# GROUP ASSIGNMENT

## TECHNOLOGY PARK MALAYSIA

## CT038-3.5-2-OODJ

## OBJECT-ORIENTED DEVELOPMENT WITH JAVA

**APU2F2102CS/CS(DA)/CS(DF)/CS(IS)/IT(BIS)/IT(CC)/IT(MBT)/CE/ IT(IOT)/IT(NC)/SE/IT/IT(ISS)/CS(CYB))**
**UC2F2102IT(ISS)/CS/CS(DA)/CS(IS)/IT(BIS)/SE**

**HAND OUT DATE: 22 MARCH 2021**

**HAND IN DATE: 28 MAY 2021**

**WEIGHTAGE: 50%**

---

**INSTRUCTIONS TO CANDIDATES:**

1. **Submit your assignment at the administrative counter**
2. **Students are advised to underpin their answers with the use of references (cited using the Harvard Name System of Referencing)**
3. **Late submission will be awarded zero (0) unless Extenuating Circumstances (EC) are upheld**
4. **Cases of plagiarism will be penalized**
5. **The assignment should be bound in an appropriate style (comb bound or stapled)**

6 **Where the assignment should be submitted in both hardcopy and softcopy, the softcopy of the written assignment and source code (where appropriate) should be on a CD in an envelope / CD cover and attached to the hardcopy**

7 **You must obtain 50% overall to pass this module.**

| **Group Member :** | **TP Number :** |
|---|---|
| **Ong Cheng Kei** | **TP055620** |
| **Kong Kei Zhong** | **TP055016** |

# Table of Contents

# Introduction

REAL CHAMPIONS SPORTS ACADEMY is a multi-branch Sports Centre that is growing rapidly in terms of popularity and success. This is due to their wide range of selections when coming to the availability of different sport lessons and coaches. However, one arising problem faced by the academy is their inefficient records filling system. With a large volume of students, coaches, and sports records to manage, it can be stressful and difficult to do so using a traditional method for record-keeping. So, this documentation discusses the designing process and as well as the Object-Oriented Programming concepts applied when creating a fully functional and digitized management system for the sports academy using the Java programming language. The end goal is to create a system where the admins can use to systematically manage the records in the sports center, and students can use to manage their own records and track their sports details including its coaches and schedules, and lastly for unregistered guest to provide them information about the available sports lessons provided at the academy as well as allowing them to register as a student.

## Assumptions

**a. Admins are only responsible for managing records in their own branch**

In the system, it was programmed so that admins of specific sport centre branches can only access, view, and modify the records in their respective sport centres. This design was implemented with the assumption that every sport centre has its own admin and their access to records should not overlap as mismanaging records in one specific sport centre may potentially cause complications to another.

**b. Unregistered guest must have their account registration approved by an admin**

When an unregistered student registers their account, it has to first be approved by the respective sport centre admin before they are officially registered as a student in the system. This comes with the assumption that certain guests might create multiple similar accounts either accidentally or intentionally. Regardless, the guest will have to wait for the admin to approve their account creation before they can login to the sports centre system as a registered student.

**c. Records with the same name are not allowed to exist**

When creating a record for either students, coaches, or even sports, records that are using names that already exist in the past records will not be allowed and the program will throw an error code. This also applies to unregistered guests, as they are not allowed to request for an account created using a name that was already previously used. Because, it is assumed that admins or guests may try to create a new record without realising that it already exists and allowing the duplicate records to be created will cause a huge number of technical complications to the system in the future.

**d. Error in user input will be a common occurrence**

Another assumption that was made when designing the system is that users are most likely to enter invalid or irrelevant inputs at times of using the program. Therefore, the system is programmed to always check for the format and validity of user input in every field of the

program to ensure that only inputs using the approved formats will be saved into the system records.

Design Diagrams

UML Use Case Diagram

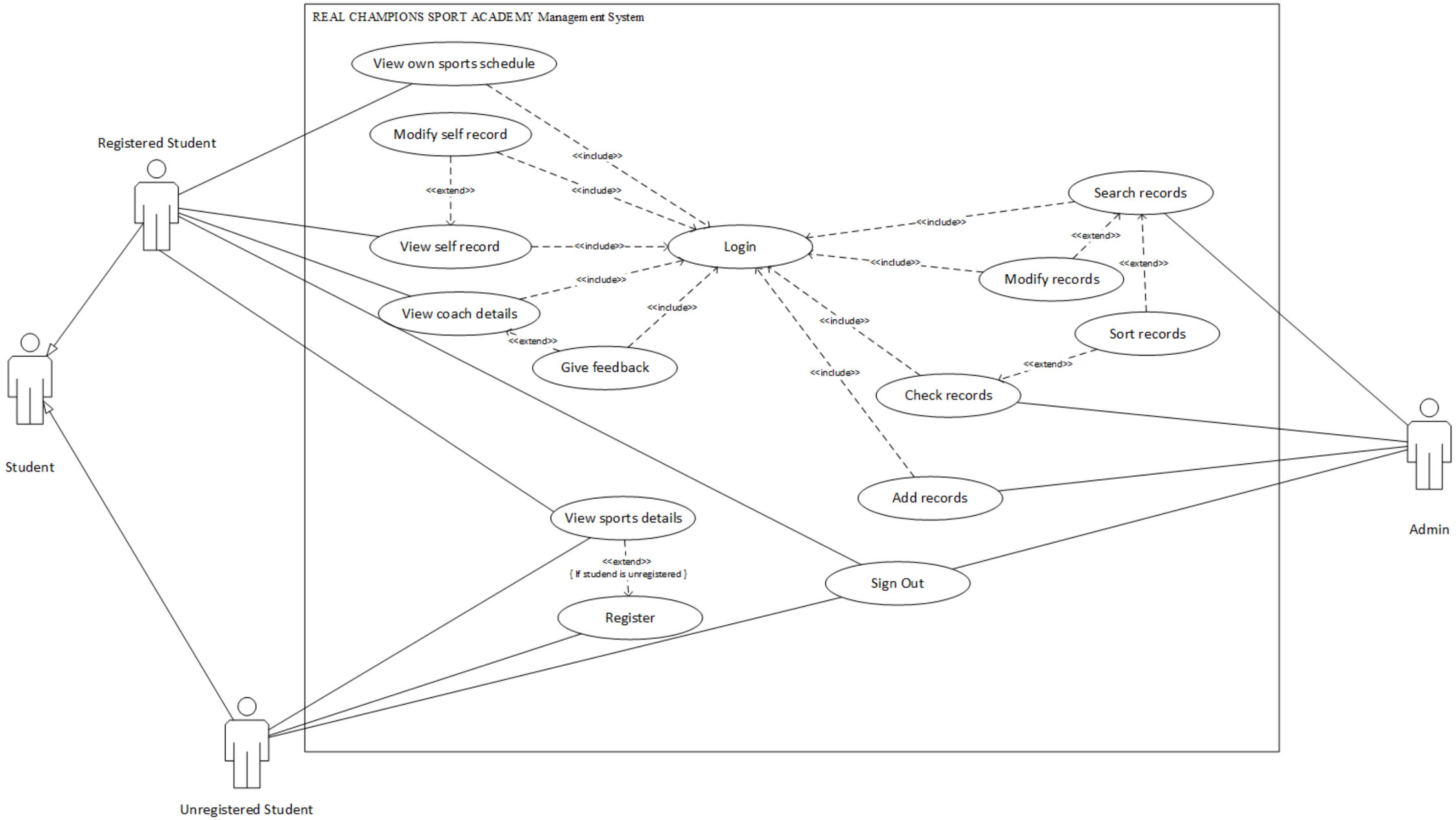



*Figure 1 : UML Use Case Diagram for REAL CHAMPIONS SPORT ACADEMY Management System*

UML Class Diagram

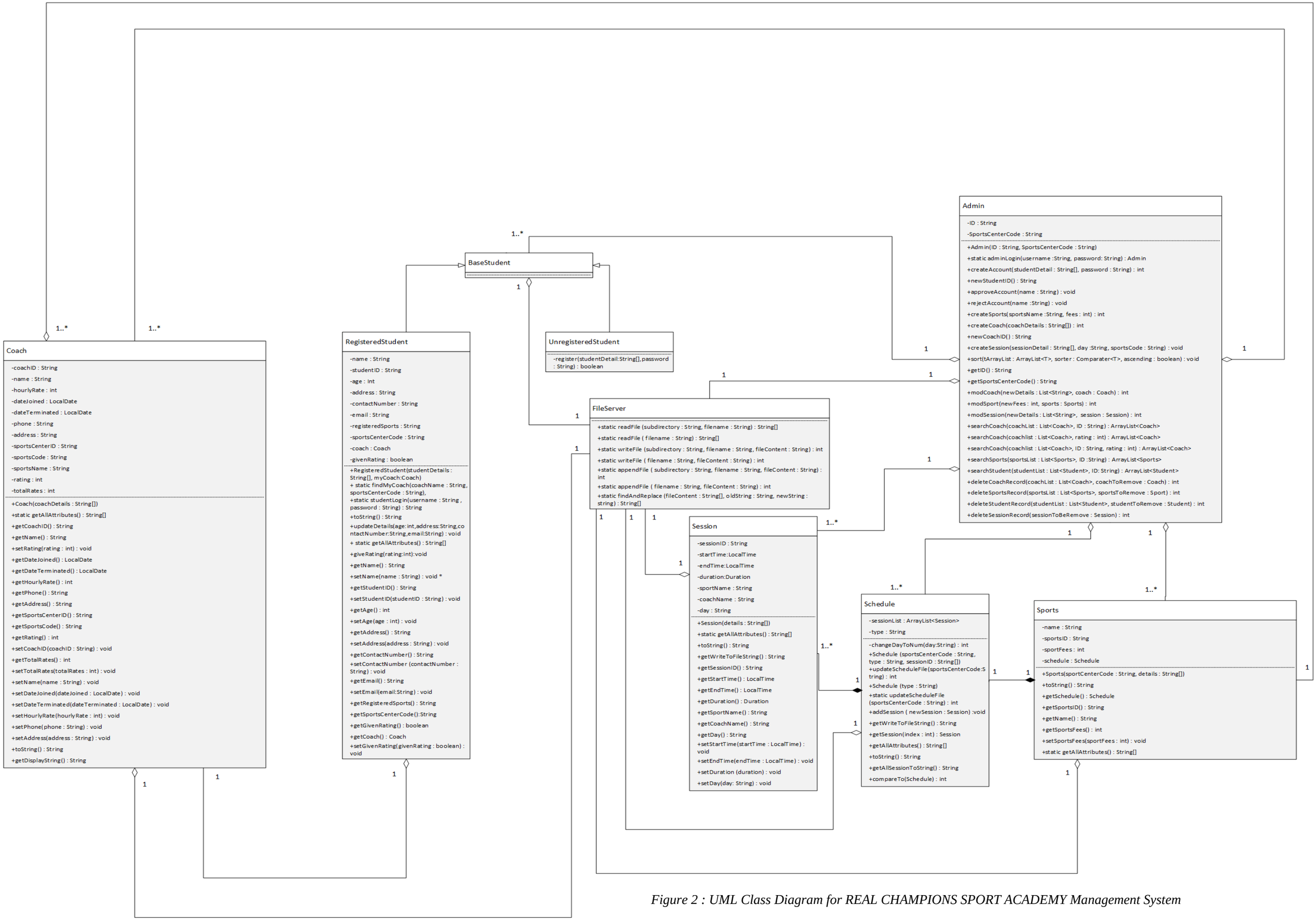



*Figure 2 : UML Class Diagram for REAL CHAMPIONS SPORT ACADEMY Management System*

## Object-Oriented Concepts

### Encapsulation

The concept of encapsulation can be described as packaging data variables and their corresponding methods together as one component (tutorialspoint, n.d.). What encapsulation aims to achieve is to set certain attributes of classes as private and non-accessible to all other class by default, and they can only be changed or viewed using public getters and setters method where extra layers of authentication or restriction can be added to them to ensure that only specific verified classes can access them. An example of the implementation of encapsulation in the program with explanations can be seen below.

```
public class Student extends BaseStudent {
    private String name;
    private String studentID;
    private int age;
    private String address;
    private String contactNumber;
    private String email;
    private final String registeredSports;
    private final String sportsCenterCode;
    private final Coach coach;
    private boolean givenRating;
```

*Figure 3: Attributes of Student class with private access modifier*

As shown in the image, all the attributes within the student class are set to private using access modifiers. This is so that no class other than student itself can access its attributes by default which is the first part of encapsulation.

```java
/*----------Getters and Setters----------*/

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getStudentID() {
    return studentID;
}

public void setStudentID(String studentID) {
    this.studentID = studentID;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}
```

*Figure 4: Examples of getters and setters in Student class*

With the private attributes within the student class, they all have their respective getters and setters as well. As mentioned above, this method is used so that an extra layer of authentication can be included to ensure only the allowed classes can excess, change, and view this specific information, which suffices as code encapsulation. With that explained, additional examples of encapsulation implementation will be displayed below.

```java
public class Coach {
    private String coachID;
    private String name;
    private int hourlyRate;
    private LocalDate dateJoined;
    private LocalDate dateTerminated;
    private String phone;
    private String address;
    private final String sportsCenterID;
    private final String sportsCode;
    private final String sportsName;
    private int rating;
    private int totalRates;
```

*Figure 5: Attributes of Coach class*

```java
/*----------Getters and Setters----------*/

public String getCoachID() { return coachID; }

public String getName() { return name; }

public void setRating(int rating) { this.rating = rating; }

public LocalDate getDateJoined() { return dateJoined; }

public LocalDate getDateTerminated() { return dateTerminated; }

public int getHourlyRate() { return hourlyRate; }

public String getPhone() { return phone; }

public String getAddress() { return address; }

public String getSportsCenterID() { return sportsCenterID; }

public String getSportsCode() { return sportsCode; }

public int getRating() { return rating; }

public void setCoachID(String coachID) { this.coachID = coachID; }

public int getTotalRates() { return totalRates; }

public void setTotalRates(int totalRates) { this.totalRates = totalRates; }
```

*Figure 6: Examples of getters and setters in Coach class*

```java
public class Sports {
    private final String name;
    private final String sportsID;
    private int sportFees;
    private Schedule schedule;
```

*Figure 7: Attributes of Sports class*

```java
/*----------Getters and Setters----------*/

@Override
public String toString() { return name + "|" + sportsID + "|" + sportFees ; }

public Schedule getSchedule() { return schedule; }

public String getSportsID (){return sportsID;}
public String getName() {return name;}

public int getSportFees() { return sportFees; }
public void setSportFees(int sportFees) { this.sportFees = sportFees; }
```

*Figure 8: Getters and setters in Sports class*

## Abstraction

Abstraction is one of the four pillars of object-oriented programming concepts. Abstraction essentially means hiding unnecessary implementation details from the user. For example, if a student wants to give a feedback rating to their coach. They do not need to worry about how their rating is calculated, and stored, etc. All they need to know is that what rating they want to give to their coach. Through this, complex logic can be implemented with a layer of abstraction and make things simpler, and that is the beauty of abstraction.

Abstraction has been implemented in this assignment as shown in the examples provided below.

```java
@Override
public int compareTo(Schedule o) {
    if (changeDayToNum(this.type) == changeDayToNum(o.type)){
        return this.type.compareTo(o.type);
    } else
        return changeDayToNum(this.type) - changeDayToNum(o.type);
}
```

*Figure 9: Java code snippet*

The figure above shows a public method implemented in the schedule class which helps to sort schedule objects based on their values in their “type” field. For example, schedule objects are typically sorted from Sunday to Saturday, and followed by sports names in alphabetical order. In the above method, another method called changeDayToNum is called. This method essentially returns an integer based on which day is passed into it as a string value, and the integer will be representing the number for the day when ordering. The changeDayToNum method is shown below.

```java
/*
    Method : changeDayToNum
    Description : Assign a number to each day to facilitate the sorting method based on days
    Parameter : day (String)
    Return    : Integer (represents the day Sunday == 1, etc...)
 */
private int changeDayToNum (String day){
    if (day.equalsIgnoreCase( anotherString: "monday"))
        return 1;
    else if (day.equalsIgnoreCase( anotherString: "tuesday"))
        return 2;
    else if (day.equalsIgnoreCase( anotherString: "wednesday"))
        return 3;
    else if (day.equalsIgnoreCase( anotherString: "thursday"))
        return 4;
    else if (day.equalsIgnoreCase( anotherString: "friday"))
        return 5;
    else if (day.equalsIgnoreCase( anotherString: "saturday"))
        return 6;
    else if (day.equalsIgnoreCase( anotherString: "sunday"))
        return 0;
    else
        return 7;
}
```

*Figure 10: Java code snippet*

In the figure above, it can be seen that changeDayToNum is a private method in the schedule class. The key point here is that by setting the access modifier of changeDayToNum method to private, the implementation details are hidden away from other classes. For instance, if other classes wanted to sort schedule objects, they do not have to worry about how to sort them instead, they can rely on the hidden implementation details to do their job.

Another example is the creation of FileServer class. FileServer is an abstract class with static methods that facilitate the File input and output operations, such as writing, reading, and appending to files. The code is shown below.

```java
/*
    Description : FileServer class is responsible for fileIO operations. Class should be used without instantiation.
 */
public abstract class FileServer {
    /*
        Method Name : readFile
        Return : Array of strings that represents each line in the file.
     */
    public static String[] readFile (String subDirectory, String fileName){
        String fileContent = "";
        File file = null;

        if (subDirectory.isEmpty()) {
             file = new File(fileName);
        }
        else {
             file = new File(subDirectory, fileName);
        }
        try {
            FileReader fr = new FileReader(file);
            int charValue = fr.read();
            while (charValue != -1){
                fileContent += (char) charValue ;
                charValue = fr.read();
            }
            fr.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        if (fileContent.isEmpty()){
            return new String[0];
        }
        return fileContent.split( regex: "\\r?\\n");
    }
```

*Figure 11: Java Code Snippet*

In the code snippet above, it shows the static “readFile” method from the FileServer class. It returns a string array that contains all the file contents.

```java
public static int writeFile (String subDirectory, String fileName, String fileContent){
    File file = null;
    if (subDirectory.isEmpty()) {
        file = new File(fileName);
    }
    else {
        file = new File(subDirectory, fileName);
    }
    try {
        FileWriter fw = new FileWriter(file);
        BufferedWriter writer = new BufferedWriter(fw);
        writer.write(fileContent);
        writer.close();
        return 0;
    } catch (IOException e){
        e.printStackTrace();
    }
    return 1;
}
```

*Figure 12 : Java Code Snippet*

The code snippet above shows the static method “writeFile” from the FileServer class, which writes content to a file.

```java
/*
    Method Name : appendFile
    Return        : 0 represents append to file is successful
                    1 represents append to file is unsuccessful
 */
public static int appendFile (String subDirectory, String fileName, String fileContent){
    File file = null;
    if (subDirectory.isEmpty()) {
        file = new File(fileName);
    }
    else {
        file = new File(subDirectory, fileName);
    }
    try {
        FileWriter fw = new FileWriter(file, append: true);
        BufferedWriter writer = new BufferedWriter(fw);
        writer.write(fileContent);
        writer.close();
        return 0;
    } catch (IOException e){
        e.printStackTrace();
    }
    return 1;
}
```

*Figure 13 : Java Code Snippet*

The code snippet above shows the static method “appendFile” from the FileServer. It helps to append a line to a specific file.

Through the FileServer class shown above, it hides away the implementation details of writing text to text files or appending text to text files. File IO errors such as file not found, are all handled by the FileServer class, and outside class that wants to write text to files, do not need to worry about all these implementation details. As such, FileServer class provides a clean and easy interface to interacts with other objects. The FileServer interface can be seen in the figure below.

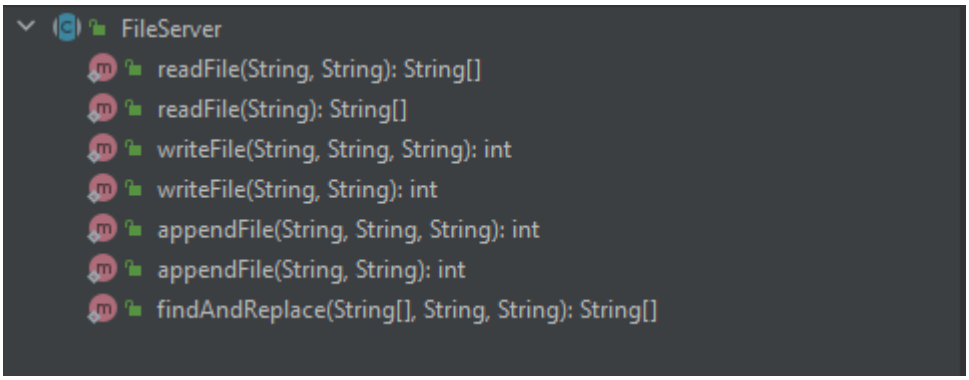



*Figure 14: FileServer interface to other classes*

Therefore, other classes can just call the write file method to write text to files and the read file method to read files as shown below.

```
/*
Method      : modSports
Description : Modify a sport details
Parameter   : newFees (updated sport fees) , sports object with fees to be modified
Return      : 0 -- modify success
              1 -- modify failed
*/
public int modSports (Integer newFees, Sports sports){
    String oldString = sports.toString();
    sports.setSportFees(newFees);
    String[] currentFileContent = FileServer.readFile(SportsCenterCode, fileName: "Sports.txt");
    FileServer.findAndReplace(currentFileContent,oldString,sports.toString());
    return FileServer.writeFile(SportsCenterCode, fileName: "Sports.txt", fileContent: String.join( delimiter: "\n",currentFileContent)+"\n");
}
```

*Figure 15 : Java Code Snippet*

modSports method from Admin class uses FileServer.writeFile() to write text to “sports.txt”. It also uses FileServer.readFile() to read file from “sports.txt”.

```java
public Schedule (String sportsCenterCode, String type, String[] sessionID){
    String[] sessionFile = FileServer.readFile(sportsCenterCode, fileName: "Session.txt");
    for (String ID : sessionID){
        for (String line : sessionFile){
            String[] tokens = line.split( regex: "\\|");
            if (tokens[1].equals(ID)) {
                sessionList.add(new Session(tokens));
                break;
            }
        }
    }
    this.type = type;
}
```

*Figure 16 : Java Code Snippet*

Schedule class constructor uses FileServer.readFile() to read file content from “session.txt”.

Another example of abstraction can be also found in the GUI classes. For instance, the DisplayAllRecords class.

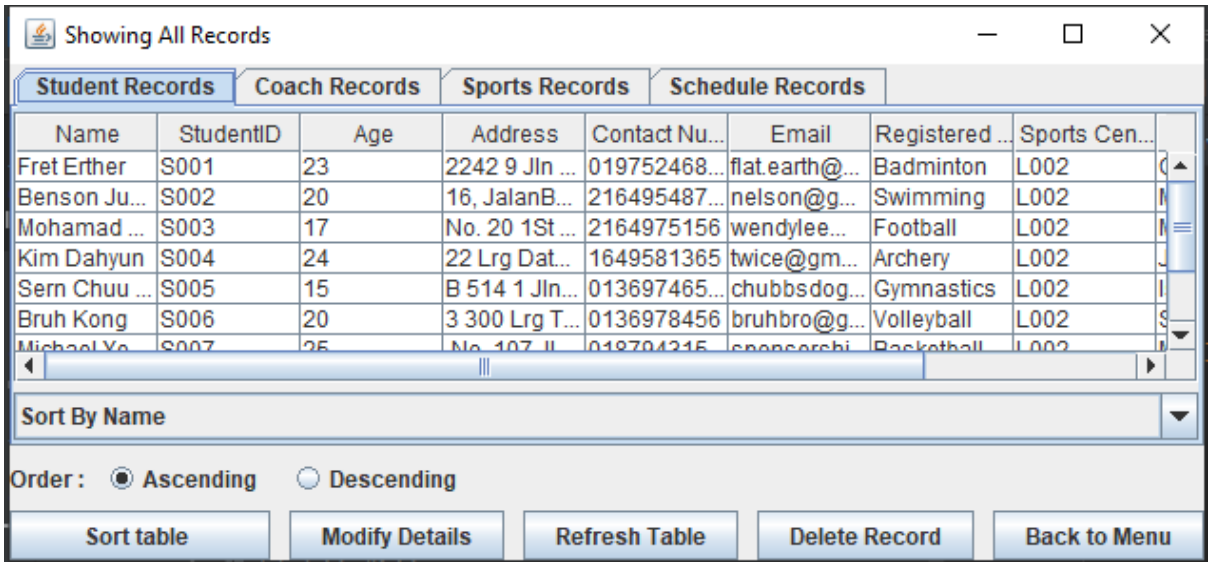



*Figure 17 : GUI for displaying all records*

In the figure above, the display all records screen consists of four panels that display all the records of students, coaches, sports, and schedule. To provide an abstraction over each panel or tab, inner classes are implemented, so that each panel is handled by a panel manager which will be shown below.

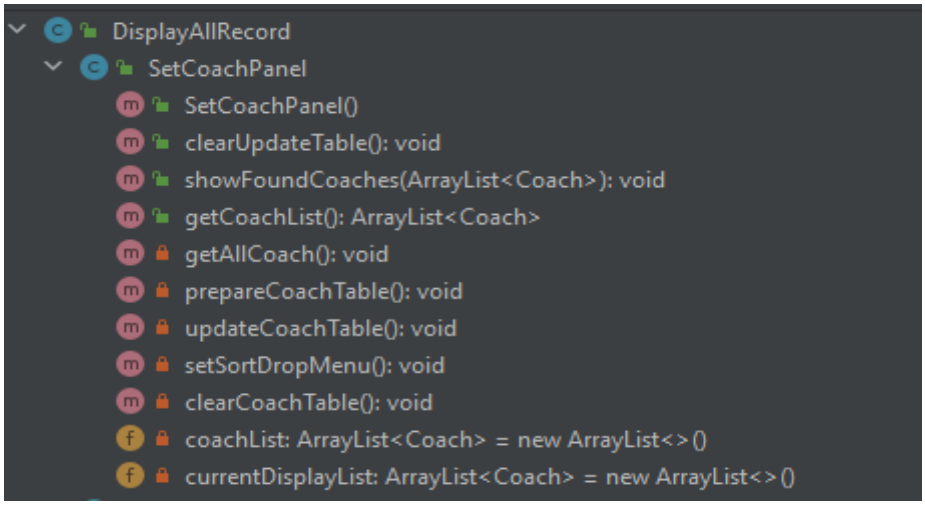



*Figure 18 : Structure of DisplayAllRecord class with SetCoachPanel as inner class*

```java
public class SetCoachPanel implements TablePanelManager {

    private ArrayList<Coach> coachList = new ArrayList<>();
    private ArrayList<Coach> currentDisplayList = new ArrayList<>();

    public SetCoachPanel(){
        getAllCoach();
        prepareCoachTable();
        updateCoachTable();
        setSortDropMenu();
    }
    @Override
    public void clearUpdateTable() {
        clearCoachTable();
        coachList.clear();
        getAllCoach();
        updateCoachTable();
    }

    /*  Method Name : getAllCoach
        Description : Read coach file and instantiate all coach objects & store it in array list
     */

    private void getAllCoach () {
        String[] coachFileContent = FileServer.readFile(admin.getSportsCenterCode(), fileName: "Coach.txt");
        for (String coachInfo: coachFileContent){
            Coach coach = new Coach(coachInfo.split( regex: "\\|"));
            coachList.add(coach);
        }
        currentDisplayList = (ArrayList<Coach>) coachList.clone();
    }

    /*  Method Name : prepareCoachTable
        Description : Set the number of columns in JTable
     */
    private void prepareCoachTable (){
        coachRecordTable.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
        coachTableModel = new DefaultTableModel(Coach.getAllAttributes(), rowCount: 0){
            @Override
```

*Figure 19 : Java Code Snippet*

In the figure above, the inner class called SetCoachPanel is responsible for setting up the table in the coach panel to display all the coach records. There are some methods that are set to private access modifier to hide away the implementation details, such as preparing the table in a suitable format, and etc. All of these helps to minimize complexities especially when other classes want to interact with DisplayAllRecord class. For instance, other class who wants to display data in a table format, do not have to figure out the table size and format since it is handled by the “panel manager” in DisplayAllRecord.

As a recap, the techniques that have been used to implement abstraction in this assignment are, setting private access modifier for methods to hide implementation details, creating a whole new class that facilitates a specific job such as file handling, and using inner classes to further break down complexities and hide away those complexities, therefore making things simple and abstract to outer classes.

## Inheritance

Another pillar of Object-Oriented Concepts is inheritance. Inheritance allows a class to be derived from other classes, and through this, it forms a hierarchy of classes that share the same set of attributes, and methods. Since java, does not support multiple inheritance, a class can only be deriving from one class. A class is known as a subclass if it is being derived from some other class. Whereas, a class is known as a superclass, or parent class, if it derives from other classes. Inheritance is very useful in terms of code reusability.

