2023

Module 2 | SDLC Assignment Part 2

CEN-3024C-17125

AXEL DIAZ MIJARES

Monday, August 28, 2023

DEFINE REQUIREMENTS

FEATURES AND FUNCTIONALITY

The LMS is a console-based application with specific functionalities:

- Add New Books: The system must allow the librarian to add new books to the collection by importing them from a text file. Each book has a unique ID, title, and author.
- Remove a Book: The system must enable the librarian to remove a book from the collection using its unique ID number.
- List All Books: The system must provide a feature to list all the books currently in the collection.

USERS

Understanding the users and their roles is crucial:

- Librarians: They are responsible for managing the library's collection, including adding new books from a text file, removing books using the ID number, and viewing the entire collection.
- Readers: They can view the list of books in the collection but do not have permission to modify it.

CONSTRAINTS

The system must adhere to specific constraints:

- Console-Based Interface: The application must run in a console or terminal window.
- Text File Format: The text file used to import books must be formatted with each line representing a book, and the ID, title, and author separated by a comma.

GATHER REQUIREMENTS

INTENDED USERS AND INTERACTION

Understanding how users will interact with the system is vital:

Librarians:

- **Add Books**:
    - **Provide a Text File**: Librarians can add multiple books at once by providing a text file that follows a specific format. This allows for efficient management of large collections.
    - **Confirmation and Error Messages**: The system will provide feedback on the success or failure of the addition, including detailed error messages if the file format is incorrect or other issues arise.
- **Remove Books**:
    - **Enter ID Number**: Librarians can remove a specific book by entering its unique ID number.
    - **Confirmation and Error Messages**: The system will confirm the removal or provide an error message if the ID is not found.
- **View Collection**:
    - **View Anytime**: The entire collection can be viewed at any time, providing a comprehensive overview of the library's holdings.
    - **Pagination**: If the collection is large, the system will display it in pages, making it easier to navigate.
- **Error Handling**:
    - **Clear Error Messages**: The system will provide clear and informative error messages for invalid inputs, file errors, and other issues.
    - **Continued Operation**: Even if an error occurs in one operation, the system will allow the librarian to continue with other tasks.

Readers:

- **View Collection**:
    - **View Only**: Readers can view the list of books but cannot modify it, ensuring that only authorized personnel can make changes.

- **Pagination**: The search results will be displayed in pages if there are many results, providing a user-friendly experience.

IMPLEMENTATION PLAN

The implementation phase will be structured into smaller, manageable tasks:

CREATE BOOK CLASS

The **Book** class in the Library Management System serves as the data model for individual books. It contains attributes like ID, title, and author, and provides methods for data manipulation and access.

1. CONSTRUCTOR BOOK(INT ID, STRING TITLE, STRING AUTHOR)

Arguments: **int id**, **String title**, **String author**

Return Value: None

Algorithm: Initialize a new **Book** object with the provided ID, title, and author.

2. GETID()

Arguments: None

Return Value: **int** (Book's ID)

Algorithm: Return the ID attribute of the **Book** object.

3. GETTITLE()

Arguments: None

Return Value: **String** (Book's title)

Algorithm: Return the title attribute of the **Book** object.

4. GETAUTHOR()

Arguments: None

Return Value: **String** (Book's author)

Algorithm: Return the author attribute of the **Book** object.

5. SETTITLE(STRING TITLE)

Arguments: **String title**

Return Value: None

Algorithm: Update the title attribute of the **Book** object.

6. SETAUTHOR(STRING AUTHOR)

Arguments: **String author**

Return Value: None

Algorithm: Update the author attribute of the **Book** object.

7. TODICT()

Arguments: None

Return Value: **HashMap<String, String>** (Dictionary representation of the book)

Algorithm:

Create a new **HashMap**.

Populate it with the **id**, **title**, and **author** attributes.

Return the populated **HashMap**.

