

Final Project

CPS2232: DATA STRUCTURES

Project Name: Library Management System

|  |  |
| --- | --- |
| Student Name: | Chun Chen (Dave)  QU Wenyi  Cheng Leyi |
| Student ID: | 1234930  1307956  1307840 |
| Lecture: | Dr. Kennedy Ehimwenma Ph.D. |

Spring 2025

**Introduction:**

* Abstract
* Introduction
* Related work(review of Literature)
* Research methodology
* Analysis
* Discussion
* Conclusion
* Reference

**Abstract**

This project focuses on the design and implementation of a Student Library Management System developed using the Java programming language and grounded in object-oriented programming (OOP) principles such as inheritance, encapsulation, abstraction, and modularity. The system aims to streamline the daily operations of a library by providing essential functionalities, including user registration, book management, borrowing and returning books, and differentiated access control for administrators and students. By employing a structured class hierarchy—where the User class serves as a superclass for both Admin and Student—the system supports role-specific behavior while promoting code reuse and simplicity.To ensure data management is both efficient and scalable, the system makes extensive use of the Java Collection Framework, particularly Map<Integer, User> for user records and List<Book> for handling book inventories and student borrowings. The interface, though implemented in a text-based command-line format, provides clear and functional interactions, which can be extended to a graphical user interface (GUI) or web-based platform in future developments.The design and methodology were informed by an extensive review of related literature, with particular attention to the widely-used open-source library automation software Koha. Studies on Koha’s implementation in various universities across regions such as Malaysia, Pakistan, Nigeria, and Uganda provided practical insights into modular system design, user accessibility, and cost-effectiveness. Through detailed UML diagrams, algorithm design, and clear separation of class responsibilities, the project demonstrates effective software engineering practices. Error handling mechanisms were built in to prevent duplicate user IDs, invalid operations, and enforce rules such as borrowing limits. Although the current version lacks features like password encryption, book search filters, and due date tracking, it successfully fulfills its educational purpose by highlighting key programming concepts and delivering a functional, extensible, and easy-to-understand system. Overall, this Library Management System serves as both a practical tool for academic libraries and a foundational learning platform for understanding object-oriented software development.

**Chapter1:**

### **1.1 Problem Statement**

Despite the widespread availability of commercial and open-source library automation systems, many small academic institutions still rely on manual processes or outdated software. These systems often lack customization, are expensive to maintain, and are not scalable to meet the growing needs of educational environments. As a result, librarians and students face inefficiencies in managing and accessing library resources. The key research problem addressed in this project is:

**How can a lightweight, extensible, and role-specific library management system be developed using object-oriented principles to meet the specific needs of educational institutions, while ensuring usability, maintainability, and cost-effectiveness without requiring complex infrastructure?**

### **1.2 Purpose**

The purpose of this project is to develop a modular, maintainable, and scalable Library Management System using Java and object-oriented programming techniques. The system is designed to streamline and simplify library operations by providing a structured, role-based interface for both administrators and students. Key functionalities include user registration, book management, borrowing and returning processes, and differentiated permissions for users based on their roles.

By applying object-oriented principles such as inheritance, encapsulation, and abstraction, the system models real-world library interactions in an efficient and organized way. This makes the system not only suitable for small academic libraries but also adaptable to larger educational institutions, with potential for future extension and customization.

**Chapter2:**

**Related work(Review of Literature)**

In the development of this Library Management System, we reviewed several academic articles and technical resources to understand best practices in software design, information systems, and library automation. Prior research has provided valuable insights into how library systems can improve access to information, optimize book circulation, and enhance user interaction through digital platforms.

One major area of reference comes from studies on **library automation systems**, particularly **Koha**, which has been widely adopted across academic and public libraries. For example, Sharifah Nur Amirah et al. (2023) analyzed the cataloguing module of Koha in Malaysian universities, emphasizing its cost-effectiveness and modularity [1]. Similarly, Jabeen (2024) explored how Koha was adopted in Pakistani universities and the sociotechnical challenges involved [2]. These studies helped guide our system’s role-based structure and resource control.

Further insights into regional implementations can be seen in the work by Ojo (2024), which assessed the use of Koha in Nigerian public universities [3], and by Kulkarni et al. (2023), who discussed intelligent usage strategies to maximize Koha's utility [4]. Their findings supported our choice to implement a lightweight, role-differentiated borrowing system using object-oriented principles.

Additionally, Mutungisa (2018) provided a case study on Koha’s deployment in Uganda’s Bishop Stuart University, offering real-world feedback on system usability, which inspired our simplified, text-based UI [5]. From a development standpoint, Singh (2019) highlighted migration lessons from proprietary systems to open-source platforms like Koha [6], reinforcing the importance of modular design.

In terms of software engineering foundations, we referred to Pressman and Maxim’s *Software Engineering: A Practitioner’s Approach* (2015), which emphasizes object-oriented development as a robust modeling strategy. Usability guidelines were drawn from Nielsen (2000), advocating for simple and user-focused interactions.

By integrating these insights into our project, we aligned our system with proven strategies while customizing the solution to reflect academic and educational needs.

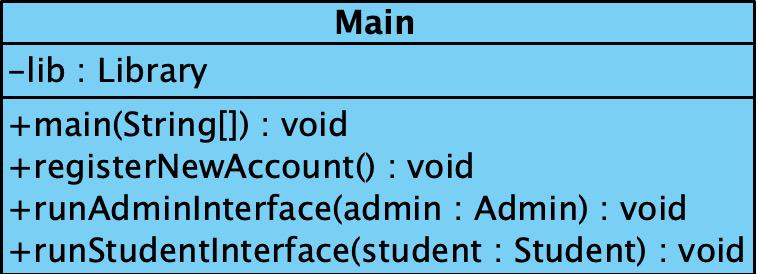
**Chapter3.**

**Research Methodology**

The development of this Student Library Management System is based on object-oriented design and implemented using Java programming language. The methodology follows a modular and layered approach, with a focus on encapsulation, inheritance, abstraction, and reuse.

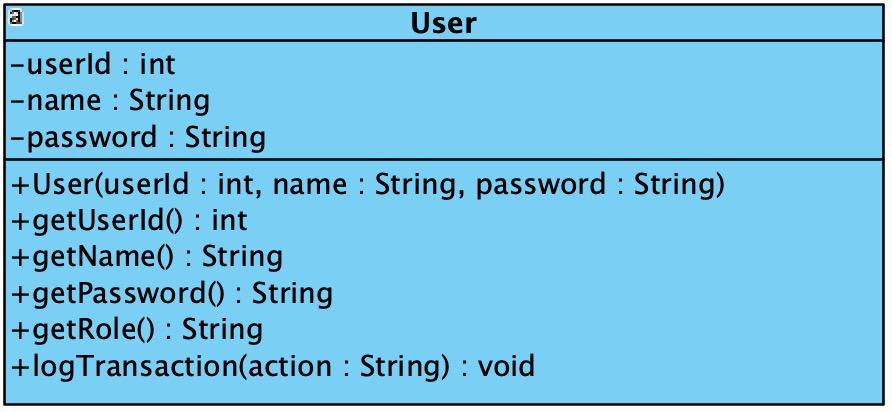
Key components of the methodology include:

* **UML Diagrams**  
  Class diagrams were used to represent the relationships between entities such as Main, User, Student, Admin, Library, and Book. Inheritance is used to model shared behaviors between different user types.
* Main class Diagram(User Interface):



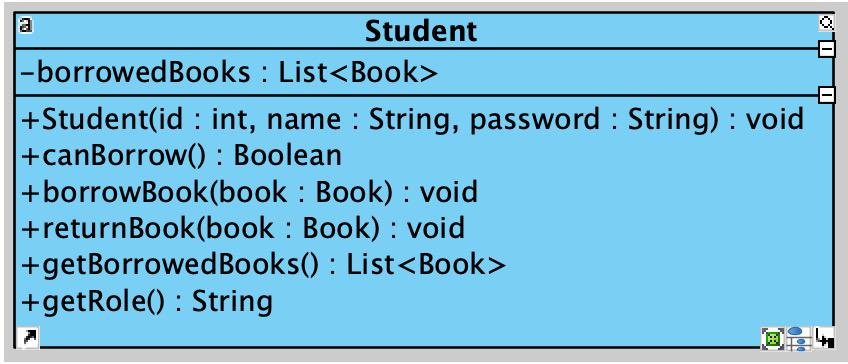
In the Main Class we use the method registerNewAccount() to registerNewAccount for both Admin and Student, and we use the method runAdminInterface() and runStudentInterface() to handle the corresponding requirements.

