Project Report

Library Management System

OOP1 Project 2025-2026

A00325358: Pronoti Saha
11-22-2025

Contents

Introduction

The ***Library Management System*** (***LMS***) is a Java-based application designed to manage library resources such as books and magazines. It provides a menu-driven interface for users to view, search, borrow, and return library items.

The project demonstrates modern Java (Java 25) features and adheres to object-oriented programming (OOP) principles.

Purpose

- Demonstrate OOP concepts in a real-world scenario.
- Provide a modular, extensible, and maintainable codebase for educational use.
- Utilize Java 25 features such as compact source files, records, sealed interfaces, and flexible constructors.

Features

- View all library items (books and magazines)
- Search for books by title or author
- Borrow and return books
- View library statistics
- Exception handling with custom exceptions

User Stories

1. View All Items

User Story:

As a user, I want to see all books and magazines in the library so I can choose what interests me.

Acceptance Criteria:

- The system displays a list of all books with their status (available/borrowed).
- The system displays a list of all magazines.
- The output is clear and well-formatted.

2. Search Books by Title

User Story:

As a user, I want to search for books by title to find specific items quickly.

Acceptance Criteria:

- The user can enter a title.
- The system displays all matching books.
- If no matches are found, an appropriate message is shown.

3. Search Books by Author

User Story:

As a user, I want to search for books by author to find specific items quickly.

Acceptance Criteria:

- The user can enter a author name.
- The system displays all matching books.
- If no matches are found, an appropriate message is shown.

4. Borrow a Book

User Story:

As a user, I want to borrow a book by providing its ID and my name so I can read it at home.

Acceptance Criteria:

- The user provides a valid book ID and their name.
- The system marks the book as borrowed if available.
- If the book is already borrowed or the ID is invalid, an error message is shown.

5. Return a Book

User Story:

As a user, I want to return a borrowed book by providing its ID so it becomes available for others.

Acceptance Criteria:

- The user provides a valid book ID.
- The system marks the book as available if it was borrowed.
- If the book was not borrowed or the ID is invalid, an error message is shown.

6. View Library Statistics

User Story:

As a user, I want to see statistics about the library's collection and usage.

Acceptance Criteria:

- The system displays the total number of books.
- The system displays the number of borrowed and available books.
- The system displays the number of magazines.

7. Menu-based Operations

User Story:

As a user, I want to perform various operations using a interactive menu.

Acceptance Criteria:

- The system displays a menu for various operations.
- The system performs the operation based on user input.
- The system displays appropriate error message for invalid input

Implementation

Fundamentals

Classes

this() and this

In the immutable Magazine class, ***this()*** *– to call it own constructors* and ***this*** *– to refer to the instance properties* are used to instantiate a Magazine object.

```java
/**
 * Magazine is an immutable library item
 * @author A00325358 Pronoti Saha
 */
public final class Magazine implements LibraryItem {

    private final String title;
    private final Map<String, String> metaData;
    private final String[] optionalParams;

    /**
     * Magazine Constructor
     * @param title Title of the Magazine
     */
    public Magazine(String title) {
        this(title, Collections.emptyMap());
    }

    /**
     * Magazine Constructor
     * @param title Title of the Magazine
     * @param metaData Metadata
     */
    public Magazine(String title, Map<String, String> metaData) {
        this(title, metaData, new String[]{});
    }

    /**
     * Magazine Constructor
     * @param title Title of the Magazine
     * @param metaData Metadata
     * @param optionalParams Varargs
     */
    public Magazine(String title, Map<String, String> metaData, String ...optionalParams) {
        this.title = title;
        this.metaData = Optional.ofNullable(metaData).orElse(Collections.emptyMap());
        this.optionalParams = optionalParams;
    }
```