In this section, ways of implementing inheritance in the project are discussed.

```java
class FormChecker {
    public boolean onlyDigits(String str) {
        if (str.isEmpty()){
            return false;
        }
        for (int index = 0; index < str.length(); index++) {
            if (!Character.isDigit(str.charAt(index)))
                return false;
        }
        return true;
    }

    public boolean isDateObject(String str) {
        try {
            LocalDate.parse(str);
        } catch (Exception e) {
            return false;
        }
        return true;
    }

    public final boolean isIntegerObject(String str) {
        try {
            Integer.parseInt(str);
        } catch (Exception e) {
            return false;
        }
        return true;
    }

    public final boolean isDay(String str) {
        ArrayList<String> listOfDays = new ArrayList<~>(Arrays.asList(
                "monday", "tuesday", "wednesday", "thursday", "friday", "saturday", "sunday"));
        return listOfDays.contains(str.toLowerCase());
    }
```

*Figure 20 : Java Code Snippet*

```java
    public boolean isTime(String str) {
        if (str.length() > 5)
            return false;
        String[] tokens = str.split( regex: ":");
        if (tokens.length == 2) {
            try {
                LocalTime.of(Integer.parseInt(tokens[0]), Integer.parseInt(tokens[1]));
                return true;
            } catch (Exception e) {
            }
        }
        return false;
    }

    public int isLogicalDuration(String startStr, String endStr) {
        String[] startTokens = startStr.split( regex: ":");
        String[] endTokens = endStr.split( regex: ":");
        try {
            LocalTime startTime = LocalTime.of(Integer.parseInt(startTokens[0]), Integer.parseInt(startTokens[1]));
            LocalTime endTime = LocalTime.of(Integer.parseInt(endTokens[0]), Integer.parseInt(endTokens[1]));
            if (startTime.isBefore(endTime))
                return 0;
            else
                return 1;
        } catch (Exception e) {
            return 2;
        }
    }
}
```

*Figure 21 : Java Code Snippet*

From the figure in 20 and 21, it shows the form checker class. A form checker class is a class that is responsible for checking and validating inputs entered by the user into the GUI. As shown above, it contains methods such as onlyDigits(), isDateObject(), and others. The purpose of showing this class is that later, some GUI classes especially those that require users to fill in their user details, will extend this class.

```java
public class StudentProfile extends FormChecker{
    private JFrame frame;
    private JPanel rootPanel;
    private JPanel studentProfilePanel;
    private JPanel saveDetailsPanel;
    private JPanel modifyDetailsPanel;
    private JPanel changePasswordPanel;
```

*Figure 22 : Java Code Snippet*

In the figure above, the StudentProfile class extends the form checker class.

```java
private int updateStudentDetails () {
    int returnNum = 0;
    if (!onlyDigits(ageField.getText()) || ageField.getText().length()>2 || ageField.getText().isEmpty()) {
        setBorderRed(ageField, message: "Invalid age provided");
        returnNum = 1;
    }
    if (!onlyDigits(phoneField.getText())|| phoneField.getText().isEmpty()) {
        setBorderRed(phoneField, message: "Invalid contact number provided");
        returnNum = 1;
    }
    if (!emailField.getText().contains("@") || emailField.getText().isEmpty()) {
        setBorderRed(emailField, message: "Invalid email address provided");
        returnNum = 1;
    }
    if (addressField.getText().isEmpty()){
        setBorderRed(addressField, message: "Empty values provided");
        returnNum = 1;
    }
    if (returnNum == 0) {
        student.updateDetails(Integer.parseInt(ageField.getText()), addressField.getText(), phoneField.getText(), emailField.getText());
        return returnNum;
    }
    else
        return returnNum;
```

*Figure 23 : Java Code Snippet*

In the code snippet above, it can be seen the "updateStudentDetails" method inside the StudentProfile class uses the FormChecker methods, such as onlyDigits() to validate the inputs entered by the user.

```java
private class SetCoachTab extends FormChecker{
    public SetCoachTab() {
        coachIDField.setText(coach.getCoachID());
        nameField.setText(coach.getName());
        dateJoinedField.setText(coach.getDateJoined().toString());
        try {
            dateTerminatedField.setText(coach.getDateTerminated().toString());
        } catch (Exception e) {
            dateTerminatedField.setText("null");
        }
```

*Figure 24 : Java Code Snippet*

The code snippet above shows that the SetCoachTab class extends FormChecker class.

```java
private int verifyCoachDetails (List<String> coachDetails){
    int returnNum = 0;
    // Check for each length
    for (int index = 0; index<coachDetails.size(); index++){
        if (coachDetails.get(index).length() >= 100){
            setCoachBorderRed(index, message: "Value is too unrealistic large/long");
            returnNum = 1;
        }
        else {
            switch (index){
                case 0 :  // no action is performed is the field does not allowed modification
                    break;
                case 1:
                    break;
                case 2:
                    if (!isDateObject(coachDetails.get(2))) {
                        setCoachBorderRed(index, message: "Date format should be YYYY-MM-DD");
                        returnNum = 1;
                    }
                    break;
                case 3:
                    if (!isDateObject(coachDetails.get(3)) && !coachDetails.get(3).equals("null")) {
                        setCoachBorderRed(index, message: "Date format should be YYYY-MM-DD");
                        returnNum = 1;
                    }
                    break;
                case 4:
                    if (!isIntegerObject(coachDetails.get(4))) {
                        setCoachBorderRed(index, message: "Invalid integer provided");
                        returnNum = 1;
                    }
                    break;
```

*Figure 25 : Java Code Snippet*

The code snippet above shows a method called “verifyCoachDetails” which resides inside the setCoachTab class. Since the setCoachTab is derived from the FormChecker class, therefore, the verifyCoachDetails method is able to use the methods defined in the FormChecker class. For example, in the figure above, the isDateObject() method, isIntegerObject() method are used to validate the user input. Hence, the advantage of inheritance is demonstrated above, where methods from the FormChecker class can be reused in two different classes. Without inheritance, code reusability would be impossible.

Next, inheritance also allows subclasses to inherit the type of its parent class (Janssen,2017). An example is shown below.

```java
package com.company;

public abstract class BaseStudent {}
```

*Figure 26 : Java Code Snippet*

In the code snippet above, it shows an abstract class which is called BaseStudent. It is an abstract class because, in the project, there should be no instances of BaseStudent.

```
public class RegisteredStudent extends BaseStudent {
    private String name;
    private String studentID;
    private int age;
    private String address;
    private String contactNumber;
    private String email;
    private final String registeredSports;
    private final String sportsCenterCode;
    private final Coach coach;
    private boolean givenRating;
```

*Figure 27 : Java Code Snippet*

```
public class UnregisteredStudent extends BaseStudent{
```

*Figure 28 : Java Code Snippet*

Based on the code snippet in figure 27 and 28 above, they show that the RegisteredStudent class extends from BaseStudent class and the same goes for UnregisteredStudent. The main purpose of this inheritance is to allow RegisteredStudent class and UnregisteredStudent class to inherit the type of BaseStudent class. Through this, RegisteredStudent and UnregisteredStudent instances can also be counted as an instance of the BaseStudent class. This helps to group both of the instances together in some situations which will be shown below.

```java
public StudentMenu (BaseStudent studentA){
    frame = new JFrame( title: "Main Menu");
    if (studentA instanceof RegisteredStudent) {
        regStudent = (RegisteredStudent) studentA;
        guestStudent = null;
        viewSportsDetailsButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                frame.setVisible(false);
                new ViewSports(regStudent);
            }
        });
    }
    else{
        guestStudent = (UnregisteredStudent) studentA;
        regStudent = null;
        viewSportsDetailsButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                frame.setVisible(false);
                new ViewSports(guestStudent);
            }
        });
    }
```

*Figure 29 : Java Code Snippet*

The figure above shows the StudentMenu Class, which is a GUI class that creates a student menu for registered and non-registered students. Instead of using constructor overloading to build two different constructors with different signatures to cater for two different types of students. Both types of students can be classified as BaseStudent type since both of the RegisteredStudent class and UnregisteredStudent class inherit BaseStudent class. Therefore, RegisteredStudent and UnregisteredStudent instances can be accepted with the same constructor. Another feature in inheritance that is implemented is downcasting. Downcasting can be said as casting the type of an instance to its subclasses. In the figure above, the studentA instance which was accepted as a BaseStudent class, is downcasted to UnregisteredStudent or RegisteredStudent after determining which subclass it belongs to.

All in all, inheritance is used in this assignment to share methods across different classes, and to inherit the type of the superclass, so that both child classes can be considered having the same type after upcasting, which is casting the type of a class to its superclass type.

## Polymorphism

In layman's terms, “polymorphism” means the ability of an organism or object to take on several forms based on its condition. This explanation does not stray too far from the programming variation, as in Java, polymorphism indicates the ability of a single object or method to perform a specific action in more than one way (Singh, 2014). Generally, polymorphism can be divided into 2 parts which are “Static polymorphism” and “Dynamic polymorphism”, both of which were implemented when developing this program. Common examples of static polymorphism include “Method overloading” and “Constructor overloading”. Method overloading allows a class to have multiple methods with the same name if the data type, sequence, and name of the variables passed into their arguments are different. Similarly, constructor overloading allows for a class to have multiple constructors with different lists of arguments for each of them. An instance of method overloading in the program is shown below.

```
public  ArrayList<Coach> searchCoach(List<Coach>coachList,String ID){
    ArrayList<Coach>found = new ArrayList<>();
    for (Coach coach:coachList){
        if (ID.equalsIgnoreCase(coach.getCoachID()))
            found.add(coach);
    }
    return found;
}
public ArrayList<Coach> searchCoach(List<Coach>coachList,int rating){
    ArrayList<Coach>found = new ArrayList<>();
    for (Coach coach:coachList){
        try {
            if (rating == (coach.getRating() / coach.getTotalRates()))
                found.add(coach);
        }catch (ArithmeticException e){
            if (rating == 0)
                found.add(coach);
        }
    }
    return found;
}
public ArrayList<Coach> searchCoach(List<Coach>coachList,String ID, int rating){
    return searchCoach(searchCoach(coachList,ID),rating);
}
```

*Figure 30: Method overloading for “searchCoach” in Admin class*

The snapshot above shows the overloading of the method “searchCoach” used in the Admin class. When searching for a record of a specific coach, the admin will be given a choice within the GUI to either search for said coach using coach ID, coach ratings, or both. And depending on what the admin chose to search with, the program will execute the suitable variant of the “searchCoach” method to look for the coach’s profile. And regardless of which code was executed based on the parameters passed in, the methods still have the same purpose and will show a similar output result.

```
public static int appendFile (String subDirectory, String fileName, String fileContent){
    File file = null;
    if (subDirectory.isEmpty()) {
        file = new File(fileName);
    }
    else {
        file = new File(subDirectory, fileName);
    }
    try {
        FileWriter fw = new FileWriter(file, append: true);
        BufferedWriter writer = new BufferedWriter(fw);
        writer.write(fileContent);
        writer.close();
        return 0;
    } catch (IOException e){
        e.printStackTrace();
    }
    return 1;
}

// Method overloaded
public static int appendFile(String fileName,String fileContent) { return appendFile( subDirectory: "",fileName,fileContent); }
```

*Figure 31: Method overloading for “appendFile” in FileServer class*

Another example of method overloading can be found in the FileServer class. The “appendFile” method is overloaded so that it is able to accept either 3 or 2 arguments. The arguments that are needed to be passed into the method are the files’ subdirectory, file name, and content to be written to the file. However, if the subdirectory argument was not specified, the program will proceed to set it as empty or null and pass it into the 3 arguments variant of the method and run it as normal.

```java
public Schedule (String sportsCenterCode, String type, String[] sessionID){
    String[] sessionFile = FileServer.readFile(sportsCenterCode, fileName: "Session.txt");
    for (String ID : sessionID){
        for (String line : sessionFile){
            String[] tokens = line.split( regex: "\\|");
            if (tokens[1].equals(ID)) {
                sessionList.add(new Session(tokens));
                break;
            }
        }
    }
    this.type = type;
}

// Overloaded constructor that creates a schedule with empty session list
public Schedule(String type){
    this.type = type;
}
```

*Figure 32: Constructor overloading for "Schedule" class*

The screenshot of the code above shows the use of constructor overloading in the "Schedule" class. The first constructor accepts a sports center code, a type string, and an array containing session IDs for its argument and the second one only requires the type string to be passed in. Under normal circumstances, a Schedule class object will be initiated by reading a text file containing the sports session details and adding those sessions with identical session IDs passed into the constructor via an array, and adding them into the session array list attribute in the schedule class. However, in the scenario where a specific sport has yet to have any existing sessions, the program will instead only pass "type" for the parameter and the session array list will not be initialized, remaining as null.

```java
@Override
public String toString () {

    return name + "|"+ coachID + "|" + dateJoined + "|" +
            dateTerminated + "|" + hourlyRate + "|"
            + phone + "|" + address + "|" + sportsCenterID + "|"+
            sportsCode + "|" + rating + "|" + totalRates + "|" + sportsName; }
```

*Figure 33: Overriding toString() method in Coach class*

Moving onto dynamic polymorphism, a common example for it is method overriding. Calling an overridden method involves determining the variant of that method being used based on the object type referred to at the time the method is called (GeeksforGeeks, 2018). All of this is computed during run time of the program instead of compile-time, hence being categorized as dynamic polymorphism. Looking at the code snippet above, it shows an example of the implementation of method overriding in the Coach class where the default toString method that was inherited from the Object class is overridden to return a specific set of Strings with variables from within the Coach class.

```java
public StudentMenu (BaseStudent studentA){
    frame = new JFrame( title: "Main Menu");
    if (studentA instanceof RegisteredStudent) {
        regStudent = (RegisteredStudent) studentA;
        guestStudent = null;
        viewSportsDetailsButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                frame.setVisible(false);
                new ViewSports(regStudent);
            }
        });
    }
    else{
        guestStudent = (UnregisteredStudent) studentA;
        regStudent = null;
        viewSportsDetailsButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                frame.setVisible(false);
                new ViewSports(guestStudent);
            }
        });
    }
```

*Figure 34: Dynamic polymorphism in StudentMenu GUI class*

Another example of dynamic polymorphism can be seen implemented in the StudentMenu GUI class. From the code snippet above, the constructor of the StudentMenu class takes in a BaseStudent as its argument, which can be inherited by either the RegisteredStudent or UnregisteredStudent classes. The constructor will then check whether or not the BaseStudent passed in as the argument is an instance of the RegisteredStudent class. And based on that, the class constructor will proceed to run 1 of 2 different sets of codes, resulting in slight differences in the menus displayed for registered students and unregistered students respectively.

## Extra features

### Implementation of HashMap Data Structure

```java
/* Method : updateScheduleFile
   Description : Update changes made to session.txt to schedule.txt
                 (Schedule.txt is derived from session.txt)

   Parameter : sportCenterCode (String) -- indicates which sport center schedule to update
   Return    : 0 for update successful
               1 for update failed
 */
public static int updateScheduleFile (String sportCenterCode) {
    // Use hash map structure to store schedules mapped to their type (day / sports)
    HashMap<String,Schedule> scheduleHashMap = new HashMap<>();
    String[] scheduleFile = FileServer.readFile(sportCenterCode, fileName: "Schedule.txt");

    // Only create schedule instances for sunday - saturday (first 7 lines)
    for (int line = 0;line < 7;line++){
        String[] tokens = scheduleFile[line].split( regex: "\\|");
        scheduleHashMap.put(tokens[0], new Schedule(tokens[0]));
    }

    // Initiate all sessions from session.txt & add them into respective schedule (day & sports)
    String[] sessionFile = FileServer.readFile(sportCenterCode, fileName: "Session.txt");
    for (String line:sessionFile){
        String[] tokens = line.split( regex: "\\|");
        Session sessionToAdd = new Session(tokens);
        // Check for whether the day schedule is present in hashmap, if not create a new one & add session
        if (scheduleHashMap.get(tokens[0]) == null)
            scheduleHashMap.put(tokens[0], new Schedule(tokens[0]));

        scheduleHashMap.get(tokens[0]).addSession(sessionToAdd);

        // Check for whether the sports schedule is present in hashmap, if not create a new one & add session
        if (scheduleHashMap.get(tokens[6]) == null)
            scheduleHashMap.put(tokens[6], new Schedule(tokens[6]));

        scheduleHashMap.get(tokens[6]).addSession(sessionToAdd);
    }
```

*Figure 35 : Java Code Snippet*

From the java code snippet above, a HashMap data structure is implemented from the java.util package to store schedule objects. HashMap is a data structure that stores data in key and value pairs, and the data can be accessed with the key that is mapped to the value (Geeks for Geeks,2020). By using a hash map data structure, the schedule objects can be stored, and then retrieved by their “type” field, which is a string value, instead of using index value if the schedule objects were to be stored in an array or array list. For example, in the code shown above, schedule object is retrieved by using scheduleHashMap.get(tokens[0]), scheduleHashMap.get(tokens[6]), in this case, tokens[0], and tokens[6] is a string that represents the “type” of the schedule, such as Monday Schedule, Badminton schedule, etc.

Implementation of comparator and comparable interface

```java
/*
Class       : sortByPay (implements Comparator interface)
Description : Paired with built in list sorting method to sort coaches by hourly rate
*/
public static class sortByPay implements Comparator<Coach>{
    @Override
    public int compare(Coach coach1, Coach coach2) { return (coach1.hourlyRate-coach2.hourlyRate); }

    @Override
    public String toString() { return "Sort by Hourly Rate"; }

}

/*
Class       : sortByID (implements Comparator interface)
Description : Paired with built in list sorting method to sort coaches by coach ID
*/
public static class sortByID implements Comparator<Coach>{
    @Override
    public int compare(Coach coach1, Coach coach2) { return coach1.coachID.compareTo(coach2.coachID); }

    @Override
    public String toString () { return "Sort by Coach ID"; }

}
```

*Figure 36 : Java Code Snippet*

The java code snippet above shows two inner static class of Coach class implements the Comparator interface.

```java
public class Schedule implements Comparable<Schedule> {
```

```java
@Override
public int compareTo(Schedule o) {
    if (changeDayToNum(this.type) == changeDayToNum(o.type)){
        return this.type.compareTo(o.type);
    } else
        return changeDayToNum(this.type) - changeDayToNum(o.type);
}
```

*Figure 37 : Java Code Snippet*

The figure above shows the Schedule class implements the Comparable interface and overriding the method compareTo from the Comparable interface.

In simple terms, both Comparable and Comparator interfaces implemented allow java to sort the coaches and schedule objects accordingly, such as sorting the coaches based on the hourly rate, and sorting the coaches based on coach ID.

Usage of anonymous inner class

```java
/*  Method Name : prepareCoachTable
    Description : Set the number of columns in JTable
 */
private void prepareCoachTable (){
    coachRecordTable.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
    coachTableModel = new DefaultTableModel(Coach.getAllAttributes(), rowCount: 0){
        @Override
        public boolean isCellEditable (int row, int column){ return false; }
    };
    coachRecordTable.setModel(coachTableModel);
    coachRecordTable.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
}
```

*Figure 38 : Java Code Snippet*

```java
goBackButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        frame.setVisible(false);
        new StudentMenu(student);
    }
});
```

*Figure 39 : Java Code Snippet*

An anonymous inner class is a class that is declared and instantiated at the same time without a proper name in Java (Java Anonymous Inner Class, n.d.). It is commonly used when the are some methods in the parent class that requires overriding. An anonymous class can be used with abstract classes, concrete classes, and interfaces.

In figure 38 above, an anonymous class that extends the DefaultTableModel is created and instantiated at the same time, with the method “isCellEditable” overridden. Next, in figure 39, an anonymous class that implements ActionListener is created and instantiated at the same time, with the method “actionPerformed” overridden.

Usage of generics method

```
/*  Method name : sort (Generic Method)
    Parameter   : tArrayList (Array list containing T typed *preferably instances of Coach/Students*) ,
                  sorter (Comparator instance tells how to sort tArrayList ),
                  ascending (True for ascending, False for descending)
 */
public <T> void sort (ArrayList<T> tArrayList, Comparator<T> sorter, boolean ascending){
    Collections.sort(tArrayList,sorter);
    if (!ascending) // reverse the sorted list if its descending
        Collections.reverse(tArrayList);
}
```

*Figure 40 : Java code snippet*

In the figure above, it shows a generic method named “sort” in the Admin class. In java, generics method allows a method to handle different types of objects passed into the method. For example, in the situation above, the “sort” method can sort for schedule objects, coach objects, student objects, and sports objects. Hence, the usage of generics method can encourage code reusability, without generics method sorting different types of objects might require many different methods with similar implementation logic. This also implies the concept of generalization is applied here.

Program output screens

Admin

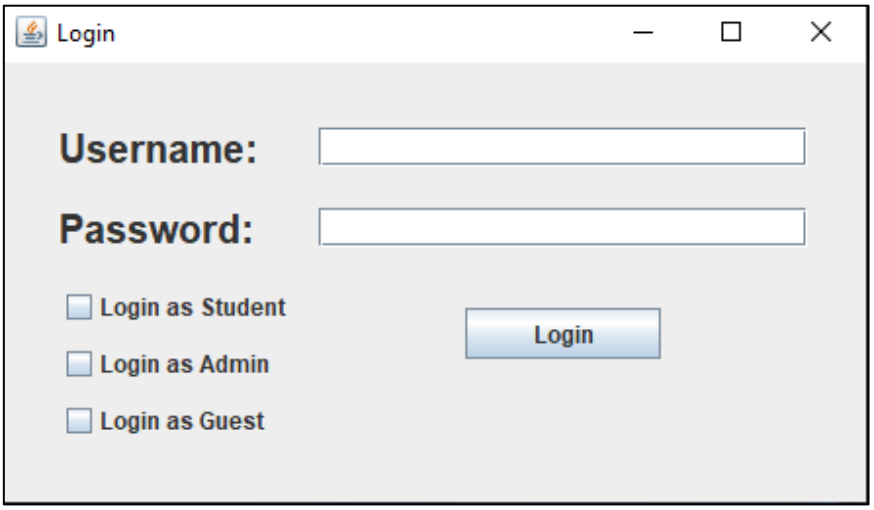



*Figure 41 : Login screen*

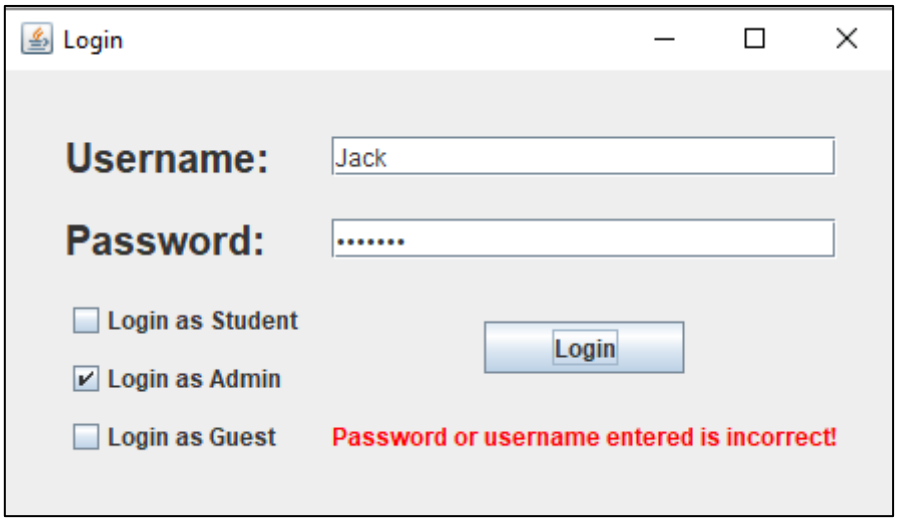



*Figure 42 : Login screen when wrong username and password is entered*

Both figure above shows the possible output screen when admin is logging into the system.

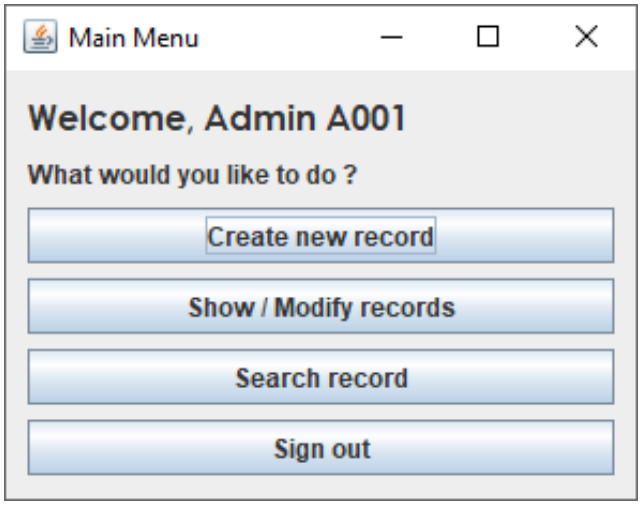



*Figure 43 : Main menu screen for the admin*

The figure above shows the main menu screen for the admin. If the admin chooses to sign out, then it will lead the admin user back to the login screen shown above.

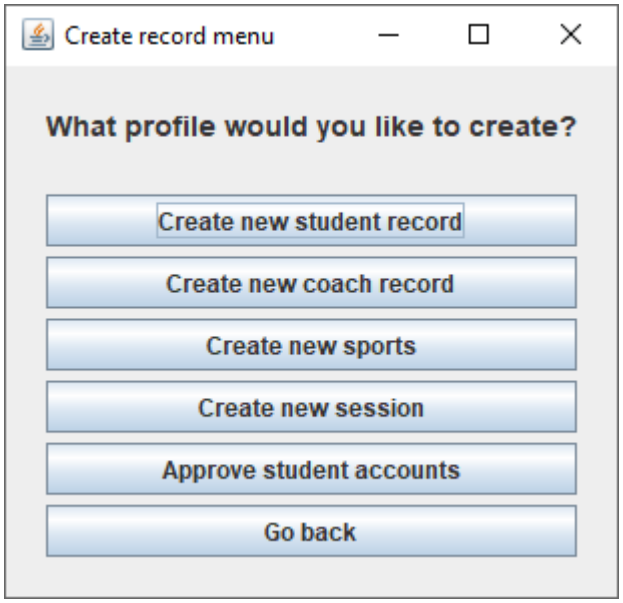



*Figure 44 : Create record menu*

When the admin chooses to create a new record, a create record menu will pop up which prompt the admin to choose which type of record to create.

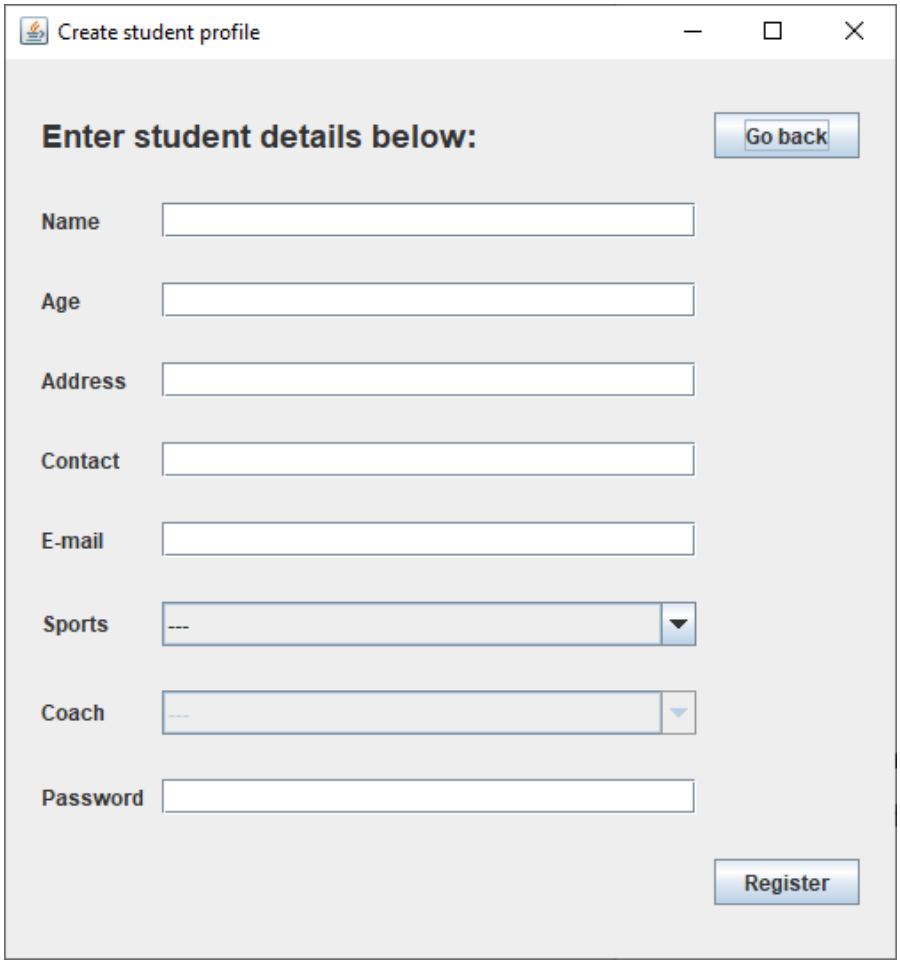



*Figure 45 : Create student profile menu*

In this screen, the admin can input the new student details to create a new student profile.