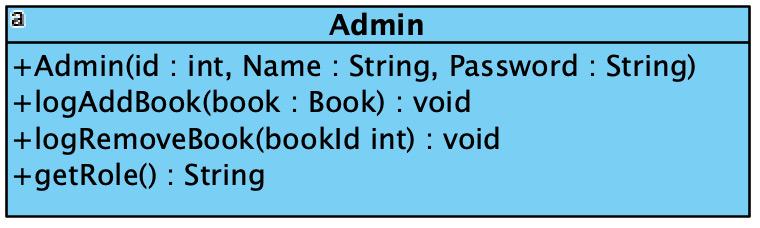
* User Class Diagram:



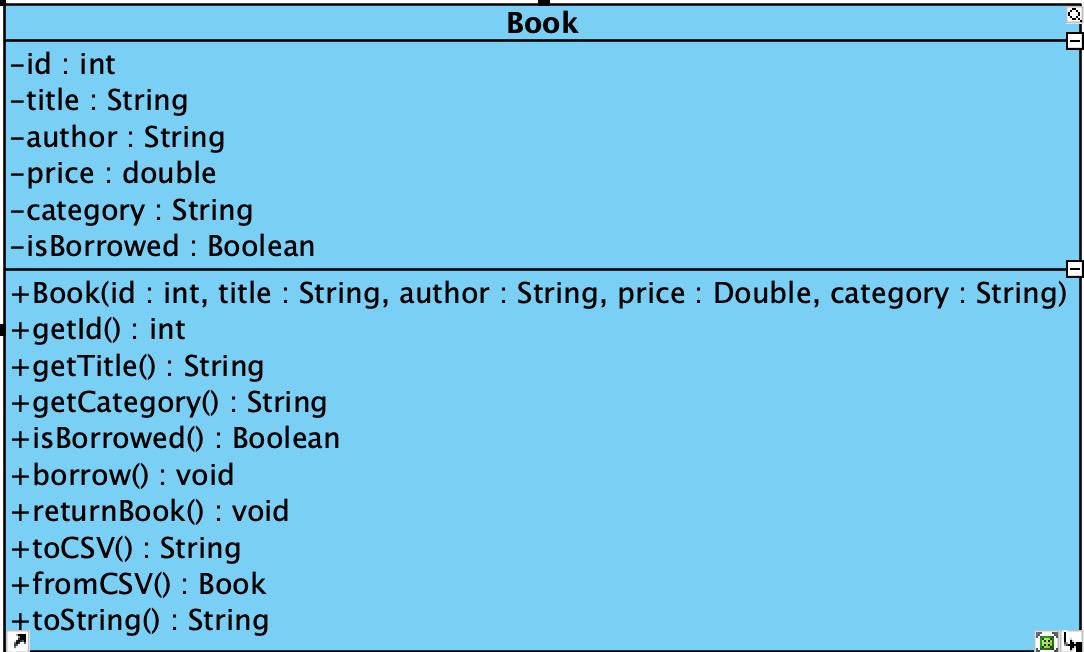
In the User class we define the common features(attribues and method) for the User's child class.

* Student and Admin Class Diagram:



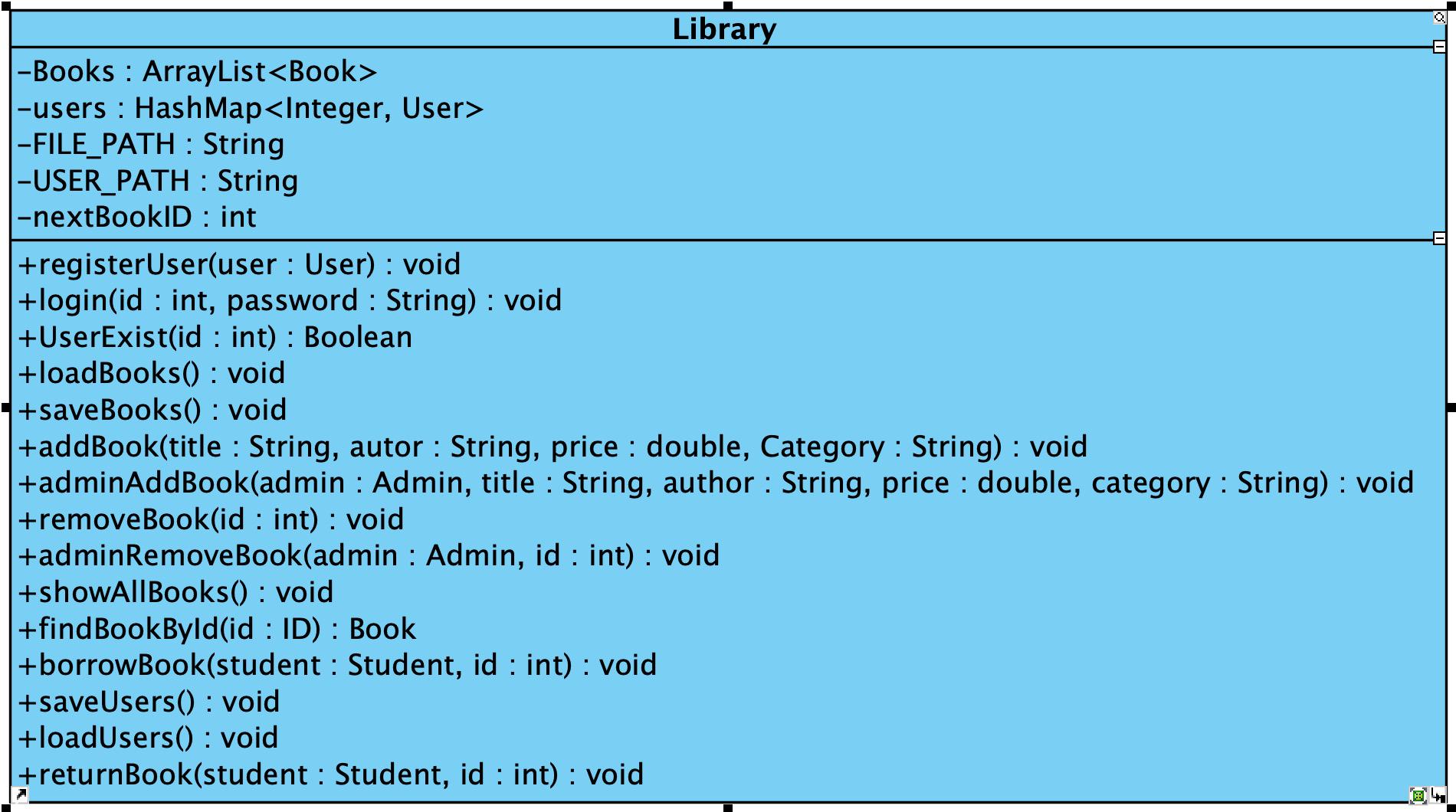


As two child class of User class they both have unique method inoder to deal with different taskes.

