## 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

## 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
  + Choosing to close the application.
* The application doesn’t have any GUI (Graphical User Interface). The user must use the CLI (Command Line Interface).
* The latest JDK (Java Development Kit) must be installed.
* The application must run either on Windows or Mac OS operating application.
* Application must run when converted into a JAR file.

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 CLI onscreen menu.
13. There will be exceptions to ensure the stored information is correct.
14. There will be loops 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.
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.
     + 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.
     + userConfirmation()
       - **Arguments:** Scanner scanner, String message (Methods used for user input)
       - **Returns:** String 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:** Searchers patrons ArrayList for ID.
5. **FileHandling:**
   * **Variables:**
     + none
   * **Methods:**
     + addPatronsbyFile()
       - **Arguments:** String filePath, List<Patron> patrons (String for file location and patrons ArrayList)
       - **Returns:** A new patron is added to the patrons ArrayList.
6. **Algorithms:**

* **Add Patron Manually Algorithm:**
  1. User enters 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, a Patron object (instance) is created.
  4. Patron instance is added to the next available position in the patrons ArrayList.
  5. Patron instance information is stored within this list in the application.
* **Add Patron by File Algorithm:**
  1. User enters 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 iterate an error message until the correction are made. Any patron within the file that has a duplicate ID is skipped.
  3. Once correct information is uploaded, multiple Patron objects (instances) are created.
  4. Patron instances are added to the next available positions in the patrons ArrayList.
  5. 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.
  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 decision.
  6. The patron record is deleted from the patrons ArrayList.
* **View Patron Algorithm**
  1. User selects the option to view all patrons from the onscreen menu.
  2. The application searches for the information stored in the patrons ArrayLlist.
  3. A for loop is activated to iterate through the patrons ArrayList.
  4. All patrons are displayed.

1. **UML Diagrams:**

|  |
| --- |
| **Librarian** |
| - patronManaging = new PatronManaging(): final  - userHandling = new UserHandling(): final |
| + 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 |

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

|  |
| --- |
| **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. Source Code:**

* **Library:**
* // 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.  
    
  // Create instances of PatronManager and UserInputHandler  
  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)!\n");  
    
   // Main menu will iterate until the user chooses to close the application.  
   do {  
   *viewMenu*(); // Display the main menu options  
   // Calls method to verify user input.  
   userChoice = *userHandling*.usersChoice(scanner);  
    
   // Switch statement to handle user choices  
   switch (userChoice) {  
   case 1:  
   *patronManaging*.addPatronManual(scanner); // Option to add a patron manually  
   break;  
   case 2:  
   *patronManaging*.addPatronFile(scanner); // Option to add patrons by file upload  
   break;  
   case 3:  
   *patronManaging*.removePatron(scanner); // Option to remove a patron  
   break;  
   case 4:  
   *patronManaging*.viewAllPatrons(); // Option to view all patrons  
   break;  
   case 5:  
   System.*out*.println("Thank you for using the LMS application. Goodbye!"); // Exit message  
   break;  
   default:  
   System.*out*.println("Invalid option. Please enter a number from 1 to 5."); // Error message for invalid input  
   break;  
   }  
   } while (userChoice != 5); // Continue looping until the user chooses to exit  
    
   scanner.close(); // Close the scanner resource  
  }  
    
  // Method to 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**
* // 1. Patron Class: used to define what a kind of information a patron will contain.  
  //It will be called by the Librarian class to store, view, or remove patrons.  
    
  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 "\nID: " + id + "\nName: " + name + "\nAddress: " + address + "\nAmount Owed: " + amountOwed;  
   }  
  }
* **UserHandling**
* // UserInputHandler Class  
  // This class handles user input for menu selection  
  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(); // Clear the buffer  
   if (choice >= 1 && choice <= 5) {  
   return choice; // Return valid choice  
   }  
   System.*out*.println("Invalid input. Please select a number between 1 and 5."); // Error message for invalid input  
   } catch (InputMismatchException e) {  
   System.*out*.println("Invalid input. Please enter a number."); // Error message for non-numeric input  
   scanner.nextLine(); // Clear the invalid input  
   }  
   }  
   }  
    
    
  }
* **PatronManaging**
* // PatronManager Class  
  // This class manages all operations related to patrons in the library system  
  import java.util.\*;  
    
  public class PatronManaging {  
   // Attribute to store all patrons in a list  
   private final List<Patron> patrons = new ArrayList<>();  
    
   // Method 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 details through user input  
   int id = manualIdInput(scanner); // Get unique ID for the patron  
   String name = StringInput(scanner, "Full Name: "); // Get patron's name  
   String address = StringInput(scanner, "Address:\nExample (123 Street Rd. Orlando, FL 12345)\n"); // Get patron's address  
   double amountOwed = amountInput(scanner); // Get the amount owed by the patron  
    
