Desktop E-commerce

Group 32

Student names :

Adel Waheed Mohamed 23P0213

Seif Tamer Safwat 23P0240

Asaad Ramzy Asaad 23P0228

Ali Ashraf Hamdy 23P0233

GITHUB LINK:

https://github.com/seiftamerr/E-Commerce-Java-Backend

Teams link:

https://engasuedu-my.sharepoint.com/:v:/g/personal/23p0240_eng_asu_edu_eg/EZnN6DyVQsJDjIctLr5_OlQBsq62thH0PzNwVW-ujEp9Dg?e=APuuL7&nav=eyJwbGF5YmFja09wdGlvbnMiOnt9LCJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJTdHJlYW1XZWJBcHAiLCJyZWZlcnJhbE1vZGUiOiJtaXMiLCJyZWZlcnJhbFZpZXciOiJwb3N0cm9sbC1jb3B5bGluayIsInJlZmVycmFsUGxheWJhY2tTZXNzaW9uSWQiOiIzMWY0ZDQ3MC1mZDI0LTRhZWQtODgxMi1kZDQwZmRmMzc5MjYifX0%3D

Classes used (abstract class):

1-User:

-attributes (username ,password ,date of birth,gender,address)

-getters and setters

-Abstract functions like login and registration

2-Admin:

-extends User class

-additional attributes (working hours, roles)

-getters and setters

-implements abstract functions

3-Customer:

-additional attributes (balance, interests)

-extends User class

-getters and setters

-implements abstract functions

4-Category:

-attributes (id, name, description)
-getters and setters

-to String overriding

5-Products:

-attributes (id, name, price, description, stock quantity, category)

-getters and setters

-to String overriding

6-Cart:

-attributes (items, quantities, total price)

- getters and setters

- methods (add item, view cart, calculate total price, clear cart , remove item , check if empty)

7-Order:

-attributes (customer , payment method , cart)

- getters and setters

-method (get receipt)

8-Database:

-attributes (customers, categories, products, orders, carts, admins)

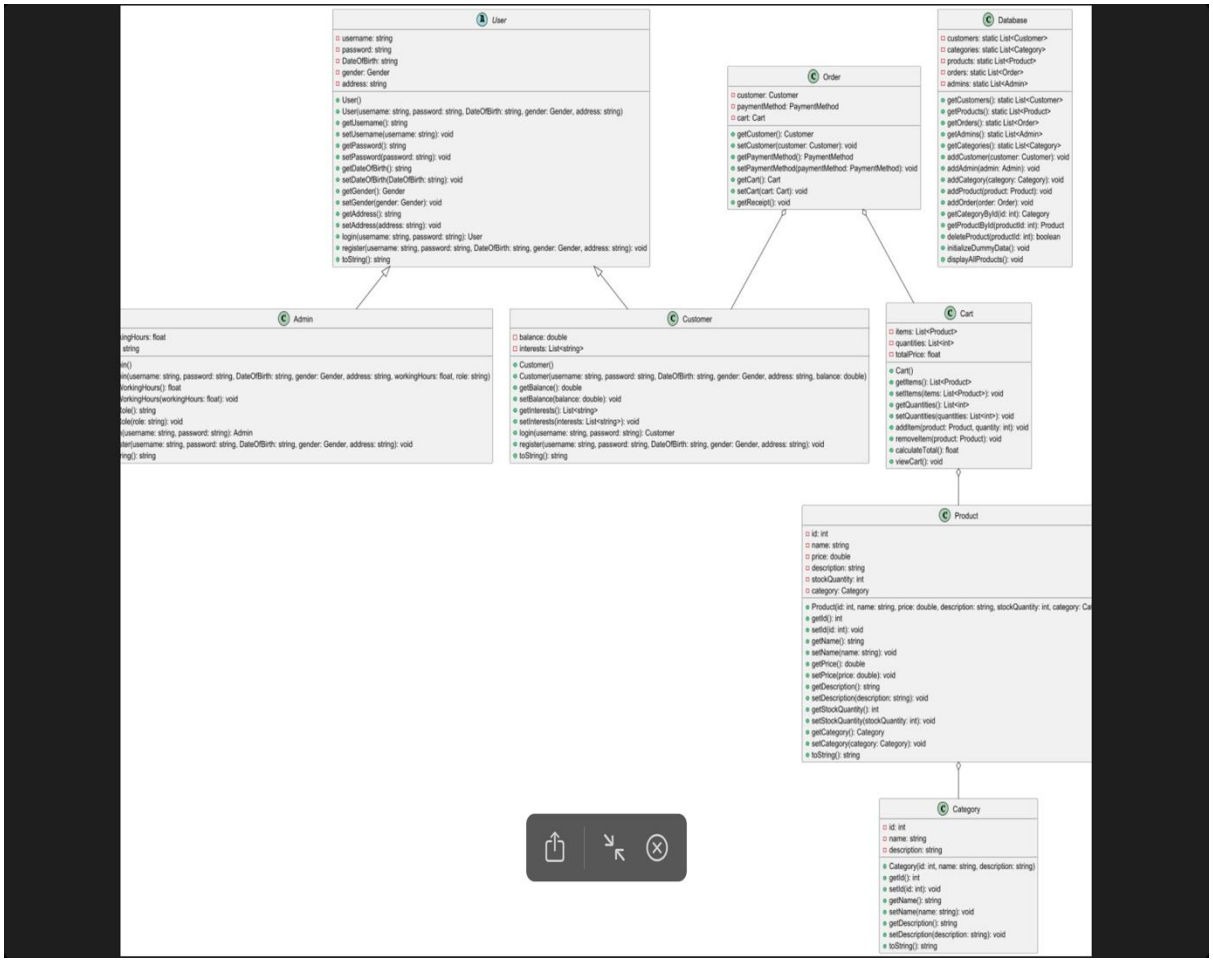
- getters

-methods (add customer, add product, add admin , add order, get category by id, get product by id , delete product)

-initializing dummy data

- Enums (gender, payment method)

UML diagram :



GUI :

Scenes :

1- role selection

Purpose: Allows the user to select their role (Admin, Customer, or Register).

Components:

A Label prompting the user to select their role.

Three Buttons:

Admin Login: Redirects to the login screen for admins.

Customer Login: Redirects to the login screen for customers.

Register: Redirects to the registration screen.

```
private Scene createRoleSelectionScreen(Stage primaryStage) {  1 usage
    VBox layout = new VBox( v: 10);
    layout.setPadding(new Insets( v: 10));

    Label prompt = new Label( s: "Select your role:");
    Button adminLoginButton = new Button( s: "Admin Login");
    Button customerLoginButton = new Button( s: "Customer Login");
    Button registerButton = new Button( s: "Register");

    adminLoginButton.setOnAction(e -> primaryStage.setScene(createLoginScreen(primaryStage,  isAdmin: true)));
    customerLoginButton.setOnAction(e -> primaryStage.setScene(createLoginScreen(primaryStage,  isAdmin: false)));
    registerButton.setOnAction(e -> primaryStage.setScene(createRegistrationScreen(primaryStage)));

    layout.getChildren().addAll(prompt, adminLoginButton, customerLoginButton, registerButton);
    return new Scene(layout,  v: 400,  v1: 200);
}
```

2-Login screen

Purpose: Allows the user to log in based on their role.

Components:

Label for username and password fields.

TextField and PasswordField for input.

Login button to authenticate the user.

Back button to return to the role selection screen.

Admin vs Customer: Controlled by the isAdmin flag.

Validation: Checks credentials in the Database.

```
private Scene createLoginScreen(Stage primaryStage, boolean isAdmin) {  2 usages
    GridPane grid = new GridPane();
    grid.setHgap(10);
    grid.setVgap(10);
    grid.setPadding(new Insets( v: 10));

    Label usernameLabel = new Label( s: "Username:");
    TextField usernameField = new TextField();
    Label passwordLabel = new Label( s: "Password:");
    PasswordField passwordField = new PasswordField();
    Button loginButton = new Button( s: "Login");
    Button backButton = new Button( s: "Back");

    loginButton.setOnAction(e -> {
        String username = usernameField.getText();
        String password = passwordField.getText();

        if (isAdmin) {
            Admin admin = Database.getAdmins().stream()  Stream<Admin>
                    .filter(a -> a.getUsername().equals(username) && a.getPassword().equals(password))
                    .findFirst()  Optional<Admin>
                    .orElse( other: null);

            if (admin != null) {
                adminScene = createAdminDashboard(primaryStage, admin);
                primaryStage.setScene(adminScene);
            } else {
                showAlert("Invalid Admin credentials!");
            }
```

```
        } else {
            Customer customer = Database.getCustomers().stream()  Stream<Customer>
                    .filter(c -> c.getUsername().equals(username) && c.getPassword().equals(password))
                    .findFirst()  Optional<Customer>
                    .orElse( other: null);

            if (customer != null) {
                customerScene = createCustomerDashboard(primaryStage, customer);
                primaryStage.setScene(customerScene);
            } else {
                showAlert("Invalid Customer credentials!");
            }
        }
    });

    backButton.setOnAction(e -> primaryStage.setScene(mainScene));
```

3-Registration screen

Purpose: Allows a new customer to register by filling in required details.