* Book Class Diagram：

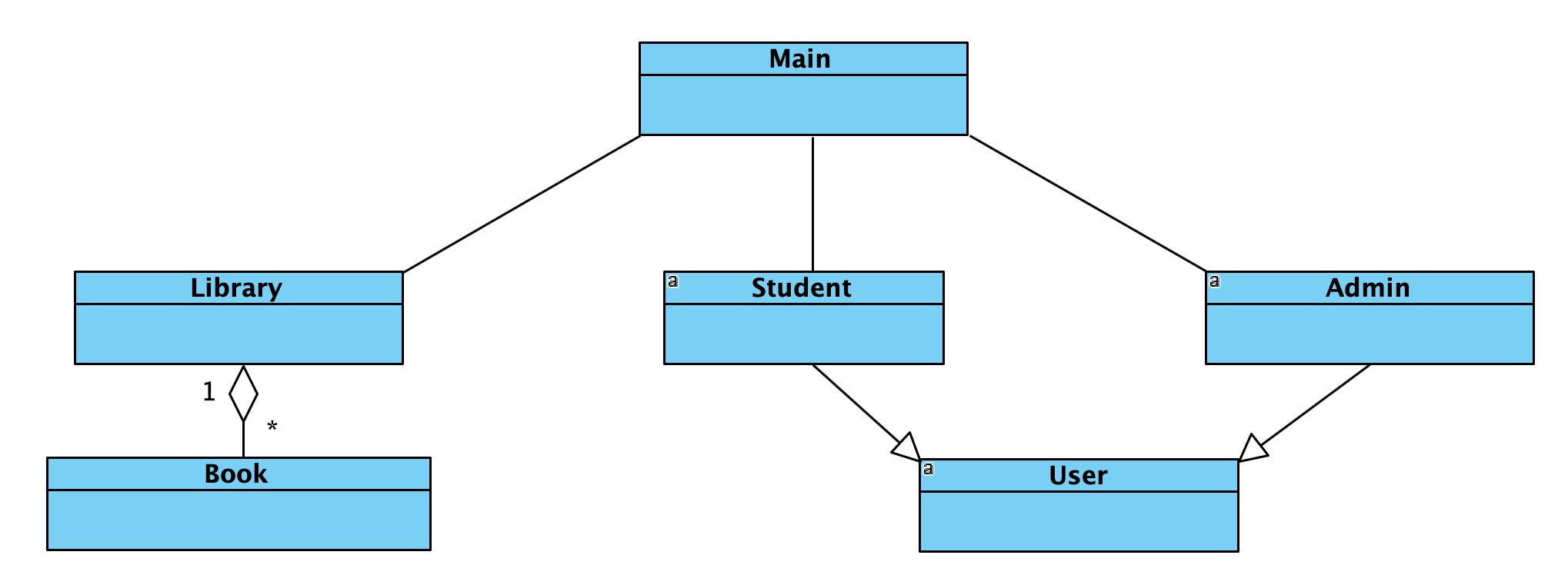
the Book class contain the attributes to discribe the Book featue also have some method to load and output the book list.

* Library Class Diagram:



the relationship between Library class and Book class is aggresive, Library use arraylist to store book object and use method such adminRemoveBook to deal with it, same as HashMap used to store User object.

the whole class diagram are below(see fig1.), it shows the association and compositon between each class



(fig1. Whole class diagram)

* **Whole Code**
  + - **Main Class**

package StudentLibraryManagementSystem;

import java.util.Scanner;

public class Main {

private static final Scanner sc = new Scanner(System.in);

private static final Library lib = new Library();

public static void main(String[] args) {

lib.loadBooks();

lib.loadUsers();

System.out.println("=== Welcome to the Library System ===");

System.out.println("1. Register New Account");

System.out.println("2. Login");

System.out.print("Choose an option: ");

String option = sc.nextLine();

if (option.equals("1")) {

registerNewAccount();

}

System.out.print("Enter User ID: ");

int id = Integer.parseInt(sc.nextLine());

System.out.print("Enter Password: ");

String password = sc.nextLine();

User user = lib.login(id, password);

if (user == null) {

System.out.println("Login failed. Please check your credentials.");

return;

}

System.out.println("Login successful. Role: " + user.getRole());

if (user instanceof Admin) {

runAdminInterface((Admin) user);

} else if (user instanceof Student) {

runStudentInterface((Student) user);

}

lib.saveBooks();

lib.saveUsers();

System.out.println("Session ended. Goodbye!");

}

private static void registerNewAccount() {

System.out.print("Register as (admin/student): ");

String role = sc.nextLine().trim().toLowerCase();

System.out.print("Choose a unique ID (number): ");

int id = Integer.parseInt(sc.nextLine());

while (lib.userExists(id)) {

System.out.println("User ID already exists. Please try a different ID:");

id = Integer.parseInt(sc.nextLine());

}

System.out.print("Enter your name: ");

String name = sc.nextLine();

System.out.print("Choose a password: ");

String pw = sc.nextLine();

if (role.equals("admin")) {

lib.registerUser(new Admin(id, name, pw));

} else {

lib.registerUser(new Student(id, name, pw));

}

lib.saveUsers();

System.out.println("Account registered successfully. You can now log in.");

}

private static void runAdminInterface(Admin admin) {

while (true) {

System.out.println("\n=== Admin Menu ===");

System.out.println("1. Add Book");

System.out.println("2. Remove Book");

System.out.println("3. View All Books");

System.out.println("4. Exit");

System.out.print("Choose an option: ");

String choice = sc.nextLine();

switch (choice) {

case "1":

System.out.print("Title: ");

String title = sc.nextLine();

System.out.print("Author: ");

String author = sc.nextLine();

System.out.print("Price: ");

double price = Double.parseDouble(sc.nextLine());

System.out.print("Category: ");

String category = sc.nextLine();

lib.adminAddBook(admin, title, author, price, category);

lib.saveBooks();

break;

case "2":

lib.showAllBooks();

System.out.print("Enter Book ID to remove: ");

int removeId = Integer.parseInt(sc.nextLine());

lib.adminRemoveBook(admin, removeId);

break;

case "3":

lib.showAllBooks();

break;

case "4":

return;

default:

System.out.println("Invalid choice.");

}

}

}

private static void runStudentInterface(Student student) {

while (true) {

System.out.println("\n=== Student Menu ===");

System.out.println("1. View All Books");

System.out.println("2. Borrow Book");

System.out.println("3. Return Book");

System.out.println("4. View My Borrowed Books");

System.out.println("5. Exit");

System.out.print("Choose an option: ");

String choice = sc.nextLine();

switch (choice) {

case "1":

lib.showAllBooks();

break;

case "2":

lib.showAllBooks();

System.out.print("Enter Book ID to borrow: ");

int borrowId = Integer.parseInt(sc.nextLine());

lib.borrowBook(student, borrowId);

lib.saveBooks();

break;

case "3":

student.showBorrowedBooks();

System.out.print("Enter Book ID to return: ");

int returnId = Integer.parseInt(sc.nextLine());

lib.returnBook(student, returnId);

lib.saveBooks();

break;

case "4":

student.showBorrowedBooks();

break;

case "5":

return;

default:

System.out.println("Invalid choice.");

}

lib.saveBooks();

}

}

}

* + - **User Class**package StudentLibraryManagementSystem  
      import java.io.FileWriter;  
      import java.io.IOException;  
      import java.time.LocalDateTime;  
      import java.time.format.DateTimeFormatter;

public abstract class User {  
 protected int userId;  
 protected String name;  
 protected String password;

public User(int userId, String name, String password) {  
 this.userId = userId;  
 this.name = name;  
 this.password = password;  
}

public int getUserId() { return userId; }  
public String getName() { return name; }  
public String getPassword() { return password; }  
public abstract String getRole();  
public void logTransaction(String action) {

String filename = "user\_" + userId + ".txt";  
 try (FileWriter fw = new FileWriter(filename, true)) {  
String timestamp = LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));

fw.write("[" + timestamp + "] " + action + "\n");  
} catch (IOException e) {  
System.out.println("Failed to log transaction for user " + userId);  
 }  
}  
}

* + - **Student Class**

package StudentLibraryManagementSystem;

import java.util.ArrayList;  
import java.util.List;

public class Student extends User {

private List<Book> borrowedBooks = new ArrayList<>();

public Student(int id, String name, String password) {

super(id, name, password);

}

public boolean canBorrow() {

return borrowedBooks.size() < 3;

}

public void borrowBook(Book book) {

borrowedBooks.add(book);

logTransaction("Borrowed book: " + book.getTitle());

}

public void returnBook(Book book) {

borrowedBooks.remove(book);

logTransaction("Returned book: " + book.getTitle());

}

public List<Book> getBorrowedBooks() {

return borrowedBooks;

}

public void showBorrowedBooks() {

System.out.println("Books borrowed by " + name + ":");

for (Book b : borrowedBooks)

System.out.println(" " + b);

}

@Override

public String getRole() {

return "Student";

}

}

* + - **Admin Class**

package StudentLibraryManagementSystem;

public class Admin extends User {

public Admin(int id, String name, String password) {

super(id, name, password);

}

public void logAddBook(Book book) {

logTransaction("Added book: " + book.getTitle());

}

public void logRemoveBook(int bookId) {

logTransaction("Removed book ID: " + bookId);

}

@Override

public String getRole() {

return "Admin";

}

}

* + - **Book Class**

package StudentLibraryManagementSystem;

public class Book {

private int id;

private String title;

private String author;

private double price;

private String category;

private boolean isBorrowed;

public Book(int id, String title, String author, double price, String category) {

this.id = id;

this.title = title;

this.author = author;

this.price = price;

this.category = category;

this.isBorrowed = false;