Method/Constructor overloading

Magazine class has 3 overloaded constructors

```
/**
 * Magazine Constructor
 * @param title Title of the Magazine
 */
public Magazine(String title) {
    this(title, Collections.emptyMap());
}

/**
 * Magazine Constructor
 * @param title Title of the Magazine
 * @param metaData Metadata
 */
public Magazine(String title, Map<String, String> metaData) {
    this(title, metaData, new String[]{});
}

/**
 * Magazine Constructor
 * @param title Title of the Magazine
 * @param metaData Metadata
 * @param optionalParams Varargs
 */
public Magazine(String title, Map<String, String> metaData, String ...optionalParams) {
    this.title = title;
    this.metaData = Optional.ofNullable(metaData).orElse(Collections.emptyMap());
    this.optionalParams = optionalParams;
}
```

varargs

One of the constructors of the Magazine class takes in varargs optionalParams as an argument

```
/**
 * Magazine Constructor
 * @param title Title of the Magazine
 * @param metaData Metadata
 * @param optionalParams Varargs
 */
public Magazine(String title, Map<String, String> metaData, String ...optionalParams) {
    this.title = title;
    this.metaData = Optional.ofNullable(metaData).orElse(Collections.emptyMap());
    this.optionalParams = optionalParams;
}
```

LVTI

```java
/**
 * Search a book using a filter predicate
 * @param content
 * @param filter
 */
private void search(String content, Predicate<Book> filter) {
    // LVTI
    var matches = books.stream().filter(filter).toList();
    if (matches.isEmpty()) {
        IO.println(new LibraryItemNotFoundException("No book found for: " + content));
    } else {
        // Java Lambda method reference
        matches.forEach(Book::info);
    }
}
```

Encapsulation

The ***Library*** class encapsulates ***Books***, ***Magazines*** and ***LibraryRecords*** in it

```java
package com.library;

import com.library.exception.LibraryItemNotFoundException;
import com.library.items.Book;
import com.library.items.Magazine;
import com.library.records.LibraryRecord;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;
import java.util.function.Supplier;

/**
 * library class
 * @author A00325358 Pronoti Saha
 */
public class Library{

    private final List<Book> books = new ArrayList<>();
    private final List<Magazine> magazines = new ArrayList<>();
    private final List<LibraryRecord> records = new ArrayList<>();

    /**
     * No argument constructor
     */
    public Library(){
        super();
    }
```

Interfaces

There are two main interfaces in the project - ***LibraryItem*** and ***Borrowable***

LibraryItem

```java
package com.library.items;

/**
 * LibraryItem is a sealed interface
 * Permits Book and Magazine
 * @author A00325358 Pronoti Saha
 */
public sealed interface LibraryItem permits Book, Magazine {
    /**
     * Library item info
     */
    void info();
}
```

Borrowable

```java
package com.library.items;

/**
 * Borrowable interface
 * @author A00325358 Pronoti Saha
 */
public interface Borrowable {

    /**
     * Borrow item
     * @param studentName Name of the student
     */
    void borrowItem(String studentName);

    /**
     * Return item
     */
    void returnItem();

    /**
     * Default method
     */
    default void greet() {
        System.out.println(x: "Welcome to the Library!");
        getName();
    }

    /**
     * Static methods
     */
    static void libraryInfo() {
        System.out.println(x: "Library System 2025");
    }

    private void getName(){
        System.out.println(x: "Borrowable");
    }
}
```

Inheritance

Overriding and Polymorphism

When a subclass provides a specific implementation for a method that is already defined in its parent class, it is called ***method overriding***. The overridden method in the subclass must have the same name, parameters, and return type as the method in the parent class.

Polymorphism is one of the core concepts in Object Oriented Programming that allows objects to behave differently based on their specific class type. In Java, polymorphism allows the same method or object to behave differently based on the context, especially on the project's actual runtime class.

LibraryItem is a sealed interface for library items (***Books*** and ***Magazines***) (***Appendix 1***: Class UML Diagram).

```java
package com.library.items;

/**
 * LibraryItem is a sealed interface
 * Permits Book and Magazine
 * @author A00325358 Pronoti Saha
 */
public sealed interface LibraryItem permits Book, Magazine {
    /**
     * Library item info
     */
    void info();
}
```

Contrast super() and super.

The ***super*** keyword in Java is a reference variable that is used to refer parent class objects. super can be used to call parent class' variables and methods.

