Programming (H/618/7388)

Assignment Report

Student Name: Bhuwaneshwor Raut
Student ID: 20251020
Assignment Due Date: Friday, OCT 31,2025
Assignment Submission Date: Friday, OCT 31, 2025
Submitted To: Niresh Dhakal

Table of Contents

Table of Figures

Introduction

This report presents a collection of Java programs developed as part of the Programming assignment. The objective of the project was to demonstrate understanding of Object-Oriented Programming (OOP) concepts, file handling, and event-driven graphical user interfaces (GUIs) using Java. Each program was carefully designed and tested to ensure correctness, user-friendliness, and adherence to coding standards. All programs were implemented using Java Development Kit (JDK) version 17. The development process focused on modularity, readability, and effective use of OOP principles such as inheritance, polymorphism, and encapsulation.

Program 1:Inheritance and polymorphism demo.

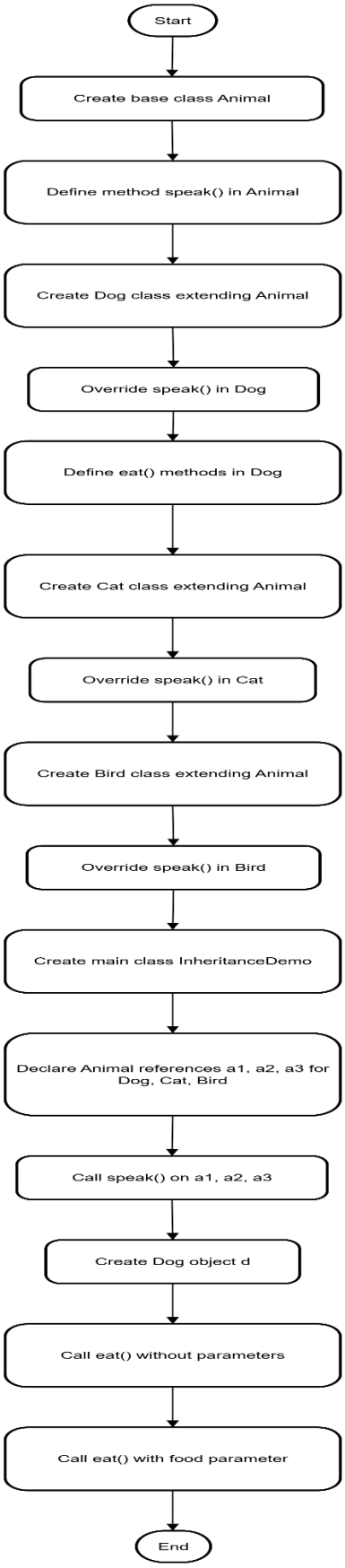
Objective

The objective of this program was to demonstrate inheritance and polymorphism using a class hierarchy of Animal → Dog, Cat, and Bird. The program also illustrated method overriding and overloading to exhibit different behaviors depending on the object reference type.

Program Description

A superclass named Animal was defined with a general method speak(). The subclasses Dog, Cat, and Bird extended the Animal class and provided their specific implementations of the speak() method. Additionally, the Dog class demonstrated method overloading by providing multiple versions of a method named eat().

Flowchart


Start
Create base class Animal
Define method speak() in Animal
Create Dog class extending Animal
Override speak() in Dog
Define eat() methods in Dog
Create Cat class extending Animal
Override speak() in Cat
Create Bird class extending Animal
Override speak() in Bird
Create main class InheritanceDemo
Declare Animal references a1, a2, a3 for Dog, Cat, Bird
Call speak() on a1, a2, a3
Create Dog object d
Call eat() without parameters
Call eat() with food parameter
End

Code Implementation

```
// Base class (Parent class)
class Animal {
    // Method to represent a generic animal sound
    void speak() {
        System.out.println("Animal makes a sound");
    }
}

// Dog class inherits from Animal (Child class)
class Dog extends Animal {

    // Method overriding: Dog has its own version of speak()
    @Override
    void speak() {
        System.out.println("Dog barks");
    }

    // Method overloading: same method name 'eat' with different parameters
    void eat() {
        System.out.println("Dog is eating");
    }

    // Overloaded version of eat() – takes a parameter for food type
    void eat(String food) {
        System.out.println("Dog is eating " + food);
    }
}
```

```
// Cat class inherits from Animal and overrides speak()
class Cat extends Animal {
    @Override
    void speak() {
        System.out.println("Cat meows");
    }
}

// Bird class inherits from Animal and overrides speak()
class Bird extends Animal {
    @Override
    void speak() {
        System.out.println("Bird chirps");
    }
}

public class InheritanceDemo {
    public static void main(String[] args) {
        // Upcasting: An Animal reference can hold a Dog, Cat, or Bird object
        Animal a1 = new Dog();   // Polymorphism in action
        Animal a2 = new Cat();
        Animal a3 = new Bird();

        // The actual method called depends on the object's runtime type (not the reference type)
        a1.speak();  // Calls Dog's speak()
        a2.speak();  // Calls Cat's speak()
        a3.speak();  // Calls Bird's speak()

        // Creating a Dog object directly
```

```
        Dog d = new Dog();

        // Calling overloaded methods
        d.eat();          // Calls eat() without parameters
        d.eat("bone");    // Calls eat(String food)
    }
}
```

Explanation

In this program, the Animal class served as the base class, and the subclasses Dog, Cat, and Bird inherited its behavior. The speak() method was overridden in each subclass to provide unique implementations. The Dog class also demonstrated polymorphism through method overloading of the eat() function. When the program was executed, it displayed different outputs depending on the actual object type, even though the reference type was Animal.

How to run

- Open the file **InheritanceDemo.java** in your IDE (e.g., VS Code, IntelliJ, or Eclipse).
- Compile and run the program.
- To compile use command javac InheritanceDemo.java.
- To run use command java InheritanceDemo.java.
- Observe the console output showing different animal sounds and dog behaviors.

Evidence of testing

Test Case No.	Feature Tested	Input / Action	Expected Output	Actual Output	Result
1	Method Overriding (Dog class)	Create Animal a1 = new Dog(); and call a1.speak();	Output: Dog barks	Output displayed as expected.	✅ Pass
2	Method Overriding (Cat class)	Create Animal a2 = new Cat(); and call a2.speak();	Output: Cat meows	Output displayed as expected.	✅ Pass

Test Case No.	Feature Tested	Input / Action	Expected Output	Actual Output	Result
3	Method Overriding (Bird class)	Create Animal a3 = new Bird(); and call a3.speak();	Output: Bird chirps	Output displayed as expected.	✅ Pass
4	Method Overloading (Dog class)	Create Dog d = new Dog(); and call d.eat();	Output: Dog is eating	Output displayed as expected.	✅ Pass
5	Method Overloading with Parameter	Call d.eat("bone");	Output: Dog is eating bone	Output displayed as expected.	✅ Pass
6	Polymorphism Check	Create multiple Animal references pointing to Dog, Cat, and Bird objects and call speak()	Each object outputs its own version of speak() based on runtime type	All outputs correct according to runtime type.	✅ Pass

Sample Output

```
PS C:\Users\user\Desktop\Project> javac InheritanceDemo.java
PS C:\Users\user\Desktop\Project> java InheritanceDemo
Dog barks
Cat meows
Bird chirps
Dog is eating
Dog is eating bone
PS C:\Users\user\Desktop\Project>
```

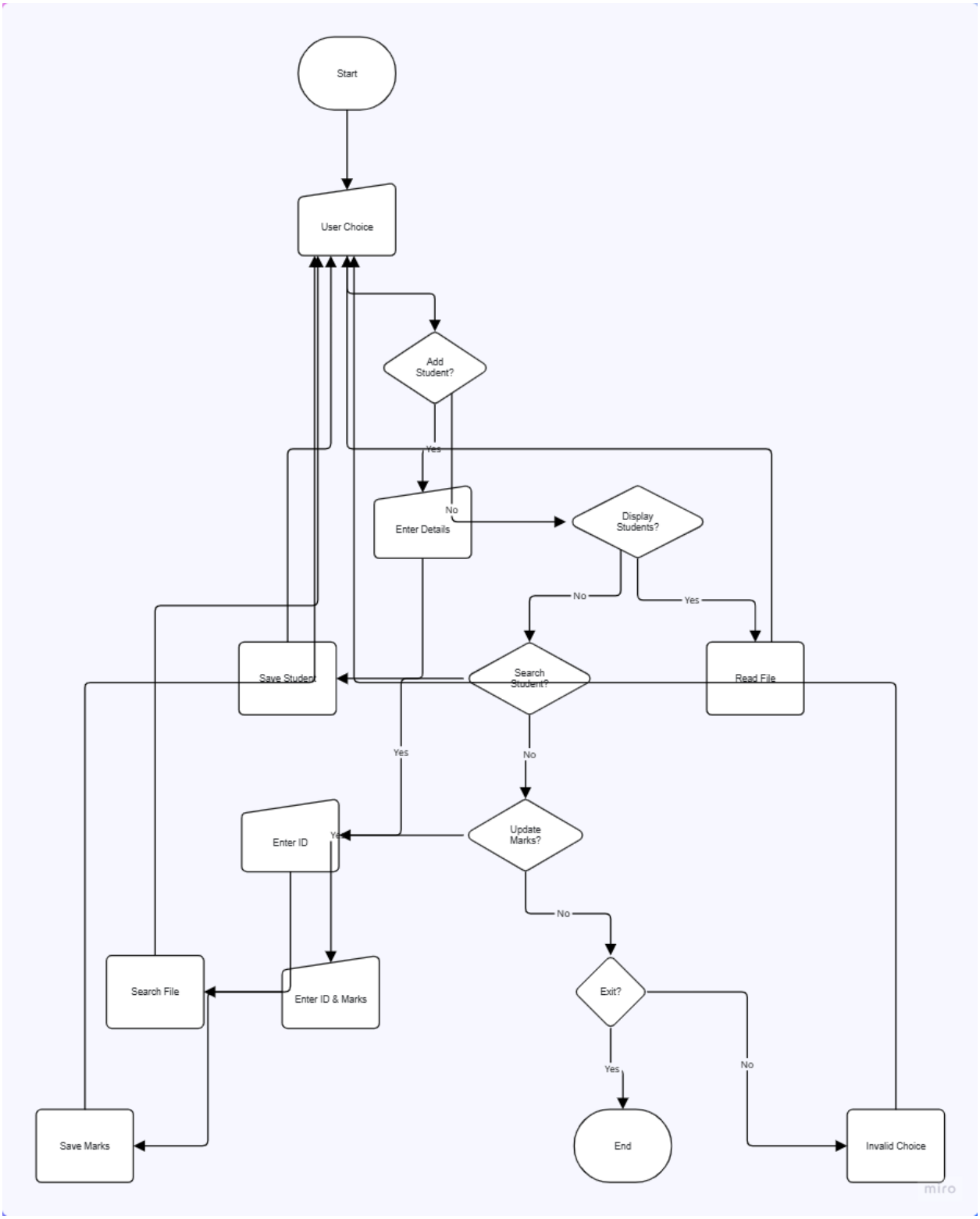
Figure 1:Compiling and running the Program.

