Even Semester (2023)

BINUS UNIVERSITY

BINUS INTERNATIONAL

Final Project Cover Letter

(Individual Work)

Student Information:

Surname: Kusnadi **Given Name:** Clarissa Audrey Fabiola **Student ID:** 2602118490

Course Code : COMP6699001 **Course Name :** Object-Oriented Programming

Class : L2AC **Lecturer :** Jude Joseph Lamug Martinez, MCS

Type of Assignment: Final Project

Submission Pattern

Due Date : 16 June 2023 **Submission Date :** 16 June 2023

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating, and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity, and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept, and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student:

Clarissa Audrey Fabiola Kusnadi

Table of Contents

Introduction

Background

Students are expected and encouraged to create and design a comprehensive application that is beyond the topic of the Object-Oriented Programming course, by applying all the lessons that we have learned about the Java Object-Oriented Programming language and problem solving.

Following careful consideration and extensive research, I have elected to develop a bookstore management system named 'Literarium'. This system will be constructed utilizing programming libraries that have not been previously addressed during the current semester of Object-Oriented Programming. Specifically, I will be employing Java Swing and Java AWT to create a Graphic User Interface (GUI).

Problem Identification

The majority of contemporary activities are regulated by software programmes that are computer-operated. As human lifestyles have become increasingly intricate, a diverse range of software systems have infiltrated every aspect of human interaction. These systems include real-time, business, simulation, embedded, web-based, personal, and more recently, artificial intelligence software.

Based on the aforementioned information, it is possible to regulate the management and upkeep of a bookstore through the utilization of effective software. The primary objective of the project is to develop software that is efficient and reliable in managing the book and customer inventory, as well as enhancing the overall customer shopping experience of a bookstore.

In practical settings, there is a tendency to favor automated systems due to their ability to offer numerous advantages over manual processes. As previously stated, a system has been presented for the purpose of managing a bookstore. In the context of manual operations within a bookstore, a significant challenge is the inefficient utilization of time. The customer is required to invest a significant amount of time when seeking to purchase a book, as all associated tasks, including searching and purchasing, are delegated to staff members. In summary, the manual process is characterized by a significant lack of speed. The implementation of automation is expected to result in a reduction of the overall duration of the process.

When operating a bookstore, we often deal with large stores where the individual in charge of the management of the store is required to look after its maintenance through the relevant record documentation. As a result, there is a possibility of incorrect or flawed reports. In addition, the company has to hire additional personnel to ensure the completion of stationery maintenance. Furthermore, the organization incurs additional costs.

Therefore, once we become acquainted with this particular system, we will be capable of achieving the desired outcomes. The efficiency of the system will enhance the efficacy of communication with administrations and customers.

Project Specification

Program Name and Logo

- Literarium. Derived from the words “literary” and “-arium”. The term “literary” pertains to the realm of literature or written compositions, whereas the suffix “-arium” is commonly used to denote a place.
- The images attached below are the different versions of the logo designed for the application. The logo prominently showcases a vivid hue of golden yellow, evoking a sense of warmth, knowledge, and enlightenment. The symbol of an opened book symbolizes knowledge, imagination, and the exploration of different worlds through reading. Hovering above the book are sparkles, which add a touch of magic and wonder to the logo that represents the enchantment and inspiration that can be found within the pages of books. They symbolize the transformative experience of reading, where ideas and emotions sparkle and come to life.

Program Description

- Literarium is a Java Swing-based desktop application designed to simulate a digital bookstore. It provides a user-friendly graphical interface and supports two roles: Admin and Customer. In the Admin role, users can efficiently manage the bookstore's inventory and customer information. The Customer role enables users to browse books, add them to their bill, and print the bill for their purchases. The program incorporates various files and directories for effective data management. It includes classes such as Start, Login, BookManagement, CustomerManagement, Store, and Admin. Additionally, it utilizes text files like customer.txt and book.txt to store customer and book information. The 'Books' folder contains book cover images (JPG) and corresponding synopsis files (TXT).
- Literarium's goal is to provide its users with a full digital bookshop experience that allows them to interact with the virtual store easily. The platform has an easy-to-use interface that allows for simple navigation and interaction with the digital store, giving the users an ease of online book browsing and purchase along with its inventory management.

Program Flow Summary

- Admins have to have a registered account login credentials (declared in the program manually) in order to be able to log into the admin database. Once logged in,

administrators can navigate to the book management or customer management tabs. They may manage the book inventory database on the book management tab by adding, updating, or deleting book details. Similarly, they may manage the customer inventory database on the customer management page by adding, updating, or deleting customer info. After their task is completed, they can log out and exit the program.

- Customers who wish to use the system must login using the credentials stored in the customer database. Customer databases and authentication processes are managed by the admin. As previously stated, admins have the authority to create customer accounts and store their credentials in the customer inventory database. Once the customer is logged in, they have the option to browse the book via search or scroll through the book table. They can browse specific genres, authors, or titles to find the book they want. They can add the books they want to buy to their bill, and when they are done, they can print their bill, which contains the total cost of the selected books. After they are finished, they can log out and exit the program.

Program Libraries/Modules

- javax.swing: A library for creating graphical user interfaces (GUIs) in Java. It is used in the project for displaying message dialogs, manipulating table models, and creating graphical components for the user interface.
- java.awt: A library for creating GUI components and handling basic graphics operations in Java. It is used in the project for working with fonts, measuring font metrics, and managing images.
- java.io: A library for performing input and output operations in Java. It is used in the project for reading and writing files.
- java.nio.file: A library for performing file-related operations in Java. It is used in the project for copying files with the option to replace existing files.

Program Files

- 'Start.java': The start class serves as the entry point of the application and initializes the necessary components for the bookstore management system.
- 'Login.java': The login class handles the user authentication process, allowing authorized users to access the system.
- 'BookStore.java': The BookStore.java interface declares methods for retrieving and setting information used in the bookstore.
- 'BookManagement.java': The BookManagement class implements the BookStore interface and serves as a component responsible for managing books within the system.
- 'CustomerManagement.java': The CustomerManagement class implements the BookStore interface and serves as a component responsible for managing customers within the system.

- 'Store.java': The Store class handles the overall functionality of the bookstore for customers.
- 'Admin.java': The Admin class provides administrative functionalities for the bookstore management system, such as customer and book inventory.
- 'customer.txt': This file contains information about customers in a bookstore. It stores data such as customer IDs, names, passwords, and gender.
- 'book.txt': This file contains information about books in the bookstore. It stores data such as book IDs, titles, quantities, prices, and authors.
- 'Books' folder: This folder contains book cover images in JPG format and their corresponding synopsis in TXT files.

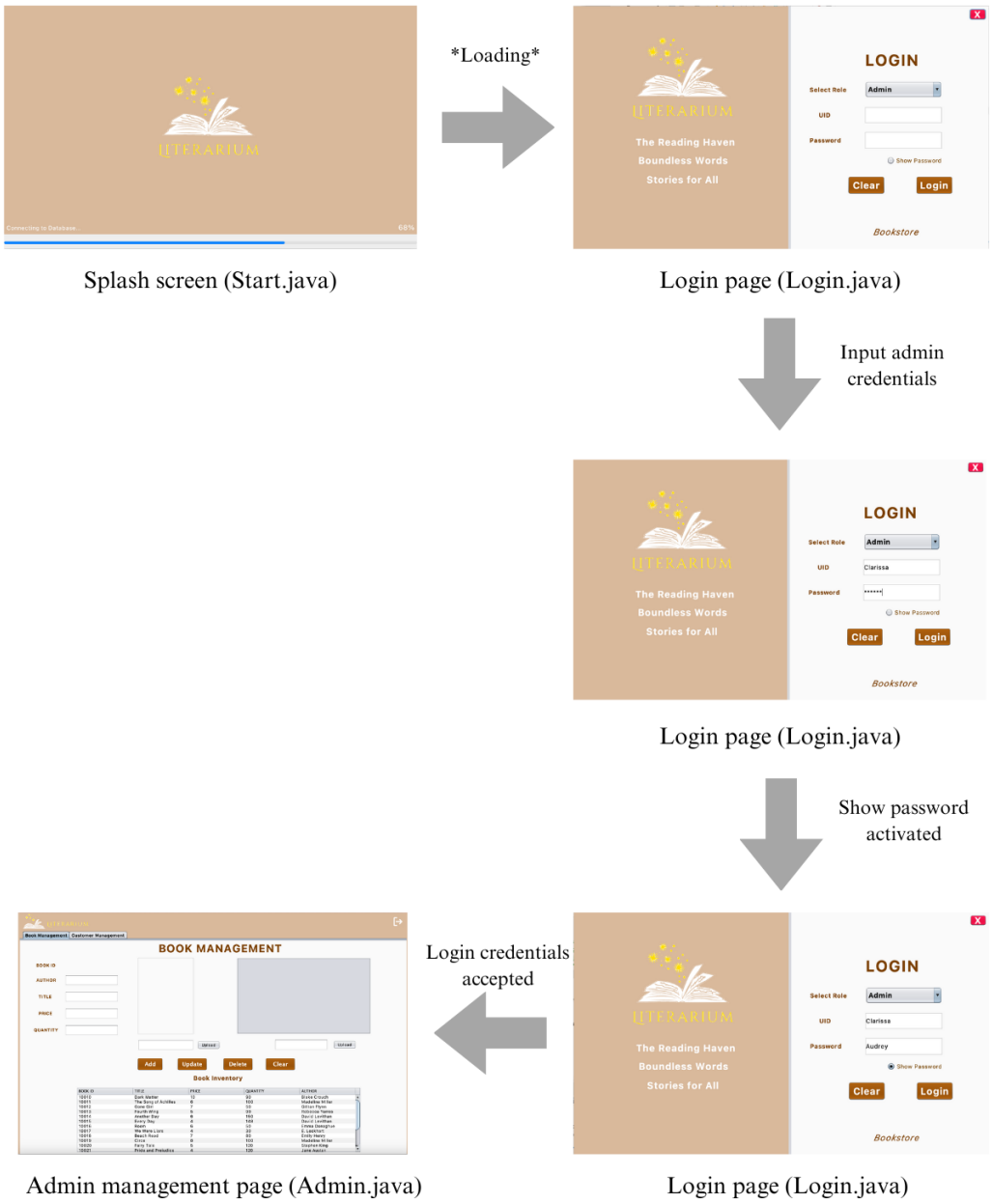
Solution Design

Program UML Diagram

- The program files Start.java, Login.java, Store.java, and Admin.java inherit the JFrame class in order to create a GUI, therefore showing the inheritance connection to the javax.swing.JFrame class.
- The program files BookManagement.java and CustomerManagement.java implement the BookStore.java interface, serving as components responsible for managing books and customers databases within the system.

Program Flow

- **Admin Login**



Splash screen (Start.java)

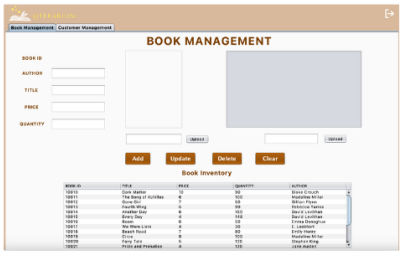
Login page (Login.java)

Login page (Login.java)

Login page (Login.java)

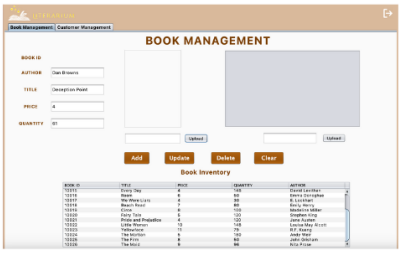
Admin management page (Admin.java)

- **Admin Book Management - Add**



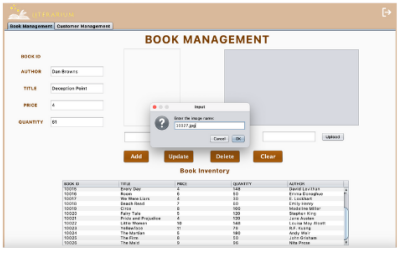
Book management tab (Admin.java)

Input new book details



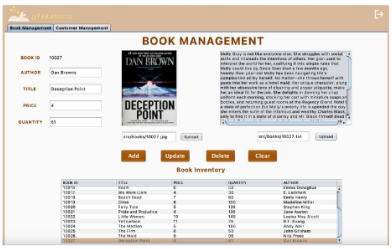
Book management tab (Admin.java)

Upload book's image



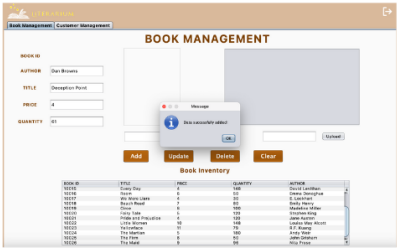
Book management tab (Admin.java)

Image uploaded successfully message



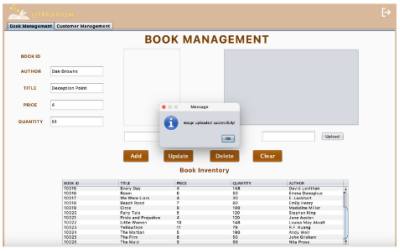
Book management tab (Admin.java)

Book data added to database



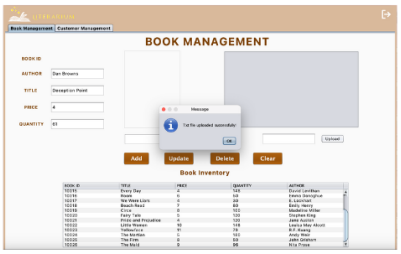
Book management tab (Admin.java)

Upload book's synopsis



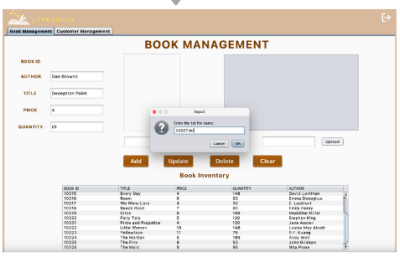
Book management tab (Admin.java)

