SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA 1st semester

PROGRAMMING USING JAVA (PCA20C01J)

Lab Manual

Lab 1: Learning to work with Java IDE and Writing Simple Conversion Programs

Title

Introduction to Java IDE and Basic Data Type Conversion

Aim

To familiarize with a Java Integrated Development Environment (IDE) like Eclipse or IntelliJ IDEA, and to write simple Java programs for basic data type conversions (e.g., Celsius to Fahrenheit, inches to centimeters).

Procedure

1. **IDE Setup:**
    - Install a Java Development Kit (JDK) if not already present.
    - Download and install a Java IDE (e.g., Eclipse, IntelliJ IDEA, VS Code with Java extensions).
    - Configure the IDE to recognize the installed JDK.
2. **Project Creation:**
    - Create a new Java project in the IDE.
    - Create a new Java class within the project (e.g., `TemperatureConverter.java` or `UnitConverter.java`).
3. **Code Implementation:**
    - Write the Java code to perform a simple conversion. For example, to convert Celsius to Fahrenheit:
        - Declare a variable for Celsius temperature.
        - Assign a value to the Celsius variable.
        - Apply the conversion formula: `Fahrenheit = (Celsius * 9/5) + 32`.
        - Print both the Celsius and Fahrenheit temperatures to the console.
    - For unit conversion (e.g., inches to centimeters):
        - Declare a variable for inches.
        - Assign a value to the inches variable.
        - Apply the conversion formula: `Centimeters = Inches * 2.54`.
        - Print both the inches and centimeters to the console.
4. **Compilation and Execution:**
    - Compile the Java code using the IDE's built-in compiler.
    - Run the program and observe the output in the console.

5. **Experimentation:**
    - Modify the input values and re-run the program to see how the output changes.
    - Try different simple conversion programs.

Source Code

```
// Example: Celsius to Fahrenheit Converter
public class TemperatureConverter {
    public static void main(String[] args) {
        // Declare a variable for Celsius temperature
        double celsius = 25.0; // Example Celsius temperature

        // Apply the conversion formula: Fahrenheit = (Celsius * 9/5) + 32
        double fahrenheit = (celsius * 9/5) + 32;

        // Print both the Celsius and Fahrenheit temperatures
        System.out.println("Celsius: " + celsius + "°C");
        System.out.println("Fahrenheit: " + fahrenheit + "°F");
    }
}
```

Input

No direct user input for this simple example, values are hardcoded. (For interactive programs, specify command-line arguments or values entered via `Scanner`.)

Expected Output

```
Celsius: 25.0°C
Fahrenheit: 77.0°F
```

Lab 2: Operators

Title

Understanding and Implementing Java Operators

Aim

To understand and implement various types of operators in Java, including arithmetic, relational, logical, assignment, increment/decrement, and conditional operators.

Procedure

1. **Project and Class Creation:**
    - Create a new Java project and a class (e.g., `OperatorExamples.java`).
2. **Arithmetic Operators:**
    - Declare two integer variables.
    - Perform addition, subtraction, multiplication, division, and modulus operations.
    - Print the results of each operation.
3. **Relational Operators:**
    - Declare two integer variables.
    - Use ==, `!=`, <, >, <=, >= to compare them.
    - Print the boolean results of each comparison.
4. **Logical Operators:**
    - Declare two boolean variables or expressions.
    - Use `&&` (AND), `||` (OR), `!` (NOT) operators.
    - Print the results.
5. **Assignment Operators:**
    - Declare an integer variable.
    - Use =, +=, -=, `*= /=`, `%=` to modify its value.
    - Print the variable's value after each operation.
6. **Increment/Decrement Operators:**
    - Declare an integer variable.
    - Demonstrate pre-increment (`++var`), post-increment (`var++`), pre-decrement (`--var`), and post-decrement (`var--`) and observe their effects.
    - Print the variable's value before and after each operation.
7. **Conditional (Ternary) Operator:**
    - Use the `? :` operator to assign a value based on a condition.
    - Print the result.
8. **Compilation and Execution:**
    - Compile and run the program.

Source Code

```
// Example: Operator Examples
public class OperatorExamples {
    public static void main(String[] args) {
        int a = 10;
        int b = 5;
        boolean x = true;
        boolean y = false;

        // Arithmetic Operators
```

```
        System.out.println("--- Arithmetic Operators ---");
        System.out.println("a + b = " + (a + b));
        System.out.println("a - b = " + (a - b));
        System.out.println("a * b = " + (a * b));
        System.out.println("a / b = " + (a / b));
        System.out.println("a % b = " + (a % b));

        // Relational Operators
        System.out.println("\n--- Relational Operators ---");
        System.out.println("a == b : " + (a == b));
        System.out.println("a != b : " + (a != b));
        System.out.println("a > b : " + (a > b));
        System.out.println("a < b : " + (a < b));
        System.out.println("a >= b : " + (a >= b));
        System.out.println("a <= b : " + (a <= b));

        // Logical Operators
        System.out.println("\n--- Logical Operators ---");
        System.out.println("x && y : " + (x && y));
        System.out.println("x || y : " + (x || y));
        System.out.println("!x : " + (!x));

        // Assignment Operators
        System.out.println("\n--- Assignment Operators ---");
        int c = a;
        System.out.println("c = a : " + c);
        c += b; // c = c + b
        System.out.println("c += b : " + c);
        c -= b; // c = c - b
        System.out.println("c -= b : " + c);

        // Increment/Decrement Operators
        System.out.println("\n--- Increment/Decrement Operators ---");
        int i = 5;
        System.out.println("Initial i: " + i);
        System.out.println("++i : " + (++i)); // pre-increment
        System.out.println("i after pre-increment: " + i);
        System.out.println("i++ : " + (i++)); // post-increment
        System.out.println("i after post-increment: " + i);

        // Conditional (Ternary) Operator
        System.out.println("\n--- Conditional Operator ---");
        String result = (a > b) ? "a is greater" : "b is greater";
        System.out.println("Result of (a > b) ? 'a is greater' : 'b is greater'
: " + result);
    }
}
```

Input

No direct user input for this example.

Expected Output

```
--- Arithmetic Operators ---
a + b = 15
a - b = 5
a * b = 50
a / b = 2
a % b = 0

--- Relational Operators ---
a == b : false
a != b : true
```

```
a > b : true
a < b : false
a >= b : true
a <= b : false

--- Logical Operators ---
x && y : false
x || y : true
!x : false

--- Assignment Operators ---
c = a : 10
c += b : 15
c -= b : 10

--- Increment/Decrement Operators ---
Initial i: 5
++i : 6
i after pre-increment: 6
i++ : 6
i after post-increment: 7

--- Conditional Operator ---
Result of (a > b) ? 'a is greater' : 'b is greater' : a is greater
```

Lab 3: Arrays, Control Statements

Title

Working with Arrays and Control Flow Statements in Java

Aim

To understand and implement one-dimensional and multi-dimensional arrays, and to use various control statements (if-else, switch, for, while, do-while, break, continue) in Java programs.

Procedure

1. **Project and Class Creation:**
    - Create a new Java project and a class (e.g., `ArrayControlExamples.java`).
2. **One-Dimensional Arrays:**
    - Declare and initialize a one-dimensional integer array.
    - Access elements using index.
    - Iterate through the array using a `for` loop and an enhanced `for` loop (for-each).
    - Calculate sum/average of array elements.
3. **Multi-Dimensional Arrays:**
    - Declare and initialize a two-dimensional integer array (matrix).
    - Access elements using row and column indices.
    - Iterate through the 2D array using nested `for` loops.
4. **`if-else` Statement:**
    - Write a program to check if a number is positive, negative, or zero.
    - Demonstrate `if-else if-else` ladder.
5. **`switch` Statement:**
    - Write a program that takes a day number (1-7) and prints the corresponding day name.
6. **Looping Constructs (`for`, `while`, `do-while`):**
    - **`for` loop:** Print numbers from 1 to 10.
    - **`while` loop:** Print numbers from 1 to 5.
    - **`do-while` loop:** Print numbers from 1 to 3 (guaranteeing at least one execution).
7. **`break` and `continue` Statements:**
    - Demonstrate `break` to exit a loop prematurely.
    - Demonstrate `continue` to skip the current iteration of a loop.
8. **Compilation and Execution:**
    - Compile and run the program.

Source Code

```
// Example: Arrays and Control Statements
public class ArrayControlExamples {
    public static void main(String[] args) {
        // --- One-Dimensional Array ---
        System.out.println("--- One-Dimensional Array ---");
        int[] numbers = {10, 20, 30, 40, 50};
        System.out.println("Elements of numbers array:");
        for (int i = 0; i < numbers.length; i++) {
            System.out.println("numbers[" + i + "] = " + numbers[i]);
        }
        int sum = 0;
```

```
for (int num : numbers) { // Enhanced for loop
    sum += num;
}
System.out.println("Sum of array elements: " + sum);

// --- Multi-Dimensional Array (2D Array) ---
System.out.println("\n--- Multi-Dimensional Array ---");
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
System.out.println("Elements of matrix:");
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}

// --- if-else Statement ---
System.out.println("\n--- if-else Statement ---");
int num = -7;
if (num > 0) {
    System.out.println(num + " is positive.");
} else if (num < 0) {
    System.out.println(num + " is negative.");
} else {
    System.out.println(num + " is zero.");
}

// --- switch Statement ---
System.out.println("\n--- switch Statement ---");
int day = 3;
switch (day) {
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    case 3: System.out.println("Wednesday"); break;
    case 4: System.out.println("Thursday"); break;
    case 5: System.out.println("Friday"); break;
    case 6: System.out.println("Saturday"); break;
    case 7: System.out.println("Sunday"); break;
    default: System.out.println("Invalid day");
}

// --- for loop ---
System.out.println("\n--- for loop ---");
for (int i = 1; i <= 5; i++) {
    System.out.print(i + " ");
}
System.out.println();

// --- while loop ---
System.out.println("\n--- while loop ---");
int count = 1;
while (count <= 5) {
    System.out.print(count + " ");
    count++;
}
System.out.println();

// --- do-while loop ---
System.out.println("\n--- do-while loop ---");
int doWhileCount = 1;
do {
    System.out.print(doWhileCount + " ");
    doWhileCount++;
```

```
        } while (doWhileCount <= 3);
        System.out.println();

        // --- break statement ---
        System.out.println("\n--- break statement ---");
        for (int i = 1; i <= 10; i++) {
            if (i == 6) {
                break; // Exit the loop when i is 6
            }
            System.out.print(i + " ");
        }
        System.out.println();

        // --- continue statement ---
        System.out.println("\n--- continue statement ---");
        for (int i = 1; i <= 5; i++) {
            if (i == 3) {
                continue; // Skip current iteration when i is 3
            }
            System.out.print(i + " ");
        }
        System.out.println();
    }
}
```

Input

No direct user input for this example.