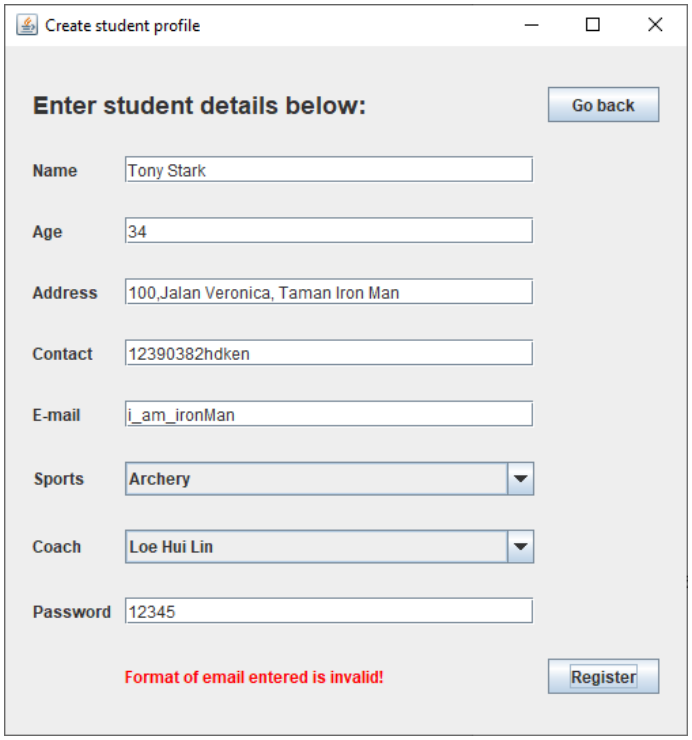



*Figure 46 : Error label indicates the information entered is invalid*

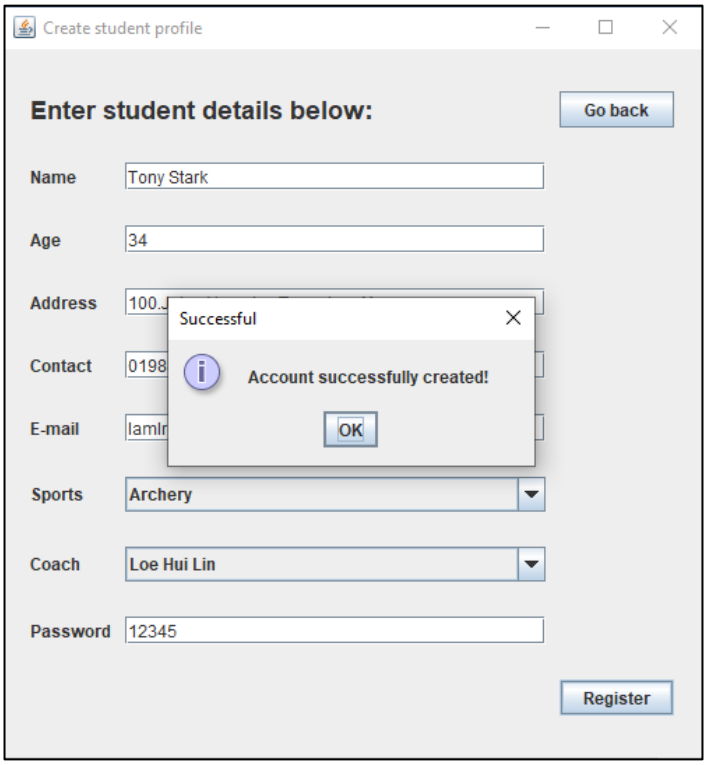



*Figure 47 : Success pop menu is shown when the account is created successfully*

If the account is created successfully, a message is shown and admin is directed back to main menu.

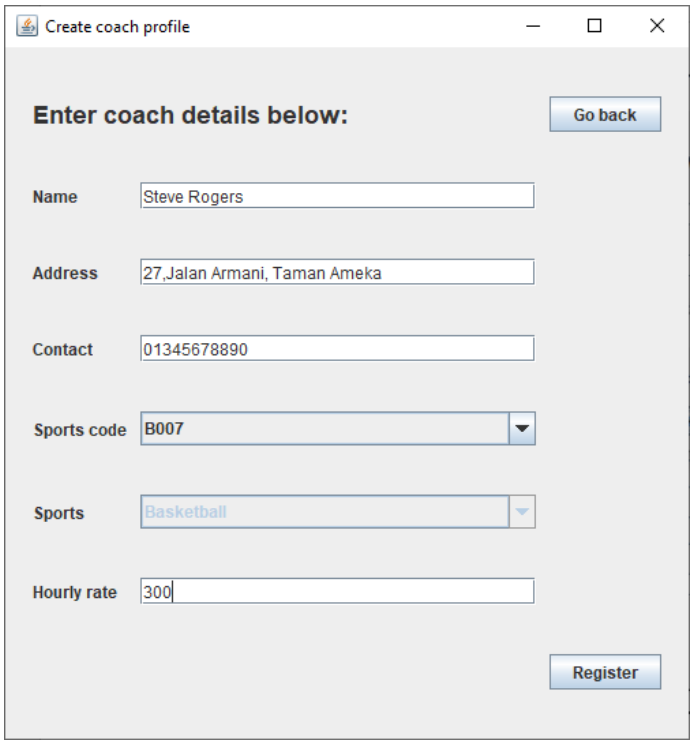



*Figure 48 : Create coach profile menu*

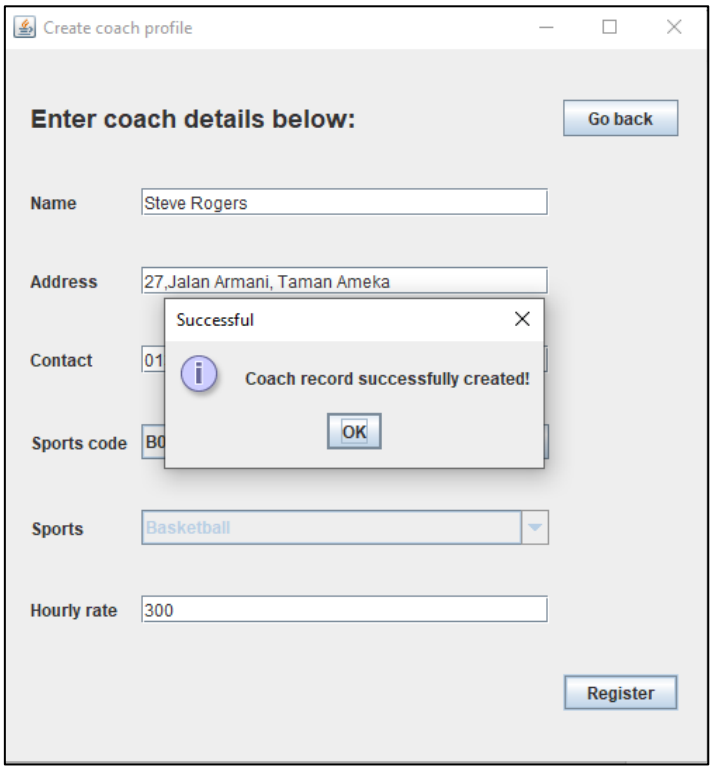



*Figure 49 : Create coach profile menu*

Similar to the student profile menu, if all the coach details entered are valid, then a coach record is created, and a successful message will be shown. Otherwise, a red label indicating the error will be shown similar to figure 46.

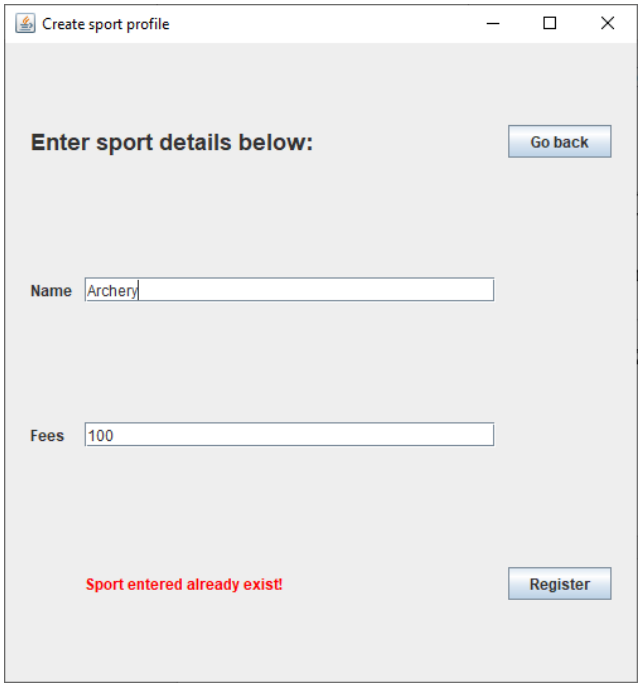



*Figure 50 : Create sport profile screen*

An error will be shown to the admin if the admin is trying to duplicate a sport.

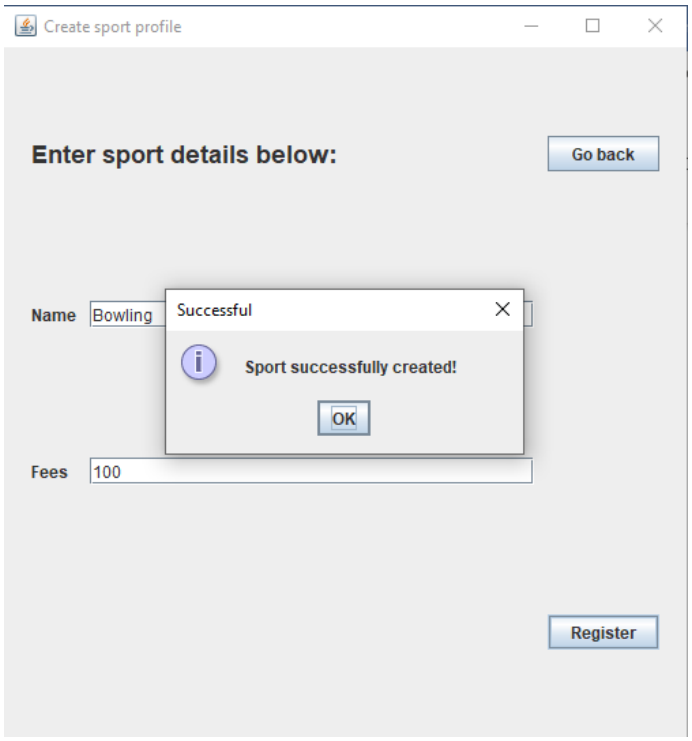



*Figure 51 : Create sport profile screen*

The admin can reenter a new sports name that does not exist currently in the system. A sport is successfully created if a pop-up message says that it is created successfully.

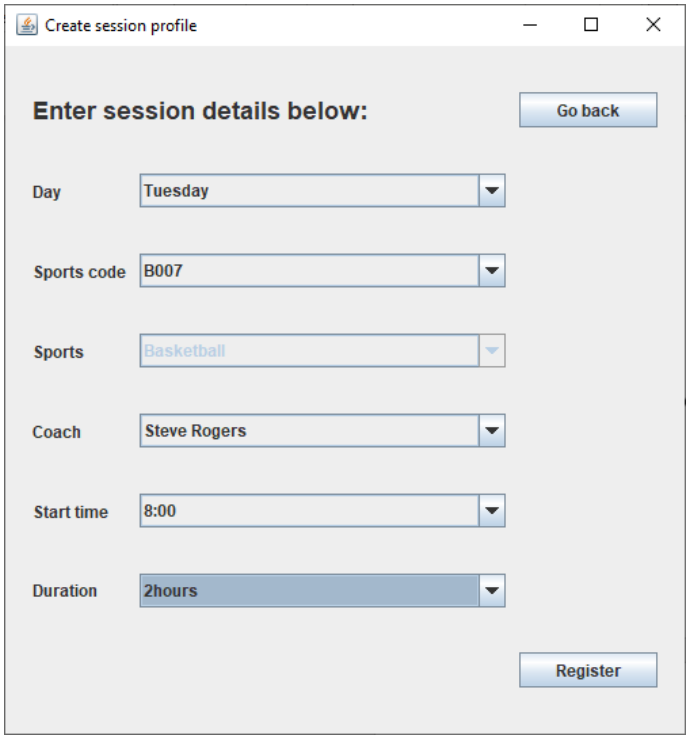



*Figure 52 : Create session profile form*

The admin can create a new session for a sport by selecting the details of the session in the create session profile form shown above.

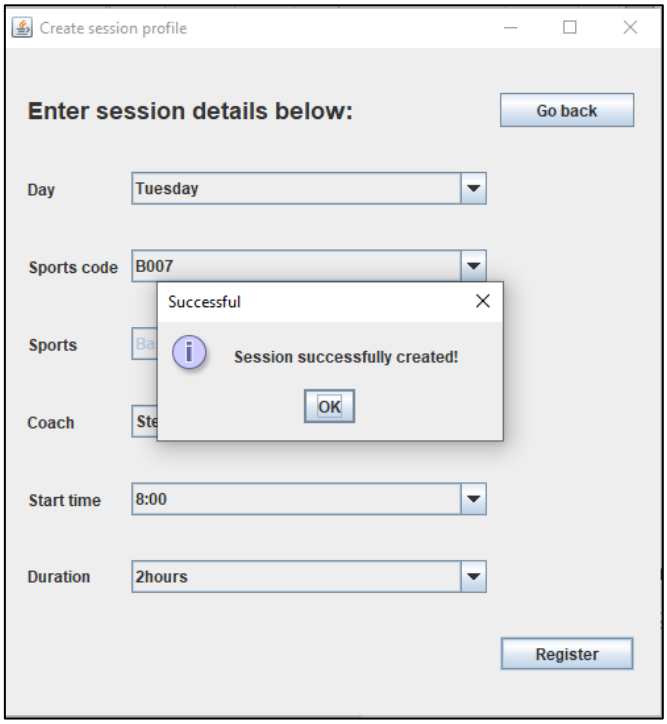



*Figure 53 : Create session profile*

The figure above shows the session is created successfully.

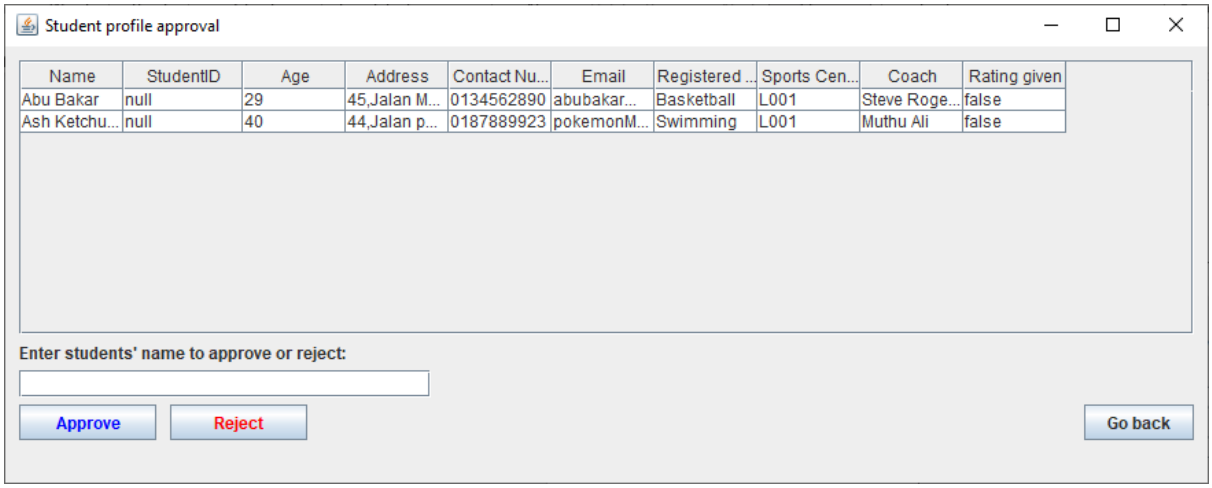



*Figure 54 : Student profile approval screen*

In the figure above, the admin can enter the name of students that are waiting to be approved for registering a sport. Then the admin can either approve their request for joining the sport listed in the sport center, or simply reject their requests. If approved, then a new student account and record will be created, otherwise no student record is created.

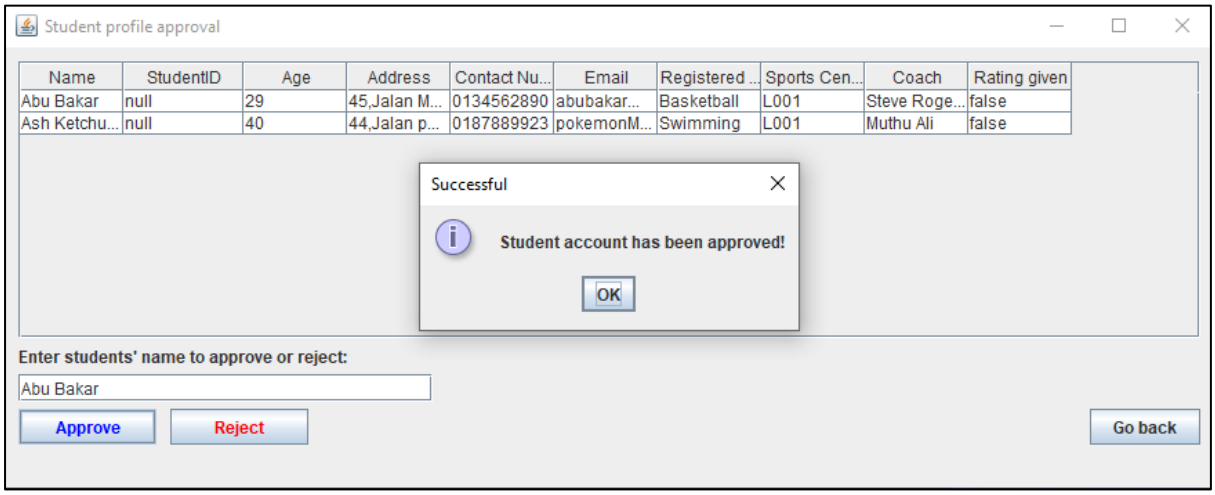



*Figure 55 : Student profile approval screen*

The screen above shows the student is approved and a student record named “Abu Bakar” is created.

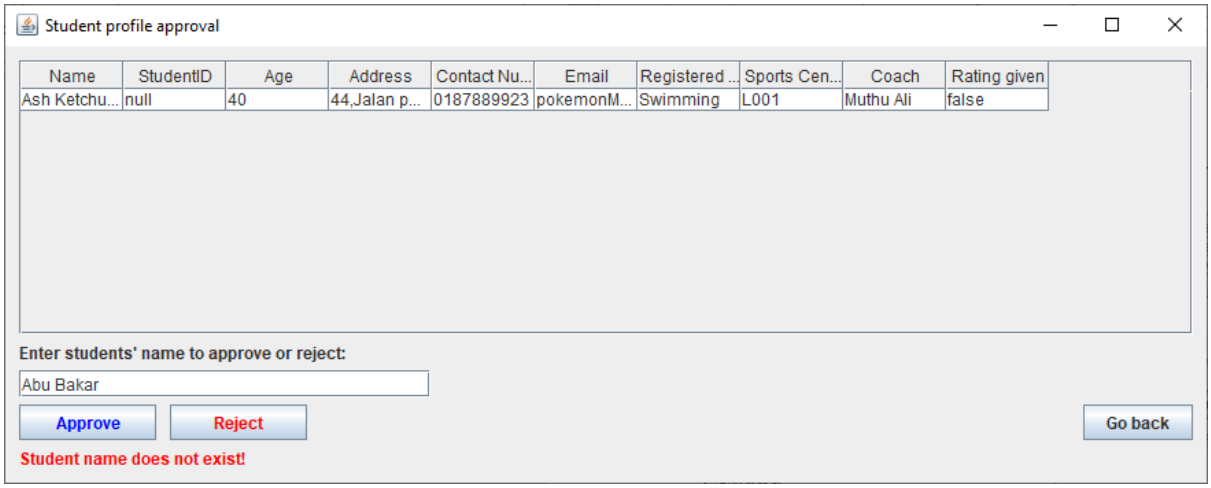



*Figure 56 : An error label if the admin types a student name that does not exist in the table*

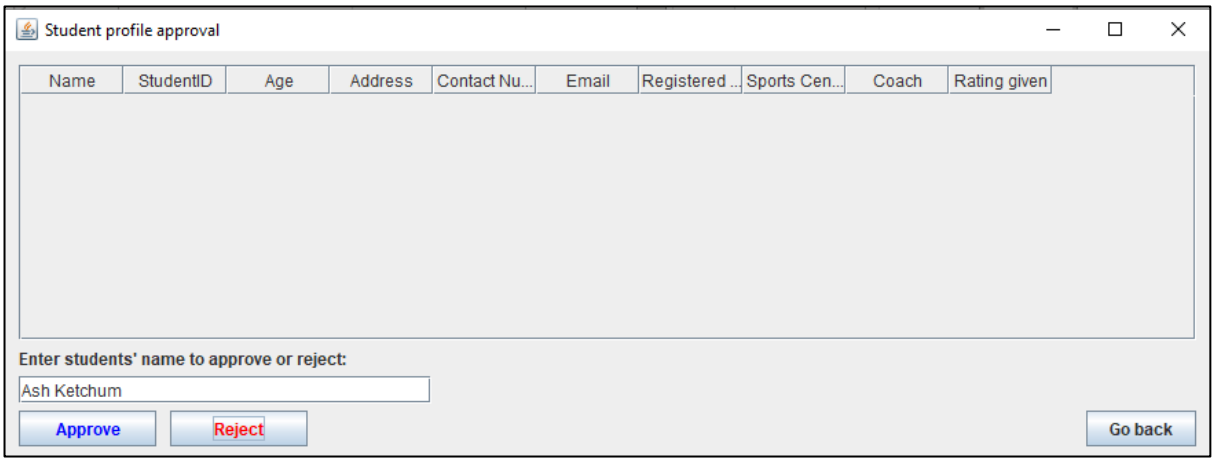



*Figure 57 : Empty student profile approval table*

If the admin selects reject, then the record "Ash Ketchum" will be removed from the table above, and no student profile is created.

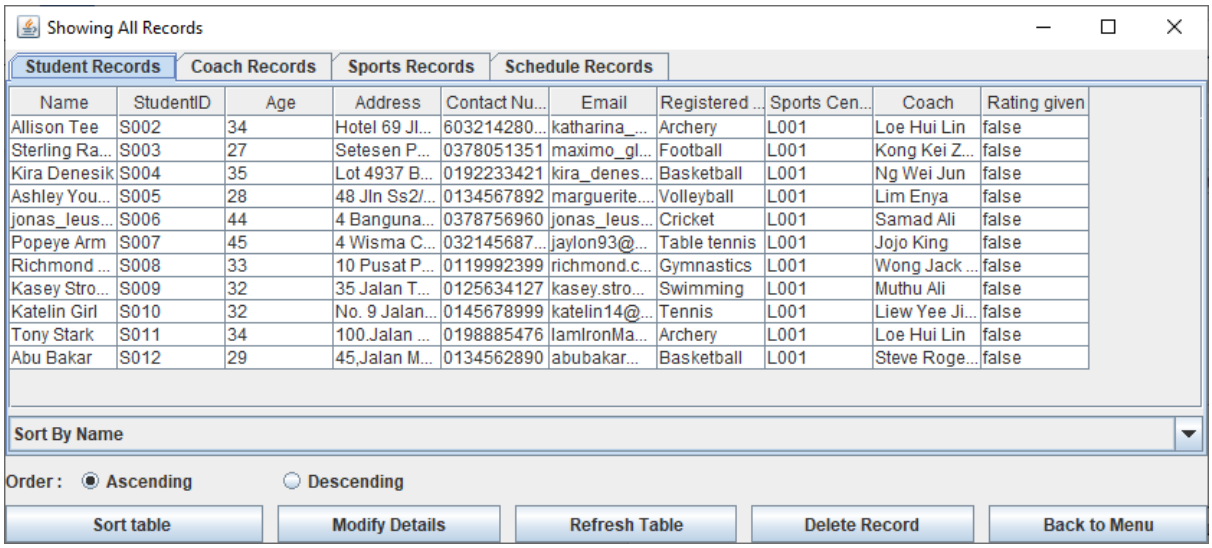



*Figure 58 : Display all records screen*

The figure above shows the display all record screen which shows all the records for students in the sports center.

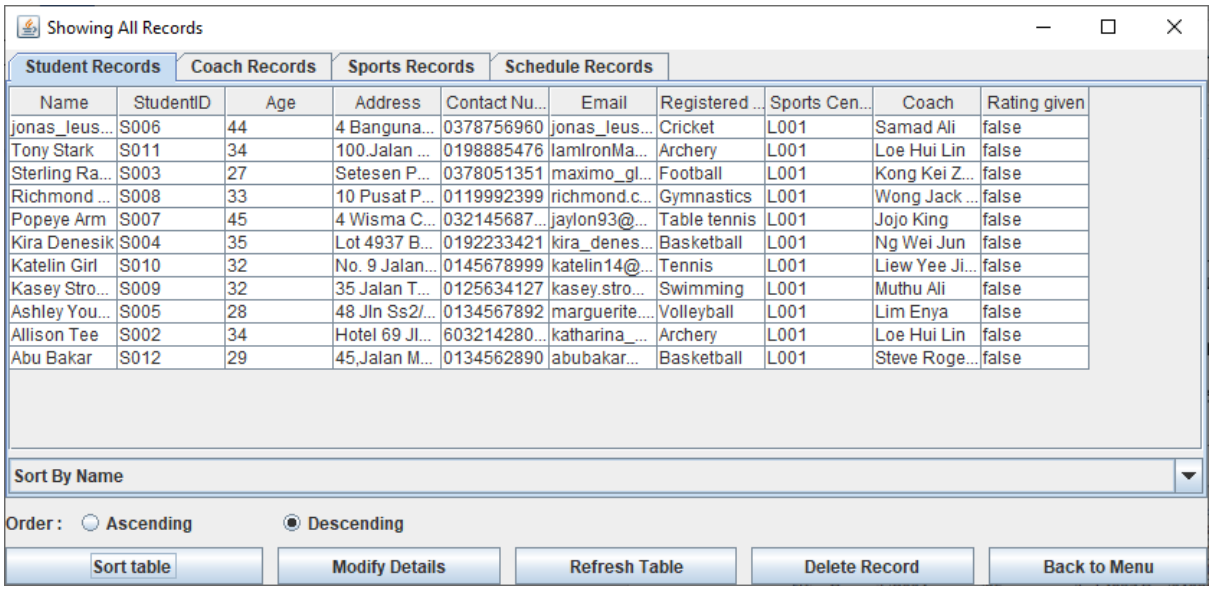



*Figure 59 : Display All Record Screen*

The admin user can select the sort table button and specify the order to sort the table. The above table displayed is sorted in descending order based on student names.

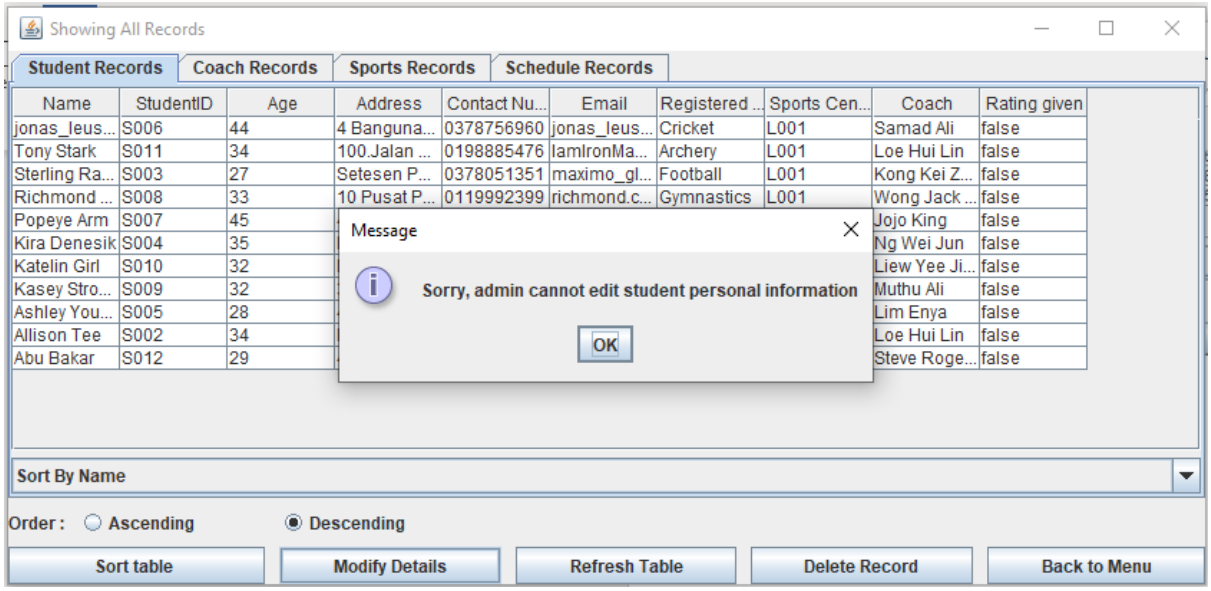



*Figure 60 : Display All Record Screen*

However, the admin user is not given the privilege to modify any student record information, this is to prevent any data tampering that may be caused by the admin. Therefore, a message is popped up whenever the admin tries to modify a student details.

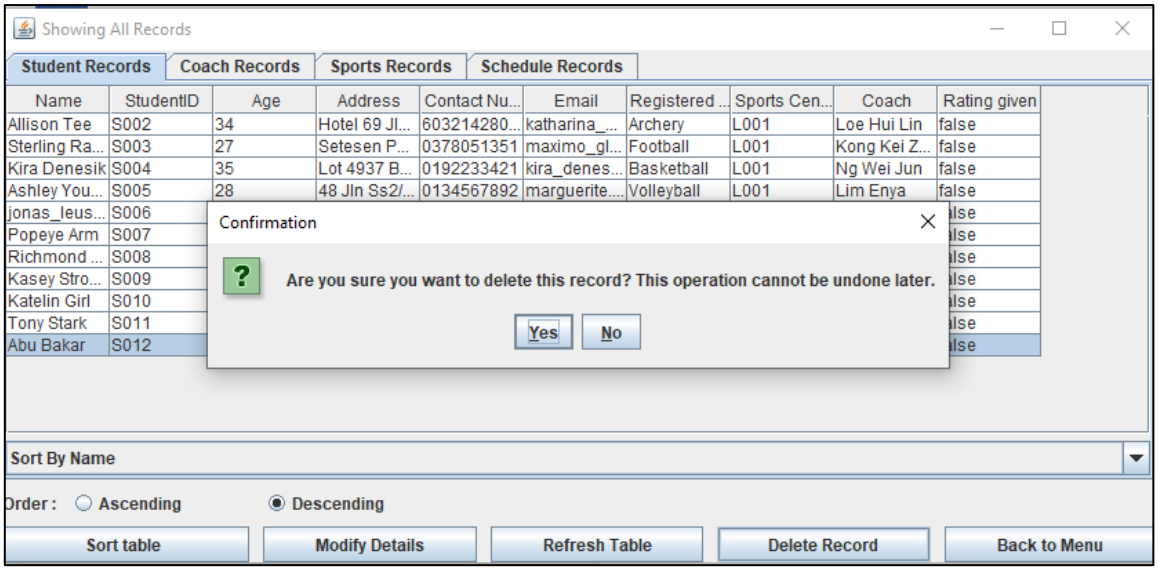



*Figure 61 : Display All Record screen*

In the figure above, when the admin tries to delete a student record, a confirmation message is shown to double confirm whether the admin really wants to proceed with the delete operation. If the admin chooses yes, then the student record is deleted, otherwise, the selected student record is not deleted.

