SRM Institute of Science and Technology

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

Department of Computer Applications

Circular – 2023-24

B.Sc. CS 3rd Sem

PROGRAMMING IN JAVA (USA23301J)- Lab Manual

Laboratory 1: Basic Java Programs

Title

Program 1: Hello World and Basic Arithmetic

Aim

To write and execute a basic Java program that prints "Hello, World!" and performs simple arithmetic operations.

Procedure

1. Open a text editor (like Notepad, VS Code, or an IDE like IntelliJ/Eclipse).
2. Type the Java source code provided below.
3. Save the file as `HelloWorld.java`.
4. Open a command prompt or terminal.
5. Navigate to the directory where you saved the file.
6. Compile the Java program using the Java compiler: `javac HelloWorld.java`
7. Run the compiled program using the Java Virtual Machine: `java HelloWorld`
8. Observe the output in the console.

Source Code

```
// HelloWorld.java
public class HelloWorld {
    public static void main(String[] args) {
        // Print a simple message
        System.out.println("Hello, World!");

        // Perform basic arithmetic operations
        int num1 = 10;
        int num2 = 5;

        int sum = num1 + num2;
        int difference = num1 - num2;
        int product = num1 * num2;
        int quotient = num1 / num2;

        System.out.println("Sum: " + sum);
        System.out.println("Difference: " + difference);
        System.out.println("Product: " + product);
        System.out.println("Quotient: " + quotient);
    }
}
```

Input

No explicit input is required for this program as values are hardcoded.

Expected Output

```
Hello, World!
Sum: 15
Difference: 5
Product: 50
Quotient: 2
```

Laboratory 2: Operators

Title

Program 2: Demonstrating Various Operators

Aim

To understand and implement different types of operators in Java, including arithmetic, relational, logical, and assignment operators.

Procedure

1. Open a text editor or IDE.
2. Write the Java code for `OperatorDemo.java`.
3. Compile the program: `javac OperatorDemo.java`
4. Run the program: `java OperatorDemo`
5. Analyze the output to observe the effect of each operator.

Source Code

```
// OperatorDemo.java
public class OperatorDemo {
    public static void main(String[] args) {
        int a = 20;
        int b = 10;
        boolean x = true;
        boolean y = false;

        // Arithmetic Operators
        System.out.println("--- Arithmetic Operators ---");
        System.out.println("a + b = " + (a + b));   // Addition
        System.out.println("a - b = " + (a - b));   // Subtraction
        System.out.println("a * b = " + (a * b));   // Multiplication
        System.out.println("a / b = " + (a / b));   // Division
        System.out.println("a % b = " + (a % b));   // Modulus (remainder)

        // Relational Operators
        System.out.println("\n--- Relational Operators ---");
        System.out.println("a > b is " + (a > b));   // Greater than
        System.out.println("a < b is " + (a < b));   // Less than
        System.out.println("a >= b is " + (a >= b)); // Greater than or equal
to
        System.out.println("a <= b is " + (a <= b)); // Less than or equal to
        System.out.println("a == b is " + (a == b)); // Equal to
        System.out.println("a != b is " + (a != b)); // Not equal to

        // Logical Operators
        System.out.println("\n--- Logical Operators ---");
        System.out.println("x && y is " + (x && y)); // Logical AND
        System.out.println("x || y is " + (x || y)); // Logical OR
        System.out.println("!x is " + (!x));         // Logical NOT

        // 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);
```

```
    }
}
```

Input

No explicit input is required.

Expected Output

```
--- Arithmetic Operators ---
a + b = 30
a - b = 10
a * b = 200
a / b = 2
a % b = 0

--- Relational Operators ---
a > b is true
a < b is false
a >= b is true
a <= b is false
a == b is false
a != b is true

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

--- Assignment Operators ---
c = a: 20
c += b: 30
c -= b: 20
```

Laboratory 3: Arrays, Control Statements

Title

Program 3: Array Sum and Finding Max/Min using Control Statements

Aim

To demonstrate the use of arrays to store a collection of elements and control statements (loops and conditionals) to process them, specifically calculating the sum and finding the maximum and minimum values in an array.

Procedure

1. Open a text editor or IDE.
2. Write the Java code for `ArrayOperations.java`.
3. Compile the program: `javac ArrayOperations.java`
4. Run the program: `java ArrayOperations`
5. Verify the sum, maximum, and minimum values printed.

Source Code

```
// ArrayOperations.java
public class ArrayOperations {
    public static void main(String[] args) {
        // Declare and initialize an array of integers
        int[] numbers = {5, 12, 9, 3, 15, 7};

        // Calculate the sum of array elements
        int sum = 0;
        for (int i = 0; i < numbers.length; i++) {
            sum += numbers[i];
        }
        System.out.println("Sum of array elements: " + sum);

        // Find the maximum and minimum elements in the array
        if (numbers.length == 0) {
            System.out.println("Array is empty.");
            return; // Exit if array is empty
        }

        int max = numbers[0]; // Assume first element is max
        int min = numbers[0]; // Assume first element is min

        for (int i = 1; i < numbers.length; i++) {
            if (numbers[i] > max) {
                max = numbers[i]; // Update max if current element is greater
            }
            if (numbers[i] < min) {
                min = numbers[i]; // Update min if current element is smaller
            }
        }
        System.out.println("Maximum element: " + max);
        System.out.println("Minimum element: " + min);

        // Demonstrate enhanced for loop (for-each loop)
        System.out.println("\nElements in the array:");
        for (int num : numbers) {
            System.out.print(num + " ");
        }
```

```
        System.out.println();
    }
}
```

Input

No explicit input is required; the array elements are hardcoded.

