## LMS Software Development Plan

## Otoniel Rodriguez-Perez

## CEN-3024C

## 24204

## 01/26/2025

## Table of Contents:

## Introduction 3

## Application Requirements 4

## Features and Functionalities

## Application Constraints

## Requirements Gathering 6

## User Needs

## User Interactions

## User Stories

## Implementation Plan 8

## Coding Strategies

## Algorithms

## UML Diagrams

## Testing Plan 18

## Testing Strategies

## Test Cases

## Deployment 20

## Instructions

## Source Code

1. **Introduction**

This work forms the software development plan in order to illustrate the development of a new LMS (Learning Management Application) software. Through the development of this software, any library will have an easy way of managing patron information. This program will be able to create, delete, or show all patrons. It is directly inputted and saved inside the application. Users will be able to communicate with the applications by an ONS (On-screen) menu from the CLI (Command Line Interface). Users (librarians) will then be able to choose any of the following options: upload patron information with a text (.txt) file, enter individual patron data, delete a patron from the application, display all patrons, or quit the application.

1. **Application Requirements**

**a. The following application features and functionalities will be implemented:**

* **Patron Information Management:**
  + Each patron will contain a unique 7-digit ID number. The ID will be used to access and see personal information including:
    - Name
    - Address
    - Amount owed to the Library (anywhere from $0 – 250 dollars). A dollar sign will not be included.
* **Librarian Capabilities:**
  + **Add a new patron:** Users can add patrons by entering patron information manually.
  + **Add multiple patrons:** Users can upload multiple patrons by uploading a list within a text file (.txt).
  + **Remove a patron:** Users can remove patrons from by using their unique ID number.
  + **Display all patrons:** Users can also have the options to view all patrons stored.
  + **Close application:** After the users perform any of the functions mentioned above, they will have the option to exit the application.
* **How data is stored and formatted:**
  + All information entered or uploaded will be saved within the application.
  + Patron information will be stored and displayed using the following formatting:
    - ID-name-address-amountOwed
    - Examples:
      * 1245789-Sarah Jones-1136 Gorden Ave. Orlando, FL 32822-40.54
      * 3256897-Mason Arby-6060 Saginaw St. Casselberry, FL 34852-0.0

1. **Application Constraints:**

* Information must be in the correct structure as described below:
  + ID number must be seven digits (integers).
  + Each ID must be unique and not match any other ID.
  + There must be a name and address inputted in a single line.
  + Dollar amount must be entered as a double variable (0.0, 1.56,231.89)
  + Dollar amount cannot be outside the range specified in the features/functionally section.
* When a file is uploaded, it must contain the formatting specified in the features/functionally section.
* The user is required to confirm the decision in the application when:
  + Adding a patron manually.
  + Uploading a list of patrons by file.
  + Removing a patron from the list
* The application doesn’t have any GUI (Graphical User Interface). The user must use the CLI (Command Line Interface).
* Application must run when converted into a .JAR file.
* The user must choose between executing the .JAR file directly in the CLI or using a.bat file.
* The JDK (Java Development Kit) version 17 (or higher) must be installed.
* The application must run on Windows or MacOS (Instructions will only be made for Windows).

1. **Requirements Gathering**
2. **User needs :**
   * Users (Librarians) need a simple and easy way to view and manage patron information.
     + The application will have the options to:
       - Enter patron information manually.
       - Upload a list of patrons by importing a text file.
       - View a list of all patrons.
       - Delete a patron using his or her ID number.
3. **User interactions:**
   * The user will be able to:
     + Interact with the application via the application’s CLI.
     + Enter data manually or upload information through a file with the specified formatting.
     + View all patrons.
     + Be asked to confirm any decisions made.
     + Close the application.
4. **User Stories:**

**The following contains actions made by a librarian accessing the application:**

1. **Add a patron:**
   * **User Story:** As a librarian, I want to add a patron’s ID, name, address, and owed amount.
   * **Action:** The librarian will input the information specified. He or she will enter a unique ID number for the patron and then fill out personal details.
   * **Output:** Shows a patron added to the storage within the application.
2. **Remove patron:**
   * **User Story:** As a librarian, I should be able to remove a patron from the application storage.
   * **Action:** The librarian enters the member's unique ID to remove them.
   * **Output:** The patron is removed from the application, and their records are deleted.
3. **Display patron information:**
   * **User Story:** As a librarian, I should be able to display all patrons stored.
   * **Action:** The librarian selects the option to view all patrons.
   * **Output:** A list is displayed of all patrons.
4. **Implementation Plan**
5. **The application will use the following** **coding strategies**:
6. Code will be written in Java (Object Oriented Programing Language)
7. It will import:
   * java.util.\*
   * import java.io.BufferedReader;
   * import java.io.FileReader;
   * import java.io.IOException;
   * import java.util.\*;
   * import java.util.stream.Collectors
   * import java.util.InputMismatchException;
   * import java.util.Scanner;
8. There will be 5 classes:
   * Librarian (Main Method)
   * Patron
   * UserHandling
   * PatronManaging
   * FileHandling
9. There will be methods, constructors, arguments, and returns within each class.
10. There will be algorithms to provide functionaries for each method called from the main menu.
11. OOP strategies and functionalities will be used for organization and unit testing.
12. There will be user input via the CLI onscreen menu.
13. There will be exceptions to ensure the stored information is correct.
14. Loops and conditional statements will be made for user input, choices, and confirmation.