   // Add the new patron to the list  
   patrons.add(new Patron(id, name, address, amountOwed));  
   System.*out*.println("\nPatron successfully added!\n");  
    
   // Ask if the user wants to add another patron  
   } while (userConfirmation(scanner, "Would you like to add another patron? y or n\n"));  
   }  
    
   // Method to add patrons by file upload  
   public void addPatronFile(Scanner scanner) {  
   // Prompt user for the file path  
   System.*out*.print("Enter 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 FileHandler instance to process the file  
   FileHandling fileHandling = new FileHandling();  
   fileHandling.addPatronsByFile(filePath, patrons); // Add patrons from the file  
   }  
    
   // Method to remove a patron  
   public void removePatron(Scanner scanner) {  
   // Loop to allow user to remove multiple patrons  
   while (true) {  
   int id = patronIdDelete(scanner); // Get the ID of the patron to remove  
   Patron patron = findPatronById(id); // Find the patron by ID  
    
   // Check if the patron exists  
   if (patron != null) {  
   // Ask for confirmation before deletion  
   if (userConfirmation(scanner, "Are you sure you want to delete this patron? (y/n): ")) {  
   patrons.remove(patron); // Remove the patron from the list  
   System.*out*.println("\nPatron removed successfully.");  
   } else {  
   System.*out*.println("Patron deletion canceled."); // Inform user if deletion is canceled  
   }  
   return; // Exit the method after handling the removal  
   } else {  
   System.*out*.println("Patron not found in the system."); // Inform user if patron is not found  
    
   // Ask if the user wants to try again  
   if (!userConfirmation(scanner, "Would you like to try again? (y/n): ")) {  
   return; // Exit the method if user chooses not to try again  
   }  
   }  
   }  
   }  
    
    
   // Method to view all patrons  
   public void viewAllPatrons() {  
   // Check if there are any patrons in the list  
   if (patrons.isEmpty()) {  
   System.*out*.println("No patrons found."); // Inform user if no patrons are present  
   } else {  
   patrons.forEach(System.*out*::println); // Print all patrons  
   }  
   }  
    
   // Method to get a patron ID from user input  
   private int manualIdInput(Scanner scanner) {  
   int id; // Local variable to hold the patron ID  
    
   // Loop until a valid 7-digit ID is entered  
   while (true) {  
   System.*out*.print("Enter ID: ");  
   String input = scanner.nextLine().trim();  
   if (input.matches("\\d{7}")) { // Check if the input is a 7-digit number  
   id = Integer.*parseInt*(input);  
    
   // Check if the ID is unique  
   int finalId = id;  
   if (patrons.stream().noneMatch(patron -> patron.getId() == finalId)) {  
   return finalId; // Return the valid ID  
   } else {  
   System.*out*.println("ID is already in the system. Please enter a different number.");  
   }  
   } else {  
   System.*out*.println("Invalid ID. It must be a 7-digit number. Please try again.");  
   }  
   }  
   }  
    
   // Method to get a string input from the user  
   private String StringInput(Scanner scanner, String prompt) {  
   String input; // Local variable to hold the user input  
   // Loop to ensure valid input is received  
   do {  
   System.*out*.print(prompt);  
   input = scanner.nextLine().trim();  
   } while (input.isEmpty()); // Repeat until a non-empty string is entered  
   return input; // Return the valid input  
   }  
    
   // Method to get the amount owed from the user  
   private double amountInput(Scanner scanner) {  
   double amountOwed; // Local variable to hold the amount owed  
   // Loop until a valid amount is entered  
   while (true) {  
   System.*out*.print("Amount Owed (0 - 250): ");  
   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("Invalid amount. Number must be between 0 and 250: ");  
   }  
   } else {  
   System.*out*.print("Invalid input. Please enter a valid number: ");  
   scanner.nextLine(); // Clears invalid input  
   }  
   }  
   }  
    
   // Method to get confirmation from the user  
   private 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 normalize 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  
   }  
   // Handle invalid input  
   else {  
   // Inform the user about the invalid input  
   System.*out*.println("Oops! That's not a valid response. Please enter 'y' for yes or 'n' for no.");  
   }  
   }  
   }  
   // Method to get a valid patron ID to delete  
   private int patronIdDelete(Scanner scanner) {  
   while (true) {  
   System.*out*.print("Enter patron ID to remove (7 digits, or type 'b' to return): ");  
   String input = scanner.nextLine().trim().toLowerCase();  
    
   // Check if the user wants to return to the main menu  
   if (input.equals("b")) {  
   System.*out*.println("Returning to the main menu");  
   return -1; // Signal to return  
   }  
    
   try {  
   int id = Integer.*parseInt*(input);  
   if (id >= 1 && id <= 9999999) {  
   return id; // Return 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.");  
   }  
   }  
   }  
    