The ***super()*** in Java is a reference variable that is used to refer parent class constructors. ***super()*** can be used to call parent class' constructors only.

```java
package com.library;

import com.library.exception.LibraryItemNotFoundException;
import com.library.items.Book;
import com.library.items.Magazine;
import com.library.records.LibraryRecord;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Predicate;
import java.util.function.Supplier;

/**
 * library class
 * @author A00325358 Pronoti Saha
 */
public class Library{

    private final List<Book> books = new ArrayList<>();
    private final List<Magazine> magazines = new ArrayList<>();
    private final List<LibraryRecord> records = new ArrayList<>();

    /**
     * No argument constructor
     */
    public Library(){
        super();
    }
```

Exceptions (checked and unchecked)

LibraryException (Checked Exception)

```java
/**
 * Checked LibraryException
 * @author A00325358 Pronoti Saha
 */
public class LibraryException extends Exception {

    /**
     * Constructor
     * @param message Error message
     */
    public LibraryException(String message) {
        super(message);
    }
}
```

LibraryOperationException (Unchecked Exception)

```java
/**
 * Unchecked Library Operation Exception
 * @author A00325358 Pronoti Saha
 */
public sealed class LibraryOperationException extends RuntimeException permits LibraryItemNotFoundException {

    /**
     * Constructor
     * @param message Error Message
     */
    public LibraryOperationException(String message) {
        super(message);
    }
}
```

LibraryItemNotFoundException (Unchecked Exception)

```java
/**
 * Item Not Found Exception
 * @author A00325358 Pronoti Saha
 */
public non-sealed class LibraryItemNotFoundException extends LibraryOperationException{
    /**
     * Constructor
     * @param message Error Message
     */
    public LibraryItemNotFoundException(String message) {
        super(message);
    }
}
```

Enums

```java
/**
 * Status of a Borrowable Item
 * @author A00325358 Pronoti Saha
 */
public enum BorrowableItemStatus {  Pronoti Saha *
    /**
     * Status - Available
     */
    AVAILABLE( value: "Available"),
    /**
     * Status - Borrowed
     */
    BORROWED( value: "Borrowed"),
    /**
     * Status - Reserved
     */
    RESERVED( value: "Reserved");

    /**
     * Status value
     */
    public final String value;

    /**
     * @param value Status
     */
    private BorrowableItemStatus(String value) { this.value = value; }
}
```

Use of Java Core API (String, StringBuilder, List, Map, Date API)

String Builder is used to generate receipt for a borrowed item

```java
/**
 * Borrow a book for a Student
 * @param studentName Name of the Student
 */
@Override
public void borrowItem(String studentName) {
    if (!isBorrowed) {
        isBorrowed = true;
        dueDate = LocalDate.now().plusDays(daysToAdd: 7); // 7-day loan
        StringBuilder receipt = new StringBuilder();
        receipt.append(str: "\n*** RECEIPT ***\n");
        receipt.append(str: "Student: ").append(studentName).append(str: "\n");
        receipt.append(str: "Book: ").append(title).append(str: "\n");
        receipt.append(str: "Due Date: ").append(dueDate).append(str: "\n");
        receipt.append(str: "*****************\n");
        System.out.println(receipt);
    } else {
        System.out.println(new LibraryException("Book id: "+this.id+" is already borrowed."));
    }
}
```

Date API is used to calculate late fine while returning a borrowed item

```
/**
 * Return a book
 */
@Override
public void returnItem() {
    if (!isBorrowed) {
        System.out.println(x: "Book was not borrowed.");
        return;
    }
    LocalDate today = LocalDate.now();
    long lateDays = 0;
    if (today.isAfter(dueDate)) {
        lateDays = ChronoUnit.DAYS.between(dueDate, today);
    }
    isBorrowed = false;
    System.out.println(x: "\n*** RETURN RECEIPT ***");
    System.out.println("Book Returned: " + title);
    System.out.println("Due Date: " + dueDate);
    System.out.println("Returned On: " + today);
    if (lateDays > 0) {
        System.out.println("Late by: " + lateDays + " days");
        System.out.println("Fine: €" + (lateDays * 1));
    } else {
        System.out.println(x: "Returned on time. No fine.");
    }
    System.out.println(x: "**********************\n");
}
```

Advanced:

Call-by-value and Defensive copying

In ***Call-by-Value***, method parameters values are copied to another variable and then the copied object is passed, that's why it's called pass by value where actual value does not change but the changes are done on the copied value.