8. TOSTRING()

Arguments: None

Return Value: **String** (String representation of the book)

Algorithm: Return a string concatenating the **id**, **title**, and **author** attributes, formatted as "ID: [id], Title: [title], Author: [author]".

equals(Book other)

Arguments: **Book other**

Return Value: **boolean** (True if IDs are equal, false otherwise)

Algorithm: Compare the ID of the current **Book** object with that of another **Book** object to determine equality.

CREATE LIBRARY CLASS

The **Library** class serves as the core data structure for managing a collection of books in the Library Management System. It offers functionalities such as adding books from a file, removing books by ID, listing all books with pagination, and retrieving attributes of the book collection.

1. CONSTRUCTOR **LIBRARY()**

Arguments: None

Return Value: None

Algorithm: Initialize an empty ArrayList of **Book** objects.

2. ADDBOOK(BOOK BOOK)

Arguments: **Book book**

Return Value: None

Algorithm: Add a **Book** object to the ArrayList **books**.

3. ADDBOOKSFROMFILE(STRING FILEPATH, INT BATCHSIZE)

Arguments: **String filePath**, **int batchSize**

Return Value: **String** (Summary of errors, if any)

Algorithm:

Read a file line by line.

Validate and parse each line into **Book** objects.

Add the **Book** objects to the **books** ArrayList in batches.

4. REMOVEBOOKBYID(INT ID)

Arguments: **int id**

Return Value: **boolean** (true if removed, false otherwise)

Algorithm:

Remove a **Book** object with a matching ID from the **books** ArrayList.

Return **true** if a book was successfully removed; otherwise, return **false**.

5. LISTALLBOOKS(INT PAGE, INT PAGESIZE, INT DYNAMICWIDTH)

Arguments: **int page**, **int pageSize**, **int dynamicWidth**

Return Value: None

Algorithm:

Calculate the start and end indices for pagination.

Print out the books within those indices, formatted to fit a dynamically calculated table width.

6. GETBOOKBYINDEX(INT INDEX)

Arguments: **int index**

Return Value: **Book** or **null**

Algorithm:

Check if the index is within the bounds of the **books** ArrayList.

Return the **Book** object at the specified index or **null** if out of bounds.

7. GETTOTALBOOKS()

Arguments: None

Return Value: **int** (Total number of books)

Algorithm: Return the size of the **books** ArrayList.

8. GETMAXID()

Arguments: None

Return Value: **int** (Maximum book ID)

Algorithm:

Iterate through the **books** ArrayList.

Return the highest ID value found.

9. GETMAXTITLELENGTH()

Arguments: None

Return Value: **int** (Maximum title length)

Algorithm:

Use Java Streams to find the maximum title length among all **Book** objects in the ArrayList.

10. GETMAXAUTHORLENGTH()

Arguments: None

Return Value: **int** (Maximum author name length)

Algorithm:

Use Java Streams to find the maximum author name length among all **Book** objects in the ArrayList.

CREATE MAIN PROGRAM

CONSOLE-BASED INTERFACE:

The Main program serves as the entry point for the Library Management System. It is responsible for initializing the **Library** class and providing a console-based interface for managing the library. Below are the key methods and functionalities:

MAIN METHOD

Arguments: **String[] args**

Return Value: None

Algorithm:

Initialize the **Library** and **Scanner** objects.

Add books from a file using **addBooksFromFile()**.

Calculate the dynamic table width for displaying information.

Run an infinite loop to display the menu and capture user choices.

2. CLEARSCREEN()

Arguments: None

Return Value: None

Algorithm: Clear the console screen.

3. CALCULATEDYNAMICWIDTH(LIBRARY LIBRARY)

Arguments: **Library library**

Return Value: **int** (Dynamic table width)

Algorithm:

Calculate the maximum lengths for IDs, titles, and authors in the library.

Calculate a dynamic width for the table based on these dimensions.