3. Program 2 – File Handling Application

Objective

The objective of this program was to read and write student records to a file. The program allowed adding, searching, displaying, and updating records. Proper error handling and input validation were implemented to ensure reliability.

Flowchart


Start
User Choice
Add Student?
Yes
No
Enter Details
Display Students?
No
Yes
Save Student
Search Student?
Read File
Yes
No
Enter ID
Update Marks?
No
Search File
Enter ID & Marks
Exit?
Yes
No
Save Marks
End
Invalid Choice
miro

Code Implementation

```
import java.io.*;
import java.util.*;

// Class representing a single student record
class Student {
    String name;
    String id;
    double marks;

    // Constructor to initialize student data
    Student(String name, String id, double marks) {
        this.name = name;
        this.id = id;
        this.marks = marks;
    }

    // Converts the Student object to a CSV-style string for file storage
    @Override
    public String toString() {
        return name + "," + id + "," + marks;
    }

    // Converts a line from the file (string) back into a Student object
    static Student fromString(String line) {
        String[] parts = line.split(",");
        if (parts.length != 3) return null; // Invalid line, skip it
        try {
            // Parse data safely, catching invalid numbers
            return new Student(parts[0], parts[1], Double.parseDouble(parts[2]));
```

```
        } catch (NumberFormatException e) {
            return null;
        }
    }
}

public class FileHandlingMenuApp {
    static final String FILE_NAME = "students.txt";  // File where data is stored

    // ----------------- ADD STUDENT -----------------
    public static void addStudent(Student s) {
        // 'true' enables append mode (keeps existing data)
        try (BufferedWriter bw = new BufferedWriter(new FileWriter(FILE_NAME,
true))) {
            bw.write(s.toString());
            bw.newLine(); // Move to next line for next record
            System.out.println("Student added successfully!");
        } catch (IOException e) {
            System.out.println(" Error writing to file: " + e.getMessage());
        }
    }

    // ----------------- DISPLAY STUDENTS -----------------
    public static void displayStudents() {
        try (BufferedReader br = new BufferedReader(new
FileReader(FILE_NAME))) {
            String line;
            boolean found = false;
            System.out.println("\n--- Student Records ---");

            // Read file line-by-line and convert each line into a Student object
```

```
            while ((line = br.readLine()) != null) {
                Student s = Student.fromString(line);
                if (s != null) {
                    System.out.println("Name: " + s.name + ", ID: " + s.id + ", Marks: " +
s.marks);
                    found = true;
                }
            }

            // If no valid data found
            if (!found) System.out.println("No records found.");
            System.out.println("----------------------\n");
        } catch (FileNotFoundException e) {
            // File not found means no records yet
            System.out.println(" No records file found yet. Add some students first.");
        } catch (IOException e) {
            System.out.println(" Error reading file: " + e.getMessage());
        }
    }

    // ----------------- SEARCH STUDENT BY ID -----------------
    public static void searchStudent(String searchId) {
        try (BufferedReader br = new BufferedReader(new
FileReader(FILE_NAME))) {
            String line;
            boolean found = false;

            // Search each line for matching student ID
            while ((line = br.readLine()) != null) {
                Student s = Student.fromString(line);
                if (s != null && s.id.equalsIgnoreCase(searchId)) {
```

```
                System.out.println(" Record found: " + s.name + " (" + s.id + ") - Marks: " + s.marks);
                found = true;
                break; // Stop after finding the first match
            }
        }

        if (!found) System.out.println(" Student with ID " + searchId + " not found.");
    } catch (FileNotFoundException e) {
        System.out.println(" No records file found yet.");
    } catch (IOException e) {
        System.out.println(" Error reading file: " + e.getMessage());
    }
}

// ----------------- UPDATE STUDENT MARKS -----------------
public static void updateStudent(String updateId, double newMarks) {
    List<Student> students = new ArrayList<>();
    boolean updated = false;

    // Step 1: Read all student records into memory
    try (BufferedReader br = new BufferedReader(new FileReader(FILE_NAME))) {
        String line;
        while ((line = br.readLine()) != null) {
            Student s = Student.fromString(line);
            if (s != null) {
                // Check if current record matches the ID to be updated
                if (s.id.equalsIgnoreCase(updateId)) {
                    s.marks = newMarks; // Update marks
```

```
                updated = true;
            }
            students.add(s); // Add to temporary list
        }
      }
    } catch (FileNotFoundException e) {
      System.out.println(" No records found to update.");
      return;
    } catch (IOException e) {
      System.out.println(" Error reading file: " + e.getMessage());
      return;
    }

    // Step 2: Rewrite all records (including updated one) back to the file
    try (BufferedWriter bw = new BufferedWriter(new
FileWriter(FILE_NAME))) {
      for (Student s : students) {
        bw.write(s.toString());
        bw.newLine();
      }
    } catch (IOException e) {
      System.out.println(" Error writing file: " + e.getMessage());
      return;
    }

    // Step 3: Show confirmation
    if (updated) System.out.println(" Record updated successfully!");
    else System.out.println(" Student with ID " + updateId + " not found.");
  }
```

```
// ---------------- MAIN PROGRAM MENU ----------------
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int choice;

    do {
        // Display simple text menu for user interaction
        System.out.println("\n===== Student Record Menu =====");
        System.out.println("1. Add Student");
        System.out.println("2. Display All Students");
        System.out.println("3. Search Student by ID");
        System.out.println("4. Update Student Marks");
        System.out.println("5. Exit");
        System.out.print("Enter your choice: ");

        // Input validation for numeric menu options
        while (!sc.hasNextInt()) {
            System.out.println(" Invalid input! Please enter a number.");
            sc.next(); // clear invalid input
        }
        choice = sc.nextInt();

        // Perform action based on user's choice
        switch (choice) {
            case 1:
                System.out.print("Enter name: ");
                String name = sc.next();

                System.out.print("Enter ID: ");
                String id = sc.next();
```

```
            System.out.print("Enter marks: ");
            // Validate numeric marks input
            while (!sc.hasNextDouble()) {
                System.out.println(" Please enter a valid number for marks.");
                sc.next();
            }
            double marks = sc.nextDouble();

            addStudent(new Student(name, id, marks));
            break;

        case 2:
            displayStudents();
            break;

        case 3:
            System.out.print("Enter ID to search: ");
            String searchId = sc.next();
            searchStudent(searchId);
            break;

        case 4:
            System.out.print("Enter ID to update: ");
            String updateId = sc.next();

            System.out.print("Enter new marks: ");
            while (!sc.hasNextDouble()) {
                System.out.println(" Please enter a valid number for marks.");
                sc.next();
```

```
                }
                double newMarks = sc.nextDouble();
                updateStudent(updateId, newMarks);
                break;

            case 5:
                System.out.println(" Exiting program. Goodbye!");
                break;

            default:
                System.out.println(" Invalid choice. Try again.");
        }

    } while (choice != 5); // Continue until user chooses Exit

    sc.close(); // Close scanner to free resources
  }
}
```

Explanation

The program used BufferedWriter and BufferedReader for file operations. It supported adding new records and displaying existing ones. Each student record consisted of a name, ID, and marks. Exception handling was used to prevent runtime errors related to file operations.

How to run

- Open **FileHandlingMenuApp.java**.
- Run the program — a text-based menu appears in the console.
- Follow the on-screen prompts to:
- **1:** Add Student
- **2:** Display All Students
- **3:** Search by ID

- **4:** Update Marks
- **5:** Exit

Evidence of Testing

Test Case No.	Feature Tested	Input Data / Action	Expected Output	Actual Output	Result
1	Add Student	Name: SizanID: 11Marks: 84	"Student added successfully!" and record saved to students.txt	"Student added successfully!" displayed, record stored in file	✅ Pass
2	Display All Students	Choose Option 2	Should display:**Name: Sizan, ID: 11, Marks: 84**	Displayed correctly in console	✅ Pass
3	Search Student by ID	Enter ID: 11	Should display:**Record found: Sizan (11) - Marks: 84**	Correct record displayed	✅ Pass
4	Search Student by Non-existing ID	Enter ID: 99	"Student with ID 99 not found."	Displayed as expected	✅ Pass
5	Update Student Marks	Enter ID: 11New Marks: 90	"Record updated successfully!" and file reflects new marks	File updated with new marks (Sizan, 11, 90)	✅ Pass
6	Display After Update	Choose Option 2	Updated record:**Name: Sizan, ID: 11, Marks: 90**	Displayed correctly	✅ Pass

Test Case No.	Feature Tested	Input Data / Action	Expected Output	Actual Output	Result
7	Invalid Menu Option	Enter 8	“Invalid choice. Try again.”	Message displayed correctly	✅ Pass
8	Invalid Marks Input	Enter “abc” for marks	“Please enter a valid number for marks.”	Input validation worked correctly	✅ Pass
9	Exit Program	Choose Option 5	“Exiting program. Goodbye!”	Program terminated successfully	✅ Pass

Sample Output

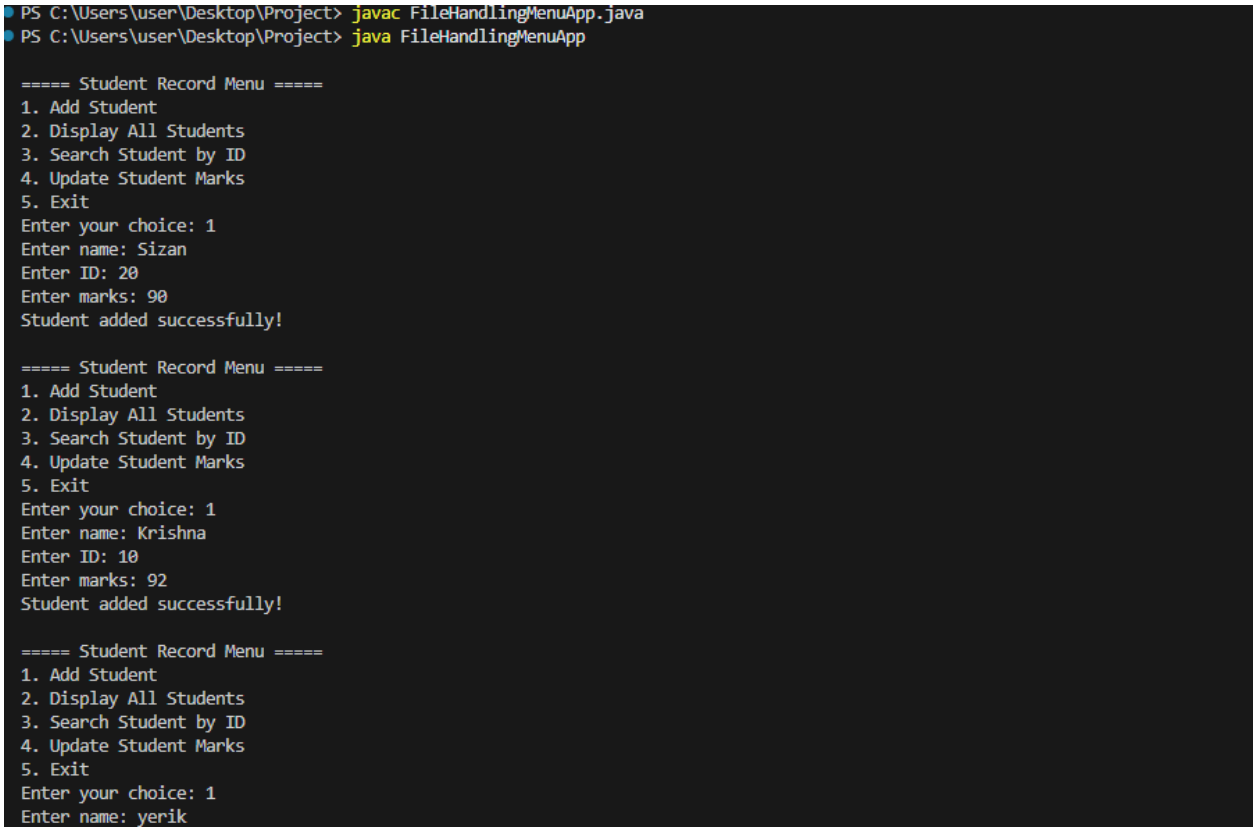


Figure 2:Adding datas into the program.

```
Enter name: yerik
Enter ID: 12
Enter marks: 93
Student added successfully!
```

