School of Computer Science

COMP 2120 Object-Oriented Programming Using Java
Summer 2024
Final Report

Group No:

Group 10

Group Name:

BugEyed

Team Members:

Inan Syed

Paul Osuji

Temirlan Rashid

Variant chosen:

Variant 7
CLI-Based Gym Management System

Project Goals: The goal of this project was to develop a command-line interface application for managing a gym system that allows staff to register and manage members, schedule and manage fitness classes, and generate reports on gym usage and membership statistics.

Core Features:

- User Registration: Adding different users to the system (e.g., gym staff, members).
- Membership Management: Add, update, remove, and search for gym members.
- Class Scheduling: Schedule, update, and cancel fitness classes.
- Search Functionality: Search for members, classes, or equipment by various criteria.
- Report Generation: Generate reports on membership renewals and class attendance in any format of your choice (e.g. *.txt).

Technical Specifications:

- Introduction to Java: Utilize Java's basic syntax and control structures to handle user input and display outputs.
- Objects and Classes: Create classes representing members, staff, fitness classes, etc.
- Inheritance and Polymorphism: Use inheritance to differentiate between different types of users (e.g., Gym Staff, Gym Member) with specific functionalities.

- Generics: Use generics to manage lists of members, classes, and equipment items securely.
- Exception Handling: Implement robust exception handling for data input errors and system operational errors.
- Collections and Data Structures: Use Java collections (e.g. ArrayList) for efficient data storage and retrieval.
- Multithreading: Apply multithreading for handling multiple user interactions with the system concurrently.
- I/O Programming: Use file I/O to read/write member and class details from/to files, ensuring data persistence.

Possible Java Classes and Interfaces

- Member.java: Represents a gym member.
- Staff.java: Represents a gym staff member.
- FitnessClass.java: Represents a scheduled fitness class.
- GymManagementSystem.java: Manages members, classes, and staff.
- GymInterface.java: Handles the CLI interactions.

Development Steps:

1. Design the System:
    - Define the data models (staff, fitness classes, etc.).
    - Outline the functionalities and interfaces for user interactions.
2. Implement Core Functionalities:
    - Set up the basic user interface for command input and result display.
    - Implement classes and methods for managing classes and users.
3. Integrate Advanced Features:
    - Add features for searching, reporting, and multithreading support.
    - Ensure robustness via comprehensive error handling.
4. Testing and Refinement:
    - Test the system for various use cases to ensure stability and performance.
    - Refine the code based on testing outcomes and feedback.

Outcome:

This project will help students consolidate their understanding of Java and OOP principles through a practical application that requires a combination of data management, user interaction, and performance considerations.

This Gym Management System is designed to be a comprehensive final project that incorporates all the essential OOP features and Java programming techniques, making it an ideal culmination of a Java OOP course.

Group member contribution:

Names of group members and their contribution with the names of java files in your source code:

Inan Syed:
Java Files: Equipment.java, FitnessClass.java, GymInterface.java, GymManagementSystem.java, InvalidUserException.java, Main.java, Member.java, Staff.java, User.java
Contributions:

- Developed core system functionality.
- Implemented I/O programming for the project, including a Load & Save method utilizing multithreading for efficient data handling.
- Created a toggle-autosave function to ensure consistent background data saving every 10 seconds.
- Contributed to the logic for generating comprehensive reports
- Participated in code reviews to maintain code quality and consistency

Paul Osuji:
Java Files: Equipment.java, FitnessClass.java, GymInterface.java, GymManagementSystem.java, InvalidUserException.java, Main.java, Member.java, Staff.java, User.java
Contributions:

- Implemented the Command-Line Interface to manage all user interactions.
- Contributed to the logic in the GymManagement System for handling Members, Staff, Equipment, and Fitness objects (including adding, removing, updating, and searching).
- Implemented the login and logout functionality to ensure basic system security.
- Implemented the Abstract User class, serving as a common base for other classes.
- Created object templates for Member, Staff, Equipment, and FitnessClass

Temirlan Rashid:
Java Files: Equipment.java, FitnessClass.java, GymInterface.java, GymManagementSystem.java, InvalidUserException.java, Main.java, Member.java, Staff.java, User.java

Contributions:

