LibrarySystem

A Java-Based Library Management System

Rahima Rahman Promy
Student ID: 2024-3-60-776

Course ID: CSE110

Course Name: Object Oriented Programming

June 1, 2025

Abstract

This report presents the design and implementation of `LibrarySystem`, a Java console application that simulates a basic library management system. The system reads book, student, and teacher data from Excel files, allows authenticated users to borrow and return books, and demonstrates core object-oriented principles, custom exception handling, and Excel file integration via Apache POI. Example data used for validation includes the following Excel contents:

books.xlsx

book_id	title	author	copies
301	Intro to Java	James	5
302	Python Basics	Mark	3
303	Data Structures	Alice	6
304	Operating Systems	Tanmay	2
305	DBMS	Robert	4
306	Web Dev with JS	Nina	3
307	AI Fundamentals	Luna	5
308	ML with Python	Sam	2
309	Networks	Rashid	6
310	Software Eng	Joy	4

teachers.xlsx

id	name	designation	total
2001	Dr. Karim	Professor	0
2002	Ms. Nila	Lecturer	0
2003	Mr. Zahid	Asst. Professor	0
2004	Dr. Rumi	Professor	0
2005	Ms. Farah	Lecturer	0
2006	Mr. Anis	Asst. Professor	0
2007	Dr. Lima	Professor	0
2008	Mr. Tushar	Lecturer	0
2009	Ms. Saba	Lecturer	0
2010	Dr. Imran	Professor	0

students.xlsx

student_id	name	department	total_book
1001	Anika	CSE	0
1002	Rafi	EEE	0
1003	Mou	BBA	0
1004	Tariq	CSE	0
1005	Sadia	ENG	0
1006	Riyan	EEE	0
1007	Lamia	CSE	0
1008	Tanvir	BBA	0
1009	Elita	ENG	0
1010	Hasib	CSE	0

Contents

1 Introduction

The `LibrarySystem` project aims to demonstrate fundamental Java programming skills through the development of a console-based library management application. By integrating Excel file input/output, custom exception handling, and object-oriented design, this system simulates a real-world scenario where students and teachers authenticate, browse available books, borrow and return copies, and track their own borrowed items. Apache POI is leveraged for reading and writing Excel spreadsheets, ensuring data persistence. The subsequent sections describe each core component in detail, focusing on class design, interface usage, exception hierarchies, and Excel file handling.

2 Class Design

2.1 Book

The `Book` class represents a single book entity within the library. Each `Book` instance contains four fields: `bookId` (a unique string identifier), `title` (the book's title), `author` (the book's author), and `availableCopies` (an integer tracking how many copies remain available). Two constructors are provided: a no-argument default constructor that initializes a placeholder book with zero copies, and a parameterized constructor that accepts all four properties. The `displayInfo()` method prints the book's details (ID, title, author) along with the current number of copies remaining. To support borrowing functionality, `borrowCopy()` decrements `availableCopies` if at least one copy remains; otherwise, it returns `false` to indicate unavailability. The `returnCopy()` method increments the count of available copies. Accessor methods (`getBookId()`, `getTitle()`, `getAuthor()`, `getAvailableCopies()`) allow other classes to retrieve book properties without modifying them directly. By encapsulating all book-related data and behaviors within this class, the system follows the single-responsibility principle, isolating data representation from higher-level logic.

2.2 User Abstract Class

The `User` class is declared as `abstract` and implements the `Borrowable` interface. It encapsulates common properties and methods shared by all concrete user types. Key fields include `userId` (a unique string identifying the user), `name` (the user's full name), and `borrowedBooks` (an `ArrayList<Book>` that tracks which books the user currently has borrowed). The constructor accepts `userId` and `name` and initializes the `borrowedBooks` list as empty. Because each user type has a different borrowing policy, the abstract method `borrowBook(Book book)` is declared without implementation. Common functionality such

as `returnBook(Book book)` is provided in the abstract class: it checks whether the given `Book` exists in `borrowedBooks`, removes it if present, calls `book.returnCopy()`, and displays a confirmation message. If the user had not borrowed that book, a message indicates the mismatch. The `viewBorrowedBooks()` method iterates over `borrowedBooks`, invoking `displayInfo()` on each `Book`, or prints a notice if no books are borrowed. By centralizing these shared behaviors, subclasses need only implement borrowing logic, promoting code reuse and reducing duplication.