4. DISPLAYMENU(INT TABLEWIDTH)

Arguments: **int tableWidth**

Return Value: None

Algorithm: Display the main menu with options to add, remove, list, and exit.

5. DISPLAYHEADER(INT DYNAMICWIDTH)

Arguments: **int dynamicWidth**

Return Value: None

Algorithm: Display the header for the Library Management System.

6. DISPLAYLOG(INT DYNAMICWIDTH)

Arguments: **int dynamicWidth**

Return Value: None

Algorithm: Display a log of messages, including errors and notifications.

7. LOGMESSAGE(STRING TYPE, STRING MESSAGE)

Arguments: **String type**, **String message**

Return Value: None

Algorithm: Log messages to a predefined log list, keeping only the last 5 messages.

8. LISTALLBOOKS(LIBRARY LIBRARY, SCANNER SCANNER, INT DYNAMICWIDTH)

Arguments: **Library library**, **Scanner scanner**, **int dynamicWidth**

Return Value: None

Algorithm:

List all books in the library with pagination.

Handle user navigation for next, previous, and quit actions.

9. ADDBOOKSFROMFILE(LIBRARY LIBRARY, SCANNER SCANNER)

Arguments: **Library library**, **Scanner scanner**

Return Value: None

Algorithm:

Call the **addBooksFromFile** method from the **Library** class.

Log any errors or successes.

10. REMOVEBOOKBYID(LIBRARY LIBRARY, SCANNER SCANNER)

Arguments: **Library library**, **Scanner scanner**

Return Value: None

Algorithm:

Prompt the user for the ID of the book to be removed.

Call the **removeBookById** method from the **Library** class.

Log whether the operation was successful or not.

11. EXIT(SCANNER SCANNER)

Arguments: **Scanner scanner**

Return Value: None

Algorithm:

Log the exit message.

Close the scanner to terminate the program.

TESTING PLAN

Testing is essential to ensure that the system functions correctly:

UNIT TESTING

BOOK CLASS:

1. TEST FOR PROPER INITIALIZATION WITH VARIOUS ATTRIBUTES.
2. TEST STRING REPRESENTATION.
3. TEST EQUALITY COMPARISON.
4. TEST GETTERS AND SETTERS.

LIBRARY CLASS:

1. TEST ADDING A SINGLE BOOK AND DUPLICATE BOOK HANDLING.
2. TEST REMOVING A BOOK BY ID.
3. TEST GETTING A BOOK BY ID.
4. TEST LISTING ALL BOOKS.

INTEGRATION TESTING

SIMULATE USER INTERACTIONS TO TEST THE ENTIRE SYSTEM'S FUNCTIONALITY.

TEST THE INTEGRATION OF THE BOOK, LIBRARY, AND MAIN PROGRAM CLASSES.

TEST THE COMPLETE WORKFLOW OF ADDING, REMOVING AND LISTING.

EDGE CASES:

1. TEST WITH EMPTY FILES.
2. TEST WITH INCORRECT FILE FORMATS.
3. TEST WITH INVALID IDS.
4. TEST WITH OTHER UNEXPECTED INPUTS.
5. TEST WITH LARGE FILES (STRESS TESTING).

5. DEPLOY SOFTWARE:

Main.java:

```
import java.io.*;
import java.util.*;

/**Axel Diaz | CEN 3024C Software Development | - CRN: 17125
 * Main
 * The Main class serves as the entry point for a Library Management System.
 * The primary objective of this program is to provide a console-based interface
 * for managing a library of books. Users can import books from a file, remove books by ID,
 * list all books, and exit the program.
 */
public class Main {
	private static List<String> log = new ArrayList<>();
	private static final int MIN_TABLE_WIDTH = 80;
	private static final String case_path = "test_case_1.txt";

	public static void main(String[] args) {
		Library library = new Library();
		Scanner scanner = new Scanner(System.in);
		addBooksFromFile(library, scanner);
		int dynamicWidth = MIN_TABLE_WIDTH;

		while (true) {
			dynamicWidth = calculateDynamicWidth(library);
			clearScreen();
			displayHeader(dynamicWidth);
			displayMenu(dynamicWidth);
			displayLog(dynamicWidth);
```

```
                int choice = scanner.nextInt();
                scanner.nextLine();

                switch (choice) {
                case 1:
                        addBooksFromFile(library, scanner);
                        break;
                case 2:
                        removeBookById(library, scanner);
                        break;
                case 3:
                        listAllBooks(library, scanner, dynamicWidth);
                        break;
                case 4:
                        exit(scanner);
                        return;
                default:
                        logMessage("[ERROR]", "Invalid choice. Please try again.");
                }
        }
}

private static void clearScreen() {
        System.out.print("\033[H\033[2J");
        System.out.flush();
}
/**calculateDynamicWidth
```

```
    * Calculates the dynamic width of the table based on the maximum title length and author length in the library.
    *
    * @param library The library instance.
    * @return The dynamic width.
    */
    private static int calculateDynamicWidth(Library library) {
            int maxTitleLength = library.getMaxTitleLength();
            int maxAuthorLength = library.getMaxAuthorLength();

            int idPadding = 3;
            int maxIdLength = String.valueOf(library.getMaxId()).length() + idPadding;

            int extraSpacesForSeparators = 20;
            int bookRowWidth = maxTitleLength + maxAuthorLength + maxIdLength + extraSpacesForSeparators;

            int statsWidth = String.format("Total Books: %d | Total Pages: %d | Current Page: %d", library.getTotalBooks(), (library.getTotalBooks() / 15) + 1, (library.getTotalBooks() / 15) + 1).length();

            int menuPadding = 2;
            int menuWidth = "4. Exit Program".length() + menuPadding;

            int logPadding = 2;
            int logWidth = "Books added with some errors:".length() + logPadding;

            int navPromptPadding = 2;
            int navPromptWidth = "Press 'n' for next page, 'p' for previous page, 'q' to quit listing.".length() + navPromptPadding;
```

```
            int uniformWidth = Math.max(Math.max(Math.max(Math.max(bookRowWidth, menuWidth),
logWidth), navPromptWidth), statsWidth);

            return uniformWidth;

    }


    /**displayMenu

     * Displays the menu options.

     *

     * @param tableWidth The width of the table.

     */

    private static void displayMenu(int tableWidth) {

            String formatString = "| %-" + (tableWidth - 2) + "s|\n";

            System.out.printf(formatString, "Menu Options");

            System.out.printf(formatString, "1. Import Books from a File");

            System.out.printf(formatString, "2. Remove Book by ID");

            System.out.printf(formatString, "3. List All Books");

            System.out.printf(formatString, "4. Exit Program");


    }

    /**displayHeader

     * Displays the header of the library management system.

     *

     * @param dynamicWidth The dynamic width of the table.

     */

    private static void displayHeader(int dynamicWidth) {

            System.out.println("+"

                    + "-".repeat(dynamicWidth - 2) + "+");
```

```
        System.out.printf("| %-" + (dynamicWidth - 2) + "s|\n", "Library Management System");
        System.out.println("+"
                + "-".repeat(dynamicWidth - 2) + "+");
}
/**displayLog
 * Displays the log of messages.
 *
 * @param dynamicWidth The dynamic width of the table.
 */
private static void displayLog(int dynamicWidth) {

            //Border
            System.out.println("+"
            + "-".repeat(dynamicWidth - 2) + "+");
        String formatString = "| %-" + (dynamicWidth - 2) + "s|\n";
        int logCount = 0;
        for (String message : log) {
            String[] logStrings = message.split("\n");
            for (String string : logStrings) {
                System.out.printf(formatString, string);
                logCount++;
            }
        }
        for (int i = logCount; i < 5; i++) {
            System.out.printf(formatString, " ");
        }
            System.out.println("+"
```

```
                + "-".repeat(dynamicWidth - 2) + "+");
    }

    /**
     * Logs a message to the log.
     *
     * @param type    The type of message.
     * @param message The message to log.
     */
    private static void logMessage(String type, String message) {
        String logMsg = type + ": " + message;
        // Preserve the last 5 distinct messages, not lines
        if (log.size() >= 5) {
            log.remove(0);
        }
        log.add(logMsg);
    }

    /**listAllBooks
     * Lists all books in the library.
     *
     * @param library      The library instance.
     * @param scanner      The scanner for user input.
     * @param dynamicWidth The dynamic width of the table.
     */
    private static void listAllBooks(Library library, Scanner scanner, int dynamicWidth) {
        int page = 0;
```

```
int pageSize = 15;
int totalBooks = library.getTotalBooks();
int totalPages = (totalBooks + pageSize - 1) / pageSize;

// Calculate the lengths of the longest ID, Title, and Author
int maxIdLength = String.valueOf(library.getMaxId()).length();
int maxTitleLength = library.getMaxTitleLength() - 1;
int maxAuthorLength = library.getMaxAuthorLength() - 2;

// Calculate the required table width
int requiredWidth = maxIdLength + maxTitleLength + maxAuthorLength + 6;

// Calculate extra spaces needed to fill the dynamic table width
int extraSpaces = dynamicWidth - requiredWidth;

// Distribute extra spaces equally to Title and Author fields
int extraSpacesForTitle = extraSpaces / 2;
int extraSpacesForAuthor = extraSpaces / 2;

// If there's an odd number of extra spaces, add the extra one to Title
if (extraSpaces % 2 != 0) {
        extraSpacesForTitle += 1;
}

// Update the max lengths for Title and Author
maxTitleLength += extraSpacesForTitle;
maxAuthorLength += extraSpacesForAuthor;
```

```
        String rowFormat = "| %-" + maxIdLength + "s | %-" + maxTitleLength + "s | %-" +
maxAuthorLength + "s |\n";

        int tableWidth = Math.max(dynamicWidth, requiredWidth);


        while (true) {

            clearScreen();

            displayHeader(dynamicWidth);

            displayMenu(dynamicWidth);

            displayLog(dynamicWidth);

                    String title = String.format(rowFormat, "ID", "Title", "Author");

            System.out.printf("%-" + (tableWidth - 3) + "s", title);

                    //Top-Bottom Border

            System.out.println("+"

                    + "-".repeat(tableWidth) + "+");

            library.listAllBooks(page, pageSize, dynamicWidth);

                    //Top-Bottom Border

                    System.out.println("+"

                + "-".repeat(tableWidth) + "+");

            String stats = String.format("Total Books: %d | Total Pages: %d | Current Page: %d",
totalBooks, totalPages, page + 1);

            System.out.printf("| %-" + (tableWidth - 2) + "s |\n", stats);

                    //Top-Bottom Border

            System.out.println("+"

                    + "-".repeat(tableWidth) + "+");

            System.out.printf("| %-" + (tableWidth - 2) + "s |\n", "Press 'n' for next page, 'p' for
previous page, 'q' to quit listing.");

                    //Bottom Border

            System.out.println("+"
```

```
                + "-".repeat(tableWidth) + "+");


            String command = scanner.nextLine().trim().toLowerCase();
            if (command.equals("n")) {
                page = Math.min(page + 1, totalPages - 1);
            } else if (command.equals("p")) {
                page = Math.max(0, page - 1);
            } else if (command.equals("q")) {
                break;
            } else {
                logMessage("[ERROR]", "Invalid choice. Please try again.");
            }
        }
    }


    /**
     * Imports books from a file.
     *
     * @param library The library instance.
     * @param scanner The scanner for user input.
     */
    private static void addBooksFromFile(Library library, Scanner scanner) {
        String errorSummary = library.addBooksFromFile(case_path, 1000);
        if (errorSummary.isEmpty()) {
            logMessage("[INFO]", "Books added successfully!");
        } else {
            logMessage("[ERROR]", "Books added with some errors:\n" + errorSummary);
```

```
        }
}

/**
 * Removes a book from the library by ID.
 *
 * @param library The library instance.
 * @param scanner The scanner for user input.
 */
private static void removeBookById(Library library, Scanner scanner) {
        System.out.print("Enter book ID to remove: ");
        int id = scanner.nextInt();
        scanner.nextLine();
        boolean isRemoved = library.removeBookById(id);
        if (isRemoved) {
                logMessage("[INFO]", "Book removed successfully!");
        } else {
                logMessage("[ERROR]", "Failed to remove book. ID may not exist.");
        }
}

/**
 * Exits the program.
 *
 * @param scanner The scanner for user input.
 */
private static void exit(Scanner scanner) {
```

```
                logMessage("[INFO]", "Exiting...");
                scanner.close();
            }
        }
```

Library.java:

```
import java.io.*;
import java.util.*;
import java.util.regex.Pattern;

/**Axel Diaz | CEN 3024C Software Development | - CRN: 17125
 * Main
 * The Library class represents a collection of books and provides methods
 * for managing this collection. It serves as the core data structure in a
 * Library Management System. The class allows for adding books from a file,
 * removing books by their ID, listing all books with pagination, and
 * retrieving attributes of the book collection, such as the total
 * number of books, maximum book ID, maximum title length, and maximum
 * author name length.
 */
public class Library {
        private ArrayList<Book> books;


        public Library() { books = new ArrayList<>(); }


        public void addBook(Book book) { books.add(book); }

        /**addBooksFromFile
         * Adds books to the library from a given file.
         *
```

```
 * @param filePath  The path to the file containing book information.
 * @param batchSize The number of books to add in a single batch.
 * @return A summary of any errors encountered.
 */
public String addBooksFromFile(String filePath, int batchSize) {
        File inputFile = new File(filePath);
        ArrayList<Book> bookBatch = new ArrayList<>();
        int skippedLineCount = 0;
        HashSet<Integer> uniqueBookIds = new HashSet<>();
        StringBuilder errorSummary = new StringBuilder();

        try (Scanner fileScanner = new Scanner(inputFile, "UTF-8")) {
                int currentLineNumber = 0;

                while (fileScanner.hasNextLine()) {
                        // Increment the line number for each new line read from the file
                        currentLineNumber++;

                        // Read the next line from the file and trim any leading/trailing whitespace
                        String currentLine = fileScanner.nextLine().trim();

                        // Skip empty lines
                        if (currentLine.isEmpty()) {
                                continue;
                        }

                        // Split the line into parts based on commas
```

```
                String[] bookInfo = currentLine.split(",");


                // Skip lines that don't contain exactly 3 parts (ID, title, author)
                if (bookInfo.length != 3) {
                        // System.out.println("Skipping malformed line " +
currentLineNumber + ": " + currentLine);
                        skippedLineCount++;
                        continue;
                }


                try {
                        // Parse the book ID, title, and author from the line
                        int bookId = Integer.parseInt(bookInfo[0].trim());
                        String bookTitle = bookInfo[1].trim();
                        String bookAuthor = bookInfo[2].trim();


                        // Define a pattern for valid characters in titles and authors
                        Pattern validCharsPattern = Pattern.compile("[a-zA-Z0-9 ',-]+");


                        // Skip lines with invalid characters in the title or author
                        if (!validCharsPattern.matcher(bookTitle).matches() ||
!validCharsPattern.matcher(bookAuthor).matches()) {
                                // System.out.println("Skipping line with invalid characters
at line " + currentLineNumber + ": " + currentLine);
                                skippedLineCount++;
                                continue;
                        }


                        // Skip lines with duplicate book IDs
```

```
                    if (uniqueBookIds.contains(bookId)) {
                        // System.out.println("Skipping line with duplicate ID at line " + currentLineNumber + ": " + currentLine);
                        skippedLineCount++;
                        continue;
                    }

                    // Add the book ID to the set of unique IDs
                    uniqueBookIds.add(bookId);

                    // Skip lines with empty title or author
                    if (bookTitle.isEmpty() || bookAuthor.isEmpty()) {
                        // System.out.println("Skipping line with empty title or author at line " + currentLineNumber + ": " + currentLine);
                        skippedLineCount++;
                        continue;
                    }

                    // Create a new Book object and add it to the batch
                    Book newBook = new Book(bookId, bookTitle, bookAuthor);
                    bookBatch.add(newBook);

                    // If the batch size is reached, add all books in the batch to the library and clear the batch
                    if (bookBatch.size() == batchSize) {
                        books.addAll(bookBatch);
                        bookBatch.clear();
                    }
                } catch (NumberFormatException e) {
```

```
                                        // Skip lines with invalid book IDs
                                        // System.out.println("Skipping line with invalid ID at line " +
currentLineNumber + ": " + currentLine);
                                        skippedLineCount++;
                                }
                        }

                        // Add any remaining books in the batch to the library
                        if (!bookBatch.isEmpty()) {
                                books.addAll(bookBatch);
                        }
                } catch (FileNotFoundException e) {
                        errorSummary.append("Error: File not found - " + filePath);
                } catch (SecurityException e) {
                        errorSummary.append("Error: Insufficient permissions to read the file.");
                } catch (Exception e) {
                        errorSummary.append("An unexpected error occurred: " + e.getMessage());
                }

                if (skippedLineCount > 0) {
                        errorSummary.append("Lines with errors:
").append(skippedLineCount).append("\n");
                }

                return errorSummary.toString();
        }

        /**removeBookById
```

```
 * Removes a book by its ID.
 *
 * @param id The ID of the book to remove.
 * @return {@code true} if a book was removed, {@code false} otherwise.
 */
public boolean removeBookById(int id) {
        int initialSize = books.size();          // Get the initial size of the books list
        books.removeIf(book -> book.getId() == id); // Remove the book
        int newSize = books.size();              // Get the new size of the books list

        return initialSize > newSize; // Return true if a book was removed, false otherwise
}

/**listAllBooks
 * Lists all books with pagination.
 *
 * @param page      The page number.
 * @param pageSize   The number of books to display per page.
 * @param rowFormat  The format for displaying each book row.
 */
public void listAllBooks(int page, int pageSize, int dynamicWidth) {
        int start = page * pageSize;
        int end = Math.min(start + pageSize, books.size());

        int idWidth = String.valueOf(getMaxId()).length();
        int titleWidth = getMaxTitleLength();
        int authorWidth = getMaxAuthorLength();
```

```
            // Adding 8 for the four '|' separators and spaces
            int contentWidth = idWidth + titleWidth + authorWidth + 8;
            int remainingSpace = dynamicWidth - contentWidth;

            int extraSpaceForTitle = remainingSpace / 2;
            int extraSpaceForAuthor = remainingSpace / 2;

            if (remainingSpace % 2 != 0) {
                extraSpaceForTitle += 1;
            }

            // Create a format string that respects the dynamicWidth
            String rowFormat = "| %-" + idWidth + "s | %-" + (titleWidth + extraSpaceForTitle) + "s | %-" +
(authorWidth + extraSpaceForAuthor) + "s |";

            for (int i = start; i < end; i++) {
                Book book = books.get(i);
                // Use String.format to ensure extra spaces are added at the correct positions
                String formattedTitle = String.format("%-" + (titleWidth + extraSpaceForTitle) + "s",
book.getTitle());
                String formattedAuthor = String.format("%-" + (authorWidth + extraSpaceForAuthor)
+ "s", book.getAuthor());
                System.out.printf(rowFormat, book.getId(), formattedTitle, formattedAuthor);
                System.out.println();
            }
    }
    /**getBookByIndex
     * Gets a book by its index in the ArrayList.
```

```
     *
     * @param index The index of the book.
     * @return The book, or {@code null} if the index is out of bounds.
     */
    public Book getBookByIndex(int index) {
        if (index >= 0 && index < books.size()) {
            return books.get(index);
        }
        return null; // Return null if the index is out of bounds
    }

    /**
     * Gets the total number of books in the library.
     *
     * @return The total number of books.
     */
    public int getTotalBooks() { return books.size(); }

    /**getMaxId
     * Gets the maximum book ID in the library.
     *
     * @return The maximum book ID, or 0 if the library is empty.
     */
    public int getMaxId() {
        int maxId = 0; // Initialize maxId to 0

        // Iterate through all books in the library
```

```
        for (Book book : books) {
                // Update maxId if the current book's ID is greater
                if (book.getId() > maxId) {
                        maxId = book.getId();
                }
        }

        return maxId; // Return the maximum ID found
}

/**getMaxTitleLength
 * Gets the maximum length of all book titles in the library.
 *
 * @return The maximum title length, or 20 if the library is empty.
 */
public int getMaxTitleLength() {
        // Stream through all books, get their titles, and find the maximum length
        return books.stream()
           .map(Book::getTitle)     // Extract the titles from the books
           .mapToInt(String::length) // Convert titles to their lengths
           .max()                // Find the maximum length
           .orElse(20);           // Return 20 if no maximum is found
}

/**getMaxAuthorLengt
 * Gets the maximum length of all book authors in the library.
 *
```

```
     * @return The maximum author name length, or 20 if the library is empty.
     */
    public int getMaxAuthorLength() {
        // Stream through all books, get their authors, and find the maximum length
        return books.stream()
            .map(Book::getAuthor)     // Extract the authors from the books
            .mapToInt(String::length) // Convert authors to their lengths
            .max()                    // Find the maximum length
            .orElse(20);              // Return 20 if no maximum is found
    }
}
```