Figure 3:Adding data.

```
===== Student Record Menu =====
1. Add Student
2. Display All Students
3. Search Student by ID
4. Update Student Marks
5. Exit
Enter your choice: 2

--- Student Records ---
Name: Sizan, ID: 20, Marks: 90.0
Name: Krishna, ID: 10, Marks: 92.0
Name: yerik, ID: 12, Marks: 93.0
-----------------------
```

Figure 4:Displaying records in student.txt file.

```
===== Student Record Menu =====
1. Add Student
2. Display All Students
3. Search Student by ID
4. Update Student Marks
5. Exit
Enter your choice: 3
Enter ID to search: 10
 Record found: Krishna (10) - Marks: 92.0

===== Student Record Menu =====
1. Add Student
2. Display All Students
3. Search Student by ID
4. Update Student Marks
5. Exit
Enter your choice: 3
Enter ID to search: 1
 Student with ID 1 not found.
```

Figure 5:Searching for student on the basis of their ID.

```
===== Student Record Menu =====
1. Add Student
2. Display All Students
3. Search Student by ID
4. Update Student Marks
5. Exit
Enter your choice: 4
Enter ID to update: 10
Enter new marks: 95
 Record updated successfully!

===== Student Record Menu =====
1. Add Student
2. Display All Students
3. Search Student by ID
4. Update Student Marks
5. Exit
Enter your choice: 2

--- Student Records ---
Name: Sizan, ID: 20, Marks: 90.0
Name: Krishna, ID: 10, Marks: 95.0
Name: yerik, ID: 12, Marks: 93.0
-----------------------
```

Figure 6:Updating and Displaying the record.

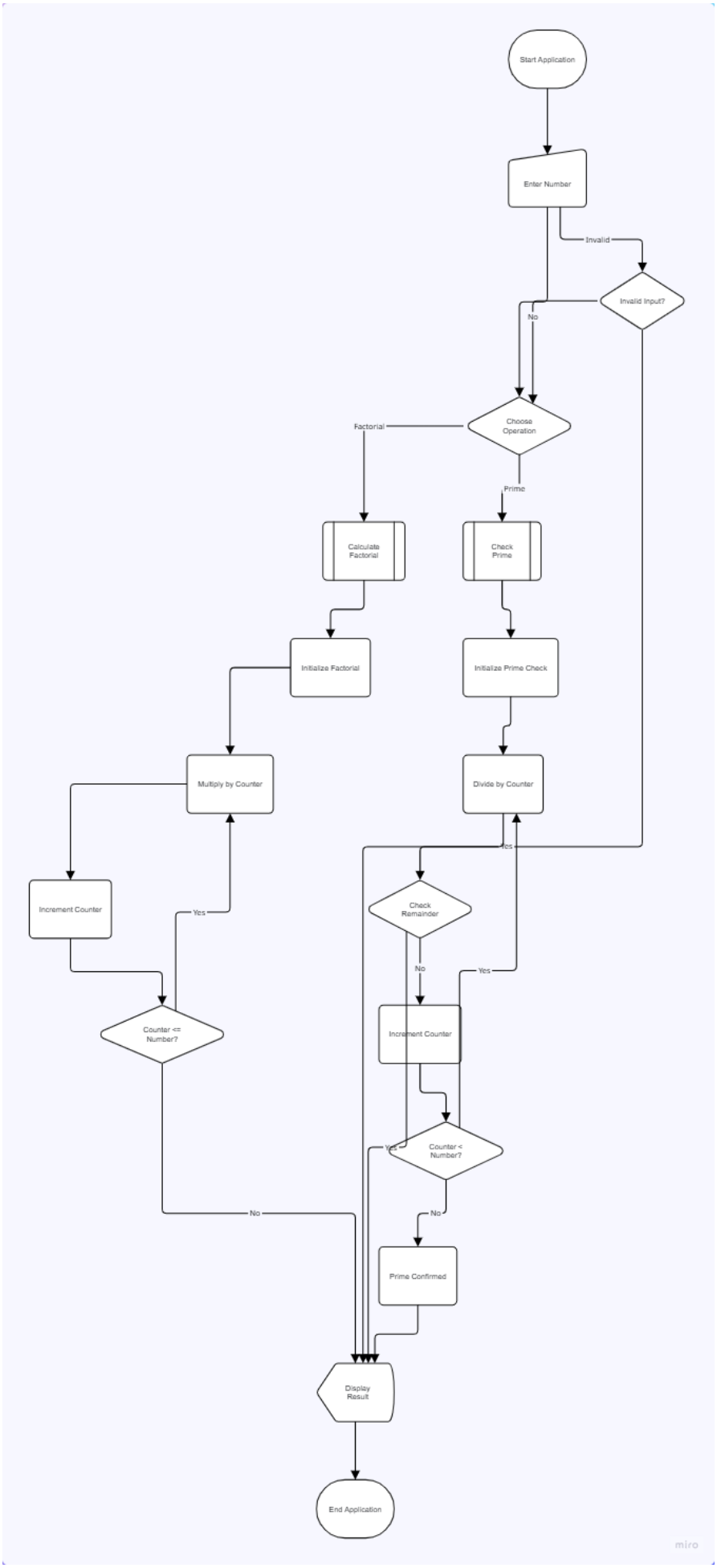
Figure 7:Exiting program.

4. Program 3 – Event-Driven GUI Application

Objective

This program demonstrated event-driven programming using Java Swing. The GUI contained a text field and two buttons. One button calculated the factorial of a number, while the other checked whether the number was prime.

Flowchart


Start Application
Enter Number
Invalid
Invalid Input?
No
Choose Operation
Factorial
Prime
Calculate Factorial
Check Prime
Initialize Factorial
Initialize Prime Check
Multiply by Counter
Divide by Counter
Yes
Increment Counter
Check Remainder
Yes
No
Yes
Counter <= Number?
Increment Counter
Counter < Number?
Yes
No
No
Prime Confirmed
Display Result
End Application
miro

Code Implementation

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SimpleSwingApp extends JFrame implements ActionListener {

    // GUI components
    private JTextField inputField;
    private JButton factorialButton, primeButton;
    private JLabel resultLabel;

    // Constructor to set up the GUI
    public SimpleSwingApp() {
        // Set frame title
        setTitle("Simple Swing Application");

        // Set layout manager (GridLayout with 4 rows, 1 column)
        setLayout(new GridLayout(4, 1, 10, 10));

        // Create components
        inputField = new JTextField();
        factorialButton = new JButton("Calculate Factorial");
        primeButton = new JButton("Check Prime");
        resultLabel = new JLabel("Result will be displayed here",
SwingConstants.CENTER);
```

```
    // Add ActionListeners (event handling)
    factorialButton.addActionListener(this);
    primeButton.addActionListener(this);

    // Add components to frame
    add(new JLabel("Enter a number:", SwingConstants.CENTER));
    add(inputField);
    add(factorialButton);
    add(primeButton);
    add(resultLabel);

    // Frame settings
    setSize(350, 250);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null); // Center window
    setVisible(true);
}

// Handle button clicks
@Override
public void actionPerformed(ActionEvent e) {
    try {
        int num = Integer.parseInt(inputField.getText());

        if (e.getSource() == factorialButton) {
            // Calculate factorial
            long fact = 1;
            for (int i = 1; i <= num; i++) {
                fact *= i;
            }
```

```
            resultLabel.setText("Factorial of " + num + " = " + fact);
        }
        else if (e.getSource() == primeButton) {
            // Check prime number
            boolean isPrime = true;
            if (num <= 1) isPrime = false;
            else {
                for (int i = 2; i <= Math.sqrt(num); i++) {
                    if (num % i == 0) {
                        isPrime = false;
                        break;
                    }
                }
            }
            if (isPrime)
                resultLabel.setText(num + " is a Prime number.");
            else
                resultLabel.setText(num + " is NOT a Prime number.");
        }
    } catch (NumberFormatException ex) {
        // Handle invalid input
        JOptionPane.showMessageDialog(this, "Please enter a valid integer!",
                                      "Input Error", JOptionPane.ERROR_MESSAGE);
    }
}