Expected Output

```
--- One-Dimensional Array ---
Elements of numbers array:
numbers[0] = 10
numbers[1] = 20
numbers[2] = 30
numbers[3] = 40
numbers[4] = 50
Sum of array elements: 150

--- Multi-Dimensional Array ---
Elements of matrix:
1 2 3
4 5 6
7 8 9

--- if-else Statement ---
-7 is negative.

--- switch Statement ---
Wednesday

--- for loop ---
1 2 3 4 5

--- while loop ---
1 2 3 4 5

--- do-while loop ---
1 2 3

--- break statement ---
1 2 3 4 5
```

```
--- continue statement ---
1 2 4 5
```

Lab 4: Classes and Objects

Title

Implementing Classes and Objects in Java

Aim

To understand the concepts of classes and objects in Java, and to create and use custom classes with attributes (fields) and behaviors (methods).

Procedure

1. **Project and Class Creation:**
    - Create a new Java project.
    - Create a class representing a real-world entity (e.g., `Car`, `Student`, `Book`). This will be your blueprint.
2. **Define Attributes (Fields):**
    - Inside the class, declare instance variables to represent the attributes of the entity (e.g., for `Car`: `make`, `model`, `year`, `speed`).
3. **Define Behaviors (Methods):**
    - Inside the class, define methods to represent the actions or behaviors of the entity (e.g., for `Car`: `startEngine()`, `accelerate()`, `brake()`, `displayInfo()`).
4. **Constructor:**
    - Create a constructor for the class to initialize the object's attributes when it's created.
5. **Main Method (in a separate class or the same class):**
    - Create another class (e.g., `CarDemo.java`) with a `main` method.
    - Inside the `main` method, create multiple objects (instances) of your custom class using the `new` keyword and the constructor.
    - Access the attributes of these objects using the dot operator (.).
    - Call the methods of these objects to perform actions.
6. **Compilation and Execution:**
    - Compile and run the program.

Source Code

```
// Example: Car Class
class Car {
    // Attributes (fields)
    String make;
    String model;
    int year;
    double speed;

    // Constructor
    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.speed = 0.0; // Initial speed
    }

    // Methods (behaviors)
    public void startEngine() {
```

```
        System.out.println(make + " " + model + " engine started.");
    }

    public void accelerate(double increment) {
        this.speed += increment;
        System.out.println(make + " " + model + " accelerating. Current speed: "
+ speed + " km/h");
    }

    public void brake(double decrement) {
        if (this.speed - decrement >= 0) {
            this.speed -= decrement;
        } else {
            this.speed = 0;
        }
        System.out.println(make + " " + model + " braking. Current speed: " +
speed + " km/h");
    }

    public void displayInfo() {
        System.out.println("Car Info: " + year + " " + make + " " + model + ",
Current Speed: " + speed + " km/h");
    }
}

// Example: CarDemo Class with main method
public class CarDemo {
    public static void main(String[] args) {
        // Create objects of the Car class
        Car myCar = new Car("Toyota", "Camry", 2020);
        Car yourCar = new Car("Honda", "Civic", 2022);

        // Access attributes and call methods for myCar
        System.out.println("--- My Car Actions ---");
        myCar.displayInfo();
        myCar.startEngine();
        myCar.accelerate(50);
        myCar.accelerate(20);
        myCar.brake(30);
        myCar.displayInfo();

        System.out.println("\n--- Your Car Actions ---");
        yourCar.displayInfo();
        yourCar.startEngine();
        yourCar.accelerate(60);
        yourCar.brake(40);
        yourCar.displayInfo();
    }
}
```

Input

No direct user input for this example.

Expected Output

```
--- My Car Actions ---
Car Info: 2020 Toyota Camry, Current Speed: 0.0 km/h
Toyota Camry engine started.
Toyota Camry accelerating. Current speed: 50.0 km/h
Toyota Camry accelerating. Current speed: 70.0 km/h
Toyota Camry braking. Current speed: 40.0 km/h
Car Info: 2020 Toyota Camry, Current Speed: 40.0 km/h
```

```
--- Your Car Actions ---
Car Info: 2022 Honda Civic, Current Speed: 0.0 km/h
Honda Civic engine started.
Honda Civic accelerating. Current speed: 60.0 km/h
Honda Civic braking. Current speed: 20.0 km/h
Car Info: 2022 Honda Civic, Current Speed: 20.0 km/h
```

Lab 5: Overloading Methods and Constructors, finalize() method

Title

Method and Constructor Overloading, and `finalize()` Method in Java

Aim

To understand and implement method overloading and constructor overloading, and to explore the usage of the `finalize()` method for object cleanup in Java.

Procedure

1. **Project and Class Creation:**
    - Create a new Java project and a class (e.g., `OverloadingAndFinalize.java`).
2. **Method Overloading:**
    - Define a method (e.g., `add`) that performs addition.
    - Create multiple versions of this `add` method with the same name but different parameter lists (different number of arguments, different data types of arguments, or different order of arguments).
    - Call each overloaded method with appropriate arguments.
3. **Constructor Overloading:**
    - Define a class (e.g., `Box`).
    - Create multiple constructors for the `Box` class, each with a different set of parameters (e.g., one default constructor, one with dimensions, one with dimensions and color).
    - Create objects of the `Box` class using different constructors.
4. **`finalize()` Method:**
    - Override the `finalize()` method in a class.
    - Inside `finalize()`, add a print statement to indicate when an object is being garbage collected.
    - In the `main` method, create an object of this class, set its reference to `null`, and explicitly call `System.gc()` to suggest garbage collection (note: `finalize()` is not guaranteed to be called immediately or at all).
5. **Compilation and Execution:**
    - Compile and run the program.

Source Code

```
// Example: Method and Constructor Overloading, finalize()
class Calculator {
    // Method Overloading: add method
    public int add(int a, int b) {
        System.out.println("Adding two integers:");
        return a + b;
    }

    public double add(double a, double b) {
        System.out.println("Adding two doubles:");
        return a + b;
    }

    public int add(int a, int b, int c) {
```

```
        System.out.println("Adding three integers:");
        return a + b + c;
    }
}

class Box {
    double width;
    double height;
    double depth;
    String color;

    // Constructor Overloading: Default constructor
    public Box() {
        System.out.println("Default constructor called.");
        width = -1; // Indicate uninitialized
        height = -1;
        depth = -1;
        color = "No Color";
    }

    // Constructor Overloading: Constructor with dimensions
    public Box(double w, double h, double d) {
        System.out.println("Constructor with dimensions called.");
        width = w;
        height = h;
        depth = d;
        color = "No Color";
    }

    // Constructor Overloading: Constructor with dimensions and color
    public Box(double w, double h, double d, String c) {
        System.out.println("Constructor with dimensions and color called.");
        width = w;
        height = h;
        depth = d;
        color = c;
    }

    public void displayVolume() {
        System.out.println("Volume: " + (width * height * depth) + ", Color: " +
color);
    }

    // finalize() method
    @Override
    protected void finalize() throws Throwable {
        System.out.println("Box object is being garbage collected.");
        super.finalize(); // Call superclass finalize
    }
}

public class OverloadingAndFinalize {
    public static void main(String[] args) {
        // --- Method Overloading Demo ---
        System.out.println("--- Method Overloading ---");
        Calculator calc = new Calculator();
        System.out.println("Sum of 5 and 10: " + calc.add(5, 10));
        System.out.println("Sum of 5.5 and 10.5: " + calc.add(5.5, 10.5));
        System.out.println("Sum of 1, 2, and 3: " + calc.add(1, 2, 3));

        // --- Constructor Overloading Demo ---
        System.out.println("\n--- Constructor Overloading ---");
        Box box1 = new Box(); // Calls default constructor
        box1.displayVolume();

        Box box2 = new Box(10, 20, 15); // Calls constructor with dimensions
        box2.displayVolume();
```

```
        Box box3 = new Box(5, 5, 5, "Red"); // Calls constructor with dimensions
and color
        box3.displayVolume();

        // --- finalize() method Demo ---
        System.out.println("\n--- finalize() method ---");
        Box tempBox = new Box(1, 1, 1, "Temporary");
        System.out.println("Temporary Box created.");
        tempBox = null; // Dereference the object
        System.out.println("Suggesting garbage collection...");
        System.gc(); // Request garbage collection (not guaranteed to run
immediately)

        // Keep main thread alive for a moment to allow GC to run (for
demonstration)
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("End of program.");
    }
}
```

Input

No direct user input for this example.

Expected Output

```
--- Method Overloading ---
Adding two integers:
Sum of 5 and 10: 15
Adding two doubles:
Sum of 5.5 and 10.5: 16.0
Adding three integers:
Sum of 1, 2, and 3: 6

--- Constructor Overloading ---
Default constructor called.
Volume: -1.0, Color: No Color
Constructor with dimensions called.
Volume: 3000.0, Color: No Color
Constructor with dimensions and color called.
Volume: 125.0, Color: Red

--- finalize() method ---
Constructor with dimensions and color called.
Temporary Box created.
Suggesting garbage collection...
Box object is being garbage collected.
End of program.
```

(Note: The output for `finalize()` might appear at a different time or not at all, as garbage collection is managed by the JVM and `System.gc()` is merely a suggestion.)

Lab 6: String Class, Command Line Arguments

Title

Working with the `String` Class and Command Line Arguments in Java

Aim

To explore the various methods of the `String` class for string manipulation and to understand how to pass and process command-line arguments in Java programs.