If the mutable object field's state should be changed only by the native class, then a *defensive copy* of the mutable object *must* be made any time it's passed into (constructors and set methods) or out of (get methods) the class. If this is *not* done, then it is simple for the caller to break encapsulation, by changing the state of an object which is simultaneously visible to *both* the class and its caller.

Example

Magazine has a mutable object field ***metaData***, which is defensively copied in ***getMetaData()*** method. ***Magazine*** represents an immutable class and has no ***setter*** methods for its fields. Note that if the defensive copy of ***metaData*** is not made, then ***Magazine*** is no longer immutable!

```java
/**
 * Magazine is an immutable library item
 * @author A00325358 Pronoti Saha
 */
public final class Magazine implements LibraryItem {

    private final String title;
    private final Map<String, String> metaData;
    private final String[] optionalParams;

    /**...
    public Magazine(String title) {...

    /**...
    public Magazine(String title, Map<String, String> metaData) {...

    /**...
    public Magazine(String title, Map<String, String> metaData, String ...optionalParams) {...

    /**...
    public String getTitle() {...

    /**
     * Get Metadata
     * @return metadata
     */
    public Map<String, String> getMetadata() {
        return new HashMap<>(metaData);
    }

    /**...
    public String[] getOptionalParams() {...

    /**...
    @Override
    public void info() {...
}
```

private, default and static interface methods

```java
package com.library.items;

/**
 * Borrowable interface
 * @author A00325358 Pronoti Saha
 */
public interface Borrowable {

    /**
     * Borrow item
     * @param studentName Name of the student
     */
    void borrowItem(String studentName);

    /**
     * Return item
     */
    void returnItem();

    /**
     * Default method
     */
    default void greet() {
        System.out.println(x: "Welcome to the Library!");
        getName();
    }

    /**
     * Static methods
     */
    static void libraryInfo() {
        System.out.println(x: "Library System 2025");
    }

    private void getName(){
        System.out.println(x: "Borrowable");
    }
}
```

Custom Immutable type

Magazine is a custom immutable type implementing ***LibraryItem*** interface. It is declared a ***final*** class to prevent the class from being extended and no access to its properties directly or via any method that can modify the values after initialisation through constructor call.

```java
package com.library.items;

import java.util.*;

/**
 * Magazine is an immutable library item
 * @author A00325358 Pronoti Saha
 */
public final class Magazine implements LibraryItem {

    private final String title;
    private final Map<String, String> metaData;
    private final String[] optionalParams;

    /**
     * Magazine Constructor
     * @param title Title of the Magazine
     */
    public Magazine(String title) {…

    /**
     * Magazine Constructor
     * @param title Title of the Magazine
     * @param metaData Metadata
     */
    public Magazine(String title, Map<String, String> metaData) {…

    /**
     * Magazine Constructor
     * @param title Title of the Magazine
     * @param metaData Metadata
     * @param optionalParams Varargs
     */
    public Magazine(String title, Map<String, String> metaData, String ...optionalParams) {…

    /**…
    public String getTitle() {
        return title;
    }

    /**
     * Get Metadata
     * @return metadata
     */
    public Map<String, String> getMetadata() {…

    /**
     * Get Optional Parameters
     * @return Otional Parameters
     */
    public String[] getOptionalParams() {…

    /**
     * Get info
     */
    @Override
    public void info() {
        System.out.println("Magazine: " + title + ", Metadata: " + metaData);
    }
}
```

Lambdas

Discussion of 'final' or 'effectively final'

A final variable is a variable that is declared with a keyword - '***final***'.

Eg – *public static final int NUMBER=10*;

The local variables that a lambda expression may use are referred to as "effectively final".

An effectively final variable is one whose value doesn't change after it is first assigned. There is no need to explicitly declare such a variable as final, although doing so would not be an error.