}

public int getId() { return id; }

public String getTitle() { return title; }

public String getCategory() { return category; }

public boolean isBorrowed() { return isBorrowed; }

public void borrow() { isBorrowed = true; }

public void returnBook() { isBorrowed = false; }

public String toCSV() {

return String.format("%d,%s,%s,%.2f,%s,%s",

id, title, author, price, category, isBorrowed ? "yes" : "no");

}

public static Book fromCSV(String line) {

String[] parts = line.split(",", -1);

Book book = new Book(

Integer.parseInt(parts[0]),

parts[1],

parts[2],

Double.parseDouble(parts[3]),

parts[4]

);

if (parts[5].equalsIgnoreCase("yes")) {

book.borrow();

}

return book;

}

@Override

public String toString() {

return String.format("Book[ID=%d, Title='%s', Author='%s', Price=%.2f, Category=%s, %s]",

id, title, author, price, category, isBorrowed ? "Borrowed" : "Available");

}

}

* + - **Library Class**

package StudentLibraryManagementSystem;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.time.LocalDateTime;  
import java.time.format.DateTimeFormatter;

public abstract class User {  
 protected int userId;  
 protected String name;  
 protected String password;

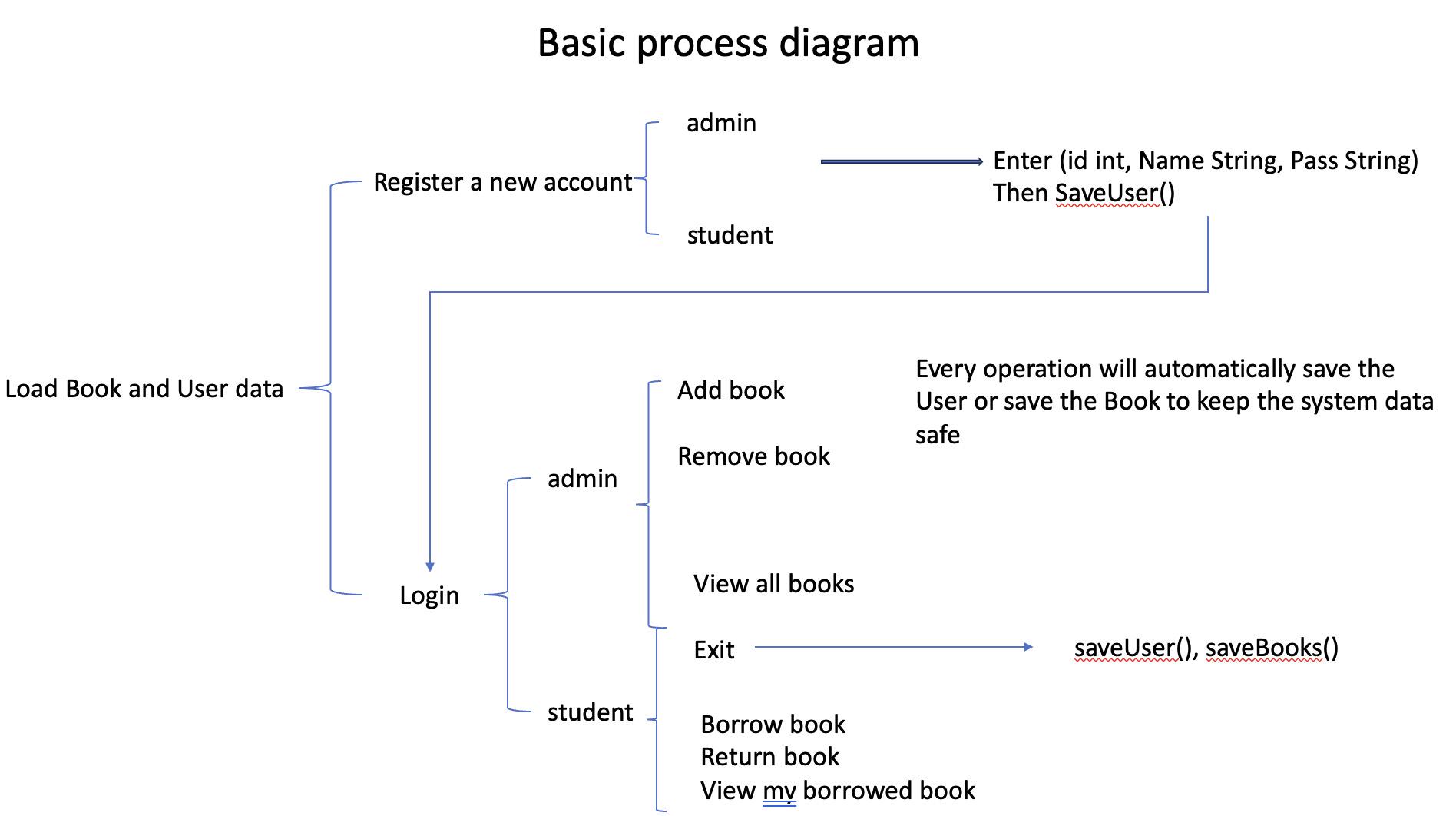
public User(int userId, String name, String password) {  
 this.userId = userId;  
 this.name = name;  
 this.password = password;  
}

public int getUserId() { return userId; }  
public String getName() { return name; }  
public String getPassword() { return password; }  
public abstract String getRole();  
public void logTransaction(String action) {

String filename = "user\_" + userId + ".txt";  
 try (FileWriter fw = new FileWriter(filename, true)) {  
String timestamp = LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));

fw.write("[" + timestamp + "] " + action + "\n");  
} catch (IOException e) {  
System.out.println("Failed to log transaction for user " + userId);  
 }  
}  
}

* **Java Collection Framework**  
  The system makes significant use of the Java Collection Framework:
  + Map<Integer, User>: Used in the Library class to manage all registered users. This provides fast lookup by user ID and ensures uniqueness of IDs.
  + List<Book>: Used to store both the entire collection of books and the list of books borrowed by each student.  
    These collection classes allow for dynamic memory management, efficient searching, and easy iteration over objects. Use of interfaces (Map, List) rather than concrete implementations increases code flexibility and maintainability.
* **Algorithm Design**  
  Logical flowcharts and control structures were designed for major operations like login, borrow/return of books, and file I/O. Decisions were based on user type and the current state of books.
* **Code Structure**
  + Abstract class User with subclasses Admin and Student
  + Library class for centralized logic, file reading/writing
  + Main class for user interface and control flow  
    Each method was written with single-responsibility and reuse in mind.
* **Basic Process diagram**



* **File-based Data Persistence**  
  The system uses .csv files (books.csv, users.csv) for saving and loading data persistently across sessions. Book records include headers and use comma-separated values for compatibility and readability.
* **Error Handling and Validation**  
  Includes logic to avoid duplicate user IDs, invalid book selections, and input mismatches.