Components:

Labels and input fields (TextField, PasswordField, ComboBox) for user details.

Register button to save the new customer.

Back button to return to the role selection screen.

Validation: Ensures all fields are filled and valid before registration.

```
private Scene createRegistrationScreen(Stage primaryStage) {  1 usage
    GridPane grid = new GridPane();
    grid.setHgap(10);
    grid.setVgap(10);
    grid.setPadding(new Insets( v: 10));

    Label usernameLabel = new Label( s: "Username:");
    TextField usernameField = new TextField();
    Label passwordLabel = new Label( s: "Password:");
    PasswordField passwordField = new PasswordField();
    Label dobLabel = new Label( s: "Date of Birth:");
    TextField dobField = new TextField();
    Label genderLabel = new Label( s: "Gender:");
    ComboBox<String> genderComboBox = new ComboBox<>();
    genderComboBox.getItems().addAll( ...es: "Male", "Female");
    Label addressLabel = new Label( s: "Address:");
    TextField addressField = new TextField();
    Button registerButton = new Button( s: "Register");
    Button backButton = new Button( s: "Back");

    registerButton.setOnAction(e -> {
        String username = usernameField.getText();
        String password = passwordField.getText();
        String dob = dobField.getText();
        String genderInput = genderComboBox.getValue();
        String address = addressField.getText();

        if (username.isEmpty() || password.isEmpty() || dob.isEmpty() || genderInput == null || address.isEmpty()) {
            showAlert("All fields are required!");
            return;
        }
```

```
        }

        Gender gender = Gender.valueOf(genderInput.toUpperCase());
        Customer customer = new Customer(username, password, dob, gender, address,  balance: 0.0, new ArrayList<>());
        Database.addCustomer(customer);
        showAlert("Registration successful!");
        primaryStage.setScene(mainScene);
    });

    backButton.setOnAction(e -> primaryStage.setScene(mainScene));

    grid.add(usernameLabel,  i: 0,  i1: 0);
    grid.add(usernameField,  i: 1,  i1: 0);
    grid.add(passwordLabel,  i: 0,  i1: 1);
    grid.add(passwordField,  i: 1,  i1: 1);
    grid.add(dobLabel,  i: 0,  i1: 2);
    grid.add(dobField,  i: 1,  i1: 2);
    grid.add(genderLabel,  i: 0,  i1: 3);
    grid.add(genderComboBox,  i: 1,  i1: 3);
    grid.add(addressLabel,  i: 0,  i1: 4);
    grid.add(addressField,  i: 1,  i1: 4);
    grid.add(registerButton,  i: 0,  i1: 5);
    grid.add(backButton,  i: 1,  i1: 5);

    return new Scene(grid,  v: 400,  v1: 300);
}
```

4-Admin dashboard

Admin Dashboard

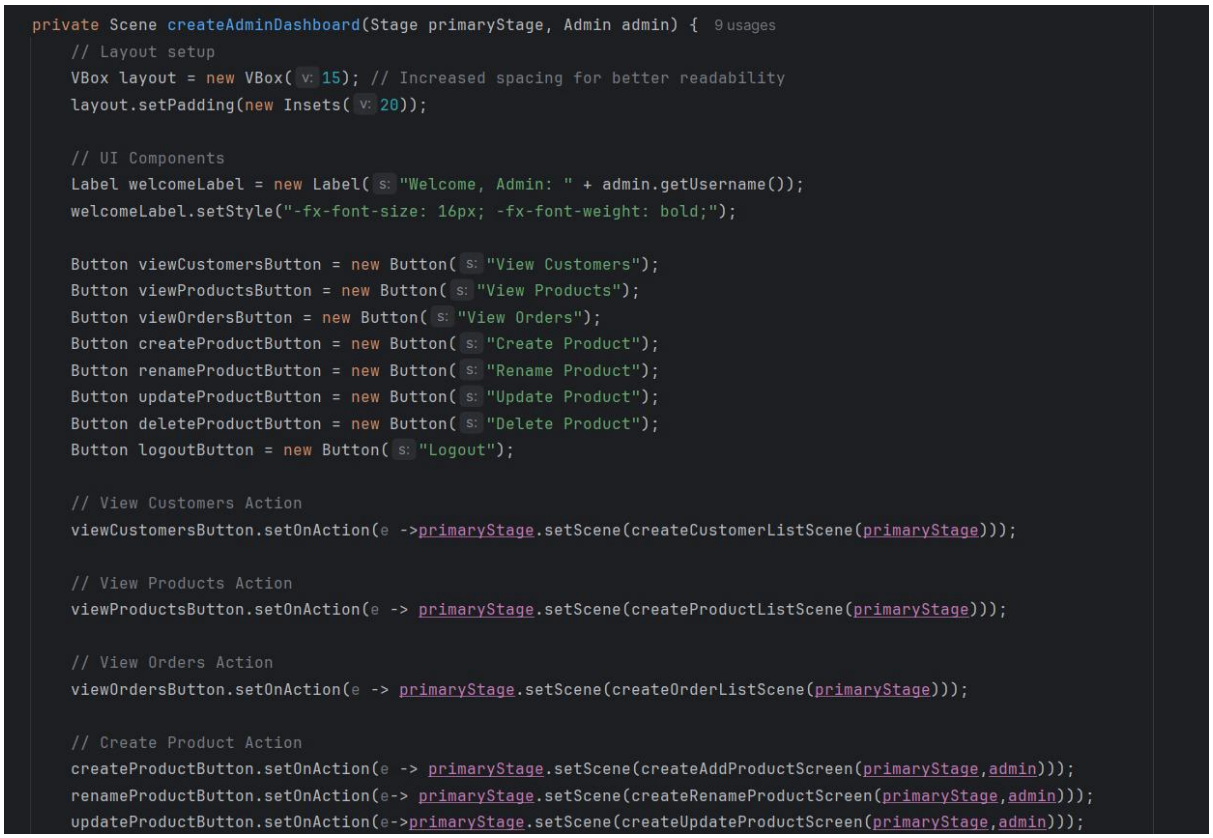
Purpose: Displays admin-specific functionality (e.g., managing customers or products).

Components:

Label welcoming the admin.

Buttons for actions like viewing customers/products and logging out.

Navigation: Clicking buttons can redirect to new screens.



```
private Scene createAdminDashboard(Stage primaryStage, Admin admin) {  9 usages
    // Layout setup
    VBox layout = new VBox( v: 15); // Increased spacing for better readability
    layout.setPadding(new Insets( v: 20));

    // UI Components
    Label welcomeLabel = new Label( s: "Welcome, Admin: " + admin.getUsername());
    welcomeLabel.setStyle("-fx-font-size: 16px; -fx-font-weight: bold;");

    Button viewCustomersButton = new Button( s: "View Customers");
    Button viewProductsButton = new Button( s: "View Products");
    Button viewOrdersButton = new Button( s: "View Orders");
    Button createProductButton = new Button( s: "Create Product");
    Button renameProductButton = new Button( s: "Rename Product");
    Button updateProductButton = new Button( s: "Update Product");
    Button deleteProductButton = new Button( s: "Delete Product");
    Button logoutButton = new Button( s: "Logout");

    // View Customers Action
    viewCustomersButton.setOnAction(e ->primaryStage.setScene(createCustomerListScene(primaryStage)));

    // View Products Action
    viewProductsButton.setOnAction(e -> primaryStage.setScene(createProductListScene(primaryStage)));

    // View Orders Action
    viewOrdersButton.setOnAction(e -> primaryStage.setScene(createOrderListScene(primaryStage)));

    // Create Product Action
    createProductButton.setOnAction(e -> primaryStage.setScene(createAddProductScreen(primaryStage,admin)));
    renameProductButton.setOnAction(e-> primaryStage.setScene(createRenameProductScreen(primaryStage,admin)));
    updateProductButton.setOnAction(e->primaryStage.setScene(createUpdateProductScreen(primaryStage,admin)));
```

```
    // Logout Action
    logoutButton.setOnAction(e -> primaryStage.setScene(mainScene));

    // Add components to layout
    layout.getChildren().addAll(
            welcomeLabel,
            viewCustomersButton,
            viewProductsButton,
            viewOrdersButton,
            createProductButton,
            renameProductButton,
            updateProductButton,
            deleteProductButton,
            logoutButton
    );

    // Return the scene
    return new Scene(layout,  v: 500,  v1: 400);
}
```