Expected Output

```
Sum of array elements: 51
Maximum element: 15
Minimum element: 3

Elements in the array:
5 12 9 3 15 7
```

Laboratory 4: Classes and Objects

Title

Program 4: Creating and Using Classes and Objects

Aim

To understand the concept of Object-Oriented Programming (OOP) by defining a class, creating objects from that class, and accessing their attributes and methods.

Procedure

1. Open a text editor or IDE.
2. Create two files: `Car.java` for the class definition and `CarDemo.java` for the main program.
3. Type the code for both files.
4. Compile both files: `javac Car.java CarDemo.java` (or `javac *.java`)
5. Run the main program: `java CarDemo`
6. Observe how object attributes are set and methods are called.

Source Code

```
// Car.java
class Car {
    // Attributes (instance variables)
    String make;
    String model;
    int year;
    String color;

    // Constructor to initialize the car object
    public Car(String make, String model, int year, String color) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.color = color;
    }

    // Method to display car details
    public void displayCarDetails() {
        System.out.println("Make: " + make);
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
        System.out.println("Color: " + color);
    }

    // Method to simulate driving
    public void drive() {
        System.out.println(make + " " + model + " is driving...");
    }
}

// CarDemo.java
public class CarDemo {
    public static void main(String[] args) {
        // Create objects (instances) of the Car class
        Car car1 = new Car("Toyota", "Camry", 2020, "Blue");
        Car car2 = new Car("Honda", "Civic", 2022, "Red");
```

```
        // Access attributes and call methods for car1
        System.out.println("--- Car 1 Details ---");
        car1.displayCarDetails();
        car1.drive();

        System.out.println("\n--- Car 2 Details ---");
        // Access attributes directly (though methods are preferred)
        System.out.println("Make: " + car2.make);
        System.out.println("Model: " + car2.model);
        car2.drive();
    }
}
```

Input

No explicit input is required.

Expected Output

```
--- Car 1 Details ---
Make: Toyota
Model: Camry
Year: 2020
Color: Blue
Toyota Camry is driving...

--- Car 2 Details ---
Make: Honda
Model: Civic
Honda Civic is driving...
```

Laboratory 5: Overloading Methods and Constructors

Title

Program 5: Demonstrating Method and Constructor Overloading

Aim

To understand and implement method overloading (multiple methods with the same name but different parameters) and constructor overloading (multiple constructors with different parameters) within a single class.

Procedure

1. Open a text editor or IDE.
2. Write the Java code for `Calculator.java`.
3. Compile the program: `javac Calculator.java`
4. Run the program: `java Calculator`
5. Observe how different methods/constructors are invoked based on the arguments provided.

Source Code

```
// Calculator.java
class MathOperations {
    // Constructor Overloading
    String operationType;

    public MathOperations() {
        this.operationType = "General Operations";
        System.out.println("Calculator initialized: " + operationType);
    }

    public MathOperations(String type) {
        this.operationType = type;
        System.out.println("Calculator initialized for: " + operationType);
    }

    // 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;
    }

    // Method Overloading: multiply method
    public int multiply(int a, int b) {
        System.out.println("Multiplying two integers:");
        return a * b;
    }
```

```
    public double multiply(double a, double b) {
        System.out.println("Multiplying two doubles:");
        return a * b;
    }
}

public class Calculator {
    public static void main(String[] args) {
        // Demonstrate Constructor Overloading
        MathOperations calc1 = new MathOperations();
        MathOperations calc2 = new MathOperations("Advanced Calculations");

        System.out.println("\n--- Method Overloading Demo ---");

        // Demonstrate Method Overloading for 'add'
        System.out.println("Sum of 5 and 10: " + calc1.add(5, 10));
        System.out.println("Sum of 2.5 and 3.5: " + calc1.add(2.5, 3.5));
        System.out.println("Sum of 1, 2, and 3: " + calc1.add(1, 2, 3));

        // Demonstrate Method Overloading for 'multiply'
        System.out.println("\nProduct of 4 and 6: " + calc1.multiply(4, 6));
        System.out.println("Product of 10.0 and 2.0: " + calc1.multiply(10.0,
2.0));
    }
}
```

Input

No explicit input is required.

Expected Output

```
Calculator initialized: General Operations
Calculator initialized for: Advanced Calculations

--- Method Overloading Demo ---
Adding two integers:
Sum of 5 and 10: 15
Adding two doubles:
Sum of 2.5 and 3.5: 6.0
Adding three integers:
Sum of 1, 2, and 3: 6

Multiplying two integers:
Product of 4 and 6: 24
Multiplying two doubles:
Product of 10.0 and 2.0: 20.0
```

Laboratory 6: String Class, Command Line Arguments

Title

Program 6: String Manipulation and Command Line Arguments

Aim

To explore common methods of the `String` class for text manipulation and to learn how to accept and process command-line arguments passed to a Java program.

Procedure

1. Open a text editor or IDE.
2. Write the Java code for `StringAndArgs.java`.
3. Compile the program: `javac StringAndArgs.java`
4. Run the program, providing arguments after the class name: `java StringAndArgs Hello Java Programming`
5. Experiment with different string methods and command-line arguments.