**Classes:**

1. **Librarian (Main method):**
   * **Variables:**
     + PatronManaging patronManaging = new PatronManaging();(Creates instance of PatronManaging class)
     + UserHandling userHandling = new UserHandling();(Creates instance of UserHandling class)
   * **Methods:**
     + viewMenu()
       - **Arguments:** none
       - **Returns:** The user’s choice will determine which method to call.
2. **Patron:**
   * **Variables:**
     + private int id; (7-digit unique member)
     + private String name; (Full name)
     + private String address (Full address)
     + private double amoundOwed (Amount in “double” variable format);
   * **Constructor:**
     + public Patron()
       - **Arguments**:
         1. int id
         2. String name
         3. String address
         4. double amountOwed
       - **Returns:**
         1. this.id = id
         2. this.name = name
         3. this.address = address
         4. this.amountOwed = amountOwed
   * **Getters:**
     + public int getId()
       - **Arguments**: none
       - **Returns**: return id
     + public String getName()
       - **Arguments**: none
       - **Returns**: return name
     + public String getAddress()
       - **Arguments**: none
       - **Returns**: return address
     + public double getAmountOwed()
       - **Arguments**: none
       - **Returns**: return amountOwed
   * **Setters**:
     + public void setID()
       - **Arguments**: int id
       - **Returns**: this.id = id
     + public void setName()
       - **Arguments**: String name
       - **Returns**: this.name = name
     + public void setAddress()
       - **Arguments**: String address
       - **Returns**: this.address = adress
     + public void setAmountOwed()
       - **Arguments**: double amoutOwed
       - **Returns**: this.amoutOwed = amoutOwed
   * **Methods:**
     + Public String toString()
       - Arguments: none
       - Returns: Application.out.println(id + name + address + amountOwed)
3. **UserHandling:**
   * **Variables:**
     + none
   * **Methods:**
     + usersChoice()
       - **Arguments:** Scanner scanner (Method used for user input)
       - **Returns:** Input is returned.
     + manualIdInput()
       - **Arguments:** Scanner scanner (Method used for user input)
       - **Returns:** Integer ID user input is returned.
     + StringInput()
       - **Arguments:** Scanner scanner, String prompt (Methods used for user input)
       - **Returns:** String user input is returned.
     + amountInput()
       - **Arguments:** Scanner scanner (Method used for user input)
       - **Returns:** Double amount user input is returned.
     + patronIdDelete()
       - **Arguments:** Scanner scanner (Method used for user input)
       - **Returns:** User ID input is returned for deletion.
     + findPatronById()
       - **Arguments:** int id (integer search)
       - **Returns:** Complete patron information is returned.
     + userConfirmation()
       - **Arguments:** Scanner scanner, String message (Methods used for user input)
       - **Returns:** Boolean true or false is returned.
4. **PatronManaging:**
   * **Variables:**
     + List<Patron> patrons = new ArrayList<>() (Stores all patrons)
   * **Methods:**
     + addPatronManual()
       - **Arguments:** Scanner scanner (Method used for user input)
       - **Returns:** A new patron is added to the patrons ArrayList.
     + addPatronFile()
       - **Arguments:** Scanner scanner (Method used for user input)
       - **Returns:** Multiple patrons are to the patrons ArrayList.
     + removePatron()
       - **Arguments:** Scanner scanner (Method used for user input)
       - **Returns:** It deletes specified patron from the patrons ArrayList.
5. **FileHandling:**
   * **Variables:**
     + none
   * **Methods:**
     + addPatronsbyFile()
       - **Arguments:** String filePath, List<Patron> patrons (String for file location and patrons ArrayList)
       - **Returns:** New patron(s) added to the patrons ArrayList.
6. **Algorithms:**

* **Add Patron Manually Algorithm:**
  1. The user enters the option to enter patron information from the onscreen menu.
  2. If the ID entered is already in the application or any manual entry has incorrect formatting, an exception is created to iterate an error message until corrections are made.
  3. Once correct information is entered, the user is asked to confirm.
  4. If confirmed, a Patron object (instance) is created.
  5. Patron instance is added to the next available position in the patrons ArrayList.
  6. Patron instance information is stored within this list in the application.
* **Add Patron by File Algorithm:**
  1. The user enters the option to upload a file from the onscreen menu.
  2. If the file has any ID that is already in the application, has incorrect formatting, or the file does not exist, an exception is created to display an error message.
  3. Any patron within the file that has a duplicate ID within the file or the LMS is skipped.
  4. Any row containing incorrect values or formatting is skipped.
  5. The user will be asked to confirm valid patron(s) upload.
  6. Once correct information is uploaded, one or more Patron objects (instances) are created.
  7. Patron instances are added to the next available positions in the patrons ArrayList.
  8. Patron instances are stored within this ArrayList in the application.
* **Remove Patron Algorithm:**
  1. User selects option to delete a patron from the onscreen menu.
  2. User enters the patron’s ID number.
  3. If the ID is not in the application or has incorrect formatting, an exception is created to iterate an error and notify user to try again or return to the main menu.
  4. The application searches for the corresponding ID number of the patron in the patrons ArrayList.
  5. If the patron is in the patrons ArraysList, the user will be asked to confirm the decision.
  6. If confirmed, the patron record is deleted from the patrons ArrayList.
* **View Patron Algorithm**
  1. The user selects the option to view all patrons from the onscreen menu.
  2. The application searches for the information stored in the patrons ArrayList.
  3. If the ArrayList is empty, a message will be displayed to the user.
  4. If there a patrons, they will be sorted by ID.
  5. A loop is then activated to iterate through the patrons ArrayList.
  6. All patrons are displayed.