5-Customer dashboard

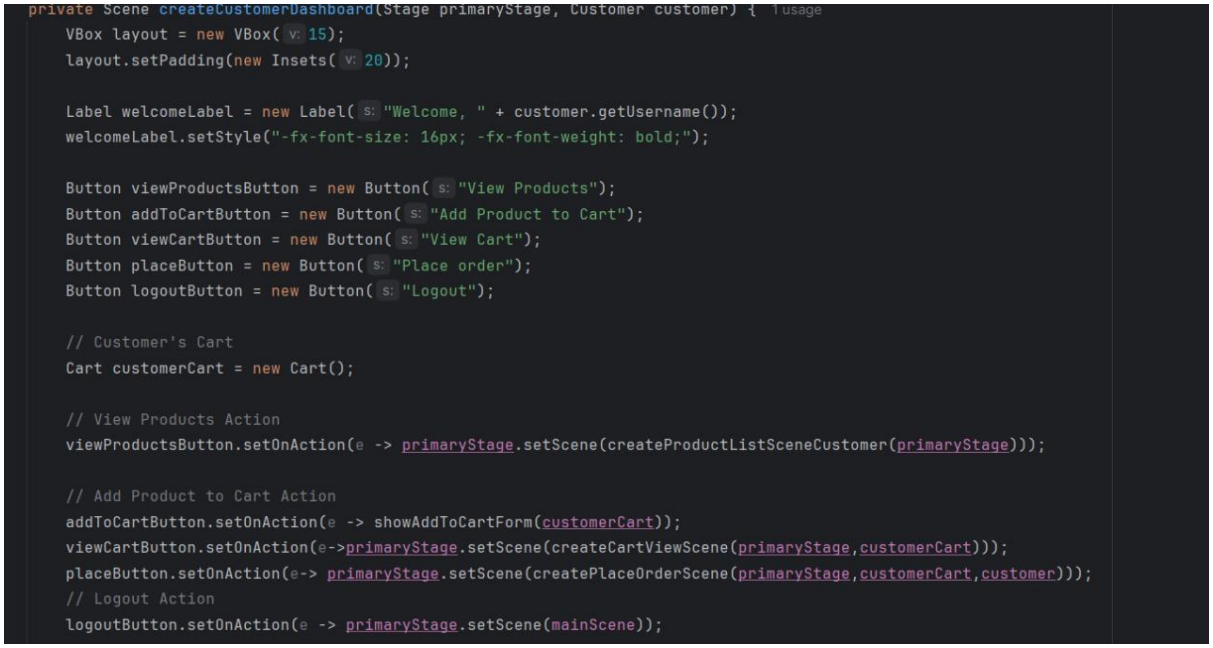
Purpose: Displays customer-specific functionality (e.g., viewing products or cart).

Components:

Label welcoming the customer.

Buttons for actions like viewing products or cart and logging out.

Navigation: Clicking buttons can redirect to new screens.



```
private Scene createCustomerDashboard(Stage primaryStage, Customer customer) {  1 usage
    VBox layout = new VBox( v: 15);
    layout.setPadding(new Insets( v: 20));

    Label welcomeLabel = new Label( s: "Welcome, " + customer.getUsername());
    welcomeLabel.setStyle("-fx-font-size: 16px; -fx-font-weight: bold;");

    Button viewProductsButton = new Button( s: "View Products");
    Button addToCartButton = new Button( s: "Add Product to Cart");
    Button viewCartButton = new Button( s: "View Cart");
    Button placeButton = new Button( s: "Place order");
    Button logoutButton = new Button( s: "Logout");

    // Customer's Cart
    Cart customerCart = new Cart();

    // View Products Action
    viewProductsButton.setOnAction(e -> primaryStage.setScene(createProductListSceneCustomer(primaryStage)));

    // Add Product to Cart Action
    addToCartButton.setOnAction(e -> showAddToCartForm(customerCart));
    viewCartButton.setOnAction(e->primaryStage.setScene(createCartViewScene(primaryStage,customerCart)));
    placeButton.setOnAction(e-> primaryStage.setScene(createPlaceOrderScene(primaryStage,customerCart,customer)));
    // Logout Action
    logoutButton.setOnAction(e -> primaryStage.setScene(mainScene));
```

6-Add product screen

Purpose

Enables an admin to add a product to the database.

Captures all necessary details such as:

Product ID

Product Name

Price

Description

Stock Quantity

Category (linked to existing categories in the system)

Components

The GUI is built using a GridPane layout for structured placement of input fields and labels.

1. Input Fields

Product ID:

A TextField for entering a numeric identifier for the product.

Product Name:

A TextField for the name of the product.

Product Price:

A TextField to input the price (numeric and validated).

Product Description:

A TextField for a brief description of the product.

Stock Quantity:

A TextField for the available stock count (numeric and validated).

Category:

A ComboBox that dynamically lists categories from the database.

The category name and ID are displayed for better selection context.

2. Buttons

Submit Button:

Validates the input fields and adds the product to the database.

Displays an appropriate success or error message.

Cancel Button:

Returns the user to the Admin Dashboard without saving any data.

Features & Validations

Category Selection:

Displays category names and IDs in the combo box.

Extracts the category ID for association with the product.

Input Validation:

Numeric validation for fields like Product ID, Product Price, and Stock Quantity.

Checks for empty fields or invalid selections (e.g., no category selected).

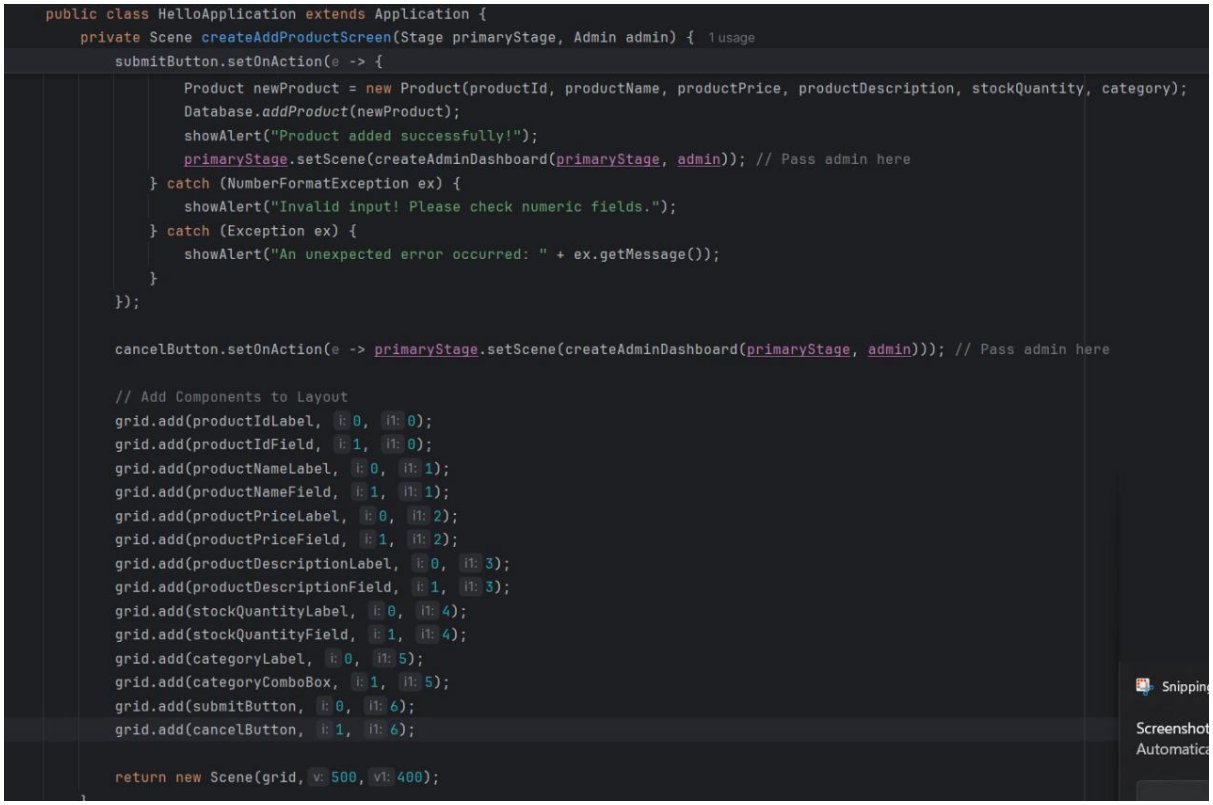
Error Handling:

Alerts the user if any field is incorrectly filled or left blank.