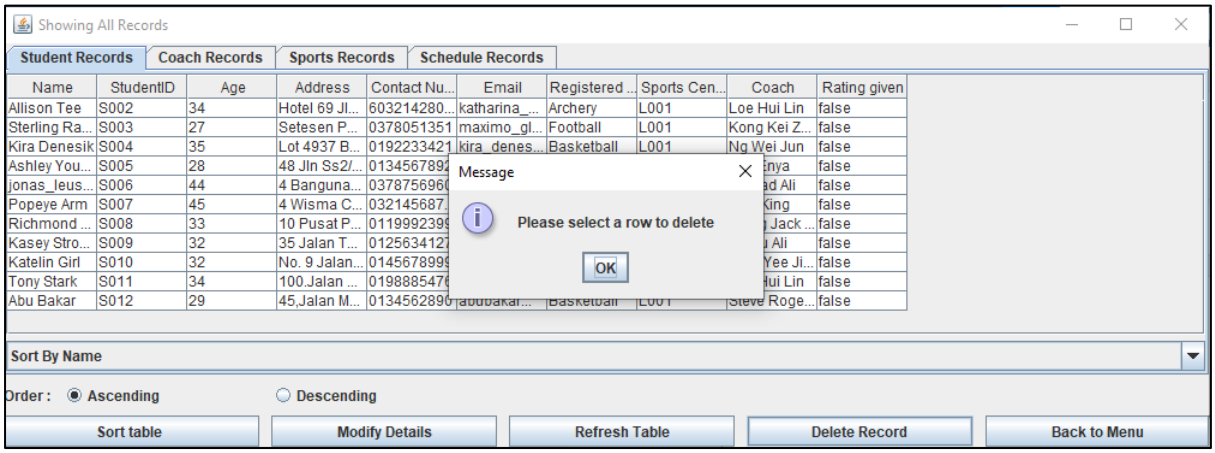



*Figure 62 : Display All Record screen*

The figure above shows the message prompting admin to select a row to delete, if the admin did not select any row when clicking the delete record button.

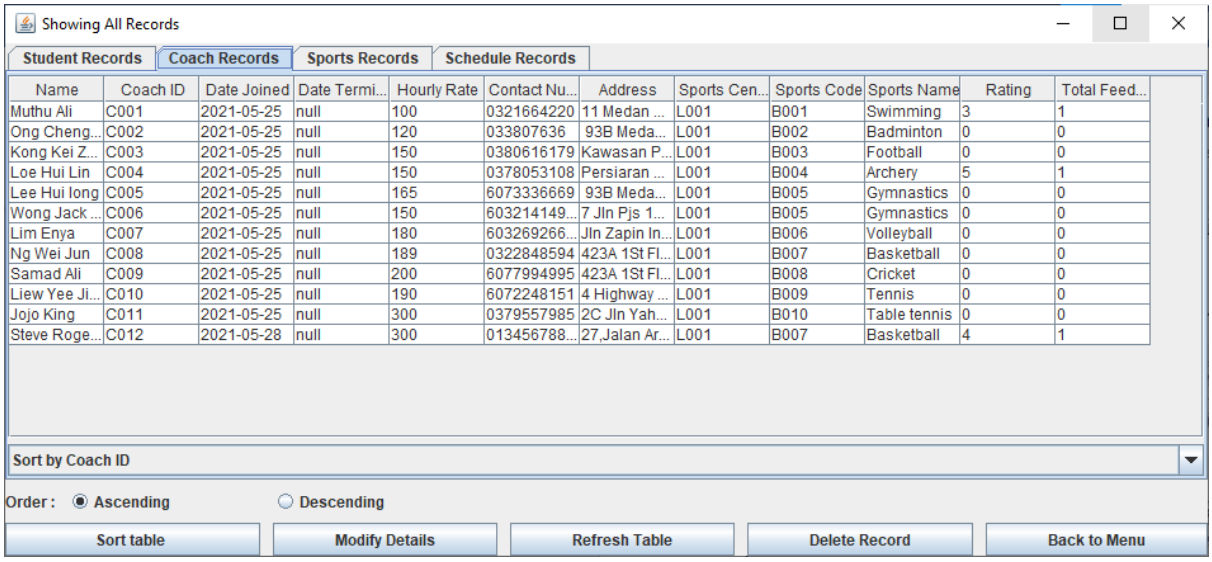



*Figure 63 : Displaying all coach records*

The screen above displays all the coach records.

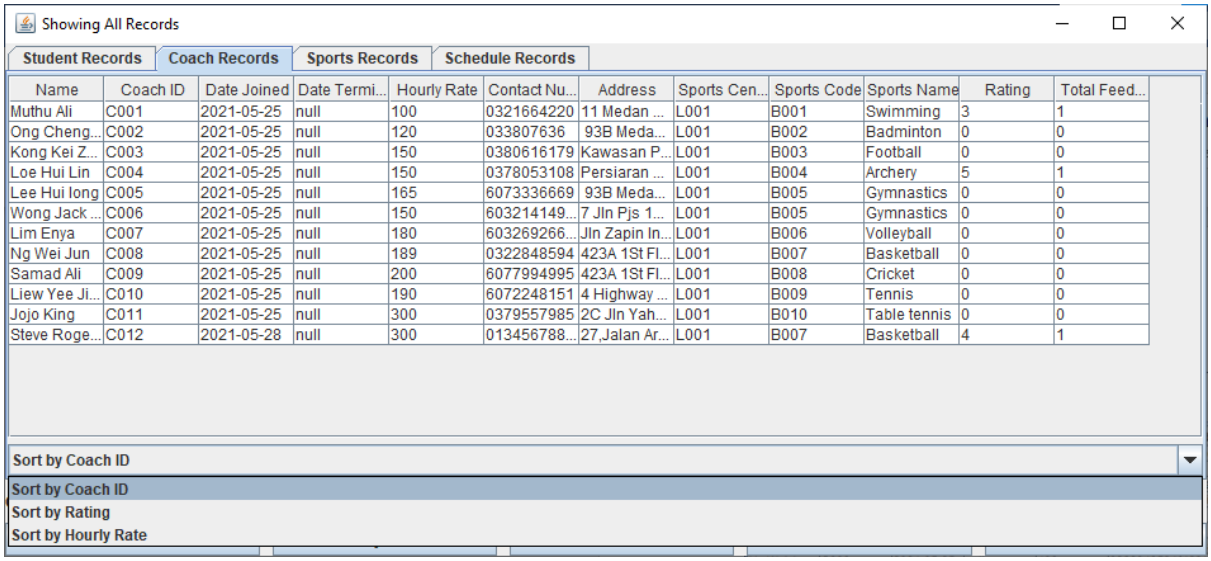
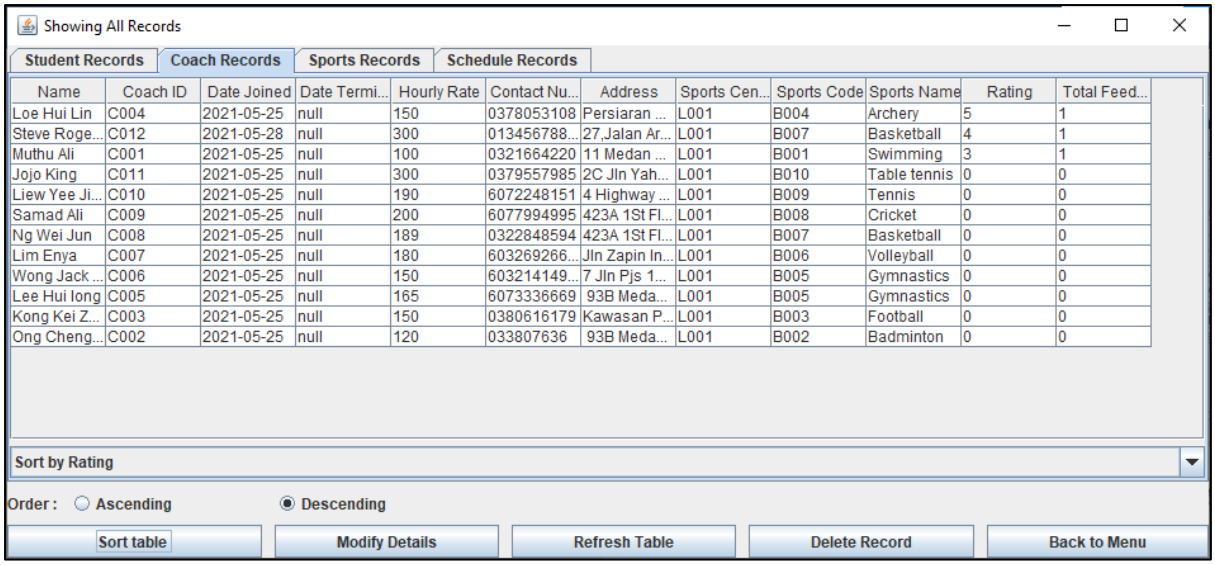



*Figure 64 : Displaying all coach records*

Other than that, the admin can select the sorting method that can be used to sort the list of coaches. After selecting a sorting method, the admin can specify the order, and click the “sort button”.




| Name | Coach ID | Date Joined | Date Termi... | Hourly Rate | Contact Nu... | Address | Sports Cen... | Sports Code | Sports Name | Rating | Total Feed... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Loe Hui Lin | C004 | 2021-05-25 | null | 150 | 0378053108 | Persiaran ... | L001 | B004 | Archery | 5 | 1 |
| Steve Roge... | C012 | 2021-05-28 | null | 300 | 013456788... | 27,Jalan Ar... | L001 | B007 | Basketball | 4 | 1 |
| Muthu Ali | C001 | 2021-05-25 | null | 100 | 0321664220 | 11 Medan ... | L001 | B001 | Swimming | 3 | 1 |
| Jojo King | C011 | 2021-05-25 | null | 300 | 0379557985 | 2C Jln Yah... | L001 | B010 | Table tennis | 0 | 0 |
| Liew Yee Ji... | C010 | 2021-05-25 | null | 190 | 6072248151 | 4 Highway ... | L001 | B009 | Tennis | 0 | 0 |
| Samad Ali | C009 | 2021-05-25 | null | 200 | 6077994995 | 423A 1St Fl... | L001 | B008 | Cricket | 0 | 0 |
| Ng Wei Jun | C008 | 2021-05-25 | null | 189 | 0322848594 | 423A 1St Fl... | L001 | B007 | Basketball | 0 | 0 |
| Lim Enya | C007 | 2021-05-25 | null | 180 | 603269266... | Jln Zapin In... | L001 | B006 | Volleyball | 0 | 0 |
| Wong Jack ... | C006 | 2021-05-25 | null | 150 | 603214149... | 7 Jln Pjs 1... | L001 | B005 | Gymnastics | 0 | 0 |
| Lee Hui long | C005 | 2021-05-25 | null | 165 | 6073336669 | 93B Meda... | L001 | B005 | Gymnastics | 0 | 0 |
| Kong Kei Z... | C003 | 2021-05-25 | null | 150 | 0380616179 | Kawasan P... | L001 | B003 | Football | 0 | 0 |
| Ong Cheng... | C002 | 2021-05-25 | null | 120 | 033807636 | 93B Meda... | L001 | B002 | Badminton | 0 | 0 |

*Figure 65 : Displaying all coach records*

The table above shows the display results if the admin chooses to sort the coaches by their respective rating in descending order.

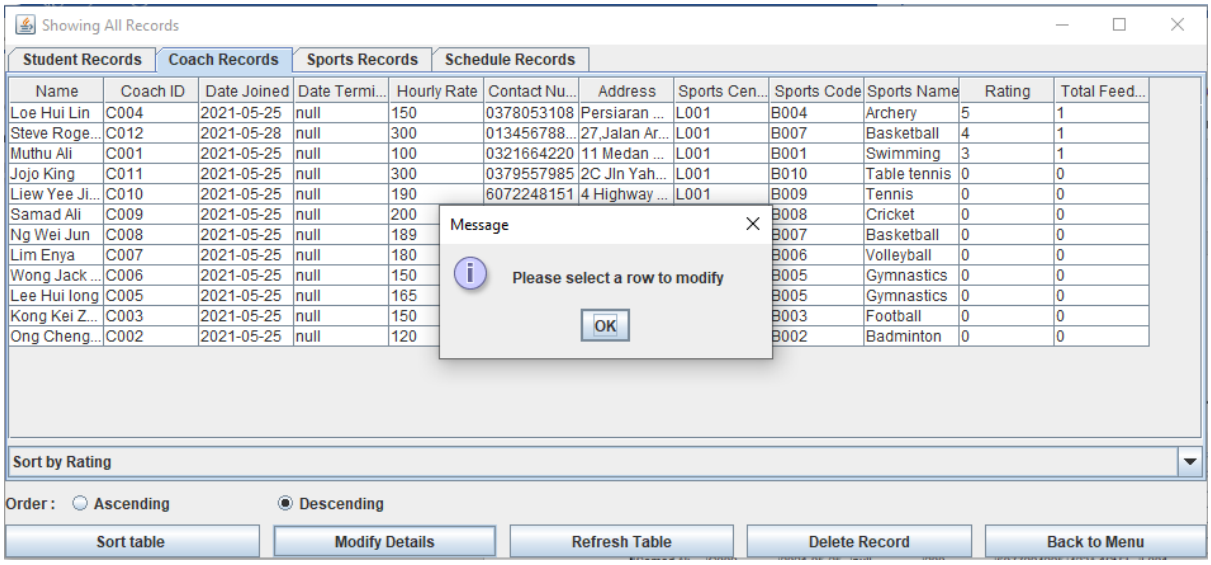



*Figure 66 : Displaying all coach records*

If the admin wants to modify a coach's details such as contact number. The admin is required to select a row that contains the coach and press “Modify Details” button. Otherwise, a message is shown to the admin as shown above in the figure.

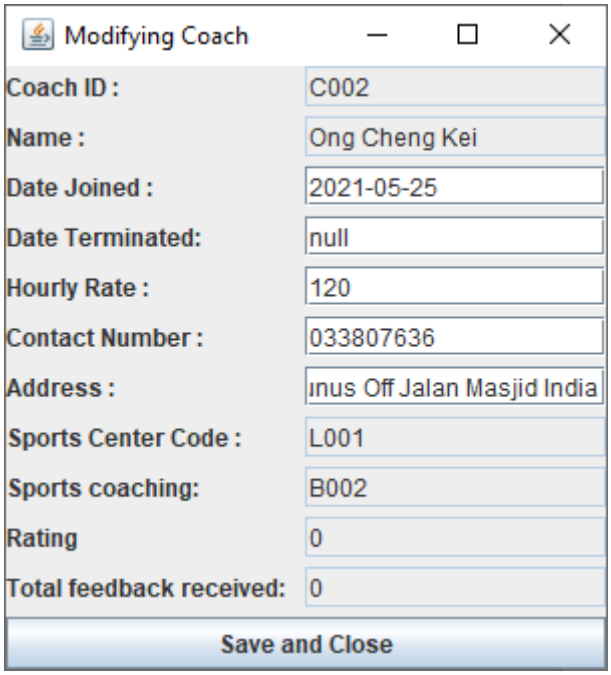



*Figure 67 : Screen that shows modifying coach details*

If the admin has selected a row, and the modify details button is clicked, the admin will be led to this screen shown in the figure above.

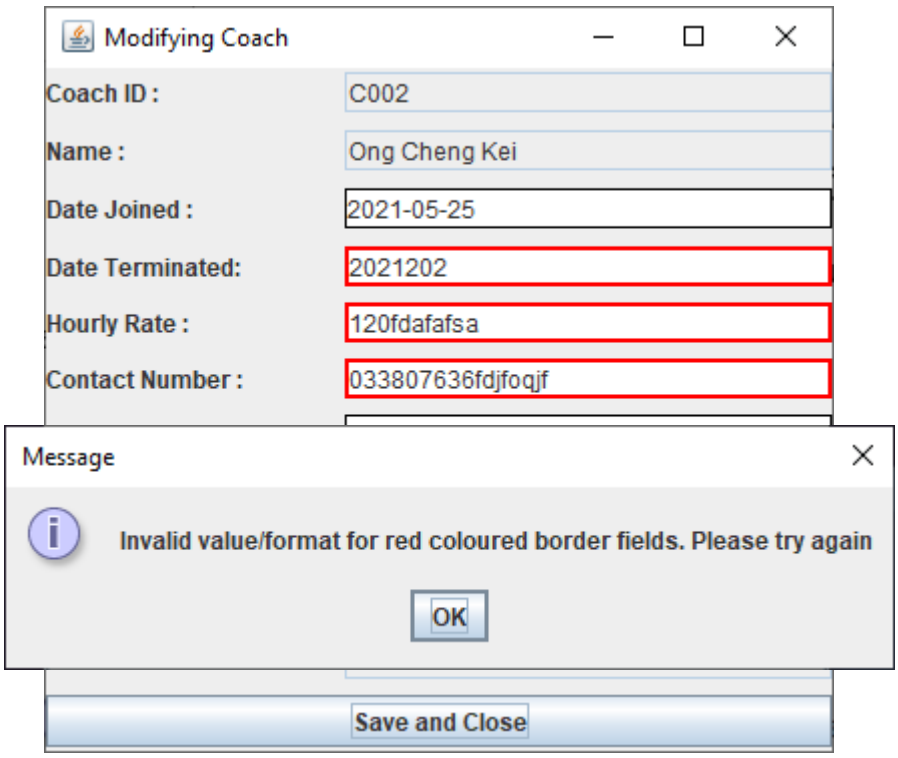
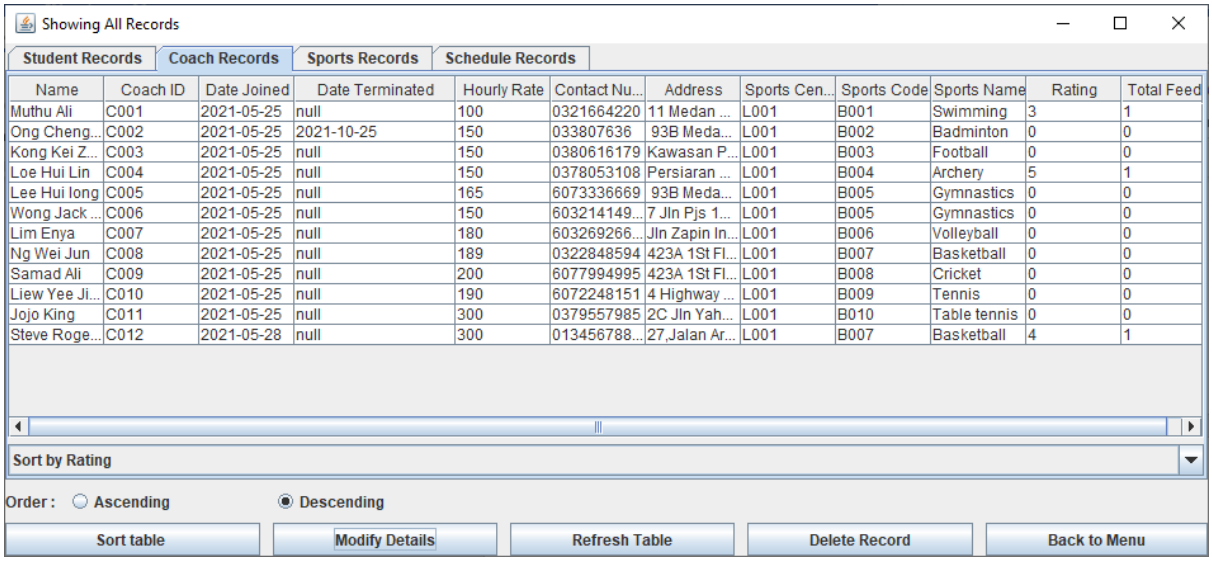



*Figure 68 : Modifying Coach Screen*

In the event of admin passing in invalid values, for the field that can be modified, an error will occur, and a message is displayed that tells the admin the values are invalid or in invalid format. Those values are also highlighted with a bright red border as shown in the figure above.




| Name | Coach ID | Date Joined | Date Terminated | Hourly Rate | Contact Nu... | Address | Sports Cen... | Sports Code | Sports Name | Rating | Total Feed |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Muthu Ali | C001 | 2021-05-25 | null | 100 | 0321664220 | 11 Medan ... | L001 | B001 | Swimming | 3 | 1 |
| Ong Cheng... | C002 | 2021-05-25 | 2021-10-25 | 150 | 033807636 | 93B Meda... | L001 | B002 | Badminton | 0 | 0 |
| Kong Kei Z... | C003 | 2021-05-25 | null | 150 | 0380616179 | Kawasan P... | L001 | B003 | Football | 0 | 0 |
| Loe Hui Lin | C004 | 2021-05-25 | null | 150 | 0378053108 | Persiaran ... | L001 | B004 | Archery | 5 | 1 |
| Lee Hui long | C005 | 2021-05-25 | null | 165 | 6073336669 | 93B Meda... | L001 | B005 | Gymnastics | 0 | 0 |
| Wong Jack ... | C006 | 2021-05-25 | null | 150 | 603214149... | 7 Jln Pjs 1... | L001 | B005 | Gymnastics | 0 | 0 |
| Lim Enya | C007 | 2021-05-25 | null | 180 | 603269266... | Jln Zapin In... | L001 | B006 | Volleyball | 0 | 0 |
| Ng Wei Jun | C008 | 2021-05-25 | null | 189 | 0322848594 | 423A 1St Fl... | L001 | B007 | Basketball | 0 | 0 |
| Samad Ali | C009 | 2021-05-25 | null | 200 | 6077994995 | 423A 1St Fl... | L001 | B008 | Cricket | 0 | 0 |
| Liew Yee Ji... | C010 | 2021-05-25 | null | 190 | 6072248151 | 4 Highway ... | L001 | B009 | Tennis | 0 | 0 |
| Jojo King | C011 | 2021-05-25 | null | 300 | 0379557985 | 2C Jln Yah... | L001 | B010 | Table tennis | 0 | 0 |
| Steve Roge... | C012 | 2021-05-28 | null | 300 | 013456788... | 27,Jalan Ar... | L001 | B007 | Basketball | 4 | 1 |

*Figure 69 : Display all records screen*

Otherwise, if the values provided are valid, then the admin will be led back to this screen, with the selected coach details updated.

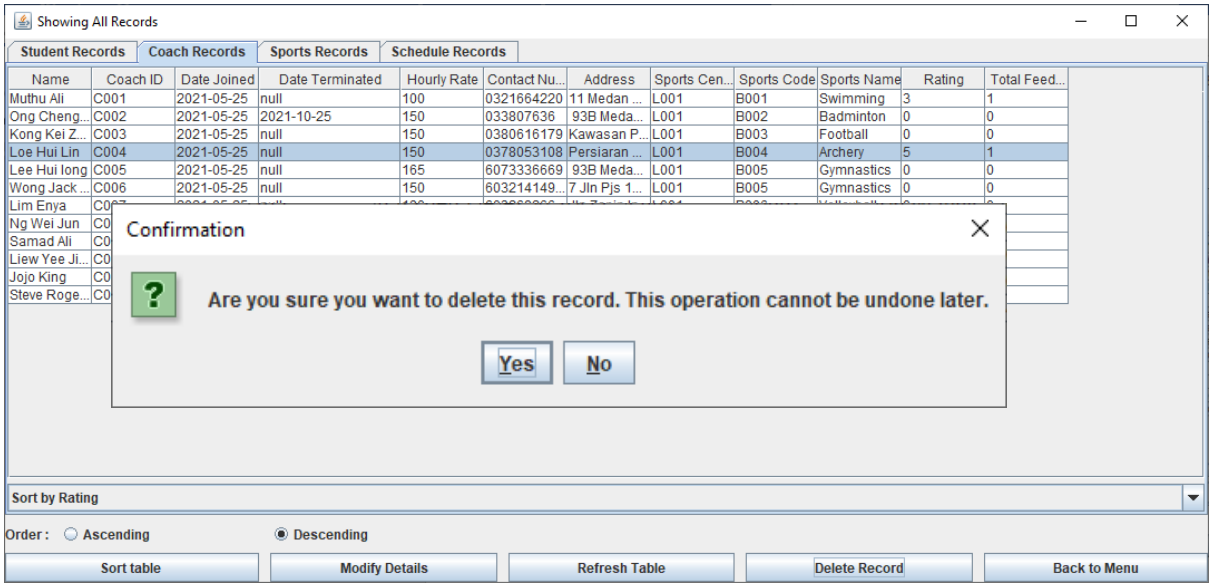



*Figure 70 : Displaying all records screen*

The figure above shows a confirmation message on whether the admin really wants to delete a specific coach record.

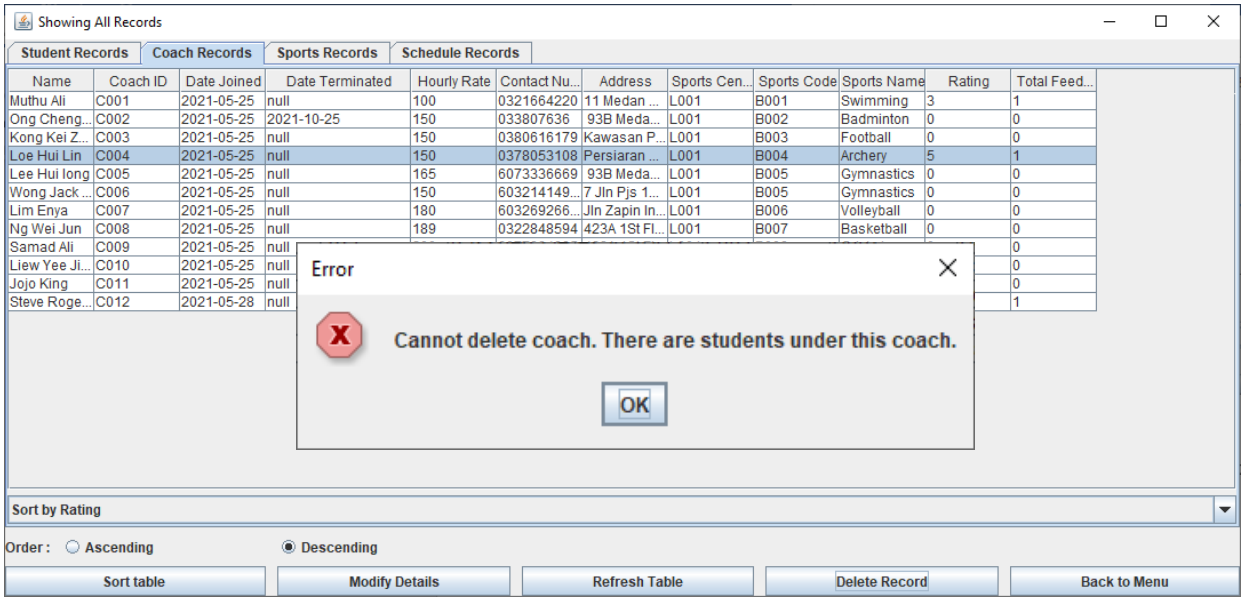



*Figure 71 : Displaying all records screen*

In the figure above, an error message will be displayed if the coach that the admin wants to delete, is currently coaching some students. Therefore, the admin would have to make sure there is no student under the coach before deleting it.

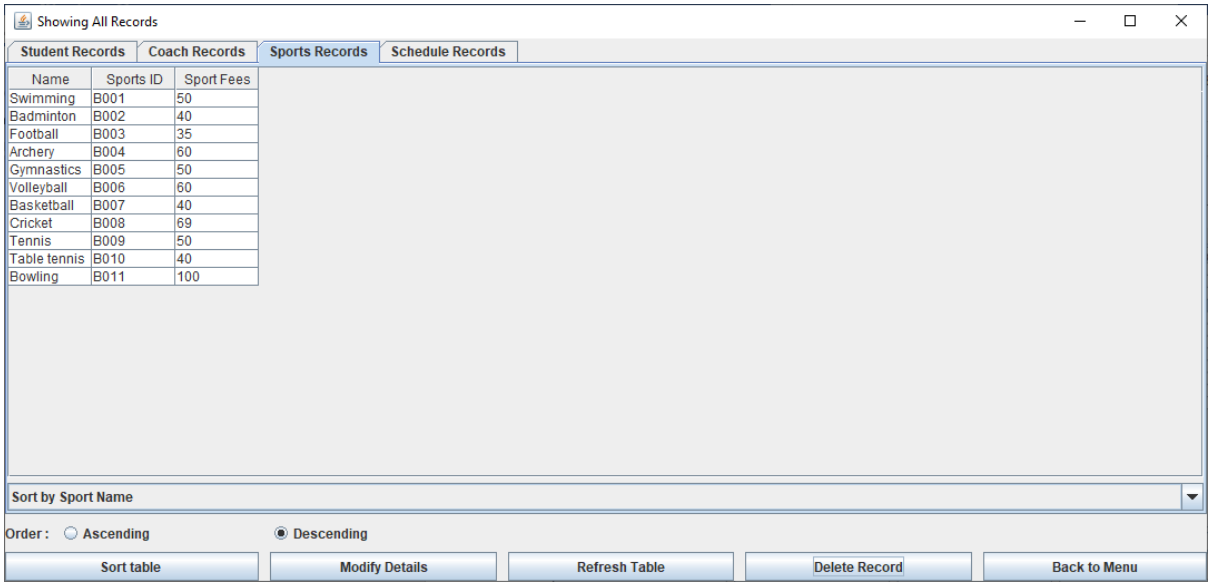



*Figure 72 : Displaying all records screen*

If the admin selects the sports records panel, then a table displaying all the sports records are shown to the admin.