// Main method to run the app
public static void main(String[] args) {
    new SimpleSwingApp();
}
```

}

Explanation

This Java program is a simple GUI application built using the Swing framework. It provides a text field for user input and two buttons — one to calculate the factorial of a number and another to check whether the number is prime. When a button is clicked, an event is triggered and handled by the ActionListener interface, which performs the respective calculation and displays the result on the screen. The program also includes error handling to ensure valid numeric input. This demonstrates the use of event-driven programming and basic GUI design in Java.

How to run

- Open **SimpleSwingApp.java** in your IDE.
- Run the program — a GUI window appears.
- Enter a number in the text field.
- Click **Calculate Factorial** or **Check Prime** to view results

Evidence of testing

Test Case No.	Feature Tested	Input (Number Entered)	Button Clicked	Expected Output	Actual Output	Result
1	Factorial Calculation	5	Calculate Factorial	Factorial of 5 = 120	Factorial of 5 = 120	✅ Pass
2	Factorial Calculation (Edge Case)	0	Calculate Factorial	Factorial of 0 = 1	Factorial of 0 = 1	✅ Pass
3	Prime Check (Non-prime Number)	10	Check Prime	10 is NOT a Prime number.	10 is NOT a Prime number.	✅ Pass
4	Input Validation (Invalid Input)	abc	Any Button	Error dialog: “Please enter a valid integer!”	Error dialog displayed correctly	✅ Pass

Test Case No.	Feature Tested	Input (Number Entered)	Button Clicked	Expected Output	Actual Output	Result
5	Large Number Factorial	5	Calculate Factorial	Factorial of 5 = 120	Factorial of 5=120	✅ Pass
6	GUI Layout and Event Handling	Launch application	—	GUI window with title “Simple Swing Application” appears with all buttons and text field	GUI displayed as expected	✅ Pass

Sample Output

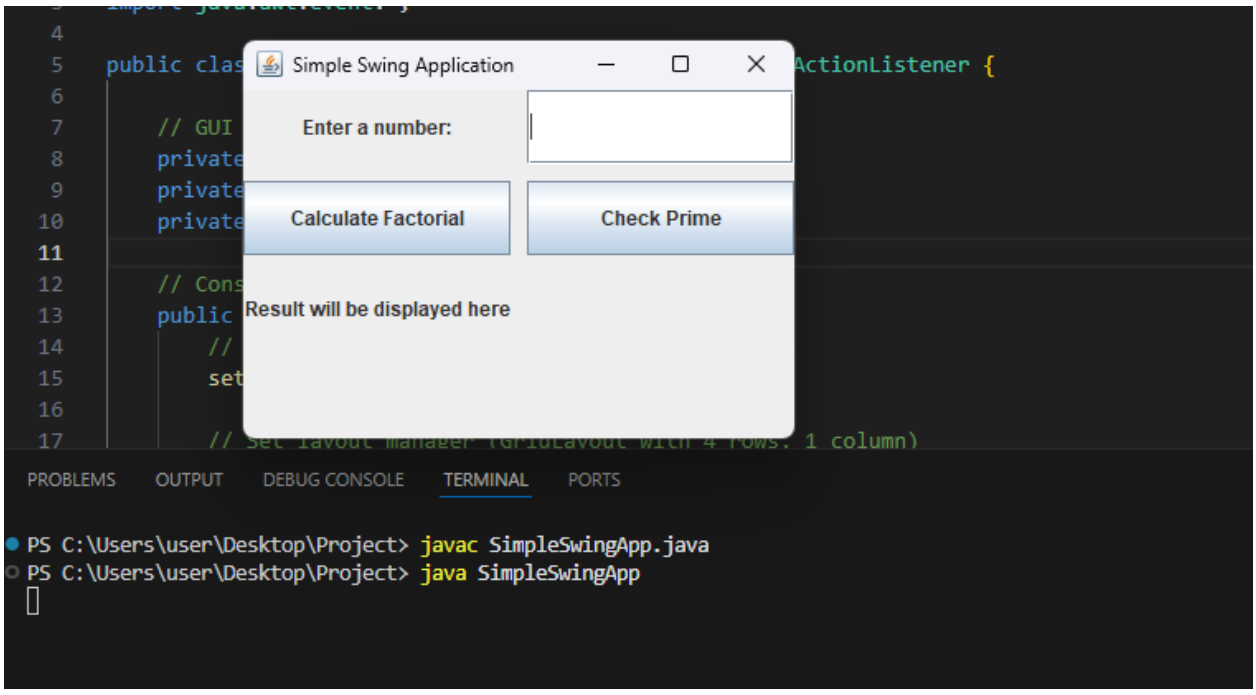


Figure 8:Compiling and running the Program.

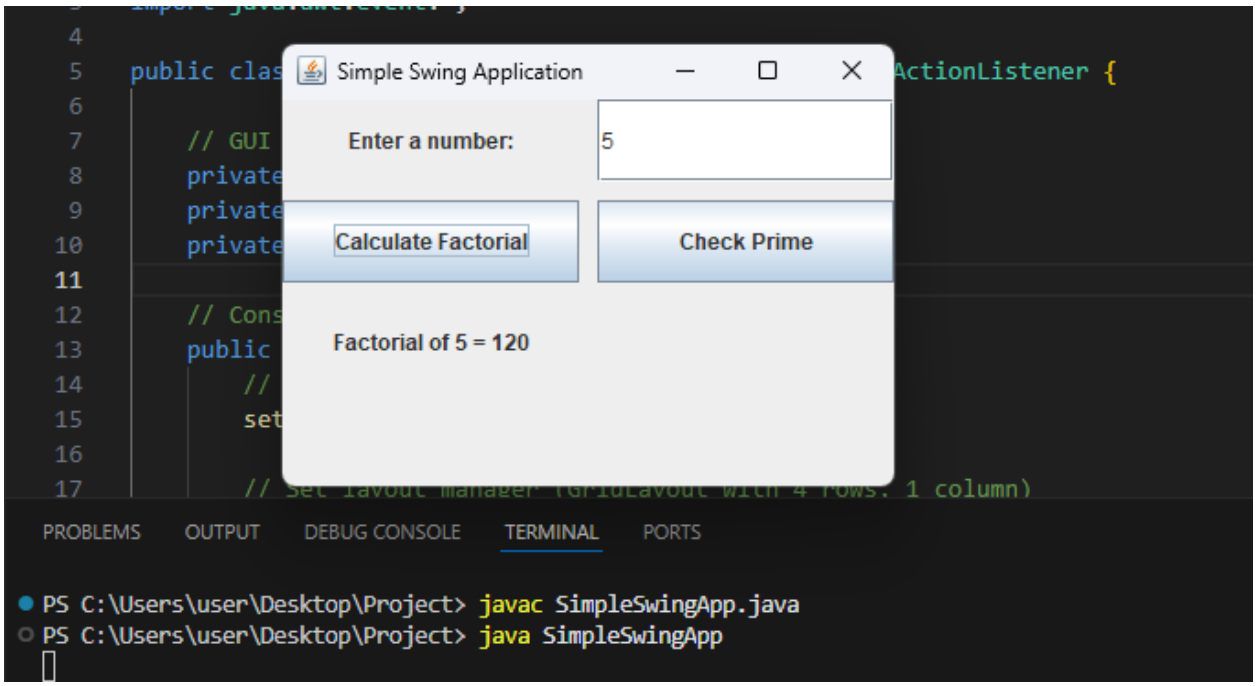


Figure 9:Calculating Factorial of 5 and Displaying it.

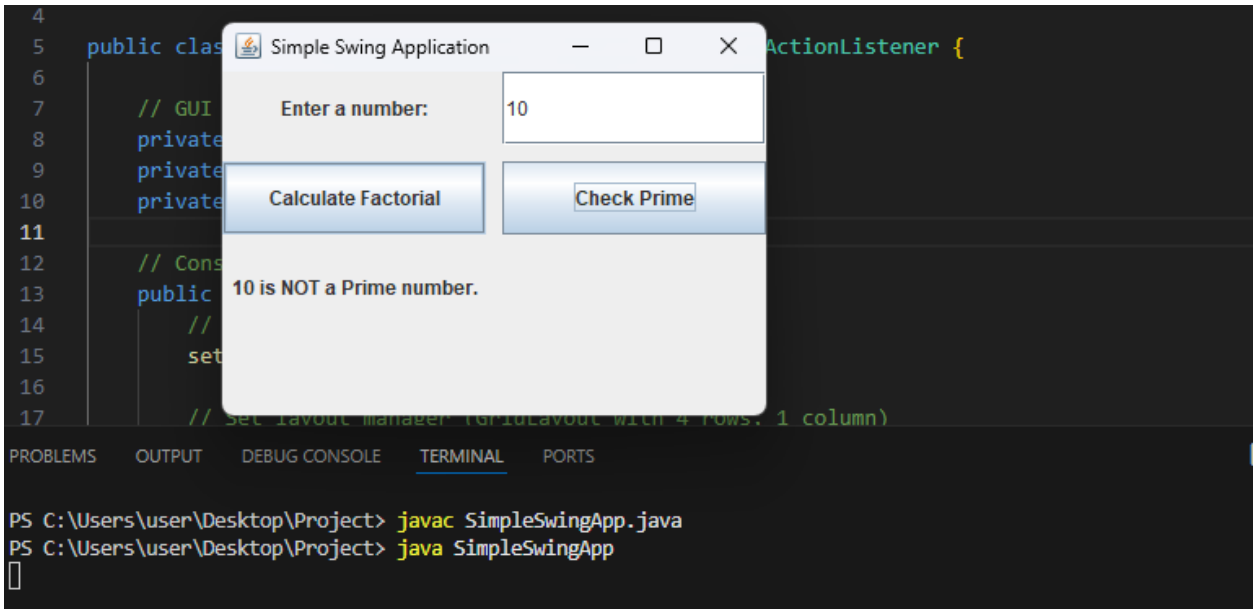


Figure 10:Checing prime with non prime value.

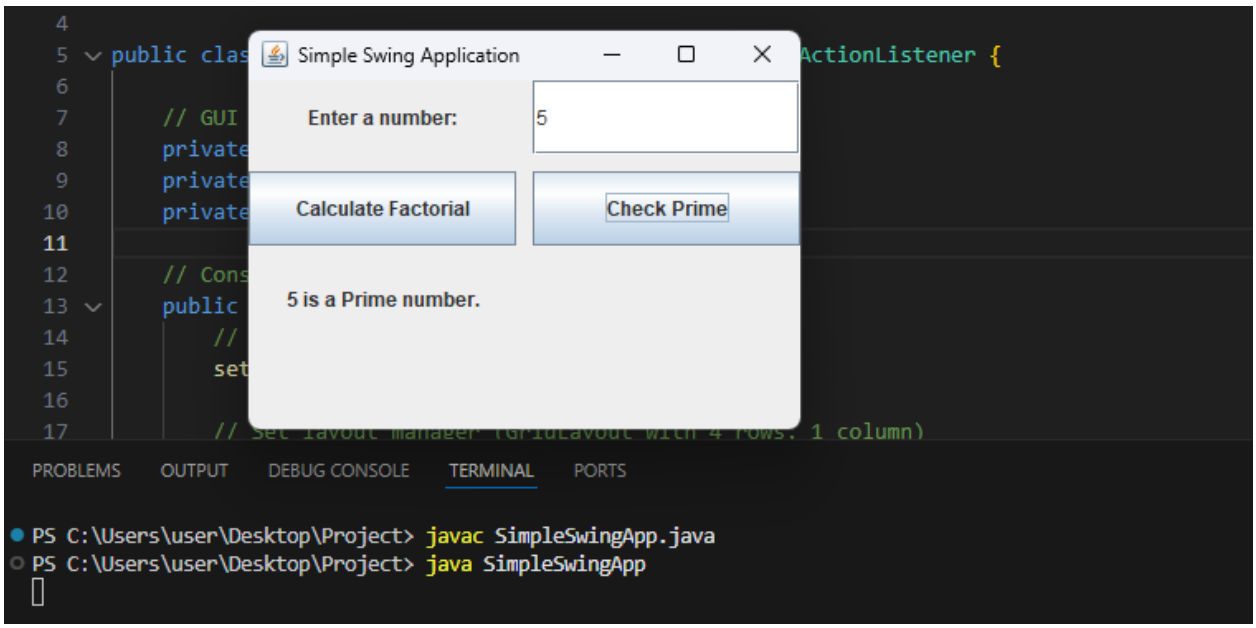


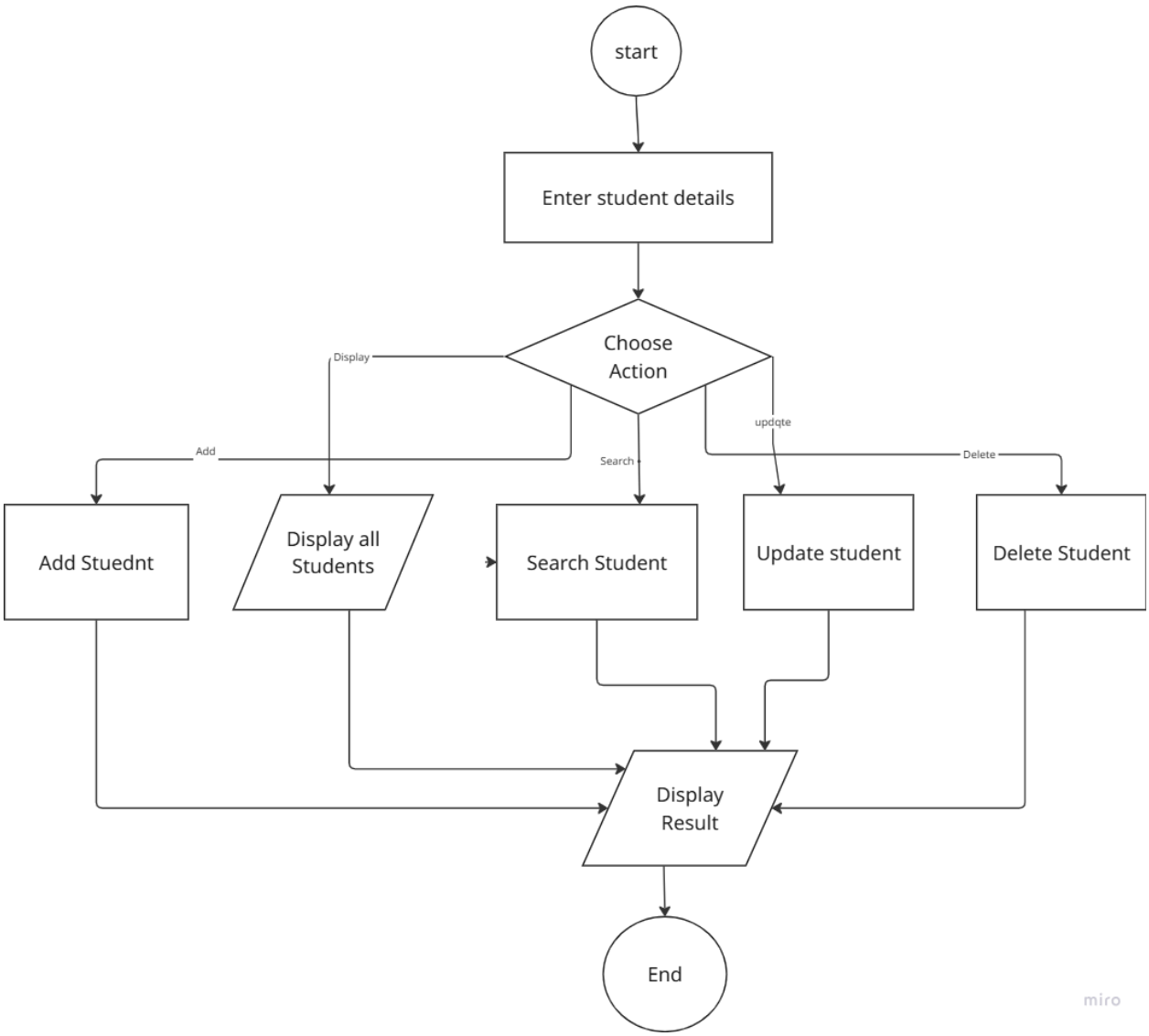
Figure 11:Checking Prime with prime value.

5. Program 4 – Mini Project: Student Management System

Objective

This mini project combined the concepts of file handling, inheritance, and event-driven programming to create a GUI-based Student Management System. The application allowed adding new students, searching by ID, displaying all students, and saving/retrieving data from files.

Flowchart



Code Implementation

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

// Student class
class Student {
    String name;
```

```
    String id;
    double marks;

    Student(String name, String id, double marks) {
        this.name = name;
        this.id = id;
        this.marks = marks;
    }

    @Override
    public String toString() {
        return name + "," + id + "," + marks;
    }

    static Student fromString(String line) {
        String[] parts = line.split(",");
        if (parts.length != 3) return null;
        try {
            return new Student(parts[0], parts[1], Double.parseDouble(parts[2]));
        } catch (NumberFormatException e) {
            return null;
        }
    }
}

// Main GUI class
public class StudentManagementSystem extends JFrame implements
ActionListener {

    private JTextField nameField, idField, marksField, searchField;
```

```
    private JButton addButton, displayButton, searchButton, updateButton,
deleteButton;
    private JTextArea outputArea;
    private static final String FILE_NAME = "students.txt";

    public StudentManagementSystem() {
        setTitle("Student Management System");
        setSize(550, 500);
        setLayout(new BorderLayout(10, 10));
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Input Panel
        JPanel inputPanel = new JPanel(new GridLayout(4, 2, 5, 5));
        inputPanel.setBorder(BorderFactory.createTitledBorder("Student Details"));

        inputPanel.add(new JLabel("Name:"));
        nameField = new JTextField();
        inputPanel.add(nameField);

        inputPanel.add(new JLabel("ID:"));
        idField = new JTextField();
        inputPanel.add(idField);

        inputPanel.add(new JLabel("Marks:"));
        marksField = new JTextField();
        inputPanel.add(marksField);

        addButton = new JButton("Add Student");
        addButton.addActionListener(this);
        inputPanel.add(addButton);
```

```
updateButton = new JButton("Update Student");
updateButton.addActionListener(this);
inputPanel.add(updateButton);

add(inputPanel, BorderLayout.NORTH);

// Output Area
outputArea = new JTextArea();
outputArea.setEditable(false);
JScrollPane scrollPane = new JScrollPane(outputArea);
scrollPane.setBorder(BorderFactory.createTitledBorder("Output"));
add(scrollPane, BorderLayout.CENTER);

// Action Panel
JPanel actionPanel = new JPanel(new GridLayout(3, 2, 5, 5));
actionPanel.setBorder(BorderFactory.createTitledBorder("Actions"));

displayButton = new JButton("Display All Students");
displayButton.addActionListener(this);
actionPanel.add(displayButton);

actionPanel.add(new JLabel("Search/Delete by ID:"));
searchField = new JTextField();
actionPanel.add(searchField);

searchButton = new JButton("Search");
searchButton.addActionListener(this);
actionPanel.add(searchButton);
```

```
        deleteButton = new JButton("Delete");
        deleteButton.addActionListener(this);
        actionPanel.add(deleteButton);

        add(actionPanel, BorderLayout.SOUTH);

        setLocationRelativeTo(null);
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == addButton) addStudent();
        else if (e.getSource() == displayButton) displayStudents();
        else if (e.getSource() == searchButton) searchStudent();
        else if (e.getSource() == updateButton) updateStudent();
        else if (e.getSource() == deleteButton) deleteStudent();