Add button clicked



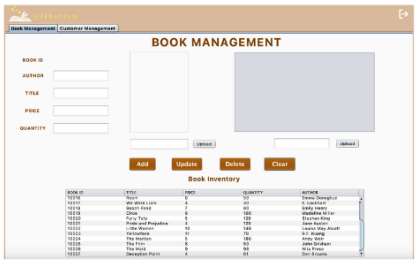
Book management tab (Admin.java)

txt uploaded successfully message



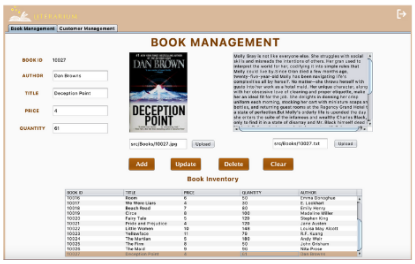
Book management tab (Admin.java)

- **Admin Book Management - Update**



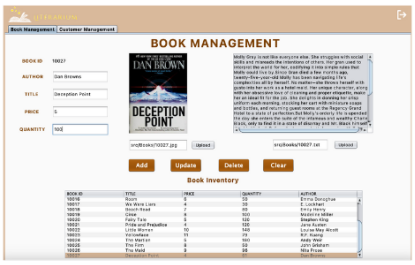
Book management tab (Admin.java)

Row clicked



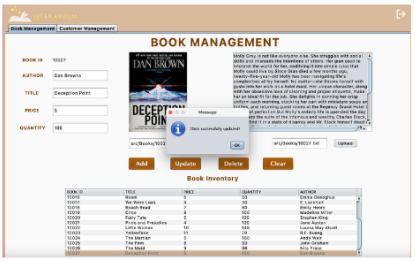
Book management tab (Admin.java)

Edit book details



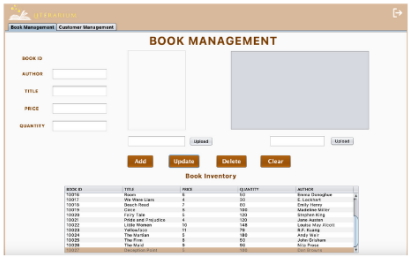
Book management tab (Admin.java)

Update button clicked



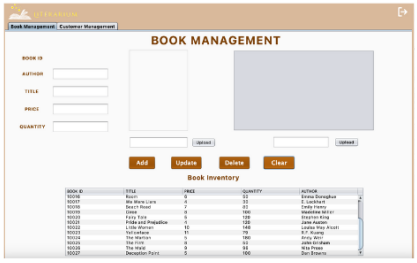
Book management tab (Admin.java)

Book data updated in database



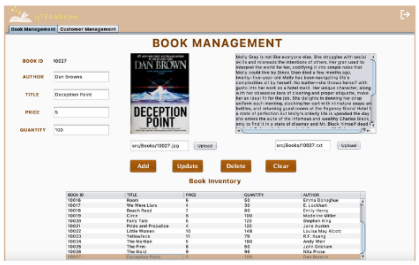
Book management tab (Admin.java)

- **Admin Book Management - Delete**



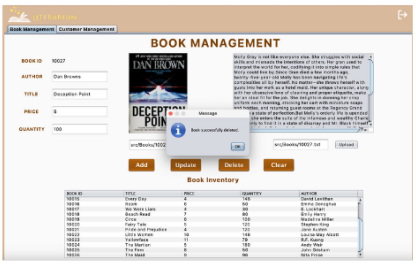
Book management tab (Admin.java)

Row clicked



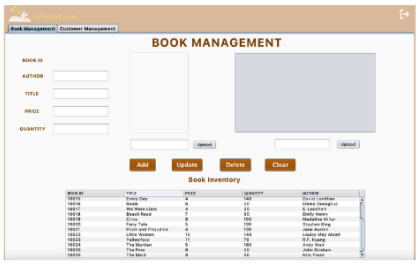
Book management tab (Admin.java)

Delete button clicked



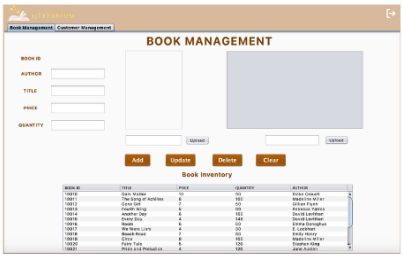
Book management tab (Admin.java)

Book data removed from database



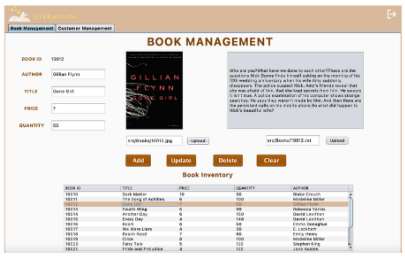
Book management tab (Admin.java)

- **Admin Book Management - Clear**



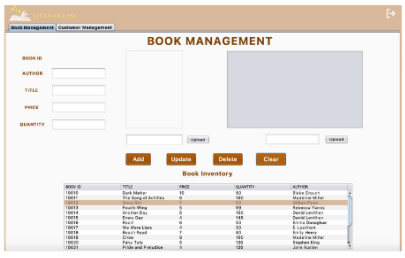
Book management tab (Admin.java)

Row clicked

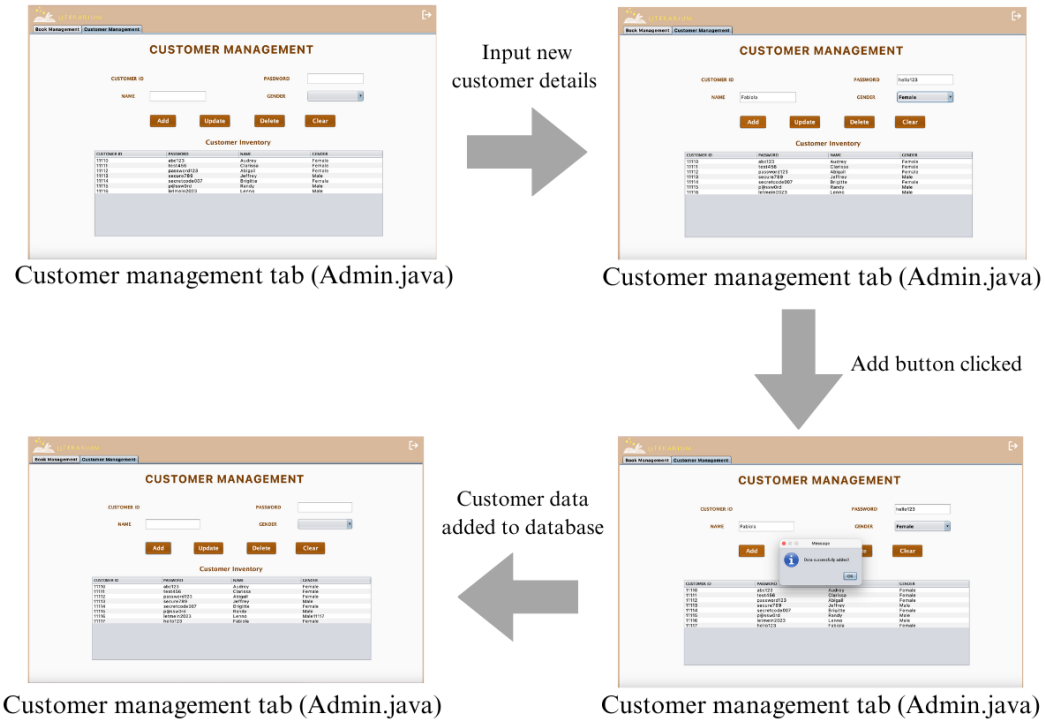


Book management tab (Admin.java)

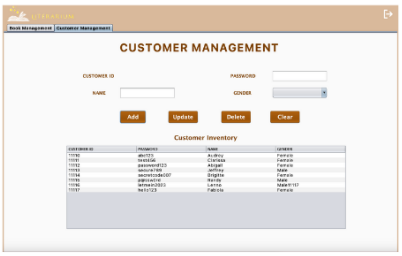
Clear button clicked



- **Admin Customer Management - Add**

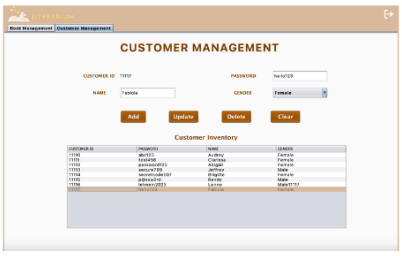


- **Admin Customer Management - Update**



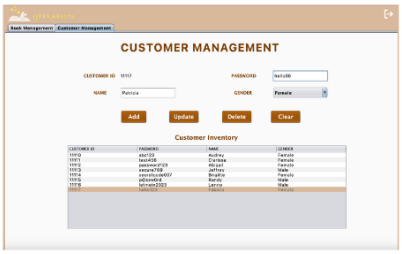
Customer management tab (Admin.java)

Row clicked



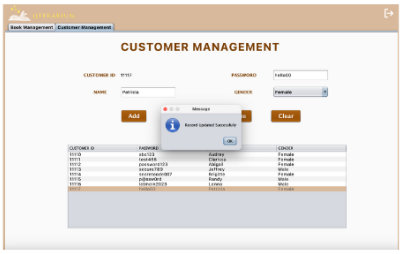
Customer management tab (Admin.java)

Edit customer details



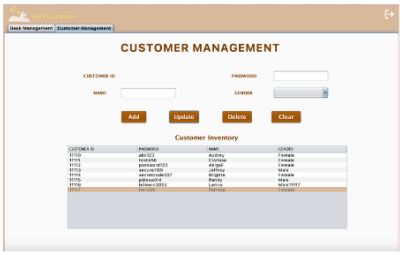
Customer management tab (Admin.java)

Update button clicked



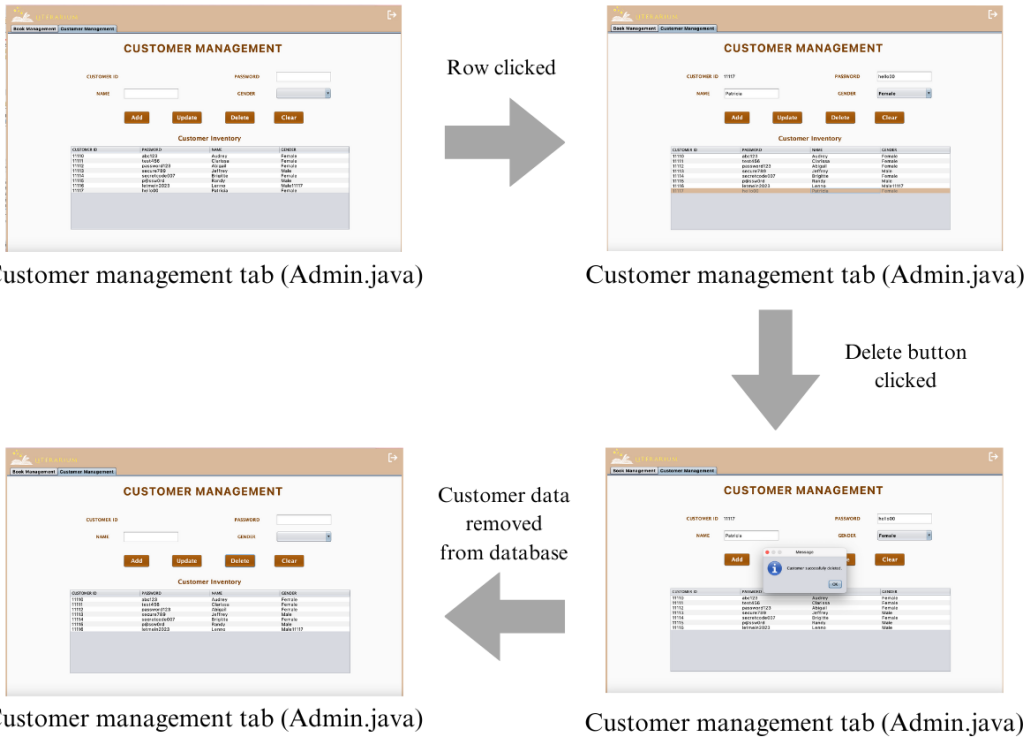
Customer management tab (Admin.java)

Customer data updated in database

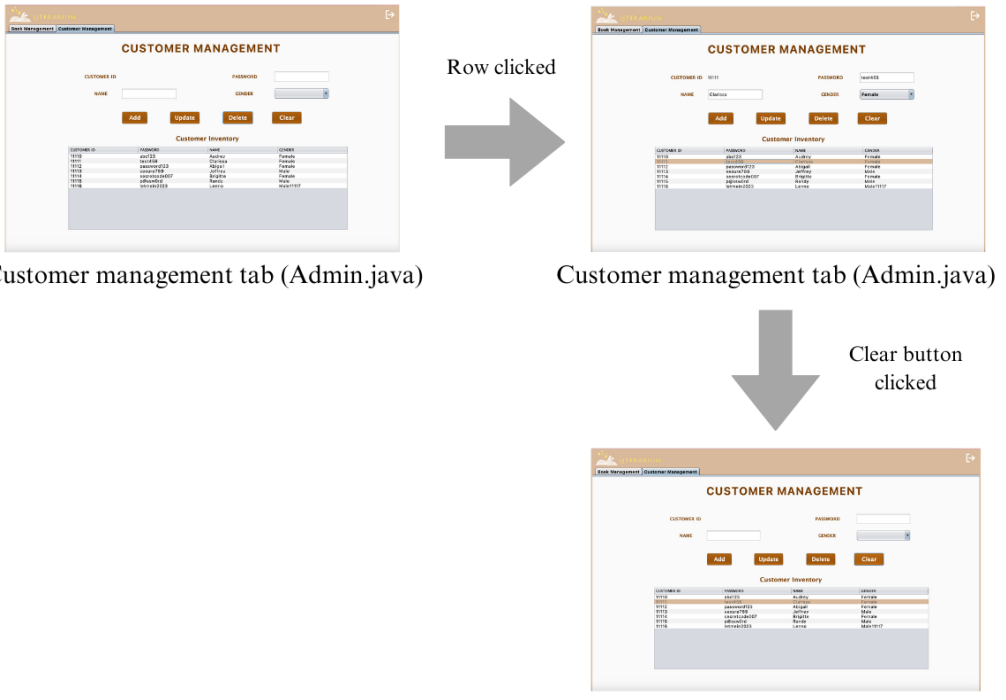


Customer management tab (Admin.java)