1. **UML Diagrams:**

|  |
| --- |
| **Librarian** |
| - patronManaging = new PatronManaging(): final  - userHandling = new UserHandling(): final |
| + Librarian()  + main(String[] args): void  + viewMenu(): void |

|  |
| --- |
| **Patron** |
| - id : int  - name : String  - address : String  - amountOwed : double |
| + Patron(int id, String name, String address, double amountOwed)  + getId(): int  + getName(): String  + getAddress(): String  + getAmountOwed(): double  + setId(id: int): void  + setName(name: String): void  + setAddress(address: String): void  + setAmountOwed(amount: double): void  + toString(): String |

|  |
| --- |
| **UserHandling** |
| none |
| + usersChoice(Scanner scanner): int  + manualIdInput(Scanner scanner): int  + StringInput(Scanner scanner, String prompt): String  + amountInput(Scanner scanner): double  + userConfirmation(Scanner scanner, String message): boolean  + patronIdDelete(Scanner scanner): int  + findPatronById(int id): Patron |

|  |
| --- |
| **PatronManaging** |
| Patron patrons = new ArrayList<>(): final |
| + addPatronManual(Scanner scanner): void  + addPatronFile(Scanner scanner): void  + removePatron(Scanner scanner): void  + viewAllPatrons(): void |

|  |
| --- |
| **FileHandling** |
| None |
| + addPatronsByFile(String filePath, List<Patron> patrons): void |

**5. Testing Plan**

**a. Testing Strategies:**

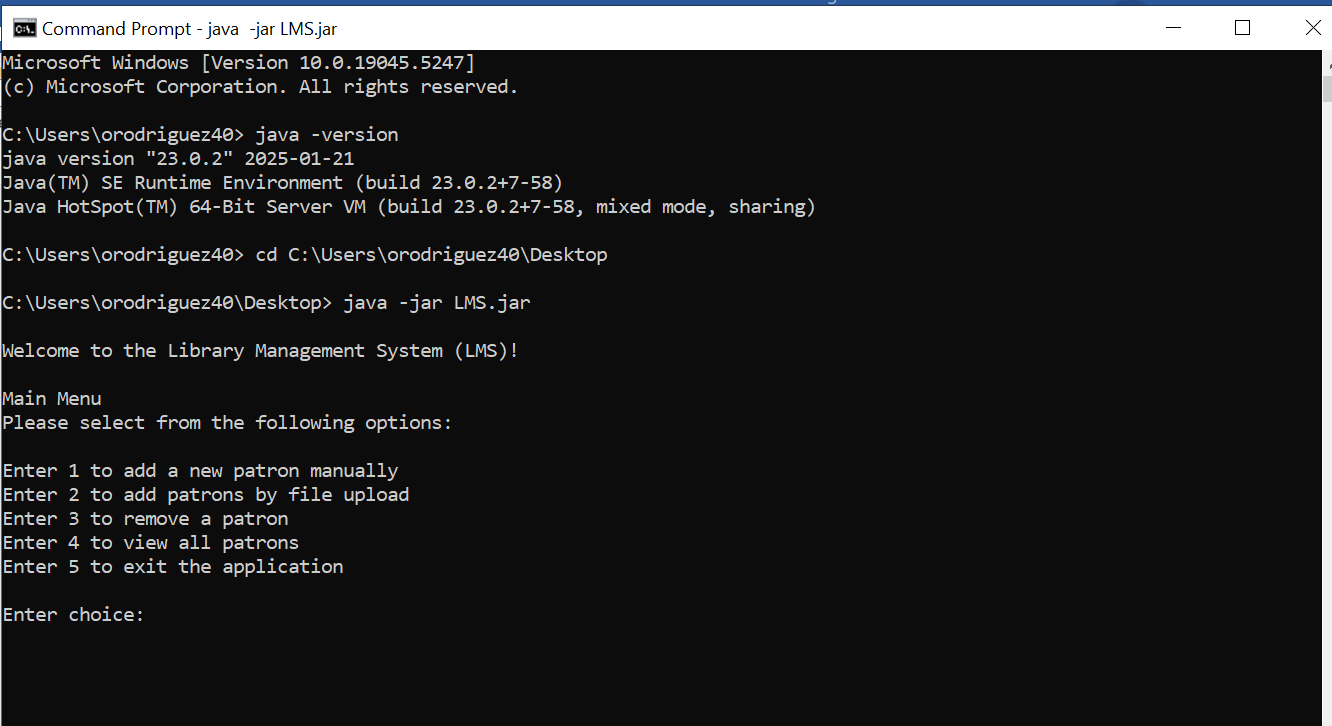
1. **Class Testing:** Each method within each class will be tested individually.
2. **Application Testing:** The application will be tested in its entirety.
3. **User Testing:** Users (librarians) will test by entering patron data manually and by using a text file containing patron information.