- Implemented the InvalidUserException class as a custom exception to manage errors across the system.
- Contributed to the logic for unique ID generation and input validation (such as regex for email and contact phone number).
- Developed the logic behind polymorphism in the system.
- Conducted extensive testing to identify errors and ensure system correctness and communicated findings to the team.
- Participated in code reviews to maintain code quality and consistency

Description of topics used for each feature developed:

Inheritance and Polymorphism:

User.java

```
abstract class User implements Serializable {
```

```
 //Overridden method to get basic user information.
 public String getUserInfo() {
     return "ID: " +id + ", Name: " + name;
 }
```

Member.java

```
//Member is a class that represents a member in our gym management system.
//It extends the User superclass inheriting it properties, methods  and
//implements the Serializable interface for object serialization.
class Member extends User implements Serializable{
```

```
  // Overridden method to get detailed member information
    @Override
   public String getUserInfo() {
         return  getId() + ", Name: " + getName() +
                 ", Membership Type: " + membershipType +
                 ", Contact Number: " + getContactNumber() +
                 ", Email: " + getEmail();
   }
```

Staff.java

```
//The Staff class represents a staff member in our gym management system.
//It extends the User superclass, inheriting its properties and methods.
//Each Staff object represents a unique staff member in the system
public class Staff extends User implements Serializable{
```

```
   @Override
   public String getUserInfo() {
       return getId() + ", Name: " + getName() +
              ", Role: " + role +
              ", Contact Number: " + getContactNumber() +
              ", Email: " + getEmail();
   }
```

Description:

In our gym management system project, inheritance and polymorphism are effectively demonstrated through the User, Member, and Staff classes. The User class is an abstract base class that implements the Serializable interface, providing common attributes such as id, name, contactNumber, and email, along with methods to manage these attributes. Both the Member and Staff classes extend the User class, inheriting its properties and methods. The Member class adds a specific attribute membershipType, while the Staff class includes role and password. Each of these subclasses overrides the getUserInfo() method to provide detailed information specific to members and staff, showcasing polymorphism by allowing different behaviors through the same method call. This use of inheritance and polymorphism simplifies code maintenance and enhances scalability by promoting code reuse and flexibility.

Generics:

GymManagementSystem.java

```
// Method to read and display class summary from file
private void printClassSummary() {
    // Create HashMaps to store counts of instructors, rooms, and times
    //Generic used
    Map<String, Integer> instructorCount = new HashMap<>();
    Map<String, Integer> roomCount = new HashMap<>();
    Map<String, Integer> timeCount = new HashMap<>();
```

```
// Method to read and display equipment summary from file
private void printEquipmentSummary() {
    // Create a HashMap to store counts of equipment
    Map<String, Integer> equipmentCount = new HashMap<>();
    int lowStockThreshold = 5; //threshold value
    int highStockThreshold = 10; //threshold value
    List<String> lowStockEquipment = new ArrayList<>();
    List<String> highStockEquipment = new ArrayList<>();
```

Description:

Generics are fundamental to achieving type safety and flexibility in handling collections of various objects. By leveraging generics, we can write more general and reusable code. This is clearly demonstrated in the printClassSummary() and printEquipmentSummary() methods of the GymManagementSystem class. Within these methods, HashMap<String, Integer> is employed to track counts of instructors, rooms, and times, while ArrayList<String> is used to manage lists of equipment with low and high stock levels. Utilizing generics ensures that these collections are restricted to the specified types, thereby minimizing the risk of runtime errors and improving code readability and maintainability. This method provides a dependable way to manage various types of data consistently throughout the application, enabling us to efficiently organize and process large amounts of information in a type-safe manner.

Exception Handling:

InvalidUserException.java

```
// This is a custom exception class for handling invalid user input.
public class InvalidUserException extends Exception {
    private static final long serialVersionUID = 1L; // Unique ID for serialization and deserialization
    // The constructor takes a string message as a parameter.
    public InvalidUserException(String message) {
        //passed to the superclass constructor (Exception).
        super(message);
    }
}

// This is a custom exception class for handling invalid class operations.
class InvalidClassException extends Exception { // Unique ID for serialization and deserialization
    private static final long serialVersionUID = 1L;
    public InvalidClassException(String message) {
        super(message);
    }
}
```