Procedure

1. **Project and Class Creation:**
    - Create a new Java project and a class (e.g., `StringAndArgs.java`).
2. **`String` Class Methods:**
    - Declare several `String` variables.
    - Demonstrate common `String` methods:
        - `length()`: Get string length.
        - `charAt(index)`: Get character at a specific index.
        - `substring(startIndex)`, `substring(startIndex, endIndex)`: Extract substrings.
        - `concat(String str)` or `+` operator: Concatenate strings.
        - `equals(Object obj)`, `equalsIgnoreCase(String anotherString)`: Compare strings.
        - `indexOf(char ch)`, `indexOf(String str)`: Find index of character/substring.
        - `toUpperCase()`, `toLowerCase()`: Change case.
        - `trim()`: Remove leading/trailing whitespace.
        - `replace(char oldChar, char newChar)`, `replace(CharSequence target, CharSequence replacement)`: Replace characters/substrings.
        - `startsWith(String prefix)`, `endsWith(String suffix)`: Check prefixes/suffixes.
    - Print the results of each operation.
3. **Command Line Arguments:**
    - In the `main` method, access the `args` array.
    - Check the number of arguments passed.
    - Iterate through the `args` array and print each argument.
    - Convert command-line arguments from `String` to numeric types (e.g., `Integer.parseInt()`, `Double.parseDouble()`) and perform an operation (e.g., sum two numbers passed as arguments).
4. **Compilation and Execution:**
    - Compile the program.
    - Run the program from the command line, passing various arguments.

Source Code

```
// Example: String Class and Command Line Arguments
public class StringAndArgs {
    public static void main(String[] args) {
        // --- String Class Methods ---
        System.out.println("--- String Class Methods ---");
```

```
        String s1 = "Hello Java";
        String s2 = "hello java";
        String s3 = "  Trim Me  ";

        System.out.println("s1: \"" + s1 + "\"");
        System.out.println("Length of s1: " + s1.length());
        System.out.println("Character at index 6 in s1: " + s1.charAt(6));
        System.out.println("Substring of s1 from index 6: \"" + s1.substring(6)
+ "\"");
        System.out.println("Substring of s1 from 0 to 5: \"" + s1.substring(0,
5) + "\"");
        System.out.println("s1 concatenated with \" World\": \"" + s1.concat("
World") + "\"");
        System.out.println("s1 equals \"Hello Java\": " + s1.equals("Hello
Java"));
        System.out.println("s1 equals s2 (case-sensitive): " + s1.equals(s2));
        System.out.println("s1 equals s2 (case-insensitive): " +
s1.equalsIgnoreCase(s2));
        System.out.println("Index of 'J' in s1: " + s1.indexOf('J'));
        System.out.println("Index of \"Java\" in s1: " + s1.indexOf("Java"));
        System.out.println("s1 in uppercase: \"" + s1.toUpperCase() + "\"");
        System.out.println("s1 in lowercase: \"" + s1.toLowerCase() + "\"");
        System.out.println("Trimmed s3: \"" + s3.trim() + "\"");
        System.out.println("s1 with 'a' replaced by '*': \"" + s1.replace('a',
'*') + "\"");
        System.out.println("s1 starts with \"Hello\": " +
s1.startsWith("Hello"));
        System.out.println("s1 ends with \"Java\": " + s1.endsWith("Java"));

        // --- Command Line Arguments ---
        System.out.println("\n--- Command Line Arguments ---");
        if (args.length == 0) {
            System.out.println("No command line arguments provided.");
        } else {
            System.out.println("Number of arguments: " + args.length);
            System.out.println("Arguments received:");
            for (int i = 0; i < args.length; i++) {
                System.out.println("args[" + i + "] = \"" + args[i] + "\"");
            }

            // Example: Summing two numbers from command line arguments
            if (args.length >= 2) {
                try {
                    int num1 = Integer.parseInt(args[0]);
                    int num2 = Integer.parseInt(args[1]);
                    int sum = num1 + num2;
                    System.out.println("Sum of first two arguments (" + num1 + "
+ " + num2 + ") = " + sum);
                } catch (NumberFormatException e) {
                    System.out.println("Error: First two arguments must be
integers for summation.");
                }
            }
        }
    }
}
```

Input

To test command-line arguments, compile the code and run it from your terminal:

Scenario 1: No arguments

```
java StringAndArgs
```

Scenario 2: With arguments

```
java StringAndArgs arg1 10 20 "another argument"
```

Expected Output

Scenario 1: No arguments

```
--- String Class Methods ---
s1: "Hello Java"
Length of s1: 10
Character at index 6 in s1: J
Substring of s1 from index 6: "Java"
Substring of s1 from 0 to 5: "Hello"
s1 concatenated with " World": "Hello Java World"
s1 equals "Hello Java": true
s1 equals s2 (case-sensitive): false
s1 equals s2 (case-insensitive): true
Index of 'J' in s1: 6
Index of "Java" in s1: 6
s1 in uppercase: "HELLO JAVA"
s1 in lowercase: "hello java"
Trimmed s3: "Trim Me"
s1 with 'a' replaced by '*': "Hello J*v*"
s1 starts with "Hello": true
s1 ends with "Java": true

--- Command Line Arguments ---
No command line arguments provided.
```

Scenario 2: With arguments

```
--- String Class Methods ---
s1: "Hello Java"
Length of s1: 10
Character at index 6 in s1: J
Substring of s1 from index 6: "Java"
Substring of s1 from 0 to 5: "Hello"
s1 concatenated with " World": "Hello Java World"
s1 equals "Hello Java": true
s1 equals s2 (case-sensitive): false
s1 equals s2 (case-insensitive): true
Index of 'J' in s1: 6
Index of "Java" in s1: 6
s1 in uppercase: "HELLO JAVA"
s1 in lowercase: "hello java"
Trimmed s3: "Trim Me"
s1 with 'a' replaced by '*': "Hello J*v*"
s1 starts with "Hello": true
s1 ends with "Java": true

--- Command Line Arguments ---
Number of arguments: 4
Arguments received:
args[0] = "arg1"
args[1] = "10"
args[2] = "20"
args[3] = "another argument"
Error: First two arguments must be integers for summation.
```

(If you run `java StringAndArgs 10 20`*, the output for summation would be* `Sum of first two arguments (10 + 20) = 30`*)*

Lab 7: Inheritance, Method Overriding, Abstract classes and methods

Title

Implementing Inheritance, Method Overriding, Abstract Classes and Methods in Java

Aim

To understand and implement key object-oriented programming concepts: inheritance (single, multilevel), method overriding, and the use of abstract classes and methods to achieve polymorphism and enforce contract.

Procedure

1. **Project and Class Creation:**
    - Create a new Java project.
2. **Inheritance (Single):**
    - Create a `Parent` class with some fields and methods.
    - Create a `Child` class that `extends` the `Parent` class.
    - In the `Child` class, add its own fields and methods.
    - Create objects of both `Parent` and `Child` and demonstrate access to inherited members.
3. **Method Overriding:**
    - In the `Parent` class, define a method (e.g., `display()`).
    - In the `Child` class, override the `display()` method to provide a specialized implementation.
    - Demonstrate calling the overridden method using both `Parent` and `Child` references (polymorphism).
    - Optionally, use `super.display()` in the child's overridden method.
4. **Multilevel Inheritance:**
    - Create a `Grandparent` class.
    - Create a `Parent` class that `extends Grandparent`.
    - Create a `Child` class that `extends Parent`.
    - Demonstrate method calls across the inheritance hierarchy.
5. **Abstract Classes and Methods:**
    - Create an `abstract` class (e.g., `Shape`) with an `abstract` method (e.g., `calculateArea()`).
    - The abstract method should have no body.
    - Add a concrete method to the abstract class.
    - Create concrete subclasses (e.g., `Circle`, `Rectangle`) that `extend` the `Shape` class.
    - Implement (override) the `calculateArea()` method in each subclass.
    - Demonstrate that you cannot instantiate an abstract class directly, but can use its reference to point to subclass objects.
    - Call the `calculateArea()` method polymorphically.
6. **Compilation and Execution:**
    - Compile and run the program.

Source Code

```
// Example: Inheritance, Method Overriding, Abstract Classes

// --- Single Inheritance & Method Overriding ---
class Animal {
    String name;

    public Animal(String name) {
        this.name = name;
    }

    public void eat() {
        System.out.println(name + " is eating.");
    }

    public void makeSound() {
        System.out.println(name + " makes a generic sound.");
    }
}

class Dog extends Animal {
    String breed;

    public Dog(String name, String breed) {
        super(name); // Call parent class constructor
        this.breed = breed;
    }

    @Override // Annotation to indicate method overriding
    public void makeSound() {
        System.out.println(name + " barks: Woof! Woof!");
    }

    public void fetch() {
        System.out.println(name + " is fetching the ball.");
    }
}

// --- Multilevel Inheritance ---
class Grandparent {
    void displayGrandparent() {
        System.out.println("This is a Grandparent class.");
    }
}

class Parent extends Grandparent {
    void displayParent() {
        System.out.println("This is a Parent class.");
    }
}

class Child extends Parent {
    void displayChild() {
        System.out.println("This is a Child class.");
    }
}

// --- Abstract Class and Methods ---
abstract class Shape {
    String color;

    public Shape(String color) {
        this.color = color;
    }

    // Abstract method (no body)
    public abstract double calculateArea();
```

```
    // Concrete method
    public void displayColor() {
        System.out.println("Shape color: " + color);
    }
}

class Circle extends Shape {
    double radius;

    public Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    double length;
    double width;

    public Rectangle(String color, double length, double width) {
        super(color);
        this.length = length;
        this.width = width;
    }

    @Override
    public double calculateArea() {
        return length * width;
    }
}


public class InheritanceDemo {
    public static void main(String[] args) {
        // --- Single Inheritance & Method Overriding Demo ---
        System.out.println("--- Single Inheritance & Method Overriding ---");
        Animal genericAnimal = new Animal("Generic Animal");
        genericAnimal.eat();
        genericAnimal.makeSound();

        Dog myDog = new Dog("Buddy", "Golden Retriever");
        myDog.eat(); // Inherited method
        myDog.makeSound(); // Overridden method
        myDog.fetch(); // Dog's specific method

        Animal polymorphicAnimal = new Dog("Max", "Labrador");
        polymorphicAnimal.makeSound(); // Calls Dog's makeSound due to
polymorphism

        // --- Multilevel Inheritance Demo ---
        System.out.println("\n--- Multilevel Inheritance ---");
        Child c = new Child();
        c.displayGrandparent(); // Inherited from Grandparent
        c.displayParent();      // Inherited from Parent
        c.displayChild();       // Own method

        // --- Abstract Class and Methods Demo ---
        System.out.println("\n--- Abstract Class and Methods ---");
        // Shape s = new Shape("Green"); // ERROR: Cannot instantiate abstract
class

        Circle circle = new Circle("Blue", 5.0);
```

```
        circle.displayColor();
        System.out.println("Area of Circle: " + circle.calculateArea());

        Rectangle rectangle = new Rectangle("Red", 4.0, 6.0);
        rectangle.displayColor();
        System.out.println("Area of Rectangle: " + rectangle.calculateArea());

        // Polymorphism with abstract class reference
        Shape s1 = new Circle("Yellow", 3.0);
        System.out.println("Area of s1 (Circle): " + s1.calculateArea());

        Shape s2 = new Rectangle("Orange", 7.0, 2.0);
        System.out.println("Area of s2 (Rectangle): " + s2.calculateArea());
    }
}
```

Input

No direct user input for this example.