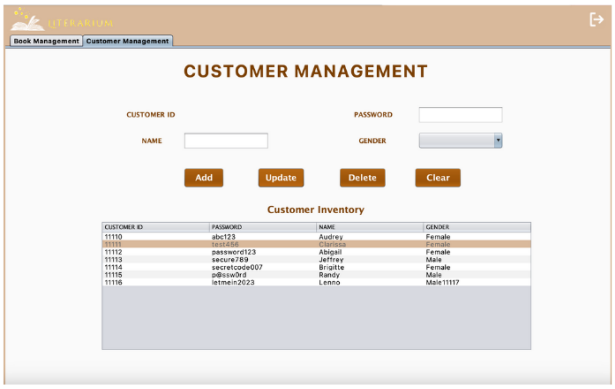
- **Admin Customer Management - Delete**



- **Admin Customer Management - Clear**

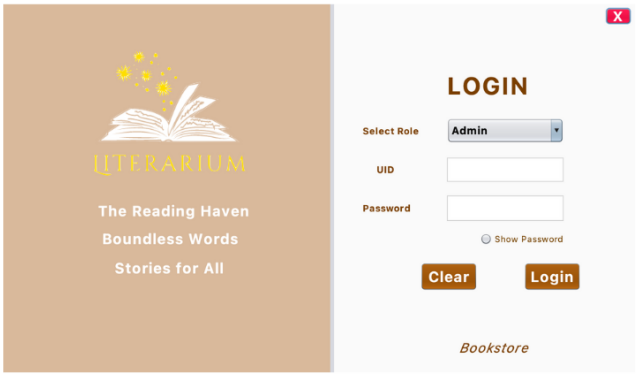


- **Admin Logout**



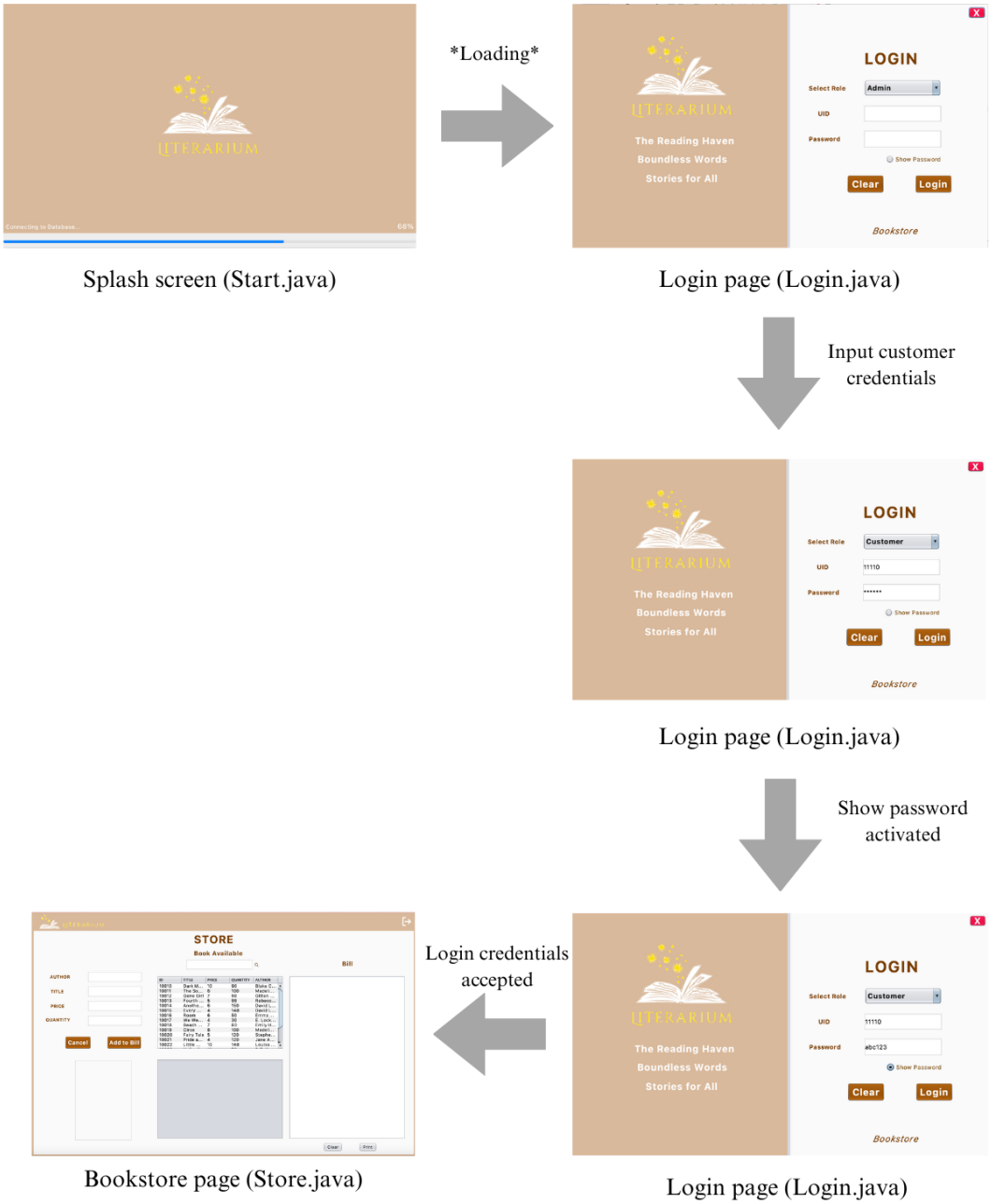
Customer management tab (Admin.java)

Logout button clicked

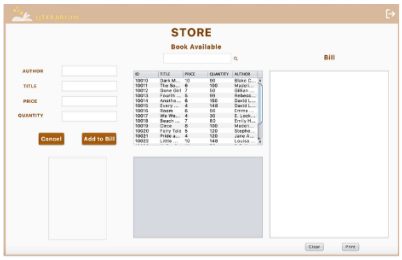


Login page (Login.java)

- **Customer Login**

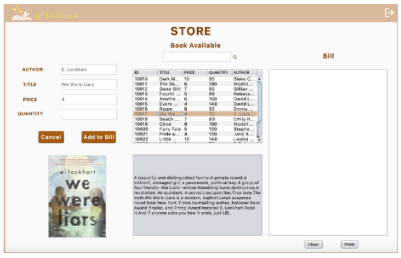


- **Customer Store Mechanism**



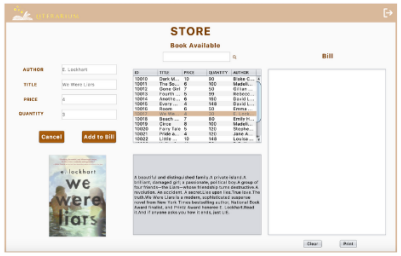
Bookstore page (Store.java)

Row clicked



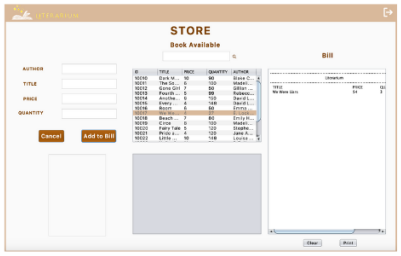
Bookstore page (Store.java)

Input number of quantity



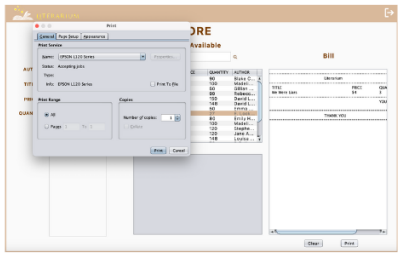
Bookstore page (Store.java)

Add to bill button clicked



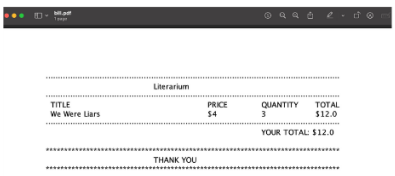
Bookstore page (Store.java)

Add to bill button clicked

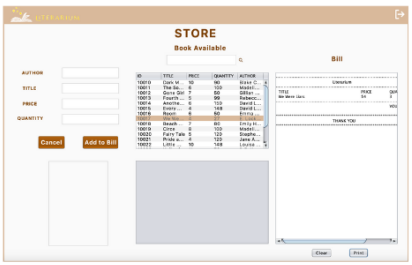


Bookstore page (Store.java)

Print to file

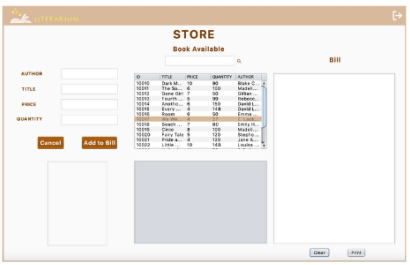


- **Customer Store - Clear Bill**



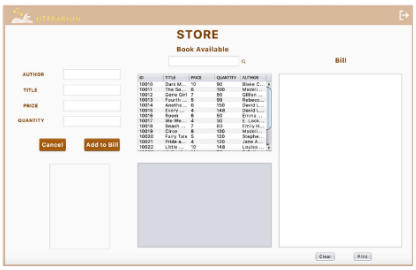
Bookstore page (Store.java)

Clear button clicked



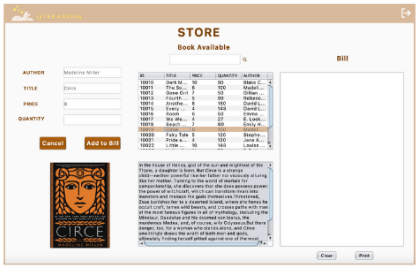
Bookstore page (Store.java)

- **Customer Store - Cancel**



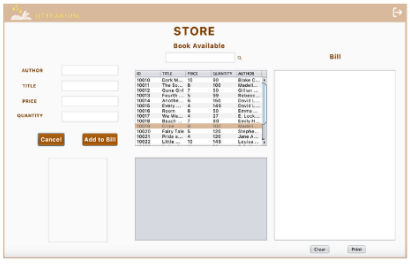
Bookstore page (Store.java)

Row clicked



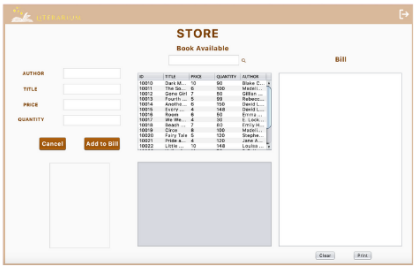
Bookstore page (Store.java)

Cancel button clicked



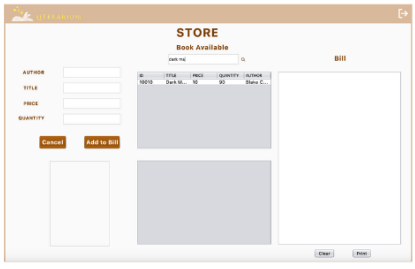
Bookstore page (Store.java)

- **Customer Store - Search**



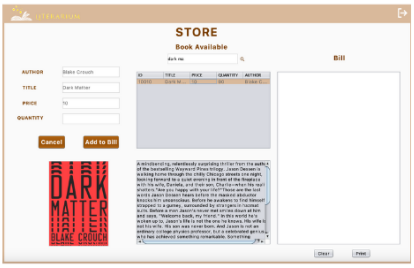
Bookstore page (Store.java)

Input search for books



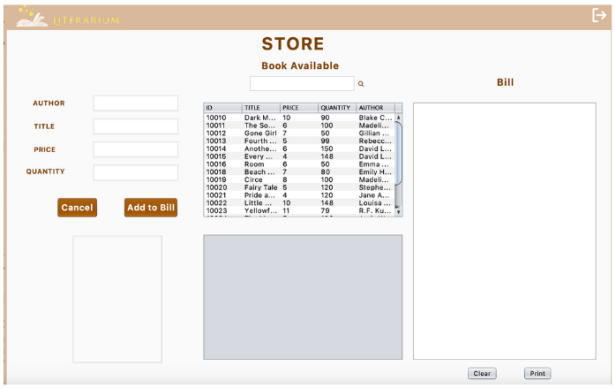
Bookstore page (Store.java)

Row clicked



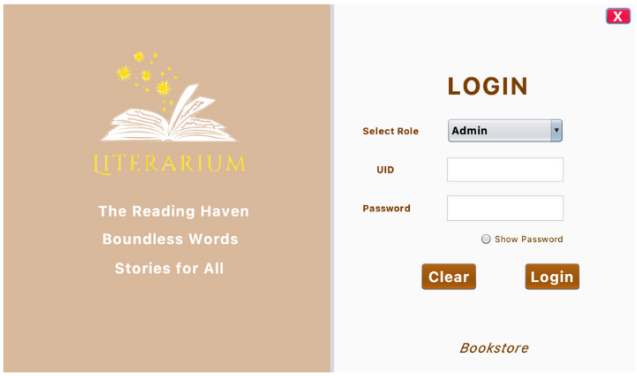
Bookstore page (Store.java)

- **Customer Logout**



Bookstore page (Store.java)

Logout button clicked

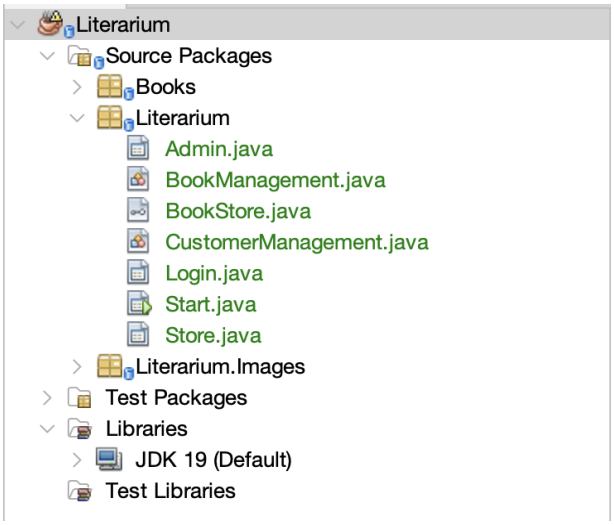


Login page (Login.java)

Code Design and Explanation

Project Structure

- All files, with the runnable file being Start.java, are located in the Source Packages package, inside the Literarium package display in Netbeans IDE.
- All images used in the files are included in the Literarium.images package.
- All external libraries that need to be imported are included in the Libraries folder.
- To ease display and avoid errors, it is **best to access the Literarium files with the Netbeans IDE.**



Start.java

- The Start class sets up the splash screen, handles the loading process, and transitions to the login screen of the application.

```
package Literarium;

import javax.swing.JOptionPane;
```