Predicate

```
/**
 * Search book by title using Predicate
 * @param title Title
 */
public void searchByTitle(String title) {
    Predicate<Book> titleFilter = b -> b.getTitle().toLowerCase().contains(title.toLowerCase());
    search(title, titleFilter);
}

/**
 * Search book by author
 * @param author Author
 */
public void searchByAuthor(String author) {
    Predicate<Book> authorFilter = b -> b.getAuthor().toLowerCase().contains(author.toLowerCase());
    search(author, authorFilter);
}

/**
 * Search a book using a filter predicate
 * @param content
 * @param filter
 */
private void search(String content, Predicate<Book> filter) {
    // LVTI
    var matches = books.stream().filter(filter).toList();
    if (matches.isEmpty()) {
        IO.println(new LibraryItemNotFoundException("No book found for: " + content));
    } else {
        // Java Lambda method reference
        matches.forEach(Book::info);
    }
}
```

Consumer

```
/**
 * Return the book and update records
 * @param id
 * @return
 */
private Consumer<Book> updateRecords(int id) {
    return book -> {
        book.returnItem();
        records.removeIf(record -> record.id() == id);
    };
}
```

Supplier

```java
/**
 * Library Record supplier
 * @param book
 * @return
 */
private Supplier<LibraryRecord> createRecord(Book book) {
    return () -> new LibraryRecord(book.getId(), book.getTitle(), book.getAuthor(), LocalDate.now(), LocalDate.now().plusDays(daysToAdd: 7));
}
```

Stream

```java
/**
 * Borrow book by Id and Borrower Name
 * @param id Book Id
 * @param studentName Student Name
 */
public void borrowBook(int id, String studentName) {
    // Java Stream
    books.stream()
            .filter(b -> b.getId() == id)
            .findFirst()
            .ifPresentOrElse(b -> {
                b.borrowItem(studentName);
                records.add(new LibraryRecord(b.getId(), b.getTitle(), b.getAuthor(), LocalDate.now(), LocalDate.now().plusDays(daysToAdd: 7)));
            }, () -> System.out.println(x: "Book not found!"));
}

/**
 * Return borrowed book by Id
 * @param id Book Id
 */
public void returnBook(int id) {
    // Java Stream
    books.stream()
            .filter(b -> b.getId() == id)
            .findFirst()
            .ifPresentOrElse(updateRecords(id), () -> System.out.println(x: "Book not found!"));
}

/**
 * Return the book and update records
 * @param id
 * @return
 */
private Consumer<Book> updateRecords(int id) {
    return book -> {
        book.returnItem();
        records.removeIf(record -> record.id() == id);
    };
}
```

Method Reference

```java
/**
 * Search a book using a filter predicate
 * @param content
 * @param filter
 */
private void search(String content, Predicate<Book> filter) {
    // LVTI
    var matches = books.stream().filter(filter).toList();
    if (matches.isEmpty()) {
        IO.println(new LibraryItemNotFoundException("No book found for: " + content));
    } else {
        // Java Lambda method reference
        matches.forEach(Book::info);
    }
}
```

Switch expressions

```java
/**
 * Compact source main method (JEP 512)
 * @author A00325358 Pronoti Saha
 * @param args Arguments
 */
Run | Debug
void main(String[] args) {

    try (Scanner sc = new Scanner(System.in)) {
        IO.println(obj: "Starting Library System");
        var library = new Library();
        loadLibraryItems(library);

        // static interface method
        Borrowable.libraryInfo();

        int choice;
        do {
            try {
                showMenu();
                choice = sc.nextInt();
                sc.nextLine(); // consume newline

                switch (choice) {
                    case 1 -> {
                        library.showAllItems();
                        waitForEnter(sc);
                    }
                    case 2 -> {
                        IO.print(obj: "Enter title to search: ");
                        String title = sc.nextLine();
                        library.searchByTitle(title);
                        waitForEnter(sc);
                    }
                    case 3 -> {
                        IO.print(obj: "Enter author to search: ");
                        String author = sc.nextLine();
                        library.searchByAuthor(author);
                        waitForEnter(sc);
                    }
                    case 4 -> {
                        IO.print(obj: "Enter book ID to borrow: ");
                        int id = sc.nextInt();
                        sc.nextLine();
                        IO.print(obj: "Enter your name: ");
                        String name = sc.nextLine();
                        library.borrowBook(id, name);
                        waitForEnter(sc);
                    }
                    case 5 -> {
                        IO.print(obj: "Enter book ID to return: ");
                        int id = sc.nextInt();
                        sc.nextLine();
                        library.returnBook(id);
                        waitForEnter(sc);
                    }
                    case 6 -> {
                        library.showBooksStatistics();
                        waitForEnter(sc);
                    }
                    case 7 -> IO.println(obj: "Exiting...");
                    default -> {
                        IO.println(obj: "Invalid choice!");
                        waitForEnter(sc);
                    }
                }
```