    }

    private void addStudent() {
        String name = nameField.getText().trim();
        String id = idField.getText().trim();
        String marksText = marksField.getText().trim();

        if (name.isEmpty() || id.isEmpty() || marksText.isEmpty()) {
            JOptionPane.showMessageDialog(this, "Fill all fields.", "Input Error",
JOptionPane.WARNING_MESSAGE);
            return;
        }
```

```
        try {
            double marks = Double.parseDouble(marksText);
            Student s = new Student(name, id, marks);

            try (BufferedWriter bw = new BufferedWriter(new
FileWriter(FILE_NAME, true))) {
                bw.write(s.toString());
                bw.newLine();
            }

            outputArea.setText("Student added successfully!\n");
            nameField.setText("");
            idField.setText("");
            marksField.setText("");

        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(this, "Marks must be a number.", "Input
Error", JOptionPane.ERROR_MESSAGE);
        } catch (IOException ex) {
            JOptionPane.showMessageDialog(this, "Error writing to file.", "File Error",
JOptionPane.ERROR_MESSAGE);
        }
    }

    private void displayStudents() {
        outputArea.setText("");
        try (BufferedReader br = new BufferedReader(new
FileReader(FILE_NAME))) {
            String line;
            while ((line = br.readLine()) != null) {
                Student s = Student.fromString(line);
```

```
            if (s != null) outputArea.append("Name: " + s.name + ", ID: " + s.id + ",
Marks: " + s.marks + "\n");
        }
    } catch (FileNotFoundException ex) {
        outputArea.setText("No student records found.\n");
    } catch (IOException ex) {
        outputArea.setText("Error reading file.\n");
    }
}

private void searchStudent() {
    String searchId = searchField.getText().trim();
    if (searchId.isEmpty()) {
        JOptionPane.showMessageDialog(this, "Enter an ID to search.", "Input
Error", JOptionPane.WARNING_MESSAGE);
        return;
    }

    boolean found = false;
    try (BufferedReader br = new BufferedReader(new
FileReader(FILE_NAME))) {
        String line;
        while ((line = br.readLine()) != null) {
            Student s = Student.fromString(line);
            if (s != null && s.id.equalsIgnoreCase(searchId)) {
                outputArea.setText("Student Found:\nName: " + s.name + "\nID: " +
s.id + "\nMarks: " + s.marks + "\n");
                found = true;
                break;
            }
        }
```

```
            if (!found) outputArea.setText("No student found with ID: " + searchId +
"\n");
        } catch (IOException ex) {
            outputArea.setText("Error reading file.\n");
        }
    }

    private void updateStudent() {
        String id = idField.getText().trim();
        String name = nameField.getText().trim();
        String marksText = marksField.getText().trim();

        if (id.isEmpty() || name.isEmpty() || marksText.isEmpty()) {
            JOptionPane.showMessageDialog(this, "Fill all fields for update.", "Input
Error", JOptionPane.WARNING_MESSAGE);
            return;
        }

        try {
            double marks = Double.parseDouble(marksText);
            java.util.List<Student> students = new ArrayList<>();
            boolean updated = false;

            try (BufferedReader br = new BufferedReader(new
FileReader(FILE_NAME))) {
                String line;
                while ((line = br.readLine()) != null) {
                    Student s = Student.fromString(line);
                    if (s != null) {
                        if (s.id.equalsIgnoreCase(id)) {
                            s.name = name;
```

```
                    s.marks = marks;
                    updated = true;
                }
                students.add(s);
            }
        }
    }

    try (BufferedWriter bw = new BufferedWriter(new
FileWriter(FILE_NAME))) {
        for (Student s : students) {
            bw.write(s.toString());
            bw.newLine();
        }
    }

    if (updated) outputArea.setText("Student updated successfully!\n");
    else outputArea.setText("No student found with ID: " + id + "\n");

  } catch (NumberFormatException ex) {
    JOptionPane.showMessageDialog(this, "Marks must be a number.", "Input
Error", JOptionPane.ERROR_MESSAGE);
  } catch (IOException ex) {
    outputArea.setText("Error updating file.\n");
  }
}

private void deleteStudent() {
  String id = searchField.getText().trim();
  if (id.isEmpty()) {
```

```
            JOptionPane.showMessageDialog(this, "Enter an ID to delete.", "Input
Error", JOptionPane.WARNING_MESSAGE);
            return;
        }

        try {
            java.util.List<Student> students = new ArrayList<>();
            boolean deleted = false;

            try (BufferedReader br = new BufferedReader(new
FileReader(FILE_NAME))) {
                String line;
                while ((line = br.readLine()) != null) {
                    Student s = Student.fromString(line);
                    if (s != null) {
                        if (s.id.equalsIgnoreCase(id)) {
                            deleted = true;
                            continue; // skip this student
                        }
                        students.add(s);
                    }
                }
            }

            try (BufferedWriter bw = new BufferedWriter(new
FileWriter(FILE_NAME))) {
                for (Student s : students) {
                    bw.write(s.toString());
                    bw.newLine();
                }
            }
```

```
            if (deleted) outputArea.setText("Student deleted successfully!\n");
            else outputArea.setText("No student found with ID: " + id + "\n");

        } catch (IOException ex) {
            outputArea.setText("Error updating file.\n");
        }
    }

    public static void main(String[] args) {
        new StudentManagementSystem();
    }
}
```