Source Code

```
// StringAndArgs.java
public class StringAndArgs {
    public static void main(String[] args) {
        // --- String Class Demonstration ---
        System.out.println("--- String Class Demo ---");
        String str1 = "Hello World";
        String str2 = "Java";
        String str3 = "  programming  ";

        System.out.println("Original String 1: \"" + str1 + "\"");
        System.out.println("Length of str1: " + str1.length());
        System.out.println("Character at index 6 in str1: " +
str1.charAt(6));
        System.out.println("Concatenated string (str1 + str2): " +
str1.concat(" " + str2));
        System.out.println("Does str1 contain 'World'? " +
str1.contains("World"));
        System.out.println("Substring of str1 from index 6: " +
str1.substring(6));
        System.out.println("Uppercase str1: " + str1.toUpperCase());
        System.out.println("Lowercase str1: " + str1.toLowerCase());
        System.out.println("Trimmed str3: \"" + str3.trim() + "\"");
        System.out.println("str1 equals 'hello world'? " + str1.equals("hello
world"));
        System.out.println("str1 equals 'hello world' (ignore case)? " +
str1.equalsIgnoreCase("hello world"));
        System.out.println("Replaced 'o' with 'X' in str1: " +
str1.replace('o', 'X'));


        // --- Command Line Arguments Demonstration ---
        System.out.println("\n--- Command Line Arguments Demo ---");

        if (args.length > 0) {
            System.out.println("Number of command line arguments: " +
args.length);
            System.out.println("Arguments received:");
            for (int i = 0; i < args.length; i++) {
                System.out.println("Argument " + (i + 1) + ": " + args[i]);
```

```
            }
            // Example: Concatenate first two arguments if available
            if (args.length >= 2) {
                System.out.println("Concatenating first two arguments: " +
args[0] + args[1]);
            }
        } else {
            System.out.println("No command line arguments provided.");
        }
    }
}
```

Input

Example input when running the program from the command line: `java StringAndArgs Apple Banana Cherry 123`

Expected Output

For the example input `java StringAndArgs Apple Banana Cherry 123`:

```
--- String Class Demo ---
Original String 1: "Hello World"
Length of str1: 11
Character at index 6 in str1: W
Concatenated string (str1 + str2): Hello World Java
Does str1 contain 'World'? true
Substring of str1 from index 6: World
Uppercase str1: HELLO WORLD
Lowercase str1: hello world
Trimmed str3: "programming"
str1 equals 'hello world'? false
str1 equals 'hello world' (ignore case)? true
Replaced 'o' with 'X' in str1: HellX WXrld

--- Command Line Arguments Demo ---
Number of command line arguments: 4
Arguments received:
Argument 1: Apple
Argument 2: Banana
Argument 3: Cherry
Argument 4: 123
Concatenating first two arguments: AppleBanana
```

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

Title

Program 7: Demonstrating Inheritance, Method Overriding, and Abstract Classes

Aim

To understand and implement key OOP concepts: inheritance (creating a subclass from a superclass), method overriding (redefining a superclass method in a subclass), and abstract classes/methods (defining a blueprint for subclasses).

Procedure

1. Open a text editor or IDE.
2. Create three files: `Vehicle.java`, `Car.java`, and `Motorcycle.java`.
3. Type the code for all three files.
4. Compile all files: `javac Vehicle.java Car.java Motorcycle.java` (or `javac *.java`)
5. Run the program: `java Vehicle` (assuming `Vehicle` contains the `main` method for demonstration).
6. Observe how methods are inherited and overridden.

Source Code

```
// Vehicle.java (Abstract Superclass)
abstract class Vehicle {
    String brand;

    public Vehicle(String brand) {
        this.brand = brand;
    }

    // Abstract method (must be implemented by concrete subclasses)
    public abstract void start();

    // Concrete method
    public void stop() {
        System.out.println(brand + " vehicle stopped.");
    }

    public void displayBrand() {
        System.out.println("Vehicle Brand: " + brand);
    }
}

// Car.java (Subclass inheriting from Vehicle)
class Car extends Vehicle {
    int numberOfDoors;

    public Car(String brand, int numberOfDoors) {
        super(brand); // Call superclass constructor
        this.numberOfDoors = numberOfDoors;
    }

    @Override // Annotation indicating method overriding
    public void start() {
        System.out.println(brand + " Car started with a key.");
```

```
    }

    public void accelerate() {
        System.out.println(brand + " Car is accelerating.");
    }

    public void displayDoors() {
        System.out.println("Number of doors: " + numberOfDoors);
    }
}

// Motorcycle.java (Another Subclass inheriting from Vehicle)
class Motorcycle extends Vehicle {
    boolean hasSidecar;

    public Motorcycle(String brand, boolean hasSidecar) {
        super(brand);
        this.hasSidecar = hasSidecar;
    }

    @Override
    public void start() {
        System.out.println(brand + " Motorcycle started with a kick-start.");
    }

    public void wheelie() {
        System.out.println(brand + " Motorcycle is doing a wheelie!");
    }
}

// Main program to demonstrate
public class VehicleDemo { // Renamed from Vehicle to VehicleDemo to avoid
conflict
    public static void main(String[] args) {
        // Cannot instantiate an abstract class directly
        // Vehicle myVehicle = new Vehicle("Generic"); // This would cause a
compile-time error

        Car myCar = new Car("Toyota", 4);
        Motorcycle myMotorcycle = new Motorcycle("Harley", true);

        System.out.println("--- Car Operations ---");
        myCar.displayBrand();
        myCar.displayDoors();
        myCar.start();      // Calls overridden method in Car class
        myCar.accelerate(); // Calls Car-specific method
        myCar.stop();       // Calls inherited method from Vehicle class

        System.out.println("\n--- Motorcycle Operations ---");
        myMotorcycle.displayBrand();
        System.out.println("Has Sidecar: " + myMotorcycle.hasSidecar);
        myMotorcycle.start();   // Calls overridden method in Motorcycle
class
        myMotorcycle.wheelie(); // Calls Motorcycle-specific method
        myMotorcycle.stop();    // Calls inherited method from Vehicle class
    }
}
```

Input

No explicit input is required.

Expected Output

```
--- Car Operations ---
Vehicle Brand: Toyota
Number of doors: 4
Toyota Car started with a key.
Toyota Car is accelerating.
Toyota vehicle stopped.

--- Motorcycle Operations ---
Vehicle Brand: Harley
Has Sidecar: true
Harley Motorcycle started with a kick-start.
Harley Motorcycle is doing a wheelie!
Harley vehicle stopped.
```

Laboratory 8: Packages and Interfaces

Title

Program 8: Creating and Using Packages and Interfaces

Aim

To understand how to organize classes into packages for better modularity and to define and implement interfaces for achieving abstraction and multiple inheritance-like behavior in Java.