   // Method to find a patron by ID  
   private Patron findPatronById(int id) {  
   return patrons.stream().filter(patron -> patron.getId() == id).findFirst().orElse(null); // Return the patron if found  
   }  
  }
* **FileHandling**
* // FileHandler Class  
  // This class handles file operations for adding patrons  
  import java.io.BufferedReader;  
  import java.io.FileReader;  
  import java.io.IOException;  
  import java.util.\*;  
  import java.util.stream.Collectors;  
    
  public class FileHandling {  
   // Method asks the user to upload a file and add patrons from it  
   public void addPatronsByFile(String filePath, List<Patron> patrons) {  
   // BufferedReader will try to read the file  
   try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {  
   // Lists to track errors and duplicates  
   List<String> duplicateIDs = new ArrayList<>();  
   List<String> invalidEntries = new ArrayList<>();  
   List<String> invalidIDs = new ArrayList<>();  
   List<String> outOfRangeAmountOwed = new ArrayList<>();  
   // Create a set of existing IDs for quick lookup  
   Set<Integer> existingIDs = patrons.stream().map(Patron::getId).collect(Collectors.*toSet*());  
   Map<Integer, Integer> fileIDCounts = new HashMap<>();  
   List<Patron> validPatrons = new ArrayList<>();  
   boolean hasErrors = false; // Flag to check if any errors occurred  
    
   // Read lines from the file  
   String line;  
   while ((line = reader.readLine()) != null) {  
   // Split the line into details based on the hyphen  
   String[] details = line.split("-");  
   if (details.length != 4) {  
   invalidEntries.add(line); // Add to invalid entries if format is incorrect  
   hasErrors = true; // Set error flag  
   continue;  
   }  
    
   // Validate the ID and other details  
   try {  
   int id = Integer.*parseInt*(details[0].trim());  
   fileIDCounts.put(id, fileIDCounts.getOrDefault(id, 0) + 1); // Count occurrences of each ID  
    
   // Check if ID is in the valid range  
   if (id < 1 || id > 9999999) {  
   invalidIDs.add(line); // Add to invalid IDs if out of range  
   hasErrors = true; // Set error flag  
   continue;  
   }  
   // Check for duplicates in existing patrons  
   if (existingIDs.contains(id)) {  
   duplicateIDs.add(line); // Add to duplicates if already exists  
   hasErrors = true; // Set error flag  
   continue;  
   }  
   // Store valid patron information  
   String name = details[1].trim();  
   String address = details[2].trim();  
    
   // Validate the amount owed  
   double amountOwed = Double.*parseDouble*(details[3].trim());  
   if (amountOwed < 0 || amountOwed > 250) {  
   outOfRangeAmountOwed.add(line); // Add to out of range amounts  
   hasErrors = true; // Set error flag  
   continue;  
   }  
    
   // Add the valid patron to the list  
   validPatrons.add(new Patron(id, name, address, amountOwed));  
   } catch (NumberFormatException e) {  
   invalidEntries.add(line); // Add to invalid entries on format error  
   hasErrors = true; // Set error flag  
   }  
   }  
    
   // Check for duplicates in the valid patrons list  
   for (Map.Entry<Integer, Integer> entry : fileIDCounts.entrySet()) {  
   if (entry.getValue() > 1) {  
   int duplicateID = entry.getKey();  
   duplicateIDs.addAll(  
   validPatrons.stream().filter(p -> p.getId() == duplicateID).map(Patron::toString).toList()  
   );  
   validPatrons.removeIf(p -> p.getId() == duplicateID); // Remove duplicates  
   }  
   }  
    
   // Add all valid patrons to the main list  
   patrons.addAll(validPatrons);  
    
   // Display messages for any issues encountered  
   if (!duplicateIDs.isEmpty()) {  
   System.*out*.println("\nThe following entries have duplicate IDs and were skipped:");  
   duplicateIDs.forEach(System.*out*::println); // Print duplicates  
   }  
    
   if (!invalidIDs.isEmpty()) {  
   System.*out*.println("\nThe following entries have invalid IDs (must be exactly 7 digits) and were skipped:");  
   invalidIDs.forEach(System.*out*::println); // Print invalid IDs  
   }  
    
   if (!outOfRangeAmountOwed.isEmpty()) {  
   System.*out*.println("\nThe following entries have invalid amounts owed (must be between $0 and $250) and were skipped:");  
   outOfRangeAmountOwed.forEach(System.*out*::println); // Print out of range amounts  
   }  
    
   // Final message based on whether any errors occurred  
   if (hasErrors) {  
   System.*out*.println("\nFile processing completed with errors.");  
   } else {  
   System.*out*.println("\nFile processing completed with no errors."); // Success message  
   }  
   } catch (IOException e) {  
   System.*out*.println("Error reading the file. Please ensure the file path is correct."); // Error message for file issues  
   }  
   }  
  }