Explanation

The Student Management System is a Java GUI application developed using the Swing framework. It allows users to manage student records efficiently through a simple and interactive interface. The system provides five main functionalities — adding new students, displaying all records, searching for a student by ID, updating existing details, and deleting student records.

All student information, including name, ID, and marks, is stored in a text file for data persistence. The program uses object-oriented programming concepts such as classes, methods, and encapsulation. Event handling is implemented through action listeners, enabling the buttons to perform specific actions based on user interaction. Proper error handling ensures that invalid inputs or missing fields are managed gracefully, making the application user-friendly and reliable.

How to run

- Open **StudentManagementSystem.java** in your IDE.
- Run the program — a window opens.
- Use the buttons to perform actions:
  - **Add Student** – enter name, ID, marks.
  - **Display All** – shows all records.
  - **Search by ID** – retrieves a specific student.

- **Update Student** – modify marks.
- **Delete Student** – remove a record.

Evidence of Testing

Test Case No.	Feature Tested	Input / Action	Expected Output	Actual Output	Result
1	Add Student (Valid Input)	Name = "Sizan"ID = "11"Marks = 92	Student record added to file.Message: *"Student added successfully!"*	Record added successfully and shown in display.	✅ Pass
2	Add Student (Missing Input)	Name = ""ID = "10"Marks = 75	Message: *"Fill all fields."*	Error message displayed correctly.	✅ Pass
3	Display All Students	Click *Display All Students* button	All student records displayed in output area.	Records displayed correctly from file.	✅ Pass
4	Search Student (Existing ID)	ID = "11"	Displays details of "Sizan (11)".	Record found and displayed accurately.	✅ Pass
5	Search Student (Non-existing ID)	ID = "99"	Message: *"No student found with ID: 99"*	Message displayed correctly.	✅ Pass
6	Update Student	ID = "11"New Name = "Sizan Raut"Marks = 84	File updated with new details.Message: *"Student updated successfully!"*	Record updated successfully.	✅ Pass

Test Case No.	Feature Tested	Input / Action	Expected Output	Actual Output	Result
7	Delete Student	ID = "10"	Record removed from file.Message: *"Student deleted successfully!"*	Record deleted and message shown.	✅ Pass
8	File Not Found / Empty File	Delete students.txt before running	Program should show *"No student records found."*	Message displayed, no crash occurred.	✅ Pass

Sample Output

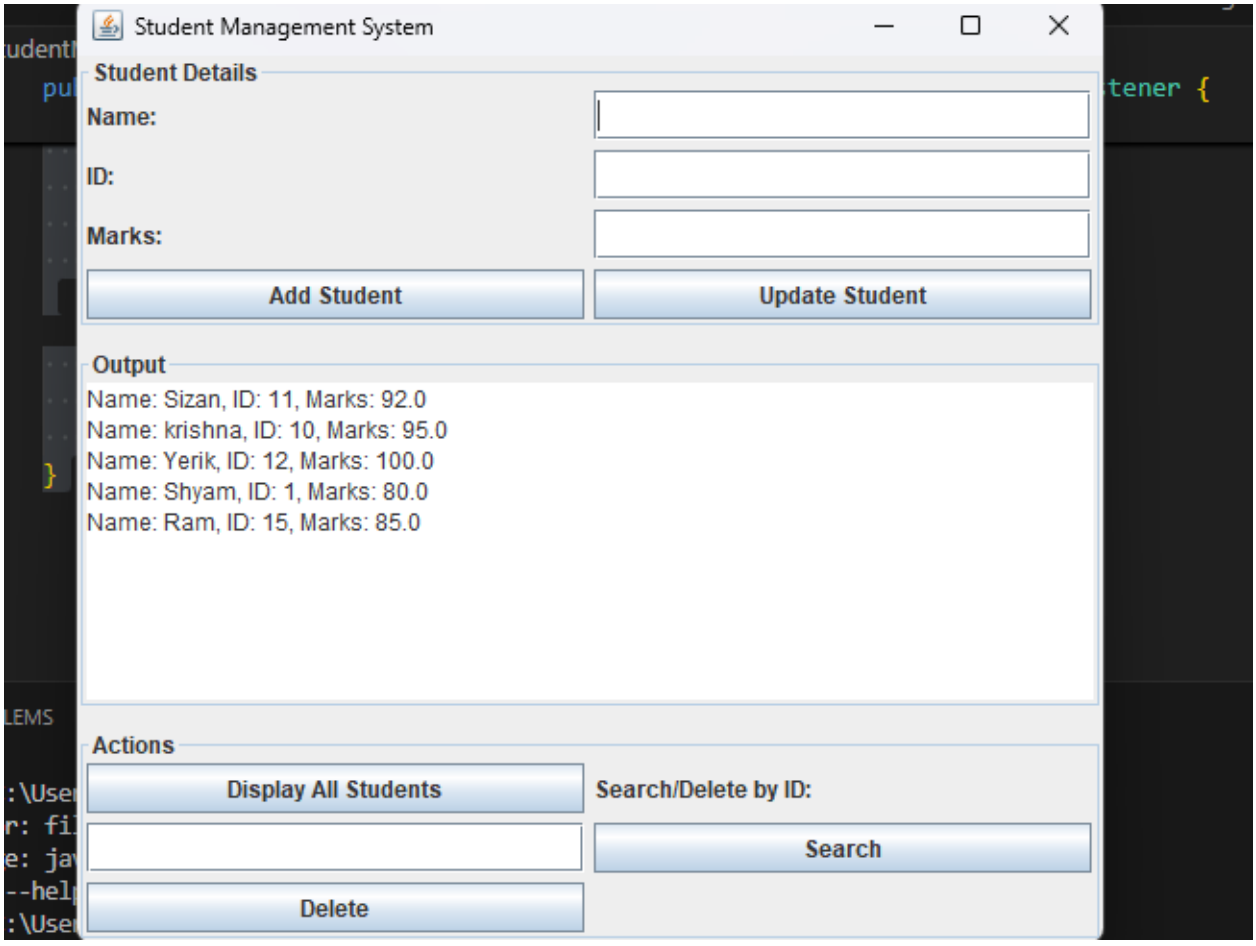


Figure 12:Displaying the existing record on student.txt file.

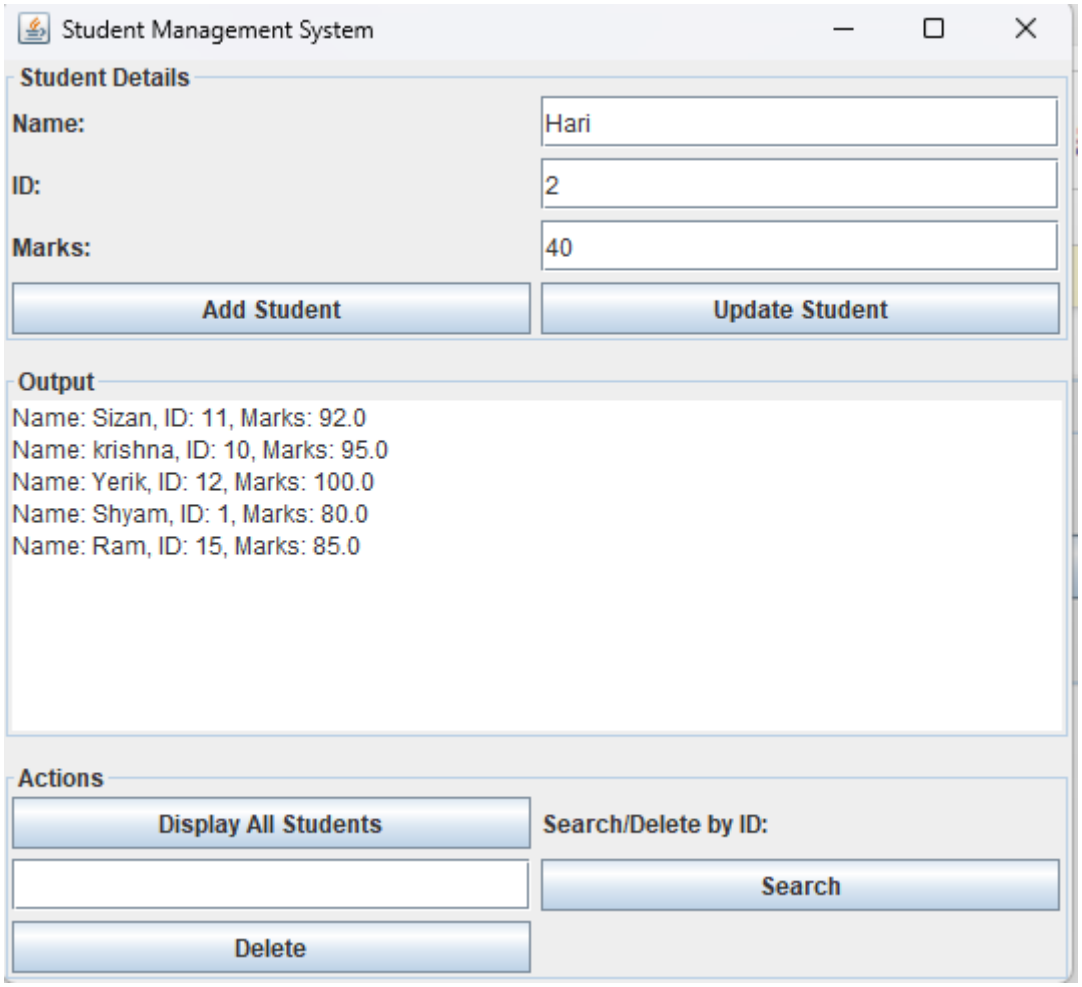


Figure 13:Adding new student Hari.