- Importing libraries and packages for the class, including:
    - javax.swing.JOptionPane: Dialog box for displaying messages.

```
/**
 *
 * @author Clarissa Audrey Fabiola
 */
public class Start extends javax.swing.JFrame {
    /**
     * Initializes the components of the splash screen page.
     */
    public Start() {
        initComponents();
    }

    /**
     * This method is called from within the constructor to initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is always
     * regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    Generated Code
```

- Creating the Start class and inheriting it to the JFrame class.
- Front-end design setup and code customization to set up the GUI. Code is generated by Netbeans and designed by me with the Netbeans design console.

```java
    /**
     * The main method of the program.
     * @param args
     */
    public static void main(String args[]) {
        Start splashScreen = new Start();

        /* Set the Nimbus look and feel */
        Look and feel setting code (optional)


        // Starting the main program
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                splashScreen.setVisible(b: true);
            }
        });

        try {
            for (int i = 0; i <= 100; i++) {
                Thread.sleep(millis:2);
                splashScreen.progressBar.setValue(n: i);
                splashScreen.percentage.setText(Integer.toString(i) + "%");

                // Display specific loading messages at different loading percentage
                if (i == 10) {
                    splashScreen.loadingLable.setText(text: "Turning on Modules...");
                }
                if (i == 20) {
                    splashScreen.loadingLable.setText(text: "Loading Modules...");
                }
                if (i == 50) {
                    splashScreen.loadingLable.setText(text: "Connecting to Database...");
                }
                if (i == 70) {
                    splashScreen.loadingLable.setText(text: "Connection Successful!");
                }
                if (i == 80) {
                    splashScreen.loadingLable.setText(text: "Launching Application...");
                }
            }
        } catch (Exception e) {
            JOptionPane.showMessageDialog(parentComponent: null, message: e);
        }
        new Login().setVisible(b: true);

        splashScreen.dispose();
    }
```

- The main(String[] args) method is the entry point of the program. It creates and displays the splash screen by creating an instance of the Start class. It sets the look and feel of the application, updates the progress bar and loading label, and handles any exceptions that occur during the loading process. Once the loading is complete, the splash screen is closed, and the login screen is opened by creating an instance of the Login class.

```java
    // Variables declaration - do not modify
    private javax.swing.JLabel loadingLable;
    private javax.swing.JLabel logo;
    private javax.swing.JLabel percentage;
    private javax.swing.JProgressBar progressBar;
    private javax.swing.JPanel splashScreenPanel;
    // End of variables declaration
}
```

- Variables declaration section in the Start class.

Login.java

- The Login class is responsible for handling the login functionality of the application.

```
package Literarium;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import javax.swing.JOptionPane;
```

- Importing libraries and packages for the class, including:
    - java.io.BufferedReader: Reading from text files.
    - java.io.IOException and java.io.FileNotFoundException: Handling input/output operations and file exceptions.
    - java.io.FileReader: To read character files.
    - javax.swing.JOptionPane: Dialog box for displaying messages.

```
/**
 *
 * @author Clarissa Audrey Fabiola
 */
public class Login extends javax.swing.JFrame {

    @SuppressWarnings("unchecked")
    Generated Code

    /**
     * Initializes the components of the login page.
     */
    public Login() {
        initComponents();
    }
```

- Creating the Login class and inheriting it to the JFrame class.
- Front-end design setup and code customization to set up the GUI. Code is generated by Netbeans and designed by me with the Netbeans design console.

```
    /**
     * This represents a user database that stores user IDs and passwords.
     */
    private final int MAX_USERS = 15;

    String[] ids = new String[MAX_USERS];
    String[] passwords = new String[MAX_USERS];

    // Counter variable to keep track of the number of users
    private static int userCount = 0;
```

- Declares variables and arrays to represent a user database that stores user IDs and passwords. It includes a constant for the maximum number of users, arrays to store the IDs and passwords, and a counter to track the number of users in the database.

```java
/**
 * loginActionPerformed
 * This method is called when the user clicks the "Login" button.
 * @param evt
 */
private void loginActionPerformed(java.awt.event.ActionEvent evt) {
    String IDText = id.getText();
    String passwordText = new String(value: password.getPassword());
    String selectedRole = roleComboBox.getSelectedItem().toString();

    if (selectedRole.equals(anObject: "Admin")) {
        if (IDText.equals(anObject: "Clarissa") && passwordText.equals(anObject: "Audrey")) {
            new Admin().setVisible(b: true);
            dispose();
        } else {
            JOptionPane.showMessageDialog(parentComponent: this, message: "Invalid Admin ID or Password.");
        }

    } else if (selectedRole.equals(anObject: "Customer")) {
        boolean credentialsFound = false;
        convertArray();

        // Search for matching ID and password in the user data arrays
        for (int i = 0; i < userCount && !credentialsFound; i++) {
            if (ids[i] != null && passwords[i] != null) { // Ensures that we don't encounter null values causing NullPointerException
                if (ids[i].equals(anObject: IDText) && passwords[i].equals(anObject: passwordText)) {
                    credentialsFound = true;
                }
            }
        }

        if (credentialsFound) {
            new Store().setVisible(b: true);
            dispose();
        } else {
            JOptionPane.showMessageDialog(parentComponent: this, message: "Invalid Customer ID or Password.");
        }
    }
}
```

- The loginActionPerformed() method is called when the user clicks the "Login" button. It retrieves the entered ID, password, and selected role from the corresponding components. If the selected role is "Admin" and the entered ID (“Clarissa”) and password (“Audrey”) match the predefined values, the Admin page is displayed; otherwise, a pop-up message dialog will appear and an invalid admin ID or password message is shown.
- Meanwhile, if the selected role is "Customer," it searches for a matching ID and password in the user database. If a match is found, the Store page is displayed; otherwise, a pop-up message dialog will appear and an invalid customer ID or password message is shown.

```java
/**
 * convertArray
 * This method reads the data from the "customer.txt" file and separates the password and ID.
 */
protected void convertArray() {
    try {
        String fileName = "customer.txt";
        FileReader fr = new FileReader(fileName);
        BufferedReader br = new BufferedReader(in: fr);

        String line;

        while ((line = br.readLine()) != null) {
            String[] parts = line.split(regex: "/");
            if (parts.length >= 2) {
                String id = parts[0];
                String password = parts[1];

                ids[userCount] = id;
                passwords[userCount] = password;
                userCount++;
            }
        }
        } catch (FileNotFoundException e) {
            JOptionPane.showMessageDialog(parentComponent: null, message: "File not found.");
        } catch (IOException e) {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Error reading the file.");
        }
}
```

- The convertArray() method reads data from the "customer.txt" file, separates the ID and password, and stores them in the ids and passwords arrays, respectively. This method is used to populate the user database.

```
    /**
     * clearActionPerformed
     * This method is called when the user clicks the "Clear" button to clear inputted credentials.
     * @param evt
     */
    private void clearActionPerformed(java.awt.event.ActionEvent evt) {
        id.setText(t: null);
        password.setText(t: null);
    }
```

- The clearActionPerformed() method is called when the user clicks the "Clear" button to clear the inputted credentials. It sets the ID and password fields to null.

```
    /**
     * showPasswordActionPerformed
     * This method is called when the user clicks on the radio box for showing password.
     * @param evt
     */
    private void showPasswordActionPerformed(java.awt.event.ActionEvent evt) {
        if (showPassword.isSelected()) {
            password.setEchoChar((char)0);
        } else {
            password.setEchoChar(c: '*');
        }
    }
```

- The showPasswordActionPerformed() method is called when the user clicks on the checkbox to show the password. If the checkbox is selected, it sets the EchoChar of the password field to 0, displaying the password as plaintext. If the checkbox is not selected, it sets the EchoChar back to '*', hiding the password.

```
    /**
     * closeActionPerformed
     * This method is called when the user clicks the close "X" button.
     * @param evt
     */
    private void closeActionPerformed(java.awt.event.ActionEvent evt) {
        System.exit(status:0);
    }
```

- The closeActionPerformed() method is called when the user clicks the close "X" button. It exits the application by calling System.exit(0).

```
    // Variables declaration - do not modify
    private javax.swing.JButton clear;
    private javax.swing.JButton close;
    private javax.swing.JPanel decoPanel;
    private javax.swing.JLabel descText;
    private javax.swing.JTextField id;
    private javax.swing.JLabel idSubtitle;
    private javax.swing.JPasswordField jPasswordField1;
    private javax.swing.JButton login;
    private javax.swing.JPanel loginPanel;
    private javax.swing.JLabel logo;
    private javax.swing.JLabel motto1;
    private javax.swing.JLabel motto2;
    private javax.swing.JLabel motto3;
    private javax.swing.JLabel pageTitle;
    private javax.swing.JPasswordField password;
    private javax.swing.JLabel passwordSubtitle;
    private javax.swing.JComboBox<String> roleComboBox;
    private javax.swing.JLabel roleSubtitle;
    private javax.swing.JRadioButton showPassword;
    // End of variables declaration
}
```

- Variables declaration section in the Login class.

BookStore.java

- The BookStore interface defines a contract for classes that represent a bookstore, specifying methods for accessing and manipulating bookstore information.

```
package Literarium;
```

- Importing package for the class.

```
/**
 *
 * @author Clarissa Audrey Fabiola
 */
public interface BookStore{
    public String getId();
    public void setId(String id);
    public String getName();
    public void setName(String name);
}
```

- Declares two getter methods: getId() and getName(), both of which are abstract methods. These methods are responsible for getting the ID and name of a bookstore, respectively.
- Declares two setter methods: setId(String id) and setName(String name). These methods are used to set the ID and name of a bookstore.

BookManagement.java

- The BookManagement class is an implementation of the BookStore interface and represents a class responsible for managing book information.

```
package Literarium;
```

- Importing package for the class.

```
/**
 *
 * @author Clarissa Audrey Fabiola
 */
public class BookManagement implements BookStore{
    private String id, name, qty, price, author;

    @Override
    public String getId() {
        return id;
    }

    @Override
    public void setId(String id) {
        this.id = id;
    }

    @Override
    public String getName() {
        return name;
    }

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

    public String getQty() {
        return qty;
    }

    public void setQty(String qty) {
        this.qty = qty;
    }

    public String getPrice() {
        return price;
    }

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

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }
}
```

- The BookManagement class has getter and setter methods for id, name, qty (quantity), price, and author properties, allowing access to and modification of these attributes. These methods enable retrieving and setting values for each property of a book in the book management system.

CustomerManagement.java

- The CustomerManagement class is an implementation of the BookStore interface and represents a class responsible for managing customer information.

```
package Literarium;
```

- Importing package for the class.

```
/**
 *
 * @author Clarissa Audrey Fabiola
 */
public class CustomerManagement implements BookStore{
    private String id, name, custPassword, custGender;

    @Override
    public String getName() {
        return name;
    }

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

    @Override
    public String getId() {
        return id;
    }

    @Override
    public void setId(String id) {
        this.id = id;
    }

    public String getcustPassword() {
        return custPassword;
    }

    public void setcustPassword(String custPassword) {
        this.custPassword = custPassword;
    }

    public String getcustGender() {
        return custGender;
    }

    public void setcustGender(String custGender) {
        this.custGender = custGender;
    }
}
```

- The CustomerManagement class has getter and setter methods for id, name, custPassword (customer password), and custGender (customer gender) properties, allowing access to and modification of these attributes. These methods enable retrieving and setting values for each property of a customer in the customer management system.

Store.java

- The Store class is responsible for handling the store functionality of the application.

```
package Literarium;

import javax.swing.JOptionPane;
import javax.swing.RowFilter;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableRowSorter;
import java.awt.Image;
import javax.swing.ImageIcon;
import java.io.IOException;
import java.io.FileNotFoundException;
import java.awt.Font;
import java.awt.FontMetrics;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
```

- Importing libraries and packages for the class, including:
    - javax.swing.JOptionPane: Dialog box for displaying messages.
    - javax.swing.RowFilter and javax.swing.table.DefaultTableModel: Handling and filtering table data.
    - javax.swing.table.TableRowSorter: Sorting and filtering table rows.
    - java.awt.Image and javax.swing.ImageIcon: Manipulating and displaying images.
    - java.io.IOException and java.io.FileNotFoundException: Handling input/output operations and file exceptions.
    - java.awt.Font and java.awt.FontMetrics: Working with fonts and font metrics.
    - java.io.BufferedReader and java.io.BufferedWriter: Reading from and writing to text files.
    - java.io.File, java.io.FileReader, and java.io.FileWriter: Handling file operations.

```
/**
 *
 * @author Clarissa Audrey Fabiola
 */
public class Store extends javax.swing.JFrame {
    /**
     * This method is called from within the constructor to initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is always
     * regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    Generated Code

    /**
     * Initializes the components of the store page.
     */
    public Store() {
        initComponents();

        // Convert book.txt into rows of a table.
        try {
            String fileName = "book.txt";
            FileReader fr = new FileReader(fileName);
            BufferedReader br = new BufferedReader(in: fr);
            DefaultTableModel model = (DefaultTableModel) bookStoreTable.getModel();

            String line;

            while ((line = br.readLine()) != null) {
                if (line != null) {
                    String[] dataRow = line.split(regex: "/");
                    model.addRow(rowData: dataRow);
                }
            }
        } catch (FileNotFoundException e) {
                JOptionPane.showMessageDialog(parentComponent: null, message: "File not found.");
        } catch (IOException e) {
                JOptionPane.showMessageDialog(parentComponent: null, message: "Error reading the file.");
        }
    }
```

- Creating the Store class and inheriting it to the JFrame class.
- Front-end design setup and code customization to set up the GUI. Code is generated by Netbeans and designed by me with the Netbeans design console.
- Reads the content of a file named "book.txt" and converts each line into rows of a table. It uses a FileReader and BufferedReader to read the file, and then retrieves the DefaultTableModel from a table component. Each line is split by the "/" delimiter, and the resulting array of data is added as a row to the table model. Any file-related exceptions, such as file not found or error reading the file, are caught and appropriate error messages are displayed using JOptionPane.showMessageDialog.

```
    /**
     * logoutActionPerformed
     * This method is called when the user clicks the logout icon button.
     * @param evt
     */
    private void logoutActionPerformed(java.awt.event.ActionEvent evt) {
        Login login = new Login();
        login.setVisible(b: true);
        dispose();
    }
```