Sealed classes and Interfaces

Sealed class

```java
/**
 * Unchecked Library Operation Exception
 * @author A00325358 Pronoti Saha
 */
public sealed class LibraryOperationException extends RuntimeException permits LibraryItemNotFoundException {

    /**
     * Constructor
     * @param message Error Message
     */
    public LibraryOperationException(String message) {
        super(message);
    }
}
```

Sealed interface

```java
/**
 * LibraryItem is a sealed interface
 * Permits Book and Magazine
 * @author A00325358 Pronoti Saha
 */
public sealed interface LibraryItem permits Book, Magazine {
    /**
     * Library item info
     */
    void info();
}
```

Records for immutable data

```java
package com.library.records;

import java.time.LocalDate;

/**
 * Library Record is used to maintain records of borrowed library items
 *
 * @author A00325358 Pronoti Saha
 * @param id Id
 * @param title Title
 * @param author Author
 * @param borrowDate Borrow Date
 * @param dueDate Due Date
 */
public record LibraryRecord(int id, String title, String author, LocalDate borrowDate, LocalDate dueDate) {
}
```

Java 25 Features:

Compact source files (no explicit class for main)

```
/**
 * Compact source main method (JEP 512)
 * @author A00325358 Pronoti Saha
 * @param args Arguments
 */
Run | Debug
void main(String[] args) {
    try (Scanner sc = new Scanner(System.in)) {
        IO.println(obj: "Starting Library System");
        var library = new Library();
        loadLibraryItems(library);

        // static interface method
        Borrowable.libraryInfo();

        int choice;
        do {
            try {…
            catch (InputMismatchException e) {…
        } while (choice != 7);
    }
}

/**
 * Load library with books and magazines
 * @param library
 */
private static void loadLibraryItems(Library library) {…

/**
 * Program Menu
 */
private static void showMenu() {…

/**
 * Wait for Enter to display Menu
 * @param sc
 */
private static void waitForEnter(Scanner sc) {…
```

Flexible constructor bodies

```java
/**
 * Book is a borrowable library item
 * @author A00325358 Pronoti Saha
 */
public final class Book implements LibraryItem, Borrowable {

    private final int id;
    private final String title;
    private final String author;
    private boolean isBorrowed;
    private LocalDate dueDate;

    /**
     * Flexible constructor body Java 25 (JEP 513)
     * @param id Book Id
     * @param title Book Title
     * @param author Book Author
     */
    public Book(int id, String title, String author) {
        // validate before initializing fields
        if (id <= 0) throw new IllegalArgumentException(s: "ID must be positive");
        if (title == null || title.isEmpty()) throw new IllegalArgumentException(s: "Title cannot be empty");
        if (author == null || author.isEmpty()) throw new IllegalArgumentException(s: "Author cannot be empty");

        this.id = id;
        this.title = title;
        this.author = author;
        this.isBorrowed = false;
    }
```