Procedure

1. Create a directory structure: `mycompany/utility/`.
2. Open a text editor or IDE.
3. Create `Printable.java` inside `mycompany/utility/` for the interface.
4. Create `Document.java` inside `mycompany/utility/` for the class implementing the interface.
5. Create `PackageAndInterfaceDemo.java` in the parent directory (e.g., `src/`).
6. Compile from the parent directory: `javac mycompany/utility/*.java PackageAndInterfaceDemo.java`
7. Run the main program from the parent directory: `java PackageAndInterfaceDemo`
8. Observe how classes from a package are imported and how interface methods are implemented.

Source Code

```
// mycompany/utility/Printable.java (Interface)
package mycompany.utility;

public interface Printable {
    void printDetails(); // Abstract method
    double getVersion(); // Another abstract method
}

// mycompany/utility/Document.java (Class implementing the interface)
package mycompany.utility;

public class Document implements Printable {
    private String title;
    private String author;
    private double version;

    public Document(String title, String author, double version) {
        this.title = title;
        this.author = author;
        this.version = version;
    }

    @Override
    public void printDetails() {
        System.out.println("--- Document Details ---");
        System.out.println("Title: " + title);
        System.out.println("Author: " + author);
        System.out.println("Version: " + version);
    }

    @Override
    public double getVersion() {
```

```
        return version;
    }

    public void editDocument(String newTitle) {
        this.title = newTitle;
        System.out.println("Document title updated to: " + newTitle);
    }
}

// PackageAndInterfaceDemo.java (Main program)
// This file should be in the directory above 'mycompany'
import mycompany.utility.Document;
import mycompany.utility.Printable;

public class PackageAndInterfaceDemo {
    public static void main(String[] args) {
        System.out.println("--- Demonstrating Packages and Interfaces ---");

        // Create an object of the Document class from the
'mycompany.utility' package
        Document report = new Document("Annual Report 2023", "John Doe",
1.0);

        // Call methods defined in the Document class
        report.printDetails();
        report.editDocument("Annual Report 2024 (Draft)");
        report.printDetails();

        // Demonstrate polymorphism using the interface reference
        Printable printableItem = new Document("Meeting Minutes", "Jane
Smith", 1.1);
        System.out.println("\n--- Using Interface Reference ---");
        printableItem.printDetails();
        System.out.println("Version from interface: " +
printableItem.getVersion());

        // Cannot call editDocument directly on printableItem because it's
not in the Printable interface
        // printableItem.editDocument("New Title"); // This would be a
compile-time error
    }
}
```

Input

No explicit input is required.

Expected Output

```
--- Demonstrating Packages and Interfaces ---
--- Document Details ---
Title: Annual Report 2023
Author: John Doe
Version: 1.0
Document title updated to: Annual Report 2024 (Draft)
--- Document Details ---
Title: Annual Report 2024 (Draft)
Author: John Doe
Version: 1.0

--- Using Interface Reference ---
--- Document Details ---
Title: Meeting Minutes
Author: Jane Smith
```

Version: 1.1
Version from interface: 1.1

Laboratory 9: Exception Handling

Title

Program 9: Demonstrating Exception Handling

Aim

To understand how to handle runtime errors (exceptions) gracefully in Java using `try`, `catch`, `finally` blocks, and to demonstrate how to throw and create custom exceptions.

Procedure

1. Open a text editor or IDE.
2. Write the Java code for `ExceptionHandlingDemo.java`.
3. Compile the program: `javac ExceptionHandlingDemo.java`
4. Run the program: `java ExceptionHandlingDemo`
5. Observe how different exceptions are caught and handled, and how the `finally` block always executes.

Source Code

```
// CustomException.java (Optional: Define a custom exception class)
class MyCustomException extends Exception {
    public MyCustomException(String message) {
        super(message);
    }
}

// ExceptionHandlingDemo.java
public class ExceptionHandlingDemo {

    // Method that might throw an ArithmeticException
    public static void divide(int numerator, int denominator) {
        try {
            System.out.println("\nAttempting division...");
            int result = numerator / denominator;
            System.out.println("Result of division: " + result);
        } catch (ArithmeticException e) {
            System.err.println("Error: Cannot divide by zero! " +
e.getMessage());
        } finally {
            System.out.println("Division attempt finished (finally block
always executes).");
        }
    }

    // Method that might throw an ArrayIndexOutOfBoundsException
    public static void accessArray(int[] arr, int index) {
        try {
            System.out.println("\nAttempting array access...");
            System.out.println("Value at index " + index + ": " +
arr[index]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.err.println("Error: Array index out of bounds! " +
e.getMessage());
        } finally {
            System.out.println("Array access attempt finished.");
        }
    }
```

```
    // Method that throws a custom exception
    public static void validateAge(int age) throws MyCustomException {
        if (age < 0 || age > 120) {
            throw new MyCustomException("Invalid Age: " + age + ". Age must
be between 0 and 120.");
        }
        System.out.println("\nAge " + age + " is valid.");
    }

    public static void main(String[] args) {
        // Case 1: Arithmetic Exception
        divide(10, 2);
        divide(10, 0); // This will cause an ArithmeticException

        // Case 2: Array Index Out Of Bounds Exception
        int[] numbers = {1, 2, 3};
        accessArray(numbers, 1);
        accessArray(numbers, 5); // This will cause an
ArrayIndexOutOfBoundsException

        // Case 3: Custom Exception
        try {
            validateAge(30);
            validateAge(-5); // This will throw MyCustomException
        } catch (MyCustomException e) {
            System.err.println("Caught custom exception: " + e.getMessage());
        }

        System.out.println("\nProgram continues after exception handling.");
    }
}
```

Input

No explicit input is required.