- The logoutActionPerformed() method is called when the user clicks the logout button. It creates a new instance of the Login class, which represents the login page, makes it visible, and disposes of the current Store page.

```
/**
 * bookStoreTableMouseClicked
 * This method is called when the user clicks on a row in the bookStoreTable component.
 * Retrieves relevant data from the selected row index and populate labels and image icon.
 * @param evt The mouse click event.
 */
```

```
private void bookStoreTableMouseClicked(java.awt.event.MouseEvent evt) {
    int index = bookStoreTable.getSelectedRow();
    DefaultTableModel model = (DefaultTableModel) bookStoreTable.getModel();

    id = model.getValueAt(row:index, column:0).toString();
    bookTitleLabel.setText(t: model.getValueAt(row:index, column:1).toString());
    bookPriceLabel.setText(t: model.getValueAt(row:index, column:2).toString());
    bookAuthorLabel.setText(t: model.getValueAt(row:index, column:4).toString());

    String bookImage = "";
    String bookSynopsis = "";

    try {
        FileReader fr = new FileReader(fileName: "book.txt");
        BufferedReader br = new BufferedReader(in: fr);

        String line;
        String IDData = model.getValueAt(row:index, column:0).toString();

        // Add image and book synopsis according to the book id
        while ((line = br.readLine()) != null) {
            String[] parts = line.split(regex: "/");
            if (parts.length >= 1 && parts[0].equals(anObject: IDData)) {
                bookImage = "src/Books/" + parts[0] + ".jpg";
                String bookSynopsisFileName = "src/Books/" + parts[0] + ".txt";
                bookSynopsis = readTextFromFile(fileName: bookSynopsisFileName);
            }
        }
    } catch (FileNotFoundException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"File not found.");
    } catch (IOException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Error reading the file.");
    }

    // Ensure that there is a valid image path before attempting to load image
    if (!bookImage.isEmpty()) {
        try {
            ImageIcon icon = new ImageIcon(filename: bookImage);
            Image image = icon.getImage().getScaledInstance(width: bookImageLabel.getWidth(), height:bookImageLabel.getHeight(), hints: Image.SCALE_SMOOTH);
            bookImageLabel.setIcon(new ImageIcon(image));
        } catch (Exception e) {
            JOptionPane.showMessageDialog(parentComponent: null, message:"Error loading image");
        }
    }

    // Set the font and text style for the book synopsis
    bookSynopsisLabel.setText("<html>" + wrapText(text: bookSynopsis, width: bookSynopsisLabel.getWidth(), height:bookSynopsisLabel.getHeight()) + "</html>");
    Font labelFont = bookSynopsisLabel.getFont();
    bookSynopsisLabel.setFont(new Font(name: labelFont.getName(), style: Font.PLAIN, size: 13));
}
```

- The bookStoreTableMouseClicked() method is called when the user clicks on a row in the bookStoreTable. It retrieves relevant data from the selected row index and populates labels and an image icon accordingly. It reads the book image and synopsis from files based on the book's ID.

```
/**
 * readTextFromFile
 * This method reads the content from a file and returns it as a string.
 * @param fileName
 * @return
 */
private String readTextFromFile(String fileName) {
    StringBuilder sb = new StringBuilder();

    try {
        FileReader fr = new FileReader(fileName);
        BufferedReader br = new BufferedReader(in: fr);

        String line;

        while ((line = br.readLine()) != null) {
            sb.append(str:line);
        }
    } catch (FileNotFoundException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"File not found.");
    } catch (IOException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Error reading the file.");
    }

    return sb.toString();
}
```

- The readTextFromFile(String fileName) method reads the content from a file and returns it as a string. It takes the file name as a parameter, reads the file line by line, and appends each line to a StringBuilder. Finally, it returns the accumulated text as a string.

```
/**
 * wrapText
 * This method wraps a given text to fit within the specified width and height, adding line breaks if necessary.
 * @param text
 * @param width
 * @param height
 * @return
 */
private String wrapText(String text, int width, int height) {
    FontMetrics metrics = bookSynopsisLabel.getFontMetrics(font:bookSynopsisLabel.getFont());
    StringBuilder wrappedText = new StringBuilder();

    // Line width and height represents the maximum size available for displaying text on a single line
    int lineWidth = 0;

    // Split a string by space or whitespace characters
    String[] words = text.split(regex: "\\s+");

    for (String word : words) {
        int wordWidth = metrics.stringWidth(str:word);
        int wordWithSpaceWidth = wordWidth + metrics.stringWidth(str:" ");

        if (lineWidth + wordWithSpaceWidth <= width) {
            wrappedText.append(str:word).append(str:" ");
            lineWidth += wordWithSpaceWidth;
        } else {
            wrappedText.append(str:"<br>").append(str:word).append(str:" ");
            lineWidth = wordWithSpaceWidth;
        }
    }
    return wrappedText.toString();
}
```

- The wrapText(String text, int width, int height) method wraps a given text to fit within the specified width and height, adding line breaks if necessary. It takes the text, width, and height as parameters. It splits the text into words, calculates the width of each word, and checks if adding the word exceeds the available width. If it does, it appends a line break before adding the word. The wrapped text is returned as a string.

```
int count = 0;
String id;
double total;
double netTotal = 0;
```

```java
/**
 * addToBillActionPerformed
 * This method is called when the "Add to Bill" button is clicked.
 * To update the database and get user order to the bill.
 * @param evt
 */
private void addToBillActionPerformed(java.awt.event.ActionEvent evt) {
    if (bookTitleLabel.getText().isEmpty() || bookPriceLabel.getText().isEmpty() || bookQtyLabel.getText().isEmpty() || bookAuthorLabel.getText().isEmpty()) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Incomplete data.");
        return;
    }

    try {
        int qty = Integer.parseInt(s: bookQtyLabel.getText());
        if (qty <= 0) {
            JOptionPane.showMessageDialog(parentComponent: null, message:"Please enter only a positive integer.");
            return;
        }
    // If inputted value is not an integer
    } catch (NumberFormatException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Please enter only a positive integer.");
    }

    // Update quantity of the book
    DefaultTableModel model = (DefaultTableModel) bookStoreTable.getModel();
    BookStore[] bookStoreArray = new BookStore[1];
    bookStoreArray[0] = new BookManagement();

    Integer updatedQty = 0;

    // Iterating through the rows to find the product with matching book ID that was clicked from the table
    for (int i = 0; i < bookStoreTable.getRowCount(); i++) {
        if (model.getValueAt(row:i, column:0).equals(obj:id)) {
            updatedQty = Integer.parseInt(s: model.getValueAt(row:i, column:3).toString()) - Integer.parseInt(s: bookQtyLabel.getText());

            // Setting the modified book details to the BookManagement instance
            bookStoreArray[0].setId(id: model.getValueAt(row:i, column:0).toString().trim());
            bookStoreArray[0].setName(name: model.getValueAt(row:i, column:1).toString().trim());
            ((BookManagement) bookStoreArray[0]).setPrice(price: model.getValueAt(row:i, column:2).toString().trim());
            ((BookManagement) bookStoreArray[0]).setQty(qty:model.getValueAt(row:i, column:3).toString().trim());
            ((BookManagement) bookStoreArray[0]).setAuthor(author:model.getValueAt(row:i, column:4).toString().trim());

            // Update the table with updated value
            if (updatedQty >= 0) {
               model.setValueAt(aValue:updatedQty, row:i, column:3);
               break;
            } else {
                JOptionPane.showMessageDialog(parentComponent: null, message:"Unable to place order. Insufficient quantity available.");
                break;
            }
        }
    }

    // Adding order to the bill text area
    if (updatedQty >= 0) {
        count++;
        total = Double.parseDouble(s: bookPriceLabel.getText()) * (Double.valueOf(s: bookQtyLabel.getText()));
        netTotal += total;
        if (count == 1) {
            billText.setText(billText.getText() + "·····");
            billText.setText(billText.getText() + "\t\tLiterarium \n");
            billText.setText(billText.getText() + "·····");
            billText.setText(billText.getText() + "   " + "TITLE" + "\t\t\tPRICE" + "\tQUANTITY" + "\tTOTAL\n " + " " + String.format(format:"%-25s", args: bookTitleL
        }
        if (count > 1) {
            billText.setText(billText.getText() + "   " + String.format(format:"%-25s", args: bookTitleLabel.getText()) + "\t\t" + "$" + bookPriceLabel.getText() + "\

        }

        // Update book details to the database
        removeBookFromDataBase(id: bookStoreArray[0].getId());
        addBookToDataBase(
            id: bookStoreArray[0].getId(),
            name: bookStoreArray[0].getName(),
            price: ((BookManagement) bookStoreArray[0]).getPrice(),
            qty:updatedQty.toString(),
            author:((BookManagement) bookStoreArray[0]).getAuthor()
        );
    }
    bookAuthorLabel.setText(t: null);
    bookQtyLabel.setText(t: null);
    bookTitleLabel.setText(t: null);
    bookPriceLabel.setText(t: null);
    bookImageLabel.setIcon(icon: null);
    bookSynopsisLabel.setText(text: null);
}
```

- The addToBillActionPerformed() method is called when the "Add to Bill" button is clicked. It checks if the required fields (book title, price, quantity, and author) are not empty and if the quantity is a positive integer. It then updates the quantity of the book in the table, calculates the total price, and adds the order details to a bill text area (billText). It also updates the book details in the database.

```java
/**
 * removeBookFromDataBase
 * This method is used to remove specific line from the "book.txt" based on the book ID
 * user chose.
 * @param id
 */
private void removeBookFromDataBase(String id) {
    String fileName = "book.txt";
    String tempFile = "temp.txt";

    File oldFile = new File(pathname: fileName);
    File newFile = new File(pathname: tempFile);

    try {
        FileReader fr = new FileReader(file: oldFile);
        BufferedReader br = new BufferedReader(in: fr);

        FileWriter fw = new FileWriter(file: newFile);
        BufferedWriter bw = new BufferedWriter(out: fw);

        String line;

        while ((line = br.readLine()) != null) {
            int slashIndex = line.indexOf(str: "/");
            if (slashIndex != -1) {
                String specifiedId = line.substring(beginIndex: 0, endIndex: slashIndex);
                // Skipping the data line that corresponds to the ID user chose to order
                if (specifiedId.equals(anObject: id)) {
                    continue;
                }
            }
            // Write non-matching lines to "temp.txt"
            bw.write(str: line);
            bw.newLine();
        }
        bw.close();
        fr.close();

        oldFile.delete();
        newFile.renameTo(dest: oldFile);
//          System.out.println(new File(".").getAbsolutePath());
    } catch (IOException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Error in deleting data from txt file.");
    }
}
```

- The removeBookFromDataBase(String id) method is used to remove a specific line from the "book.txt" file based on the book ID. It reads the contents of the original file, skips the line with the specified ID, and writes the remaining lines to a temporary file. Finally, it replaces the original file with the temporary file.

```java
/**
 * addBookToDataBase
 * This method is used to add a new line of data to the "book.txt" file to update data.
 * @param id
 * @param name
 * @param price
 * @param qty
 * @param author
 */
private void addBookToDataBase(String id, String name, String price, String qty, String author) {
    try {
        // Append data to existing file
        FileWriter fw = new FileWriter(fileName: "book.txt", append:true);
        fw.write(id + "/" + name + "/" + price + "/" + qty + "/" + author);
        fw.write(str:System.getProperty(key:"line.separator"));
        fw.close();

    } catch (IOException ex) {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Error in adding data to txt file.");
    }
}
```

- The addBookToDataBase(String id, String name, String price, String qty, String author) method is used to add a new line of data to the "book.txt" file to update the book details. It appends the book details to the existing file.

```
    /**
     * cancelActionPerformed
     * This method is called when the "Cancel" button is clicked.
     * It resets the values of various UI components.
     * @param evt
     */
    private void cancelActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        bookAuthorLabel.setText(t: null);
        bookTitleLabel.setText(t: null);
        bookPriceLabel.setText(t: null);
        bookQtyLabel.setText(t: null);
        bookImageLabel.setIcon(icon: null);
        bookSynopsisLabel.setText(text: null);
    }
```

- The cancelActionPerformed() method is called when the "Cancel" button is clicked. It resets the values of various UI components (bookAuthorLabel, bookTitleLabel, bookPriceLabel, bookQtyLabel, bookImageLabel, bookSynopsisLabel).

```
/**
 * printBillActionPerformed
 * This method is called when the "Print" button is clicked.
 * Generates the bill PDF file and prompts the user to select a directory to save it.
 * @param evt
 */
private void printBillActionPerformed(java.awt.event.ActionEvent evt) {
    billText.setText(billText.getText() + "..........");
    billText.setText(billText.getText() + "\t\t\t\tYOUR TOTAL: $" + netTotal);
    billText.setText(billText.getText() + "\n\n**********\n");
    billText.setText(billText.getText() + "\t\tTHANK YOU \n");
    billText.setText(billText.getText() + "**********\n");

    try {
        billText.print();
    }
    catch (Exception e) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Failed to print the bill.");
    }
}
```

- The printBillActionPerformed() method is called when the "Generate PDF" button is clicked. It generates the bill PDF file by appending the final total and a thank you message to the billText area. Then it attempts to print the contents of billText.

```
    /**
     * This method is called when customer search in the search field.
     * @param evt
     */
    private void searchKeyReleased(java.awt.event.KeyEvent evt) {
        DefaultTableModel model = (DefaultTableModel) bookStoreTable.getModel();

        // Create a new TableRowSorter and associate it with the table model
        TableRowSorter<DefaultTableModel> trs = new TableRowSorter<>(model);
        bookStoreTable.setRowSorter(sorter:trs);

        // Apply the filter based on the search query (case-insensitive by using (?i))
        trs.setRowFilter(filter:RowFilter.regexFilter("(?i)" + search.getText()));
    }
```

- The searchKeyReleased() method is called when the customer types in the search field. It retrieves the table model, creates a new TableRowSorter associated with the model, and sets it as the row sorter for the table. It then applies a filter to the row sorter based on the search query, allowing case-insensitive matching.

```
    /**
     * clearBillActionPerformed
     * This method is called when the "Clear" button is clicked.
     * @param evt
     */
    private void clearBillActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
        billText.setText(t: null);
        netTotal = 0;
        count = 0;
    }
```

- The clearBillActionPerformed() method is called when the "Clear" button is clicked. It clears the contents of the billText field and resets the netTotal and count variables to their initial values, essentially resetting the bill calculation.