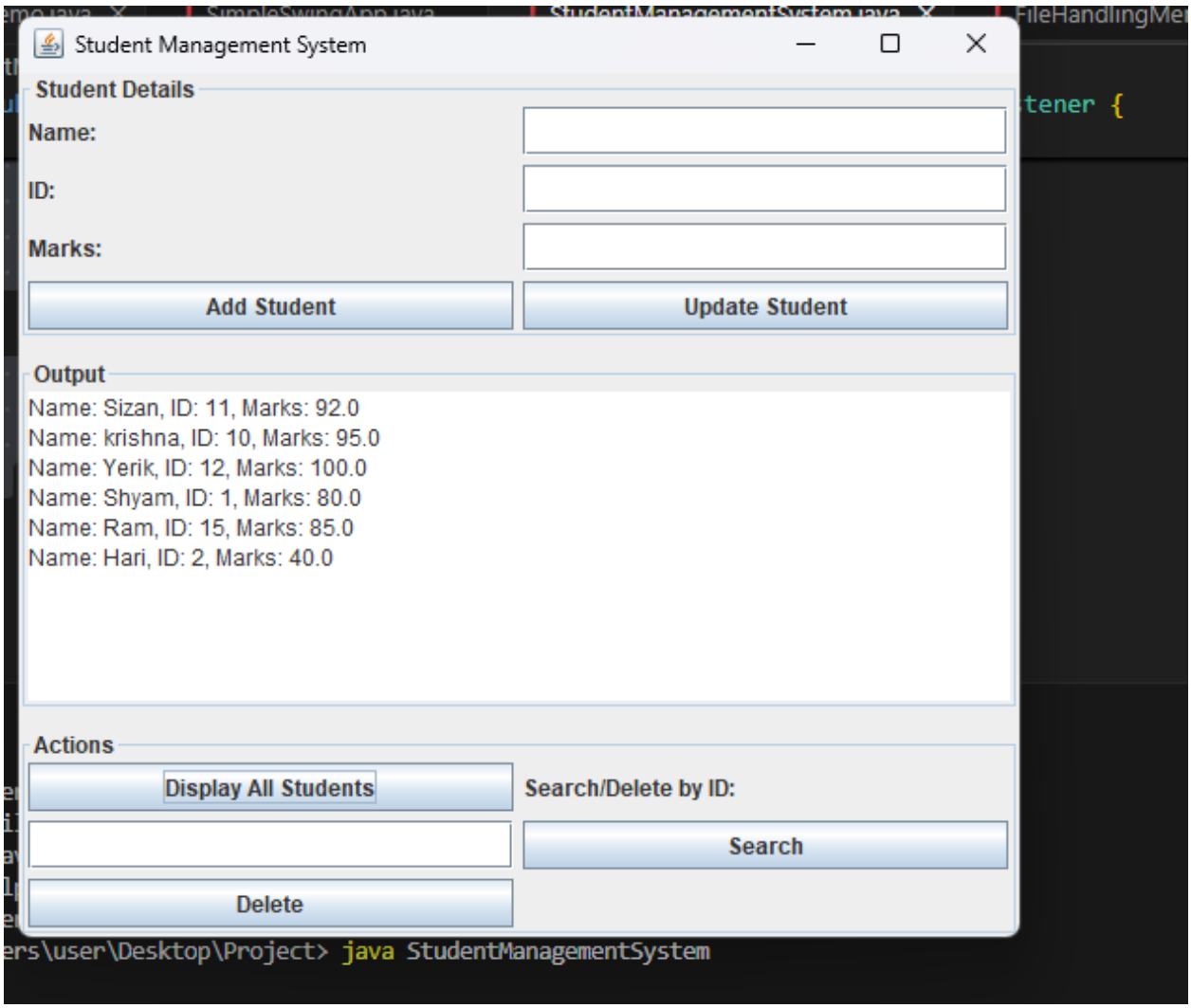


Figure 14:Displaying the record where new student is added.

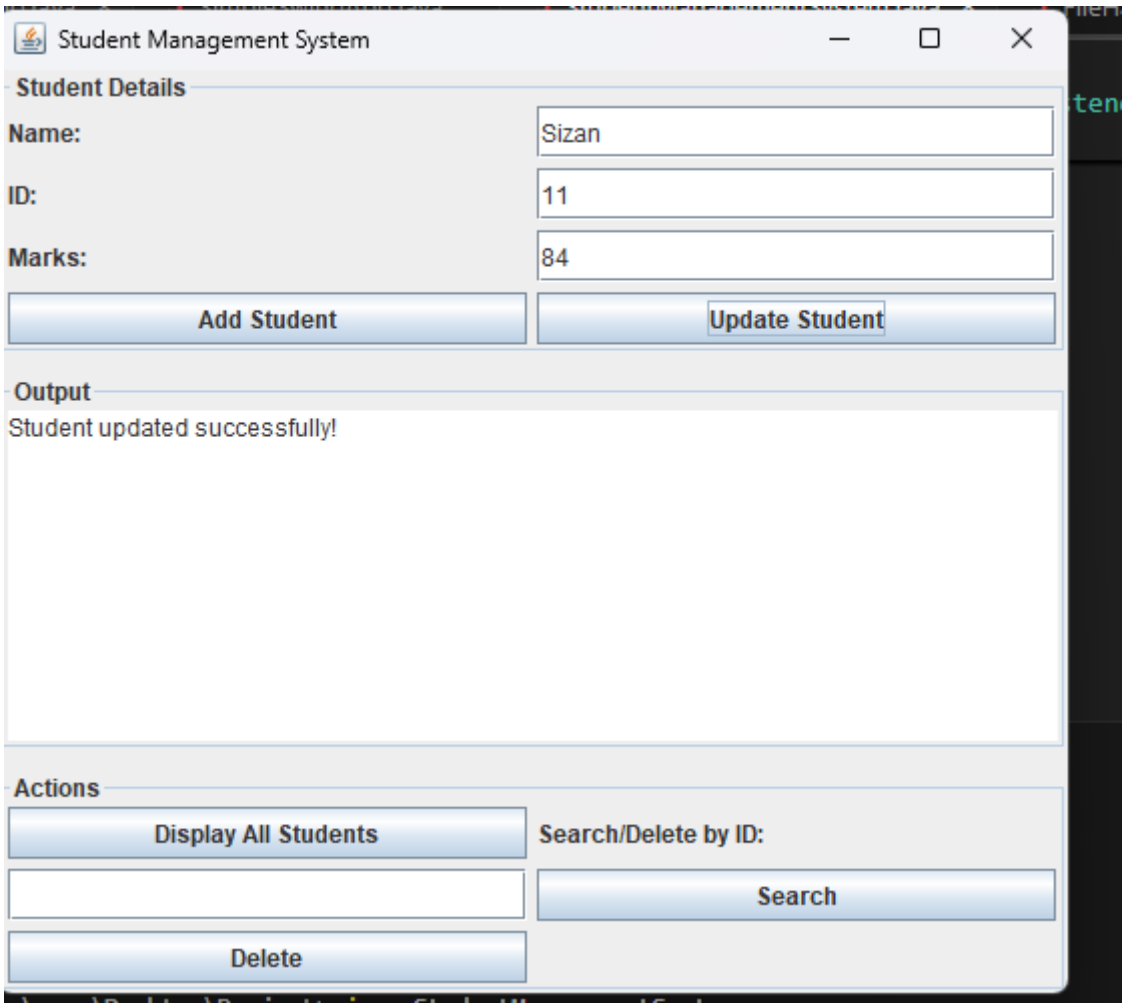


Figure 15:Updating marks of Sizan.

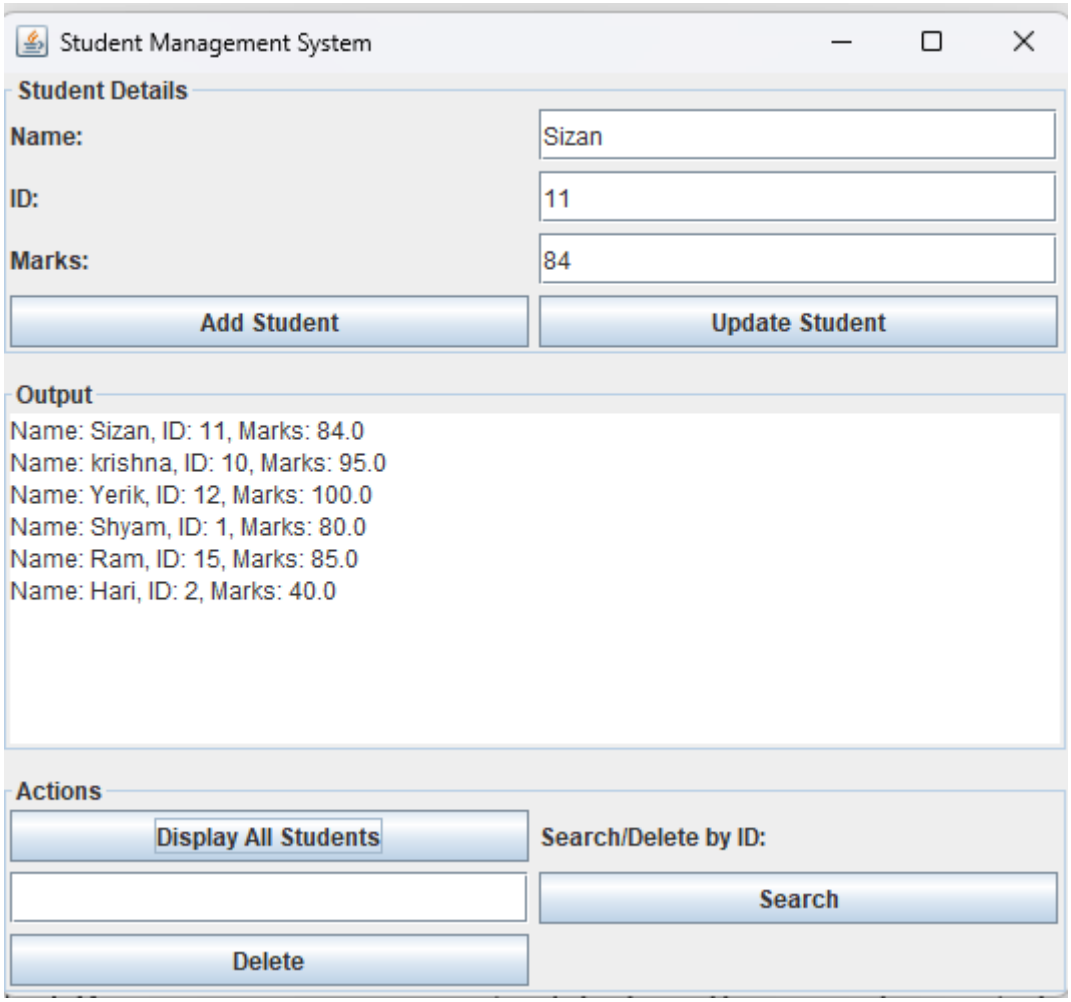


Figure 16:Displaying the updated record.