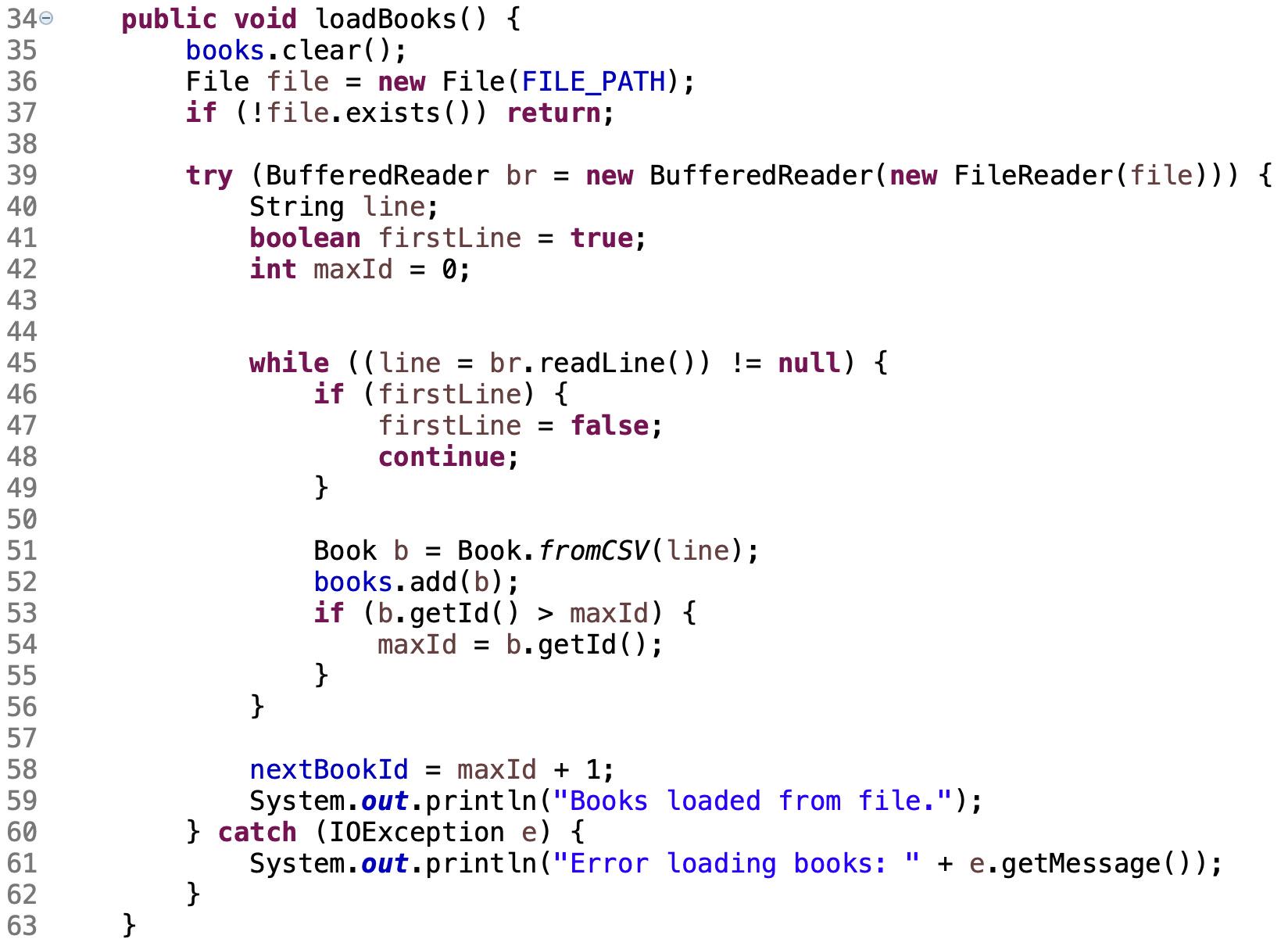
## **Chapter4.**

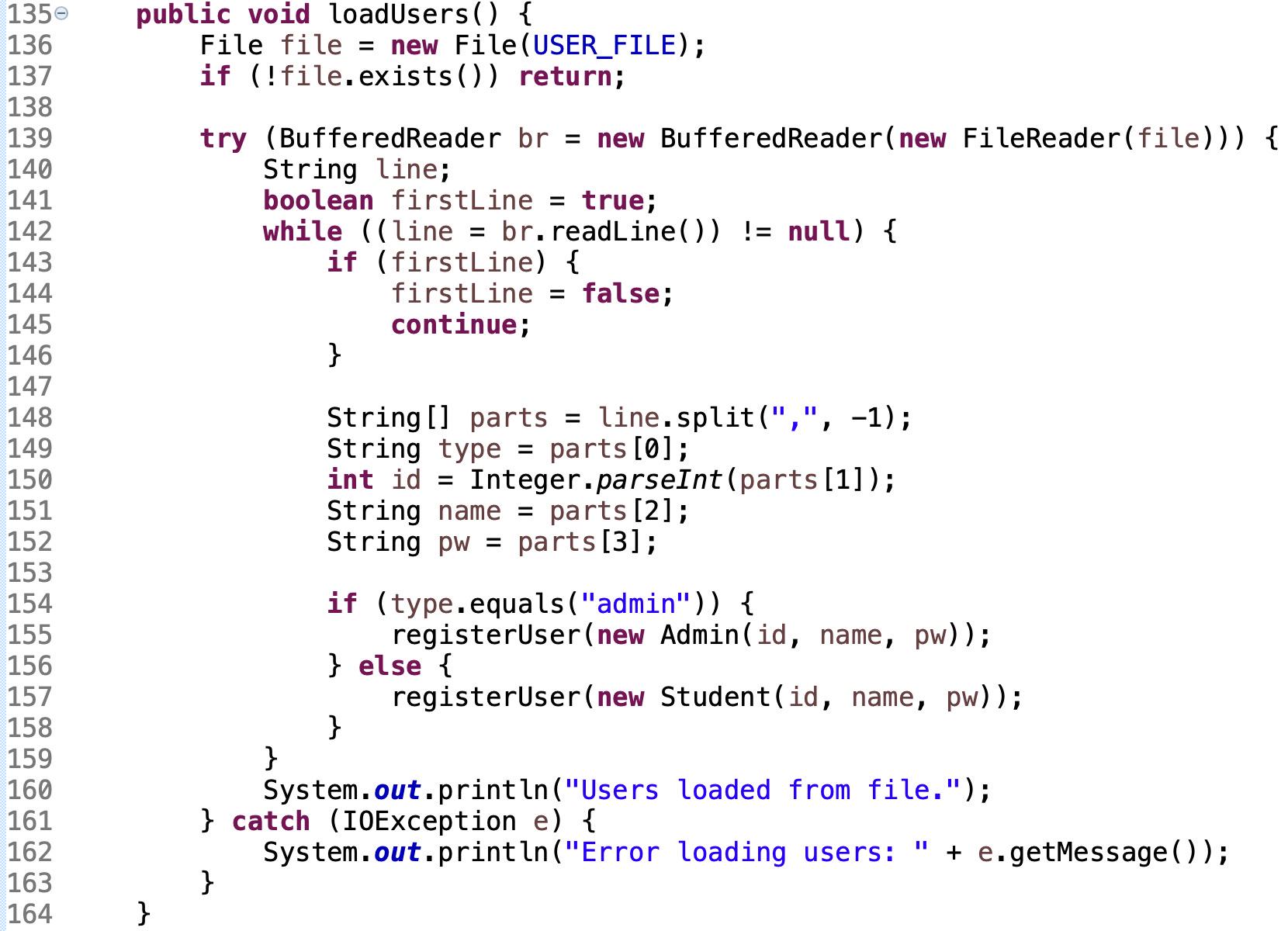
### **4.1 Analysis**

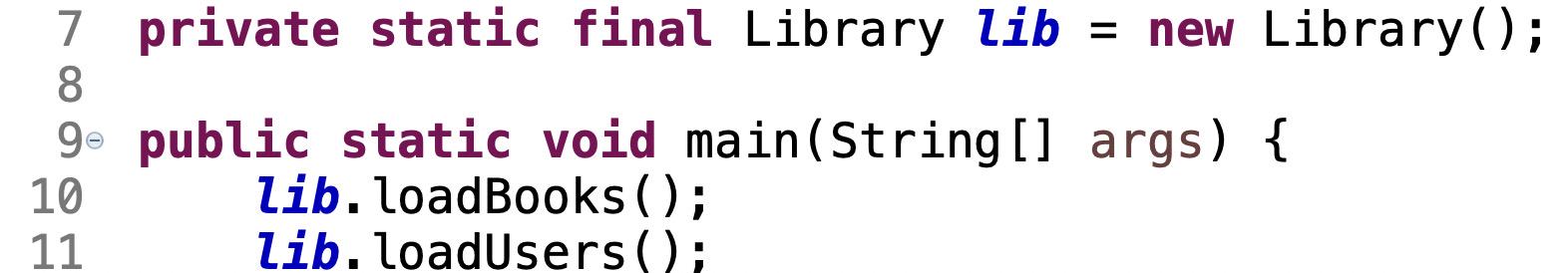
This section highlights key features of the system and presents visual and textual analysis of how the components interact.

* **Code Screenshots**:  
  The system includes modular files such as:
  + loadUsers(see fig2.) and loadBooks(see fig3.) method to store the existing user information and book information in the database into the library object(see fig4.) in the main method.

In both loadBooks and loadUsers method we need follow the parameter order we define in the Book class and User class's constructor.

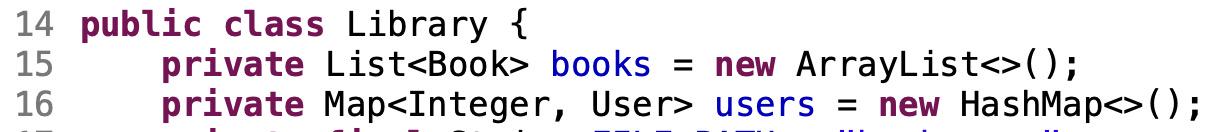
  
(fig2. loadBook method in library class)

  
(fig3. loadUsers method in Library class)



(fig4. the Library object and main class use two method to first store the data)

* + The architecture Library object use to store the book information is ArrayList and HashMap(see fig5.), and the reason why we use HashMap we will discuss in the discussion part later.