Expected Output

```
Attempting division...
Result of division: 5
Division attempt finished (finally block always executes).

Attempting division...
Error: Cannot divide by zero! / by zero
Division attempt finished (finally block always executes).

Attempting array access...
Value at index 1: 2
Array access attempt finished.

Attempting array access...
Error: Array index out of bounds! Index 5 out of bounds for length 3
Array access attempt finished.

Age 30 is valid.

Caught custom exception: Invalid Age: -5. Age must be between 0 and 120.

Program continues after exception handling.
```

Laboratory 10: Multithreading

Title

Program 10: Implementing Multithreading

Aim

To understand and implement multithreading in Java using both the `Thread` class and the `Runnable` interface, demonstrating concurrent execution of tasks.

Procedure

1. Open a text editor or IDE.
2. Create two files: `MyThread.java` and `MyRunnable.java`.
3. Create `MultithreadingDemo.java` for the main program.
4. Compile all files: `javac MyThread.java MyRunnable.java MultithreadingDemo.java` (or `javac *.java`)
5. Run the main program: `java MultithreadingDemo`
6. Observe the interleaved output, indicating concurrent execution.

Source Code

```
// MyThread.java (Extending Thread class)
class MyThread extends Thread {
    private String threadName;
    private int iterations;

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

    @Override
    public void run() {
        try {
            for (int i = 1; i <= iterations; i++) {
                System.out.println("Thread: " + threadName + ", Count: " +
i);
                // Pause for a short period to demonstrate interleaving
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        System.out.println("Thread " + threadName + " exiting.");
    }
}

// MyRunnable.java (Implementing Runnable interface)
class MyRunnable implements Runnable {
    private String threadName;
    private int iterations;

    public MyRunnable(String name, int iterations) {
        this.threadName = name;
        this.iterations = iterations;
        System.out.println("Creating " + threadName);
    }
```

```
    @Override
    public void run() {
        try {
            for (int i = 1; i <= iterations; i++) {
                System.out.println("Runnable: " + threadName + ", Count: " +
i);
                Thread.sleep(70); // Different sleep time to show more
interleaving
            }
        } catch (InterruptedException e) {
            System.out.println("Runnable " + threadName + " interrupted.");
        }
        System.out.println("Runnable " + threadName + " exiting.");
    }
}

// MultithreadingDemo.java (Main program)
public class MultithreadingDemo {
    public static void main(String[] args) {
        System.out.println("--- Multithreading Demo ---");

        // Create and start a thread by extending Thread class
        MyThread thread1 = new MyThread("Thread-A", 5);
        thread1.start(); // Invokes the run() method

        // Create and start a thread by implementing Runnable interface
        MyRunnable runnable1 = new MyRunnable("Runnable-B", 4);
        Thread thread2 = new Thread(runnable1);
        thread2.start(); // Invokes the run() method of the Runnable object

        // Main thread continues execution
        try {
            for (int i = 1; i <= 3; i++) {
                System.out.println("Main Thread, Count: " + i);
                Thread.sleep(100);
            }
        } catch (InterruptedException e) {
            System.out.println("Main Thread interrupted.");
        }
        System.out.println("Main Thread exiting.");
    }
}
```

Input

No explicit input is required.

Expected Output

The output will be interleaved and might vary slightly with each run due to the nature of multithreading. Here's a possible example:

```
--- Multithreading Demo ---
Creating Thread-A
Creating Runnable-B
Thread: Thread-A, Count: 1
Runnable: Runnable-B, Count: 1
Main Thread, Count: 1
Thread: Thread-A, Count: 2
Runnable: Runnable-B, Count: 2
Main Thread, Count: 2
Thread: Thread-A, Count: 3
```

```
Runnable: Runnable-B, Count: 3
Main Thread, Count: 3
Thread: Thread-A, Count: 4
Runnable: Runnable-B, Count: 4
Runnable Runnable-B exiting.
Thread: Thread-A, Count: 5
Thread Thread-A exiting.
Main Thread exiting.
```

Laboratory 11: Legacy Classes and Interfaces

Title

Program 11: Using Legacy Collection Classes (Vector and Hashtable)

Aim

To understand and demonstrate the use of legacy collection classes like `Vector` and `Hashtable`, which were part of the original Java Collections Framework, noting their thread-safe nature.

Procedure

1. Open a text editor or IDE.
2. Write the Java code for `LegacyCollectionsDemo.java`.
3. Compile the program: `javac LegacyCollectionsDemo.java`
4. Run the program: `java LegacyCollectionsDemo`
5. Observe how elements are added, accessed, and removed from `Vector` and `Hashtable`.

Source Code

```
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration; // Used with legacy collections

public class LegacyCollectionsDemo {
    public static void main(String[] args) {
        System.out.println("--- Legacy Collections Demo ---");

        // --- Vector Demonstration ---
        System.out.println("\n--- Vector (Dynamic Array) ---");
        Vector<String> names = new Vector<>(); // Vector for Strings

        // Add elements
        names.add("Alice");
        names.add("Bob");
        names.addElement("Charlie"); // Legacy method
        names.insertElementAt("David", 1); // Insert at specific index

        System.out.println("Vector after additions: " + names);
        System.out.println("Size of Vector: " + names.size());
        System.out.println("Element at index 2: " + names.elementAt(2)); //
Legacy method

        // Iterate using Enumeration (legacy way)
        System.out.println("Iterating Vector using Enumeration:");
        Enumeration<String> en = names.elements();
        while (en.hasMoreElements()) {
            System.out.println("  " + en.nextElement());
        }

        // Remove elements
        names.remove("Bob");
        System.out.println("Vector after removing 'Bob': " + names);
        names.removeElementAt(0); // Remove "Alice" (now at index 0)
        System.out.println("Vector after removing element at index 0: " +
names);


        // --- Hashtable Demonstration ---
        System.out.println("\n--- Hashtable (Key-Value Map) ---");
```

```
        Hashtable<Integer, String> students = new Hashtable<>(); // Hashtable
for Integer keys, String values

        // Add key-value pairs
        students.put(101, "John");
        students.put(102, "Maria");
        students.put(103, "Peter");

        System.out.println("Hashtable after additions: " + students);
        System.out.println("Value for key 102: " + students.get(102));
        System.out.println("Does Hashtable contain value 'John'? " +
students.containsValue("John"));
        System.out.println("Does Hashtable contain key 104? " +
students.containsKey(104));

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

        // Remove a key-value pair
        students.remove(101);
        System.out.println("Hashtable after removing key 101: " + students);
    }
}
```

Input

No explicit input is required.