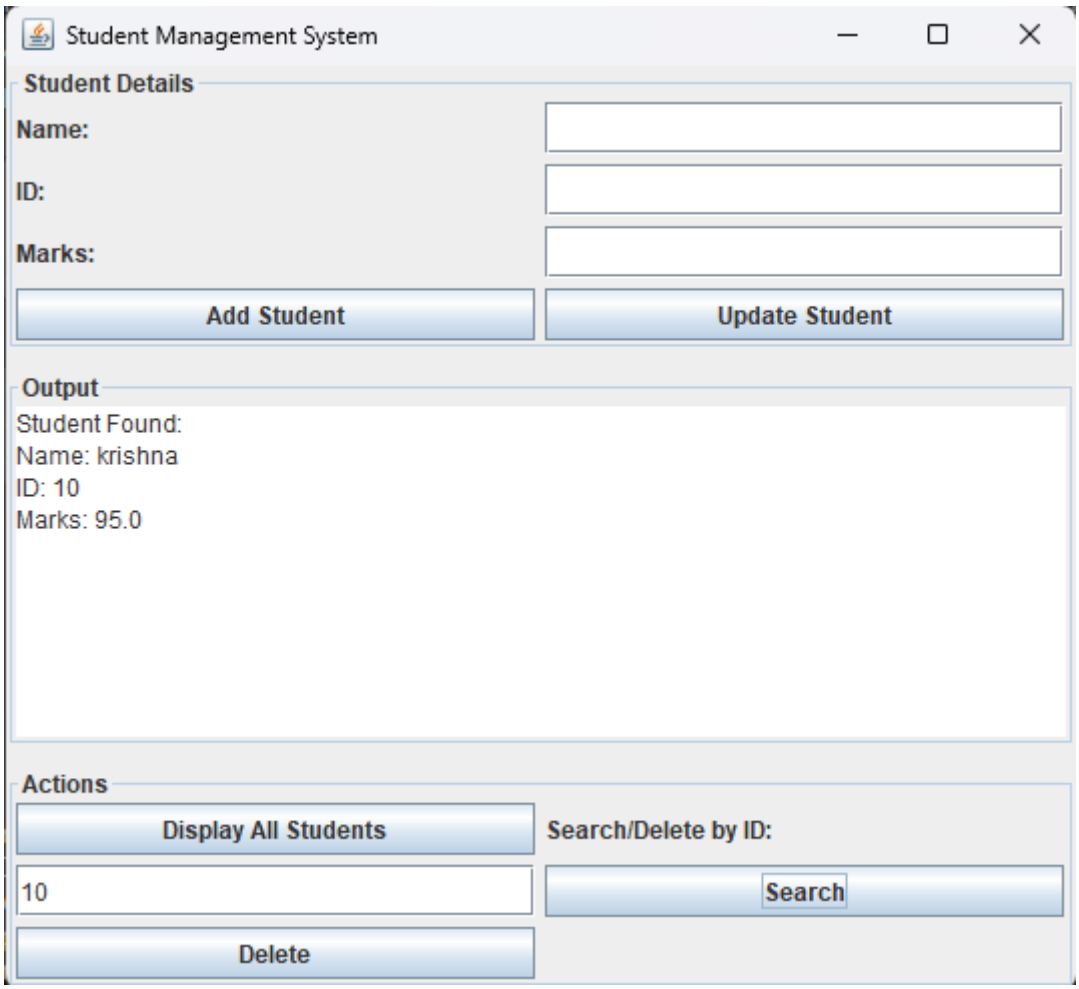


Figure 17:Searching and displaying Student record by ID.

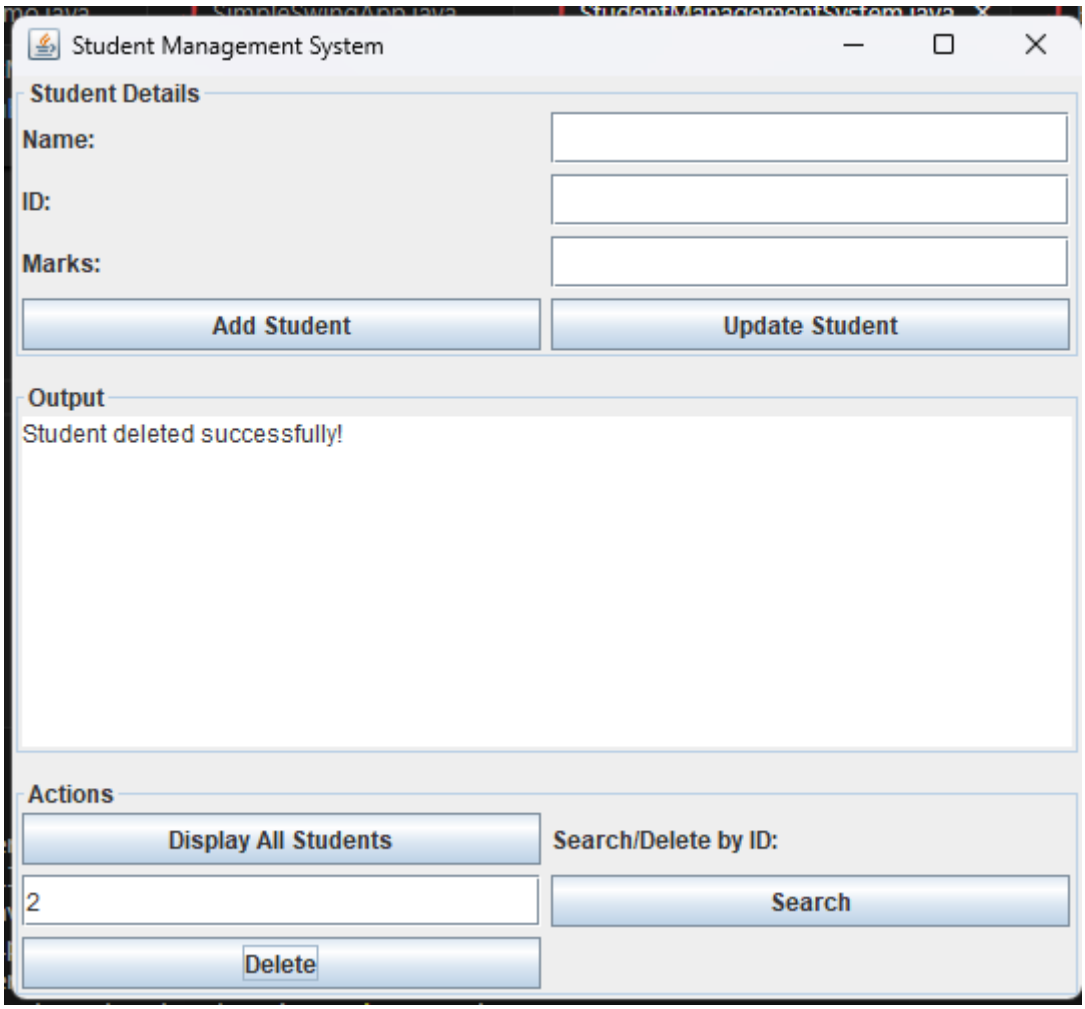


Figure 18:Deleting student record by ID.

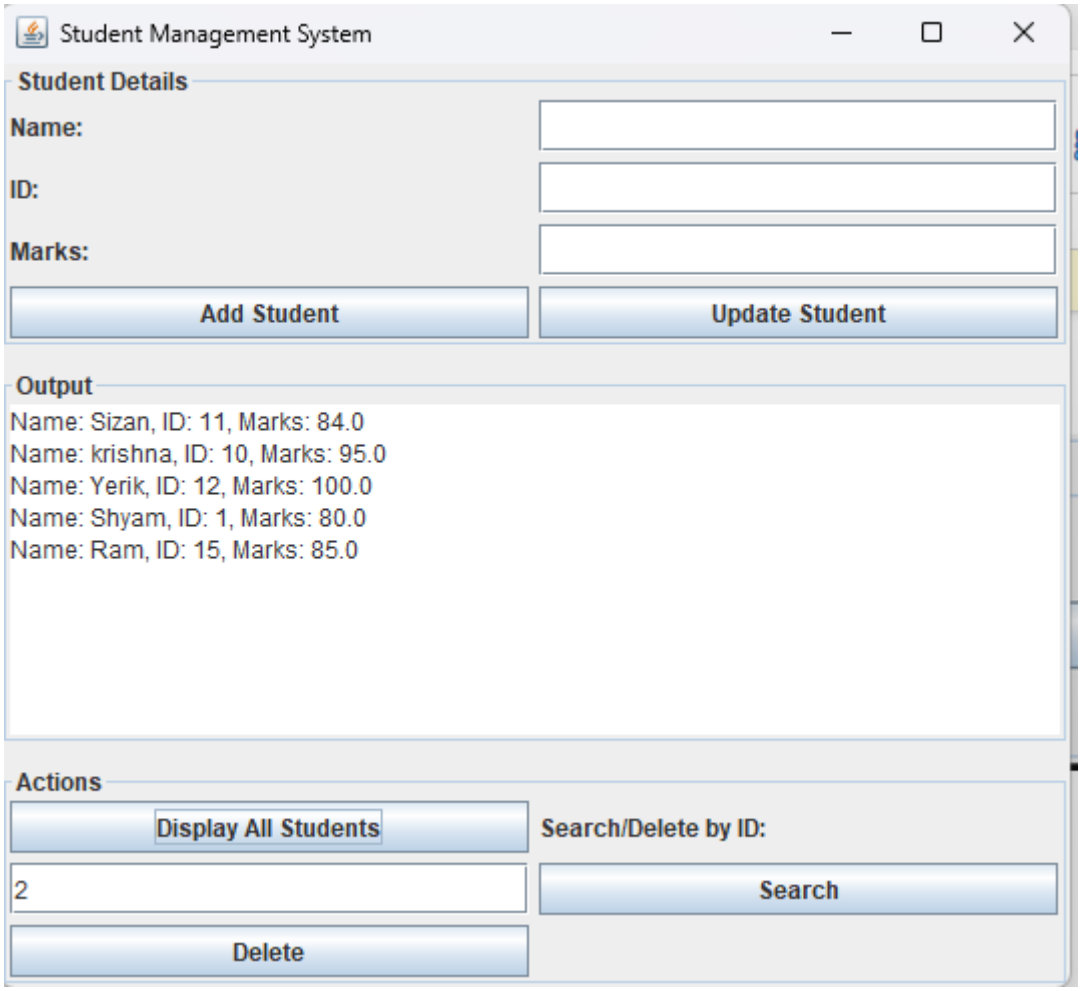


Figure 19:Displaying the record after deleting the student with ID 2.

6. Conclusion

The Java programming assignment successfully demonstrated the application of Object-Oriented Programming concepts, file handling, and event-driven GUI design. Through each program, principles such as inheritance, polymorphism, and encapsulation were applied effectively. The Student Management System integrated all the learned techniques into a cohesive and practical software application. Proper error handling, clean code structure, and user interaction were key focuses throughout development.

7. References

Oracle. (2025). Java SE Documentation. https://docs.oracle.com/en/java/

Java Swing Tutorial. (2025). https://www.javatpoint.com/java-swing