2.3 Student and Teacher Subclasses

Both `Student` and `Teacher` extend `User` and provide specific implementations of `borrowBook(Book book)` to enforce distinct borrowing limits.

Student The `Student` class includes a private field `department` (e.g., CSE, EEE, BBA) and a static constant `MAX_BORROW = 3`. In the constructor, `id`, `name`, and `dept` are passed to the `User` superconstructor; `department` is stored for reference. In `borrowBook(Book book)`, if the `borrowedBooks` list already contains three or more books, a `MaxBorrowLimitReachedException` is thrown with a descriptive message. Otherwise, `book.borrowCopy()` is invoked. If `borrowCopy()` returns `false`, indicating zero available copies, a `BookNotAvailableException` is thrown. Otherwise, the book is added to `borrowedBooks` and a confirmation message is printed. By isolating student-specific borrowing rules here, the code remains clear and flexible: changing the maximum allowed to borrow only requires updating the `MAX_BORROW` constant.

Teacher Similarly, the `Teacher` class adds a private field `designation` (e.g., Professor, Lecturer) and defines `MAX_BORROW = 5`. The constructor passes `id`, `name`, and `desig` to the parent constructor and stores `designation`. The `borrowBook(Book book)` method follows the same pattern as `Student` but checks the teacher's borrowed list against `MAX_BORROW = 5`. If the limit is exceeded, a `MaxBorrowLimitReachedException` is thrown. Otherwise, `book.borrowCopy()` is attempted; if unsuccessful, a `BookNotAvailableException` is thrown. On success, the book is added to the teacher's `borrowedBooks` list, and an appropriate message is displayed. This separation of roles ensures that future user types with different borrowing policies can be implemented without affecting existing code.

2.4 Borrowable Interface

The `Borrowable` interface declares a single method:

```
void borrowBook(Book book) throws LibraryException;
```

By requiring every user class that implements `Borrowable` to provide its own borrowing implementation, the system enforces a contract: any concrete user type must handle borrowing logic and propagate any `LibraryException` that may occur. This design encourages polymorphism; methods can accept a `Borrowable` reference and invoke `borrowBook(...)` without knowing the exact type (student or teacher), yet still catch exceptions generically as `LibraryException`.

2.5 Library

The `Library` class encapsulates all book collection logic and handles Excel integration for persistent storage. Internally, it maintains a private `List<Book> bookList`. The `loadBooksFromExcel(Str filePath)` method uses Apache POI to open the specified `.xlsx` file (e.g., `books.xlsx`) via a `FileInputStream` and an `XSSFWorkbook`. It retrieves the first `Sheet` and iterates over its `Row` objects, skipping the header row. For each data row, it reads four cells: `bookId`, `title`, `author`, and `copies`. The helper method `getStringValue(Cell cell)` returns a string representation for any cell type, ensuring numeric and boolean cells are converted properly. After parsing, a new `Book` object is constructed and appended to `bookList`. If any I/O errors occur (e.g., file not found), the method propagates an `IOException` to inform the caller.

To write updates back to Excel, `saveBooksToExcel()` creates a new `XSSFWorkbook`, builds a sheet named `"Books"`, and populates a header row followed by one row per `Book` in `bookList`. Each `Book`'s properties are written into the appropriate cells. Finally, a `FileOutputStream` to `bookFilePath` is opened, and `wb.write(...)` persists changes. If `bookFilePath` is `null`, the method exits immediately. The methods `getBookById(String id)` (which searches `bookList` by `bookId`) and `displayAvailableBooks()` (which iterates over all `Book` instances and calls `displayInfo()`) enable higher-level functionality in `Main` and user classes to retrieve and list books.