Expected Output

```
--- Single Inheritance & Method Overriding ---
Generic Animal is eating.
Generic Animal makes a generic sound.
Buddy is eating.
Buddy barks: Woof! Woof!
Buddy is fetching the ball.
Max barks: Woof! Woof!

--- Multilevel Inheritance ---
This is a Grandparent class.
This is a Parent class.
This is a Child class.

--- Abstract Class and Methods ---
Shape color: Blue
Area of Circle: 78.53981633974483
Shape color: Red
Area of Rectangle: 24.0
Area of s1 (Circle): 28.274333882308138
Area of s2 (Rectangle): 14.0
```

Lab 8: Packages and Interfaces

Title

Implementing Packages and Interfaces in Java

Aim

To understand and implement Java packages for organizing classes and interfaces for achieving abstraction and multiple inheritance of behavior.

Procedure

1. **Project and Directory Structure:**
    - Create a new Java project.
    - Create a directory structure that reflects your package names (e.g., `mymath/operations/`, `mygeometry/shapes/`).
2. **Packages:**
    - **Create a Package:** In one file (e.g., `Adder.java`), declare a package at the top (e.g., `package mymath.operations;`).
    - **Create another Package:** In another file (e.g., `Circle.java`), declare a different package (e.g., `package mygeometry.shapes;`).
    - **Importing Packages:** In your `main` class (e.g., `PackageAndInterfaceDemo.java`), use the `import` statement to bring classes from other packages into scope.
    - Demonstrate creating objects and calling methods from classes in different packages.
3. **Interfaces:**
    - **Define an Interface:** Create an interface (e.g., `Drawable`) with abstract methods (e.g., `draw()`, `getArea()`). Interfaces can also contain constants.
    - **Implement an Interface:** Create classes (e.g., `Square`, `Triangle`) that `implement` the `Drawable` interface.
    - Provide concrete implementations for all abstract methods declared in the interface.
    - Demonstrate creating objects of the implementing classes and calling the interface methods polymorphically.
4. **Compilation and Execution:**
    - Compile the Java files, ensuring the package structure is maintained.
    - Run the main class.

Source Code

```
// Example: Packages and Interfaces

// --- Package 1: mymath.operations ---
// File: mymath/operations/Adder.java
package mymath.operations;

public class Adder {
    public int sum(int a, int b) {
        return a + b;
    }

    public double sum(double a, double b) {
        return a + b;
    }
}
```

```
// File: mymath/operations/Multiplier.java
package mymath.operations;

public class Multiplier {
    public int product(int a, int b) {
        return a * b;
    }
}

// --- Package 2: mygeometry.shapes ---
// File: mygeometry/shapes/ShapeConstants.java
package mygeometry.shapes;

public interface ShapeConstants {
    double PI = 3.14159; // Implicitly public static final
}

// File: mygeometry/shapes/Drawable.java
package mygeometry.shapes;

// Interface definition
public interface Drawable {
    void draw(); // Implicitly public abstract
    double getArea(); // Implicitly public abstract
}

// File: mygeometry/shapes/Circle.java
package mygeometry.shapes;

public class Circle implements Drawable, ShapeConstants {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a Circle with radius " + radius);
    }

    @Override
    public double getArea() {
        return PI * radius * radius;
    }
}

// File: mygeometry/shapes/Rectangle.java
package mygeometry.shapes;

public class Rectangle implements Drawable {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle with length " + length + " and
width " + width);
    }

    @Override
```

```
    public double getArea() {
        return length * width;
    }
}

// --- Main Class: PackageAndInterfaceDemo.java (in default package or another
package) ---
// Assuming this file is in the same directory as mymath and mygeometry folders,
// or in a separate package that imports them.
import mymath.operations.Adder;
import mymath.operations.Multiplier;
import mygeometry.shapes.Circle;
import mygeometry.shapes.Rectangle;
import mygeometry.shapes.Drawable; // Import the interface itself for
polymorphism

public class PackageAndInterfaceDemo {
    public static void main(String[] args) {
        // --- Packages Demo ---
        System.out.println("--- Packages Demo ---");
        Adder adder = new Adder();
        System.out.println("Sum using Adder: " + adder.sum(10, 20));
        System.out.println("Sum using Adder (double): " + adder.sum(10.5,
20.5));

        Multiplier multiplier = new Multiplier();
        System.out.println("Product using Multiplier: " + multiplier.product(5,
4));

        // --- Interfaces Demo ---
        System.out.println("\n--- Interfaces Demo ---");
        Circle circle = new Circle(7.0);
        circle.draw();
        System.out.println("Area of Circle: " + circle.getArea());

        Rectangle rectangle = new Rectangle(8.0, 5.0);
        rectangle.draw();
        System.out.println("Area of Rectangle: " + rectangle.getArea());

        // Polymorphism with Interface reference
        Drawable d1 = new Circle(3.0);
        d1.draw();
        System.out.println("Area using Drawable reference (Circle): " +
d1.getArea());

        Drawable d2 = new Rectangle(6.0, 2.0);
        d2.draw();
        System.out.println("Area using Drawable reference (Rectangle): " +
d2.getArea());
    }
}
```

Input

No direct user input for this example.

Expected Output

```
--- Packages Demo ---
Sum using Adder: 30
Sum using Adder (double): 31.0
Product using Multiplier: 20

--- Interfaces Demo ---
```

```
Drawing a Circle with radius 7.0
Area of Circle: 153.93791
Drawing a Rectangle with length 8.0 and width 5.0
Area of Rectangle: 40.0
Drawing a Circle with radius 3.0
Area using Drawable reference (Circle): 28.27431
Drawing a Rectangle with length 6.0 and width 2.0
Area using Drawable reference (Rectangle): 12.0
```

(Note: `PI` value might vary slightly based on `Math.PI` vs `3.14159` used.)

Lab 9: Exception Handling

Title

Implementing Exception Handling in Java

Aim

To understand and implement Java's exception handling mechanism using `try`, `catch`, `finally`, `throw`, and `throws` keywords to gracefully manage runtime errors.

Procedure

1. **Project and Class Creation:**
    - Create a new Java project and a class (e.g., `ExceptionHandlingDemo.java`).
2. **`try-catch` Block:**
    - Write a program that might cause a `NullPointerException` (e.g., accessing a method on a `null` object).
    - Wrap the problematic code in a `try` block.
    - Add a `catch` block to handle the `NullPointerException`, printing an informative message.
    - Repeat for `ArithmeticException` (e.g., division by zero) and `ArrayIndexOutOfBoundsException`.
3. **Multiple `catch` Blocks:**
    - Write a program that can throw multiple types of exceptions (e.g., a method that takes an array and an index, potentially throwing `ArrayIndexOutOfBoundsException` or `ArithmeticException` if division by zero occurs).
    - Use multiple `catch` blocks to handle each specific exception type.
    - Demonstrate the order of `catch` blocks (most specific to most general).
4. **`finally` Block:**
    - Add a `finally` block to one of your `try-catch` examples.
    - Place code in the `finally` block that should always execute, regardless of whether an exception occurred or was caught (e.g., closing a resource).
5. **`throw` Keyword (Custom Exception):**
    - Create a custom exception class by extending `Exception` or `RuntimeException` (e.g., `InvalidAgeException`).
    - Write a method that `throws` this custom exception under certain conditions (e.g., if age is negative).
    - Call this method from `main` and handle the custom exception using `try-catch`.
6. **`throws` Keyword:**
    - Create a method that declares it `throws` a checked exception (e.g., `InterruptedException` if using `Thread.sleep()`).
    - Call this method from `main` and observe that `main` must either handle the exception or declare `throws` itself.
7. **Compilation and Execution:**
    - Compile and run the program, testing different scenarios to trigger exceptions.

