INFORMATICS INSTITUTE OF TECHNOLOGY

Software Development II

Coursework Report 2023/2024

Student Name : Palihawadana Perera
UoW ID : w2082259
IIT ID : 20232667

Table of Contents.

Task 01 - Source Code..3

Main.Java..3

Menu.Java..6

MenuOption.java...9

MenuMethods.java..11

Task 02 - Source Code..20

Student.java..20

Module.java...27

Task 03 - Source Code..28

Student.java..28

Module.java...29

Task 04 - Testing...30

Self-Evaluation Form...33

Task 01 - Source Code

Main.Java

```
import Menu.*;// import all classes from the menu package
import University.*;
import University.Module;
import java.util.Scanner;
import java.util.Stack;
import java.util.concurrent.atomic.AtomicInteger;

public class Main {

    public static final int MAX_NUMBER_OF_SEATS = 100;
    public static final int NUMBER_OF_MODULES = 3;
    // i will use an atomic integer to keep track of the student count.
    public static AtomicInteger enrolledStudentCount = new AtomicInteger(0);
    private static final String DATA_FILE_PATH = "StudentData/studentData.txt";
    public static Scanner scanner = new Scanner(System.in);
    // arrays for the students and modules
    public static Module[] allModules = new Module[NUMBER_OF_MODULES];
    public static Student[] allStudents = new Student[MAX_NUMBER_OF_SEATS];

    public static void main (String [] args) {

        try {
            // create and add the modules
            for (int i = 0; i < allModules.length; i++) {
                allModules[i] = new Module(String.format("Module 0%d", i + 1));
            }

            /*
                we will be using a simple Stack data structure to keep track of all the menus.
                this data type is a FILO type structure, so the last in item will be displayed first.
            */
            // create the menu stack
            Stack<Menu> menuStack = new Stack<>();

            // create the main menu
            Menu<Void> mainMenu = new Menu<Void>("Main Menu", scanner);

            // option 01 - check available seats
```

```
mainMenu.addOptionToMenu(MenuMethods.checkAvailableSeats(MAX_NUMBER_OF_SEATS
, enrolledStudentCount));

        // option 02 - register a student.
        mainMenu.addOptionToMenu(MenuMethods.registerStudent(scanner,
MAX_NUMBER_OF_SEATS, enrolledStudentCount, allModules, allStudents));

        // option 03 - delete a student.
        mainMenu.addOptionToMenu(MenuMethods.deleteStudent(allStudents,
enrolledStudentCount));

        // option 04 - find a student.
        mainMenu.addOptionToMenu(MenuMethods.findStudent(allStudents,
enrolledStudentCount, scanner));

        // option 05 - write data to file
        mainMenu.addOptionToMenu(MenuMethods.writeDataToFile(allModules, allStudents,
enrolledStudentCount, DATA_FILE_PATH));

        // option 06 - load data from file
        mainMenu.addOptionToMenu(MenuMethods.loadDataFromFile(allModules, allStudents,
enrolledStudentCount, DATA_FILE_PATH));

        // option 07 - display all registered students.
        mainMenu.addOptionToMenu(MenuMethods.displayRegisteredStudents(allStudents,
enrolledStudentCount));

        //create submenu
        Menu subMenu = new Menu("Sub-Menu", scanner);

        // option 08 - more options.
        mainMenu.addOptionToMenu(new MenuOption<Void>("More Options.", ()-> {
           //add the submenu to the top of the stack.
           menuStack.push(subMenu);
        }));

        // option 8.1 - edit or add student name
        subMenu.addOptionToMenu(MenuMethods.editStudentName(allStudents,
enrolledStudentCount, scanner));

        // option 8.2 - set module marks
        subMenu.addOptionToMenu(MenuMethods.setModuleMarks(allStudents,
enrolledStudentCount, scanner, allModules));
```

```
            // option 8.3 - get system summary
            subMenu.addOptionToMenu(MenuMethods.getSystemSummary(allModules,
allStudents,enrolledStudentCount));

            // option 8.4 - generate a full report.
            subMenu.addOptionToMenu(MenuMethods.displayCompleteReport(allStudents,
enrolledStudentCount));

            // option to exit submenu
            subMenu.addOptionToMenu( new MenuOption<>("Back To Main Menu.", () -> {
                // remove the submenu from the stack.
                menuStack.pop();
            }));

            // option 09 - exit program
            mainMenu.addOptionToMenu(MenuMethods.exitMenu(menuStack, allModules,
allStudents, enrolledStudentCount, DATA_FILE_PATH));

            // add the main-menu to the stack
            menuStack.push(mainMenu);

            // display the menu and run the main loop
            while (!menuStack.empty()) {
                menuStack.peek().displayMenu();
            }

            scanner.close();
        }
        // most exception handling is done by the methods themselves.
        catch (Exception e) {
            System.out.println("An Error Occurred While Running the Program:" + e);
            e.printStackTrace();
        }
    }
}
```

Menu.Java

```
package Menu;

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

@SuppressWarnings({"unused", "CallToPrintStackTrace"})

public class Menu<T> {

   /*
      this is the class for Menus. it uses an Array (list) called options to display menu options.
      this ensures this falls under the assignment brief task one.

      to use this class, create a Menu object, then once MenuObject objects are created,
      add them to the Menu object using the addOptionToMenu() method.

      to display and run the option chosen by the user, call the displayMenu() method,
      this method handles getting user input, validating it and executing the MenuOption.
   */

   //All fields will be private to follow the principle of encapsulation.
   @SuppressWarnings("FieldMayBeFinal")
   private List<MenuOption<T>> options = new ArrayList<>();
   private int numberOfOptions;
   private final String name;
   private String prompt = null;
   private Scanner scanner;

   // Constructor. Done
   public Menu (String name, Scanner scanner) {
      this.name = name;
      this.scanner = scanner;
   }

   public Menu (String name, Scanner scanner, String prompt) {
      this.name = name;
      this.scanner = scanner;
      this.prompt = prompt;
   }
```

```
    //getter methods. Done
    public String getName () {
        return this.name;
    }

    public String getOptionsList () {

        if (options.isEmpty()) {
            return "There are No Options in this Menu.";
        }

        // I'm gonna use a StringBuilder for this method cus I want to test it out.
        StringBuilder optionsListStringBuilder = new StringBuilder();
        for (MenuOption option: options) {
            optionsListStringBuilder.append(option.getOptionName()).append(", ");
        }

        // let's remove the last comma and space
        optionsListStringBuilder.setLength(optionsListStringBuilder.length() - 2);

        return optionsListStringBuilder.toString();
    }

    // Method to add MenuOption objects to the list of options. Done
    public void addOptionToMenu(MenuOption<T> option) {

        if (option != null) {
            options.add(option);
        } else {
            System.out.println("Error - MenuOption Object Not Added to Menu: Please enter a valid
menu option, MenuOption Object should not be null");
        }
    }

    // method to remove all options
    public void removeAllOptions () {
        if (!options.isEmpty()) {
            for (MenuOption option: options) {
                options.remove(option);
            }
        }
    }
    //Method to display a menu. Done
    public T displayMenu () {
        // check if there are options to display.
        if (options.isEmpty()) {
            System.out.println("There are No Options to Display in this Menu.");
            return null;
        }
        else {
            try {
                // check how many options we have; this is used to get the choice from user.
                numberOfOptions = options.size();

                System.out.println();
                // print the prompt
                if (prompt != null) {
                    System.out.println(prompt);
                } else {
                    System.out.println("Please select the Option number to select an Option");
                }

                // print all the options
                for (int i = 0; i < numberOfOptions; i++) {
                    System.out.printf("%d %s%n", i + 1, options.get(i).getOptionName());
                }
                return executeUserChoice(getUserChoice());
            }
            catch (Exception e) {
                System.out.println("The following error occurred while executing user choice: " + e);
                e.printStackTrace();
                return null; // we need to return something always.
            }
        }
    }

    //Method to get under choice as input. Done
    public int getUserChoice () {
        System.out.printf("%nPlease enter the number of the desired option: ");

        //input validation loop
        while (true) {
            if (scanner.hasNextInt()) {
                int userChoice = scanner.nextInt();
                scanner.nextLine();
                if (userChoice <= 0 || userChoice > numberOfOptions) {
```

```
                System.out.printf("Invalid choice, choice must be between 1 and %d%n",
numberOfOptions);
                System.out.print("Please enter the number of the desired option: ");
            } else {
                return userChoice;
            }
        } else {
            System.out.printf("Invalid Input, please enter a valid integer between 1 and %d : ",
numberOfOptions);
            scanner.nextLine();// breaks the infinite loop of invalid input messages.
        }
    }
  }

  // Method to run user choice. Done
  public T executeUserChoice(int userChoice) {

      System.out.printf("%nYou Chose option %d - %s%n", userChoice, options.get(userChoice
- 1).getOptionName());
      try {
          return options.get(userChoice - 1).call();
      } catch (Exception e) {
          System.out.printf("The following error occurred while executing user choice: %s%n",
e.getMessage());
          e.printStackTrace();
          return null;
      }
  }
}
```

MenuOption.java

```
package Menu;

import java.util.concurrent.Callable;

// using the generic <T> here.

public class MenuOption<T> implements Callable<T> {

  /*
      This is the class that will create MenuOption objects for the Menu class objects,
      and this class implements the runnable interface.
```

```
        When constructing a Menu.MenuOption obj, pass one runnable object as the second
argument.
        when the run() the Menu.MenuOption obj is called it will run the runnable code block.
        note this works with a lambda function as well.

        Sometimes, we will need to make an option that can return objects or stuff,
        so for that we will also implement the Callable interface.

        So depending on the use case, the user can either parse a callable or a runnable object

        These objects should be passed to Menu objects using the addMenuOption() method of
Menu class objects.
     */

    private final String optionName;
    private final Callable<T> callableAction;
    private final Runnable runnableAction;

    // Constructor for option with Callable action
    public MenuOption(String optionName, Callable<T> action) {
        this.optionName = optionName;
        this.callableAction = action;
        this.runnableAction = null;
    }

    // Constructor for menu options with Runnable action
    public MenuOption(String optionName, Runnable action) {
        this.optionName = optionName;
        this.callableAction = null;
        this.runnableAction = action;
    }

    // the method that either calls the run() method or call() method
    @Override
    public T call() throws Exception {
        if (callableAction != null) {
            return callableAction.call();
        } else if (runnableAction != null) {
            runnableAction.run();
            return null; // if not a callable still return cus that how i made the menu class work.
        } else {
            System.out.println("Error - Cannot execute MenuOption callable or runnable: No callable
or runnable action was provided.");
```

```
            return null;
        }
    }

    // now for the getter methods
    public String getOptionName() {
        return optionName;
    }
    public Callable<T> getCallableAction() {
        return callableAction;
    }
}
```

MenuMethods.java

```
package Menu;

import University.Student;
import University.Module;
import java.io.*;
import java.lang.reflect.Modifier;
import java.sql.SQLOutput;
import java.util.Scanner;
import java.util.Stack;
import static Utilities.FileUtilities.isFileEmpty;
import java.util.concurrent.atomic.AtomicInteger;

public class MenuMethods {

    private static Scanner menuMethodScanner = new Scanner(System.in);
    /*
        the methods that the main menu will use will be in this class.
        this is because i want to keep the Main.java class as small and consise as possible.
        this class will only have static methods
     */

    // method to select a student using all students.
    public static Student selectAStudent(Student[] allStudents, AtomicInteger
enrolledStudentCount) {
        try {
            // Check if there are any students registered
            if (allStudents == null || enrolledStudentCount.get() == 0) {
                throw new IllegalStateException("No students are currently enrolled.");
            }

            // Create a menu of students
            Menu<Student> selectStudent = new Menu<Student>("Select a Student Menu.",
menuMethodScanner, "To Select a Student, Please Enter the Number in front of the Student");

            // Add the students to the menu
            for (Student student : Student.getSortedStudentsByName(allStudents,
enrolledStudentCount)) {
                String studentInfoString = String.format(". Name: %s %s%n   Student ID: %s",
student.getFName(), student.getLName(), student.getStudentID());
                selectStudent.addOptionToMenu(new MenuOption<>(studentInfoString, () ->
student));
            }

            // Return the selected student
            return selectStudent.displayMenu();

        }
        catch (IllegalStateException e) {
            System.err.println("Error: " + e.getMessage());
        }
        catch (Exception e) {
            System.err.println("An unexpected error occurred: " + e.getMessage());
        }

        return null;
    }

    // method to find or select a student
    public static Student selectOrFindAStudent (Student[] allStudents, AtomicInteger
enrolledStudentCount) {
        Menu<Student> studentSelectorMenu = new Menu<Student>("Student Selector menu",
menuMethodScanner, "Please select how you would like to find a student: ");
        studentSelectorMenu.addOptionToMenu(new MenuOption<>("Select a Student Using All
Students List", () -> selectAStudent(allStudents, enrolledStudentCount)));
        studentSelectorMenu.addOptionToMenu(new MenuOption<Student>("Select a Student
Using Student ID.", () -> Student.findStudent(allStudents, enrolledStudentCount,
menuMethodScanner)));
        return studentSelectorMenu.displayMenu();
    }
```

```
   // 1. Method for checking available seats.
   public static MenuOption<Void> checkAvailableSeats(Integer MAX_NUMBER_OF_SEATS,
                                               AtomicInteger enrolledStudentCount) {

       return new MenuOption<Void>("Check Available Seats",
               () -> System.out.printf("There are Currently %d Available%n",
MAX_NUMBER_OF_SEATS - enrolledStudentCount.get()));
   }

   // 2. Method to register a student
   public static MenuOption<Void> registerStudent (Scanner scanner,
                                              Integer MAX_NUMBER_OF_SEATS,
                                              AtomicInteger enrolledStudentCount,
                                              Module[] allModules,
                                              Student[] allStudents) {
       return new MenuOption<Void>("Rejister a New Student.", () -> {
           try {
               // Ensure there are available seats before creating a new student
               if (enrolledStudentCount.get() >= MAX_NUMBER_OF_SEATS) {
                   throw new IllegalStateException("Cannot register Student: There are no open seats
available for this semester.");
               }

               System.out.println("Enter the First Name of The Student: ");
               String fName = scanner.nextLine().trim();
               if (fName.isEmpty()) {
                   throw new IllegalArgumentException("First name cannot be empty.");
               }

               System.out.println("Enter the Last Name of The Student: ");
               String lName = scanner.nextLine().trim();
               if (lName.isEmpty()) {
                   throw new IllegalArgumentException("Last name cannot be empty.");
               }

               // Construct the student
               new Student(fName, lName, allModules, MAX_NUMBER_OF_SEATS,
enrolledStudentCount, allStudents);
           }
           catch (IllegalArgumentException e) {
               System.err.println("Input Error: " + e.getMessage());
           }
           catch (IllegalStateException e) {
               System.err.println("Registration Error: " + e.getMessage());
```

```
        }
        catch (Exception e) {
            System.err.println("An unexpected error occurred: " + e.getMessage());
            e.printStackTrace();
        }
    });
}

// 3. Method to delete a student
public static MenuOption<Void> deleteStudent (Student[] allStudents,
                                              AtomicInteger enrolledStudentCount) {
    return new MenuOption<Void>("Delete a Registered Student.", () -> {
        if (enrolledStudentCount.get() > 0) {
            Student student = selectOrFindAStudent(allStudents, enrolledStudentCount);
            if (student == null) {
                return;
            }
            student.deleteStudent(allStudents, enrolledStudentCount);
        }
        else {
            System.out.println("Error - Cannot delete Student: There are no Students to Delete.");
        }
    });
}

// 4. Method to find a student: using ID
public static MenuOption<Void> findStudent (Student[] allStudents, AtomicInteger
enrolledStudentCount, Scanner scanner) {
    return new MenuOption<Void>("Find a Registered Student.", () -> {
        Student selectedStudent = Student.findStudent(allStudents, enrolledStudentCount,
scanner);
        if (selectedStudent != null) {
            selectedStudent.displayStudentSummary();
        }
    });
}


// 5. Method to write data to txt file
public static MenuOption<Void> writeDataToFile (Module[] allModules,
                                                Student[] allStudents,
                                                AtomicInteger enrolledStudentCount,
                                                String DATA_FILE_PATH) {
    return new MenuOption<Void>("Save Student and Module Data.", () -> {
        try {
            // the following code is directly referenced from SimpliLearn.com, available at:
https://www.simplilearn.com/tutorials/java-tutorial/serialization-in-java#:~:text=Serialization%20in
%20Java%20is%20the,then%20de%2Dserialize%20it%20there.

            FileOutputStream file = new FileOutputStream(DATA_FILE_PATH);
            ObjectOutputStream oos = new ObjectOutputStream(file);
            // save the modules
            oos.writeObject(allModules);
            // save the students
            oos.writeObject(allStudents);
            // save the student count
            oos.writeObject(enrolledStudentCount);
            System.out.println("Program data saved successfully.");
        }
        catch (IOException e) {
            System.out.println("Error saving program data: " + e.getMessage());
        }
        catch (Exception e) {
            System.out.println("An unexpected error occurred: " + e.getMessage());
        }
    });
}

// 6. Method to load data from file
public static MenuOption<Void> loadDataFromFile (Module[] allModules,
                                                  Student[] allStudents,
                                                  AtomicInteger enrolledStudentCount,
                                                  String DATA_FILE_PATH) {
    return new MenuOption<Void>("Load Student and Module Data from File.", () -> {

        // the following code is directly referenced from SimpliLearn.com, available at:
https://www.simplilearn.com/tutorials/java-tutorial/serialization-in-java#:~:text=Serialization%20in
%20Java%20is%20the,then%20de%2Dserialize%20it%20there.

        File file = new File(DATA_FILE_PATH);

        // check if the file is empty.
        if (isFileEmpty(file)) {
            System.out.println("Error - Cannot Load Data: File is Empty.");
            return;
        }

        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file))) {
            Module[] modulesTemp = (Module[]) ois.readObject();
            AtomicInteger tempInt = (AtomicInteger) ois.readObject();
            Student[] studentsTemp = (Student[]) ois.readObject();

            // Copy loaded data to the provided arrays
            System.arraycopy(modulesTemp, 0, allModules, 0, modulesTemp.length);
            System.arraycopy(studentsTemp, 0, allStudents, 0, studentsTemp.length);
            // restore the student count.
            enrolledStudentCount.set(tempInt.get());

            System.out.println("Program data loaded successfully.");
          }
          catch (FileNotFoundException e) {
            System.out.println("Error - Cannot Load Data: The file was not found. Please make
sure the file path is correct");
          }
          catch (IOException e) {
            System.out.println("Error loading program data: " + e.getMessage());
          }
          catch (ClassNotFoundException e) {
            System.out.println("Error - Cannot Load Data: Class not found while loading program
data");
          }
          catch (ClassCastException e) {
            System.out.println("Error - Cannot Load Data: Data in the file is corrupted or does not
match expected types");
          }
        });
    }

    // 7. Method to view the list of students based on their names
    public static MenuOption<Void> displayRegisteredStudents (Student[] allStudents,
                                              AtomicInteger enrolledStudentCount) {
      return new MenuOption<Void>("Display All Registered Students", new Runnable() {
        @Override
        public void run() {
          Student.displaySortedStudent(allStudents, enrolledStudentCount);
        }
      });
    }

    // a. Method to add or edit student name
    public static MenuOption<Void> editStudentName (Student[] allStudents, AtomicInteger
enrolledStudentCount, Scanner scanner) {
        return new MenuOption<Void>("Edit or Add a Student's Name.", () -> {

            if (allStudents == null || enrolledStudentCount.get() == 0) {
            System.out.println("Error - Cannot choose this option: There are no Students
Registered to edit.");
            return;
            }

            try {
            // Inform the user to select a student
            System.out.println("To edit or add a name, select a student");

            // Select or find a student
            Student selectedStudent = selectOrFindAStudent(allStudents, enrolledStudentCount);
            if (selectedStudent == null) {
                System.out.println("Error: No student was selected or found.");
                return;
            }

            // Get the first name
            System.out.println("Enter the First Name of the Student: ");
            String fName = scanner.nextLine().trim();
            if (fName.isEmpty()) {
                throw new IllegalArgumentException("First name cannot be empty.");
            }
            selectedStudent.setFName(fName);

            // Get the last name
            System.out.println("Enter the Last Name of the Student: ");
            String lName = scanner.nextLine().trim();
            if (lName.isEmpty()) {
                throw new IllegalArgumentException("Last name cannot be empty.");
            }
            selectedStudent.setLName(lName);

            // Display updated student information
            System.out.println("Updated Student Information");
            selectedStudent.displayStudentInfo();

            }
            catch (IllegalArgumentException e) {
            System.err.println("Error: " + e.getMessage());
            }
            catch (Exception e) {
                System.err.println("An unexpected error occurred: " + e.getMessage());
            }
        });
    }

    // b. Method to set module marks for all modules.
    public static MenuOption<Void> setModuleMarks (Student[] allStudents,
                                                   AtomicInteger enrolledStudentCount,
                                                   Scanner scanner,
                                                   Module[] allModules) {
        return new MenuOption<Void>("Set Module Marks for Modules 1 - 3.", () -> {

            // check if there are any students
            if (allStudents == null || enrolledStudentCount.get() == 0) {
                System.out.println("Error - Cannot choose this option: There are no Students
Registered to edit.");
                return;
            }

            // select the student.
            System.out.println("Select a Student to Set Their Module Marks.");
            Student selectedStudent = selectOrFindAStudent(allStudents, enrolledStudentCount);
            if (selectedStudent == null) {
                return;
            }

            // get the module
            Module selectedModule = selectAModule(allModules, scanner);

            // set the marks
            System.out.printf("Enter the marks for module '%s': %n", selectedModule.name());
            Float marks = scanner.nextFloat();
            scanner.nextLine();
            // exception handling is done by the setModuleMarks method itself.
            selectedStudent.setModuleMarks(selectedModule, marks);
        });
    }
    // method to select a module.
    public static Module selectAModule (Module[] allModules, Scanner scanner) {
        Menu<Module> moduleSelectorMenu = new Menu<Module>("Module Selector Menu",
scanner, "Please Select a Module to Set Marks For");
        // add the modules as options.
        for (Module module: allModules) {
            moduleSelectorMenu.addOptionToMenu(new MenuOption<Module>(module.name(), ()
-> module));
        }
        return moduleSelectorMenu.displayMenu();
    }

    // c. Generate Summary of Student and module data
    public static MenuOption<Void> getSystemSummary (Module[] allModules, Student[]
allStudents, AtomicInteger enrolledStudentCount) {
        return new MenuOption<Void>("Generate Summary of Students and Modules.", ()-> {

            if (allStudents == null || enrolledStudentCount.get() == 0) {
                System.out.println("There are 0 Students Registered");
                System.out.println("To get module data Register at least one student");
                return;
            }

            int totalStudents = enrolledStudentCount.get();
            System.out.printf("Total Number of Students Registered: %d%n", totalStudents);
            System.out.println("===== Module Summary =====");
            for (Module module: allModules) {
                System.out.printf("Module Name: %s%n Number of Students Who Passes: %d%n",
module.name(), module.getStudentPassedCount(allStudents));
            }
        });
    }

    // d. Generate a complete Student report with list of students
    public static MenuOption<Void> displayCompleteReport (Student[] allStudents, AtomicInteger
enrolledStudentCount) {
        return new MenuOption<Void>("Display Complete Report of All Students", () -> {
            Student.displayCompleteStudentReport(allStudents, enrolledStudentCount);
        });
    }

    // 9. Method to safely exist program
    public static MenuOption<Void> exitMenu (Stack<Menu> menuStack,
                                    Module[] allModules,
                                    Student[] allStudents,
                                    AtomicInteger enrolledStudentCount,
                                    String DATA_FILE_PATH) {
        return new MenuOption<Void>("Save and Exit Program.", new Runnable() {
            @Override
            public void run() {
                MenuOption<Void> temp = writeDataToFile(allModules, allStudents,
enrolledStudentCount, DATA_FILE_PATH);
                try {
                    temp.call();
                } catch (Exception e) {
                    throw new RuntimeException(e);
                }
                //close the local scanner
                menuMethodScanner.close();
                // remove the menu from the stack
                menuStack.pop();
            }
        });
    }
}
```

Task 02 - Source Code.

Student.java

```
package University;

import Utilities.UserIDGenerator;

import java.io.Serial;
import java.io.Serializable; // this will help me serialize the objects to a txt file using byte code
import java.util.*;
import java.util.concurrent.atomic.AtomicInteger;

public class Student implements Serializable {

    @Serial
    private static final long serialVersionUID = 1L;

    private String fName;
    private String lName;
    private final String studentID;
    // and i will use a map, which just like a dictionary in python, to manage the marks
    private Map<Module, Float> moduleMarks = new HashMap<>();
    private Map<Module, String> moduleGrades = new HashMap<>();
    // and for the grades i will use an enum
    public enum Grades {
        Distinction, Merit, Pass, Fail
    }

    // i will use atomic integers to keep track of the student count
    public Student (String fName,
                    String lName,
                    Module[] allModules,
                    Integer MaxNumOfStudents,
                    AtomicInteger enrolledStudentCount,
                    Student[] enrolledStudents) {

        this.fName = fName;
        this.lName = lName;
        this.studentID = generateStudentID();

        // add to the list of students' array
        if (enrolledStudentCount.get() < MaxNumOfStudents) {
            enrolledStudents[enrolledStudentCount.get()] = this;
            enrolledStudentCount.incrementAndGet(); // this should increment the integer in the
main.java class
            // add to the modules.
            this.addToModules(allModules);
            this.displayStudentInfo();
            System.out.println("Student was Successfully Enrolled");
        } else {
            System.out.println("Error - Cannot register Student: There are no open seats available
for this semester");
        }
    }

    // setter methods
    public void setFName(String fName) {
        this.fName = fName;
    }

    public void setLName(String lName) {
        this.lName = lName;
    }

    // getter methods
    // get student's first name => string
    public String getFName() {
        return fName;
    }

    // get student last name => string
    public String getLName() {
        return lName;
    }

    // get full name
    public String getFullName() {
        return fName + " " + lName;
    }

    // get student id => string
    public String getStudentID() {
        return studentID;
    }

    // get the map of marks as a map. returns a clone
    public Map<Module, Float> getModuleMarksMap() {
        return new HashMap<>(moduleMarks); // returns a clone of the map to protect
encapsulation.
    }

    // get the map of grades as a map, returns a clone
    public Map<Module, String> getModuleGradesMap() {
        return new HashMap<>(moduleGrades);
    }

    // get module marks
    public float getModuleMark (Module module) {
        return moduleMarks.get(module);
    }

    // get module grade
    public String getModuleGrade (Module module) {
        return moduleGrades.get(module);
    }

    // method to generate a student ID
    private String generateStudentID () {
        // creates a pseudo random and unique ID.
        UserIDGenerator idGenerator = new UserIDGenerator();
        return idGenerator.generateUniqueID(8);
    }

    // enroll the student to all modules
    public void addToModules (Module[] allModules) {
        for (Module module: allModules) {
            // add the modules as keys to the maps.
            moduleMarks.put(module, 0.0f);
            moduleGrades.put(module, null);
        }
    }

    // calculate grade
    private String calculateGrade (float marks) {
        if (marks >= 80) {
            return Grades.Distinction.toString();
        } else if (marks >= 70) {
            return Grades.Merit.toString();
        } else if (marks >= 40) {
            return Grades.Pass.toString();
        } else {
            return Grades.Fail.toString();
        }
    }

    // Method to set Student marks and grade
    public void setModuleMarks (Module module, float marks) {
        if (marks > 0 && marks <= 100 && moduleMarks.containsKey(module)) {
            // add the marks
            moduleMarks.put(module, marks);
            // add the grades
            moduleGrades.put(module, calculateGrade(marks));
            System.out.printf("Student grade: %s%n", calculateGrade(marks));
            System.out.println("Student Marks were Successfully recorded.");
        }
        else {
            System.out.println("Error - Cannot enter marks: Marks Must be between 0 - 100.");
        }
    }

    // method to calculate total marks
    public float getTotalMarks () {
        Float totalMarks = 0.0f;
        for (Float marks: moduleMarks.values()) {
            totalMarks += marks;
        }
        return totalMarks;
    }

    // get the average marks
    public float getAverageMarks () {
        // lets avoid the zero division exception.
        return !moduleMarks.isEmpty() ? getTotalMarks() / moduleMarks.size(): 0.0f;
    }

    // method to delete students
    public void deleteStudent(Student[] allStudents,
                              AtomicInteger enrolledStudentCount) {

        // Remove from allStudents array
        for (int i = 0; i < enrolledStudentCount.get(); i++) {
            if (allStudents[i].equals(this)) {
                // i'll shift remaining elements
                for (int j = i; j < enrolledStudentCount.get() - 1; j++) {
                    allStudents[j] = allStudents[j + 1];
                }
                // remove the copy of the last student from the list.
                allStudents[enrolledStudentCount.get() - 1] = null;
                enrolledStudentCount.decrementAndGet();
                break;
            }
        }
        System.out.printf("Student with ID '%s' Was Successfully deleted%n", this.getStudentID());
    }

    // static method to find a student using ID
    public static Student findStudent(Student[] allStudents,
                                      AtomicInteger enrolledStudentCount,
                                      Scanner scanner) {

        try {
            // Check if there are any students registered
            if (allStudents == null || enrolledStudentCount.get() == 0) {
                System.out.println("Error - Cannot Find Student: The student list is empty or null.");
                return null;
            }
```

```
            // Get the ID
            System.out.print("Enter the Student ID to find the Student: ");
            String enteredID = scanner.nextLine().trim();

            for (Student student : allStudents) {
                if (student != null && student.getStudentID().equalsIgnoreCase(enteredID)) {
                    System.out.println("Selected Student Details");
                    student.displayStudentInfo();
                    return student;
                }
            }

            // The student was not found, so return null
            System.out.println("Error - Unable to find Student: There was No student Matching the
Provided Student ID in the Database.");
            return null;
        }
        catch (Exception e) {
            System.err.println("An unexpected error occurred: " + e.getMessage());
            e.printStackTrace();
            return null;
        }
    }

    // sorting methods - uses bubble sort

    // method to sort the students based on names
    public static List<Student> getSortedStudentsByName(Student[] allStudents,
                                                        AtomicInteger enrolledStudentCount) {
        /*
            using a bubble sort algorithm is highly inefficient because of its time complexity.
            no real-world application would use this sorting algorithm
         */

        // check if there are any students registered
        if (allStudents == null || enrolledStudentCount.get() == 0) {
            System.out.println("Error: The student list is empty or null.");
            return Collections.emptyList(); // ill return an empty list if the list is null
        }

        // Create a local copy and filter out nulls
        List<Student> students = new ArrayList<>();
        for (Student student : allStudents) {
            if (student != null) {
```

```
            students.add(student);
        }
    }

    // Bubble sort based on names
    int size = students.size();
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            String student01Name = students.get(j).getFullName();
            String student02Name = students.get(j + 1).getFullName();

            // Compare and swap if necessary
            if (student01Name.compareTo(student02Name) > 0) {
                Student temp = students.get(j);
                students.set(j, students.get(j + 1));
                students.set(j + 1, temp);
            }
        }
    }

    return students;
}

// method to sort students based on their average marks highest to lowest
public static List<Student> getSortedStudentsByAverage(Student[] allStudents,
                                                   AtomicInteger enrolledStudentCount) {
    // pretty much the same as the above sorting method

    if (allStudents == null || enrolledStudentCount.get() == 0) {
        System.out.println("Error: The student list is empty or null.");
        return null;
    }
    // Create a local copy and filter out nulls
    List<Student> students = new ArrayList<>();
    for (Student student : allStudents) {
        if (student != null) {
            students.add(student);
        }
    }

    // Bubble sort based on average marks
    int size = students.size();
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
                if (students.get(j).getAverageMarks() < students.get(j + 1).getAverageMarks()) {
                    // Swap the two
                    Student temp = students.get(j);
                    students.set(j, students.get(j + 1));
                    students.set(j + 1, temp);
                }
            }
        }
        return students;
    }
}
```

Module.java

```
package University;

import java.io.Serial;
import java.io.Serializable;

public record Module(String name) implements Serializable {

    @Serial
    private static final long serialVersionUID = 2L;

    /*
        the Module class for this assignment will be a record class, because it does not have any
robust methods.
        the rest of the functionalities logically come under other classes and packages like the
Menu or Student classes, etc.
     */
}
```

Task 03 - Source Code.

Student.java

```
// display methods

    // method to display student data.
    public void displayStudentInfo () {
        System.out.println();
        System.out.println("===== Student Information =====");
        System.out.printf("Student Name: %s %s%n", this.fName, this.lName);
        System.out.printf("Student ID: %s%n", this.getStudentID());
    }

    // display student info with marks and grade.
    public void displayStudentSummary () {
        this.displayStudentInfo();
        System.out.println("----- Module Results -----");

        // let's iterate over the entries in the module marks map
        // i will get each entry from the map as an array and then iterate over it.
        for (Map.Entry<Module, Float> moduleEntry: moduleMarks.entrySet()) {
            System.out.printf("Module: %s%n", moduleEntry.getKey().name());
            System.out.printf("Marks: %.2f%n", moduleEntry.getValue());
            System.out.printf("Grade: %s%n", calculateGrade(moduleEntry.getValue()));
            System.out.println();
        }

        //print the summary
        System.out.println("----- Summary -----");
        System.out.printf("Total Marks: %.2f%n", this.getTotalMarks());
        System.out.printf("Average Marks: %.2f%n", this.getAverageMarks());
        System.out.printf("Overall Grade: %s%n", calculateGrade(this.getAverageMarks()));
        System.out.println("---------------- **** ----------------");
        System.out.println();
    }

    // method to display the sorted students by name
    public static void displaySortedStudent(Student[] allStudents,
                                        AtomicInteger enrolledStudentCount) {
        if (allStudents == null || enrolledStudentCount.get() == 0) {
            System.out.println("Error - Cannot Display Students: The student list is empty or null.");
            return;
        }

        List<Student> sortedStudents = getSortedStudentsByName(allStudents,
enrolledStudentCount);
        for (Student student: sortedStudents) {
            student.displayStudentInfo();
        }
    }

    // display the full report, sorted by average highest to lowest.
    public static void displayCompleteStudentReport(Student[] allStudents,
                                            AtomicInteger enrolledStudentCount) {
        if (allStudents == null || enrolledStudentCount.get() == 0) {
            System.out.println("No students to display.");
            return;
        }
        // sort the students by average
        List<Student> sortedStudents = getSortedStudentsByAverage(allStudents,
enrolledStudentCount);
        System.out.println("====== Complete Report of All Students ======");
        System.out.println();

        // print their details.
        for (Student student : sortedStudents) {
            student.displayStudentSummary();
        }
    }
```

Module.java

```
// method for getting the total number of students who passed the module.
    public int getStudentPassedCount(Student[] allStudents) {
        int count = 0;
        for (Student student : allStudents) {
            // check to see if the student is null or not.
            if (student != null && student.getModuleMark(this) >= 40) {
                count++;
            }
        }
        return count;
    }
```

Task 04 - Testing

Total Tests Passed: 26/27

Test Case	Expected Result	Actual Result	Pass/Fail
Check available seats before registering any students.	After pressing 1, it should display that there are 100 seats available.	After pressing 1, it displayed that there are 100 seats available.	Pass
Check available seats after registering 50 students	After pressing 1, it should display that there are 50 seats available.	After pressing 1, it displayed that there are 50 seats available.	Pass
Check available seats after registering 99 students	After pressing 1, it should display that there is 1 seat available.	After pressing 1, it displayed that there is 1 seat available.	Pass
Check available seats after registering 100 students	After pressing 1, it should display that there are 0 seats available.	After pressing 1, it displayed that there are 0 seats available.	Pass
Try to register a student when no other student have registered	After pressing 2, it should properly register the student	Same as expected result	Pass
Try to register a student where 99 students have been registered	After pressing 2, it should properly register the student	Same as expected result	Pass
Try to register a student when 100 students have already been registered	After pressing 2, it should notify the user that there are no free seats available and should not register a student	Same as expected result	Pass
Try registering a student with an invalid first name	After pressing 2, and leaving the first name field empty, it should warn the user, and return to main menu	Same as expected result	Pass
Try registering a student with an invalid last name	After pressing 2, and leaving the last name field empty, it should warn the user and return to main menu	Same as expected result	Pass

Delete a Student: Initial state: At least one student registered	After finding the student, delete the student then print "Student with ID '[ID]' Was Successfully deleted"	Same as expected result	Pass
Delete a student when no students registered	Print "Error - Cannot delete Student: There are no Students to Delete."	Same as expected result	Pass
Try to delete a non-existent student	Print "Error - Unable to find Student: There was No student Matching the Provided Student ID in the Database."	Same as expected result	Pass
Find a Student: At least one student registered Action: Find an existing student by ID	Print the relevant student information	Same as expected result	Pass
Try to find a student with an invalid ID	Print "Error - Unable to find Student: There was No student Matching the Provided Student ID in the Database."	Same as expected result	Pass
No students registered Action: Try to find a student	Print "Error - Cannot Find Student: The student list is empty or null."	Same as expected result	Pass
Save data to a file and load data from it where no students are registered.	Properly save and load data.	It properly Saved the data but failed to load the data properly This was due to the program reading the file in the wrong order, which was promptly fixed.	**Failed**
Display Registered Students where some students registered	Properly display the students in an alphabetical order, Note lower-case comes after Uppercase	Same as expected result	Pass
Display Registered Students where no students registered	Print "Error - Cannot Display Students: The student list is empty or null."	Same as expected result	Pass
Edit Student Name: Initial state: At least one student registered	Update and Display Updated student information displayed	Same as expected results	Pass

Initial state: No students registered Action: Try to edit a student's name	Print "Error - Cannot choose this option: There are no Students Registered to edit."	Same as expected results	Pass
Set Module Marks: At least one student registered, set valid marks for a student's module	Record marks correctly and print "Student Marks were Successfully recorded."	Same as expected results	Pass
Set Module Marks: At least one student registered, try to set invalid marks (e.g., -10 or 110)	Print "Error - Cannot enter marks: Marks Must be between 0 - 100."	Same as expected results	Pass
Set module marks where no students are registered	Print “Error - Cannot choose this option: There are no Students Registered to edit.”	Same as expected results	Pass
Get System Summary: Some students registered with varying marks, get system summary	Display of total students and module pass counts	Same as expected results	Pass
Get System Summary: No students registered, get system summary	Print "There are 0 Students Registered"	Same as expected results	Pass
Generate Full Report: Some students registered with varying marks, generate full report	Detailed report of all students, sorted by average marks	Same as expected results	Pass
Generate Full Report: No students registered, generate full report	Print "No students to display."	Same as expected results	Pass

Self-Evaluation Form

Criteria	Allocated marks	Expected marks	Total
Task 1 Three marks for each option (1,2,3,4,5,6,7,8)	24	**22**	**(30)**
Menu works correctly	6	**6**	
Student comments **fully implemented and working** The menu for this program uses three classes: Menu.java, MenuOptions.java and MenuMethods.java. This is to ensure that this menu can be reused and modified in any other java console app, this also ensures robustness in my application and it also cleans the code by cutting down duplicated code.			
Task 2 Student class works correctly	14	**12**	**(30)**
Module class works correctly	10	**7**	
Sub menu (A and B works well)	6	**6**	
Student comments **fully implemented and working** The two classes are packaged in the University package. The student class is where most of the work is done, while the Module class acts as a record class used to keep track of modules. Furthermore, I learned about inheritance and implementation in java classes as well.			

Task 3 Report – Generate a summary	7	**7**	**(20)**
Report – Generate the complete report	10	**9**	
Implementation of Bubble sort	3	**3**	
Student comments **fully implemented and working** This part of the coursework uses bubble sort to sort students based on their average, and properly displays all relevant information.			
Task 4 Test case coverage and reasons	6	**4**	**(10)**
Writeup on which version is better and why.	4	**-**	
Student Comments **Fully tested and working.**			
Coding Style (Comments, indentation, style)	7	**6**	**(10)**
Complete the self-evaluation form indicating what you have accomplished to ensure appropriate feedback.	3	**3**	
Totals	100	**84**	**(100)**