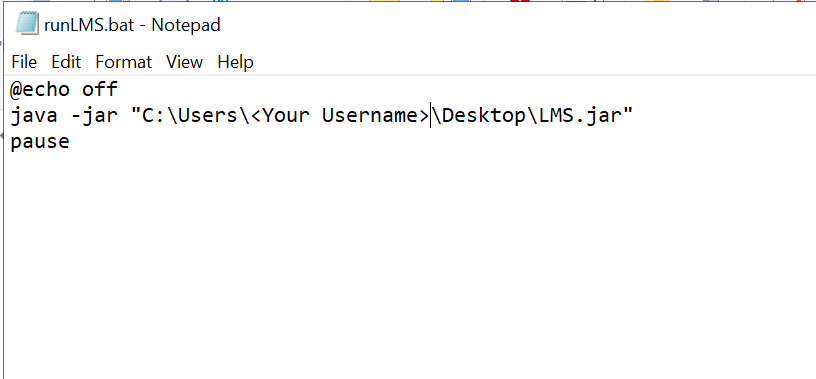
**b. Test Cases:**

1. **First Test Case:**
   * **Add Patron:**
     + **Purpose:** Patrons to be added successfully by manual entry or file upload.
     + **Steps:**
       1. Create a patron ID.
       2. Enter patron information or upload a text file.
       3. Follow correct formatting.
       4. Confirm action.
     + **Output:** One or multiple patrons are added to the application.
     + **Output:** The updated amount owed is saved and stored.
2. **Second Test Case:**
   * **Remove Patron:**
     + **Purpose:** Utilizing the patron’s ID, their record will be removed from the application.
     + **Steps:**
       1. Enters patron’s ID.
       2. Follow the correct format.
       3. Confirm deletion.
     + **Output:** The patron’s information is removed from the list.
3. **Third Test Case:**
   * **View All Patrons:**
     + **Purpose:** User will be able to see all patrons currently stored in the application.
     + **Steps:**
       1. User selects choice to view all patrons.
     + **Output:** All patrons are displayed.

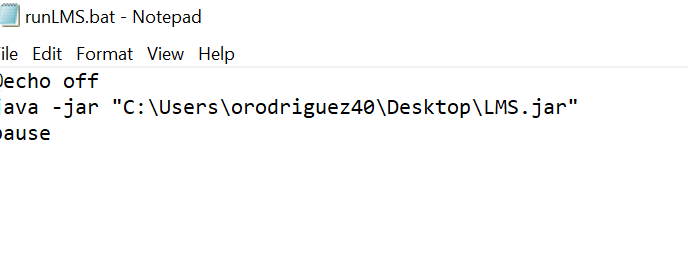
**6. Deployment**

**a. Instructions:**

* Download Steps:
  + 1. Download the *LMS.jar* and *runLMS.bat*, files.
    2. Download *PatronListCorrect.txt* and *PatronListErrors.txt* test files (optional).
    3. Move all files to your desktop.
* File Execution Steps:
  + Option 1 (Directly by using the CLI)
    1. On Windows, type “Command Prompt” on the start search bar. Click it to open the Command Line.
    2. Confirm the Java version by typing “java –version” and hit enter.
    3. Move to your desktop by typing “cd C:\Users\<Your Username>\Desktop” and hit enter.
    4. Start the application by typing “java -jar LMS.jar” and hit enter (see example below):
  + Option 2 (Using the .bat file)
    1. Verify the *LMS.jar* and *runLMS.bat* files are on your Desktop.
    2. Right-click on *runLMS.bat and* click Edit.
    3. You should see the following information in your editor:



* + 1. Change <Your Username> to your username (see example below):



* + 1. Click Save
    2. Exit the editor and double-click on the *LMS.jar* file.
* Grading Steps:
  + Test 1: Read Data
    1. Enter option 2 to start file upload.
    2. Enter the complete file path (example “C:\Users\orodriguez40\Desktop\PatronListCorrect.txt”)
    3. All valid patrons will be displayed on the list. Verify adding them to the LMS by typing ‘y’ or ‘yes’.
    4. If confirmed, these patrons are added.
    5. If there are any duplicate IDs within the file or in the LMS, these are displayed and skipped.
  + Test 2: Display Data
    1. After the file processing is complete, you will be brought to the main menu.
    2. Verify patrons added manually or file upload by entering option 4 on the main menu. All patrons will be displayed on the screen.
  + Test 3: Display a Menu
    1. Once the LMS is executed, the main menu will be displayed.
    2. You may choose to add, remove, or display all patrons.
    3. The menu will iterate until option 5 from the main menu is entered, closing the LMS.
  + Test 4: : Create Data
    1. Enter 1 on the main menu to start entering patron information manually.
    2. Must follow guidelines of ID, name, address, and amount owed.
    3. Incorrect information will cause an iteration until the user inputs correct information.
    4. Once all information is entered, the patron information will be displayed on the screen.
    5. Confirm that you want to add the patron.
    6. If confirmed, the patron will be added to the LMS.
  + Test 5: Remove Data
    1. Enter 3 on the main menu to start patron removal.
    2. Enter a valid ID that’s already in the LMS.
    3. If found, the patron’s information will be displayed on the screen.
    4. Confirmation must be made to delete the patron.
    5. If confirmed, the patron will be removed from LMS.
    6. Verify by entering 4 on the main menu.
    7. Do Ctrl + f on the keyboard to open the find dialog box.
    8. Enter the same ID. A message will be displayed that the patron is not found.
  + Test 6: Comments
    - * Option 1 (GitHub)
        1. Visit <https://github.com/orodriguez40/LMS_Application>
        2. Open the src folder.
        3. Click on any .java file to see the comments.
      * Option 2 (zip file)
        1. Download the .zip file attached.
        2. Extract files, preferably to your desktop.
        3. Go to -> LMS Application -> src
        4. Open any .java file on a text editor or IDE to see the comments.