Expected Output

```
--- Legacy Collections Demo ---

--- Vector (Dynamic Array) ---
Vector after additions: [Alice, David, Bob, Charlie]
Size of Vector: 4
Element at index 2: Bob
Iterating Vector using Enumeration:
  Alice
  David
  Bob
  Charlie
Vector after removing 'Bob': [Alice, David, Charlie]
Vector after removing element at index 0: [David, Charlie]

--- Hashtable (Key-Value Map) ---
Hashtable after additions: {103=Peter, 102=Maria, 101=John}
Value for key 102: Maria
Does Hashtable contain value 'John'? true
Does Hashtable contain key 104? false
Iterating Hashtable keys using Enumeration:
  Key: 103, Value: Peter
  Key: 102, Value: Maria
  Key: 101, Value: John
Hashtable after removing key 101: {103=Peter, 102=Maria}
```

Laboratory 12: Utility Classes

Title

Program 12: Using Modern Utility Classes (ArrayList and HashMap)

Aim

To demonstrate the use of modern utility classes from the Java Collections Framework, specifically `ArrayList` for dynamic arrays and `HashMap` for key-value pairs, highlighting their flexibility and common usage.

Procedure

1. Open a text editor or IDE.
2. Write the Java code for `UtilityClassesDemo.java`.
3. Compile the program: `javac UtilityClassesDemo.java`
4. Run the program: `java UtilityClassesDemo`
5. Observe how elements are managed in `ArrayList` and `HashMap`.

Source Code

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

public class UtilityClassesDemo {
    public static void main(String[] args) {
        System.out.println("--- Modern Utility Classes Demo ---");

        // --- ArrayList Demonstration ---
        System.out.println("\n--- ArrayList (Dynamic List) ---");
        ArrayList<String> fruits = new ArrayList<>();

        // Add elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        fruits.add(1, "Orange"); // Add at specific index

        System.out.println("ArrayList after additions: " + fruits);
        System.out.println("Size of ArrayList: " + fruits.size());
        System.out.println("Element at index 2: " + fruits.get(2));

        // Check if an element exists
        System.out.println("Does ArrayList contain 'Banana'? " +
fruits.contains("Banana"));

        // Remove elements
        fruits.remove("Cherry");
        System.out.println("ArrayList after removing 'Cherry': " + fruits);
        fruits.remove(0); // Remove element at index 0 ("Apple")
        System.out.println("ArrayList after removing element at index 0: " +
fruits);

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

```
        }


        // --- HashMap Demonstration ---
        System.out.println("\n--- HashMap (Key-Value Map) ---");
        HashMap<String, Integer> studentScores = new HashMap<>();

        // Add key-value pairs
        studentScores.put("Alice", 95);
        studentScores.put("Bob", 88);
        studentScores.put("Charlie", 92);

        System.out.println("HashMap after additions: " + studentScores);
        System.out.println("Bob's score: " + studentScores.get("Bob"));
        System.out.println("Does HashMap contain key 'David'? " +
studentScores.containsKey("David"));

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

        // Iterate through HashMap
        System.out.println("Iterating HashMap entries:");
        for (String name : studentScores.keySet()) {
            System.out.println("  " + name + ": " + studentScores.get(name));
        }

        // Remove a key-value pair
        studentScores.remove("Bob");
        System.out.println("HashMap after removing Bob: " + studentScores);


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

        Random random = new Random();
        System.out.println("Random integer: " + random.nextInt(100)); //
Random number between 0 (inclusive) and 100 (exclusive)
        System.out.println("Random double: " + random.nextDouble()); //
Random double between 0.0 (inclusive) and 1.0 (exclusive)
    }
}
```

Input

No explicit input is required.

Expected Output

The output for Date and Random will vary. Other parts will be consistent.

```
--- Modern Utility Classes Demo ---

--- ArrayList (Dynamic List) ---
ArrayList after additions: [Apple, Orange, Banana, Cherry]
Size of ArrayList: 4
Element at index 2: Banana
Does ArrayList contain 'Banana'? true
ArrayList after removing 'Cherry': [Apple, Orange, Banana]
ArrayList after removing element at index 0: [Orange, Banana]
```

```
Iterating ArrayList:
  Orange
  Banana

--- HashMap (Key-Value Map) ---
HashMap after additions: {Alice=95, Bob=88, Charlie=92}
Bob's score: 88
Does HashMap contain key 'David'? false
HashMap after updating Alice's score: {Alice=98, Bob=88, Charlie=92}
Iterating HashMap entries:
  Alice: 98
  Bob: 88
  Charlie: 92
HashMap after removing Bob: {Alice=98, Charlie=92}

--- Date and Random Classes ---
Current Date and Time: [Current date and time will appear here, e.g., Wed May
21 14:30:00 IST 2025]
Random integer: [A random integer between 0-99]
Random double: [A random double between 0.0 and 1.0]
```

Laboratory 13: Event Handling

Title

Program 13: Simple AWT Event Handling (Button Click)

Aim

To understand the basics of event handling in Java's Abstract Window Toolkit (AWT) by creating a simple GUI with a button and responding to its click event.