Try and Catch Block

```
   try {
       displayMainMenu();
       int choice = scanner.nextInt(); // get user choice
       scanner.nextLine(); // consume newline
       switch(choice) { // call appropriate method base on user choice
           case 1:
```

```
 }catch(InputMismatchException input) {
     System.out.println("\nInvalid input. Please enter a number.\n");
     scanner.nextLine(); // clear invalid input
 }
```

```
}catch(InvalidUserException error) {
    System.out.println("\nError registering member: " + error.getMessage()+"\n");
    //error.printStackTrace();
}
```

```
   // Method to Cancel a Class
   public void cancelClass(String classId) throws InvalidClassException {
       // Find the class with the given ID
```

```
       }
   }
   throw new InvalidClassException("\nClass with ID " + classId + " not found.");
```

```
  // Method to validate user input
  public static String validateInput(Scanner scan, String prompt, String errorMessage, String checkRegres) {
      System.out.print(prompt);
      String input = scan.nextLine();

      while(!input.matches(checkRegres)) {
          System.out.println(errorMessage);
          System.out.print(prompt);
          input = scan.nextLine();
      }
      return input;
  }
```

Description:

In this project, exception handling plays a crucial role in maintaining the integrity and user-friendliness of the application. We have implemented a custom exception class, InvalidUserException, defined in InvalidUserException.java. This class is used throughout the program to manage various user input errors, such as when a member is not found or when a member's contact information or email is invalid. The project also includes the InvalidClassException, another custom exception class that handles errors like a scheduled class not being found. The try-and-catch blocks are strategically used, particularly in GymInterface.java, to handle user errors such as input mismatches. These blocks ensure that users receive appropriate messages with guidance on correcting their inputs, enhancing the overall user experience. To further ensure data integrity, we have implemented a method called validInput, which sanitizes user inputs, ensuring that only valid data is collected. This approach prevents the entry of erroneous data. Overall, these exception-handling mechanisms prevent the program from crashing and provide users with clear, informative messages when incorrect input is entered, thereby ensuring a smooth and reliable application performance.

Collections and Data Structures (e.g. ArrayList, LinkedList, etc.):

GymManagementSystem.java

```
public class GymManagementSystem {
    // LinkedLists to store members, staff, fitness classes, and equipment
    //Generic used
    private LinkedList<Member> members;
    private LinkedList<Staff> staffs;
    private LinkedList<FitnessClass> fitnessClasses;
    private LinkedList<Equipment> equipmentInventory;
    private Thread saveThread;
    private volatile boolean shouldSave = true;
    // Default constructor that initializes each LinkedList
    public GymManagementSystem() {
        members = new LinkedList<>();
        staffs = new LinkedList<>();
        fitnessClasses = new LinkedList<>();
        equipmentInventory = new LinkedList<>();
```

```
// Method to read and display class summary from file
private void printClassSummary() {
    // Create HashMaps to store counts of instructors, rooms, and times
    //Generic used
    Map<String, Integer> instructorCount = new HashMap<>();
    Map<String, Integer> roomCount = new HashMap<>();
    Map<String, Integer> timeCount = new HashMap<>();
```

```
// Method to read and display equipment summary from file
private void printEquipmentSummary() {
    // Create a HashMap to store counts of equipment
    Map<String, Integer> equipmentCount = new HashMap<>();
    int lowStockThreshold = 5; //threshold value
    int highStockThreshold = 10; //threshold value
    List<String> lowStockEquipment = new ArrayList<>();
    List<String> highStockEquipment = new ArrayList<>();
```

Description:

Collections and data structures are essential for organizing and managing data efficiently. The GymManagementSystem class uses LinkedList to store collections of Member, Staff, FitnessClass, and Equipment objects. This data structure is chosen for its efficient insertion and deletion operations, which are crucial for managing dynamic datasets. Additionally, HashMap and ArrayList are used in methods like printClassSummary() and printEquipmentSummary() to handle counts and lists, respectively. These collections ensure type safety and flexibility, enabling the application to manage various types of data consistently and efficiently. By leveraging these data structures, we maintain a well-organized and performant system capable of scaling with the needs of the gym management application.