Handles unexpected exceptions with error messages.

```java
private Scene createAddProductScreen(Stage primaryStage, Admin admin) {  1 usage
    GridPane grid = new GridPane();
    grid.setHgap(10);
    grid.setVgap(10);
    grid.setPadding(new Insets( v: 20));

    // Input Fields
    Label productIdLabel = new Label( s: "Product ID:");
    TextField productIdField = new TextField();

    Label productNameLabel = new Label( s: "Product Name:");
    TextField productNameField = new TextField();

    Label productPriceLabel = new Label( s: "Product Price:");
    TextField productPriceField = new TextField();

    Label productDescriptionLabel = new Label( s: "Description:");
    TextField productDescriptionField = new TextField();

    Label stockQuantityLabel = new Label( s: "Stock Quantity:");
    TextField stockQuantityField = new TextField();

    Label categoryLabel = new Label( s: "Category:");
    ComboBox<String> categoryComboBox = new ComboBox<>();
    for (Category category : Database.getCategories()) {
        categoryComboBox.getItems().add(category.getName() + " (ID: " + category.getId() + ")");
    }
```

```java
Button submitButton = new Button( s: "Submit");
Button cancelButton = new Button( s: "Cancel");

submitButton.setOnAction(e -> {
    try {
        int productId = Integer.parseInt(productIdField.getText());
        String productName = productNameField.getText();
        double productPrice = Double.parseDouble(productPriceField.getText());
        String productDescription = productDescriptionField.getText();
        int stockQuantity = Integer.parseInt(stockQuantityField.getText());

        // Validate category selection
        String selectedCategory = categoryComboBox.getValue();
        if (selectedCategory == null) {
            showAlert("Please select a category.");
            return;
        }

        // Extract category ID
        int categoryId = Integer.parseInt(selectedCategory.replaceAll( regex: "\\D+", replacement: ""));
        Category category = Database.getCategoryById(categoryId);

        // Validate inputs
        if (productName.isEmpty() || productDescription.isEmpty() || category == null) {
            showAlert("All fields must be filled in correctly!");
            return;
        }

        // Add product to database
        Product newProduct = new Product(productId, productName, productPrice, productDescription, stockQuantity, category);
```



```
public class HelloApplication extends Application {
    private Scene createAddProductScreen(Stage primaryStage, Admin admin) {  1 usage
        submitButton.setOnAction(e -> {
                Product newProduct = new Product(productId, productName, productPrice, productDescription, stockQuantity, category);
                Database.addProduct(newProduct);
                showAlert("Product added successfully!");
                primaryStage.setScene(createAdminDashboard(primaryStage, admin)); // Pass admin here
            } catch (NumberFormatException ex) {
                showAlert("Invalid input! Please check numeric fields.");
            } catch (Exception ex) {
                showAlert("An unexpected error occurred: " + ex.getMessage());
            }
        });

        cancelButton.setOnAction(e -> primaryStage.setScene(createAdminDashboard(primaryStage, admin))); // Pass admin here

        // Add Components to Layout
        grid.add(productIdLabel,  i: 0,  i1: 0);
        grid.add(productIdField,  i: 1,  i1: 0);
        grid.add(productNameLabel,  i: 0,  i1: 1);
        grid.add(productNameField,  i: 1,  i1: 1);
        grid.add(productPriceLabel,  i: 0,  i1: 2);
        grid.add(productPriceField,  i: 1,  i1: 2);
        grid.add(productDescriptionLabel,  i: 0,  i1: 3);
        grid.add(productDescriptionField,  i: 1,  i1: 3);
        grid.add(stockQuantityLabel,  i: 0,  i1: 4);
        grid.add(stockQuantityField,  i: 1,  i1: 4);
        grid.add(categoryLabel,  i: 0,  i1: 5);
        grid.add(categoryComboBox,  i: 1,  i1: 5);
        grid.add(submitButton,  i: 0,  i1: 6);
        grid.add(cancelButton,  i: 1,  i1: 6);

        return new Scene(grid,  v: 500,  v1: 400);
```

7-Rename product

Purpose

Allows admins to rename an existing product in the database.

Requires two inputs:

Product ID: Identifies the product to rename.

New Product Name: Specifies the updated name for the product.

Components

Input Fields:

Product ID: A TextField to accept the product's unique numeric identifier.

New Product Name: A TextField to accept the updated product name.

Buttons:

Submit Button:

Validates inputs.

Checks for product existence using the ID.

Renames the product if inputs are valid and displays a success message.

Redirects back to the Admin Dashboard.

Cancel Button:

Returns to the Admin Dashboard without making changes.

Validations

Product ID Validation:

Ensures the input is numeric and corresponds to an existing product.

Displays an error message if the product is not found.

New Product Name Validation:

Ensures the new name field is not left empty.

Displays an error message if the field is empty.

Error Handling:

Manages invalid numeric input for the Product ID.

Catches unexpected errors and displays an error message.

Workflow

Admin enters the Product ID and the new product name.

When the Submit button is clicked:

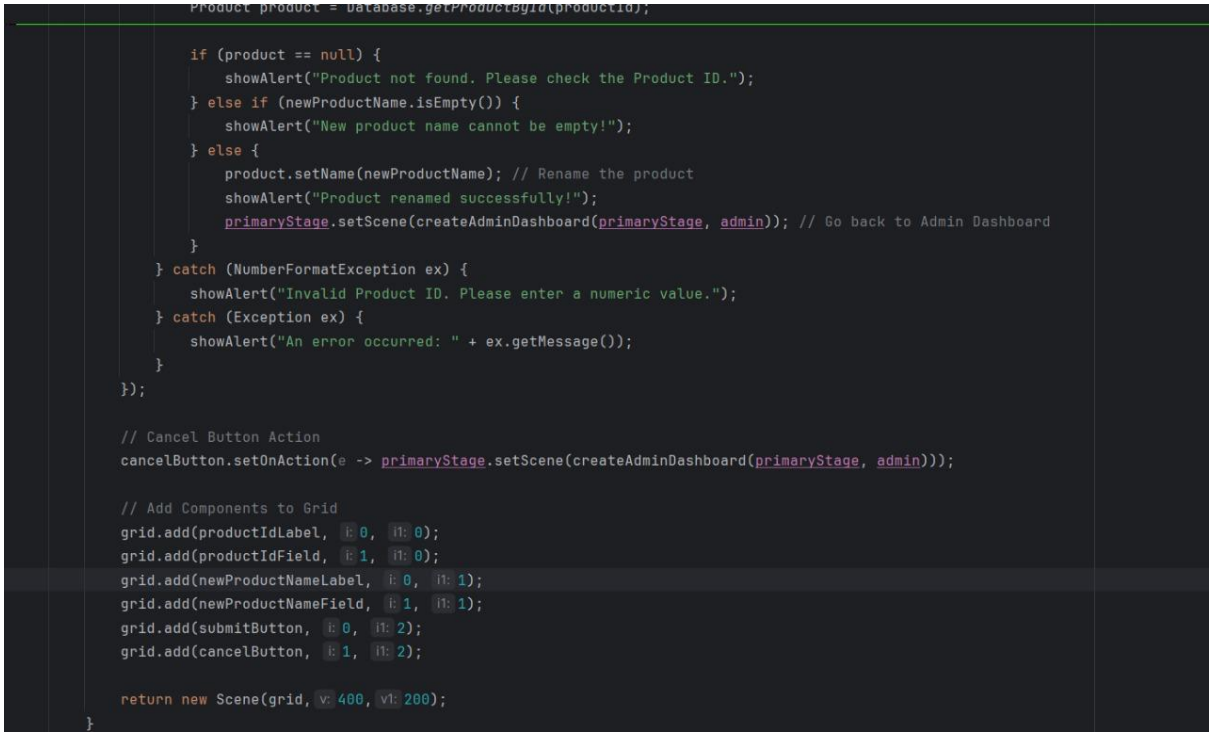
The application fetches the product by ID from the database.

If validations pass, the product's name is updated.

A success message is displayed, and the admin is redirected to the Admin Dashboard.

Clicking the Cancel button returns to the Admin Dashboard without changes.

```
private Scene createRenameProductScreen(Stage primaryStage, Admin admin) {  1 usage
    GridPane grid = new GridPane();
    grid.setHgap(10);
    grid.setVgap(10);
    grid.setPadding(new Insets( v: 20));

    // Labels and Input Fields
    Label productIdLabel = new Label( s: "Product ID:");
    TextField productIdField = new TextField();

    Label newProductNameLabel = new Label( s: "New Product Name:");
    TextField newProductNameField = new TextField();

    Button submitButton = new Button( s: "Submit");
    Button cancelButton = new Button( s: "Cancel");

    // Submit Button Action
    submitButton.setOnAction(e -> {
        try {
            int productId = Integer.parseInt(productIdField.getText());
            String newProductName = newProductNameField.getText();

            Product product = Database.getProductById(productId);

            if (product == null) {
                showAlert("Product not found. Please check the Product ID.");
            } else if (newProductName.isEmpty()) {
                showAlert("New product name cannot be empty!");
            } else {
```



```
            Product product = database.getProductById(productId);

            if (product == null) {
                showAlert("Product not found. Please check the Product ID.");
            } else if (newProductName.isEmpty()) {
                showAlert("New product name cannot be empty!");
            } else {
                product.setName(newProductName); // Rename the product
                showAlert("Product renamed successfully!");
                primaryStage.setScene(createAdminDashboard(primaryStage, admin)); // Go back to Admin Dashboard
            }
        } catch (NumberFormatException ex) {
            showAlert("Invalid Product ID. Please enter a numeric value.");
        } catch (Exception ex) {
            showAlert("An error occurred: " + ex.getMessage());
        }
    });

    // Cancel Button Action
    cancelButton.setOnAction(e -> primaryStage.setScene(createAdminDashboard(primaryStage, admin)));

    // Add Components to Grid
    grid.add(productIdLabel, i: 0, i1: 0);
    grid.add(productIdField, i: 1, i1: 0);
    grid.add(newProductNameLabel, i: 0, i1: 1);
    grid.add(newProductNameField, i: 1, i1: 1);
    grid.add(submitButton, i: 0, i1: 2);
    grid.add(cancelButton, i: 1, i1: 2);

    return new Scene(grid, v: 400, v1: 200);
}
```

8-Update product

Purpose

Allows admins to update a product's attributes:

Price: Update the product's price.