Procedure

1. Open a text editor or IDE.
2. Write the Java code for `SimpleAWTEvent.java`.
3. Compile the program: `javac SimpleAWTEvent.java`
4. Run the program: `java SimpleAWTEvent`
5. A small window will appear. Click the button and observe the message printed to the console.

Source Code

```
import java.awt.*;
import java.awt.event.*; // Import event handling classes

public class SimpleAWTEvent extends Frame implements ActionListener {

    private Button clickButton;
    private Label messageLabel;
    private int clickCount = 0;

    public SimpleAWTEvent() {
        // Set frame properties
        setTitle("AWT Event Demo");
        setSize(400, 200);
        setLayout(new FlowLayout()); // Use FlowLayout for simple arrangement

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

        // Register the button with an ActionListener
        // 'this' refers to the current class (SimpleAWTEvent),
        // which implements ActionListener and thus has an actionPerformed
method.
        clickButton.addActionListener(this);

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

        // Add a WindowListener to handle closing the frame
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                System.exit(0); // Terminate the application
            }
        });

        // 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) {
        if (e.getSource() == clickButton) {
            clickCount++;
            messageLabel.setText("Button clicked " + clickCount + "
time(s).");
            System.out.println("Button was clicked!"); // Also print to
console
        }
    }

    public static void main(String[] args) {
        new SimpleAWTEvent(); // Create an instance of the GUI
    }
}
```

Input

User interaction: Clicking the "Click Me!" button in the GUI window.

Expected Output

A GUI window titled "AWT Event Demo" will appear with a "Click Me!" button and a label. When the button is clicked:

1. The label text will update to "Button clicked X time(s)." (where X is the click count).
2. "Button was clicked!" will be printed to the console for each click.

Laboratory 14: AWT Controls

Title

Program 14: Demonstrating Various AWT Controls

Aim

To create a simple AWT application that showcases various common AWT controls such as Label, TextField, Button, Checkbox, Choice (Dropdown), and List.

Procedure

1. Open a text editor or IDE.
2. Write the Java code for `AWTControlsDemo.java`.
3. Compile the program: `javac AWTControlsDemo.java`
4. Run the program: `java AWTControlsDemo`
5. Interact with the different controls in the displayed window and observe their behavior.

Source Code

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

public class AWTControlsDemo extends Frame implements ActionListener,
ItemListener {

    private Label nameLabel, genderLabel, courseLabel, selectedItemsLabel;
    private TextField nameTextField;
    private Button submitButton;
    private Checkbox maleCheckbox, femaleCheckbox;
    private CheckboxGroup genderGroup; // For radio button behavior
    private Choice courseChoice; // Dropdown
    private List hobbiesList; // Multiple selection list

    public AWTControlsDemo() {
        setTitle("AWT Controls Demo");
        setSize(500, 400);
        setLayout(new FlowLayout(FlowLayout.LEFT, 10, 10)); // FlowLayout
with left alignment and gaps

        // --- Label and TextField ---
        nameLabel = new Label("Name:");
        nameTextField = new TextField(20); // 20 columns wide
        add(nameLabel);
        add(nameTextField);

        // --- Checkbox (Radio Buttons using CheckboxGroup) ---
        genderLabel = new Label("Gender:");
        genderGroup = new CheckboxGroup();
        maleCheckbox = new Checkbox("Male", genderGroup, false);
        femaleCheckbox = new Checkbox("Female", genderGroup, false);
        maleCheckbox.addItemListener(this); // Listen for state changes
        femaleCheckbox.addItemListener(this);
        add(genderLabel);
        add(maleCheckbox);
        add(femaleCheckbox);

        // --- Choice (Dropdown) ---
        courseLabel = new Label("Select Course:");
        courseChoice = new Choice();
```

```
        courseChoice.add("Computer Science");
        courseChoice.add("Electrical Engineering");
        courseChoice.add("Mechanical Engineering");
        courseChoice.addItemListener(this); // Listen for selection changes
        add(courseLabel);
        add(courseChoice);

        // --- List (Multiple Selection) ---
        Label hobbiesLabel = new Label("Select Hobbies (Ctrl+Click for
multiple):");
        hobbiesList = new List(4, true); // 4 visible rows, true for multiple
selection
        hobbiesList.add("Reading");
        hobbiesList.add("Sports");
        hobbiesList.add("Gaming");
        hobbiesList.add("Music");
        hobbiesList.add("Traveling");
        hobbiesList.addItemListener(this); // Listen for selection changes
        add(hobbiesLabel);
        add(hobbiesList);

        // --- Button ---
        submitButton = new Button("Submit");
        submitButton.addActionListener(this); // Listen for button clicks
        add(submitButton);

        // --- Label to display selected items ---
        selectedItemsLabel = new Label("Selected: ");
        add(selectedItemsLabel);


        // Add a WindowListener to handle closing the frame
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        setVisible(true);
    }

    // ActionListener for Button
    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == submitButton) {
            String name = nameTextField.getText();
            String gender = (genderGroup.getSelectedCheckbox() != null) ?
                            genderGroup.getSelectedCheckbox().getLabel() :
"Not selected";
            String course = courseChoice.getSelectedItem();

            StringBuilder hobbies = new StringBuilder();
            String[] selectedHobbies = hobbiesList.getSelectedItems();
            if (selectedHobbies.length > 0) {
                for (String hobby : selectedHobbies) {
                    hobbies.append(hobby).append(", ");
                }
                hobbies.setLength(hobbies.length() - 2); // Remove trailing
comma and space
            } else {
                hobbies.append("None");
            }

            String summary = "Name: " + name + "\n" +
                             "Gender: " + gender + "\n" +
```

```
                                "Course: " + course + "\n" +
                                "Hobbies: " + hobbies.toString();