Source Code

```
// Example: Exception Handling
```

```
// Custom Exception Class
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class ExceptionHandlingDemo {

    // Method demonstrating throws keyword
    public static void simulateLongOperation() throws InterruptedException {
        System.out.println("Simulating long operation...");
        Thread.sleep(2000); // This can throw InterruptedException (checked
exception)
        System.out.println("Operation completed.");
    }

    // Method demonstrating throw keyword with custom exception
    public static void checkAge(int age) throws InvalidAgeException {
        if (age < 0) {
            throw new InvalidAgeException("Age cannot be negative: " + age);
        } else if (age > 120) {
            throw new InvalidAgeException("Age seems too high: " + age);
        }
        System.out.println("Age is valid: " + age);
    }

    public static void main(String[] args) {
        // --- try-catch for NullPointerException ---
        System.out.println("--- NullPointerException Demo ---");
        String str = null;
        try {
            System.out.println("Length of string: " + str.length()); // This
will throw NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Caught NullPointerException: Cannot get length
of a null string.");
            System.out.println("Error message: " + e.getMessage());
        }

        // --- try-catch for ArithmeticException ---
        System.out.println("\n--- ArithmeticException Demo ---");
        int numerator = 10;
        int denominator = 0;
        try {
            int result = numerator / denominator; // This will throw
ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Caught ArithmeticException: Division by zero is
not allowed.");
            System.out.println("Error message: " + e.getMessage());
        }

        // --- Multiple catch blocks & finally block ---
        System.out.println("\n--- Multiple catch & finally Demo ---");
        int[] numbers = {1, 2, 3};
        try {
            System.out.println("Accessing element at index 5: " + numbers[5]);
// ArrayIndexOutOfBoundsException
            int divResult = 10 / 0; // ArithmeticException
            System.out.println("This line will not be executed.");
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught ArrayIndexOutOfBoundsException: Array
index out of bounds.");
            System.out.println("Error: " + e.getMessage());
```

```
        } catch (ArithmeticException e) {
            System.out.println("Caught ArithmeticException: Division by zero.");
            System.out.println("Error: " + e.getMessage());
        } catch (Exception e) { // General catch block (must be last)
            System.out.println("Caught a general Exception: " + e.getMessage());
        } finally {
            System.out.println("Finally block executed. This always runs.");
        }

        // --- try-catch for `throw` (Custom Exception) ---
        System.out.println("\n--- Custom Exception (throw) Demo ---");
        try {
            checkAge(25);
            checkAge(-5); // This will throw InvalidAgeException
            checkAge(150); // This line will not be executed
        } catch (InvalidAgeException e) {
            System.out.println("Caught Custom Exception: " + e.getMessage());
        }

        // --- `throws` keyword Demo ---
        System.out.println("\n--- throws Keyword Demo ---");
        try {
            simulateLongOperation(); // Method declares it throws
InterruptedException
        } catch (InterruptedException e) {
            System.out.println("Caught InterruptedException in main: " +
e.getMessage());
            Thread.currentThread().interrupt(); // Restore the interrupted
status
        }
    }
}
```

Input

No direct user input for this example.

Expected Output

```
--- NullPointerException Demo ---
Caught NullPointerException: Cannot get length of a null string.
Error message: null

--- ArithmeticException Demo ---
Caught ArithmeticException: Division by zero is not allowed.
Error message: / by zero

--- Multiple catch & finally Demo ---
Caught ArrayIndexOutOfBoundsException: Array index out of bounds.
Error: Index 5 out of bounds for length 3
Finally block executed. This always runs.

--- Custom Exception (throw) Demo ---
Age is valid: 25
Caught Custom Exception: Age cannot be negative: -5

--- throws Keyword Demo ---
Simulating long operation...
Operation completed.
```

(Note: If you change `numbers[5]` to `10 / 0` in the multiple catch block, you'll see the `ArithmeticException` caught instead. The `simulateLongOperation` will complete normally unless an external interrupt occurs.)

Lab 10: Multithreading

Title

Implementing Multithreading in Java

Aim

To understand and implement multithreading in Java using the `Thread` class and the `Runnable` interface, and to explore thread lifecycle and synchronization concepts.

Procedure

1. **Project and Class Creation:**
    - Create a new Java project and a class (e.g., `MultithreadingDemo.java`).
2. **Creating Threads by Extending `Thread` Class:**
    - Create a class that `extends java.lang.Thread`.
    - Override the `run()` method to define the task the thread will execute.
    - In the `main` method, create instances of your custom thread class.
    - Call the `start()` method on each thread object to begin execution.
3. **Creating Threads by Implementing `Runnable` Interface:**
    - Create a class that `implements java.lang.Runnable`.
    - Implement the `run()` method.
    - In the `main` method, create an instance of your `Runnable` implementation.
    - Create a `Thread` object, passing your `Runnable` instance to its constructor.
    - Call the `start()` method on the `Thread` object.
    - Discuss the advantages of `Runnable` (e.g., allows class to extend another class, better separation of concerns).
4. **Thread Lifecycle (Sleep, Join):**
    - Use `Thread.sleep()` to pause a thread's execution for a specified duration.
    - Use `thread.join()` to make the current thread wait until the specified thread terminates.
5. **Synchronization (Optional but Recommended):**
    - Create a shared resource (e.g., a counter) that multiple threads will try to modify.
    - Demonstrate a race condition without synchronization.
    - Use the `synchronized` keyword (on a method or a block) to protect the shared resource and prevent data inconsistency.
6. **Compilation and Execution:**
    - Compile and run the program. Observe the interleaved output from different threads.

Source Code

```
// Example: Multithreading

// 1. Creating a thread by extending Thread class
class MyThread extends Thread {
    private String threadName;

    public MyThread(String name) {
        this.threadName = name;
        System.out.println("Creating " + threadName);
    }
```

```
    @Override
    public void run() {
        System.out.println("Running " + threadName);
        try {
            for (int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
                // Pause for a bit
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        System.out.println("Thread " + threadName + " exiting.");
    }
}

// 2. Creating a thread by implementing Runnable interface
class MyRunnable implements Runnable {
    private String runnableName;

    public MyRunnable(String name) {
        this.runnableName = name;
        System.out.println("Creating Runnable " + runnableName);
    }

    @Override
    public void run() {
        System.out.println("Running Runnable " + runnableName);
        try {
            for (int i = 4; i > 0; i--) {
                System.out.println("Runnable: " + runnableName + ", " + i);
                Thread.sleep(70);
            }
        } catch (InterruptedException e) {
            System.out.println("Runnable " + runnableName + " interrupted.");
        }
        System.out.println("Runnable " + runnableName + " exiting.");
    }
}

// 3. Shared resource for synchronization demo
class Counter {
    private int count = 0;

    // Synchronized method to increment count
    public synchronized void increment() {
        count++;
        System.out.println(Thread.currentThread().getName() + " incremented
count to: " + count);
    }

    public int getCount() {
        return count;
    }
}

class CounterThread extends Thread {
    private Counter counter;

    public CounterThread(String name, Counter counter) {
        super(name);
        this.counter = counter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
```

```
            counter.increment();
            try {
                Thread.sleep(10); // Simulate some work
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}


public class MultithreadingDemo {
    public static void main(String[] args) {
        System.out.println("--- Thread by extending Thread class ---");
        MyThread thread1 = new MyThread("Thread-A");
        MyThread thread2 = new MyThread("Thread-B");

        thread1.start();
        thread2.start();

        // Using join to wait for threads to finish
        try {
            thread1.join(); // Wait for thread1 to complete
            thread2.join(); // Wait for thread2 to complete
            System.out.println("\nAll Thread class threads have finished.");
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted during join.");
        }

        System.out.println("\n--- Thread by implementing Runnable interface ---
");
        MyRunnable runnable1 = new MyRunnable("Runnable-X");
        MyRunnable runnable2 = new MyRunnable("Runnable-Y");

        Thread thread3 = new Thread(runnable1);
        Thread thread4 = new Thread(runnable2);

        thread3.start();
        thread4.start();

        try {
            thread3.join();
            thread4.join();
            System.out.println("\nAll Runnable interface threads have
finished.");
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted during join.");
        }

        // --- Synchronization Demo ---
        System.out.println("\n--- Synchronization Demo ---");
        Counter sharedCounter = new Counter();

        CounterThread ct1 = new CounterThread("CounterThread-1", sharedCounter);
        CounterThread ct2 = new CounterThread("CounterThread-2", sharedCounter);
        CounterThread ct3 = new CounterThread("CounterThread-3", sharedCounter);

        ct1.start();
        ct2.start();
        ct3.start();

        try {
            ct1.join();
            ct2.join();
            ct3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
        System.out.println("Final Counter value (should be 15): " +
sharedCounter.getCount());

        System.out.println("\nMain thread exiting.");
    }
}
```

Input

No direct user input for this example.

Expected Output

The output for multithreading will vary slightly each time you run it due to the nature of thread scheduling. However, the general flow and messages indicating thread creation, running, and exiting will be consistent. The synchronized counter should always result in the correct final value (15 in this case).

A sample output might look like this (order of `Thread:`, `Runnable:`, and `incremented count` lines will be interleaved):

```
--- Thread by extending Thread class ---
Creating Thread-A
Creating Thread-B
Running Thread-A
Running Thread-B
Thread: Thread-A, 4
Thread: Thread-B, 4
Thread: Thread-A, 3
Thread: Thread-B, 3
Thread: Thread-A, 2
Thread: Thread-B, 2
Thread: Thread-A, 1
Thread: Thread-B, 1
Thread Thread-A exiting.
Thread Thread-B exiting.

All Thread class threads have finished.

--- Thread by implementing Runnable interface ---
Creating Runnable Runnable-X
Creating Runnable Runnable-Y
Running Runnable Runnable-X
Running Runnable Runnable-Y
Runnable: Runnable-X, 4
Runnable: Runnable-Y, 4
Runnable: Runnable-X, 3
Runnable: Runnable-Y, 3
Runnable: Runnable-X, 2
Runnable: Runnable-Y, 2
Runnable: Runnable-X, 1
Runnable: Runnable-Y, 1
Runnable Runnable-X exiting.
Runnable Runnable-Y exiting.

All Runnable interface threads have finished.

--- Synchronization Demo ---
CounterThread-1 incremented count to: 1
CounterThread-2 incremented count to: 2
CounterThread-3 incremented count to: 3
```

```
CounterThread-1 incremented count to: 4
CounterThread-2 incremented count to: 5
CounterThread-3 incremented count to: 6
CounterThread-1 incremented count to: 7
CounterThread-2 incremented count to: 8
CounterThread-3 incremented count to: 9
CounterThread-1 incremented count to: 10
CounterThread-2 incremented count to: 11
CounterThread-3 incremented count to: 12
CounterThread-1 incremented count to: 13
CounterThread-2 incremented count to: 14
CounterThread-3 incremented count to: 15
Final Counter value (should be 15): 15

Main thread exiting.
```

Lab 11: Legacy Classes and Interfaces

Title

Exploring Legacy Classes and Interfaces in Java (Collections Framework - Older Parts)

Aim

To understand and utilize some of the older (legacy) classes and interfaces from Java's Collections Framework, such as `Vector`, `Stack`, `Hashtable`, and `Enumeration`, and to compare them with their modern counterparts.