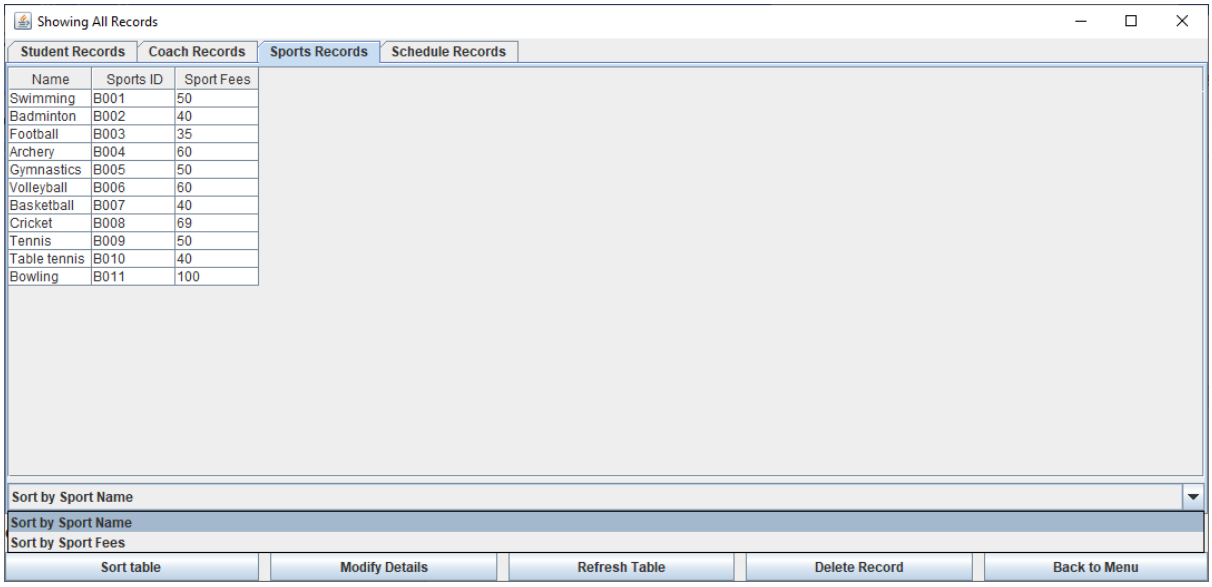



*Figure 73 : Displaying all records screen*

In the figure above, the admin also can choose what type of sorting method to be applied to sort the table displayed. The admin can choose between sort by sport name, and sort by sport fees as displayed in the figure above.

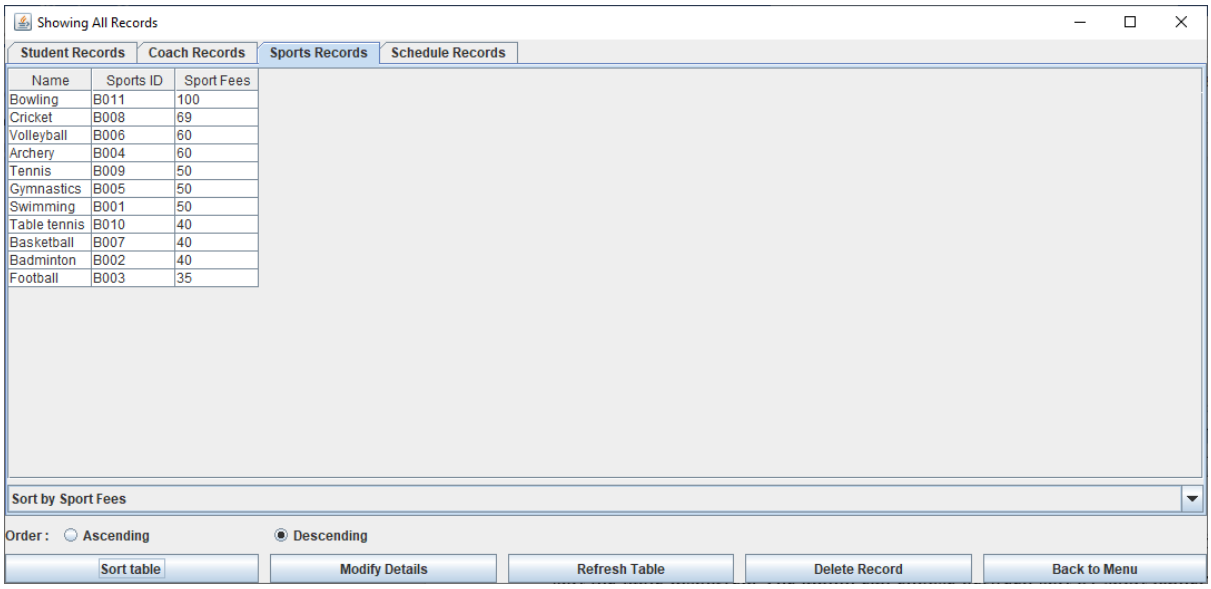



*Figure 74 : Displaying all records screen*

In the screen above, the sports records are sorted by sport fees in descending order, after the admin has pressed the sort table button with a specified order.

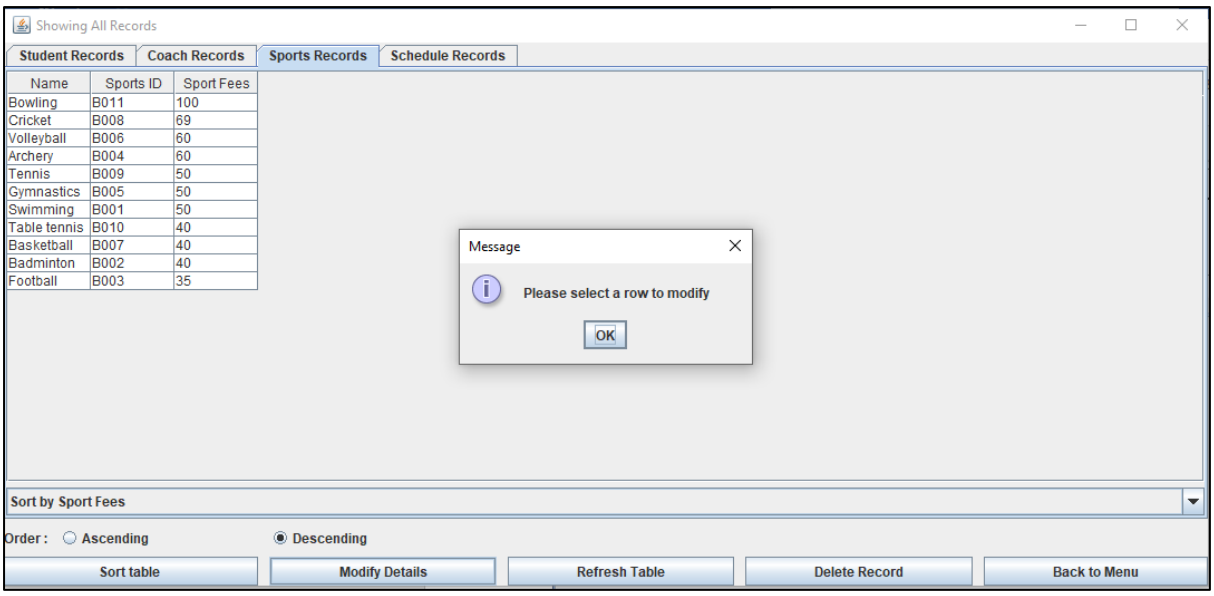



*Figure 75 : Displaying all records screen*

A message is displayed to ask the admin to choose a row in order to modify a record details, if the admin clicked the modify details button without selecting any row.


*Figure 76 : Modifying sports screen*

Once the admin selected a row to modify, then this screen shown above will pop up. The admin is only allowed to change the sports fees of the sport to prevent deep complexities.

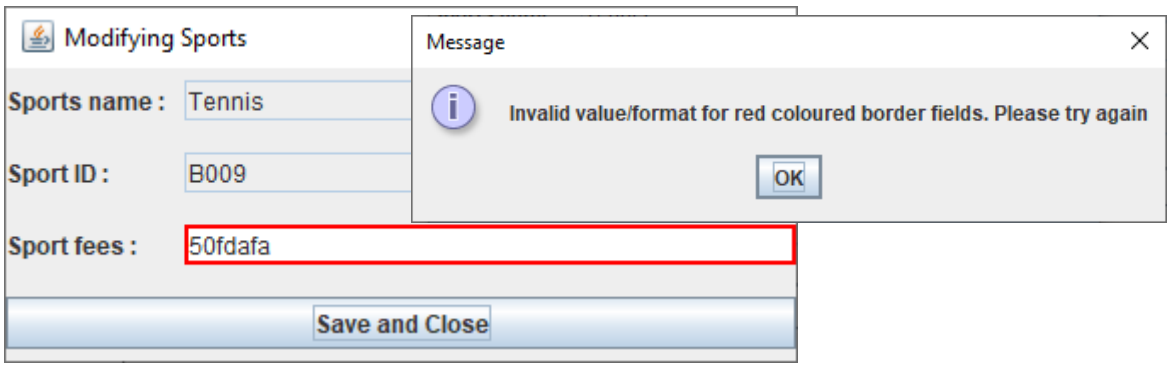



*Figure 77 : Modifying sports screen*

If the admin did not enter a valid sport fees value (integer) then an error message is pop up, and the sport fees border is highlighted with bright red.

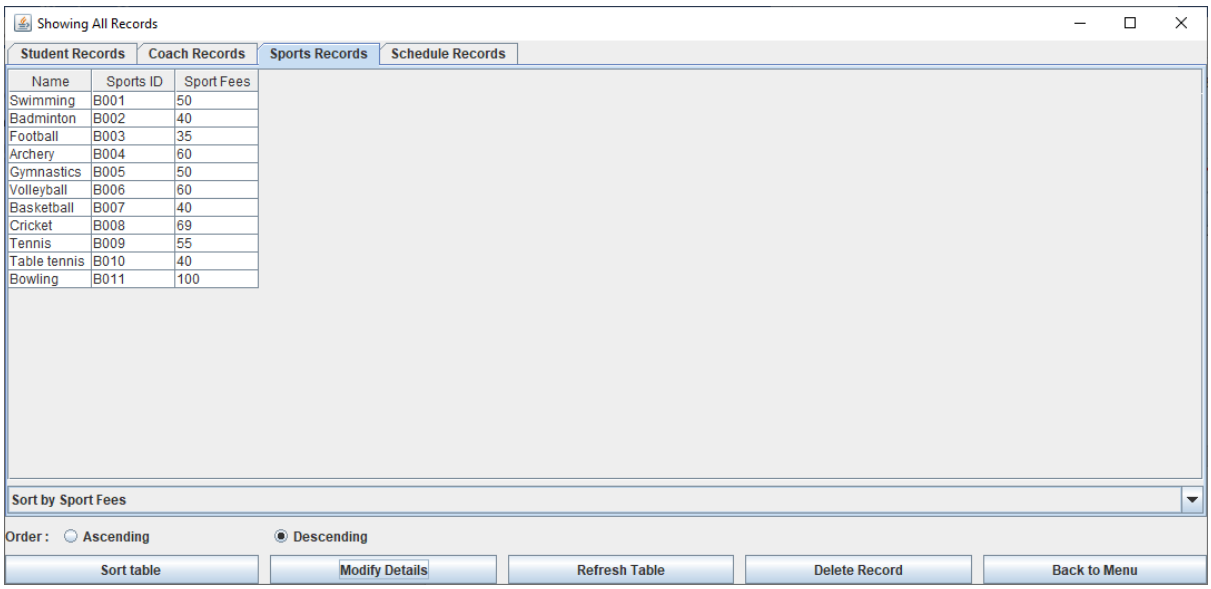



*Figure 78 : Displaying all sports record screen*

After the admin has entered a valid value for the sports fees for modification, then the admin will be directed back to the display all records screen.

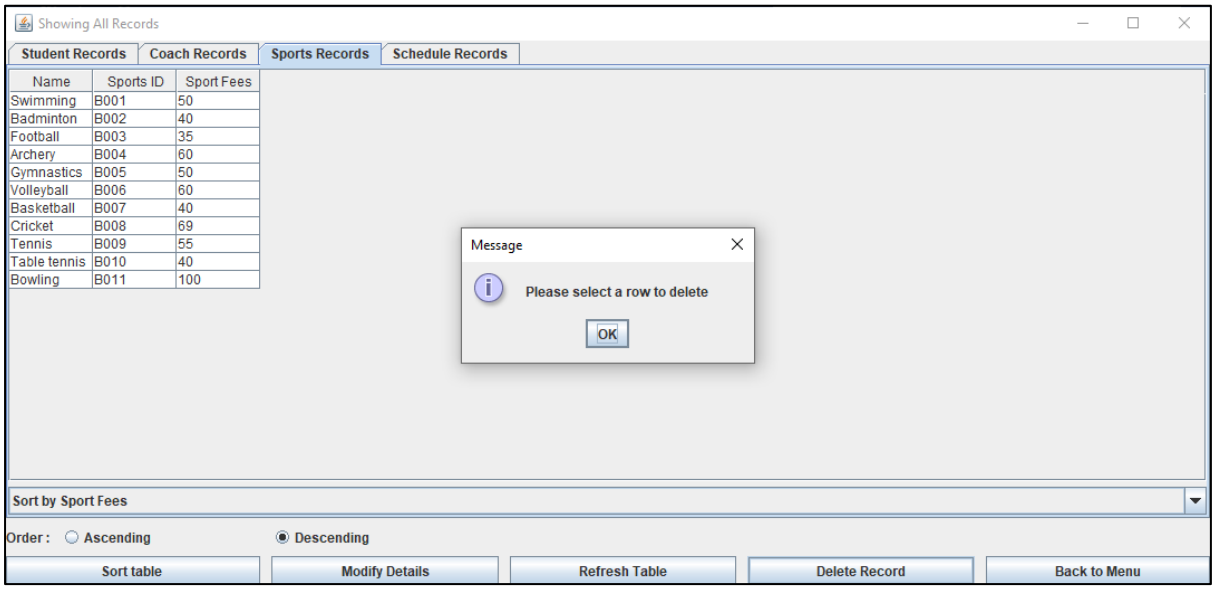



*Figure 79 : Display all sports record screen*

If the admin wants to delete a sport, the admin must specify which row to delete, otherwise, a message is shown above to ask the admin to select a row to delete.

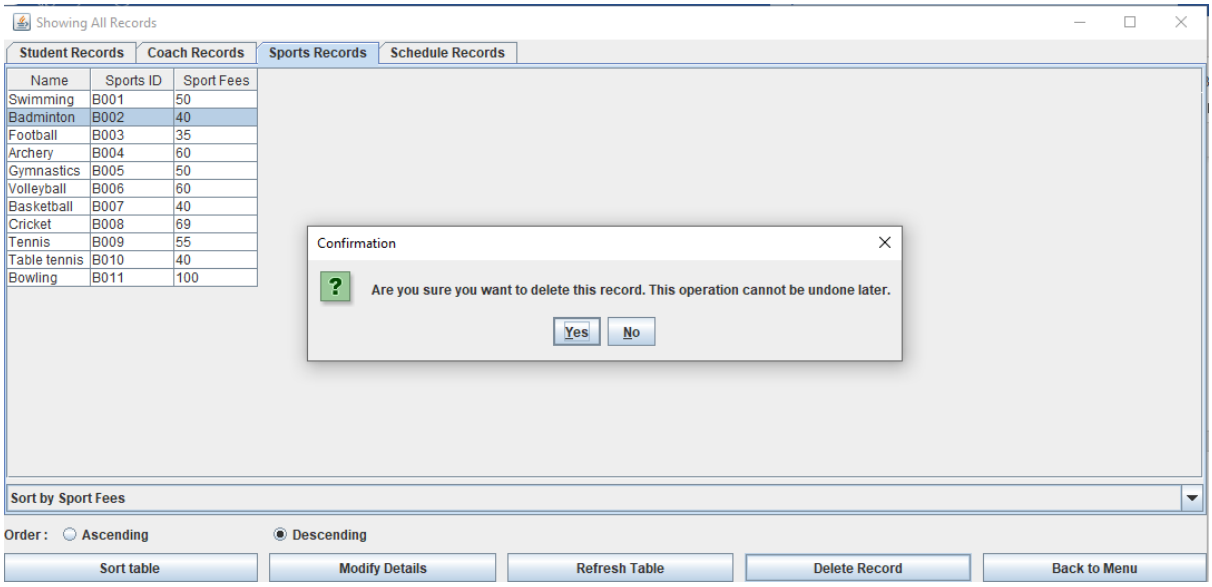



*Figure 80 : Display all sports records*

Once a sport record is selected, the admin will be prompted whether to really delete the record as the operation cannot be undone later.

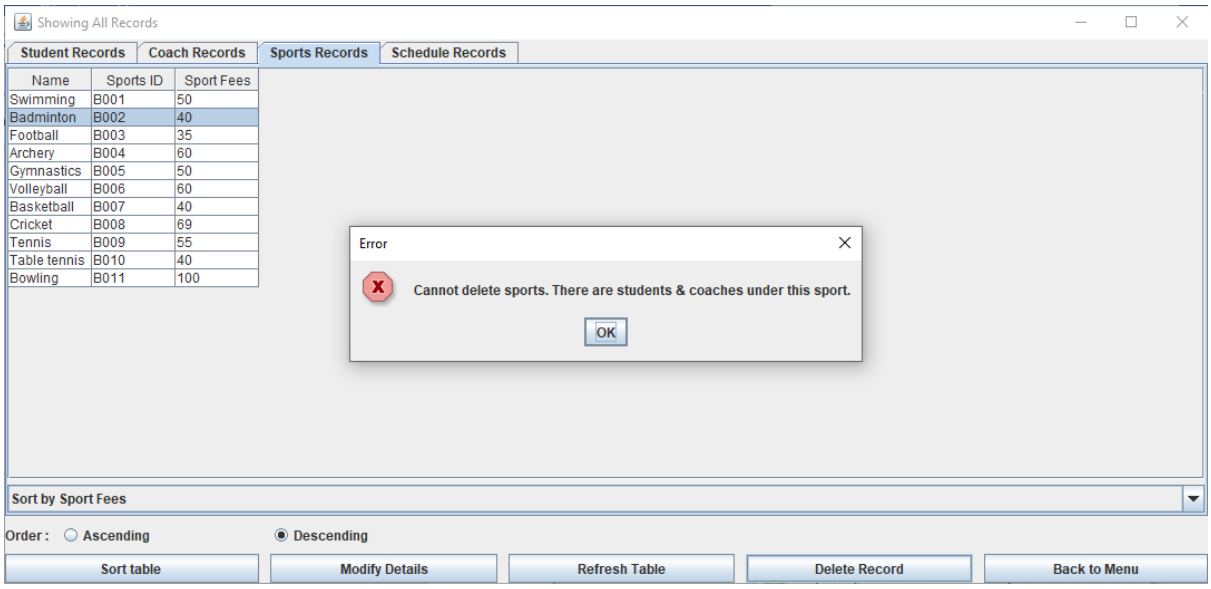



*Figure 81 : Display all sports records*

If the admin proceeds with deleting the sports, the system will check whether any students or coach is under the sports. If yes, then the system will not allow the sports to be deleted.

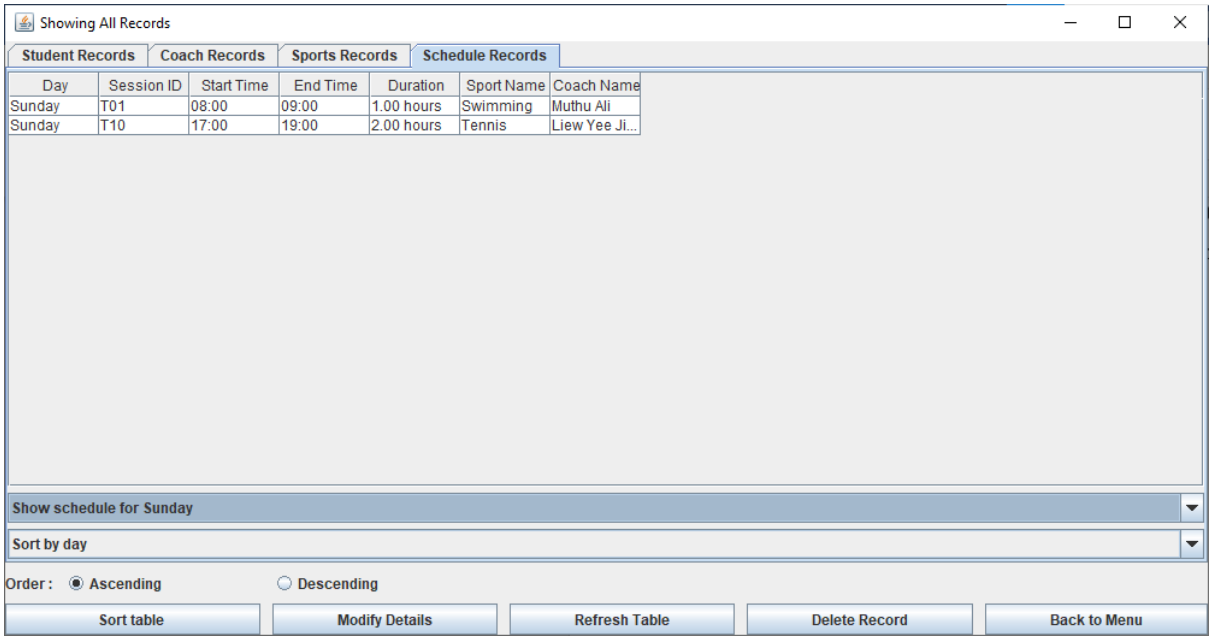



*Figure 82 : Displaying all schedule records*

In this screen shown in the figure above, the admin can view all the schedules in the sport center.

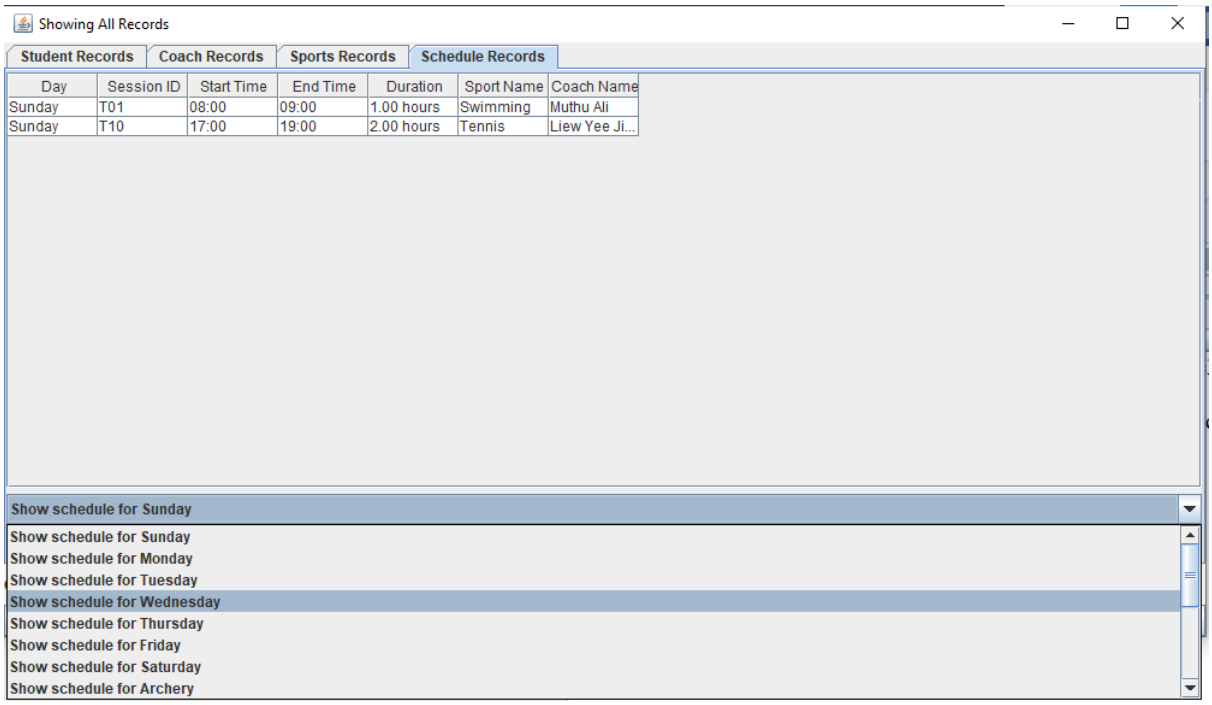



*Figure 83 : Displaying all schedule records*

The admin can select which schedule to view in the drop-down menu prepared on the screen.

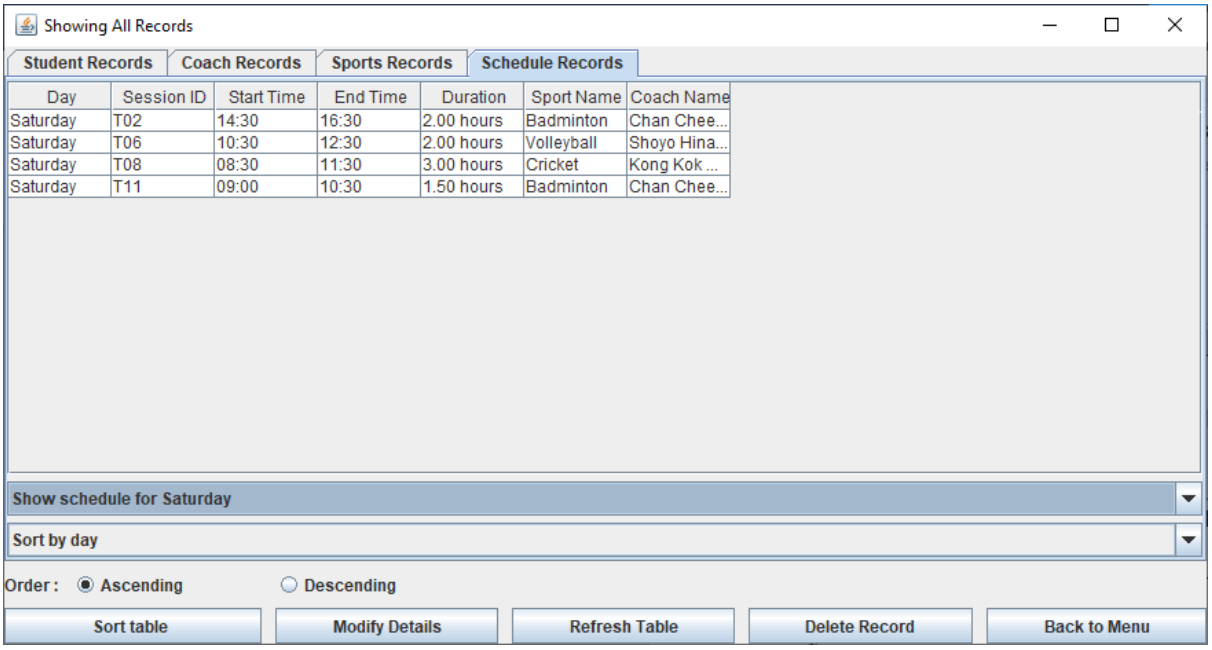



*Figure 84 : Displaying all schedule screen*

The figure above shows the screen that displays all the sessions for Saturday.

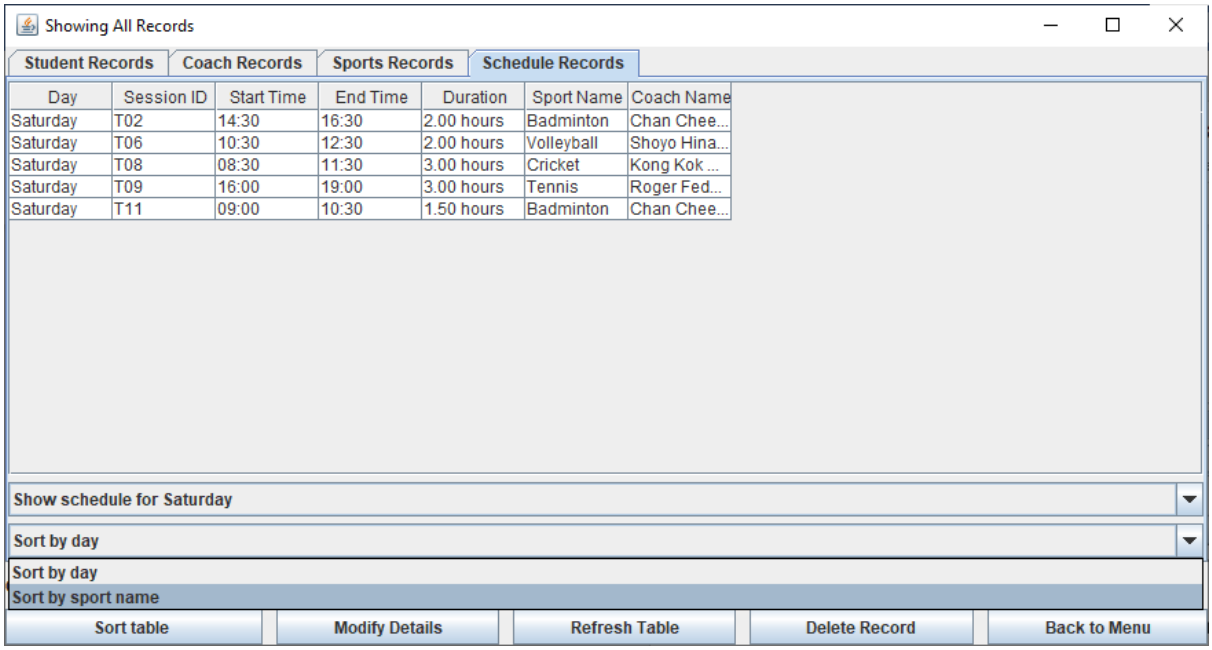



*Figure 85 : Displaying all schedule screen*

Besides that, the admin can also choose to sort the table by day or by sport name, in a specified order.

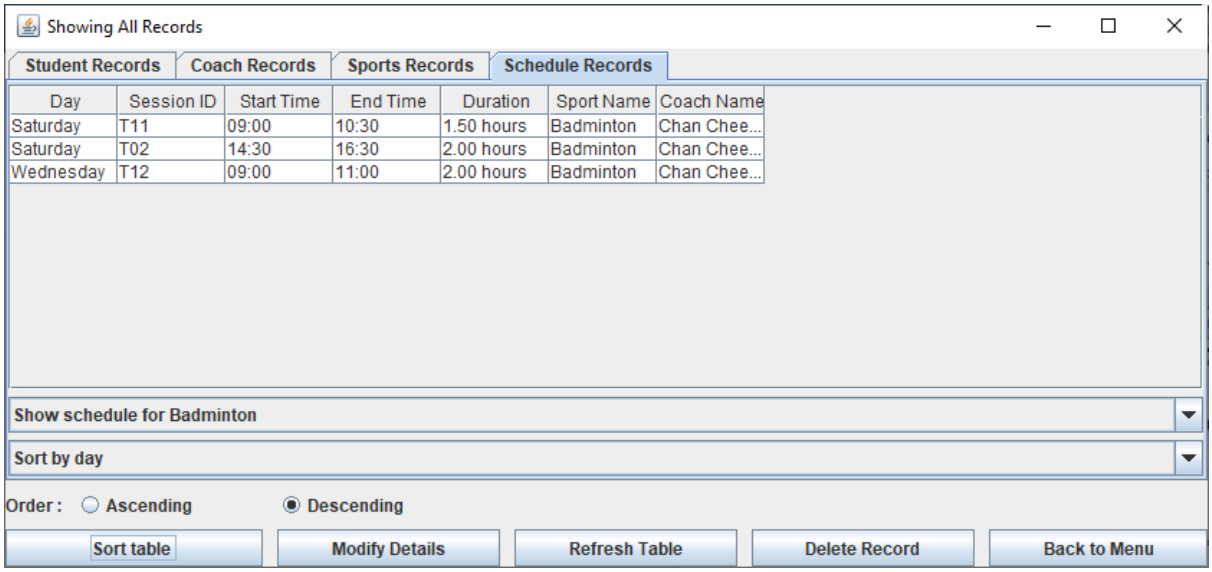



*Figure 86 : Display all schedule records*

The screen above shows the schedule record is sorted by day in descending order.

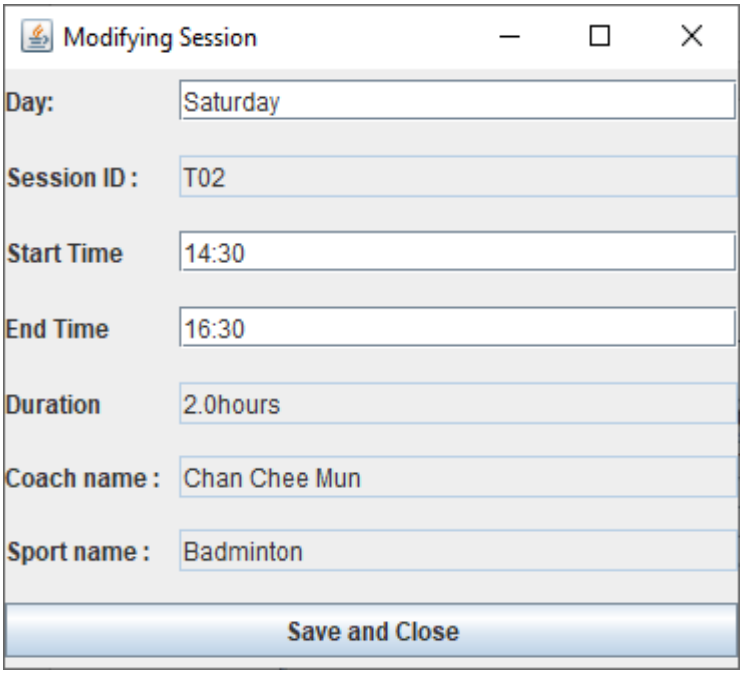



*Figure 87 : Modifying session record*

Similarly, to the sports records, the admin needs to select a row before modifying the session record. In this case, the admin is allowed to modify the day, start time, and end time of a session.

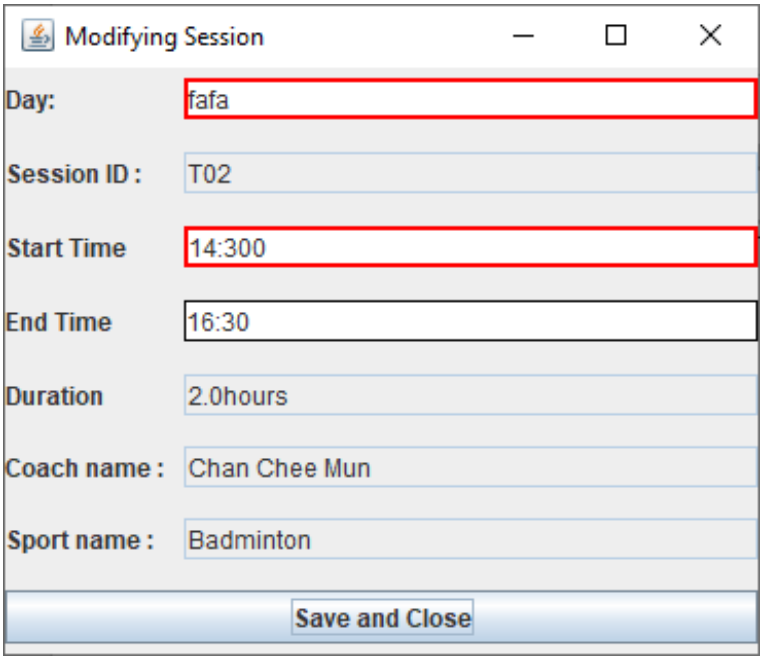



*Figure 88 : Modifying session record*

If the admin gives an invalid value the surrounding border will change to red, indicating the value is invalid and requires change.

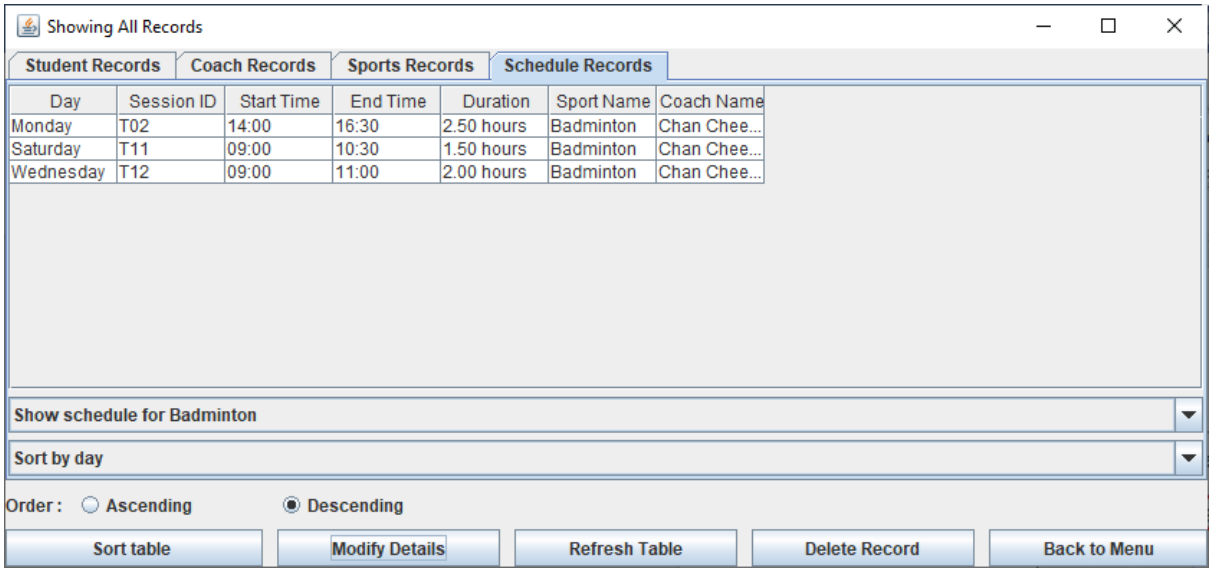



*Figure 89 : Displaying schedule record*

After successfully modifying the schedule records, the admin will be directed back to the display all schedule records screen.

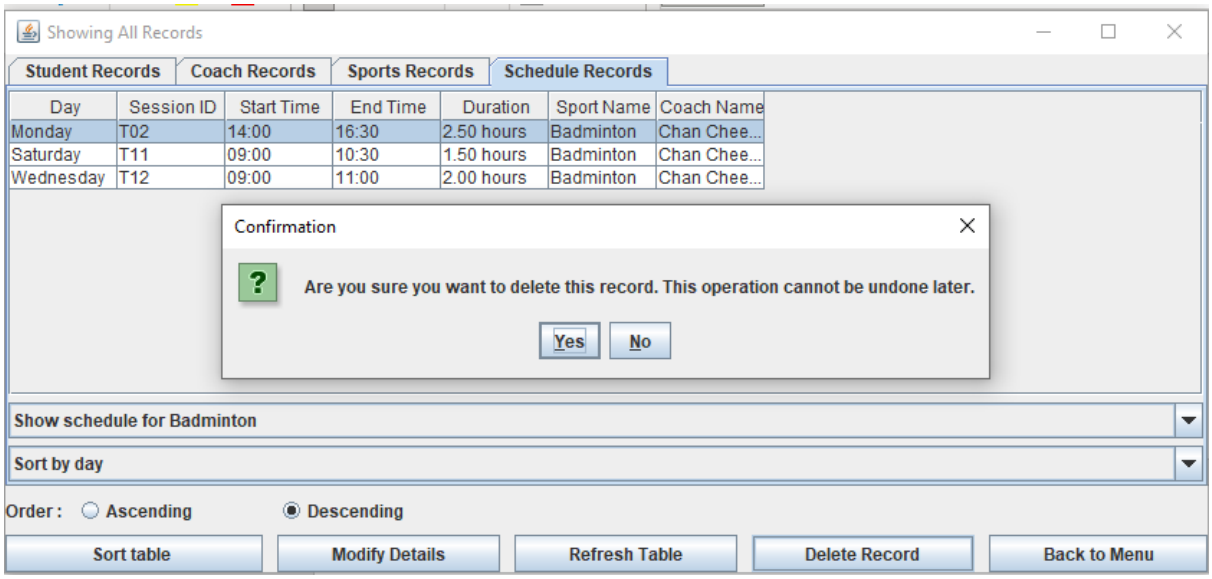



*Figure 90 : Displaying all schedule records*

Similar to other records, admin will be greeted with a prompt message when the admin attempt to delete one of the session records. If the admin selects yes, then the system will proceed to delete the record, otherwise no.

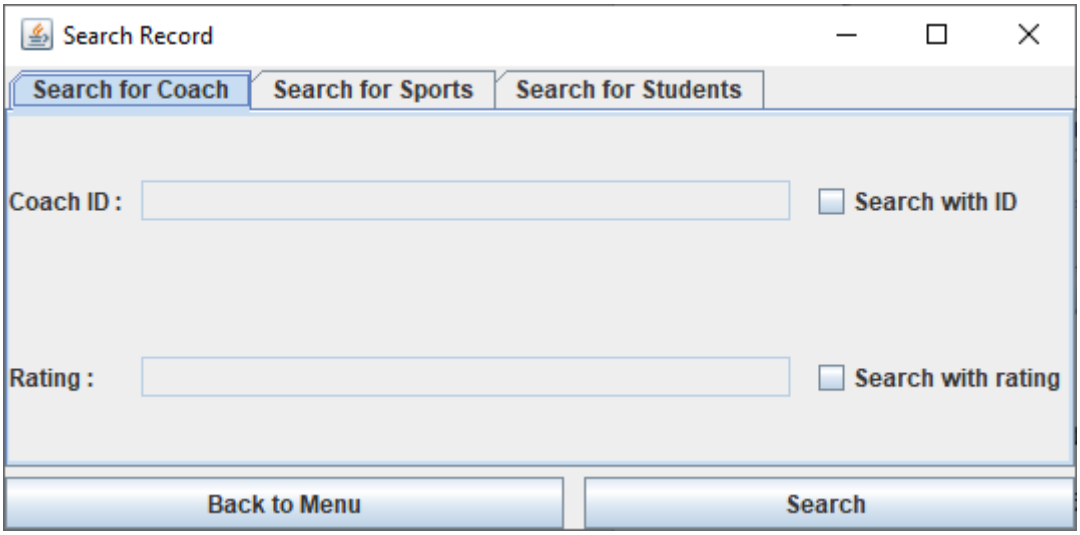



*Figure 91 : Search Record Screen*

The figure above shows the search record screen. The admin has to tick at the checkboxes to select how to search for coaches.

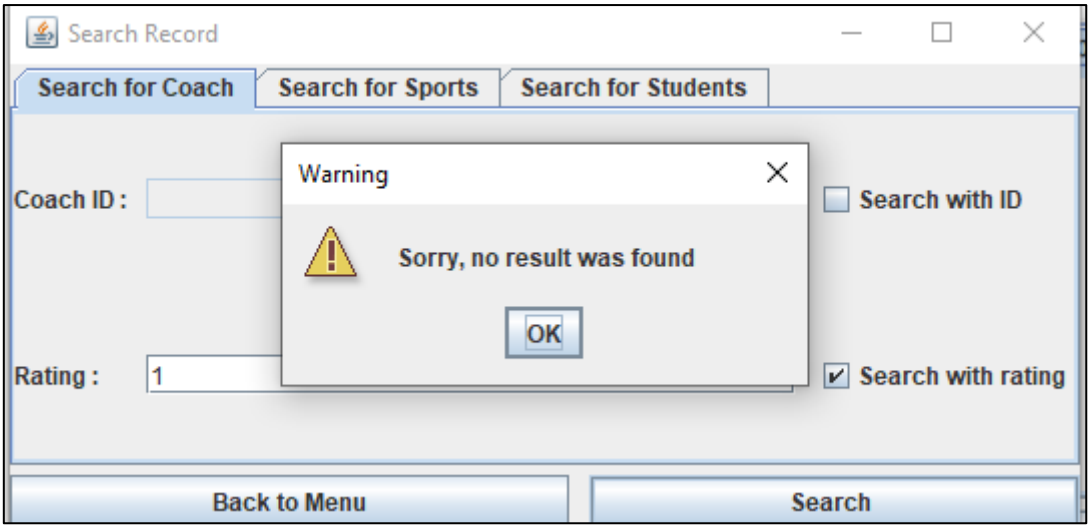



*Figure 92 : Search record screen*

If there are no coaches found, then a pop-up message is displayed above indicating no result was found.

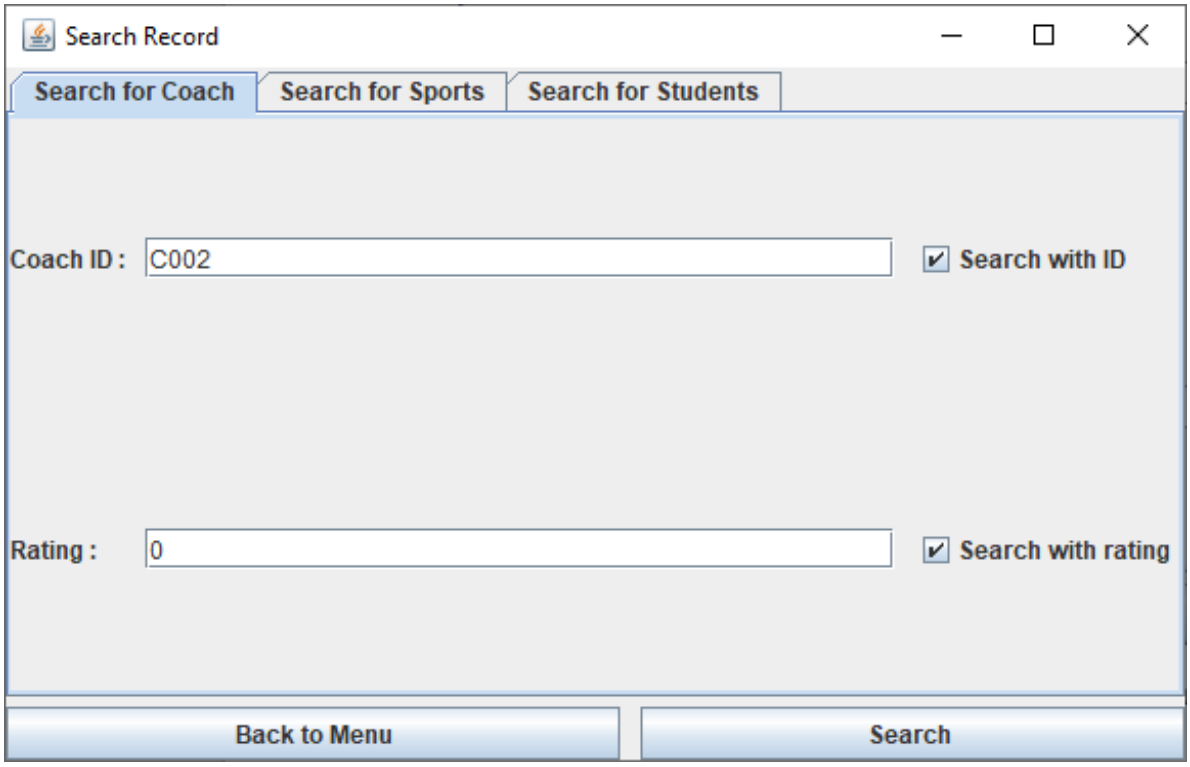



*Figure 93 : Search record screen*

In the figure above, the admin tries to search for the coach ID above.

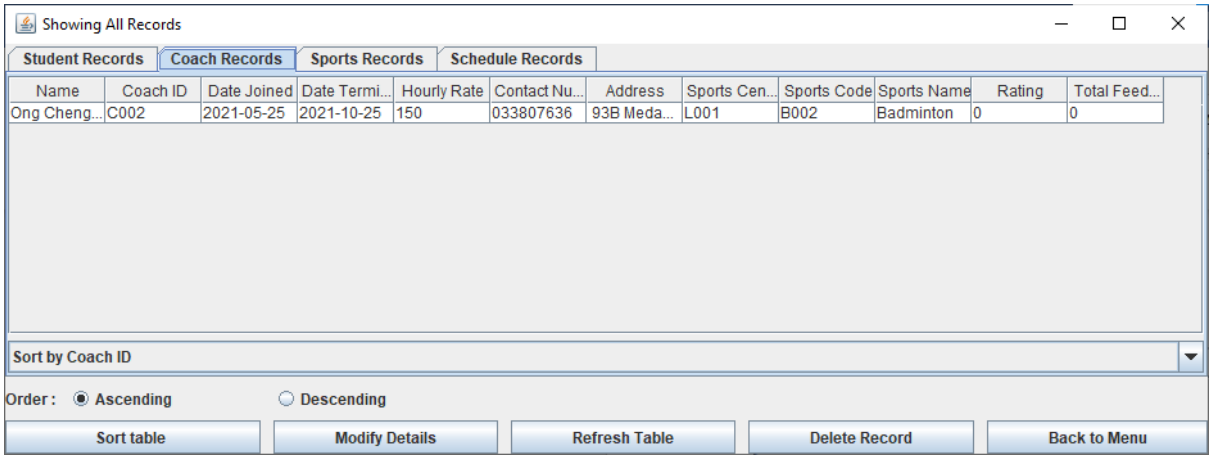



*Figure 94 : Showing all records screen*

If there are matching coaches with the rating and ID specified above, then the admin will be led back to the display all records screen, but it is only showing the search results. From here the admin can modify the row, delete records, and sort the table, same as what has been shown above the display all records screen. However, if the admin decides to get the whole list of coach records back, the admin can press the refresh table button which gets back the entire list of records.

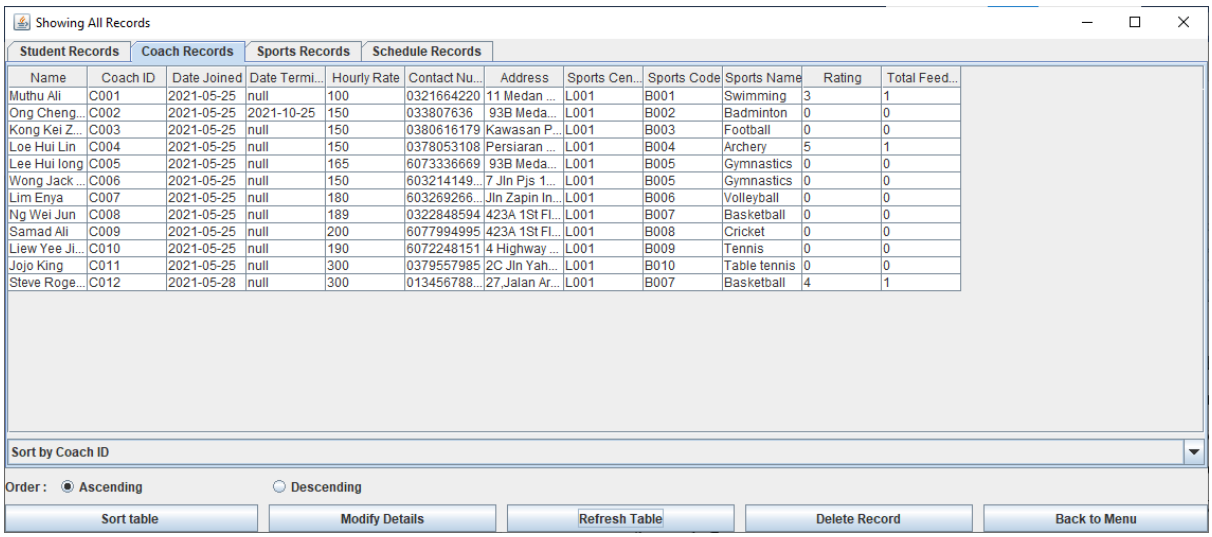



*Figure 95 : Showing all records screen*

As shown above, the entire list of coaches is shown after pressing the refresh table button. Next, searching for student records, and sports records will work similarly. Essentially, if there is a record found, the admin will be directed back to the display all record screen to show the search results, then the admin can perform the same set of functions such as sorting the table, modifying the row details, and deleting the records. If the admin wants to retrieve back the entire list of records, then the refresh table button will do the job.

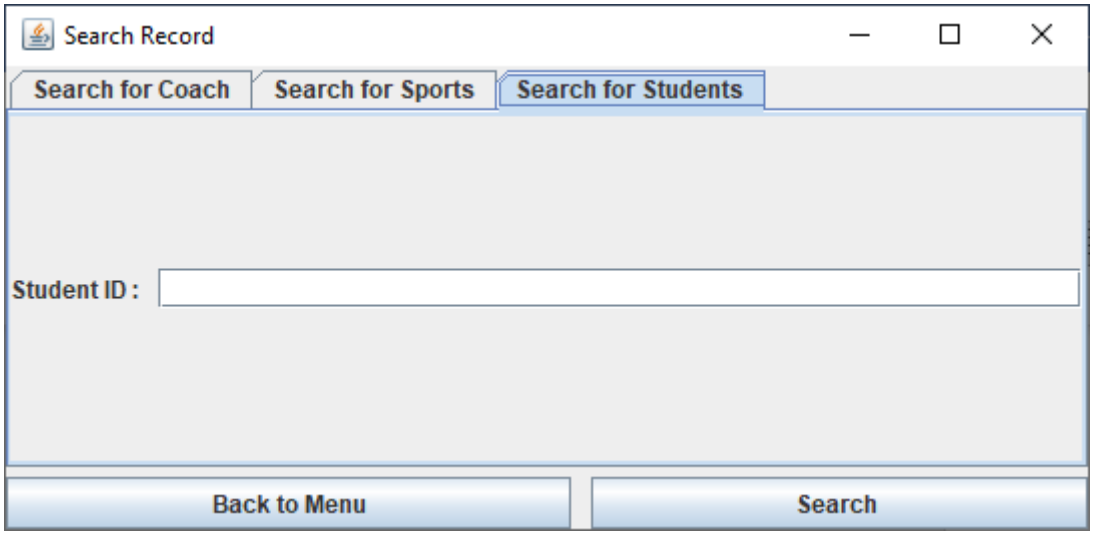



*Figure 96 : Search for students records screen*

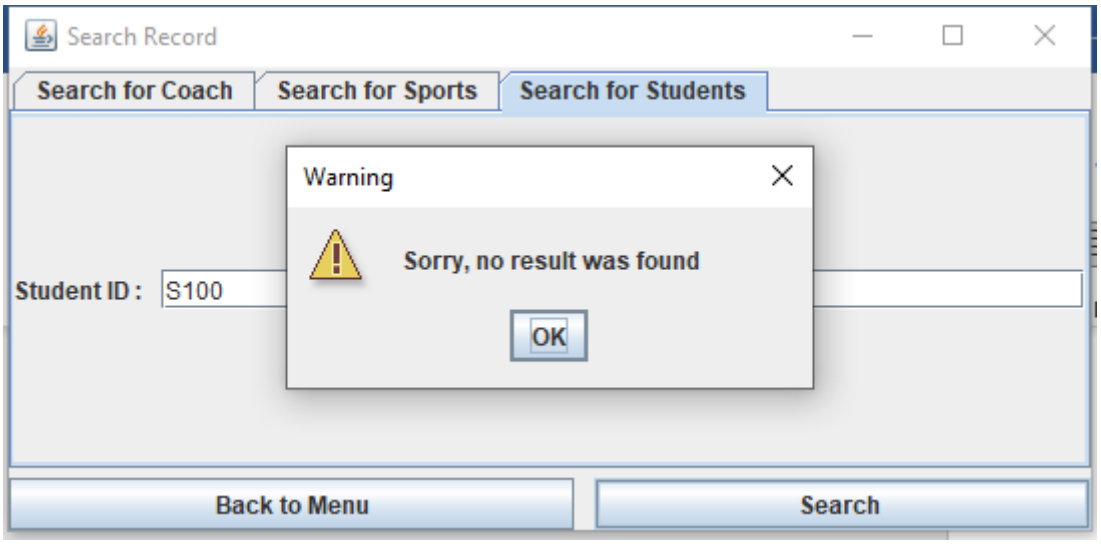



*Figure 97 : A pop up warning message will be shown if no records are found*

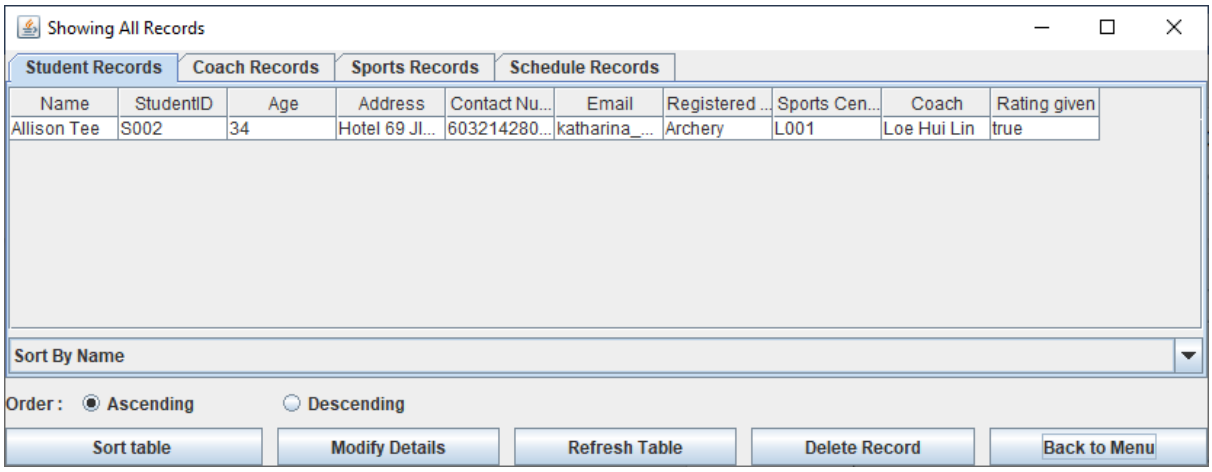



*Figure 98 : Screen showing the search results*

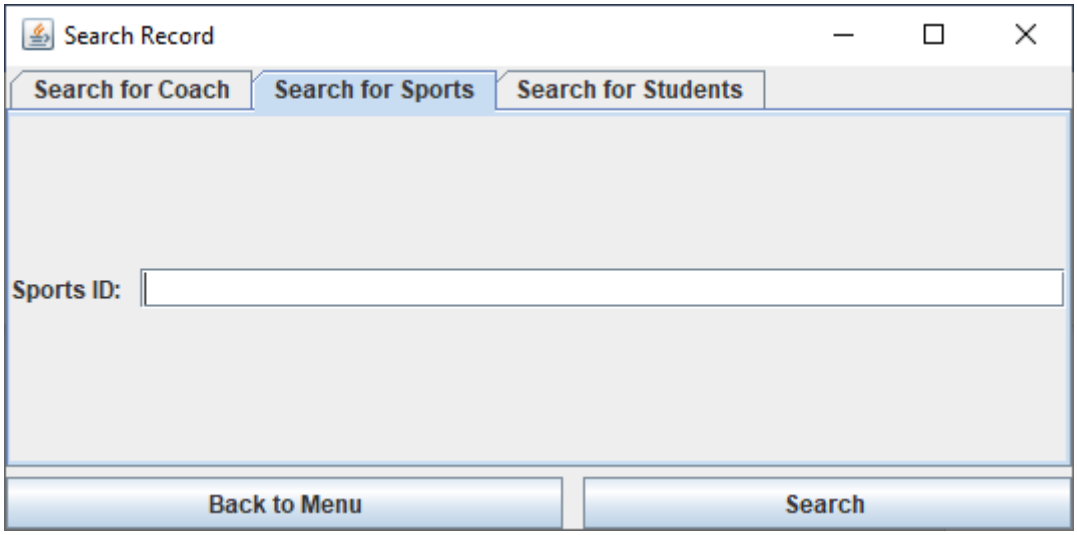



*Figure 99 : Search for Sports screen*

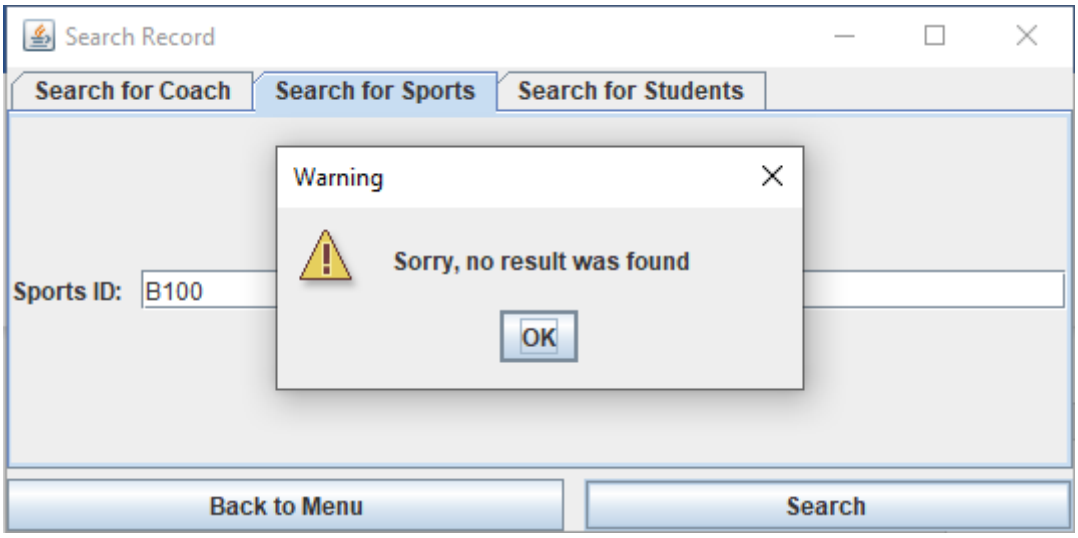



*Figure 100 : A warning pop up message showing there are no records found for specified sports ID*

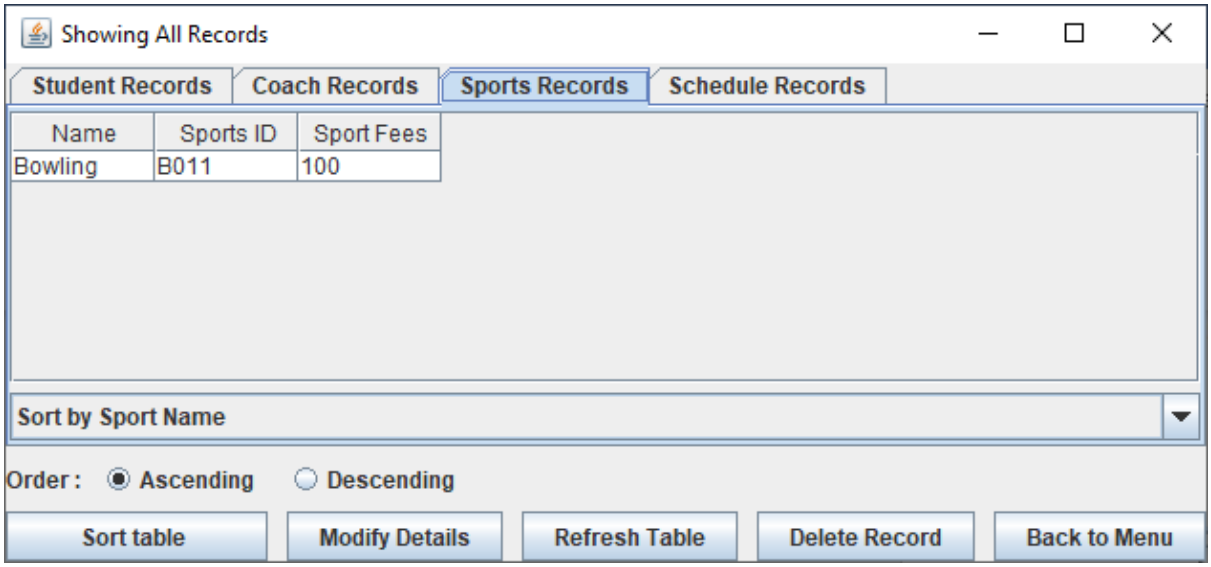



*Figure 101 : Screen showing the search results*

Registered Student

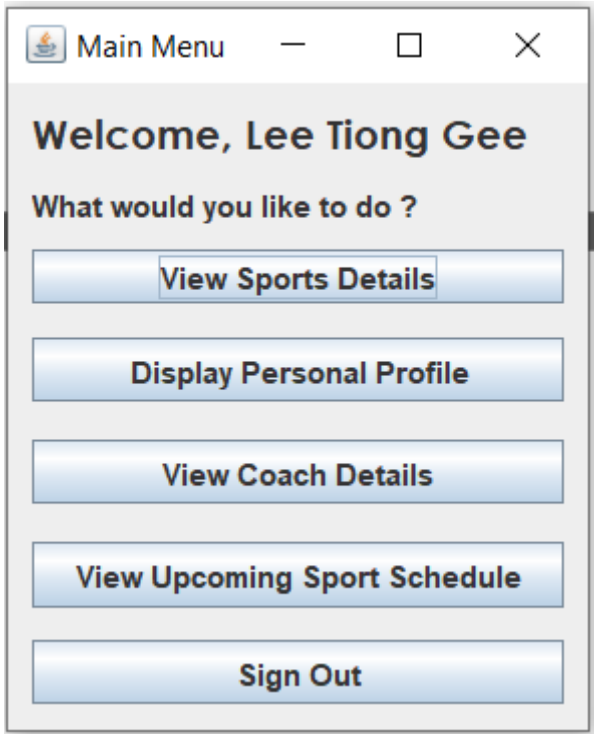



*Figure 102 : Main menu screen for students*

The figure above shows the main menu screen that students will see after they have successfully logged in with an existing account. From here, they can choose to either to choose to view details of all sports available in the sports centre, view and modify their personal profile information, view their coach details , view all upcoming sports schedules, or sign out from the system, where they will be brought back to the login screen.

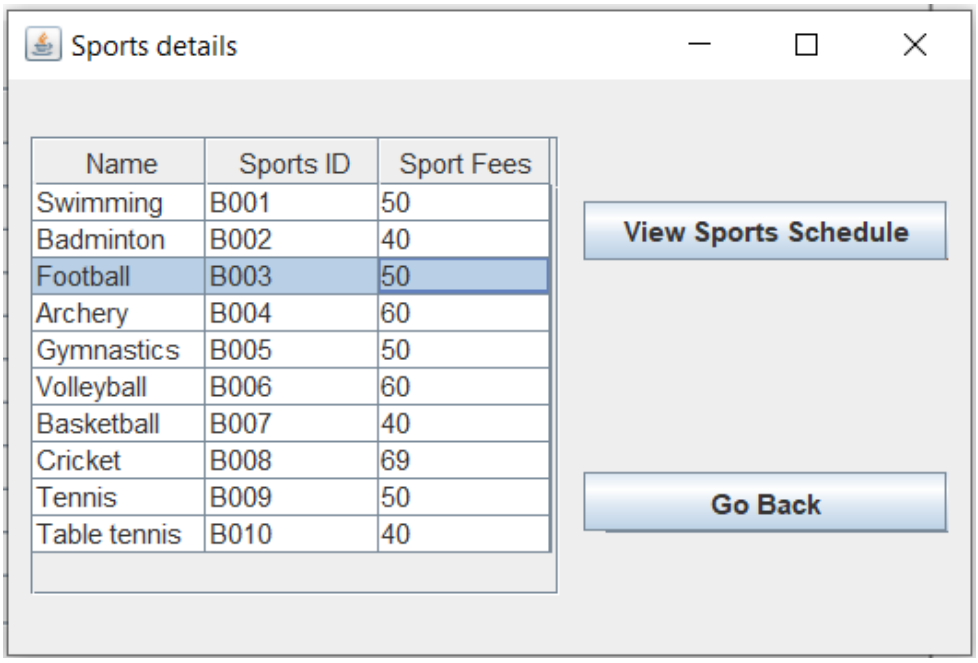



*Figure 103 : View Sports Detail Screen for students*

After selecting the "View Sports Details" option from the menu, the user will be brought to a separate page that displays information of all sports including the sports name, ID, and fees. If the students want to check the available time schedule for a specific sport, they can click on the row containing that sport and then clicking the "View Sports Schedule", otherwise they can click the "Go Back" button to be returned to the student main menu.

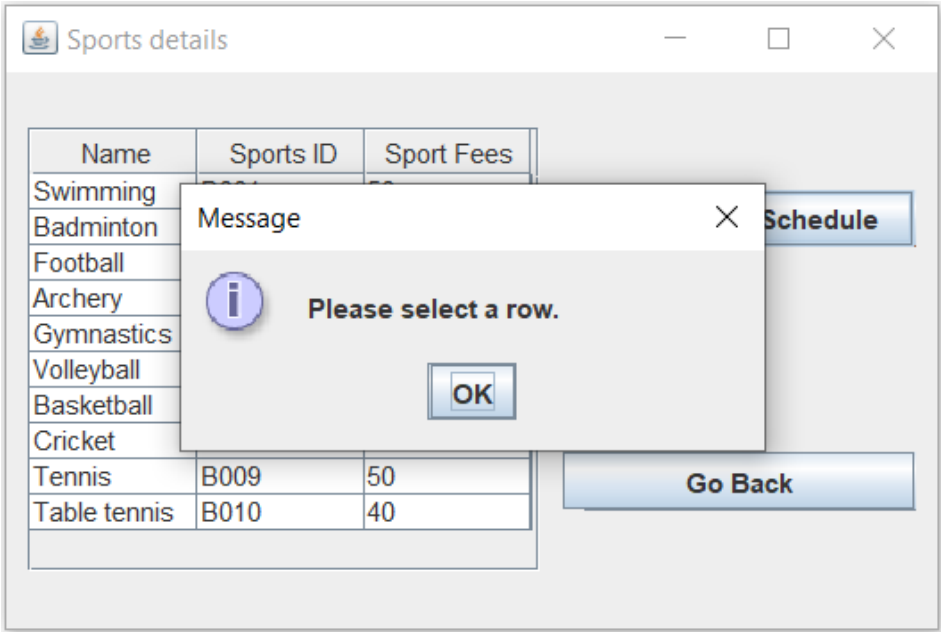



*Figure 104 : Error message if no sports row was selected*

In the scenario that the user does not select a specific row to view the sports schedule for, the system will have an error message popup requesting the user to select a row in order to view its schedule.

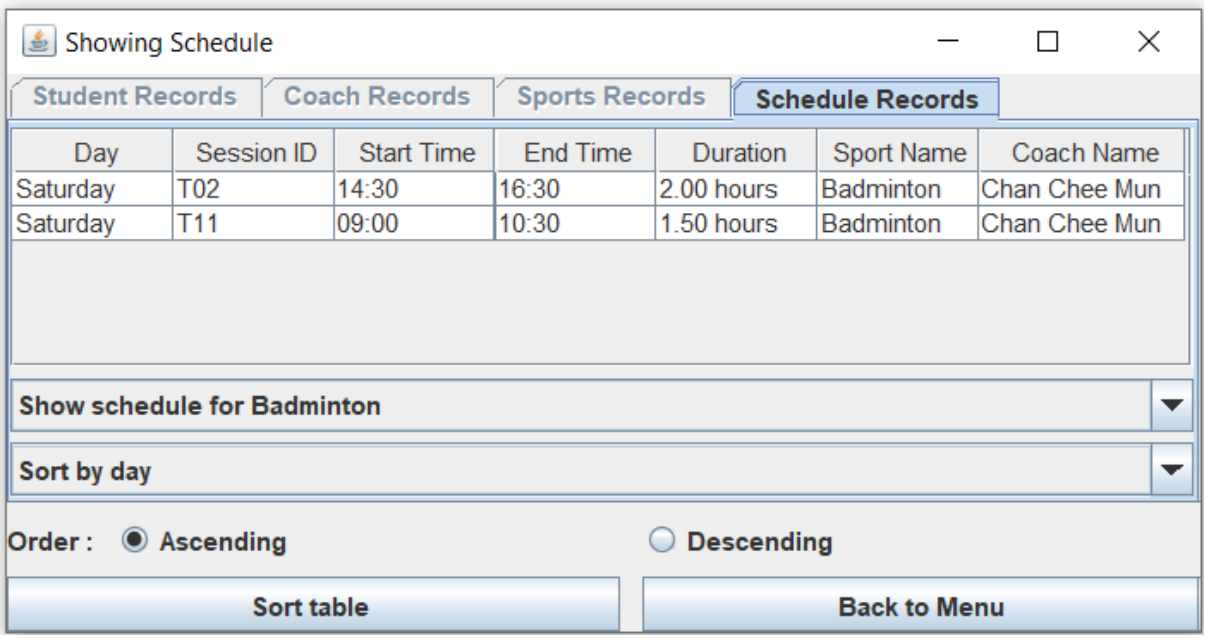



*Figure 105 : Viewing schedule of specific sports*

After selecting a specific sport to view its schedule for, the user will be brought to another page that shows all records of the schedule for that selected sport. However, unlike the admin, the student will not be able to access the tabs at the top of the page that show other students, coaches, and sports records. After seeing all the available sports schedule, the user can click the “Back to Menu” button to be returned to the main menu.

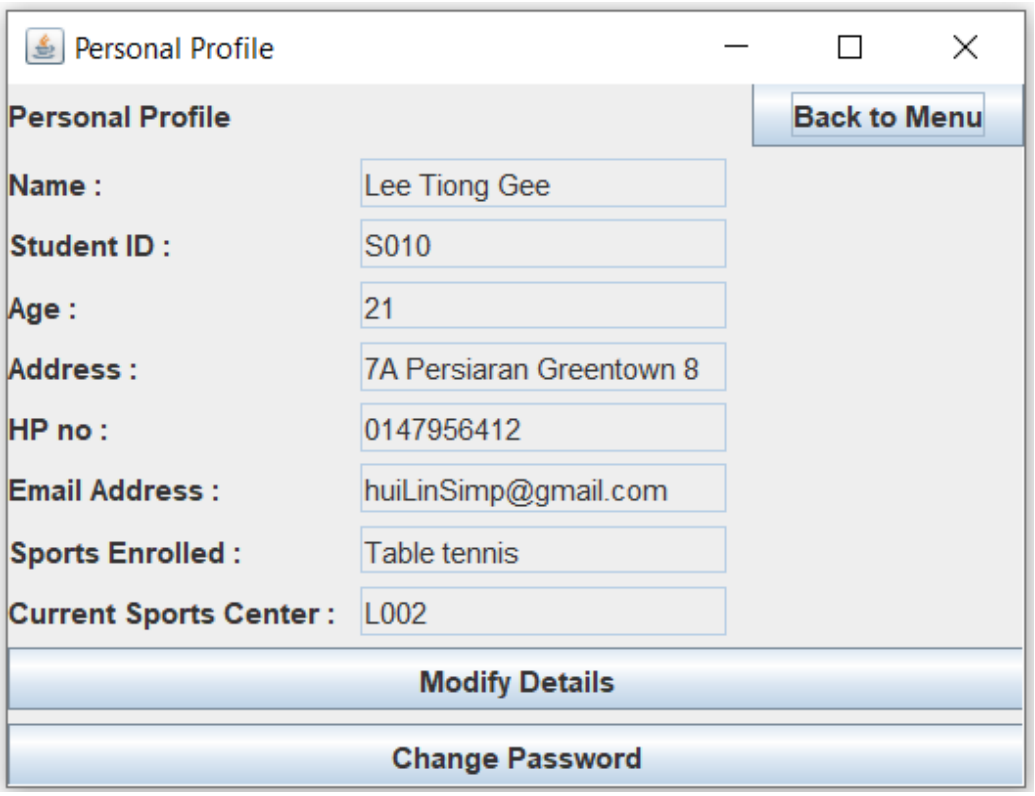



*Figure 106 : Student Display Personal Profile page*

If the user clicks the "View Personal Profile" option in the menu, they will be brought to the page shown above, which displays all the student's personal information. From here, the user can choose to go back to the main menu by clicking the "Back to Menu" button, or modify their personal details using the "Modify Details" option and change their account password with "Change Password" option.

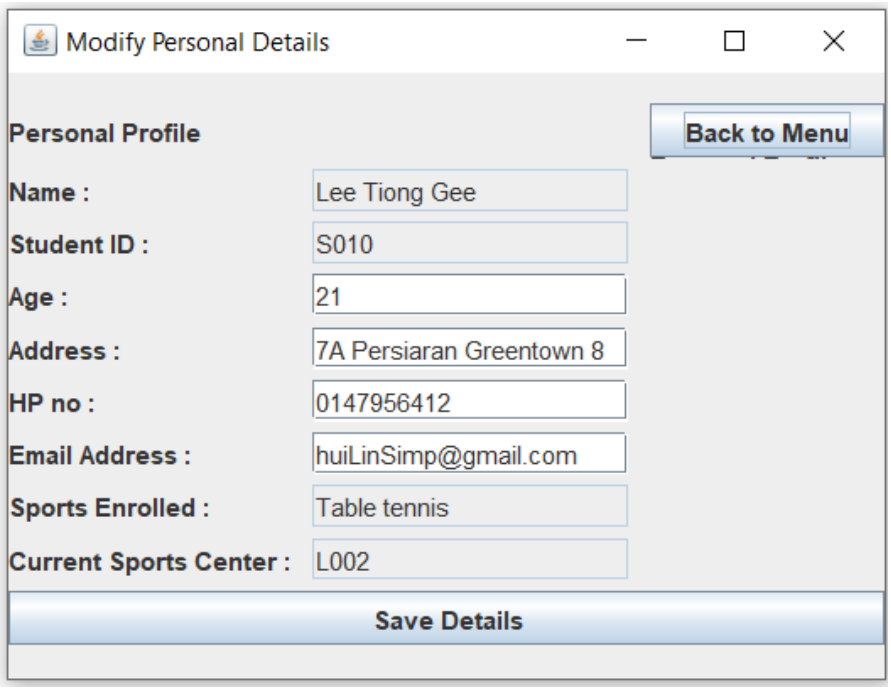



*Figure 107 : Student Modifying Personal Details*

If the student chooses to modify their personal details, the age, address, phone number, and email address text field will light up and allowing the user to edit the information. The other details that remained a gray background are the details that the students are not allowed to change. After the students have entered their new details, they can click the "Save Details" option to update their personal profiles.

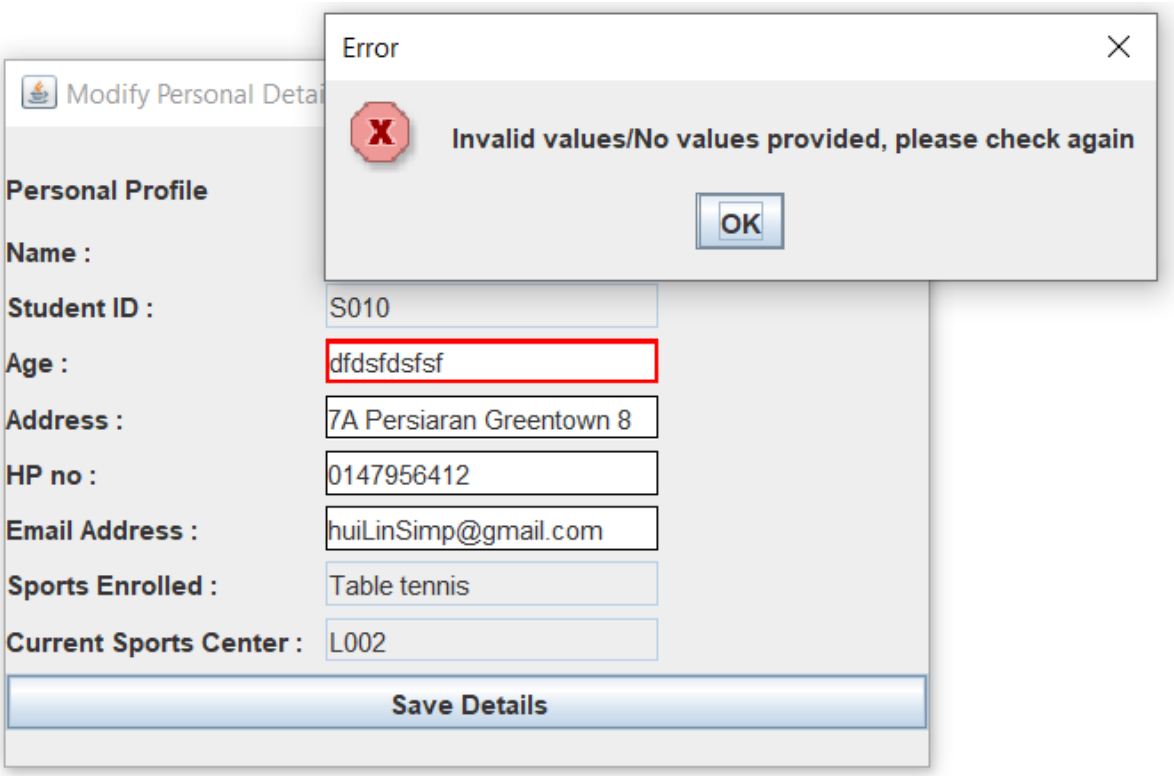



*Figure 108 : Error popup when modifying student personal details*

However, if the student attempts to enter irrelevant or non-accepted input values when modifying their details, the system will display an error message, prompting the user to make changes to that specific field. An example is shown above, if the user tries to update the age field by entering a line of strings instead of numbers, the system will highlight the text field border in red and displaying an error message, prompting users to change the input entered.

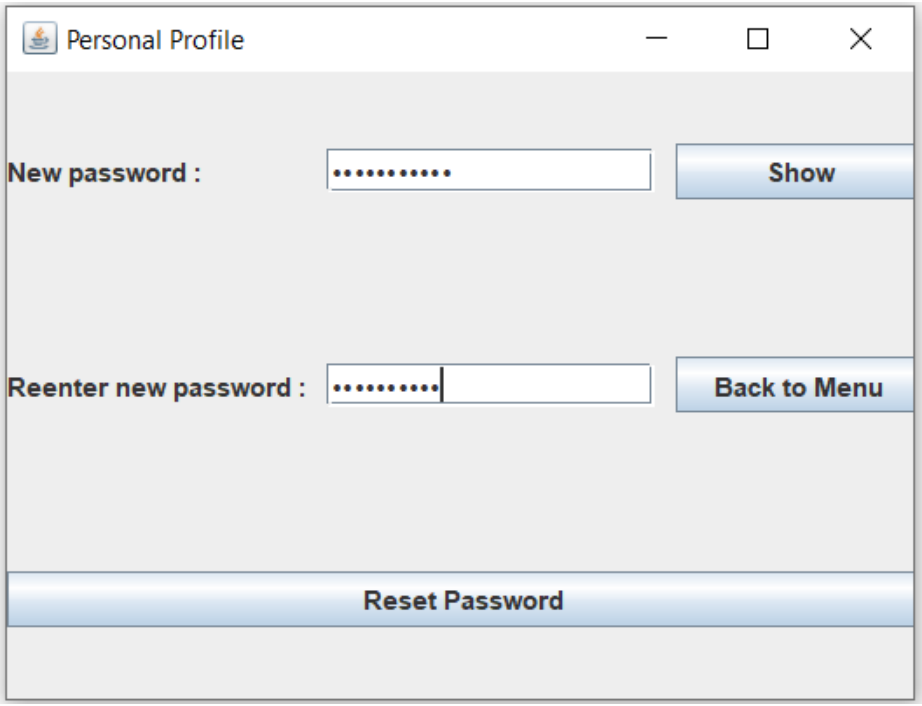



*Figure 109 : Student changing account password*

If the student chooses the “Change Password” option, they will be brought to another page where they will be prompted to enter their new password. The page asks for users to enter their new password 2 times to ensure that no typos are happening when setting their new password, as there will be no way to undo the modification process other than directly contacting the admin. But, if the users want extra assurance that the password entered is correct, they can click the “show” button to decrypt the password they have entered in the text field. After the user has entered their new password, they can click on the “Reset Password” button to officially change their account password. But, if the user suddenly decides to not change their password, they can click on the “Back to Menu” button to be brought back to the student menu page.

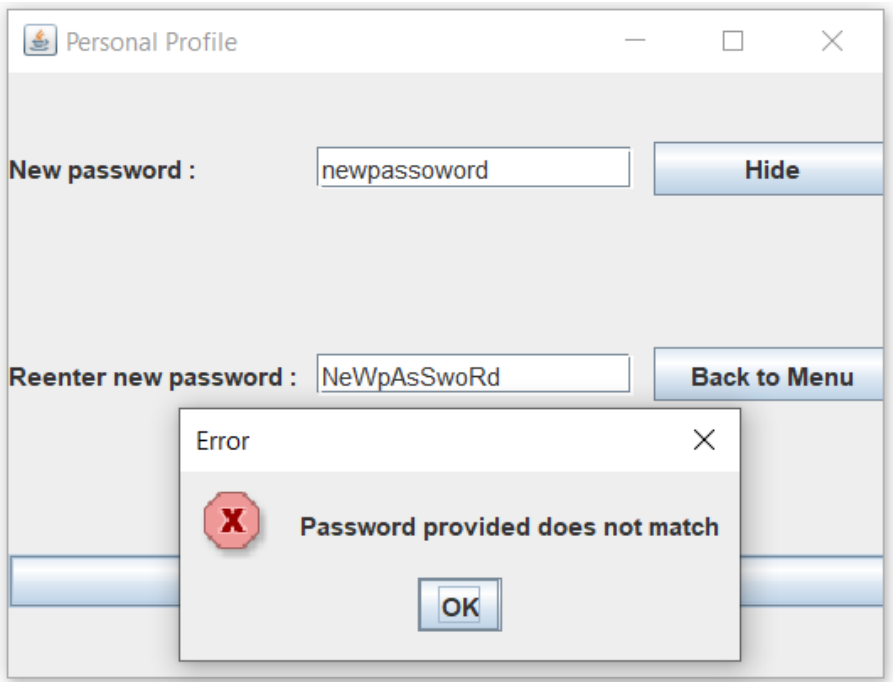



*Figure 110 : Error message for non-matching passwords entered*

In the situation that the user unintentionally enters two different passwords, the system will display an error popup indicating that the passwords do not match. And users will be prompted to reenter the password to properly change their account passwords.

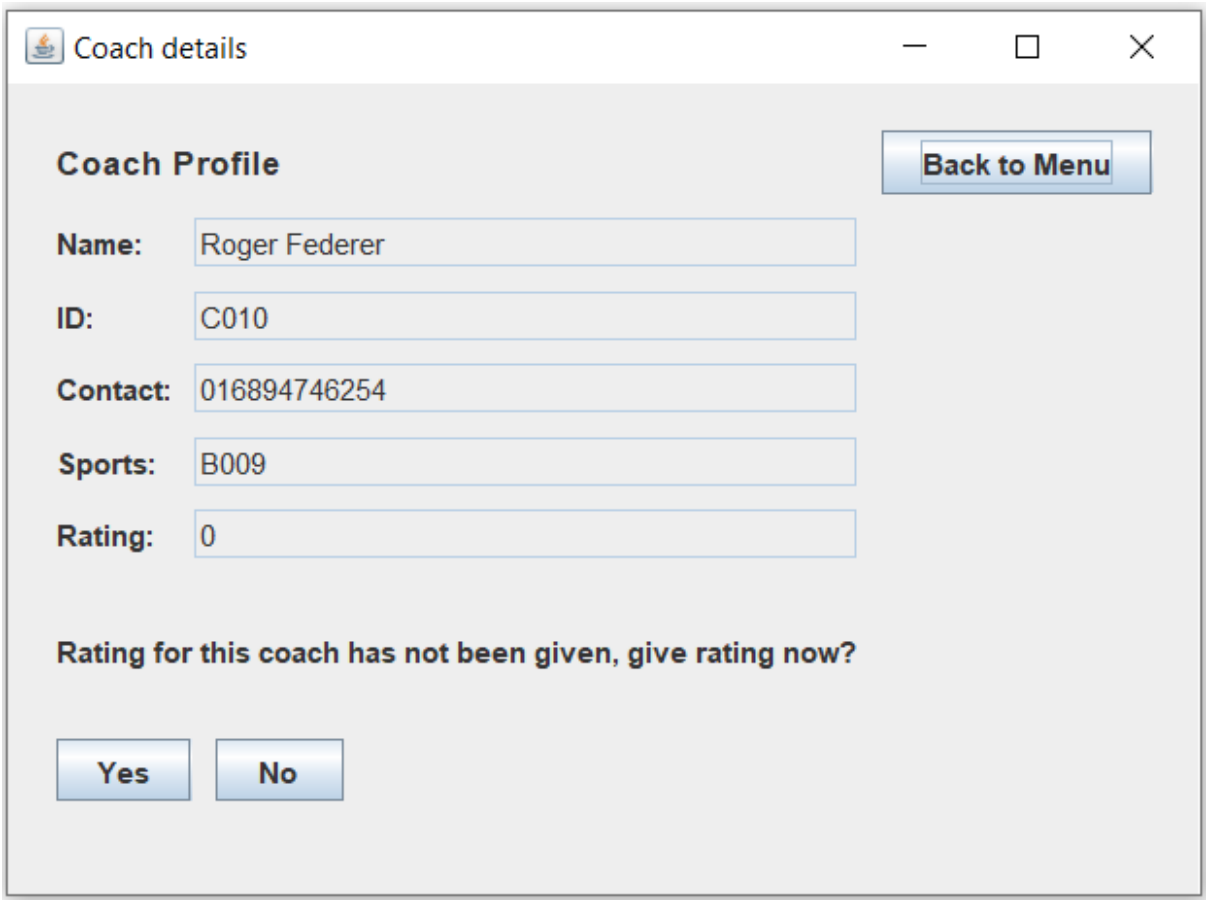



*Figure 111 : View Coach Details page*

The image above is the page users will see after selecting the "View Coach Details" option in the student menu. Details of the coach of the registered will be displayed as shown. If the user has yet to leave a feedback or rating for their coach, a notification will appear at the bottom of the screen that gives the option to give their coach a rating. If the user selects "No", then the text message will disappear.

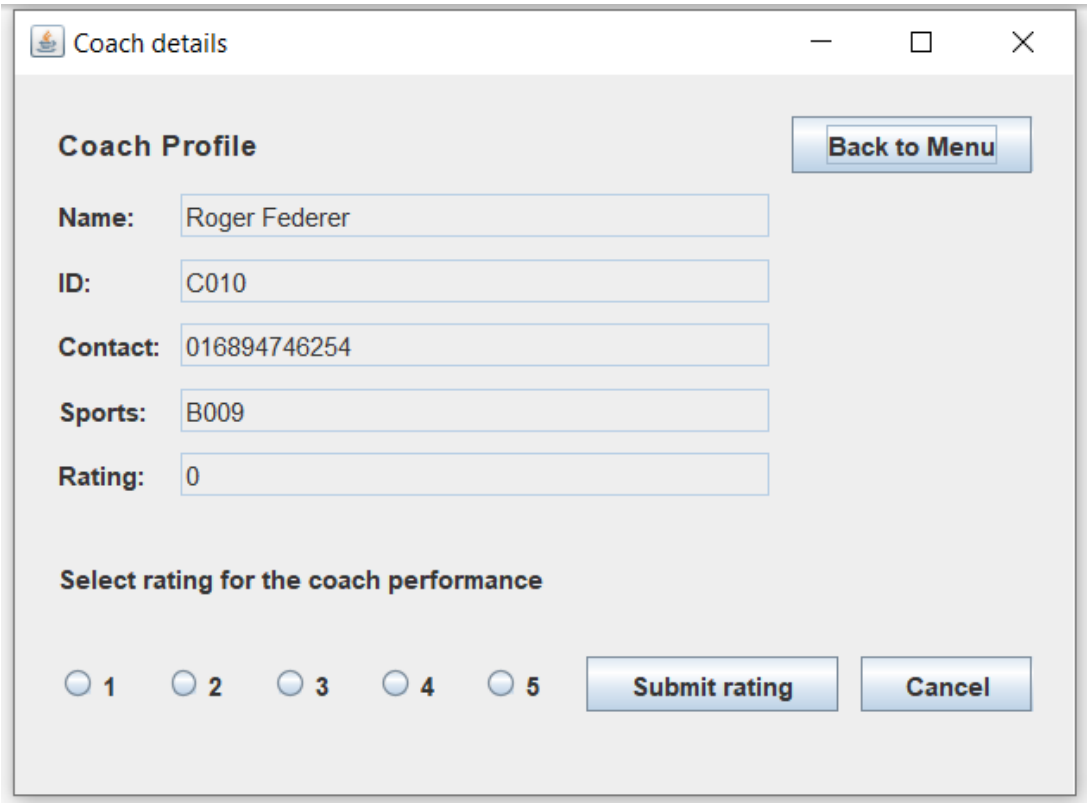



*Figure 112 : Giving rating in Coach Details page*

If the user selected "Yes" to rate the performance of their coach in the coach details page, a 1 to 5 scale will appear for the user to select to rate their coach with. But, if they decided not to rate their coach yet, they can select the 'cancel' button to rate their coach some other time.

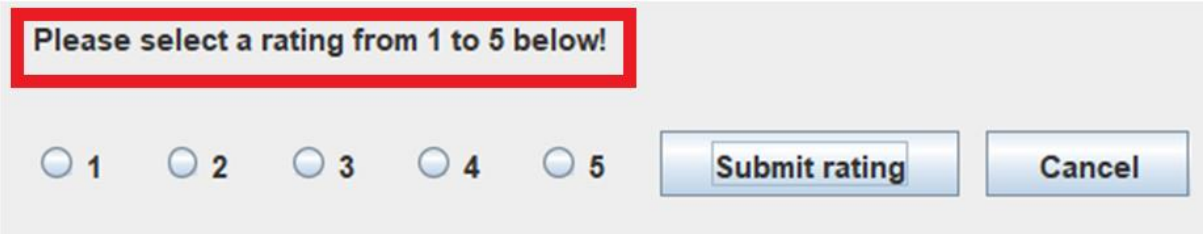



*Figure 113 : Error message if no rating score was selected*

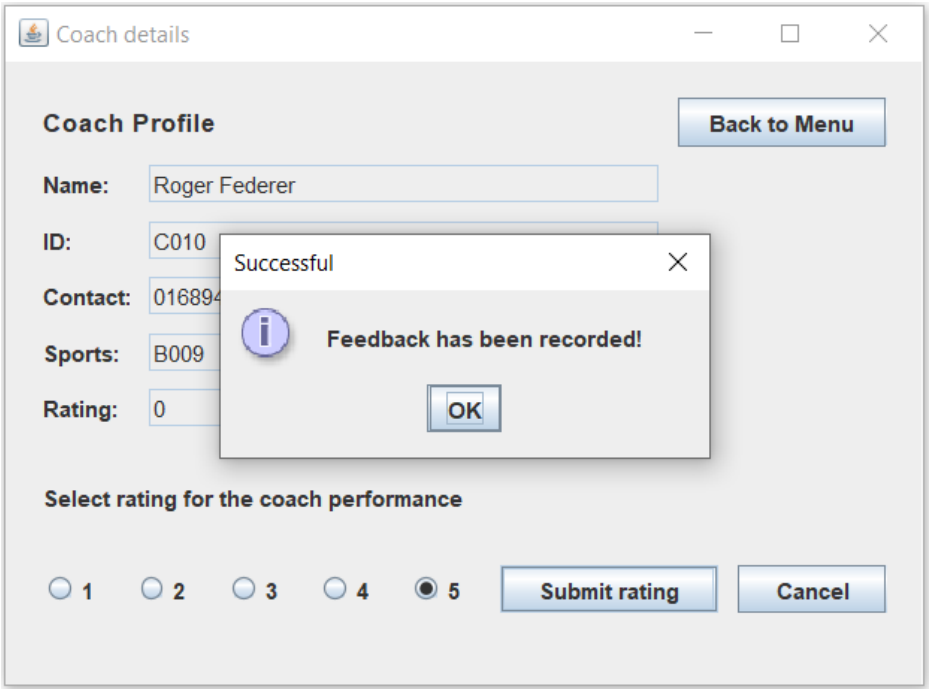



*Figure 114 : Successfully giving coach feedback*

After successfully rating the coaches, the popup will appear to notify the user that their feedback was recorded.

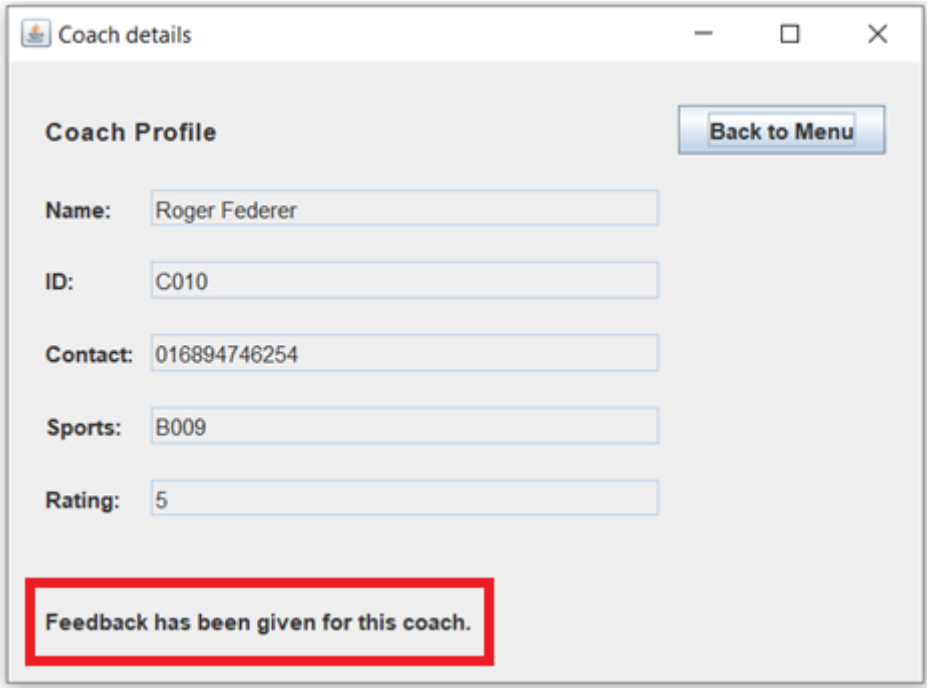



*Figure 115 : Feedback label after giving rating to coach*

After leaving a rating for the coach, revisiting the coach details page will now instead indicate that they have already rated their coaches and can no longer leave a rating.

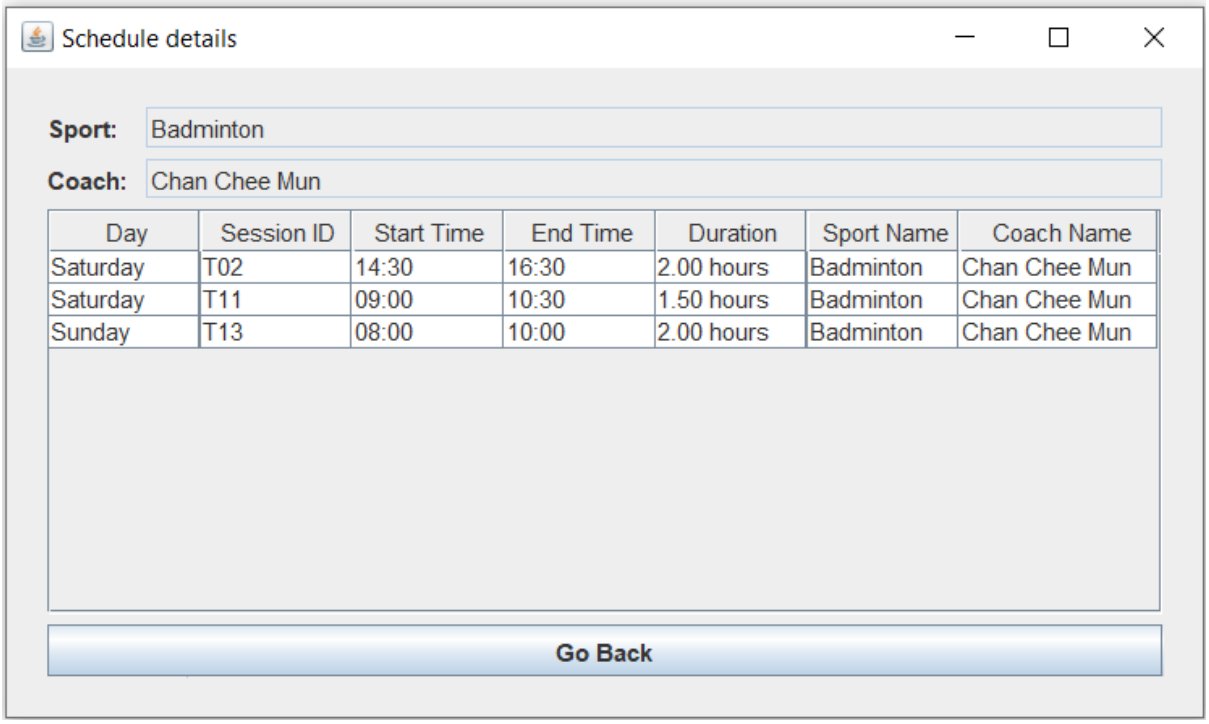



*Figure 116 : View Upcoming Sport Schedule*

The last option in the student menu, “View Upcoming Sport Schedule”, shows the user a page containing all their upcoming schedule sessions that can be attended by the student. After checking their schedule, they can click on the “Go Back” button to return to the student menu.

Unregistered Student

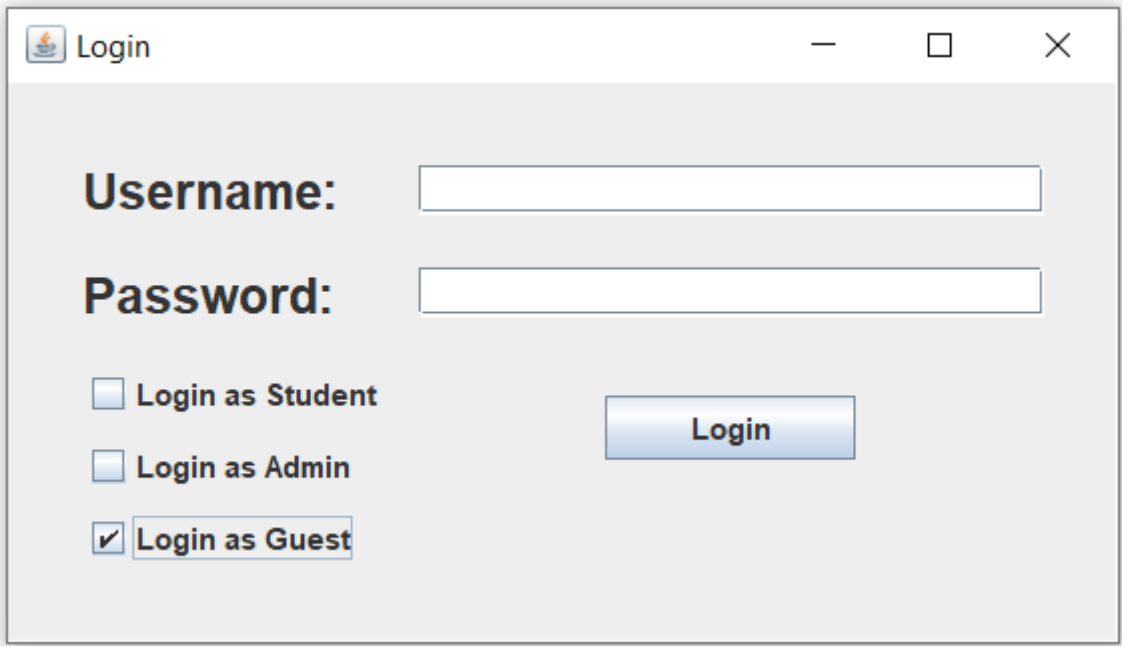



*Figure 117 : Logging in as guest*

Choosing the “Login as Guest” options allow users to login as an unregistered student and no credentials will be needed in doing so.

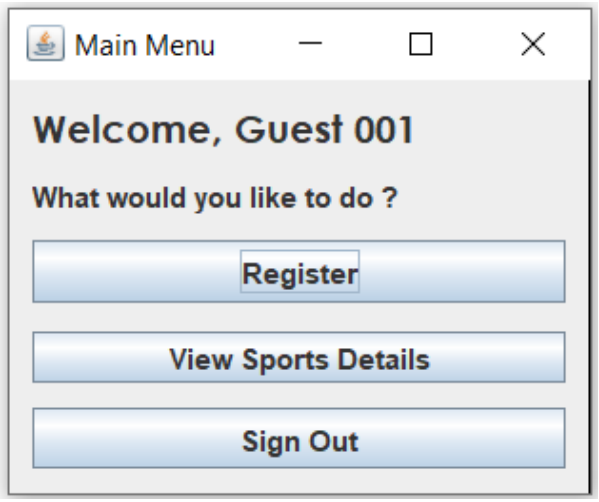



*Figure 118 : Guest main menu*

After logging in as a guest, the user can either choose to register an account, view all sports details, or sign out which returns them to the login page.

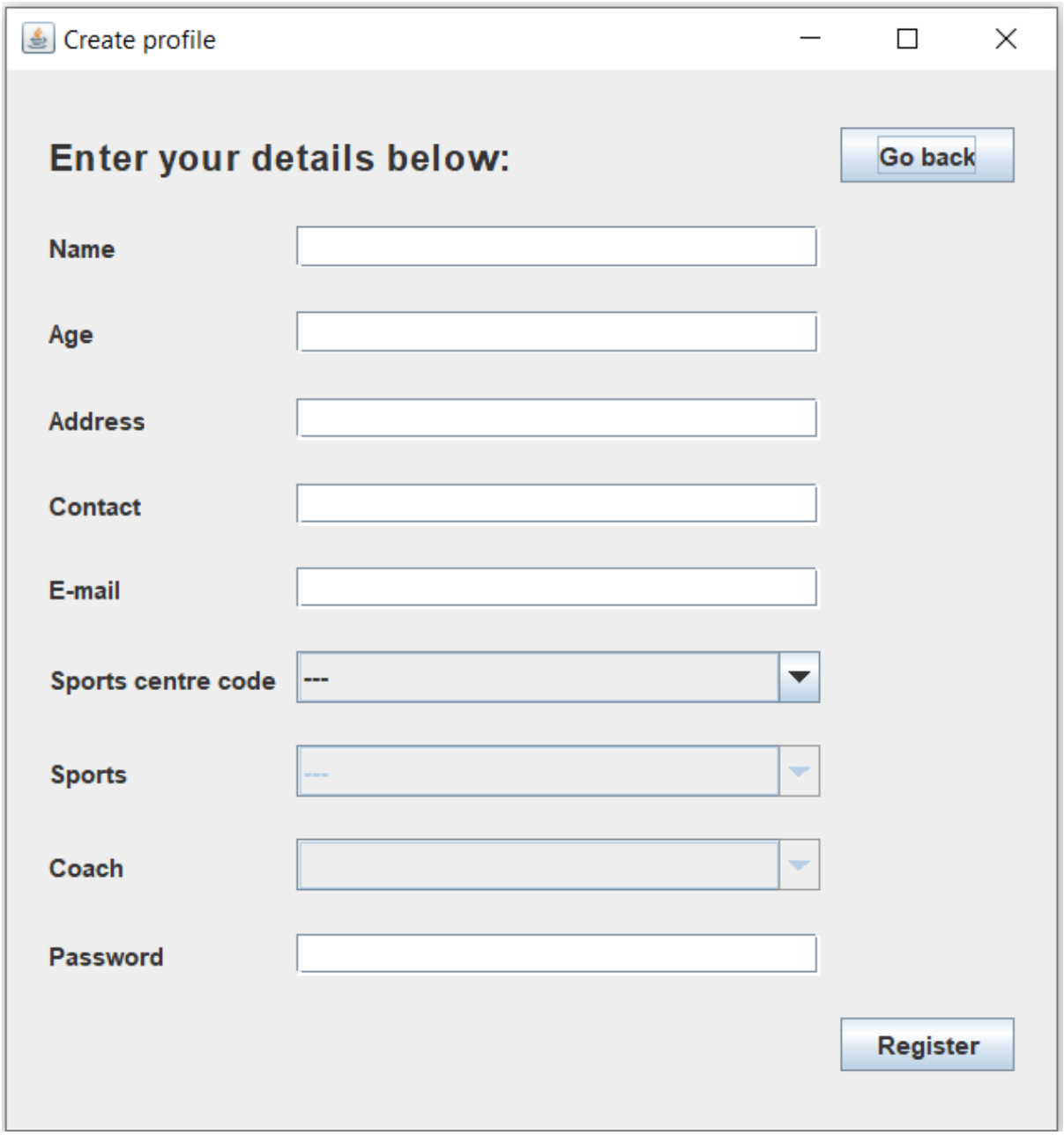



*Figure 119 : Guest account registration page*

If the guest user chooses the “Register” option, they will be brought to the page above to enter all their information to register their account. Or, they can choose the “Go back” button to return to the main menu.

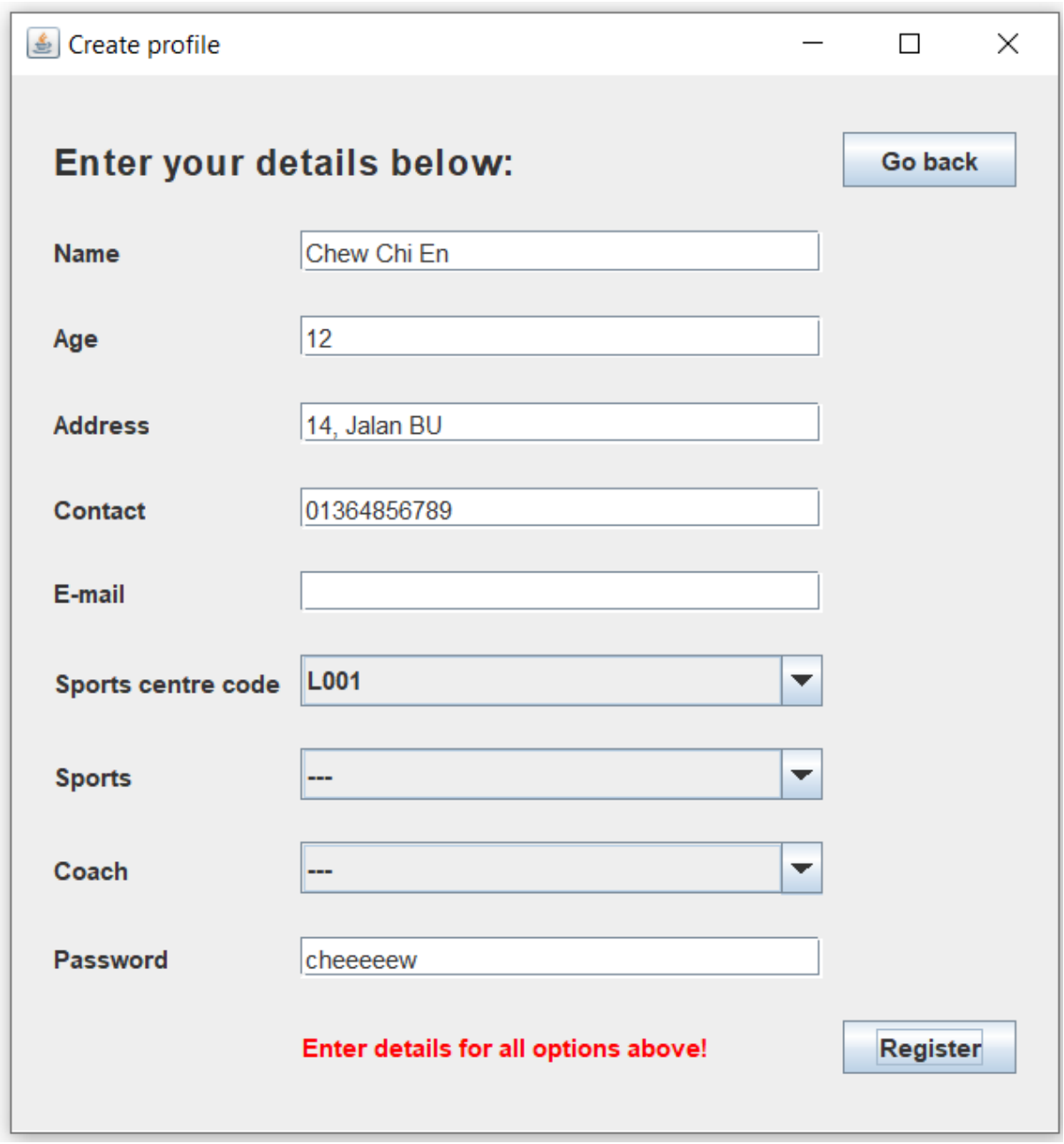



*Figure 120 : Guest registration error code*

If the user enters an input that is not in the acceptable format by the system, the program will display a red error message to prompt the user to change their details input.

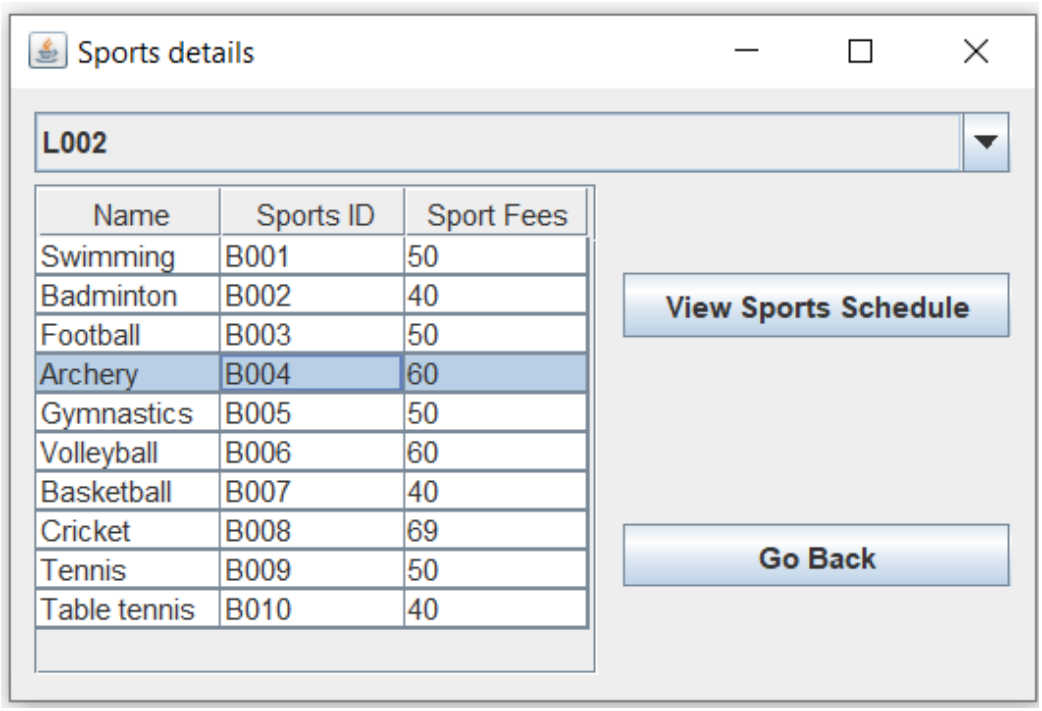



*Figure 121 : Guest View Sport Details*

Checking for the sports details as a guest allows for the user to first select a specific sports center, then only displays the corresponding sports details of the selected sports center. Similarly, like registered students, guests can also select a specific sport row and click the "View Sports Schedule" button to view all available sessions for that sport. Alternately, the user can also click the "Go Back" button to return to the main menu.

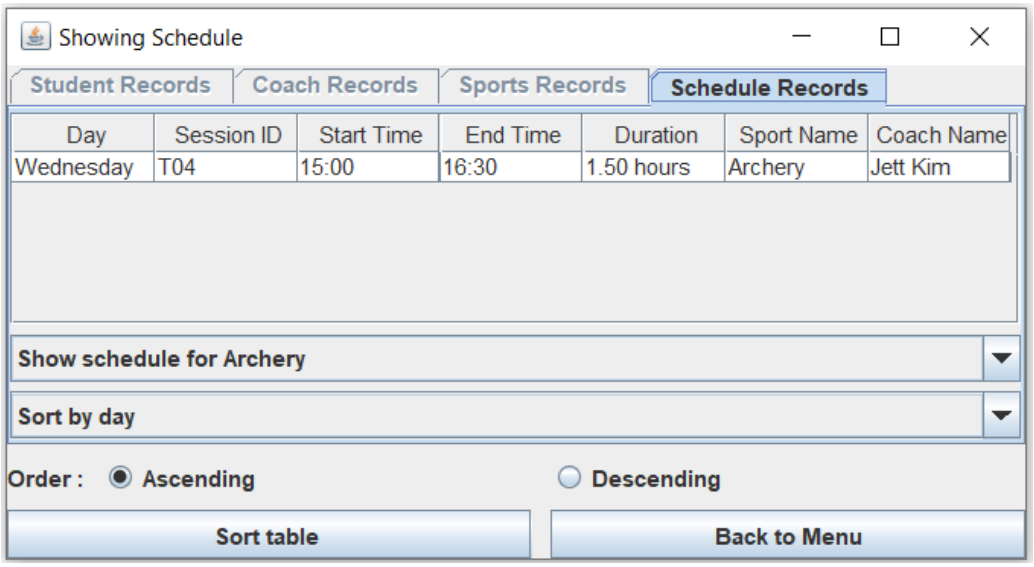



*Figure 122 : Guest viewing specific sports schedule*

## Conclusion

In a nutshell, this assignment has explored the four pillars of object-oriented programming, which are encapsulation, abstraction, inheritance, and polymorphism. In addition, all these concepts are applied to the project of building an application for the REAL CHAMPIONS SPORTS ACADEMY, and they are shown clearly in the examples above. Besides, Object Oriented Programming is an efficient programming paradigm when it comes to programming real-world objects. Therefore, this assignment really helps to introduce the concept of object-oriented programming and implement the concepts in building a real-world application.

## References


GeeksforGeeks (2018) Dynamic Method Dispatch or Runtime Polymorphism in Java. [Online]. Available at: https://www.geeksforgeeks.org/dynamic-method-dispatch-runtime-polymorphism-java/ [Accessed: 26th May 2021]

GeeksforGeeks. 2020. *HashMap in Java with Examples - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/java-util-hashmap-in-java-with-examples/ [Accessed 26 May 2021].

Janssen, T., 2017. *OOP Concept for Beginners: What is Abstraction?*. [online] Stackify. Available at: https://stackify.com/oop-concept-abstraction/ [Accessed 25 May 2021].

Janssen, T., 2017. *OOP Concept for Beginners: What is Inheritance?*. [online] Stackify. Available at: https://stackify.com/oop-concept-inheritance/ [Accessed 24 May 2021].

Programiz.com. n.d. Java Generics (With Examples). [online] Available at: https://www.programiz.com/java-programming/generics [Accessed 26 May 2021].

Singh, C. (2014) Polymorphism in Java with Example. [Online]. Available at: https://beginnersbook.com/2013/03/polymorphism-in-java/ [Accessed: 26th May 2021]

Tutorialspoint (n.d.) Java – Encapsulation. [Online]. Available at: https://www.tutorialspoint.com/java/java_encapsulation.htm [Accessed: 26th May 2021]

www.javatpoint.com. n.d. Java Anonymous Inner Class - javatpoint. [online] Available at: https://www.javatpoint.com/anonymous-inner-class [Accessed 26 May 2021].

Workload Matrix

| Group Member : | Coding | Documentation |
|---|---|---|
| Ong Cheng Kei | 55% | 50% |
| Kong Kei Zhong | 45% | 50% |