2.6 UserManager

The `UserManager` class handles loading and authenticating both students and teachers from separate Excel files. It maintains a private `List<User> userList` that stores mixed `Student` and `Teacher` objects. The method `loadUsersFromExcel(String filePath, String userType)` again uses Apache POI to read an `.xlsx` file (either `students.xlsx` or `teachers.xlsx`). It skips the header row, extracts the first three cells: `id`, `name`, and `third` (which corresponds to either department or designation). Depending on the value of `userType` (case-insensitive), it instantiates either a `Student` or a `Teacher` and adds it to `userList`. This dynamic approach allows a single method to support multiple user types. If any `IOException` arises during file

reading, the exception propagates upward to be handled in `Main`.

The `authenticateUser(String id)` method filters `userList` by matching `getUserId()` against the provided ID. If a match is found, the corresponding `User` is returned. Otherwise, a `UserNotFoundException` (a subclass of `LibraryException`) is thrown, prompting the caller to retry or exit. By centralizing authentication logic here, `Main` only needs to catch `UserNotFoundException` and present a friendly message, without duplicating lookup code.

3 Exception Handling

Custom exception classes allow the application to distinguish different error conditions and present meaningful feedback to the user. All custom exceptions extend `LibraryException`, which itself extends `Exception`. This design ensures that all user-facing exceptions share a common parent, simplifying `catch` clauses when multiple conditions might apply.

LibraryException Defined as:

```
public class LibraryException extends Exception {
    public LibraryException(String message) {
        super(message);
    }
}
```

Because many library operations can fail (e.g., invalid user ID, exceeding borrow limit, no copies available), `LibraryException` serves as a generic base. By requiring all subclasses to pass a descriptive message to its constructor, the code ensures that when an exception is thrown, the message accurately describes the failure to the end user.

UserNotFoundException Thrown when `UserManager.authenticateUser(...)` fails to locate a matching ID. Definition:

```
public class UserNotFoundException extends LibraryException {
    public UserNotFoundException(String message) {
        super(message);
    }
}
```

In `Main.main(...)` the call to `authenticateUser(uid)` is wrapped in a `try/catch`. If caught, the exception's message (`"User not found."`) is printed, prompting the user to re-enter their ID. This flow prevents the application from crashing and gives clear instructions.

BookNotAvailableException Thrown when a user attempts to borrow a book with zero remaining copies. Definition:

```
public class BookNotAvailableException extends LibraryException {
    public BookNotAvailableException(String message) {
        super(message);
    }
}
```

Within `Student.borrowBook(...)` and `Teacher.borrowBook(...)`, after checking the borrower's limit, the call to `book.borrowCopy()` returns `false` if `availableCopies == 0`. In that case, `new BookNotAvailableException("No copies left.")` is thrown, which is then caught in `Main.handleBorrowOrReturn(...)`. A concise message `"No copies left."` informs the user exactly why the operation failed.

MaxBorrowLimitReachedException Thrown when a user has already reached the maximum number of borrowed books permitted by their role. Definition:

```
public class MaxBorrowLimitReachedException extends LibraryException {
    public MaxBorrowLimitReachedException(String message) {
        super(message);
    }
}
```

In `Student.borrowBook(...)`, if `borrowedBooks.size() >= MAX_BORROW`, `new MaxBorrowLimitReachedE` `borrow limit reached (3).")` is thrown. Similarly, in `Teacher.borrowBook(...)`, the limit is 5. When caught in `Main.handleBorrowOrReturn(...)`, the message is printed, e.g., `"Student borrow limit reached (3)."`, ensuring the user knows exactly why the borrow request was denied.

4 Excel File Integration

Excel file integration is achieved exclusively through the Apache POI library (`org.apache.poi.ss.usermod` and `org.apache.poi.xssf.usermodel.XSSFWorkbook`). Two main classes—`Library` and `UserManager`—leverage identical patterns to read spreadsheet data. The steps for reading are as follows:

1. **Open FileInputStream:** A `FileInputStream fis = new FileInputStream(filePath);` is passed to `new XSSFWorkbook(fis)` to create an in-memory `Workbook`.