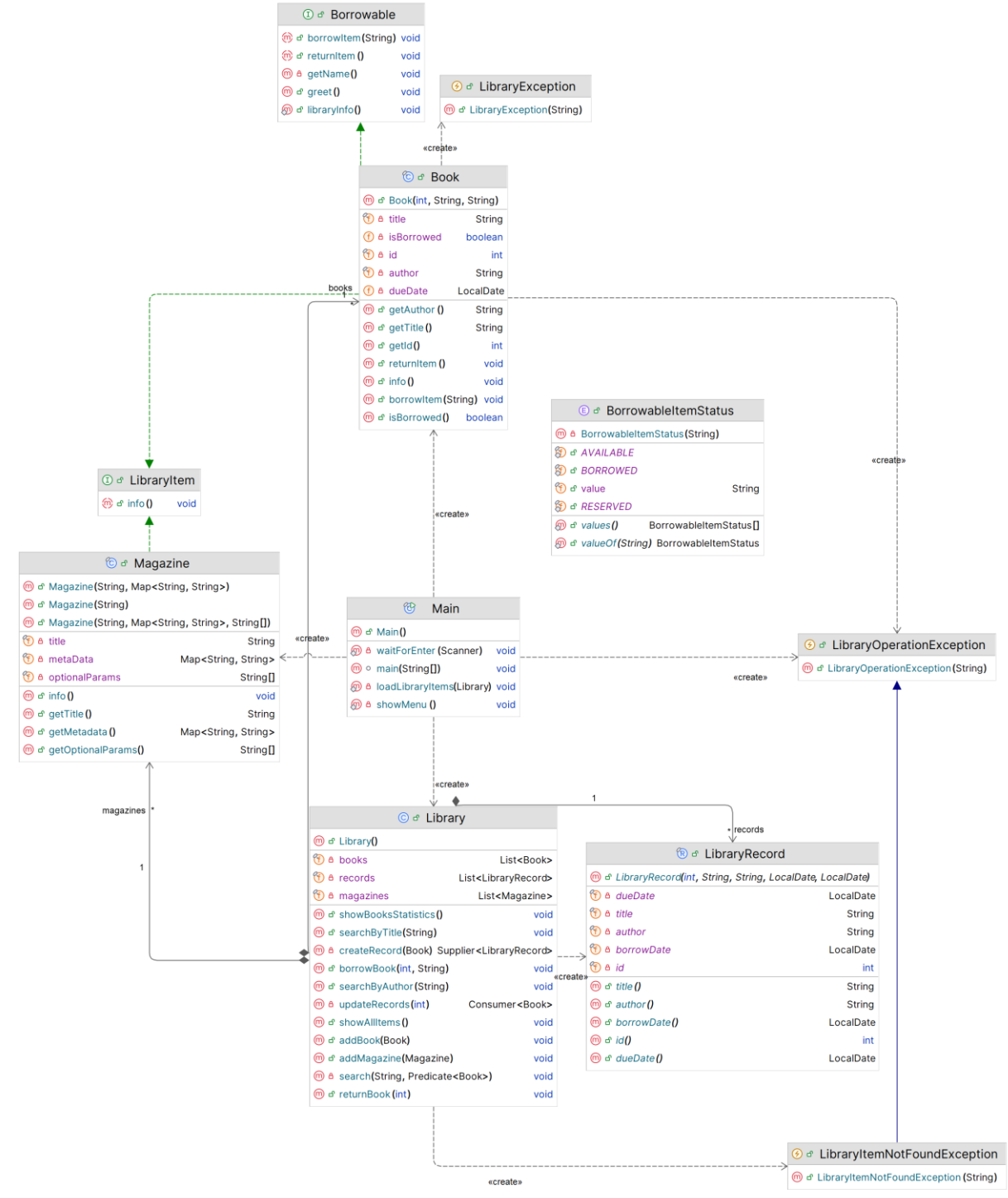
Challenges

- Integrating new Java 25 features with traditional OOP design.
- Ensuring robust exception handling for user input and library operations.
- Maintaining code readability and modularity while using advanced language features.

Areas of Improvements

- Add persistent storage (e.g., file or database) for library data.
- Implement user authentication and roles (e.g., admin, member).
- Enhance the search functionality with partial matches and filters.
- Develop a graphical user interface (GUI) for better usability.
- Add support for additional item types (e.g., DVDs, e-books).

Appendix 1 - Class UML Diagram



References

- Official Java Tutorials: https://docs.oracle.com/javase/tutorial/
- Java 25 Documentation: https://docs.oracle.com/en/java/javase/25/
- W3Schools: https://www.w3schools.com/