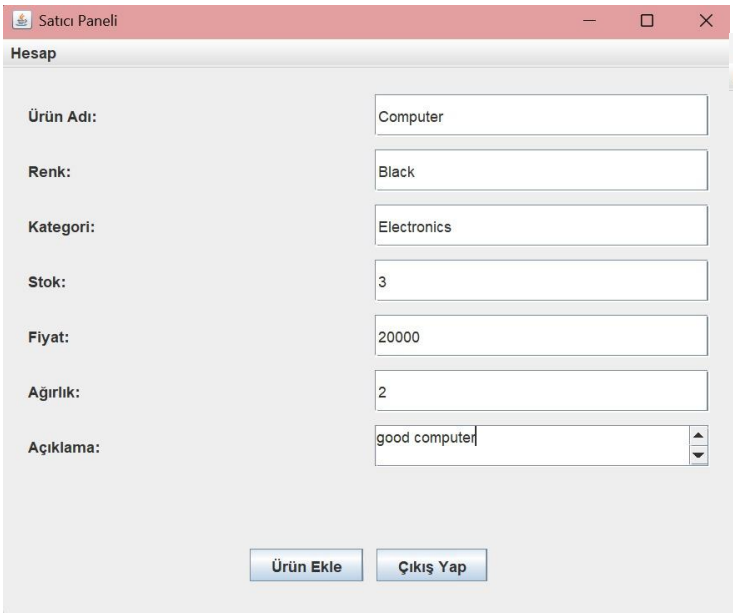

User Panel - user1
Account
Products
Favourites
Orders
Credit Cards
•••• •••• •••• 3456 - Mon Dec 01 00:00:00 TRT 2025
New Credit Card
Credit Card Number:
1234567890123456
Security Code:
123
Expiration Month:
12
Expiration Year:
2025
OK
Cancel
Add Credit Card

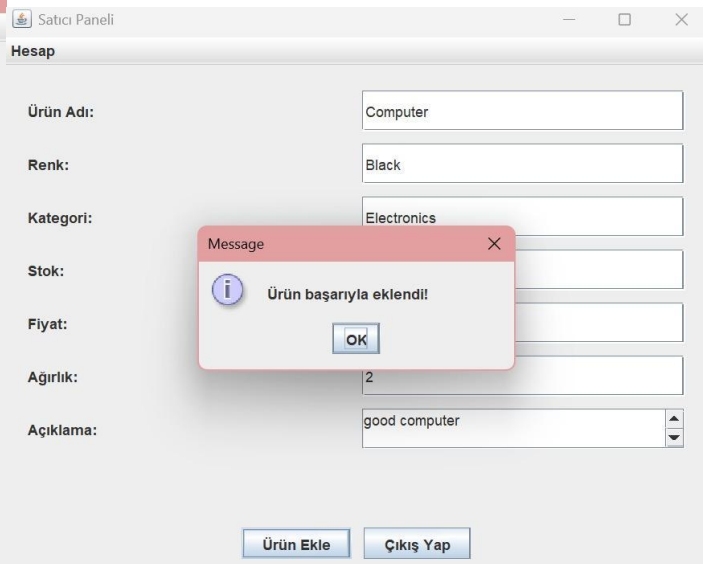

User Panel - user1
Account
Products
Favourites
Orders
Credit Cards
Category: All
Filter
Yoga Mat
Sports
৳499,00
In Stock
Puzzle Book
Books
৳79,90
In Stock
Coffee Mug
Home
৳129,00
In Stock
Desk Lamp
Home
৳399,00
In Stock
Notebook
Books
৳49,90
In Stock
Women's Blouse
Clothing
৳279,00
In Stock
Men's Dress Shirt
Clothing
৳349,90
In Stock
Order
Add to favourites
Log off
Card Selection
Select Credit Card:
Card: **** **** **** 3456 | Expiration: 12/25
OK
Cancel


User Panel - user1
Account
Products
Favourites
Orders
Credit Cards
Order #11 - Puzzle Book - 2025-06-11 17:31:31.946378


User Panel - user1
Account
Products
Favourites
Orders
Credit Cards
Category: Sports
Filter
Yoga Mat
Sports
৳499,00
In Stock
Running Shoes
Sports
৳1799,00
In Stock
Order
Add to favourites
Log off


Welcome to E-Commerce
Login to Your Account
Username:
satici
Message
Seller login successful
OK
Login
Sign Up
Seller Login





Data Output | Messages | Notifications

Showing rows: 1 to 11 | Page No: 1 of 1

	product_id [PK] integer	product_name character varying (100)	product_color character varying (50)	category character varying (50)	product_stock integer	product_price double precision	description text	product_weight double precision
1	3	Men's Dress Shirt	Light Blue	Clothing	60	349.9	Formal office shirt, slim fit	0.2
2	4	Women's Blouse	Pink	Clothing	74	279	Silk blend blouse with collar	0.3
3	11	Bluetooth Headphones	Black	Electronics	75	1299	Wireless over-ear headphones	2
4	12	Smartphone	Silver	Electronics	50	18999	Latest model with 128GB storage	3
5	21	Running Shoes	Blue	Sports	90	1799	Lightweight running shoes	4.4
6	22	Yoga Mat	Purple	Sports	70	499	Non-slip yoga mat	2.1
7	31	Notebook	Yellow	Books	199	49.9	80-page lined notebook	3.4
8	32	Puzzle Book	White	Books	84	79.9	300 puzzles and brain teasers	12.4
9	41	Coffee Mug	Red	Home	120	129	Ceramic mug 350ml capacity	17.1
10	42	Desk Lamp	White	Home	65	399	LED adjustable desk lamp	21.3
11	52	Computer	Black	Electronics	3	20000	good computer	2