Procedure

1. **Project and Class Creation:**
    - Create a new Java project and a class (e.g., `LegacyCollectionsDemo.java`).
2. **`Vector` Class:**
    - Create a `Vector` of elements (e.g., `String` or `Integer`).
    - Add elements using `addElement()`.
    - Access elements using `elementAt(index)`.
    - Remove elements using `removeElement()`.
    - Get size using `size()`.
    - Iterate using a traditional `for` loop and also using `Enumeration`.
    - Compare its behavior (synchronized, capacity increment) with `ArrayList`.
3. **`Stack` Class:**
    - Create a `Stack` object.
    - Push elements onto the stack using `push()`.
    - Pop elements from the stack using `pop()`.
    - Peek at the top element using `peek()`.
    - Check if empty using `empty()`.
    - Demonstrate LIFO (Last-In, First-Out) behavior.
4. **`Hashtable` Class:**
    - Create a `Hashtable` object to store key-value pairs (e.g., `String` keys and `Integer` values).
    - Add elements using `put()`.
    - Retrieve elements using `get()`.
    - Remove elements using `remove()`.
    - Iterate through keys and values using `keys()` (returns `Enumeration`) and `elements()` (returns `Enumeration`).
    - Compare its behavior (synchronized, no nulls) with `HashMap`.
5. **`Enumeration` Interface:**
    - Demonstrate iterating over `Vector` and `Hashtable` elements using the `Enumeration` interface (`hasMoreElements()`, `nextElement()`).
6. **Compilation and Execution:**
    - Compile and run the program.

Source Code

```
// Example: Legacy Classes and Interfaces

import java.util.Vector;
```

```
import java.util.Stack;
import java.util.Hashtable;
import java.util.Enumeration; // For iterating over legacy collections

public class LegacyCollectionsDemo {
    public static void main(String[] args) {
        // --- Vector Class Demo ---
        System.out.println("--- Vector Class Demo ---");
        Vector<String> vectorList = new Vector<>();
        vectorList.addElement("Apple");
        vectorList.addElement("Banana");
        vectorList.addElement("Cherry");
        vectorList.addElement("Date");

        System.out.println("Vector elements: " + vectorList);
        System.out.println("Size of Vector: " + vectorList.size());
        System.out.println("Element at index 1: " + vectorList.elementAt(1));

        vectorList.removeElement("Banana");
        System.out.println("Vector after removing Banana: " + vectorList);

        System.out.println("Iterating Vector using Enumeration:");
        Enumeration<String> vectorEnum = vectorList.elements();
        while (vectorEnum.hasMoreElements()) {
            System.out.println("Element: " + vectorEnum.nextElement());
        }

        // --- Stack Class Demo ---
        System.out.println("\n--- Stack Class Demo ---");
        Stack<Integer> stack = new Stack<>();
        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println("Stack: " + stack);
        System.out.println("Peek (top element): " + stack.peek());
        System.out.println("Pop: " + stack.pop());
        System.out.println("Stack after pop: " + stack);
        System.out.println("Pop: " + stack.pop());
        System.out.println("Is stack empty? " + stack.empty());
        System.out.println("Pop: " + stack.pop());
        System.out.println("Is stack empty? " + stack.empty());

        // --- Hashtable Class Demo ---
        System.out.println("\n--- Hashtable Class Demo ---");
        Hashtable<String, Integer> hashtable = new Hashtable<>();
        hashtable.put("One", 1);
        hashtable.put("Two", 2);
        hashtable.put("Three", 3);

        System.out.println("Hashtable: " + hashtable);
        System.out.println("Value for key 'Two': " + hashtable.get("Two"));

        hashtable.remove("One");
        System.out.println("Hashtable after removing 'One': " + hashtable);

        System.out.println("Iterating Hashtable keys using Enumeration:");
        Enumeration<String> keys = hashtable.keys();
        while (keys.hasMoreElements()) {
            String key = keys.nextElement();
            System.out.println("Key: " + key + ", Value: " +
hashtable.get(key));
        }

        System.out.println("Iterating Hashtable values using Enumeration:");
        Enumeration<Integer> values = hashtable.elements();
        while (values.hasMoreElements()) {
```

```
            System.out.println("Value: " + values.nextElement());
        }
    }
}
```

Input

No direct user input for this example.

Expected Output

```
--- Vector Class Demo ---
Vector elements: [Apple, Banana, Cherry, Date]
Size of Vector: 4
Element at index 1: Banana
Vector after removing Banana: [Apple, Cherry, Date]
Iterating Vector using Enumeration:
Element: Apple
Element: Cherry
Element: Date

--- Stack Class Demo ---
Stack: [10, 20, 30]
Peek (top element): 30
Pop: 30
Stack after pop: [10, 20]
Pop: 20
Is stack empty? false
Pop: 10
Is stack empty? true

--- Hashtable Class Demo ---
Hashtable: {Two=2, Three=3, One=1}
Value for key 'Two': 2
Hashtable after removing 'One': {Two=2, Three=3}
Iterating Hashtable keys using Enumeration:
Key: Two, Value: 2
Key: Three, Value: 3
Iterating Hashtable values using Enumeration:
Value: 2
Value: 3
```

(Note: The order of elements in `Hashtable` output might vary as `Hashtable` does not guarantee insertion order.)

Lab 12: Utility Classes

Title

Utilizing Key Utility Classes in Java

Aim

To understand and apply commonly used utility classes in Java, focusing on `ArrayList`, `HashMap`, `Date`, `Calendar`, and `Random`.

Procedure

1. **Project and Class Creation:**
    - Create a new Java project and a class (e.g., `UtilityClassesDemo.java`).
2. **`ArrayList` Class:**
    - Create an `ArrayList` of `String` or `Integer`.
    - Add elements using `add()`.
    - Access elements using `get(index)`.
    - Remove elements using `remove()`.
    - Get size using `size()`.
    - Iterate using a `for` loop and an enhanced `for` loop.
    - Demonstrate dynamic resizing.
3. **`HashMap` Class:**
    - Create a `HashMap` to store key-value pairs (e.g., `String` keys and `Integer` values).
    - Add elements using `put()`.
    - Retrieve elements using `get()`.
    - Remove elements using `remove()`.
    - Iterate through keys using `keySet()` and values using `values()`.
    - Iterate through entries using `entrySet()`.
4. **`Date` Class (Legacy but still seen):**
    - Create a `Date` object to represent the current date and time.
    - Print the `Date` object.
    - (Briefly mention `java.time` package as modern alternative).
5. **`Calendar` Class:**
    - Get an instance of `Calendar` (e.g., `Calendar.getInstance()`).
    - Retrieve current year, month, day, hour, minute, second.
    - Set specific date/time components.
    - Add/subtract time units (e.g., `add(Calendar.DAY_OF_MONTH, 5)`).
6. **`Random` Class:**
    - Create a `Random` object.
    - Generate random integers within a range (e.g., `nextInt(bound)`).
    - Generate random doubles.
    - Generate random booleans.
7. **Compilation and Execution:**
    - Compile and run the program.

Source Code

```
// Example: Utility Classes
```

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Date; // Legacy Date class
import java.util.Calendar; // Legacy Calendar class
import java.util.Random;

public class UtilityClassesDemo {
    public static void main(String[] args) {
        // --- ArrayList Class Demo ---
        System.out.println("--- ArrayList Class Demo ---");
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        fruits.add("Date");

        System.out.println("ArrayList elements: " + fruits);
        System.out.println("Size of ArrayList: " + fruits.size());
        System.out.println("Element at index 1: " + fruits.get(1));

        fruits.remove("Banana");
        System.out.println("ArrayList after removing Banana: " + fruits);

        System.out.println("Iterating ArrayList using enhanced for loop:");
        for (String fruit : fruits) {
            System.out.println("Fruit: " + fruit);
        }

        // --- HashMap Class Demo ---
        System.out.println("\n--- HashMap Class Demo ---");
        HashMap<String, Integer> studentScores = new HashMap<>();
        studentScores.put("Alice", 95);
        studentScores.put("Bob", 88);
        studentScores.put("Charlie", 92);

        System.out.println("HashMap: " + studentScores);
        System.out.println("Bob's score: " + studentScores.get("Bob"));

        studentScores.put("Alice", 98); // Update value
        System.out.println("HashMap after updating Alice's score: " +
studentScores);

        studentScores.remove("Charlie");
        System.out.println("HashMap after removing Charlie: " + studentScores);

        System.out.println("Iterating HashMap entries:");
        for (HashMap.Entry<String, Integer> entry : studentScores.entrySet()) {
            System.out.println("Student: " + entry.getKey() + ", Score: " +
entry.getValue());
        }

        // --- Date Class Demo ---
        System.out.println("\n--- Date Class Demo (Legacy) ---");
        Date currentDate = new Date();
        System.out.println("Current Date and Time: " + currentDate);

        // --- Calendar Class Demo ---
        System.out.println("\n--- Calendar Class Demo ---");
        Calendar calendar = Calendar.getInstance();
        System.out.println("Current Year: " + calendar.get(Calendar.YEAR));
        // Month is 0-indexed (January is 0)
        System.out.println("Current Month (0-indexed): " +
calendar.get(Calendar.MONTH));
        System.out.println("Current Day of Month: " +
calendar.get(Calendar.DAY_OF_MONTH));
        System.out.println("Current Hour (12-hour format): " +
calendar.get(Calendar.HOUR));
```

```
        System.out.println("Current Minute: " + calendar.get(Calendar.MINUTE));
        System.out.println("Current Second: " + calendar.get(Calendar.SECOND));

        // Adding 5 days to the current date
        calendar.add(Calendar.DAY_OF_MONTH, 5);
        System.out.println("Date after adding 5 days: " + calendar.getTime());

        // --- Random Class Demo ---
        System.out.println("\n--- Random Class Demo ---");
        Random random = new Random();

        System.out.println("Random integer (0 to 99): " + random.nextInt(100));
// 0 to 99
        System.out.println("Random integer (0 to 9): " + random.nextInt(10));
// 0 to 9
        System.out.println("Random double (0.0 to 1.0): " +
random.nextDouble());
        System.out.println("Random boolean: " + random.nextBoolean());
    }
}
```

Input

No direct user input for this example.

Expected Output