Admin.java

- The Admin class is responsible for handling the book and customer inventory management functionality of the application.

```
package Literarium;

import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Image;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;
import javax.swing.ImageIcon;
import javax.swing.JFileChooser;
import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;
```

- Importing libraries and packages for the class, including:
    - java.awt.Font and java.awt.FontMetrics: Font and font metrics handling.
    - java.awt.Image: Image manipulation and display.
    - java.io.BufferedReader and java.io.BufferedWriter: Buffered reading and writing of character streams.
    - java.io.File, java.io.FileReader, and java.io.FileWriter: File handling operations.
    - java.io.FileNotFoundException and java.io.IOException: Handling file-related exceptions.
    - java.nio.file.Files and java.nio.file.StandardCopyOption: File operations and copying files with options.
    - javax.swing.ImageIcon: Creating and displaying image icons.
    - javax.swing.JFileChooser: Dialog for file selection.
    - javax.swing.JOptionPane: Dialog boxes for displaying messages and obtaining user input.
    - javax.swing.table.DefaultTableModel: Managing data in a table form.

```
/**
 *
 * @author Clarissa Audrey Fabiola
 */
public class Admin extends javax.swing.JFrame {
    /**
     * This method is called from within the constructor to initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is always
     * regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    Generated Code

    /**
     * Initializes the components of the admin page.
     */
    public Admin() {
        initComponents();
```

- Creating the Admin class and inheriting it to the JFrame class.
- Front-end design setup and code customization to set up the GUI. Code is generated by Netbeans and designed by me with the Netbeans design console.

```
        // Book management
        // Convert book.txt into rows of a table.
        try {
            String fileName = "book.txt";
            FileReader fr = new FileReader(fileName);
            BufferedReader br = new BufferedReader(in: fr);
            DefaultTableModel model = (DefaultTableModel) bookInventoryTable.getModel();

            String line;

            while ((line = br.readLine()) != null) {
                if (line != null) {
                    String[] dataRow = line.split(regex: "/");
                    // Add the row to the table model
                    model.addRow(rowData: dataRow);
                }
            }

        } catch (FileNotFoundException e) {
                JOptionPane.showMessageDialog(parentComponent: null, message: "File not found.");
        } catch (IOException e) {
                JOptionPane.showMessageDialog(parentComponent: null, message: "Error reading the file.");
        }

        // Customer management
        // Convert customer.txt into rows of a table.
        try {
            String fileName = "customer.txt";
            FileReader fr = new FileReader(fileName);
            BufferedReader br = new BufferedReader(in: fr);
            DefaultTableModel model = (DefaultTableModel) customerInventoryTable.getModel();

            String line;

            while ((line = br.readLine()) != null) {
                if (line != null) {
                    String[] dataRow = line.split(regex: "/");
                    // Add the row to the table model
                    model.addRow(rowData: dataRow);
                }
            }
        } catch (FileNotFoundException e) {
                JOptionPane.showMessageDialog(parentComponent: null, message: "File not found.");
        } catch (IOException e) {
                JOptionPane.showMessageDialog(parentComponent: null, message: "Error reading the file.");
        }
    }
```

- Converting text files (book.txt and customer.txt) into rows of tables.
- For book management, it reads the book.txt file, retrieves the book inventory table model, and adds each line of the file as a data row to the table model.
- Similarly, for customer management, it reads the customer.txt file, retrieves the customer inventory table model, and adds the data rows accordingly.
- In case of any file-related issues, appropriate error messages are displayed using JOptionPane.showMessageDialog.

```java
/**
 * addBookActionPerformed
 * This method is called when the "Add" button on book management is clicked.
 * @param evt
 */
private void addBookActionPerformed(java.awt.event.ActionEvent evt) {
    if (bookTitleLabel.getText().isEmpty() || bookPriceLabel.getText().isEmpty() || bookQtyLabel.getText().isEmpty() || bookAuthorLabel.getText().isEmpty()) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Missing required information. Please complete all fields.");
        return;
    }

    try {
        int qty = Integer.parseInt(s: bookQtyLabel.getText());
        int price = Integer.parseInt(s: bookPriceLabel.getText());
        if (qty <= 0 || price <= 0) {
            JOptionPane.showMessageDialog(parentComponent: null, message:"Please enter only a positive integer.");
            return;
        }
    } catch (NumberFormatException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Please enter only a positive integer.");
        return;
    }

    DefaultTableModel model = (DefaultTableModel) bookInventoryTable.getModel();

    // Generate the new book ID
    String newId;
    if (model.getRowCount() == 0) {
        newId = "10010";
    } else {
        // Find the largest ID in the existing data and increment it by 1
        int maxId = 0;
        for (int i = 0; i < model.getRowCount(); i++) {
            String id = model.getValueAt(row:i, column:0).toString();
            int currentId = Integer.parseInt(s: id);
            if (currentId > maxId) {
                maxId = currentId;
            }
        }
        newId = String.valueOf(maxId + 1);
    }

    BookStore[] bookStoreArray = new BookStore[1];
    bookStoreArray[0] = new BookManagement();

    bookStoreArray[0].setId(id: newId);
    bookStoreArray[0].setName(name: bookTitleLabel.getText().trim());
    ((BookManagement) bookStoreArray[0]).setPrice(price: bookPriceLabel.getText().trim());
    ((BookManagement) bookStoreArray[0]).setQty(qty:bookQtyLabel.getText().trim());
    ((BookManagement) bookStoreArray[0]).setAuthor(author:bookAuthorLabel.getText().trim());
```

```java
    // Adding book details to the database
    addBookToDataBase(
        id: bookStoreArray[0].getId(),
        name: bookStoreArray[0].getName(),
        price: ((BookManagement) bookStoreArray[0]).getPrice(),
        qty:((BookManagement) bookStoreArray[0]).getQty(),
        author:((BookManagement) bookStoreArray[0]).getAuthor()
    );

    // Adding data to Jtable
    model.addRow(new Object[]{
        bookStoreArray[0].getId(),
        bookStoreArray[0].getName(),
        ((BookManagement) bookStoreArray[0]).getPrice(),
        ((BookManagement) bookStoreArray[0]).getQty(),
        ((BookManagement) bookStoreArray[0]).getAuthor()
    });

    JOptionPane.showMessageDialog(parentComponent: null, message:"Data successfully added!");

    bookIdLabel.setText(t: null);
    bookTitleLabel.setText(t: null);
    bookPriceLabel.setText(t: null);
    bookQtyLabel.setText(t: null);
    bookAuthorLabel.setText(t: null);
    bookImageLabel.setIcon(icon: null);
    bookSynopsisLabel.setText(text: null);
    bookImage.setText(t: null);
    bookSynopsis.setText(t: null);
}
```

- The addBookActionPerformed() method is called when the user clicks the "Add" button on the book management interface. It validates the required information, such as book title, price, quantity, and author, and displays error messages if any field is empty or if the quantity and price values are not positive integers. It generates a new book ID based on existing data and creates a BookManagement object to store the book details. The method adds the book to the database and updates the table model with the new book's information. It then displays a success message and clears the input fields and labels.

```java
/**
 * updateBookActionPerformed
 * This method is called when the "Update" button on book management is clicked.
 * @param evt
 */
private void updateBookActionPerformed(java.awt.event.ActionEvent evt) {
    BookStore[] bookStoreArray = new BookStore[1];
    bookStoreArray[0] = new BookManagement();

    bookStoreArray[0].setId(id: bookIdLabel.getText().trim());
    bookStoreArray[0].setName(name: bookTitleLabel.getText().trim());
    ((BookManagement) bookStoreArray[0]).setPrice(price: bookPriceLabel.getText().trim());
    ((BookManagement) bookStoreArray[0]).setQty(qty:bookQtyLabel.getText().trim());
    ((BookManagement) bookStoreArray[0]).setAuthor(author:bookAuthorLabel.getText().trim());

    // Validate quantity and price inputs
    try {
        int qty = Integer.parseInt(s: bookQtyLabel.getText());
        int price = Integer.parseInt(s: bookPriceLabel.getText());
        if (qty <= 0 || price <= 0) {
            JOptionPane.showMessageDialog(parentComponent: null, message:"Please enter only a positive integer.");
            return;
        }
    } catch (NumberFormatException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Please enter only a positive integer.");
        return;
    }

    // Update book details to the database
    removeBookFromDataBase(id: bookStoreArray[0].getId());
    addBookToDataBase(
        id: bookStoreArray[0].getId(),
        name: bookStoreArray[0].getName(),
        price: ((BookManagement) bookStoreArray[0]).getPrice(),
        qty:((BookManagement) bookStoreArray[0]).getQty(),
        author:((BookManagement) bookStoreArray[0]).getAuthor()
    );

    DefaultTableModel model = (DefaultTableModel) bookInventoryTable.getModel();

    // If a single row is selected
    if (bookInventoryTable.getSelectedRowCount() == 1) {
        if (bookIdLabel.getText().isEmpty() || bookTitleLabel.getText().isEmpty() || bookPriceLabel.getText().isEmpty() || bookQtyLabel.getText().toString().isE
            JOptionPane.showMessageDialog(parentComponent: null, message:"Missing required information. Please complete all fields.");
            return;
        }
```

```java
        // Update the table with updated value
        model.setValueAt(aValue:bookStoreArray[0].getId(), row:bookInventoryTable.getSelectedRow(), column:0);
        model.setValueAt(aValue:bookStoreArray[0].getName(), row:bookInventoryTable.getSelectedRow(), column:1);
        model.setValueAt(aValue:((BookManagement) bookStoreArray[0]).getPrice(), row:bookInventoryTable.getSelectedRow(), column:2);
        model.setValueAt(aValue:((BookManagement) bookStoreArray[0]).getQty(), row:bookInventoryTable.getSelectedRow(), column:3);
        model.setValueAt(aValue:((BookManagement) bookStoreArray[0]).getAuthor(), row:bookInventoryTable.getSelectedRow(), column:4);

        JOptionPane.showMessageDialog(parentComponent: null, message:"Data successfully updated!");
    } else {
        if (bookInventoryTable.getRowCount() == 0) {
            JOptionPane.showMessageDialog(parentComponent: null, message:"No data available in the table.");
        } else if (bookInventoryTable.getSelectedRowCount() == 0) {
            JOptionPane.showMessageDialog(parentComponent: null, message:"Please choose a single row to perform the update.");
        } else {
        // If multiple rows are selected
            JOptionPane.showMessageDialog(parentComponent: null, message:"Please choose a single row to perform the update.");
        }

    }

    bookIdLabel.setText(t: null);
    bookTitleLabel.setText(t: null);
    bookPriceLabel.setText(t: null);
    bookQtyLabel.setText(t: null);
    bookAuthorLabel.setText(t: null);
    bookImageLabel.setIcon(icon: null);
    bookSynopsisLabel.setText(text: null);
    bookImage.setText(t: null);
    bookSynopsis.setText(t: null);
}
```

- The updateBookActionPerformed() method is called when the user clicks the "Update" button in the book management interface. It performs the following tasks: retrieving input values, validating quantity and price inputs, updating book details in the database, updating the table if a single row is selected, displaying success or error messages accordingly, and resetting the input fields and labels.

```java
/**
 * bookInventoryTableMouseClicked
 * This method is called when the user clicks on a row in the bookInventoryTable component.
 * Retrieves relevant data from the selected row index and populate labels and image icon.
 * @param evt The mouse click event.
 */
private void bookInventoryTableMouseClicked(java.awt.event.MouseEvent evt) {
    // Set Data to their Text Field
    int index = bookInventoryTable.getSelectedRow();
    DefaultTableModel model = (DefaultTableModel) bookInventoryTable.getModel();

    bookIdLabel.setText(t: model.getValueAt(row: index, column: 0).toString());
    bookTitleLabel.setText(t: model.getValueAt(row: index, column: 1).toString());
    bookPriceLabel.setText(t: model.getValueAt(row: index, column: 2).toString());
    bookQtyLabel.setText(t: model.getValueAt(row: index, column: 3).toString());
    bookAuthorLabel.setText(t: model.getValueAt(row: index, column: 4).toString());

    String id = model.getValueAt(row: index, column: 0).toString();

    // Set image path if file exists
    String imagePath = "src/Books/" + id + ".jpg";
    File imageFile = new File(pathname: imagePath);
    if (imageFile.exists()) {
        bookImage.setText(t: imagePath);
        try {
            ImageIcon icon = new ImageIcon(filename: imagePath);
            Image image = icon.getImage().getScaledInstance(width: bookImageLabel.getWidth(), height: bookImageLabel.getHeight(), hints: Image.SCALE_SMOOTH);
            bookImageLabel.setIcon(new ImageIcon(image));
        } catch (Exception e) {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Error loading image");
        }
    } else {
        bookImage.setText(t: null);
        bookImageLabel.setIcon(icon: null);
    }

    // Set synopsis path if file exists
    String synopsisPath = "src/Books/" + id + ".txt";
    File synopsisFile = new File(pathname: synopsisPath);
    if (synopsisFile.exists()) {
        bookSynopsis.setText(t: synopsisPath);
        String bookSynopsis = readTextFromFile(fileName: synopsisPath);
        // Set the font and text style for the book synopsis
        bookSynopsisLabel.setText("<html>" + wrapText(text: bookSynopsis, width: bookSynopsisLabel.getWidth(), height: bookSynopsisLabel.getHeight()) + "</html>");
        Font labelFont = bookSynopsisLabel.getFont();
        bookSynopsisLabel.setFont(new Font(name: labelFont.getName(), style: Font.PLAIN, size: 13));
    } else {
        bookSynopsis.setText(t: null);
        bookSynopsisLabel.setText(text: null);
    }
}
```

- The bookInventoryTableMouseClicked() method is called when the user clicks on a row in the bookInventoryTable component. It performs the following tasks: retrieving the data from the selected row, populating the relevant labels and image icon with the retrieved data, setting the image path and displaying the corresponding image if it exists, setting the synopsis path and displaying the corresponding synopsis if it exists, formatting the book synopsis text to fit within the label's dimensions, and setting the font style for the book synopsis label. This method allows the user to view detailed information and associated images for a selected book in the table.

```java
/**
 * readTextFromFile
 * This method reads the content from a file and returns it as a string.
 * @param fileName
 * @return
 */
private String readTextFromFile(String fileName) {
    StringBuilder sb = new StringBuilder();

    try {
        FileReader fr = new FileReader(fileName);
        BufferedReader br = new BufferedReader(in: fr);

        String line;

        while ((line = br.readLine()) != null) {
            sb.append(str: line);
        }
    } catch (FileNotFoundException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message: "File not found.");
    } catch (IOException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Error reading the file.");
    }

    return sb.toString();
}
```