**b. Source Code:**

* **Librarian:**

// Otoniel Rodriguez-Perez  
// CEN-3024C-24204  
// 01/26/2025  
  
// Librarian Class (Main Application):  
// This is where the application will run.  
// The user will open the JAR file through the CLI.  
// They will have options to add a patron manually or by file,  
// remove a patron based on their ID, view all patrons, or close the application.  
  
// Imported Library  
import java.util.Scanner;  
  
public class Librarian {  
  
 // Attributes are instances of the PatronManaging and UserHandling classes  
 private static final PatronManaging *patronManaging* = new PatronManaging();  
 private static final UserHandling *userHandling* = new UserHandling();  
  
 // Main Method  
 public static void main(String[] args) {  
 // Scanner is used to accept all user input.  
 Scanner scanner = new Scanner(System.*in*);  
 int userChoice;  
  
 // Welcome message for the user.  
 System.*out*.println("\nWelcome to the Library Management System (LMS)!");  
  
 // Main menu will iterate until the user chooses to close the application.  
 do {  
 // Display the main menu options  
 *viewMenu*();  
 // Calls method to verify user input.  
 System.*out*.println("\nEnter choice:");  
 userChoice = *userHandling*.usersChoice(scanner);  
  
 // Switch statement to handle user choices.  
 switch (userChoice) {  
 case 1:  
 *patronManaging*.addPatronManual(scanner); // Adds a patron manually.  
 break;  
 case 2:  
 *patronManaging*.addPatronFile(scanner); // Add patrons by file upload.  
 break;  
 case 3:  
 *patronManaging*.removePatron(scanner); // Removes a patron.  
 break;  
 case 4:  
 *patronManaging*.viewAllPatrons(); // View all patrons.  
 break;  
 case 5:  
 System.*out*.println("Thank you for using the LMS application. Goodbye!"); // Message when user chooses to close application.  
 break;  
 default:  
 System.*out*.println("Invalid option. Please enter a number from 1 to 5."); // Checks for invalid user input.  
 break;  
 }  
 } while (userChoice != 5); // Continue looping until the user chooses to exit.  
  
 scanner.close(); // Closes the scanner instance.  
 }  
  
 // Method is called display the main menu options.  
 public static void viewMenu() {  
 System.*out*.println("\nMain Menu");  
 System.*out*.println("Please select from the following options:\n");  
 System.*out*.println("Enter 1 to add a new patron manually");  
 System.*out*.println("Enter 2 to add patrons by file upload");  
 System.*out*.println("Enter 3 to remove a patron");  
 System.*out*.println("Enter 4 to view all patrons");  
 System.*out*.println("Enter 5 to exit the application");  
 }  
  
}

* **Patron**

// Otoniel Rodriguez-Perez  
// CEN-3024C-24204  
// 01/26/2025  
  
//Patron Class:  
// Used to define what a kind of information a patron will contain.  
//It will be called by the PatronManaging class to store, view, or remove patrons.  
  
public class Patron {  
  
 //Attributes  
 private int id;  
 private String name;  
 private String address;  
 private double amountOwed;  
  
 //Constructor  
 public Patron(int id, String name, String address, double amountOwed) {  
 this.id = id;  
 this.name = name;  
 this.address = address;  
 this.amountOwed = amountOwed;  
 }  
  
 //Getters  
 public int getId() {  
 return id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public String getAddress() {  
 return address;  
 }  
  
 public double getAmountOwed() {  
 return amountOwed;  
 }  
  
 //Setters  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public void setAddress(String address) {  
 this.address = address;  
 }  
  
  
 public void setAmountOwed(double amountOwed) {  
 this.amountOwed = amountOwed;  
 }  
  
 //toString method is overwritten to display patron information.  
 @Override  
 public String toString() {  
 return id + "-" + name + "-" + address + "-" + amountOwed + "\n";  
 }  
}

* **UserHandling**

// Otoniel Rodriguez-Perez  
// CEN-3024C-24204  
// 01/26/2025  
  
// UserHandling Class:  
// This class handles all user inputs and confirmations.  
  