```
--- ArrayList Class Demo ---
ArrayList elements: [Apple, Banana, Cherry, Date]
Size of ArrayList: 4
Element at index 1: Banana
ArrayList after removing Banana: [Apple, Cherry, Date]
Iterating ArrayList using enhanced for loop:
Fruit: Apple
Fruit: Cherry
Fruit: Date

--- HashMap Class Demo ---
HashMap: {Bob=88, Charlie=92, Alice=95}
Bob's score: 88
HashMap after updating Alice's score: {Bob=88, Charlie=92, Alice=98}
HashMap after removing Charlie: {Bob=88, Alice=98}
Iterating HashMap entries:
Student: Bob, Score: 88
Student: Alice, Score: 98

--- Date Class Demo (Legacy) ---
Current Date and Time: Tue May 21 09:05:00 IST 2025 (actual date/time will vary)

--- Calendar Class Demo ---
Current Year: 2025
Current Month (0-indexed): 4 (for May)
Current Day of Month: 21
Current Hour (12-hour format): 9
Current Minute: 5
Current Second: 0 (actual time will vary)
Date after adding 5 days: Sun May 26 09:05:00 IST 2025 (actual date/time will
vary)

--- Random Class Demo ---
Random integer (0 to 99): 42 (will be different each run)
Random integer (0 to 9): 7 (will be different each run)
Random double (0.0 to 1.0): 0.123456789 (will be different each run)
Random boolean: true (will be different each run)
```

(Note: Actual date, time, and random numbers will vary based on when you run the program.)

Lab 13: Event Handling

Title

Implementing Event Handling in Java (AWT/Swing Basics)

Aim

To understand the event delegation model in Java and implement basic event handling for GUI components using AWT/Swing, specifically focusing on `ActionListener` for buttons.

Procedure

1. **Project and Class Creation:**
    - Create a new Java project.
    - Create a class (e.g., `EventHandlingDemo.java`) that extends `JFrame` (for the main window).
    - Import necessary AWT/Swing classes (`java.awt.*`, `javax.swing.*`, `java.awt.event.*`).
2. **Create GUI Components:**
    - In the constructor of your `JFrame` class:
        - Create a `JButton` (e.g., "Click Me").
        - Create a `JLabel` to display messages.
        - Set the layout manager for the frame (e.g., `FlowLayout` or `BorderLayout`).
        - Add the button and label to the frame.
        - Set frame properties (size, default close operation, visibility).
3. **Implement `ActionListener`:**
    - Make your `EventHandlingDemo` class `implement ActionListener`.
    - Override the `actionPerformed(ActionEvent e)` method. This method will contain the logic to execute when an action event occurs.
4. **Register Listener:**
    - In the constructor, register the `ActionListener` with the button using `button.addActionListener(this);`.
5. **Event Logic:**
    - Inside `actionPerformed()`:
        - Use `e.getSource()` to identify which component triggered the event.
        - Update the text of the `JLabel` based on the button click.
        - Optionally, use `e.getActionCommand()` to get the command string associated with the action.
6. **Main Method:**
    - In the `main` method, create an instance of your `EventHandlingDemo` class to make the GUI visible.
7. **Compilation and Execution:**
    - Compile and run the program. Click the button and observe the label changing.

Source Code

```
// Example: Event Handling

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

```
import java.awt.event.ActionListener;

public class EventHandlingDemo extends JFrame implements ActionListener {

    private JButton clickButton;
    private JLabel messageLabel;
    private int clickCount = 0;

    public EventHandlingDemo() {
        // Set frame properties
        setTitle("Event Handling Demo");
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null); // Center the window

        // Set layout manager (FlowLayout for simple arrangement)
        setLayout(new FlowLayout());

        // Create GUI components
        clickButton = new JButton("Click Me!");
        messageLabel = new JLabel("Button not clicked yet.");

        // Register the ActionListener with the button
        // 'this' refers to the current JFrame object, which implements
ActionListener
        clickButton.addActionListener(this);

        // Add components to the frame
        add(clickButton);
        add(messageLabel);

        // Make the frame visible
        setVisible(true);
    }

    // This method is called when an action event occurs (e.g., button click)
    @Override
    public void actionPerformed(ActionEvent e) {
        // Check if the event source is our clickButton
        if (e.getSource() == clickButton) {
            clickCount++;
            messageLabel.setText("Button clicked " + clickCount + " time(s)!");
            System.out.println("Button was clicked!"); // For console
verification
        }
    }

    public static void main(String[] args) {
        // Create an instance of the JFrame on the Event Dispatch Thread (EDT)
        // This is good practice for Swing applications
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                new EventHandlingDemo();
            }
        });
    }
}
```

Input

User interaction: Click the "Click Me!" button.

Expected Output

A GUI window will appear with a "Click Me!" button and a label. Initially, the label will display: "Button not clicked yet." Each time you click the button, the label will update to: "Button clicked 1 time(s)!", "Button clicked 2 time(s)!", and so on. The console will also print "Button was clicked!" for each click.

Lab 14: AWT Controls

Title

Working with AWT Controls (Basic GUI Components)

Aim

To understand and implement various basic AWT (Abstract Window Toolkit) controls such as `Button`, `Label`, `TextField`, `TextArea`, `Checkbox`, `Choice`, and `List` to build simple graphical user interfaces.

Procedure

1. **Project and Class Creation:**
    - Create a new Java project.
    - Create a class (e.g., `AWTControlsDemo.java`) that extends `Frame` (for the main window).
    - Import necessary AWT classes (`java.awt.*`, `java.awt.event.*`).
2. **Frame Setup:**
    - In the constructor of your `Frame` class:
        - Set the title, size, and layout manager (e.g., `FlowLayout` or `GridLayout` for simple arrangement).
        - Add a `WindowListener` to handle closing the frame (since AWT frames don't close by default).
3. **Create AWT Controls:**
    - Instantiate various AWT components:
        - `Label`: For displaying static text.
        - `TextField`: For single-line text input.
        - `TextArea`: For multi-line text input.
        - `Button`: For clickable actions.
        - `Checkbox`: For boolean choices (can be grouped using `CheckboxGroup`).
        - `Choice`: For a dropdown list of options.
        - `List`: For a scrollable list of options (single or multiple selection).
4. **Add Controls to Frame:**
    - Use `add()` method to place each component onto the frame.
5. **Event Handling (Basic):**
    - Implement `ActionListener` for `Button` and `TextField` (for Enter key).
    - Implement `ItemListener` for `Checkbox` and `Choice`/`List`.
    - In `actionPerformed()` or `itemStateChanged()`, update a `TextArea` or `Label` to show the interaction.
6. **Main Method:**
    - In the `main` method, create an instance of your `AWTControlsDemo` class to make the GUI visible.
7. **Compilation and Execution:**
    - Compile and run the program. Interact with the controls and observe the output.

Source Code

```
// Example: AWT Controls
```

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

public class AWTControlsDemo extends Frame implements ActionListener,
ItemListener {

    private Label headerLabel;
    private TextField nameTextField;
    private TextArea outputTextArea;
    private Button submitButton;
    private Checkbox maleCheckbox, femaleCheckbox;
    private CheckboxGroup genderGroup;
    private Choice countryChoice;
    private List hobbiesList;

    public AWTControlsDemo() {
        // Frame setup
        setTitle("AWT Controls Demo");
        setSize(500, 450);
        setLayout(new FlowLayout()); // Simple layout for demonstration

        // Add WindowListener for closing the frame
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });

        // --- Components ---
        headerLabel = new Label("Enter your details:");
        add(headerLabel);

        // TextField
        add(new Label("Name:"));
        nameTextField = new TextField(20);
        nameTextField.addActionListener(this); // Listen for Enter key
        add(nameTextField);

        // Button
        submitButton = new Button("Submit");
        submitButton.addActionListener(this);
        add(submitButton);

        // CheckboxGroup (for radio buttons)
        add(new Label("Gender:"));
        genderGroup = new CheckboxGroup();
        maleCheckbox = new Checkbox("Male", genderGroup, false);
        femaleCheckbox = new Checkbox("Female", genderGroup, false);
        maleCheckbox.addItemListener(this);
        femaleCheckbox.addItemListener(this);
        add(maleCheckbox);
        add(femaleCheckbox);

        // Choice (Dropdown)
        add(new Label("Country:"));
        countryChoice = new Choice();
        countryChoice.add("USA");
        countryChoice.add("India");
        countryChoice.add("Canada");
        countryChoice.add("UK");
        countryChoice.addItemListener(this);
        add(countryChoice);

        // List (Scrollable list)
        add(new Label("Hobbies:"));
        hobbiesList = new List(4, true); // 4 visible rows, multiple selection
enabled
```

```
        hobbiesList.add("Reading");
        hobbiesList.add("Sports");
        hobbiesList.add("Coding");
        hobbiesList.add("Traveling");
        hobbiesList.add("Music");
        hobbiesList.addItemListener(this);
        add(hobbiesList);

        // TextArea for output
        outputTextArea = new TextArea("Output will appear here...", 10, 40,
TextArea.SCROLLBARS_VERTICAL_ONLY);
        outputTextArea.setEditable(false);
        add(outputTextArea);

        setVisible(true);
    }

    // ActionListener for Button and TextField
    @Override
    public void actionPerformed(ActionEvent ae) {
        String command = ae.getActionCommand();
        if (command.equals("Submit")) {
            displayOutput();
        } else if (ae.getSource() == nameTextField) {
            outputTextArea.append("Name entered (via Enter key): " +
nameTextField.getText() + "\n");
        }
    }

    // ItemListener for Checkbox, Choice, List
    @Override
    public void itemStateChanged(ItemEvent ie) {
        if (ie.getSource() == maleCheckbox || ie.getSource() == femaleCheckbox)
{
            Checkbox selectedGender = genderGroup.getSelectedCheckbox();
            if (selectedGender != null) {
                outputTextArea.append("Gender selected: " +
selectedGender.getLabel() + "\n");
            }
        } else if (ie.getSource() == countryChoice) {
            outputTextArea.append("Country selected: " +
countryChoice.getSelectedItem() + "\n");
        } else if (ie.getSource() == hobbiesList) {
            String[] selectedHobbies = hobbiesList.getSelectedItems();
            outputTextArea.append("Hobbies selected: ");
            for (String hobby : selectedHobbies) {
                outputTextArea.append(hobby + " ");
            }
            outputTextArea.append("\n");
        }
    }

    private void displayOutput() {
        outputTextArea.setText(""); // Clear previous output
        outputTextArea.append("--- Submission Details ---\n");
        outputTextArea.append("Name: " + nameTextField.getText() + "\n");