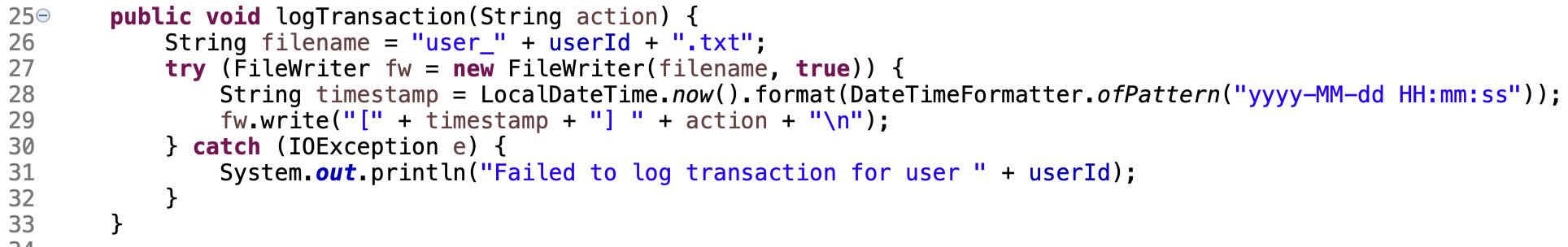


(fig5. Books' ArrayList and Users' HashMap in Library)

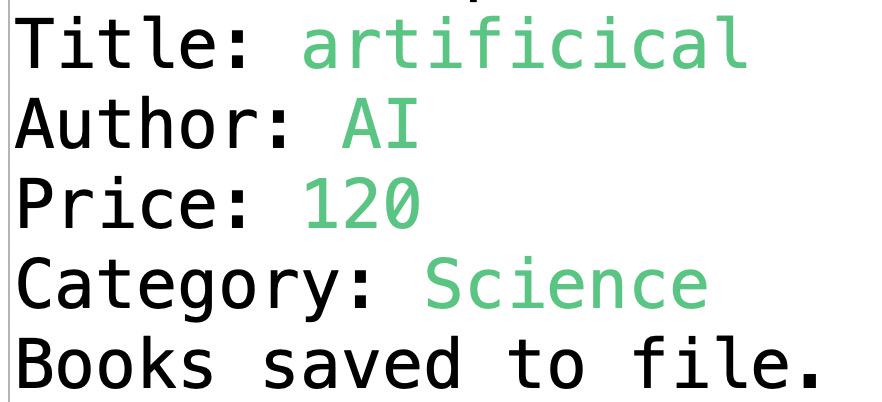
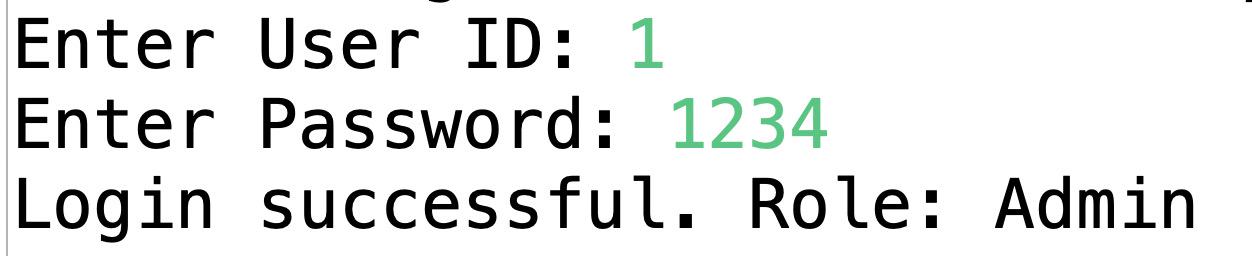
In the HashMap we use Users.getUserId() as a unique key and User object as the value.

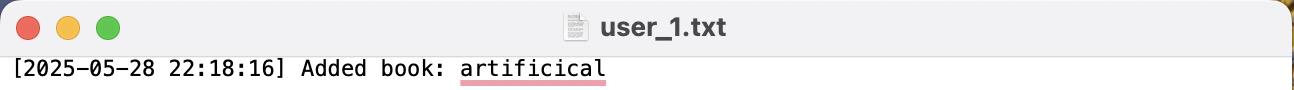
* + The method we use to store log transaction for each User

In User class we defined a common method to accept action from Admin and Student(see fig6.) and record these actions + specific times in the corresponding personal files(see fig7.)



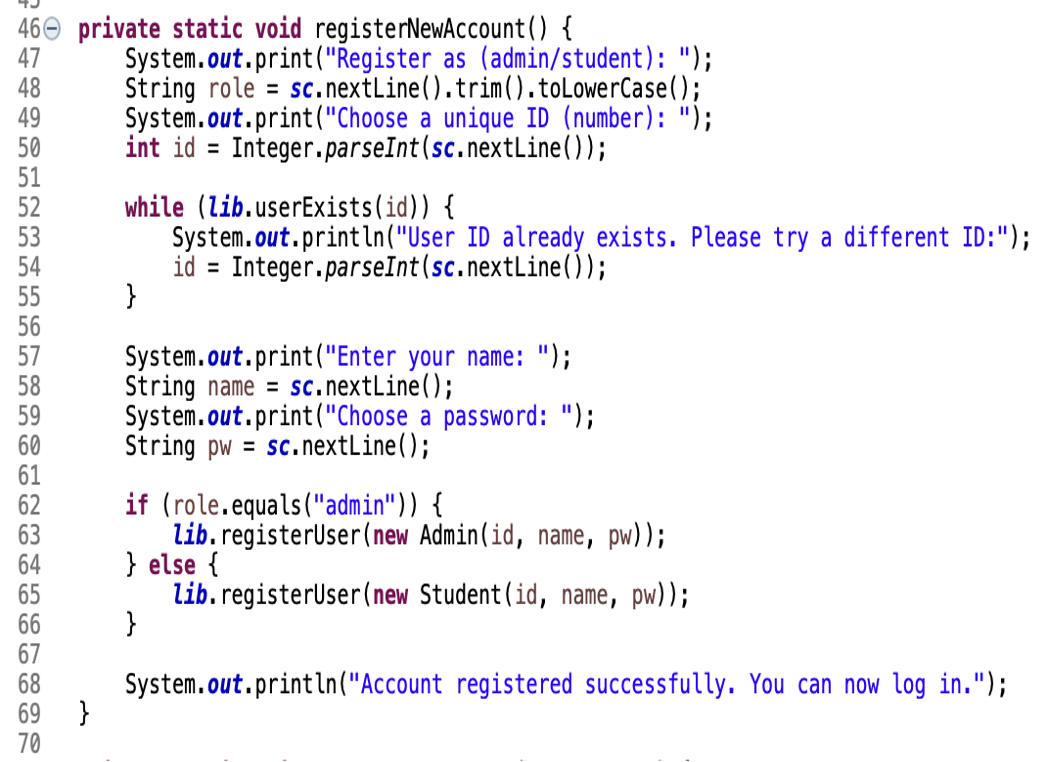
(fig6. the logTransaction method in User class)