  
//Imported Libraries  
import java.util.InputMismatchException;  
import java.util.Scanner;  
  
public class UserHandling {  
  
 // Method checks for user's choice in the main menu.  
 public int usersChoice(Scanner scanner) {  
 while (true) {  
 try {  
 // Get user input and validate it.  
 int choice = scanner.nextInt();  
 scanner.nextLine(); // Clears the buffer.  
 if (choice >= 1 && choice <= 5) {  
 return choice; // Return valid input.  
 }  
 System.*out*.println("Please try again. Select a number between 1 and 5."); // Error message for invalid numeric input.  
 } catch (InputMismatchException e) {  
 System.*out*.println("Please try again. Input must be a number."); // Error message for non-numeric input.  
 scanner.nextLine(); // Clears invalid input.  
 }  
 }  
 }  
  
 // Method to get a patron ID from user input.  
 public static int manualIdInput(Scanner scanner) {  
 int id; // Local variable to hold the patron ID.  
  
 // Loops until a valid 7-digit ID is entered.  
 while (true) {  
 System.*out*.print("ID: ");  
 String input = scanner.nextLine().trim();  
 if (input.matches("\\d{7}") && Integer.*parseInt*(input) >= 1000000) { // Check if the input is a 7-digit number and >= 1000000  
 id = Integer.*parseInt*(input);  
  
 // Checks if the ID is unique.  
 int finalId = id;  
 if (PatronManaging.*patrons*.stream().noneMatch(patron -> patron.getId() == finalId)) {  
 return finalId; // Returns the valid ID.  
 } else {  
 System.*out*.println("ID is already in use. Please enter a different number.");  
 }  
 } else {  
 System.*out*.println("Invalid ID. It must be a 7-digit number and must start with a 1. Please try again.");  
 }  
 }  
 }  
  
 // Method is called to get a string input from the user.  
 public static String StringInput(Scanner scanner, String prompt) {  
 String input; // Local variable to hold the user input.  
 // Loops to ensure valid input is received.  
 do {  
 System.*out*.print(prompt);  
 input = scanner.nextLine().trim();  
 } while (input.isEmpty()); // Repeats until a string is entered.  
 return input; // Returns the valid input.  
 }  
  
 // Method is called to get the amount owed from the user.  
 public static double amountInput(Scanner scanner) {  
 double amountOwed; // Local variable to hold the amount owed.  
 // Loops until a valid amount is entered.  
 while (true) {  
 System.*out*.print("Amount Owed: ");  
 if (scanner.hasNextDouble()) {  
 amountOwed = scanner.nextDouble();  
 scanner.nextLine(); // Clears the line  
 if (amountOwed >= 0 && amountOwed <= 250) {  
 return amountOwed; // Return the valid amount  
 } else {  
 System.*out*.print("Please try again. Number must be between 0 and 250.\n");  
 }  
 } else {  
 System.*out*.print("Invalid Input. Please enter a valid number: ");  
 scanner.nextLine(); // Clears invalid input  
 }  
 }  
 }  
 // Method is called to get a valid patron ID to delete.  
 public static int patronIdDelete(Scanner scanner) {  
 while (true) {  
 System.*out*.print("Enter patron ID to remove: ");  
 String input = scanner.nextLine().trim().toLowerCase();  
  
 // Checks ID to be the specified range.  
 try {  
 int id = Integer.*parseInt*(input);  
 if (id >= 1000000 && id <= 9999999) {  
 return id; // Returns the valid ID.  
 } else {  
 System.*out*.println("Invalid ID. It must be exactly 7 digits.");  
 }  
 } catch (NumberFormatException e) {  
 System.*out*.println("Invalid input. Please enter a valid 7-digit number.");  
 }  
 }  
 }  
  
 // Helper method is called to find a patron by ID.  
 public static Patron findPatronById(int id) {  
 return PatronManaging.*patrons*.stream().filter(patron -> patron.getId() == id).findFirst().orElse(null); // Return the patron if found.  
 }  
  
 // Method is called to get confirmation from the user.  
 public static boolean userConfirmation(Scanner scanner, String message) {  
 // Loop until we get a valid response from the user  
 while (true) {  
 // Prompt the user with the provided message.  
 System.*out*.print(message);  
  
 // Read the user's input and normalizes it.  
 String choice = scanner.nextLine().trim().toLowerCase();  
  
 // Check if the user confirmed.  
 if (choice.equals("y") || choice.equals("yes")) {  
 return true; // User confirmed, return true.  
 }  
 // Check if the user declined.  
 else if (choice.equals("n") || choice.equals("no")) {  
 return false; // User declined, return false.  
 }  
 // Checks for invalid input.  
 else {  
 // Inform the user about the invalid input.  
 System.*out*.println("\nThat's not a valid response. Please enter y for yes or n for no.");  
 }  
 }  
 }  
}

* **PatronManaging**

// Otoniel Rodriguez-Perez  
// CEN-3024C-24204  
// 01/26/2025  
  
// PatronManaging Class:  
// This class manages all functions related to patrons.  
  