        Checkbox selectedGender = genderGroup.getSelectedCheckbox();
        if (selectedGender != null) {
            outputTextArea.append("Gender: " + selectedGender.getLabel() +
"\n");
        } else {
            outputTextArea.append("Gender: Not selected\n");
        }

        outputTextArea.append("Country: " + countryChoice.getSelectedItem() +
"\n");
```

```
        String[] selectedHobbies = hobbiesList.getSelectedItems();
        if (selectedHobbies.length > 0) {
            outputTextArea.append("Hobbies: ");
            for (String hobby : selectedHobbies) {
                outputTextArea.append(hobby + ", ");
            }
            outputTextArea.replaceRange("",
outputTextArea.getText().lastIndexOf(", "), outputTextArea.getText().length());
// Remove last comma
            outputTextArea.append("\n");
        } else {
            outputTextArea.append("Hobbies: None selected\n");
        }
        outputTextArea.append("------------------------\n");
    }

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

Input

User interaction:

- Type text in the "Name" field and press Enter.
- Click the "Submit" button.
- Select "Male" or "Female" checkbox.
- Choose an option from the "Country" dropdown.
- Select one or more items from the "Hobbies" list (Ctrl+Click for multiple).

Expected Output

A GUI window will appear with various AWT controls. The `TextArea` at the bottom will display messages corresponding to your interactions:

- When you type in the name field and press Enter, it will log "Name entered...".
- When you select gender, country, or hobbies, it will log your selections.
- When you click "Submit", it will clear the `TextArea` and display a summary of all selected/entered details.

Lab 15: Layout Managers, Byte and Character Streams

Title

Understanding Layout Managers and File I/O (Byte and Character Streams) in Java

Aim

To understand and apply various AWT/Swing Layout Managers (`BorderLayout`, `GridLayout`, `FlowLayout`) for arranging GUI components, and to implement file input/output operations using both byte streams and character streams.

Procedure

1. **Project and Class Creation:**
    - Create a new Java project.
    - Create a class (e.g., `LayoutAndStreamsDemo.java`) that extends `JFrame` or `Frame`.
    - Import necessary AWT/Swing and I/O classes (`java.awt.*`, `javax.swing.*`, `java.io.*`).
2. **Layout Managers:**
    - **`FlowLayout`:**
        - Create a simple frame and add a few buttons using `FlowLayout`. Observe how components flow.
    - **`BorderLayout`:**
        - Create a frame and add buttons to `BorderLayout.NORTH`, `SOUTH`, `EAST`, `WEST`, and `CENTER`.
    - **`GridLayout`:**
        - Create a frame and add buttons to a `GridLayout` (e.g., 3 rows, 2 columns).
    - (Optional: Combine layouts by nesting panels with different layouts).
3. **Byte Streams (`FileInputStream`, `FileOutputStream`):**
    - **Writing Bytes:**
        - Create a `FileOutputStream` to write raw bytes to a file (e.g., `data.bin`).
        - Write a sequence of bytes or an integer's byte representation.
        - Close the stream.
    - **Reading Bytes:**
        - Create a `FileInputStream` to read from the `data.bin` file.
        - Read bytes one by one or into a byte array.
        - Print the read bytes.
        - Close the stream.
4. **Character Streams (`FileReader`, `FileWriter`, `BufferedReader`, `BufferedWriter`):**
    - **Writing Characters:**
        - Create a `FileWriter` to write text to a file (e.g., `text.txt`).
        - Use `write()` to write strings.
        - (Optional: Use `BufferedWriter` for efficiency with `newLine()`).
        - Close the stream.
    - **Reading Characters:**
        - Create a `FileReader` to read text from `text.txt`.
        - Use `read()` to read characters one by one or into a char array.
        - (Optional: Use `BufferedReader` with `readLine()` for reading lines).
        - Print the read characters/lines.
        - Close the stream.

5. **Error Handling for Streams:**
    - Always use `try-catch-finally` blocks to handle `IOException` and ensure streams are closed in the `finally` block.
6. **Compilation and Execution:**
    - Compile and run the program. Check the created files (`data.bin`, `text.txt`) in your project directory.

Source Code

```
// Example: Layout Managers, Byte and Character Streams

import javax.swing.*;
import java.awt.*;
import java.io.*;

public class LayoutAndStreamsDemo extends JFrame {

    public LayoutAndStreamsDemo() {
        setTitle("Layout Managers & Streams Demo");
        setSize(800, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null); // Center the window

        // --- Layout Managers Demo ---
        JPanel layoutPanel = new JPanel();
        layoutPanel.setLayout(new GridLayout(3, 1)); // Grid for different
layout sections

        // FlowLayout Demo
        JPanel flowPanel = new JPanel();
        flowPanel.setBorder(BorderFactory.createTitledBorder("FlowLayout"));
        flowPanel.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10)); // Left
aligned, 10px hgap/vgap
        flowPanel.add(new JButton("Flow 1"));
        flowPanel.add(new JButton("Flow 2"));
        flowPanel.add(new JButton("Flow 3"));
        layoutPanel.add(flowPanel);

        // BorderLayout Demo
        JPanel borderPanel = new JPanel();
        borderPanel.setBorder(BorderFactory.createTitledBorder("BorderLayout"));
        borderPanel.setLayout(new BorderLayout(5, 5)); // 5px hgap/vgap
        borderPanel.add(new JButton("North"), BorderLayout.NORTH);
        borderPanel.add(new JButton("South"), BorderLayout.SOUTH);
        borderPanel.add(new JButton("East"), BorderLayout.EAST);
        borderPanel.add(new JButton("West"), BorderLayout.WEST);
        borderPanel.add(new JButton("Center"), BorderLayout.CENTER);
        layoutPanel.add(borderPanel);

        // GridLayout Demo
        JPanel gridPanel = new JPanel();
        gridPanel.setBorder(BorderFactory.createTitledBorder("GridLayout
(2x3)"));
        gridPanel.setLayout(new GridLayout(2, 3, 5, 5)); // 2 rows, 3 columns,
5px hgap/vgap
        gridPanel.add(new JButton("Grid 1"));
        gridPanel.add(new JButton("Grid 2"));
        gridPanel.add(new JButton("Grid 3"));
        gridPanel.add(new JButton("Grid 4"));
        gridPanel.add(new JButton("Grid 5"));
        gridPanel.add(new JButton("Grid 6"));
        layoutPanel.add(gridPanel);

        add(layoutPanel, BorderLayout.CENTER); // Add the panel with layouts to
the frame
```

```
        // --- File I/O Demo (Console Output) ---
        // This part will run and print to console, not directly on GUI for
simplicity
        performFileIO();

        setVisible(true);
    }

    private void performFileIO() {
        System.out.println("\n--- File I/O Demo ---");

        // --- Byte Streams ---
        String byteFileName = "data.bin";
        try (FileOutputStream fos = new FileOutputStream(byteFileName)) {
            System.out.println("Writing bytes to " + byteFileName + "...");
            String data = "Hello Bytes!";
            fos.write(data.getBytes()); // Write string as bytes
            fos.write(123); // Write a single byte
            System.out.println("Bytes written successfully.");
        } catch (IOException e) {
            System.err.println("Error writing bytes: " + e.getMessage());
        }

        try (FileInputStream fis = new FileInputStream(byteFileName)) {
            System.out.println("Reading bytes from " + byteFileName + "...");
            int byteRead;
            StringBuilder readData = new StringBuilder();
            while ((byteRead = fis.read()) != -1) {
                readData.append((char) byteRead); // Convert byte to char for
printing
            }
            System.out.println("Read bytes: " + readData.toString());
            System.out.println("Bytes read successfully.");
        } catch (IOException e) {
            System.err.println("Error reading bytes: " + e.getMessage());
        }

        // --- Character Streams ---
        String charFileName = "text.txt";
        try (FileWriter fw = new FileWriter(charFileName);
             BufferedWriter bw = new BufferedWriter(fw)) { // Using
BufferedWriter for efficiency
            System.out.println("\nWriting characters to " + charFileName +
"...");
            bw.write("This is the first line of text.");
            bw.newLine(); // Write a new line character
            bw.write("This is the second line.");
            bw.newLine();
            bw.write("Java streams are powerful!");
            System.out.println("Characters written successfully.");
        } catch (IOException e) {
            System.err.println("Error writing characters: " + e.getMessage());
        }

        try (FileReader fr = new FileReader(charFileName);
             BufferedReader br = new BufferedReader(fr)) { // Using
BufferedReader for line-by-line reading
            System.out.println("Reading characters from " + charFileName +
"...");
            String line;
            StringBuilder readText = new StringBuilder();
            while ((line = br.readLine()) != null) {
                readText.append(line).append("\n");
            }
            System.out.println("Read text:\n" + readText.toString());
            System.out.println("Characters read successfully.");
```

```
        } catch (IOException e) {
            System.err.println("Error reading characters: " + e.getMessage());
        }
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            @Override
            public void run() {
                new LayoutAndStreamsDemo();
            }
        });
    }
}
```

Input

No direct user input for this example. The GUI demonstrates layout managers, and the file I/O operations happen automatically on program startup, with results printed to the console.

Expected Output

A GUI window will appear demonstrating `FlowLayout`, `BorderLayout`, and `GridLayout` with buttons arranged within them.

The console output will show:

```
--- File I/O Demo ---
Writing bytes to data.bin...
Bytes written successfully.
Reading bytes from data.bin...
Read bytes: Hello Bytes!{
Bytes read successfully.

Writing characters to text.txt...
Characters written successfully.
Reading characters from text.txt...
Read text:
This is the first line of text.
This is the second line.
Java streams are powerful!

Characters read successfully.
```

(Note: The character `{` in the byte stream output corresponds to the ASCII value 123 that was written.)