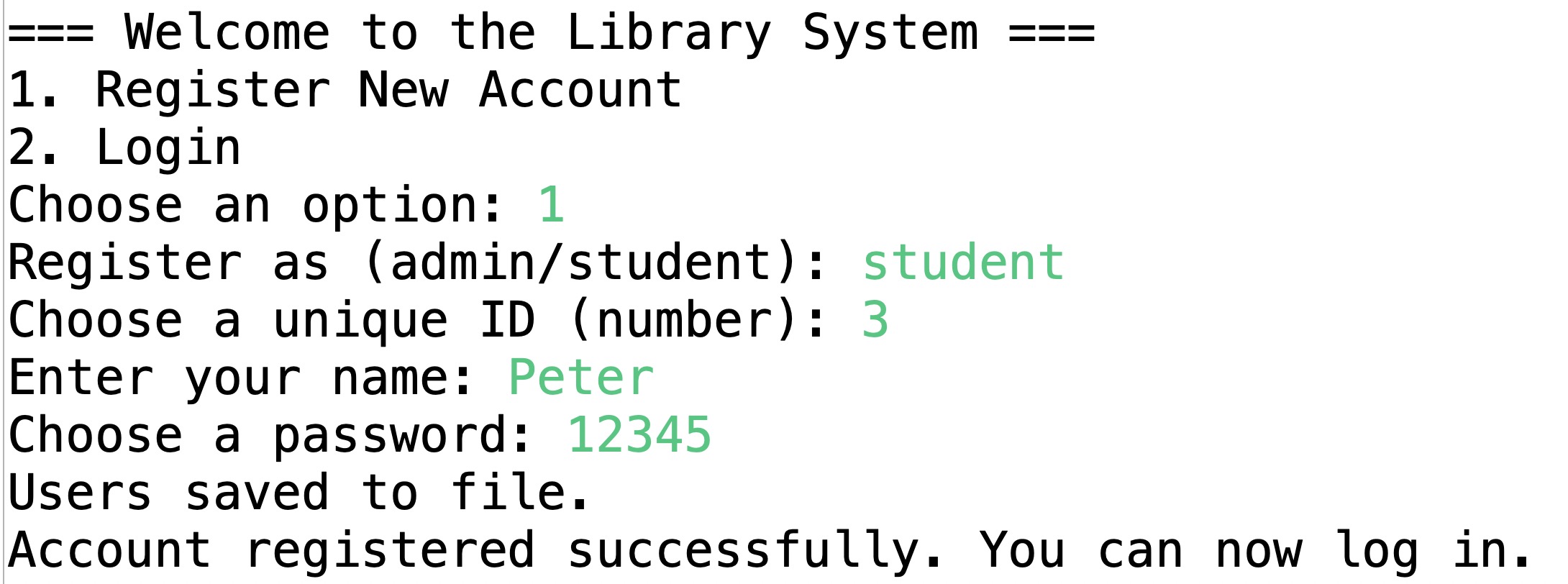




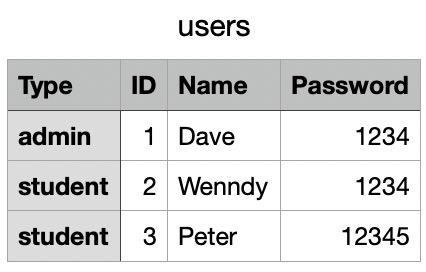
(fig7. User's unique text file with unique ID)

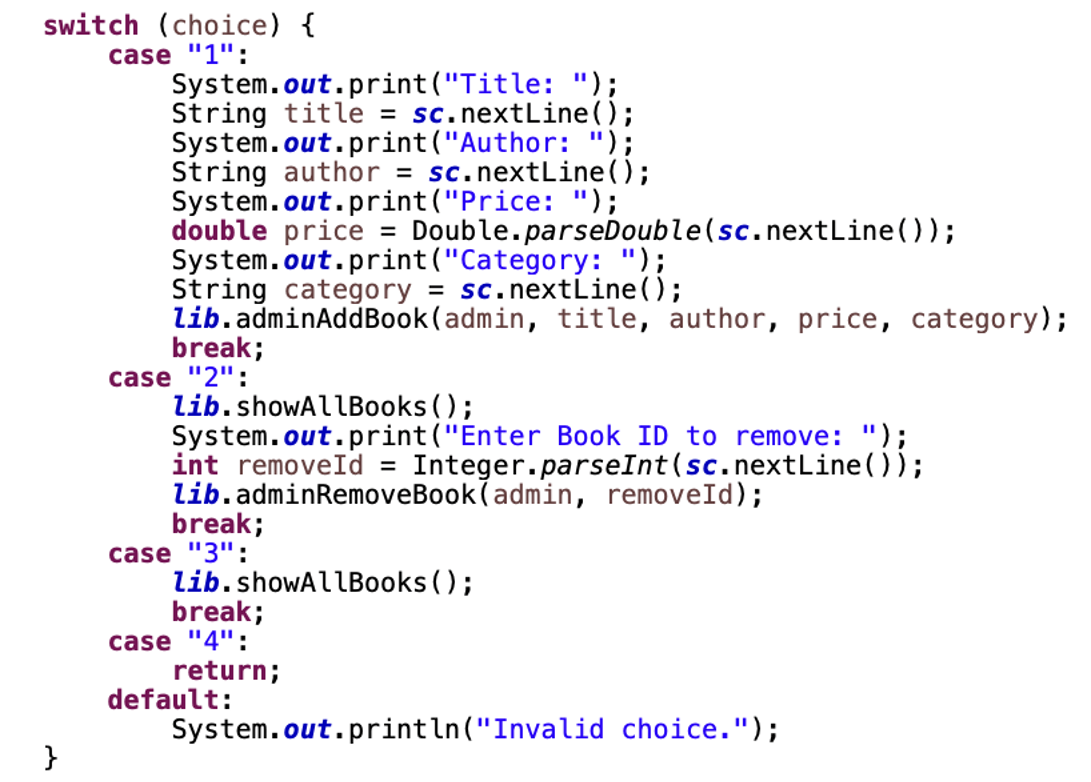
In this method(see fig6.) we save each action in there .txt file based on thier unique ID.

* + The Logic structure we use to register
  + 
  + This Java method registers new user accounts (either admin or student) by prompting for a role, validating a unique numeric ID through existence checks, collecting the user's name and password, instantiating the appropriate role-specific object (Admin or Student), registering it via a lib.registerUser() call, and confirming successful registration.

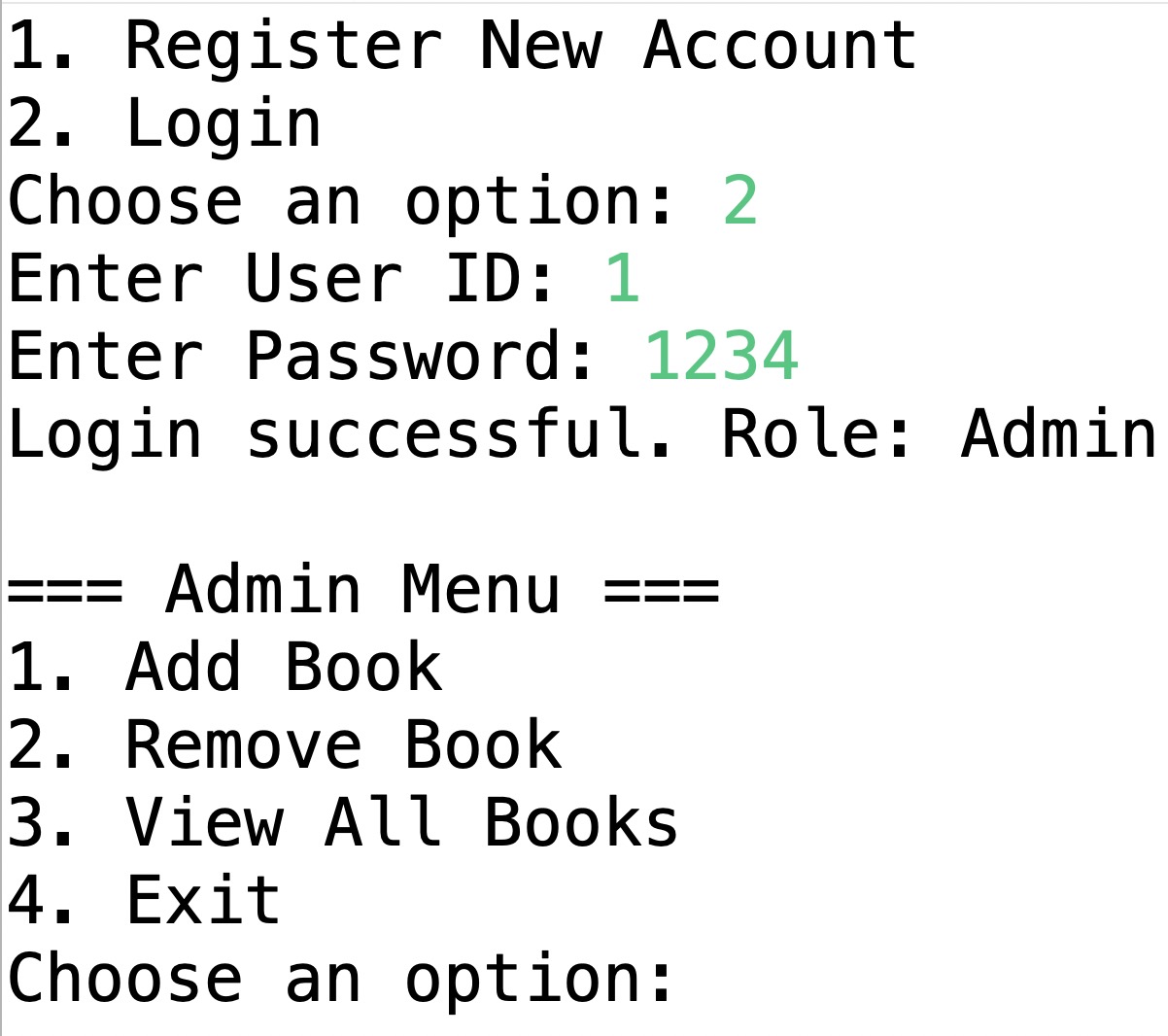


* + Users can register accounts as either administrators or students. Administrators have privileges to add or remove books, while students can borrow, return books, and track their borrowing history in real time. All user data is stored securely with encryption to ensure privacy. In the picture user Peter register is account, and his information is added in the csv file.(see fig8.)

  
(fig8. the csv form for store Admin and Student information)

* + The Logic structure we use to login
  + 

This Java switch statement processes administrative library operations based on user input: adding a book by collecting title, author, price, and category then invoking adminAddBook; removing a book by displaying all books, accepting a book ID, and calling adminRemoveBook; viewing all books via showAllBooks; exiting the menu when selecting "4"; and handling invalid choices with an error message.



(fig9. the Login interface)

This is the interface of a library management system(see fig9.), showing the process after a user logs in. For exmple admin Dave selects the login option, then enters the User ID as 1 and the password as 1234. The system confirms a successful login and displays the user's role as Admin. After logging in, the user accesses the admin menu, which provides options to add books, remove books, view all books, or exit the system.

**4.2 Discussion：**

The Student Library Management System demonstrates the effective application of object-oriented programming principles combined with Java's Collection Framework and file handling mechanisms.

One of the core design decisions was the separation of responsibilities across classes. By using an abstract User class with concrete subclasses Admin and Student, the system achieves clean inheritance, allowing user-specific behaviors while avoiding code duplication. This structure also makes it easier to extend the system in the future, for example by adding a Teacher or Librarian user type.

Another important decision was to use the **Java Collection Framework**:

* Hashmap<Integer, User> enables efficient storage and lookup of users by ID.
* List<Book> supports dynamic book collections and flexible borrowing records per student.

**Time Complexity Discussion：**Consideration of time complexity is also one of the important reasons why we choose HashMap.  
In this system, users need to log in by ID, check for duplicate names when registering, and administrators need to find specific users for management. If ArrayList or other linear structures are used, each search will take O(n) time. The core advantage of HashMap is its implementation structure based on hash tables, which makes common operations such as .get(key), .put(key, value), .containsKey(key) have an average time complexity of O(1). In contrast, although TreeMap can sort keys, its operation complexity is O(log n), and its performance is not as good as HashMap.

For example, when the number of users is 10,000：

* + Using ArrayList to find a user requires an average of 5,000 traversals (O(n));
  + Using a TreeMap to look up a user requires about log₂(10,000) ≈ 14 comparisons;
  + HashMap only needs one direct positioning on average, which is extremely efficient.

Therefore, in order to achieve more efficient user retrieval and system response speed, HashMap is the best choice, especially suitable for the situation where the number of users in this book management system may gradually increase.

**The use of CSV file:**

The use of plain text .csv files for data persistence proved to be a simple and effective choice. It avoids the complexity of a database while still enabling the system to retain state between sessions. Adding headers to the .csv files improved readability and debugging.

Error handling and validation were also key aspects of the implementation. The system prevents:

* Duplicate user registration by checking existing IDs
* Students from borrowing more than 3 books
* Borrowing or returning books that do not exist or do not belong to the user

From a user experience perspective, the command-line interface provides straightforward prompts and feedback for both administrators and students. While minimal, it is functional and can be enhanced in future versions with a graphical user interface (GUI) or a web interface.

**Some limitations of the current system include:**

* Lack of sorting or filtering features for the book list
* No timestamps or due dates for borrowed books
* No password encryption (plaintext passwords in users.csv)

Despite these limitations, the system serves its educational purpose well by demonstrating key software engineering concepts such as modularity, abstraction, data persistence, and user interaction.

**Chapter 5.**

**Conclusion**

This Library Management System project has been developed using Java and follows object-oriented design principles to support core functionalities such as book borrowing and returning, user registration, and book information management. The system architecture is modeled using UML class diagrams, with clearly defined classes such as Book, Student, Admin, and Library. These classes demonstrate well-structured relationships through inheritance, associations, and method overriding, contributing to the system’s modularity and maintainability.

In comparison with established library management systems like **Koha**, which have been widely studied and adopted in academic institutions, this system offers a simplified yet practical solution tailored for small to medium-sized organizations or educational use. Drawing on recent studies—such as those by Jabeen (2024) and Ojo (2024)—on the implementation of Koha in university settings, this project adopts similar principles in role-based access control and modular design. For instance, administrative and student users are granted different permissions, simulating a real-world library scenario with distinct user responsibilities.

The development process adhered to key software engineering principles, particularly the **Single Responsibility Principle**, ensuring that each class and method serves a specific purpose. The clarity in design and naming conventions enhances code readability, extensibility, and future maintenance. Moreover, the use of UML diagrams provided a clear visualization of the system structure, which is beneficial for documentation and collaborative development.

In conclusion, this project successfully integrates theoretical research with practical implementation, offering a functional and well-designed library management system. It not only demonstrates the application of object-oriented programming concepts but also reflects best practices in modern software development. The system provides a strong foundation for future enhancements and serves as a valuable tool for educational and demonstrative purposes.To further improve the system and align it with modern library needs, the following enhancements are proposed:

**Advanced Book Sorting and Filtering**: Implementing features such as intelligent book filtering by genre, author, or availability can enhance search efficiency. Li et al. (2019)highlight the potential for using smart filtering algorithms in digital libraries to streamline information retrieval.[9]**Automated Overdue Notifications**: Introducing automated reminders via SMS or email for due dates can reduce late returns. Research by Hamzah et al. (2020) supports the use of SMS-based reminders in improving book return rates.[10]**Secure Password Management**: Transitioning from plaintext passwords to secure storage using encryption or hashing (e.g., bcrypt) would significantly improve user data protection.

**References**

1. Sharifah Nur Amirah S.A., Nor Sa’adah M.N., Nurul Aini N.Y. (2023). *A study on the implementation of Koha cataloguing module in Malaysian academic libraries*. <https://ejournal.um.edu.my/index.php/MJLIS/article/view/38262>
2. Jabeen, M. (2024). *The adoption footprints of Koha as a library management system in university libraries of Pakistan*. <https://journals.sagepub.com/doi/10.1177/01655515231214980>
3. Ojo, T.A. (2024). *Use of Koha Integrated Library System in Public University Libraries in Southwest Nigeria*. <https://digitalcommons.unl.edu/libphilprac/8248>
4. Kulkarni, A.M., Pandiyan, M., Patankar, G.P. (2023). *Smart Usage of Koha: An Open-Source Library Management System*. <https://www.sciencepublishinggroup.com/article/10.11648/j.ajist.20230701.14>
5. Mutungisa, K. (2018). *The Application of Koha Open Source Integrated Library System in Academic Libraries: A Case Study of Bishop Stuart University Mbarara*. <https://idr.kab.ac.ug/items/381dafed-45bd-43a2-8bb5-84a32cf21810>
6. Singh, V. (2019). *Open source integrated library systems migration: Librarians share the lessons learnt*. <https://journals.sagepub.com/doi/10.1177/0961000617709059>
7. Alam, M.J., Mezbah-ul-Islam, M. (2019). *Factors for adoption of OSILS in university libraries of Bangladesh*. <https://journals.sagepub.com/doi/abs/10.1177/0340035219833706>
8. Abdussalam, T.B., Saliu, A.A. (2014). *Using Koha for cataloging and classification: a case study*. <https://www.emerald.com/insight/content/doi/10.1108/lhtn-11-2013-0070/full/html>
9. Li, X., Chen, L., & Wu, M. (2019). Smart filtering methods for enhanced digital library search systems. Library Hi Tech, 37(3), 455-470. <https://doi.org/10.1108/LHT-10-2019-0211>
10. Hamzah, S. N., Asmuni, H., & Sukiman, S. (2020). Library Application and Book Return Schedule Reminder Using Short Message Service (SMS). ResearchGate. [https://www.researchgate.net/publication/338368109](https://www.researchgate.net/publication/338368109" \t "_new)