2. **Select Sheet:** `Sheet sheet = wb.getSheetAt(0);` obtains the first (and only) sheet where data resides.
3. **Skip Header Row:** A boolean flag `header=true` skips the first row of column labels.
4. **Iterate Rows:** For each subsequent `Row row : sheet`, cell values are extracted using the helper method `getStringValue(Cell cell)` (which handles different cell types including `STRING`, `NUMERIC`, `BOOLEAN`, and `FORMULA`).
5. **Parse Fields:** For `books.xlsx`, four cells are read: `id = getStringValue(cell0); title = getStringValue(cell1); author = getStringValue(cell2); copies = (int) cell3.getNumericCellValue();`. For `students.xlsx` and `teachers.xlsx`, three cells are parsed: `id`, `name`, and `third` (department or designation).
6. **Instantiate Objects:** New `Book`, `Student`, or `Teacher` objects are created and appended to the appropriate list (`bookList` or `userList`).
7. **Close Workbook:** The `try-with-resources` block automatically closes both `Workbook` and `FileInputStream`.

In addition to reading, the `Library.saveBooksToExcel()` method demonstrates how to write updated book information back to the same Excel file:

1. **Create New Workbook:** `Workbook wb = new XSSFWorkbook();`
2. **Create Sheet:** `Sheet sheet = wb.createSheet("Books");`
3. **Write Header Row:** Row 0 is created and cells are labeled `"Book ID"`, `"Title"`, `"Author"`, and `"Copies"`.
4. **Write Data Rows:** For each `Book b : bookList`, a new `Row` is created and `b.getBookId()`, `b.getTitle()`, `b.getAuthor()`, and `b.getAvailableCopies()` are written to cells 0–3, respectively.
5. **Persist to File:** A `FileOutputStream fos = new FileOutputStream(bookFilePath);` opens the original file path, and `wb.write(fos);` writes the in-memory workbook to disk. `fos` and `wb` are closed automatically by the `try-with-resources` block.

By encapsulating all Excel interaction within these methods, the application cleanly separates data I/O from business logic. If a new file format or database back end is required in the future, only these methods would need adjustment.

5 Main Application Flow (Main.java)

The `Main` class ties together all components and orchestrates user interaction. The `main(String[] args)` method proceeds as follows:

1. **Initialize Scanner and Classes:** A `Scanner sc` reads console input. New instances of `Library lib` and `UserManager um` are created.
2. **Display Commons IO Location (Informational):** The line

```
System.out.println("Commons IO loaded from: " +
    org.apache.commons.io.input.BoundedInputStream.class
        .getProtectionDomain()
        .getCodeSource()
        .getLocation());
```

   confirms at runtime that Apache Commons IO (a transitive dependency of POI) is correctly on the classpath.
3. **Load Data Files:** A `try/catch` block calls `lib.loadBooksFromExcel("books.xlsx");`, `um.loadUsersFromExcel("students.xlsx", "student");`, and `um.loadUsersFromExcel("teache` `"teacher");`. If any `IOException` occurs (e.g., missing file), a message is printed: `" Couldn't read Excel files: <error>"` and the program returns, terminating early.
4. **User Authentication Loop:** A `while(true)` loop prompts: `"Enter your user ID: "`. The input `String uid = sc.nextLine().trim();` is passed to `um.authenticateUser(uid);`. If no matching user is found, a `UserNotFoundException` is caught, `e.getMessage()` (Üser not found.̈) is displayed, and the loop repeats. Upon successful authentication, the `User current` is set and the loop breaks.
5. **Main Menu Loop:** Another `while(true)` loop displays options:

```
1) List Books   2) Borrow   3) Return   4) My Books   5) Exit
```

   The user's choice is read as `String choice = sc.nextLine().trim();`. A `switch(choice)` executes one of five branches:

   - `"1"`: Calls `lib.displayAvailableBooks();`, which prints all books and their remaining copies.
   - `"2"` (Borrow): Prompts `"Book ID to borrow: "`, reads an ID, and calls `handleBorrowOrRetu`

`current, lib, true);` which (a) looks up the book via `lib.getBookById(id)`, (b) if null prints `"Book not found."`, else calls `current.borrowBook(book);` inside a `try/catch` to handle `LibraryException`. On success, `lib.saveBooksToExcel();` updates the spreadsheet. Any caught exceptions print their message.