- The readTextFromFile(String fileName) method reads the content from a file and returns it as a string. It takes the file name as a parameter, reads the file line by line, and appends each line to a StringBuilder. Finally, it returns the accumulated text as a string.

```
/**
 * wrapText
 * This method wraps a given text to fit within the specified width and height,
 * adding line breaks if necessary.
 * @param text
 * @param width
 * @param height
 * @return
 */
private String wrapText(String text, int width, int height) {
    FontMetrics metrics = bookSynopsisLabel.getFontMetrics(font: bookSynopsisLabel.getFont());
    StringBuilder wrappedText = new StringBuilder();

    // Line width and height represents the maximum size available for displaying text on a single line
    int lineWidth = 0;

    // Split a string by space or whitespace characters
    String[] words = text.split(regex: "\\s+");

    for (String word : words) {
        int wordWidth = metrics.stringWidth(str:word);
        int wordWithSpaceWidth = wordWidth + metrics.stringWidth(str:" ");

        if (lineWidth + wordWithSpaceWidth <= width) {
            wrappedText.append(str:word).append(str:" ");
            lineWidth += wordWithSpaceWidth;
        } else {
            wrappedText.append(str:"<br>").append(str:word).append(str:" ");
            lineWidth = wordWithSpaceWidth;
        }
    }
    return wrappedText.toString();
}
```

- The wrapText(String text, int width, int height) method wraps a given text to fit within the specified width and height, adding line breaks if necessary. It takes the text, width, and height as parameters. It splits the text into words, calculates the width of each word, and checks if adding the word exceeds the available width. If it does, it appends a line break before adding the word. The wrapped text is returned as a string.

```
/**
 * clearBookActionPerformed
 * This method is called when the "Clear" button is clicked.
 * It resets the values of various UI components.
 * @param evt
 */
private void clearBookActionPerformed(java.awt.event.ActionEvent evt) {
    bookIdLabel.setText(t: null);
    bookTitleLabel.setText(t: null);
    bookPriceLabel.setText(t: null);
    bookQtyLabel.setText(t: null);
    bookAuthorLabel.setText(t: null);
    bookImageLabel.setIcon(icon: null);
    bookSynopsisLabel.setText(text: null);
    bookImage.setText(t: null);
    bookSynopsis.setText(t: null);
}
```

- The clearBookActionPerformed() method is called when the "Clear" button is clicked in the book management interface. Its purpose is to reset the values of various UI components related to book information. By setting the text and icons to null, it clears the content displayed in the book ID label, book title label, book price label, book quantity label, book author label, book image label, book synopsis label, book image text field, and book synopsis text field.

```java
/**
 * deleteBookActionPerformed
 * This method is called when the "Delete" button on book management is clicked.
 * @param evt
 */
private void deleteBookActionPerformed(java.awt.event.ActionEvent evt) {
    BookStore[] bookStoreArray = new BookStore[1];
    bookStoreArray[0] = new BookManagement();

    bookStoreArray[0].setId(id: bookIdLabel.getText().trim());
    bookStoreArray[0].setName(name: bookTitleLabel.getText().trim());
    ((BookManagement) bookStoreArray[0]).setPrice(price: bookPriceLabel.getText().trim());
    ((BookManagement) bookStoreArray[0]).setQty(qty:bookQtyLabel.getText().trim());
    ((BookManagement) bookStoreArray[0]).setAuthor(author:bookAuthorLabel.getText().trim());

    // Delete book data from the database
    removeBookFromDataBase(id: bookStoreArray[0].getId());

    DefaultTableModel model = (DefaultTableModel) bookInventoryTable.getModel();

    if (bookInventoryTable.getSelectedRowCount() == 1) {
        model.removeRow(row:bookInventoryTable.getSelectedRow());
        JOptionPane.showMessageDialog(parentComponent: null, message:"Book successfully deleted.");
    } else if (bookInventoryTable.getRowCount() == 0) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"No data available in the table.");
    } else if (customerInventoryTable.getSelectedRowCount() == 0) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Please choose a single row to perform the update.");
    // If multiple rows are selected
    } else {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Please choose a single row to perform the update.");
    }

    bookIdLabel.setText(t: null);
    bookTitleLabel.setText(t: null);
    bookPriceLabel.setText(t: null);
    bookQtyLabel.setText(t: null);
    bookAuthorLabel.setText(t: null);
    bookImageLabel.setIcon(icon: null);
    bookSynopsisLabel.setText(text: null);
    bookImage.setText(t: null);
    bookSynopsis.setText(t: null);
}
```

- The deleteBookActionPerformed() method is called when the "Delete" button is clicked in the book management interface. It retrieves the book information from the UI components and stores them in a BookStore object. It then proceeds to remove the corresponding book data from the database by calling the removeBookFromDataBase method. If a single row is selected in the book inventory table, the selected row is removed from the table model, and a success message is displayed. If no data is available in the table or if multiple rows are selected, appropriate error messages are displayed. Finally, it resets the values of the UI components related to book information, clearing the displayed content.

```java
/**
* uploadImageActionPerformed
* This method is called when the "Upload" button is clicked.
* It handles the process of uploading an image file to the destination directory.
* @param evt
*/
private void uploadImageActionPerformed(java.awt.event.ActionEvent evt) {
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setDialogTitle(dialogTitle: "Upload Image");
    int option = fileChooser.showOpenDialog(parent:this);

    // To check if the user has chosen a file or directory and confirmed their selection.
    if (option == JFileChooser.APPROVE_OPTION) {
        try {
            File sourceFile = fileChooser.getSelectedFile();
            String imageName = JOptionPane.showInputDialog(message: "Enter the image name:");
            String destinationPath = "src/Books/" + imageName;
            File destinationFile = new File(pathname: destinationPath);

            // Create the destination directory if it doesn't exist
            File destinationDir = destinationFile.getParentFile();
            if (!destinationDir.exists()) {
                destinationDir.mkdirs();
            }

            if (destinationFile.exists()) {
                int confirm = JOptionPane.showConfirmDialog(parentComponent: this, message: "A file with the same name already exists. Do you want to overwrite it?", tit

                // User chose not to overwrite the file
                if (confirm == JOptionPane.NO_OPTION) {
                    return;
                } else {
                    // Copy the source file to the destination file, replacing the existing file
                    Files.copy(source:sourceFile.toPath(), target:destinationFile.toPath(), options:StandardCopyOption.REPLACE_EXISTING);
                    JOptionPane.showMessageDialog(parentComponent: this, message: "Image uploaded successfully!");
                    return;
                }
            }

            // Copy the source file to the destination file
            Files.copy(source:sourceFile.toPath(), target:destinationFile.toPath());

            JOptionPane.showMessageDialog(parentComponent: this, message:"Image uploaded successfully!");
        } catch (IOException e) {
            JOptionPane.showMessageDialog(parentComponent: this, message: "Error in uploading the image.");
        }
    }
}
```

- The uploadImageActionPerformed() method is called when the "Upload" button is clicked. It initiates the process of uploading an image file to a specified destination directory. The method opens a file chooser dialog to allow the user to select an image file. If a file is chosen and confirmed, the method proceeds to handle the upload process. It creates the destination directory if it doesn't exist and checks if a file with the same name already exists in the destination. If a duplicate file exists, the method prompts the user to confirm whether to overwrite it or not. If confirmed, the source file is copied to the destination file, replacing the existing file if necessary. Finally, appropriate success or error messages are displayed to indicate the outcome of the upload process.

```java
/**
 * uploadSynopsisActionPerformed
 * This method is called when the "Upload" button is clicked.
 * It handles the process of uploading an txt file to the destination directory.
 * @param evt
 */
private void uploadSynopsisActionPerformed(java.awt.event.ActionEvent evt) {
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setDialogTitle(dialogTitle: "Upload txt file");
    int option = fileChooser.showOpenDialog(parent:this);

    if (option == JFileChooser.APPROVE_OPTION) {
        try {
            File sourceFile = fileChooser.getSelectedFile();
            String txtName = JOptionPane.showInputDialog(message: "Enter the txt file name:");
            String destinationPath = "src/Books/" + txtName;
            File destinationFile = new File(pathname: destinationPath);

            // Create the destination directory if it doesn't exist
            File destinationDir = destinationFile.getParentFile();
            if (!destinationDir.exists()) {
                destinationDir.mkdirs();
            }

            if (destinationFile.exists()) {
                int confirm = JOptionPane.showConfirmDialog(parentComponent: this, message: "A file with the same name already exists. Do you want to overwrite it?", tit

                // User chose not to overwrite the file
                if (confirm == JOptionPane.NO_OPTION) {
                    return;
                } else {
                    // Copy the source file to the destination file, replacing the existing file
                    Files.copy(source:sourceFile.toPath(), target:destinationFile.toPath(), options: StandardCopyOption.REPLACE_EXISTING);
                    JOptionPane.showMessageDialog(parentComponent: this, message: "Txt file uploaded successfully!");
                    return;
                }
            }

            // Copy the source file to the destination file
            Files.copy(source:sourceFile.toPath(), target:destinationFile.toPath());

            JOptionPane.showMessageDialog(parentComponent: this, message: "Txt file uploaded successfully!");
        } catch (IOException e) {
            JOptionPane.showMessageDialog(parentComponent: this, message: "Error in uploading the txt file.");
        }
    }
}
```

- The uploadSynopsisActionPerformed() method is called when the "Upload" button is clicked. It facilitates the process of uploading a .txt file to a specified destination directory. The method opens a file chooser dialog to allow the user to select a .txt file. If a file is chosen and confirmed, the method proceeds with the upload process. It creates the destination directory if it doesn't already exist and checks if a file with the same name already exists in the destination. If a duplicate file exists, the method prompts the user to confirm whether to overwrite it or not. If confirmed, the source file is copied to the destination file, replacing the existing file if necessary. Finally, appropriate success or error messages are displayed to indicate the outcome of the upload process.

```java
/**
 * removeBookFromDataBase
 * This method is used to remove specific line from the "book.txt" based on the book ID
 * user chose.
 * @param id
 */
private void removeBookFromDataBase(String id) {
    String fileName = "book.txt";
    String tempFile = "temp.txt";

    File oldFile = new File(pathname: fileName);
    File newFile = new File(pathname: tempFile);

    try {

        FileReader fr = new FileReader(file: oldFile);
        BufferedReader br = new BufferedReader(in: fr);

        FileWriter fw = new FileWriter(file: newFile);
        BufferedWriter bw = new BufferedWriter(out: fw);

        String line;

        while ((line = br.readLine()) != null) {
            int slashIndex = line.indexOf(str: "/");
            if (slashIndex != -1) {
                String specifiedId = line.substring(beginIndex: 0, endIndex: slashIndex);
                // Skipping the data line that corresponds to the ID user chose to order
                if (specifiedId.equals(anObject: id)) {
                    continue;
                }
            }
            // Write non-matching lines to "temp.txt"
            bw.write(str: line);
            bw.newLine();
        }
        bw.close();
        fr.close();

        oldFile.delete();
        newFile.renameTo(dest: oldFile);
//          System.out.println(new File(".").getAbsolutePath());
    } catch (IOException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Error in deleting data from txt file.");
    }
}
```

- The removeBookFromDataBase(String id) method is used to remove a specific line from the "book.txt" file based on the book ID. It reads the contents of the original file, skips the line with the specified ID, and writes the remaining lines to a temporary file. Finally, it replaces the original file with the temporary file.

```java
/**
 * addBookToDataBase
 * This method is used to add a new line of data to the "book.txt" file to update data.
 * @param id
 * @param name
 * @param price
 * @param qty
 * @param author
 */
    private void addBookToDataBase(String id, String name, String price, String qty, String author) {
    try {
        // Append data to existing file
        FileWriter fw = new FileWriter(fileName: "book.txt", append:true);
        fw.write(id + "/" + name + "/" + price + "/" + qty + "/" + author);
        fw.write(str:System.getProperty(key:"line.separator"));
        fw.close();

    } catch (IOException ex) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Error in adding data to txt file.");
    }
}
```

- The addBookToDataBase(String id, String name, String price, String qty, String author) method is used to add a new line of data to the "book.txt" file to update the book details. It appends the book details to the existing file.

```java
/**
 * addCustomerActionPerformed
 * This method is called when the "Add" button on customer management is clicked.
 * It handles the process of adding a new customer to the inventory.
 * @param evt
 */
private void addCustomerActionPerformed(java.awt.event.ActionEvent evt) {
    if (customerPasswordLabel.getText().isEmpty() || customerNameLabel.getText().isEmpty() || customerGender.getSelectedItem().toString().isEmpty()) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Missing required information. Please complete all fields.");
        return;
    }

    DefaultTableModel model = (DefaultTableModel) customerInventoryTable.getModel();

    // Check if maximum customer account count has been reached
    if (model.getRowCount() >= 15) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Maximum customer account reached!");
        return;
    }

    // Generate the new Customer ID
    String newId;
    if (model.getRowCount() == 0) {
        newId = "11110";
    } else {
        // Find the largest ID in the existing data and increment it by 1
        int maxId = 0;
        for (int i = 0; i < model.getRowCount(); i++) {
            String id = model.getValueAt(row:i, column:0).toString();
            int currentId = Integer.parseInt(s: id);
            if (currentId > maxId) {
                maxId = currentId;
            }
        }
        newId = String.valueOf(maxId + 1);
    }

    BookStore[] bookStoreArray = new BookStore[1];
    bookStoreArray[0] = new CustomerManagement();

    bookStoreArray[0].setId(id: newId);
    ((CustomerManagement) bookStoreArray[0]).setcustPassword(custPassword: customerPasswordLabel.getText().trim());
    bookStoreArray[0].setName(name: customerNameLabel.getText().trim());
    ((CustomerManagement) bookStoreArray[0]).setcustGender(custGender:customerGender.getSelectedItem().toString().trim());
```

```java
    // Adding customer details to the database
    addCustomerToDataBase(
        id: bookStoreArray[0].getId(),
        password: ((CustomerManagement) bookStoreArray[0]).getcustPassword(),
        name: bookStoreArray[0].getName(),
        gender:((CustomerManagement) bookStoreArray[0]).getcustGender()
    );

    // Adding data to Jtable
    model.addRow(new Object[]{
        bookStoreArray[0].getId(),
        ((CustomerManagement) bookStoreArray[0]).getcustPassword(),
        bookStoreArray[0].getName(),
        ((CustomerManagement) bookStoreArray[0]).getcustGender()
    });

    JOptionPane.showMessageDialog(parentComponent: null, message:"Data successfully added!");

    customerPasswordLabel.setText(t: null);
    customerNameLabel.setText(t: null);
    customerGender.setSelectedItem(anObject: null);
}
```