Multithreading:

GymManagementSystem.java

```
public GymManagementSystem() {
    members = new LinkedList<>();
    staffs = new LinkedList<>();
    fitnessClasses = new LinkedList<>();
    equipmentInventory = new LinkedList<>();
    //multithreading
    saveThread = new Thread(() -> {
        while (shouldSave) {
            try {
                Thread.sleep(10000); // Sleep for 10 seconds
                saveData();
            } catch (InterruptedException e) {
                // Handle interruption if needed
                break; // Exit the loop if interrupted
            }
        }
    });
    saveThread.start(); // Start the saving thread
}
```

```java
public void saveData() {
    //String filename = "gym_data.ser";
    new Thread(() -> {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("gym_data.ser"))) {
        // Write the LinkedLists to the file
        oos.writeObject(members);
        oos.writeObject(staffs);
        oos.writeObject(fitnessClasses);
        oos.writeObject(equipmentInventory);

        //save the unique IDs
        oos.writeObject(Member.getNextUniqueId());
        oos.writeObject(Staff.getNextUniqueId());
        oos.writeObject(FitnessClass.getNextUniqueId());
        //System.out.println("\nData from " + filename + " Saved successfully.\n");
    } catch (IOException e) {
        System.out.println("Error saving data: " + e.getMessage());
    }
    }).start();

}
public void toggleAutoSave(boolean enableAutoSave) {
    shouldSave = enableAutoSave;
}
// Method to loadData from a file
@SuppressWarnings("unchecked")
//  I/O Programming
public void loadData() {
    //String filename = "gym_data.ser";
    new Thread(() -> {
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("gym_data.ser"))) {
        // Read the LinkedLists from the file
        members = (LinkedList<Member>) ois.readObject();
        staffs = (LinkedList<Staff>) ois.readObject();
        fitnessClasses = (LinkedList<FitnessClass>) ois.readObject();
        equipmentInventory = (LinkedList<Equipment>) ois.readObject();
        Member.setUniqueID((int) ois.readObject());
        Staff.setUniqueID((int) ois.readObject());
        FitnessClass.setUniqueID((int) ois.readObject());
        //System.out.println("\nData from " + filename + " loaded successfully.\n");
    } catch (IOException | ClassNotFoundException e) {
        System.out.println("Error loading data: " + e.getMessage());
    }
    }).start();
}
```

Description:

In our gym management system project, we implemented multithreading to boost performance and keep the application responsive during time-consuming tasks like data saving and loading. The GymManagementSystem class employs multithreading by creating a dedicated thread (saveThread) that periodically saves the system data every 10 seconds. This automatic saving mechanism acts as a refresh of the data, ensuring data integrity and minimizing the risk of data loss without interrupting the main application flow. Additionally, the saveData() and loadData() methods are executed in separate threads to handle I/O-intensive operations concurrently with user interactions. By using multithreading, we prevent the main application from becoming unresponsive during these critical operations, thus providing a smooth and efficient user experience. This approach ensures that background tasks do not hinder the application's performance, allowing users to continue their activities seamlessly.

I/O Programming:

GymManagementSystem.java

saveData() Method

```
// Method to saveData to a file
//I/O Programming
public void saveData() {
    //String filename = "gym_data.ser";
    try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("gym_data.ser"))) {
        // Write the LinkedLists to the file
        oos.writeObject(members);
        oos.writeObject(staffs);
        oos.writeObject(fitnessClasses);
        oos.writeObject(equipmentInventory);

        //save the unique IDs
        oos.writeObject(Member.getNextUniqueId());
        oos.writeObject(Staff.getNextUniqueId());
        oos.writeObject(FitnessClass.getNextUniqueId());
        //System.out.println("\nData from " + filename + " Saved successfully.\n");
    } catch (IOException e) {
        System.out.println("Error saving data: " + e.getMessage());
    }
}
```

loadData() Method

```
    // Method to loadData from a file
    // I/O Programming
    public void loadData() {
        //String filename = "gym_data.ser";
        new Thread(() -> {
            try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("gym_data.ser"))) {
            // Read the LinkedLists from the file
            members = (LinkedList<Member>) ois.readObject();
            staffs = (LinkedList<Staff>) ois.readObject();
            fitnessClasses = (LinkedList<FitnessClass>) ois.readObject();
            equipmentInventory = (LinkedList<Equipment>) ois.readObject();
            Member.setUniqueID((int) ois.readObject());
            Staff.setUniqueID((int) ois.readObject());
            FitnessClass.setUniqueID((int) ois.readObject());
            //System.out.println("\nData from " + filename + " loaded successfully.\n");
        } catch (IOException | ClassNotFoundException e) {
            System.out.println("Error loading data: " + e.getMessage());
        }
        }).start();
    }
```