//Imported Library  
import java.util.\*;  
  
public class PatronManaging {  
  
 // Attribute to store all patrons in an Arraylist.  
 static final List<Patron> *patrons* = new ArrayList<>();  
  
 // Method is called to add a patron manually.  
 public void addPatronManual(Scanner scanner) {  
 // Loop to allow adding multiple patrons.  
 do {  
  
 System.*out*.println("Please enter the following patron information:\n");  
  
 // Get patron information through the user's input.  
 int id = UserHandling.*manualIdInput*(scanner); // Get a unique ID for the patron.  
 String name = UserHandling.*StringInput*(scanner, "Full Name: "); // Get patron's full name.  
 String address = UserHandling.*StringInput*(scanner, "Complete Address (Example - 123 Street Rd. Orlando, FL 12345): "); // Get patron's address.  
 double amountOwed = UserHandling.*amountInput*(scanner); // Get the amount owed by the patron.  
  
 //Patron information will be displayed before confirming.  
 System.*out*.print("\nPlease verify information is correct:\n");  
 System.*out*.println("\nID: " + id +  
 "\nName: " + name +  
 "\nAddress: " + address +  
 "\nAmount Owed: " + amountOwed);  
  
 //Asks user to confirm if they want to add patron.  
 if (UserHandling.*userConfirmation*(scanner, "\nAre you sure you want to add this patron? y or n: ")) {  
 // Add the new patron to the patrons Arraylist.  
 *patrons*.add(new Patron(id, name, address, amountOwed));  
 System.*out*.println("\nPatron successfully added!");  
 } else {  
 System.*out*.println("\nPatron not added.");  
 }  
  
 // Ask if the user wants to add another patron  
 } while (UserHandling.*userConfirmation*(scanner, "\nWould you like to continue to enter patron information? y or n:\n"));  
  
 System.*out*.println("\nReturning to the main menu.");  
 }  
  
 // Method is called to add patrons by file upload.  
 public void addPatronFile(Scanner scanner) {  
 // Prompt user for the file location.  
 System.*out*.print("\nEnter the file path for the patron list:\n");  
 System.*out*.print("Example (C:\\Users\\<YourUsername>\\Desktop\\<YourFileName>.txt)\n ");  
 String filePath = scanner.nextLine().trim();  
  
 // Create FileHandling class instance to process the file.  
 FileHandling fileUpload = new FileHandling();  
 fileUpload.addPatronsByFile(filePath, *patrons*); // Add patrons from the file  
 }  
  
 // Method is called to remove a patron.  
 public void removePatron(Scanner scanner) {  
 // Loop to allow user to remove multiple patrons.  
 while (true) {  
 int id = UserHandling.*patronIdDelete*(scanner); // Calls helper method the ID of the patron to remove.  
 Patron patron = UserHandling.*findPatronById*(id); // Calls helper method to find the patron by ID.  
  
 // Checks if the patron exists.  
 if (patron != null) {  
  
 //Patron information will be displayed before confirming.  
 System.*out*.print("\nPatron Found:\n");  
 System.*out*.println("\nID: " + patron.getId() +  
 "\nName: " + patron.getName() +  
 "\nAddress: " + patron.getAddress() +  
 "\nAmount Owed: " + patron.getAmountOwed());  
  
 // Ask for confirmation before deletion.  
 if (UserHandling.*userConfirmation*(scanner, "\nAre you sure you want to delete this patron? y or n: ")) {  
 *patrons*.remove(patron); // Remove the patron from the Arraylist.  
 System.*out*.println("\nPatron removed successfully. Returning to the main menu.");  
 } else {  
 System.*out*.println("\nPatron deletion canceled. Returning to the main menu."); // Inform user does not confirm deletion.  
 }  
 return; // Exit the method after handling the deletion.  
 } else {  
 System.*out*.println("Patron not found in the system.\n"); // Inform user ID is not in the system.  
  
 // Ask if the user wants to try again.  
 if (!UserHandling.*userConfirmation*(scanner, "Would you like to try again? y or n\n(Entering n will take you back to the main menu): ")) {  
 return; // Exit the method if user chooses not to try.  
  
 }  
 }  
 }  
 }  
  
 // Method is called to view all patrons.  
 public void viewAllPatrons() {  
 // Check if there are any patrons in the ArrayList.  
 if (*patrons*.isEmpty()) {  
 System.*out*.println("\nNo patrons found.\n"); // Inform user if no patrons are present.  
 } else {  
 //Sorts by ID number.  
 *patrons*.sort(Comparator.*comparing*(Patron::getId));  
 System.*out*.println("\nList of Patrons:");  
 // Iterate through each patron and format the output.  
 for (Patron patron : *patrons*) {  
 System.*out*.println("\nID: " + patron.getId() +  
 "\nName: " + patron.getName() +  
 "\nAddress: " + patron.getAddress() +  
 "\nAmount Owed: " + patron.getAmountOwed() + "\n");  
 }  
 }  
 System.*out*.println("Returning to the main menu.");  
 }  
  
}

* **FileHandling**

// Otoniel Rodriguez-Perez  
// CEN-3024C-24204  
// 01/26/2025  
  
// FileHandler Class:  
// This class handles file operations and conditions for adding patrons.  
  