- The addCustomerActionPerformed() method is triggered when the "Add" button on the customer management interface is clicked. It handles the process of adding a new customer to the inventory. The method first checks if the required information, such as customer password, name, and gender, is provided. If any field is missing, an error message is displayed, and the method returns. Next, it checks if the maximum customer account count has been reached. If the limit is reached, a message is shown, and the method returns. Otherwise, the method generates a new customer ID based on existing data. It creates a new CustomerManagement object, sets its attributes, and adds the customer details to the database. The method then adds the customer's data to the customerInventoryTable JTable and displays a success message. Finally, it resets the input fields for the next customer entry.

```java
/**
 * updateCustomerActionPerformed
 * This method is called when the "Update" button on customer management is clicked.
 * @param evt
 */
```

```java
private void updateCustomerActionPerformed(java.awt.event.ActionEvent evt) {
    // Setting the updated book details to the CustomerManagement instance
    BookStore[] bookStoreArray = new BookStore[1];
    bookStoreArray[0] = new CustomerManagement();

    bookStoreArray[0].setId(id: customerIdLabel.getText());
    ((CustomerManagement) bookStoreArray[0]).setcustPassword(custPassword: customerPasswordLabel.getText());
    bookStoreArray[0].setName(name: customerNameLabel.getText());
    ((CustomerManagement) bookStoreArray[0]).setcustGender(custGender:customerGender.getSelectedItem().toString());

    // Update customer details to the database
    removeCustomerFromDataBase(id: bookStoreArray[0].getId());
    addCustomerToDataBase(
        id: bookStoreArray[0].getId(),
        password: ((CustomerManagement) bookStoreArray[0]).getcustPassword(),
        name: bookStoreArray[0].getName(),
        gender:((CustomerManagement) bookStoreArray[0]).getcustGender()
    );

    DefaultTableModel model = (DefaultTableModel) customerInventoryTable.getModel();

    // If a single row is selected
    if (customerInventoryTable.getSelectedRowCount() == 1) {
        if (customerIdLabel.getText().isEmpty() || customerNameLabel.getText().isEmpty()|| customerPasswordLabel.getText().isEmpty() || customerGender.get
            JOptionPane.showMessageDialog(parentComponent: null, message:"Missing required information. Please complete all fields.");
            return;
        }

        model.setValueAt(aValue:bookStoreArray[0].getId(), row:customerInventoryTable.getSelectedRow(), column:0);
        model.setValueAt(aValue:((CustomerManagement) bookStoreArray[0]).getcustPassword().toString(), row:customerInventoryTable.getSelectedRow(), column:1);
        model.setValueAt(aValue:bookStoreArray[0].getName(), row:customerInventoryTable.getSelectedRow(), column:2);
        model.setValueAt(aValue:((CustomerManagement) bookStoreArray[0]).getcustGender(), row:customerInventoryTable.getSelectedRow(), column:3);

        JOptionPane.showMessageDialog(parentComponent: this, message:"Record Updated Successfully");
    } else {
        if (customerInventoryTable.getRowCount() == 0) {
            JOptionPane.showMessageDialog(parentComponent: null, message:"No data available in the table.");
        } else if (customerInventoryTable.getSelectedRowCount() == 0) {
            JOptionPane.showMessageDialog(parentComponent: null, message:"Please choose a single row to perform the update.");
        } else {
        // If multiple rows are selected
            JOptionPane.showMessageDialog(parentComponent: null, message:"Please choose a single row to perform the update.");
        }
    }

    customerIdLabel.setText(t: null);
    customerPasswordLabel.setText(t: null);
    customerNameLabel.setText(t: null);
    customerGender.setSelectedItem(anObject: null);
}
```

- The updateCustomerActionPerformed() method is called when the "Update" button is clicked in customer management. It retrieves the updated customer details from the input fields, updates the customer details in the database, and updates the corresponding row in the customer inventory table. It performs validation checks for missing information and ensures that only a single row is selected for update. If the update is successful, a success message is displayed, and the input fields are cleared.

```java
/**
* deleteCustomerActionPerformed
* This method is called when the "Delete" button on customer management is clicked.
* @param evt
*/
private void deleteCustomerActionPerformed(java.awt.event.ActionEvent evt) {
    // Setting the updated customer details to the CustomerManagement instance
    BookStore[] bookStoreArray = new BookStore[1];
    bookStoreArray[0] = new CustomerManagement();

    bookStoreArray[0].setId(id: customerIdLabel.getText());
    ((CustomerManagement) bookStoreArray[0]).setcustPassword(custPassword: customerPasswordLabel.getText());
    bookStoreArray[0].setName(name: customerNameLabel.getText());
    ((CustomerManagement) bookStoreArray[0]).setcustGender(custGender:customerGender.getSelectedItem().toString());

    // Delete customer data from the database
    removeCustomerFromDataBase(id: bookStoreArray[0].getId());

    DefaultTableModel model = (DefaultTableModel) customerInventoryTable.getModel();

    if (customerInventoryTable.getSelectedRowCount() == 1) {
        model.removeRow(row:customerInventoryTable.getSelectedRow());
        JOptionPane.showMessageDialog(parentComponent: null, message:"Customer successfully deleted.");
    } else if (customerInventoryTable.getRowCount() == 0) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"No data available in the table.");
    } else if (customerInventoryTable.getSelectedRowCount() == 0) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Please choose a single row to perform the update.");
    // If multiple rows are selected
    } else {
        JOptionPane.showMessageDialog(parentComponent: null, message:"lease choose a single row to perform the update.");
    }

    customerIdLabel.setText(t: null);
    customerPasswordLabel.setText(t: null);
    customerNameLabel.setText(t: null);
    customerGender.setSelectedItem(anObject: null);
}
```

- The deleteCustomerActionPerformed() method is called when the "Delete" button is clicked in customer management. It retrieves the customer details from the input fields, deletes the customer data from the database, and removes the corresponding row from the customer inventory table. It performs validation checks to ensure that only a single row is selected for deletion. After the deletion is performed, the input fields are cleared.

```java
/**
* clearCustomerActionPerformed
* This method is called when the "Clear" button on customer management is clicked.
* It resets the values of various UI components.
* @param evt
*/
private void clearCustomerActionPerformed(java.awt.event.ActionEvent evt) {
    customerIdLabel.setText(t: null);
    customerNameLabel.setText(t: null);
    customerPasswordLabel.setText(t: null);
    customerGender.setSelectedItem(anObject: null);
}
```

- The clearCustomerActionPerformed() method is called when the "Clear" button is clicked in customer management. It resets the values of various UI components related to customer information, including the customer ID, name, password, and gender.

```java
/**
* logoutActionPerformed
* This method is called when the user clicks the logout icon button.
* @param evt
*/
private void logoutActionPerformed(java.awt.event.ActionEvent evt) {
    Login login = new Login();
    login.setVisible(b: true);
    dispose();
}
```

- The logoutActionPerformed() method is called when the user clicks the logout button. It creates a new instance of the Login class, which represents the login page, makes it visible, and disposes of the current Admin page.

```java
/**
 * customerInventoryTableMouseClicked
 * This method is called when the user clicks on a row in the customerInventoryTable component.
 * Retrieves relevant data from the selected row index.
 * @param evt The mouse click event.
 */
private void customerInventoryTableMouseClicked(java.awt.event.MouseEvent evt) {
    int index = customerInventoryTable.getSelectedRow();
    DefaultTableModel model = (DefaultTableModel) customerInventoryTable.getModel();

    customerIdLabel.setText(t: model.getValueAt(row:index, column:0).toString());
    customerPasswordLabel.setText(t: model.getValueAt(row:index, column:1).toString());
    customerNameLabel.setText(t: model.getValueAt(row:index, column:2).toString());
    customerGender.setSelectedItem(anObject: model.getValueAt(row:index, column:3).toString());
}
```

- The customerInventoryTableMouseClicked() method is called when the user clicks on a row in the customerInventoryTable component. It retrieves the relevant data from the selected row index and populates the corresponding UI components with the retrieved data. Specifically, it retrieves the customer ID, password, name, and gender from the selected row and sets them as the values for the customerIdLabel, customerPasswordLabel, customerNameLabel, and customerGender UI components, respectively.

```java
/**
 * addCustomerToDataBase
 * This method is used to add a new line of data to the "customer.txt" file to update data.
 * @param id
 * @param password
 * @param name
 * @param gender
 */
private void addCustomerToDataBase(String id, String password, String name, String gender) {
    try {
        // Append data to existing file
        FileWriter fw = new FileWriter(fileName: "customer.txt", append:true);
        fw.write(id + "/" + password + "/" + name + "/" + gender);
        fw.write(str:System.getProperty(key:"line.separator"));
        fw.close();

    } catch (IOException ex) {
        JOptionPane.showMessageDialog(parentComponent: null, message:"Error in adding data to txt file.");
    }
}
```

- The addCustomerToDataBase(String id, String password, String name, String gender) method is used to add a new line of data to the "customer.txt" file to update the customer details. It appends the customer details to the existing file.

```java
    /**
     * removeCustomerFromDataBase
     * This method is used to remove specific line from the "customer.txt" based on the customer ID
     * user chose.
     * @param id
     */
    private void removeCustomerFromDataBase(String id) {
        String fileName = "customer.txt";
        String tempFile = "temp.txt";

        File oldFile = new File(pathname: fileName);
        File newFile = new File(pathname: tempFile);

        try {

            FileReader fr = new FileReader(file: oldFile);
            BufferedReader br = new BufferedReader(in: fr);

            FileWriter fw = new FileWriter(file: newFile);
            BufferedWriter bw = new BufferedWriter(out: fw);

            String line;

            while ((line = br.readLine()) != null) {
                int slashIndex = line.indexOf(str: "/");
                if (slashIndex != -1) {
                    String specifiedId = line.substring(beginIndex: 0, endIndex: slashIndex);
                    // Skipping the data line that corresponds to the ID user chose to order
                    if (specifiedId.equals(anObject: id)) {
                        continue;
                    }
                }
                // Write non-matching lines to "temp.txt"
                bw.write(str: line);
                bw.newLine();
            }
            bw.close();
            fr.close();

            oldFile.delete();
            newFile.renameTo(dest: oldFile);
//              System.out.println(new File(".").getAbsolutePath());
        } catch (IOException e) {
            JOptionPane.showMessageDialog(parentComponent: null, message: "Error in deleting data from txt file.");
        }
    }
```

- The removeCustomerFromDataBase(String id) method is used to remove a specific line from the "customer.txt" file based on the customer ID. It reads the contents of the original file, skips the line with the specified ID, and writes the remaining lines to a temporary file. Finally, it replaces the original file with the temporary file.

```java
    // Variables declaration - do not modify
    private javax.swing.JButton addBook;
    private javax.swing.JButton addCustomer;
    private javax.swing.JTabbedPane adminScreen;
    private javax.swing.JLabel authorSubtitle;
    private javax.swing.JTextField bookAuthorLabel;
    private javax.swing.JLabel bookIDSubtitle;
    private javax.swing.JTextField bookIdLabel;
    private javax.swing.JTextField bookImage;
    private javax.swing.JLabel bookImageLabel;
    private javax.swing.JLabel bookInventorySubtitle;
    private javax.swing.JTable bookInventoryTable;
    private javax.swing.JScrollPane bookInventoryTableScroll;
    private javax.swing.JPanel bookManagementTab;
    private javax.swing.JTextField bookPriceLabel;
    private javax.swing.JTextField bookQtyLabel;
    private javax.swing.JTextField bookSynopsis;
    private javax.swing.JLabel bookSynopsisLabel;
    private javax.swing.JScrollPane bookSynopsisScroll;
    private javax.swing.JTextField bookTitleLabel;
    private javax.swing.JPanel borderPanel;
    private javax.swing.JButton clearBook;
    private javax.swing.JButton clearCustomer;
    private javax.swing.JComboBox<String> customerGender;
    private javax.swing.JLabel customerIDSubtitle;
    private javax.swing.JTextField customerIdLabel;
    private javax.swing.JLabel customerInventorySubtitle;
    private javax.swing.JTable customerInventoryTable;
    private javax.swing.JScrollPane customerInventoryTableScroll;
    private javax.swing.JPanel customerManagementTab;
    private javax.swing.JTextField customerNameLabel;
    private javax.swing.JTextField customerPasswordLabel;
    private javax.swing.JButton deleteBook;
    private javax.swing.JButton deleteCustomer;
    private javax.swing.JLabel genderSubtitle;
    private javax.swing.JLabel logo;
    private javax.swing.JButton logout;
    private javax.swing.JLabel nameSubtitle;
    private javax.swing.JLabel pageTitle1;
    private javax.swing.JLabel pageTitle2;
    private javax.swing.JLabel passwordSubtitle;
    private javax.swing.JLabel priceSubtitle;
    private javax.swing.JLabel qtySubtitle;
    private javax.swing.JLabel titleSubtitle;
    private javax.swing.JButton updateBook;
    private javax.swing.JButton updateCustomer;
    private javax.swing.JButton uploadImage;
    private javax.swing.JButton uploadSynopsis;
    // End of variables declaration
}
```

- Variables declaration section in the Admin class.

Lesson Learned

As this was my first ever programming project using Java, I was quite pressured by it. I had a hard time finding ideas to fit this project, as my skills and knowledge in Java were overall still very basic and at beginner level. However, after some time contemplating and consulting ideas with my friends and Mr. Jude, I finally decided to step out of my boundaries and create a program inspired by my reading hobby, which is a bookstore management system.

Through the making of this final project, I was able to get first-hand experience on how to program the front-end design with Java via Netbeans IDE. It was my first time using the IDE and building a GUI in Java with it. Through it, I learned a lot of knowledge outside of the course and received insights into how to develop Java using Object-Oriented Programming principles. Overall, this was a really fun yet stressful journey. I can say that I am proud of myself for being able to create this project using outside knowledge of the course.

References

- Troubleshooting:
  https://stackoverflow.com
- Design:
  https://canva.com
  https://lucid.app
- Book Images and Synopsis:
  https://goodreads.com

Appendices

- Github Repo:
  https://github.com/audreyfabiola/Literarium
- Video:
  https://www.youtube.com/watch?v=fxTRZ1hzUxs
- Design Canva File:
  https://www.canva.com/design/DAFkBCInBjA/hRBOC3Va7ePK-u42MDxSug/edit?utm_content=DAFkBCInBjA&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton