Pham Quang Huy - SE06304

DATA STUCTURE AND ALGORITHM

Assignment part 2

1.Student Class

```java
package StudentFile;

23 usages
public class Student {
    3 usages
    private int id;
    4 usages
    private String name;
    9 usages
    private double mark;

    2 usages
    public Student(int id, String name, double mark) {
        this.id = id;
        this.name = name;
        this.mark = mark;
    }
```

Represents a student entity with attributes:

- id: Unique student identifier.
- name: Student's name.
- mark: Student's marks (0–10 range).

1.Student Class

```java
public Student(int id, String name, double mark) {
    this.id = id;
    this.name = name;
    this.mark = mark;
}

5 usages
public int getId() { return id; }

no usages
public String getName() { return name; }

6 usages
public double getMark() { return mark; }

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

1 usage
public void setMark(double mark) { this.mark = mark; }

1 usage
public String getRanking () {
    if (mark >= 9 && mark <= 10) return "Excellent";
    else if (mark >= 7.5) return "Very Good";
    else if (mark >= 6.5) return "Good";
    else if (mark >= 5.0) return "Medium";
    else return "Fail";
}
```

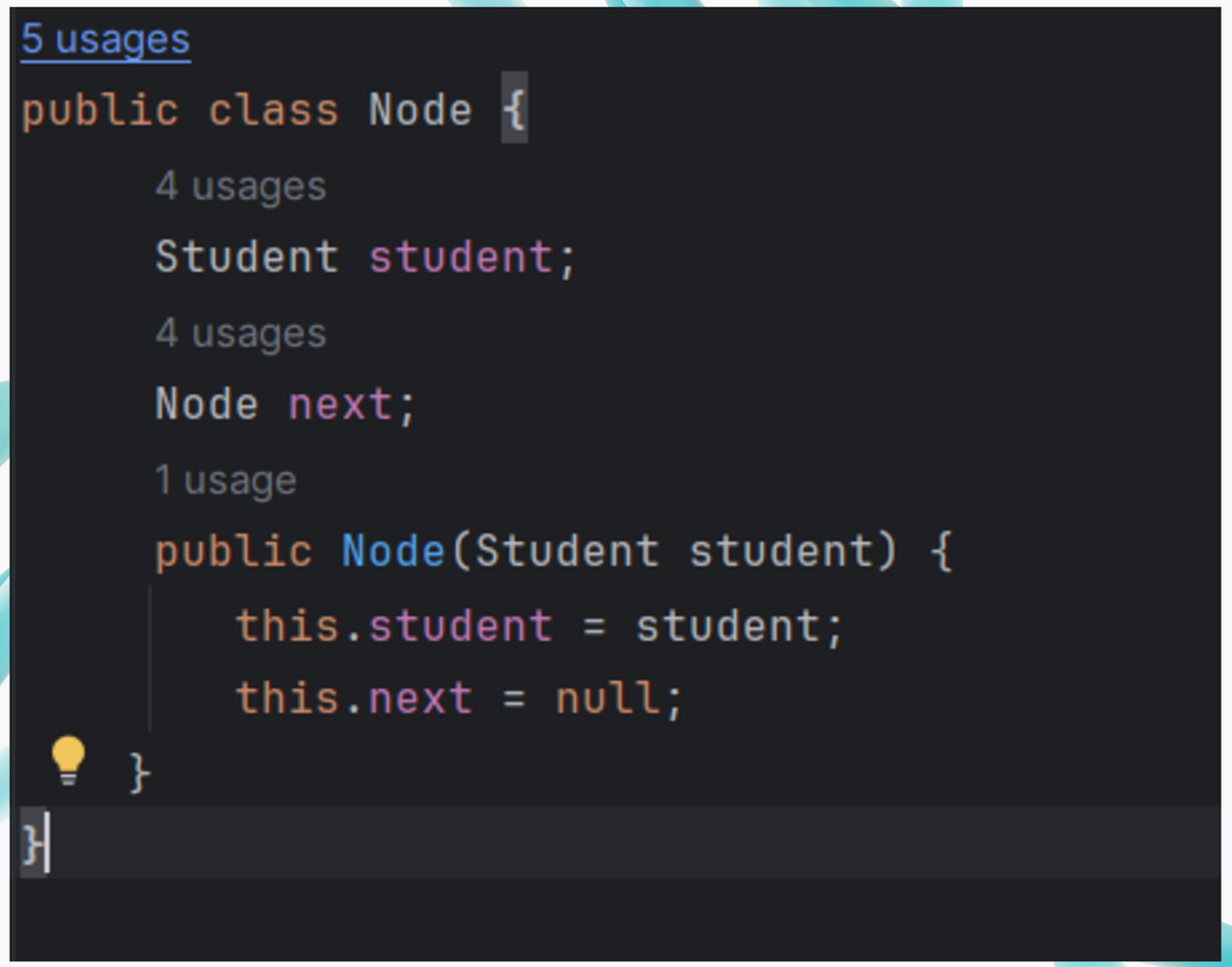
Key methods:

- Constructor initializes the attributes.
- Getters/Setters for accessing and modifying attributes.
- getRanking determines a ranking (e.g., Excellent, Good) based on marks.
- toString provides a string representation of a student (e.g., ID, name, marks, rank).

2.Node Class

Represents a node in a linked list.

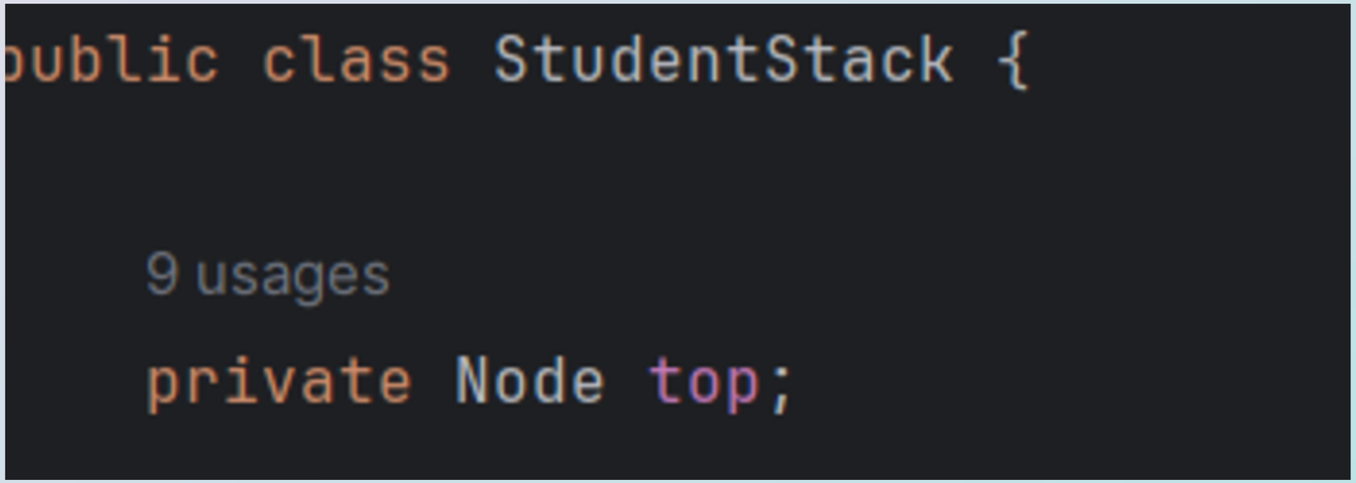
- Attributes:
  - student: Stores a Student object.
  - next: Points to the next node in the stack.
- Constructor initializes the student and sets next to null.



3. StudentStack Class

Attributes

1. Node top:
    - A reference to the top node in the stack.
    - Represents the most recently added Student.



3. StudentStack Class

Constructor

- **StudentStack()**
  - Initializes the stack with no elements:
    - Sets top to null.

```
5 usages
public StudentStack() { this.top = null; }
```

3. StudentStack Class

Methods

- **isEmpty()**
  - **Purpose**: Checks whether the stack is empty.
  - **Returns**: true if the top is null, otherwise false.
  - **Usage**: Prevents operations like pop() or peek() on an empty stack.
  - **Time** Complexity: O(1)O(1)O(1).

```
public boolean isEmpty() { return top == null; }
```

3. StudentStack Class

Methods

- **push(Student student)**
  - **Purpose:** Adds a new Student object to the top of the stack.
  - **Steps**:
    - Creates a new Node containing the provided Student.
    - Links the new Node to the current top.
    - Updates top to the new Node.
  - **Time Complexity:** O(1)

```
13 usages
public void push(Student student){
    Node newNode = new Node(student);
    newNode.next = top;
    top = newNode;
}
```

3. StudentStack Class

Methods

- **pop()**
  - **Purpose:** Removes and returns the Student at the top of the stack.
  - **Steps:**
    - Checks if the stack is empty using isEmpty().
    - **If not empty:**
      - Stores the Student from the top node.
      - Updates top to point to the next node.
      - Returns the removed Student.
    - **If empty:**
      - Displays a message and returns null.
  - **Time Complexity:** O(1)

```
11 usages
public Student pop(){
    if(isEmpty()){
        System.out.println("Stack is empty. No students to remove.");
        return null;
    }
    Student removedStudent = top.student;
    top = top.next;
    return removedStudent;
}
```

3. StudentStack Class

Methods

- **peek()**
  - **Purpose:** Returns the Student at the top of the stack without removing it.
  - **Steps:**
    - Checks if the stack is empty using isEmpty().
    - **If not empty:**
      - Returns the Student from the top node.
    - **If empty:**
      - Displays a message and returns null.
  - **Time Complexity:** O(1).

```
1 usage
public Student peek(){
    if (isEmpty()){
        System.out.println("Stack is empty. No students to show");
        return null;
    }
    return top.student;
}
```

3. StudentStack Class

Methods

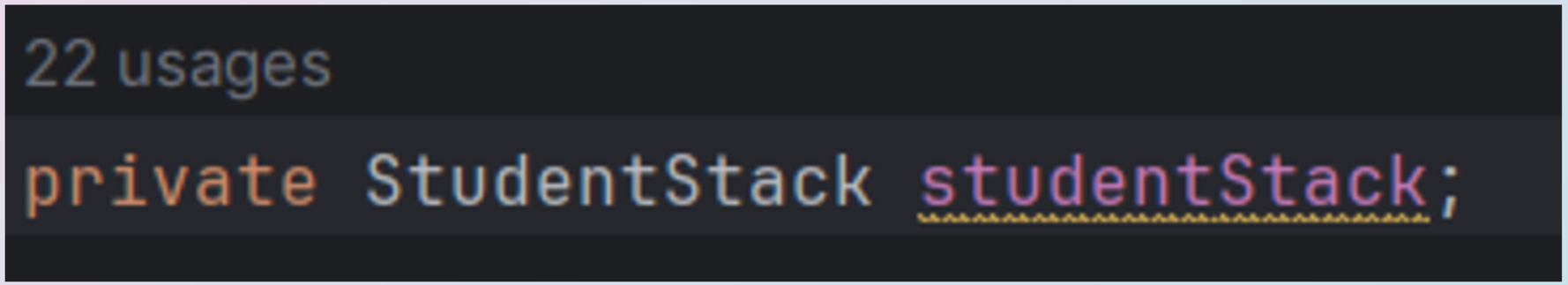
- **displayStack()**
  - **Purpose:** Prints the details of all Student objects in the stack.
  - **Steps:**
    - Checks if the stack is empty using isEmpty().
    - **If not empty:**
      - Traverses the stack from the top to the bottom, printing each Student.
    - **If empty:**
      - Displays a message.
  - **Time Complexity:** O(n), where nnn is the number of students in the stack.

```
1 usage
public void displayStack(){
    if (isEmpty()){
        System.out.println("No students in the stack.");
        return;
    }
    Node current = top;
    while (current != null){
        System.out.println(current.student);
        current = current.next;
    }
}
```

4. StudentManagement Class

Attributes

1. **StudentStack studentStack:**
    - A StudentStack instance used to store and manage the Student objects.



4. StudentManagement Class

Constructor

- **StudentManagement():**
  - Initializes a new empty StudentStack.

```
usage
public StudentManagement() { this.studentStack = new StudentStack(); }
```

4. StudentManagement Class

Methods

- **addStudent(Student student)**
  - **Purpose:** Adds a new Student to the stack if the ID is unique.
  - **Process:**
    - Checks if the student ID already exists using searchStudent.
    - If it doesn't exist, adds the student using push.
    - Displays an appropriate message.
  - **Time Complexity:** O(n), due to the searchStudent operation.

```
2 usages
public void addStudent (Student student){
    // Kiểm tra xem sinh viên đã tồn tại chưa
    if (searchStudent(student.getId()) != null) {
        System.out.println("Error: Student with ID " + student.getId() + " already exists.");
        return;
    }
    studentStack.push(student);
    System.out.println("Added student: " + student);
}
```

4. StudentManagement Class

Methods

- **updateStudent(int id, String newName, double newMark)**
  - **Purpose:** Updates the name and mark of a student identified by id.
  - **Process:**
    - Temporarily transfers all students to a secondary stack (tempStack).
    - Updates the student's details if found.
    - Restores the stack after modification.
  - **Edge Cases:**
    - Displays a message if the student is not found.
  - **Time Complexity:** O(n).

```
1 usage
public void updateStudent(int id, String newName, double newMark){
    StudentStack tempStack = new StudentStack();
    boolean found = false;

    while (!studentStack.isEmpty()){
        Student currentStudent = studentStack.pop();
        if (currentStudent.getId() == id){
            currentStudent.setName(newName);
            currentStudent.setMark(newMark);
            tempStack.push(currentStudent);
            found = true;
        } else{
            tempStack.push(currentStudent);
        }
    }

    // Khôi phục lại stack ban đầu
    while (!tempStack.isEmpty()){
        studentStack.push(tempStack.pop());
    }

    if (!found){
        System.out.println("Student with ID " + id + " not found");
    } else {
        System.out.println("Updated student with ID: " + id);
    }
}
```

4. StudentManagement Class

Methods

- **deleteStudent(int id)**
  - **Purpose:** Removes a student identified by id from the stack.
  - **Process:**
    - Transfers students to a temporary stack, skipping the one to delete.
    - Restores the stack after deletion.
  - **Edge Cases:**
    - Displays a message if the student is not found.
  - **Time Complexity:** O(n).

```
1 usage
public void deleteStudent(int id){
    StudentStack tempStack = new StudentStack();
    boolean found = false;

    while (!studentStack.isEmpty()){
        Student currentStudent = studentStack.pop();
        if (currentStudent.getId() == id) {
            // Nếu tìm thấy, bỏ qua sinh viên này
            found = true;
        } else {
            tempStack.push(currentStudent);
        }
    }
    // Khôi phục lại stack ban đầu
    while (!tempStack.isEmpty()) {
        studentStack.push(tempStack.pop());
    }

    if (!found) {
        System.out.println("Student with ID " + id + " not found.");
    } else {
        System.out.println("Deleted student with ID: " + id);
    }
}
```

4. StudentManagement Class

Methods

- **searchStudent(int id)**
  - **Purpose:** Searches for a student by id.
  - **Process:**
    - Traverses the stack to find the student.
    - Restores the stack to its original state after searching.
  - **Returns:**
    - The Student object if found, otherwise null.
  - **Time Complexity:** O(n).

```
2 usages
public Student searchStudent(int id) {
    StudentStack tempStack = new StudentStack();
    Student foundStudent = null;

    // Tìm kiếm sinh viên trong stack
    while (!studentStack.isEmpty()) {
        Student currentStudent = studentStack.pop();
        if (currentStudent.getId() == id) {
            foundStudent = currentStudent;
        }
        tempStack.push(currentStudent);
    }

    // Khôi phục lại stack ban đầu
    while (!tempStack.isEmpty()) {
        studentStack.push(tempStack.pop());
    }

    return foundStudent;
}
```

4. StudentManagement Class

Methods

- **Quick Sort**
  - Converts the stack to an ArrayList, performs the sort, and pushes sorted elements back to the stack.
  - **Time Complexity:** O(n log n) on average; O(n^2) in the worst case.
  - **Space Complexity:** O(n), for the array list.

```
1 usage
public void quickSortStudentsByMarks() {
    Runtime runtime = Runtime.getRuntime();
    runtime.gc();
    long memoryBeforeSort = runtime.totalMemory() - runtime.freeMemory();
    long start = System.nanoTime();
    // Chuyển stack sang mảng để thực hiện quicksort
    ArrayList<Student> studentList = new ArrayList<>();
    while (!studentStack.isEmpty()) {
        studentList.add(studentStack.pop());
    }

    // Gọi phương thức quicksort
    quickSort(studentList, low: 0, high: studentList.size() - 1);

    // Đưa các sinh viên đã sắp xếp trở lại stack
    for (int i = studentList.size() - 1; i >= 0; i--) {
        studentStack.push(studentList.get(i));
    }

    long end = System.nanoTime();
    long memoryAfterSort = runtime.totalMemory() - runtime.freeMemory();
    System.out.println("Students sorted by marks using Quick Sort.");
    System.out.println("Run time Quick Sort:"+ (end - start) + " ns");
    System.out.println("Quick Sort Memory Usage: " + (memoryAfterSort-memoryBeforeSort) + " bytes");
}
```

4. StudentManagement Class

Methods

- **Quick Sort**
  - Converts the stack to an ArrayList, performs the sort, and pushes sorted elements back to the stack.
  - **Time Complexity:** O(n log n) on average; O(n^2) in the worst case.
  - **Space Complexity:** O(n), for the array list.

```
private void quickSort(ArrayList<Student> arr, int low, int high) {
    if (low < high) {
        int partitionIndex = partition(arr, low, high);

        quickSort(arr, low, high: partitionIndex - 1);
        quickSort(arr, low: partitionIndex + 1, high);
    }
}

1 usage
private int partition(ArrayList<Student> arr, int low, int high) {
    double pivot = arr.get(high).getMark();
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr.get(j).getMark() <= pivot) {
            i++;
            // Swap
            Student temp = arr.get(i);
            arr.set(i, arr.get(j));
            arr.set(j, temp);
        }
    }

    // Swap pivot
    Student temp = arr.get(i + 1);
    arr.set(i + 1, arr.get(high));
    arr.set(high, temp);

    return i + 1;
}
```

4. StudentManagement Class

Methods

- **Bubble Sort**
  - Performs bubble sort on an ArrayList and restores the sorted students to the stack.
  - **Time Complexity:** O(n^2).
  - **Space Complexity:** O(n).

```
public void bubbleSortStudentsByMarks() {
    Runtime runtime = Runtime.getRuntime();
    runtime.gc();
    long memoryBeforeSort = runtime.totalMemory() - runtime.freeMemory();
    long start = System.nanoTime();

    ArrayList<Student> studentList = new ArrayList<>();
    while (!studentStack.isEmpty()) {
        studentList.add(studentStack.pop());
    }

    int n = studentList.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (studentList.get(j).getMark() > studentList.get(j + 1).getMark()) {
                // Swap
                Student temp = studentList.get(j);
                studentList.set(j, studentList.get(j + 1));
                studentList.set(j + 1, temp);
            }
        }
    }

    for (int i = studentList.size() - 1; i >= 0; i--) {
        studentStack.push(studentList.get(i));
    }
    long end = System.nanoTime();
    long memoryAfterSort = runtime.totalMemory() - runtime.freeMemory();
    System.out.println("Students sorted by marks using Bubble Sort.");
    System.out.println("Run time Bubble:"+ (end - start) + " ns");
    System.out.println("Bubble Sort Memory Usage: " + (memoryAfterSort-memoryBeforeSort) + " bytes");
}
```

4. StudentManagement Class

Methods

- **Stack-Based Sorting**
  - Uses a second stack to sort elements directly within stack constraints.
  - Compares marks and transfers elements between stacks to maintain sorted order.
  - **Time Complexity:** O(n^2), due to repeated stack operations.
  - **Space Complexity:** O(n), for the auxiliary stack.

```
1 usage
public void sortStudentsByMarks() {
    Runtime runtime = Runtime.getRuntime();
    runtime.gc();
    long memoryBeforeSort = runtime.totalMemory() - runtime.freeMemory();
    long start = System.nanoTime();
    StudentStack sortedStack = new StudentStack();

    while (!studentStack.isEmpty()) {
        Student currentStudent = studentStack.pop();
        // Chèn currentStudent vào sortedStack theo thứ tự phù hợp
        while (!sortedStack.isEmpty() && sortedStack.peek().getMark() > currentStudent.getMark()) {
            studentStack.push(sortedStack.pop());
        }
        sortedStack.push(currentStudent);
    }

    // Khôi phục lại stack ban đầu với các phần tử đã sắp xếp
    while (!sortedStack.isEmpty()) {
        studentStack.push(sortedStack.pop());
    }
    long end = System.nanoTime();
    long memoryAfterSort = runtime.totalMemory() - runtime.freeMemory();
    System.out.println("Students sorted by marks.");
    System.out.println("Run time Sort:"+ (end - start) + " ns");
    System.out.println("Sort Memory Usage: " + (memoryAfterSort-memoryBeforeSort) + " bytes");
}
```

4. StudentManagement Class

Methods

- **displayStudents()**
  - **Purpose:** Prints all students in the stack.
  - **Process:**
    - Delegates the task to StudentStack.displayStack().
  - **Time Complexity:** O(n).

```
1 usage
public void displayStudents() {
    System.out.println("Displaying all students:");
    studentStack.displayStack();
}
```

5. Main Class

Attributes

- **StudentManagement management:**
  - The main controller object that manages the stack of students.
- **Scanner scanner:**
  - Used for capturing user inputs.
- **Random random:**
  - Generates random names and marks for initial students.
- **String[] names:**
  - A predefined array of names used to create random student names.

```
StudentManagement management = new StudentManagement();
Scanner scanner = new Scanner(System.in);
Random random = new Random();

// Mảng tên để tạo ngẫu nhiên
String[] names = {
        "Huy", "Quang", "Tuan", "Chi", "Vu", "Hoang",
        "Han", "Tu", "Mai", "Lan", "Minh", "Anh",
        "Trang", "Linh", "Duc", "Khoa", "Ngoc", "Hung"
};
```

5. Main Class

Initial Setup

- **Prompts the user to enter the number of students to initialize.**
- **Generates random Student objects with:**
  - Sequential IDs.
  - Random names from the names array.
  - Random marks between 0.0 and 10.0, rounded to one decimal place.

Input Validation:

- Ensures the number of students is greater than 0.
- Handles invalid inputs gracefully by displaying appropriate error messages.

```
// Nhập số lượng sinh viên ban đầu
System.out.print("Enter the number of Student: ");
try {
    int initialStudentCount = scanner.nextInt();
    scanner.nextLine(); // Clear buffer

    if (initialStudentCount <= 0) {
        throw new IllegalArgumentException("The number of Student must be more than 0");
    }

    // Tạo danh sách sinh viên ban đầu
    for (int i = 1; i <= initialStudentCount; i++) {
        // Chọn tên ngẫu nhiên
        String randomName = names[random.nextInt(names.length)];

        // Tạo điểm số ngẫu nhiên từ 0 đến 10
        double randomMarks = Math.round(random.nextDouble() * 10 * 10.0) / 10.0;

        // Thêm sinh viên với ID tăng dần và tên, điểm số ngẫu nhiên
        management.addStudent(new Student(i, randomName, randomMarks));
    }
} catch (IllegalArgumentException e) {
    System.out.println("Error: " + e.getMessage());
    return;
} catch (Exception e) {
    System.out.println("Error: Please enter a valid number.");
    return;
}
```

5. Main Class

Menu Options

- **Add Student**
  - **Prompts for:**
    - **ID:** An integer.
    - **Name:** Validated to ensure it contains only letters and spaces.
    - **Marks:** A floating-point number between 0 and 10.
  - Calls management.addStudent() to add the new student.
  - Handles exceptions for invalid inputs or duplicate IDs.

```
case 1:
    try {
        System.out.print("Enter Student ID: ");
        int id = scanner.nextInt();
        scanner.nextLine(); // Clear buffer

        System.out.print("Enter Student Name: ");
        String name = scanner.nextLine();
        if (!name.matches( regex: "[a-zA-Z\\s]+")) { // Validate name
            if (name.matches( regex: "")) {
                throw new IllegalArgumentException("Invalid name! Name can't be blank.");
            }
            throw new IllegalArgumentException("Invalid name! Only letters and spaces are allowed.");
        }

        System.out.print("Enter Student Marks: ");
        double marks = scanner.nextDouble();
        if (marks < 0 || marks > 10) { // Validate marks
            throw new IllegalArgumentException("Invalid marks! Enter a value between 0 and 10.");
        }

        management.addStudent(new Student(id, name, marks));
    } catch (IllegalArgumentException e) {
        System.out.println("Error: " + e.getMessage());
    }
    break;
```

5. Main Class

Menu Options

- **Update Student**
  - **Prompts for:**
    - **ID:** The ID of the student to update.
    - **New Name:** Validated to ensure proper format.
    - **New Marks:** Validated to ensure they are between 0 and 10.
  - Calls management.updateStudent() with the new values.
  - Displays appropriate messages for success or failure.

```
case 2:
    try {
        System.out.print("Enter Student ID to update: ");
        int updateId = scanner.nextInt();
        scanner.nextLine(); // Clear buffer

        System.out.print("Enter new Student Name: ");
        String newName = scanner.nextLine();
        if (!newName.matches( regex: "[a-zA-Z\\s]+")) { // Validate name
            throw new IllegalArgumentException("Invalid name! Only letters and spaces are allowed.");
        }

        System.out.print("Enter new Student Marks: ");
        double newMark = scanner.nextDouble();
        if (newMark < 0 || newMark > 10) { // Validate marks
            throw new IllegalArgumentException("Invalid marks! Enter a value between 0 and 10.");
        }

        management.updateStudent(updateId, newName, newMark);
    } catch (IllegalArgumentException e) {
        System.out.println("Error: " + e.getMessage());
    }
    break;
```

5. Main Class

Menu Options

- **Delete Student**
  - Prompts for the ID of the student to delete.
  - Calls management.deleteStudent() .
  - Displays a success or failure message.

```
case 3:
    try {
        System.out.print("Enter Student ID to delete: ");
        int deleteId = scanner.nextInt();
        management.deleteStudent(deleteId);
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
    }
    break;
```

5. Main Class

Menu Options

- **Search Student**
  - Prompts for the ID of the student to search.
  - Calls management.searchStudent() .
  - Displays the student's details if found or a not-found message otherwise.

```
case 4:
    try {
        System.out.print("Enter Student ID to search: ");
        int searchId = scanner.nextInt();
        Student student = management.searchStudent(searchId);
        if (student != null) {
            System.out.println("Student Found: " + student);
        } else {
            System.out.println("Student with ID " + searchId + " not found.");
        }
    } catch (Exception e) {
        System.out.println("Error: " + e.getMessage());
    }
    break;
```

5. Main Class

Menu Options

- **Sort Students by Marks**
  - Calls management.sortStudentsByMarks() to sort students using a stack-based approach.
- **Sort Students by Marks (Bubble Sort)**
  - Calls management.bubbleSortStudentsByMarks() to sort using Bubble Sort.
- **Sort Students by Marks (Quick Sort)**
  - Calls management.quickSortStudentsByMarks() to sort using Quick Sort.

```
case 5:
    management.sortStudentsByMarks();
    break;
```

```
case 6:
    management.bubbleSortStudentsByMarks();
    break;
```

```
case 7:
    management.quickSortStudentsByMarks();
    break;
```

5. Main Class

Menu Options

- **Display All Students**
  - Calls management.displayStudents() to show all students currently in the stack.
- **Exit**
  - Gracefully exits the program by closing the Scanner and terminating execution.

```
case 8:
    management.displayStudents();
    break;

case 9:
    System.out.println("Exiting the system.");
    scanner.close();
    System.exit( status: 0);
    break;
```