- `"3"` (Return): Prompts `"Book ID to return: "`, and calls `handleBorrowOrReturn(id, current, lib, false);` which looks up the book, then calls `current.returnBook(book);` and updates Excel similarly. If the book was not borrowed by the user, a message is printed: `"You didn't borrow that book."`
- `"4"` (My Books): Calls `current.viewBorrowedBooks();`, printing the user's currently borrowed items or `"No books borrowed."` if none.
- `"5"` (Exit): Attempts to save the latest book data via `lib.saveBooksToExcel();`, then prints `"Goodbye!"`, closes the `Scanner`, and returns from `main`, terminating the program.
- `default`: Prints `"Invalid choice."` if an unrecognized option is entered.

The private helper method `handleBorrowOrReturn(...)` reduces duplication by centralizing the lookup of `Book book = lib.getBookById(id);` and catching both `LibraryException` and `IOException` from `current.borrowBook(...)` or `current.returnBook(...)` and `lib.saveBooksToExcel()`. By using this structure, `Main` remains succinct and focused on user interaction rather than detail logic.

6 Summary of Core Concepts

6.1 Object-Oriented Design

Encapsulation: Each entity (Book, User, Student, Teacher, Library, UserManager) encapsulates its own data and behaviors. Fields are marked `private` or `protected`, and public methods or constructors provide controlled access.

Inheritance & Polymorphism: `Student` and `Teacher` inherit from the abstract `User` class and override the `borrowBook(...)` method to implement role-specific logic. The `Borrowable` interface ensures that any new user type can be treated polymorphically when invoking `borrowBook(...)`.

Abstraction: The `User` class defines common methods for all users (returning and viewing borrowed books), while deferring borrowing specifics to subclasses. Similarly, `Library` abstracts away the details of Excel file handling from the rest of the application.

6.2 Exception Handling

Custom Exceptions: `LibraryException` serves as a base type. `UserNotFoundException`, `BookNotAvailableException`, and `MaxBorrowLimitReachedException` extend it, providing precise error messages that reflect different failure conditions. Methods that can fail declare `throws LibraryException`, forcing callers to either catch and handle them (as in `Main`) or propagate them further, ensuring no exception is ignored.

Try-With-Resources: Both Excel reading and writing use `try (FileInputStream fis = new FileInputStream(filePath); Workbook wb = new XSSFWorkbook(fis))` or `try (Workbook wb = new XSSFWorkbook(); FileOutputStream fos = new FileOutputStream(bookFilePath))`. This pattern automatically closes streams and workbooks, preventing resource leaks.

6.3 Excel Reading and Writing

Apache POI Integration: By leveraging Apache POI, the system can read and write `.xlsx` files without requiring manual CSV conversion. The `getStringValue(Cell cell)` helper method supports multiple cell types (STRING, NUMERIC, BOOLEAN, FORMULA), guaranteeing that data is read correctly regardless of formatting.

Data Persistence: Changes to book availability (after borrowing or returning) are immediately written back to `books.xlsx` via `Library.saveBooksToExcel()`. In this way, if the application is restarted, the latest book copy counts persist across sessions.

7 Conclusion

The `LibrarySystem` project successfully demonstrates a modular, robust Java application that integrates core object-oriented principles, custom exception hierarchies, and Excel file I/O via Apache POI. By dividing functionality into cohesive classes—`Book`, `User` (with `Student` and `Teacher` subclasses), `Library`, and `UserManager`—the system achieves clarity, extensibility, and maintainability. Custom exceptions ensure that error conditions (such as invalid user IDs, unavailable books, or exceeded borrowing limits) are communicated to users in a controlled manner without crashing the application. Excel integration provides a familiar, tabular interface for librarians to update book inventories and user lists without modifying code. Future enhancements might include a graphical user interface, database back-end support, or role-based reporting. Overall, this project effectively demonstrates how Java's rich ecosystem can be leveraged to implement a simple yet extensible library management system.