Stock Quantity: Update the product's stock quantity.

Provides the ability to skip updates for specific fields by entering -1.

Components

Input Fields:

Product ID: A TextField to accept the unique identifier of the product to update.

New Price: A TextField for entering the updated price. Entering -1 skips the update for price.

New Stock Quantity: A TextField for entering the updated stock quantity. Entering -1 skips the update for stock quantity.

Buttons:

Submit Button:

Validates the input fields.

Updates the price and stock quantity if valid values are provided.

Displays a success message upon completion.

Redirects to the Admin Dashboard.

Cancel Button:

Discards the update process and redirects to the Admin Dashboard.

Validations

Product ID Validation:

Ensures the Product ID is numeric and corresponds to an existing product in the database.

Displays an error message if the product does not exist.

Price and Stock Validation:

Price and stock fields accept numeric values only.

Skipping updates for these fields is allowed by entering -1.

Error Handling:

Manages invalid numeric inputs for Product ID, price, and stock quantity.

Handles unexpected exceptions and displays an appropriate error message.

Workflow

Admin enters the Product ID and optionally provides a new price and/or stock quantity.

When the Submit button is clicked:

The application fetches the product using the provided ID.

If valid:

Updates the price and/or stock quantity based on the input values.

Skips updates for fields where -1 is entered.

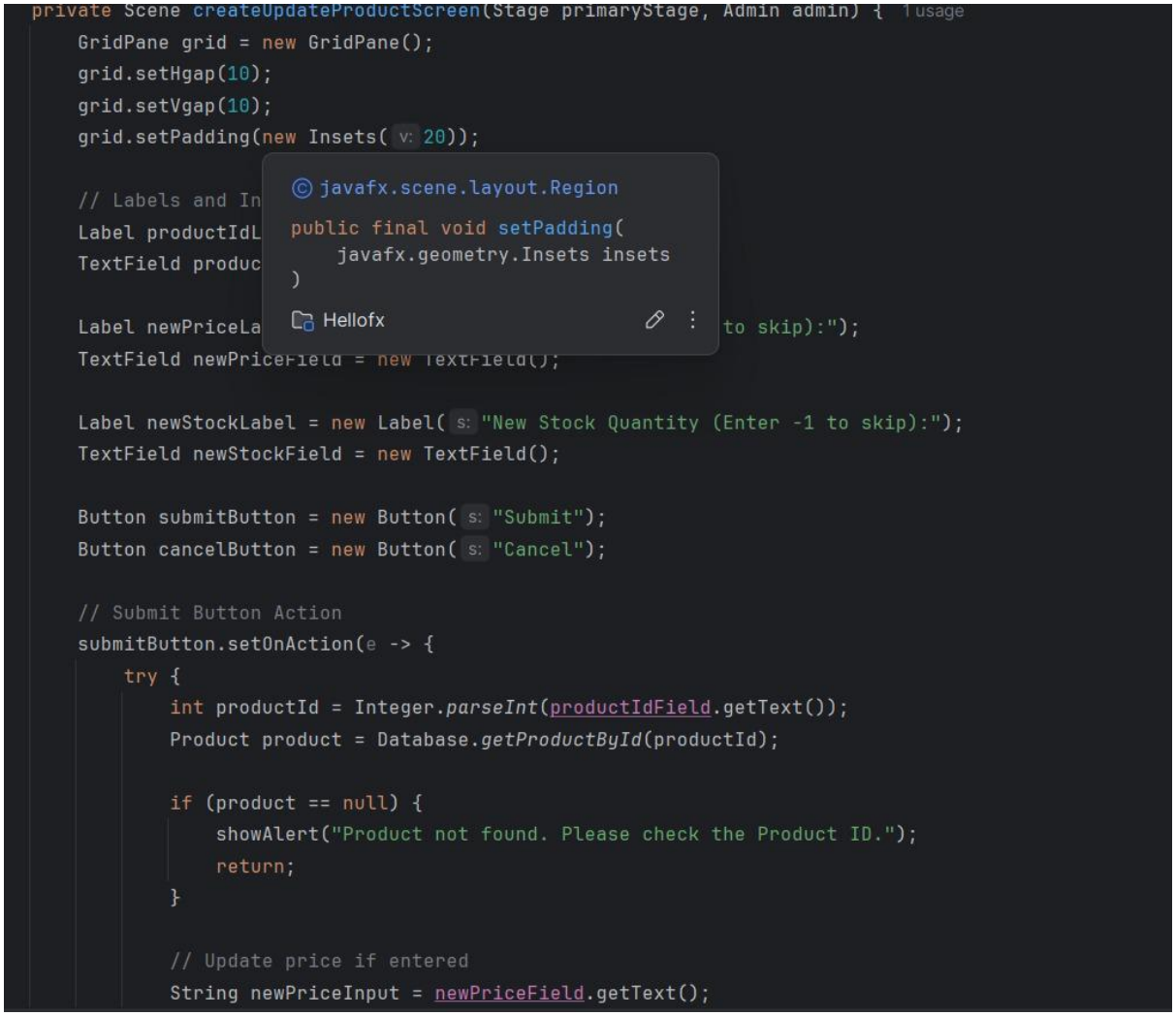
Displays a success message.

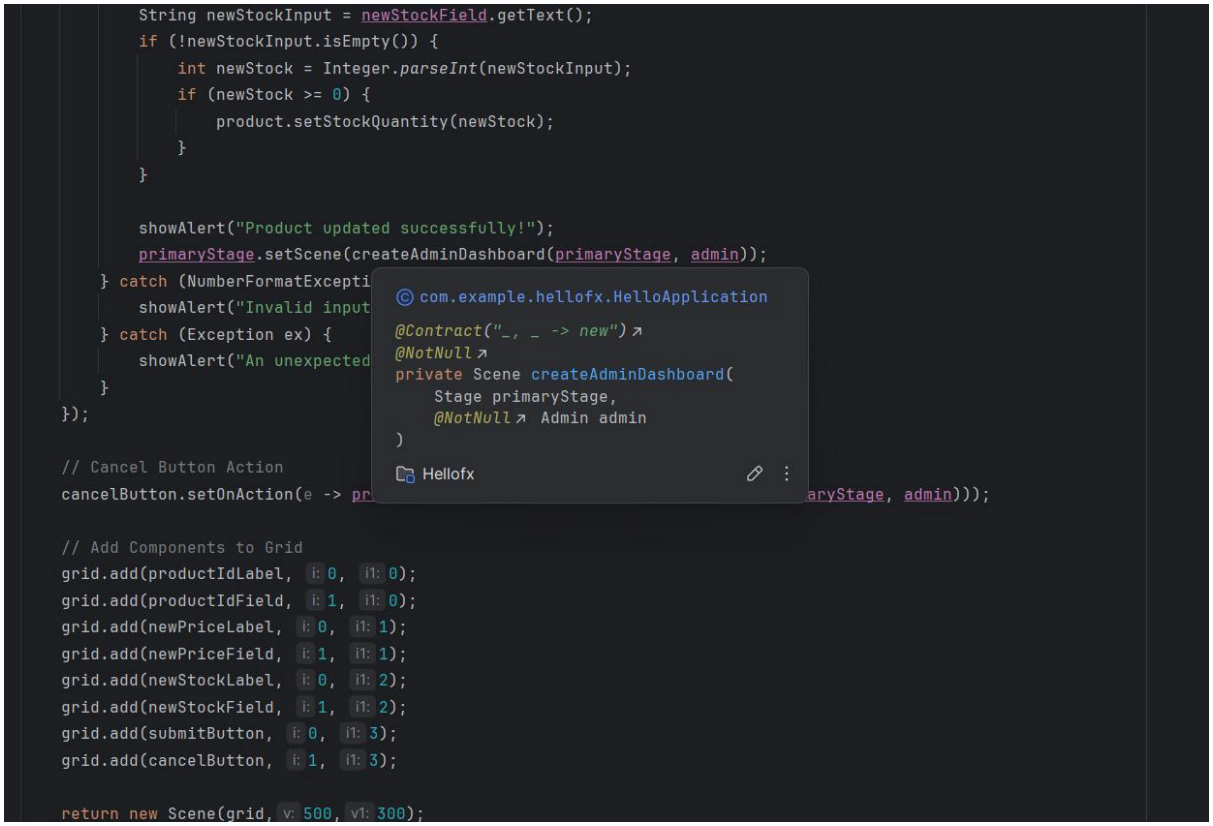
If invalid:

Displays an appropriate error message.

Returns to the Admin Dashboard upon success.

Clicking the Cancel button redirects to the Admin Dashboard without changes.


private Scene createUpdateProductScreen(Stage primaryStage, Admin admin) {
GridPane grid = new GridPane();
grid.setHgap(10);
grid.setVgap(10);
grid.setPadding(new Insets(v: 20));
javafx.scene.layout.Region
public final void setPadding(
javafx.geometry.Insets insets
)
Hellofx
// Labels and In
Label productIdL
TextField produc
Label newPriceLa
to skip):");
TextField newPriceField = new TextField();
Label newStockLabel = new Label(s: "New Stock Quantity (Enter -1 to skip):");
TextField newStockField = new TextField();
Button submitButton = new Button(s: "Submit");
Button cancelButton = new Button(s: "Cancel");
// Submit Button Action
submitButton.setOnAction(e -> {
try {
int productId = Integer.parseInt(productIdField.getText());
Product product = Database.getProductById(productId);
if (product == null) {
showAlert("Product not found. Please check the Product ID.");
return;
}
// Update price if entered
String newPriceInput = newPriceField.getText();



9-Delete product

Purpose

Enables admins to delete a product by entering its Product ID.

Provides a user-friendly interface with options to confirm deletion or cancel the operation.

Components

Input Fields:

Product ID: A TextField for entering the unique identifier of the product to be deleted.

Buttons:

Delete Button:

Confirms the deletion of the product.

Validates the entered Product ID.

Deletes the product if it exists in the database.

Displays an appropriate success or error message.

Redirects to the Admin Dashboard after the operation.

Cancel Button:

Discards the deletion operation.

Redirects back to the Admin Dashboard.

Validations

Product ID Validation:

Ensures the Product ID is numeric.

Checks if the Product ID exists in the database before attempting deletion.

Error Handling:

Handles invalid input (non-numeric Product ID) and displays an error message.

Manages unexpected exceptions and provides feedback to the user.

Workflow

Admin enters the Product ID of the product they wish to delete.

When the Delete button is clicked:

The Product ID is validated to ensure it is numeric.

The application attempts to delete the product from the database.

If successful:

Displays a success message: "Product deleted successfully!"

If the Product ID is invalid or the product does not exist:

Displays an error message: "Product not found. Please enter a valid Product ID."

Redirects to the Admin Dashboard after completion.

Clicking the Cancel button redirects back to the Admin Dashboard without making changes.

```
private Scene createDeleteProductScreen(Stage primaryStage, Admin admin) { 1 usage
    GridPane grid = new GridPane();
    grid.setHgap(10);
    grid.setVgap(10);
    grid.setPadding(new Insets( v: 20));

    // Input Fields
    Label productIdLabel = new Label( s: "Product ID:");
    TextField productIdField = new TextField();

    Button deleteButton = new Button( s: "Delete");
    Button cancelButton = new Button( s: "Cancel");

    // Delete Button Action
    deleteButton.setOnAction(e -> {
        try {
            int productId = Integer.parseInt(productIdField.getText());
            boolean removed = Database.deleteProduct(productId);

            if (removed) {
                showAlert("Product deleted successfully!");
            } else {
                showAlert("Product not found. Please enter a valid Product ID.");
            }

            // Return to Admin Dashboard
            primaryStage.setScene(createAdminDashboard(primaryStage, admin));
        } catch (NumberFormatException ex) {
            showAlert("Invalid input! Please enter a numeric Product ID.");
```

10-Customer list

VBox Layout:

The VBox is a layout container that arranges its children vertically with a gap of 10 pixels between them.

setPadding is used to provide 20 pixels of space around the content.

Title Label:

A Label with the text "Customer List" is created.

The setStyle method is used to change the font size to 16px and make the font bold.

ListView for Customers:

A ListView<String> is used to display a list of customer details.

For each customer in the Database.getCustomers() collection, a string is added to the ListView that displays:

Username

Gender

Address

Balance (formatted to two decimal places).

The String.format("%.2f", customer.getBalance()) ensures that the balance is displayed as a floating-point number with exactly two decimal places.

Back Button:

A Button labeled "Back" is created.

An event handler is set on the button. When clicked, it changes the scene back to adminScene, which is assumed to be defined elsewhere in the code.

Scene Creation:

All the elements (title, list view, and button) are added to the VBox layout using getChildren().addAll().

Finally, a new Scene is created with the VBox layout, with a width of 500 pixels and height of 400 pixels, and it is returned as the result.

Key Aspects:

The ListView dynamically displays customer details pulled from the Database.getCustomers() method, which presumably returns a list or collection of Customer objects.

The "Back" button is functional and takes the user back to a previous scene (adminScene).

The code is structured for creating a JavaFX UI where users can view customer information.

```
private Scene createCustomerListScene(Stage primaryStage) {  1 usage
    VBox layout = new VBox( v: 10);
    layout.setPadding(new Insets( v: 20));

    Label title = new Label( s: "Customer List");
    title.setStyle("-fx-font-size: 16px; -fx-font-weight: bold;");

    ListView<String> customerListView = new ListView<>();
    for (Customer customer : Database.getCustomers()) {
        customerListView.getItems().add("Username: " + customer.getUsername() +
                " | Gender: " + customer.getGender() + // Add gender here
                " | Address: " + customer.getAddress() +
                " | Balance: $" + String.format("%.2f", customer.getBalance()));
    }

    Button backButton = new Button( s: "Back");
    backButton.setOnAction(e -> primaryStage.setScene(adminScene));

    layout.getChildren().addAll(title, customerListView, backButton);
    return new Scene(layout,  v: 500,  v1: 400);
}
```

11-Products list (for customer and admin)

VBox Layout:

The VBox layout is used to arrange its children vertically with a gap of 10 pixels between them.

The setPadding(new Insets(20)) method provides 20 pixels of padding around the content, ensuring there’s space between the edges of the scene and the items inside.

2. Title Label:

A Label is created with the text "Product List".

The setStyle method is applied to make the font size 16 pixels and the font bold. This gives the title a distinctive look.

3. ListView for Products:

A ListView<String> is used to display a list of product details.

The method loops over Database.getProducts() (presumably a method that returns a collection of Product objects).

For each Product, the following information is displayed:

Product Name (product.getName())

Price, formatted to two decimal places (String.format("%.2f", product.getPrice()))

Stock Quantity (product.getStockQuantity()).

Each product’s details are added to the ListView as a string, creating a list where users can see information about each product.

4. Back Button:

A Button labeled "Back" is created.

An event handler is set for the button such that when it is clicked, the scene changes to adminScene (which is assumed to be defined elsewhere in the code, likely as a scene for the admin interface).

5. Scene Creation:

The VBox layout containing the title, product list, and back button is added to the scene.

The scene is created with dimensions of 500 pixels in width and 400 pixels in height and returned as the result.

Key Aspects:

Dynamic Product Listing: The list view dynamically generates its content based on the products returned by Database.getProducts(). Each product's name, price, and stock quantity are displayed.

Formatted Price: The product price is formatted to display two decimal places, ensuring a clean representation of currency.

Back Button Navigation: The back button allows navigation to another scene (adminScene), which is assumed to be the previous screen or the admin interface.

UI Layout: The VBox is a simple vertical layout container, and the elements (title, product list, and button) are neatly arranged with spacing and padding for a clean UI.

```
private Scene createProductListScene(Stage primaryStage) {  1 usage
    VBox layout = new VBox( v: 10);
    layout.setPadding(new Insets( v: 20));

    Label title = new Label( s: "Product List");
    title.setStyle("-fx-font-size: 16px; -fx-font-weight: bold;");

    ListView<String> productListView = new ListView<>();
    for (Product product : Database.getProducts()) {
        productListView.getItems().add("Product: " + product.getName() +
                " | Price: $" + String.format("%.2f", product.getPrice()) +
                " | Stock: " + product.getStockQuantity());
    }

    Button backButton = new Button( s: "Back");
    backButton.setOnAction(e -> primaryStage.setScene(adminScene));

    layout.getChildren().addAll(title, productListView, backButton);
    return new Scene(layout, v: 500, v1: 400);
}
```

12-Order list

VBox Layout:

VBox is used as the layout container, with a 10-pixel vertical gap between its children.

setPadding(new Insets(20)) adds padding around the layout, creating space between the edges of the scene and the content inside.

2. Title Label:

A Label with the text "Order List" is created to serve as the title of the scene.

The setStyle method is used to apply a font size of 16px and make the font bold, making the title more prominent.

3. ListView for Orders:

A ListView<String> is used to display a list of orders.

The code loops through Database.getOrders(), which presumably returns a collection of Order objects, and processes each Order object:

Order Count: The count variable tracks the order number (incremented on each loop iteration).

Order Details: For each order, the following details are displayed:

Order Number: Displayed as "Order #<count>".

Customer: The username of the customer associated with the order (order.getCustomer().getUsername()).

Items: The items in the cart are formatted by calling formatCartItems(order.getCart()). This method likely formats the list of cart items (assuming it returns a string).

Total Price: The total price of the cart is displayed, formatted to two decimal places using String.format("%.2f", order.getCart().calculateTotal()).

Payment Method: The method used for payment is displayed using order.getPaymentMethod().

Each order's details are added as a string to the orderListView list.

4. Back Button:

A Button labeled "Back" is created.

An event handler is added to the button, which changes the scene back to adminScene (assumed to be defined elsewhere as the admin interface).

5. Scene Creation:

All the components (title, order list, and back button) are added to the VBox layout using getChildren().addAll().

A new Scene is created using the VBox layout, with a width of 500 pixels and height of 400 pixels, and returned.

Key Aspects:

Dynamic Order List: The ListView dynamically displays each order's details by iterating over Database.getOrders() and formatting the order information into readable strings.

Formatted Price: The total price of each order is formatted to two decimal places to ensure consistency in currency representation.

Back Button Navigation: The back button allows the user to return to the adminScene, which is likely an admin interface or previous screen.

UI Layout: The vertical layout (VBox) arranges the elements (title, list of orders, and back button) with appropriate spacing and padding to ensure a clean and user-friendly interface.

```
private Scene createOrderListScene(Stage primaryStage) {  1 usage
    VBox layout = new VBox( v: 10);
    layout.setPadding(new Insets( v: 20));

    Label title = new Label( s: "Order List");
    title.setStyle("-fx-font-size: 16px; -fx-font-weight: bold;");

    ListView<String> orderListView = new ListView<>();
    int count = 0;
    for (Order order : Database.getOrders()) {
        count++;
        orderListView.getItems().add("Order #" + count +
                " | Customer: " + order.getCustomer().getUsername() +
                "\nItems: " + formatCartItems(order.getCart()) +
                "\nTotal Price: $" + String.format("%.2f", order.getCart().calculateTotal()) +
                " | Payment: " + order.getPaymentMethod());
    }

    Button backButton = new Button( s: "Back");
    backButton.setOnAction(e -> primaryStage.setScene(adminScene));

    layout.getChildren().addAll(title, orderListView, backButton);
    return new Scene(layout,  v: 500,  v1: 400);
}
```

13-View cart

VBox Layout:

VBox is used to arrange the UI components vertically with a 10-pixel gap between them.

setPadding(new Insets(20)) adds 20 pixels of padding around the layout, ensuring there's space between the edges of the scene and the content.

2. Title Label:

A Label with the text "Your Cart" is created to act as the title of the scene.

The setStyle method is applied to make the font size 16px and the font bold, giving the title prominence.

3. Cart Items Box:

A nested VBox called cartItemsBox is used to hold the cart items. It has a vertical gap of 10 pixels between each item.

If the cart is empty (cart.isEmpty()), a Label saying "Your cart is empty." is added to cartItemsBox.

If the cart is not empty, it loops through the list of items in the cart (cart.getItems()), displaying the following for each item:

Product Name and Quantity: A Label is created to show the product name, quantity, and price per item, formatted with two decimal places for the price.

Remove Button: A button labeled "Remove" is created next to each item. The button is styled with a red background and white text. When clicked, the button triggers an action that removes the product from the cart (cart.removeItem(product)) and refreshes the scene by resetting it with the updated cart

(primaryStage.setScene(createCartViewScene(primaryStage, cart))).

4. Total Price Label:

After displaying all items in the cart, a Label is created to show the total price of the cart, formatted to two decimal places.

This label is added to the cartItemsBox.

5. Back Button:

A Button labeled "Back" is created, allowing the user to navigate back to a previous scene (customerScene), which is assumed to be defined elsewhere in the code.

The event handler sets the scene to customerScene when clicked.

6. Scene Creation:

All the elements—title, cart items (or empty message), total label, and back button—are added to the main VBox layout (layout).

A new Scene is created with the VBox layout, sized 500 pixels in width and 400 pixels in height, and returned.

Key Aspects:

Dynamic Cart View: The cart contents are dynamically populated based on the cart object. If the cart has items, it displays them along with their quantity and price. Otherwise, it displays a message indicating the cart is empty.

Item Removal: Each item in the cart has a corresponding "Remove" button that, when clicked, removes the item from the cart and updates the view.

Formatted Price: Prices (both individual and total) are formatted to display two decimal places for consistency.

Back Button: The "Back" button provides navigation to another scene, assumed to be the customer’s previous screen or a shopping interface.

```
private Scene createCartViewScene(Stage primaryStage, Cart cart) {  2 usages
    VBox layout = new VBox( v: 10);
    layout.setPadding(new Insets( v: 20));

    Label title = new Label( s: "Your Cart");
    title.setStyle("-fx-font-size: 16px; -fx-font-weight: bold;");

    VBox cartItemsBox = new VBox( v: 10);

    if (cart.isEmpty()) {
        Label emptyLabel = new Label( s: "Your cart is empty.");
        cartItemsBox.getChildren().add(emptyLabel);
    } else {
        for (int i = 0; i < cart.getItems().size(); i++) {
            Product product = cart.getItems().get(i);
            int quantity = cart.getQuantities().get(i);

            HBox productBox = new HBox( v: 10);
            productBox.setAlignment(Pos.CENTER_LEFT);

            Label productLabel = new Label( s: product.getName() + " x " + quantity +
                    " ($" + String.format("%.2f", product.getPrice()) + " each)");

            Button removeButton = new Button( s: "Remove");
            removeButton.setStyle("-fx-background-color: red; -fx-text-fill: white;");
            removeButton.setOnAction(e -> {
                cart.removeItem(product);
                primaryStage.setScene(createCartViewScene(primaryStage, cart));
```

```
            removeButton.setStyle("-fx-background-color: red; -fx-text-fill: white;");
            removeButton.setOnAction(e -> {
                cart.removeItem(product);
                primaryStage.setScene(createCartViewScene(primaryStage, cart));
            });

            productBox.getChildren().addAll(productLabel, removeButton);
            cartItemsBox.getChildren().add(productBox);
        }

        Label totalLabel = new Label( s: "Total: $" + String.format("%.2f", cart.calculateTotal()));
        totalLabel.setStyle("-fx-font-size: 14px; -fx-font-weight: bold;");
        cartItemsBox.getChildren().add(totalLabel);
    }

    Button backButton = new Button( s: "Back");
    backButton.setOnAction(e -> primaryStage.setScene(customerScene));

    layout.getChildren().addAll(title, cartItemsBox, backButton);
    return new Scene(layout, v: 500, v1: 400);
}
```

14-Place order

VBox Layout:

The VBox layout is used to arrange the UI components vertically, with a gap of 15 pixels between the children.

setPadding(new Insets(20)) ensures there’s padding of 20 pixels around the layout, giving space between the scene edges and the elements inside.

2. Title Label:

A Label with the text "Place Order" is created to serve as the title of the scene.

The setStyle method is used to set the font size to 18px and make the font bold.

3. Payment Method Selection:

A Label is created to prompt the user to select a payment method.

A ComboBox<String> (paymentComboBox) is used to allow the customer to choose a payment method. The ComboBox is populated with the names of all PaymentMethod enum values.

The default payment method is set to "CreditCard" by calling paymentComboBox.setValue("CreditCard").

4. Status Message:

A Label (statusMessage) is used to display messages related to the order process, such as errors or successful actions.

The text color is set to red (-fx-text-fill: red), and the font size is set to 14px for visibility.

5. Place Order Button:

A Button labeled "Place Order" is created. When clicked, the following checks and actions occur:

Cart Empty Check: If the cart is empty (cart.isEmpty()), a message is displayed in the statusMessage label asking the user to add items to the cart before placing the order.

Balance Check: If the customer’s balance is insufficient to cover the total cost of the order (customer.getBalance() < totalCost), a message is displayed indicating the amount the customer is short.

Order Placement: If the cart is not empty and the customer has enough balance:

The balance is deducted by the total cost of the order (customer.setBalance(customer.getBalance() - totalCost)).

The selected payment method is retrieved from the ComboBox.

A new Order is created with the customer, selected payment method, and the cart.

The new order is stored in the database (Database.addOrder(newOrder)).

The receipt for the order is displayed in a new layout (receiptLayout), which includes the order details and a button to go back to the customer’s main scene.

Clear Cart: After the order is placed, the cart is cleared using cart.clearCart().

6. Back Button:

A Button labeled "Back" is created. When clicked, it takes the user back to the main scene (mainScene), which is assumed to be the main interface or home screen.

7. Order Receipt:

If the order is placed successfully, a new layout (receiptLayout) is created with:

A title Order Receipt.

The order details formatted using newOrder.getReceipt(). The receipt content is displayed in a monospace font for better readability.

A "Back" button that navigates the user back to the customer scene (customerScene).

8. Scene Creation:

All the UI components (title, payment method label, combo box, place order button, status message, and back button) are added to the VBox layout.

A new Scene is created with the layout, and it is returned.

Key Aspects:

Cart and Balance Validation: Before proceeding with the order, the method ensures that the cart is not empty and that the customer has sufficient balance to cover the total cost.

Payment Method: The customer can select a payment method from a predefined list of options (PaymentMethod enum).

Order Creation and Receipt: Once the order is successfully placed, an order receipt is generated and displayed. The order is also added to the database.

Cart Reset: After the order is placed, the cart is cleared, preparing for future orders.

Navigation: The scene provides buttons to navigate between different stages of the application (e.g., returning to the main scene or customer scene).

```
private Scene createPlaceOrderScene(Stage primaryStage, Cart cart, Customer customer) {  1 usage
    VBox layout = new VBox( v: 15);
    layout.setPadding(new Insets( v: 20));

    Label title = new Label( s: "Place Order");
    title.setStyle("-fx-font-size: 18px; -fx-font-weight: bold;");

    Label paymentLabel = new Label( s: "Select Payment Method:");
    ComboBox<String> paymentComboBox = new ComboBox<>();
    for (PaymentMethod method : PaymentMethod.values()) {
        paymentComboBox.getItems().add(method.name());
    }
    paymentComboBox.setValue("CreditCard"); // Default value

    Label statusMessage = new Label();
    statusMessage.setStyle("-fx-font-size: 14px; -fx-text-fill: red;");

    Button placeOrderButton = new Button( s: "Place Order");
    Button backButton = new Button( s: "Back");

    // Button Actions
    placeOrderButton.setOnAction(e -> {
        if (cart.isEmpty()) {
            statusMessage.setText("Your cart is empty. Please add items before placing an order.");
            return;
        }

        double totalCost = cart.calculateTotal();
        if (customer.getBalance() < totalCost) {
            statusMessage.setText("Insufficient funds! You need $" +
                    String.format("%.2f", (totalCost - customer.getBalance())) + " more.");
        } else {
```

```
        } else {
            // Deduct balance
            customer.setBalance(customer.getBalance() - totalCost);

            // Retrieve payment method
            PaymentMethod selectedMethod = PaymentMethod.valueOf(paymentComboBox.getValue());

            // Create order and store it in database
            Order newOrder = new Order(customer, selectedMethod, cart);
            Database.addOrder(newOrder);

            // Display the order receipt
            VBox receiptLayout = new VBox( v: 10);
            receiptLayout.setPadding(new Insets( v: 20));
            Label receiptTitle = new Label( s: "Order Receipt");
            receiptTitle.setStyle("-fx-font-size: 16px; -fx-font-weight: bold;");

            Label receiptContent = new Label(newOrder.getReceipt());
            receiptContent.setStyle("-fx-font-family: monospace;");

            Button backToCustomerButton = new Button( s: "Back");
            backToCustomerButton.setOnAction(ev -> primaryStage.setScene(customerScene));

            receiptLayout.getChildren().addAll(receiptTitle, receiptContent, backToCustomerButton);
            primaryStage.setScene(new Scene(receiptLayout,  v: 500,  v1: 400));

            // Clear cart after placing the order
            cart.clearCart();
        }
    });
```

Functions:

1-Add to cart

Stage Setup:

A new Stage called addToCartStage is created, which serves as the modal window to add a product to the cart.

The title of the stage is set to "Add Product to Cart".

2. GridPane Layout:

A GridPane layout is used for organizing the UI elements in a grid with horizontal (setHgap(10)) and vertical gaps (setVgap(10)), and padding (setPadding(new Insets(20))) for spacing between the components and the edges of the window.

3. Product Selection:

A ComboBox<Product> (productComboBox) is created to allow the user to select a product from a list.

The ComboBox is populated with the list of products retrieved from Database.getProducts().

The setCellFactory method is used to format how the products are displayed in the combo box. Each product is shown with its name and price (product.getName() + " - $" + product.getPrice()).

The setButtonCell method ensures that the selected product in the button is also displayed in the same format.

4. Quantity Input:

A TextField (quantityField) is created to allow the user to enter the quantity of the selected product.

The label "Enter Quantity:" prompts the user to input a number representing how many units of the product they want to add to the cart.

5. Buttons:

Two buttons are added:

Add to Cart Button: When clicked, it triggers the process of adding the selected product and quantity to the cart. The action does the following:

Product Validation: If no product is selected, an alert is shown asking the user to select a product.

Quantity Validation: The quantity is parsed from the text field. If it’s not a valid number or is less than or equal to zero, an alert is shown.

Stock Validation: If the quantity is greater than the available stock for the selected product, an alert is shown indicating insufficient stock.

Cart Update: If all validations pass, the product is added to the cart using customerCart.addItem(selectedProduct, quantity), and the product's stock quantity is reduced by the selected amount.

A success message is shown using showAlert("Product added to cart!"), and the modal window is closed.

Cancel Button: When clicked, it closes the modal window without making any changes to the cart.

6. Adding Components to Grid:

The productLabel, productComboBox, quantityLabel, quantityField, addButton, and cancelButton are added to the GridPane at specific grid positions using grid.add(). This organizes the components in rows and columns.

7. Scene Creation:

A new Scene is created with the GridPane layout and a size of 400x200 pixels.

The scene is set to the addToCartStage, and the stage is shown.

8. Alert Method:

The showAlert method is assumed to be a utility method (not shown here), which displays an alert message to the user (such as error or success messages).

Key Aspects:

Product Selection: The combo box allows for easy selection of a product from the list of available products.

Quantity Input and Validation: The quantity entered by the user is validated to ensure it is a positive number and that the requested quantity is available in stock.

Dynamic Cart Update: When a product is added to the cart, the product’s stock is reduced accordingly, and the cart is updated.

Modular Design: The form is displayed in a separate window (addToCartStage), keeping the main interface unchanged until the user is done.

```java
private void showAddToCartForm(Cart customerCart) {  1 usage
    Stage addToCartStage = new Stage();
    addToCartStage.setTitle("Add Product to Cart");

    GridPane grid = new GridPane();
    grid.setHgap(10);
    grid.setVgap(10);
    grid.setPadding(new Insets( v: 20));

    // Product Selection ComboBox
    Label productLabel = new Label( s: "Select Product:");
    ComboBox<Product> productComboBox = new ComboBox<>();
    productComboBox.getItems().addAll(Database.getProducts());
    productComboBox.setCellFactory(listView -> new ListCell<>() {
        @Override
        protected void updateItem(Product product, boolean empty) {
            super.updateItem(product, empty);
            if (empty || product == null) {
                setText(null);
            } else {
                setText(product.getName() + " - $" + product.getPrice());
            }
        }
    });
    productComboBox.setButtonCell(new ListCell<>() {
        @Override
        protected void updateItem(Product product, boolean empty) {
            super.updateItem(product, empty);
            if (empty || product == null) {
                setText(null);
            } else {
```

```java
        protected void updateItem(Product product, boolean empty) {
        }
    });
    productComboBox.setButtonCell(new ListCell<>() {
        @Override
        protected void updateItem(Product product, boolean empty) {
            super.updateItem(product, empty);
            if (empty || product == null) {
                setText(null);
            } else {
                setText(product.getName() + " - $" + product.getPrice());
            }
        }
    });

    // Quantity Input
    Label quantityLabel = new Label( s: "Enter Quantity:");
    TextField quantityField = new TextField();

    // Buttons
    Button addButton = new Button( s: "Add to Cart");
    Button cancelButton = new Button( s: "Cancel");

    // Add Product Action
    addButton.setOnAction(e -> {
        Product selectedProduct = productComboBox.getValue();
        if (selectedProduct == null) {
            showAlert("Please select a product.");
            return;
        }
```

```
            showAlert("Please select a product.");
            return;
        }

        try {
            int quantity = Integer.parseInt(quantityField.getText());
            if (quantity <= 0) {
                showAlert("Quantity must be greater than zero.");
                return;
            }

            if (quantity > selectedProduct.getStockQuantity()) {
                showAlert("Insufficient stock available.");
                return;
            }

            // Add the product to the cart
            customerCart.addItem(selectedProduct, quantity);
            selectedProduct.setStockQuantity(selectedProduct.getStockQuantity() - quantity); // Deduct stock
            showAlert("Product added to cart!");
            addToCartStage.close();
        } catch (NumberFormatException ex) {
            showAlert("Invalid quantity. Please enter a number.");
        }
    });

    // Cancel Action
    cancelButton.setOnAction(e -> addToCartStage.close());

    // Add Components to Grid
    grid.add(productLabel, 0, 0);
```

```
    // Cancel Action
    cancelButton.setOnAction(e -> addToCartStage.close());

    // Add Components to Grid
    grid.add(productLabel, i: 0, i1: 0);
    grid.add(productComboBox, i: 1, i1: 0);
    grid.add(quantityLabel, i: 0, i1: 1);
    grid.add(quantityField, i: 1, i1: 1);
    grid.add(addButton, i: 0, i1: 2);
    grid.add(cancelButton, i: 1, i1: 2);

    Scene scene = new Scene(grid, v: 400, v1: 200);
    addToCartStage.setScene(scene);
    addToCartStage.show();
}
```

2-Show alert

for invalid inputs

```
private void showAlert(String message) {  27 usages
    Alert alert = new Alert(Alert.AlertType.INFORMATION);
    alert.setContentText(message);
    alert.showAndWait();
}
```

Thank you !!