generateReport() Method

```
public void generateReport(String reportType, String fileName) {
    try (PrintWriter writer = new PrintWriter(new FileWriter(fileName))) {
        // Generate a report based on the report type
        switch (reportType.toLowerCase()) {
            case "membership":
                writer.println("Membership Report");
                writer.println("-------------------");
                for (Member member : members) {
                    writer.println(member.getUserInfo());

                }
                System.out.println("\nLoading Report details from " + fileName);
                delayMessage(5000);
                printMembershipSummary();
                delayMessage(2000);
                System.out.println("\nReport generated successfully.");
                break;
            case "class":
                writer.println("Class Attendance Report");
                writer.println("------------------------");
                for (FitnessClass fitnessClass : fitnessClasses) {
                    writer.println(fitnessClass.toString());
                }
                System.out.println("\nLoading Report details from " + fileName);
                delayMessage(5000);
                printClassSummary();
                delayMessage(2000);
                System.out.println("\nReport generated successfully.");
                break;
            case "equipment":
                writer.println("Equipment Inventory Report");
                writer.println("---------------------------");
                for (Equipment equipment : equipmentInventory) {
                    writer.println(equipment.toString());
                }
                System.out.println("\nLoading Report details from " + fileName);
                delayMessage(5000);
                printEquipmentSummary();
                delayMessage(2000);
                System.out.println("\nReport generated successfully.");
                break;
            default:
                System.out.println("\nInvalid report type.");
                return;
        }

    } catch (IOException e) {
        System.out.println("\nError generating report: " + e.getMessage());
    }
}
```

Description:

The saveData method utilizes I/O programming to save data to a file, ensuring no data is lost. The saveData() method captures the current state of the system, including members, staff, fitness classes, and equipment, and writes this information to a file, safeguarding all essential data. When the program restarts, the loadData() method retrieves this data, enabling immediate access to the preserved state and ensuring a smooth user experience. Furthermore, the generateReport() method produces comprehensive reports and saves them to files, which can be utilized for detailed analysis. These I/O processes are crucial for the application's efficient and reliable data management, handling tasks such as periodic data saving to avert data loss, data restoration at startup to maintain continuity, and report generation for administrative and analytical needs. This comprehensive approach ensures that critical operations are conducted seamlessly, allowing users to resume their activities effortlessly even after restarting the application.

Conclusion:

Throughout the duration of this gym management system project, we have been able to apply a variety of advanced Java programming concepts that we learned in our lectures, practical sessions, and lab exercises. The project required us to develop a comprehensive and efficient system by utilizing a combination of inheritance and polymorphism to construct a versatile class hierarchy that includes `User`, `Member`, and `Staff` classes, which allowed us to reuse and extend our codebase effectively. By implementing generics, we were able to ensure that our collections handled data safely and flexibly, reducing runtime errors and improving code clarity.

Exception handling was another critical aspect, as it helped maintain the system's stability and provided users with clear instructions for correcting errors. Our use of custom exceptions such as `InvalidUserException` and `InvalidClassException` helped manage and resolve user input errors seamlessly. We employed various collections and data structures, including `LinkedList`, `HashMap`, and `ArrayList`, to manage dynamic data sets, which ensured our system's performance and scalability.

Multithreading was a key feature that kept our application responsive, allowing background tasks like periodic data saving to occur without interrupting the main application processes. This approach ensured that data integrity was maintained while the system remained efficient. Our I/O programming techniques, including the use of `saveData()`, `loadData()`, and `generateReport()` methods, were fundamental in managing data persistence and facilitating smooth data retrieval and detailed report generation.

Overall, this project provided us with hands-on experience in applying complex Java programming concepts, reinforcing their practical applications in creating reliable and user-friendly systems. The knowledge and skills we gained from our coursework were crucial in overcoming the challenges we faced and successfully completing this project. This experience has prepared us well for future software development tasks, equipping us with a solid foundation in advanced Java programming.

References:

Java Platform SE 8. (2020). Oracle.com. https://docs.oracle.com/javase/8/docs/api/
Lecture 2: Objects and Classes in Java
Lecture 3: Inheritance, Polymorphism, and Generics
Lecture 4: Exception Handling
Lecture 6: Data Structures and Collections – 2
Lecture 8: I/O Programming