Book.java:

```
import java.util.HashMap;


/**Axel Diaz | CEN 3024C Software Development | - CRN: 17125
 * Main
 * Represents a book with attributes such as ID, title, and author.
 */
public class Book {
	private int id;
	private String title;
	private String author;



	public Book(int id, String title, String author) {
		this.id = id;
		this.title = title;
		this.author = author;
	}

	/**
	 * Gets the ID of the book.
	 *
	 * @return The book's ID.
	 */
	public int getId() { return id; }

	/**
```

```
 * Gets the title of the book.
 *
 * @return The book's title.
 */
public String getTitle() { return title; }

/**
 * Gets the author of the book.
 *
 * @return The book's author.
 */
public String getAuthor() { return author; }

/**
 * Sets the title of the book.
 *
 * @param title The new title.
 */
public void setTitle(String title) { this.title = title; }

/**
 * Sets the author of the book.
 *
 * @param author The new author.
 */
public void setAuthor(String author) { this.author = author; }
```

```
/**
 * Converts the book attributes to a dictionary representation.
 *
 * @return A HashMap containing the book's attributes.
 */
public HashMap<String, String> toDict() {
        HashMap<String, String> dict = new HashMap<>();
        dict.put("id", String.valueOf(id));
        dict.put("title", title);
        dict.put("author", author);
        return dict;
}

/**
 * Provides a string representation of the book.
 *
 * @return A string containing the book's ID, title, and author.
 */
@Override
public String toString() {
        return "ID: " + id + ", Title: " + title + ", Author: " + author;
}

/**
 * Compares the equality of this book with another based on their IDs.
 *
 * @param other The other book to compare with.
```

```
     * @return True if the IDs are equal, otherwise false.
     */
    public boolean equals(Book other) { return this.id == other.id; }
}
```