// Imported Libraries  
import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
import java.util.\*;  
import java.util.stream.Collectors;  
import java.util.Scanner;  
  
public class FileHandling {  
 //Method is called to read and process file.  
 public void addPatronsByFile(String filePath, List<Patron> patrons) {  
 //BufferedReader will attempt to read the file.  
 try (BufferedReader readFile = new BufferedReader(new FileReader(filePath))) {  
 //Lists are specifically called to check that contents within file are correct.  
 List<String> invalidEntries = new ArrayList<>();  
 List<String> invalidIDs = new ArrayList<>();  
 List<String> outOfRangeAmountOwed = new ArrayList<>();  
 Map<Integer, Integer> idCountMap = new HashMap<>(); // Tracks ID counts.  
 List<Patron> validPatrons = new ArrayList<>();  
 //User will confirm if patrons are to be added to the main patrons ArrayList.  
 Scanner scanner = new Scanner(System.*in*);  
  
 // Reads rows from the text file.  
 String row;  
 while ((row = readFile.readLine()) != null) {  
 String[] details = row.split("-");  
 if (details.length != 4) {  
 invalidEntries.add(row);  
 continue;  
 }  
  
 try {  
 int id = Integer.*parseInt*(details[0].trim());  
 // Checks for unique and duplicate IDs withing fie.  
 idCountMap.put(id, idCountMap.getOrDefault(id, 0) + 1);  
  
 // Checks for correct ID range.  
 if (id < 1000000 || id > 9999999) {  
 invalidIDs.add(row);  
 continue;  
 }  
  
 // Checks for amount owed range.  
 double amountOwed = Double.*parseDouble*(details[3].trim());  
 if (amountOwed < 0 || amountOwed > 250) {  
 outOfRangeAmountOwed.add(row);  
 continue;  
 }  
  
 // Temporary list to hold all valid patrons.  
 String name = details[1].trim();  
 String address = details[2].trim();  
 validPatrons.add(new Patron(id, name, address, amountOwed));  
 } catch (NumberFormatException e) {  
 invalidEntries.add(row);  
 }  
 }  
  
 // Identify if there are duplicates within the file.  
 Set<Integer> duplicatesInFile = idCountMap.entrySet().stream()  
 .filter(entry -> entry.getValue() > 1)  
 .map(Map.Entry::getKey)  
 .collect(Collectors.*toSet*());  
  
 // Checks for duplicates against the patrons in the ArrayList.  
 Set<Integer> existingPatronIDs = patrons.stream()  
 .map(Patron::getId)  
 .collect(Collectors.*toSet*());  
  
 // All duplicates are combined into one list for output.  
 Set<Integer> allDuplicates = new HashSet<>(duplicatesInFile);  
 allDuplicates.addAll(existingPatronIDs);  
  
 // Filter valid patrons to remove any that are duplicates.  
 validPatrons.removeIf(patron -> allDuplicates.contains(patron.getId()));  
  
 //Message is displayed if there is no valid patrons to add.  
 if (validPatrons.isEmpty()){  
 System.*out*.println("\nNo valid patrons found.");  
 }  
 // Displays all valid patron before being added.  
 if(!validPatrons.isEmpty()) {  
 System.*out*.println("\nPlease see below valid patron(s) to be uploaded to the application.\n");  
 validPatrons.forEach(System.*out*::println);  
  
 // User confirms if valid patrons are added.  
 if (UserHandling.*userConfirmation*(scanner, "Are you sure you want to add patron(s)? y or n: ")) {  
 patrons.addAll(validPatrons);  
 System.*out*.println("\nPatron(s) added."); // Add valid patron(s) to the main patrons Arraylist.  
 } else {  
 System.*out*.println("\nPatron(s) not added."); // User selects not to add patron(s).  
 }  
 }  
 // Display messages for any issues encountered.  
 if (!allDuplicates.isEmpty()) {  
 System.*out*.println("\nThe following IDs are duplicated within the file or in the LMS application and were skipped:\n");  
 allDuplicates.forEach(System.*out*::println);  
 }  
  
 // Checks if there are invalid IDs to display.  
 if (!invalidIDs.isEmpty() || !outOfRangeAmountOwed.isEmpty() || !invalidEntries.isEmpty()) {  
 if (!allDuplicates.isEmpty()) {  
 System.*out*.println();  
 }  
  
 if (!invalidIDs.isEmpty()) {  
 System.*out*.println("\nThe following entries have invalid IDs (must be exactly 7 digits and start with a 1)\nor incorrect formatting and were skipped:\n");  
 invalidIDs.forEach(System.*out*::println);  
 }  
  
 if(!invalidEntries.isEmpty()){  
 System.*out*.println("\nThe following entries have incorrect formatting and were skipped:\n");  
 invalidEntries.forEach(System.*out*::println);  
 }  
  
 if (!outOfRangeAmountOwed.isEmpty()) {  
 System.*out*.println("\nThe following entries have invalid amounts owed (must be between $0 and $250) and were skipped:\n");  
 outOfRangeAmountOwed.forEach(System.*out*::println);  
 }  
 }  
  
 // Final message is displayed whether any errors occurred or not.  
 System.*out*.println("\nFile processing complete. Returning to the main menu.");  
  
 } catch (IOException e) {  
 System.*out*.println("\nError reading the file. Please ensure the file path is correct.\nReturning to the main menu.\n");  
 }  
 }  
}