            // Display summary in a dialog or console
            System.out.println("\n--- Submission Summary ---");
            System.out.println(summary);
            selectedItemsLabel.setText("Submitted! Check console for
details.");
        }
    }

    // ItemListener for Checkbox, Choice, List
    @Override
    public void itemStateChanged(ItemEvent e) {
        // This method can be used to update the selectedItemsLabel
dynamically
        // For simplicity, we'll just print to console for now.
        if (e.getSource() == maleCheckbox || e.getSource() == femaleCheckbox)
{
            System.out.println("Gender selected: " +
genderGroup.getSelectedCheckbox().getLabel());
        } else if (e.getSource() == courseChoice) {
            System.out.println("Course selected: " +
courseChoice.getSelectedItem());
        } else if (e.getSource() == hobbiesList) {
            System.out.print("Hobbies selected: ");
            String[] selected = hobbiesList.getSelectedItems();
            for (String item : selected) {
                System.out.print(item + " ");
            }
            System.out.println();
        }
    }

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

Input

User interaction with the GUI:

- Typing text into the name field.
- Selecting Male/Female checkboxes.
- Choosing an item from the course dropdown.
- Selecting one or more hobbies from the list.
- Clicking the "Submit" button.

Expected Output

A GUI window will appear with various AWT controls. Upon interacting with controls (e.g., selecting a gender, course, or hobby), messages will be printed to the console indicating the selection. Upon clicking the "Submit" button, a detailed summary of the entered/selected information will be printed to the console, and the "Selected:" label in the GUI will update.

Example Console Output after interaction and submission:

```
Gender selected: Male
Course selected: Computer Science
Hobbies selected: Reading Sports
```

--- Submission Summary ---
Name: Jane Doe
Gender: Male
Course: Computer Science
Hobbies: Reading, Sports, Gaming

Laboratory 15: Layout Managers, Byte and Character Streams

Title

Program 15: Layout Managers and File I/O (Byte and Character Streams)

Aim

To understand and implement different AWT Layout Managers for arranging GUI components and to demonstrate file input/output operations using both byte streams (for raw data) and character streams (for text data).

Procedure

1. Open a text editor or IDE.
2. Write the Java code for `LayoutAndStreams.java`.
3. Compile the program: `javac LayoutAndStreams.java`
4. Run the program: `java LayoutAndStreams`
5. Observe the GUI layout and check for the created/read files in the same directory as your Java program.

Source Code

```
import java.awt.*;
import java.awt.event.*;
import java.io.*; // Import I/O classes

public class LayoutAndStreams extends Frame {

    public LayoutAndStreams() {
        setTitle("Layout Managers & File I/O Demo");
        setSize(700, 500); // Increased size to accommodate multiple panels
        setLayout(new BorderLayout()); // Main frame uses BorderLayout

        // --- Panel for Layout Managers ---
        Panel layoutPanel = new Panel();
        layoutPanel.setLayout(new GridLayout(1, 2, 10, 10)); // 1 row, 2
columns, with gaps
        layoutPanel.setBackground(Color.LIGHT_GRAY);

        // FlowLayout Demo Panel
        Panel flowPanel = new Panel();
        flowPanel.setLayout(new FlowLayout(FlowLayout.CENTER, 5, 5)); //
Centered, 5px horizontal/vertical gap
        flowPanel.add(new Button("Flow 1"));
        flowPanel.add(new Button("Flow 2"));
        flowPanel.add(new Button("Flow 3"));
        flowPanel.add(new Button("Flow 4"));
        flowPanel.add(new Label("FlowLayout"));
        flowPanel.setBackground(Color.CYAN);
        layoutPanel.add(flowPanel);

        // BorderLayout Demo Panel
        Panel borderPanel = new Panel();
        borderPanel.setLayout(new BorderLayout(5, 5)); // 5px
horizontal/vertical gap
        borderPanel.add(new Button("North"), BorderLayout.NORTH);
        borderPanel.add(new Button("South"), BorderLayout.SOUTH);
        borderPanel.add(new Button("East"), BorderLayout.EAST);
        borderPanel.add(new Button("West"), BorderLayout.WEST);
```

```
        borderPanel.add(new Label("BorderLayout", Label.CENTER),
BorderLayout.CENTER);
        borderPanel.setBackground(Color.MAGENTA);
        layoutPanel.add(borderPanel);

        // Add layout panel to the main frame's NORTH region
        add(layoutPanel, BorderLayout.NORTH);

        // --- Panel for File I/O ---
        Panel fileIOPanel = new Panel();
        fileIOPanel.setLayout(new GridLayout(2, 1, 10, 10)); // 2 rows, 1
column
        fileIOPanel.setBackground(Color.ORANGE);

        TextArea outputArea = new TextArea("File I/O Messages:\n", 10, 50,
TextArea.SCROLLBARS_VERTICAL_ONLY);
        outputArea.setEditable(false);
        fileIOPanel.add(outputArea);

        Button performIOButton = new Button("Perform File I/O");
        performIOButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                outputArea.setText("File I/O Messages:\n"); // Clear previous
messages
                performFileOperations(outputArea);
            }
        });
        fileIOPanel.add(performIOButton);

        // Add file I/O panel to the main frame's CENTER region
        add(fileIOPanel, BorderLayout.CENTER);


        // Add a WindowListener to handle closing the frame
        addWindowListener(new WindowAdapter() {
            @Override
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        setVisible(true);
    }

    private void performFileOperations(TextArea outputArea) {
        // --- Byte Stream Demo (Writing and Reading Bytes) ---
        String byteFileName = "byte_data.dat";
        String byteContent = "Hello from Byte Stream!";
        try (FileOutputStream fos = new FileOutputStream(byteFileName);
             FileInputStream fis = new FileInputStream(byteFileName)) {

            // Write bytes
            fos.write(byteContent.getBytes());
            outputArea.append("Byte Stream: Wrote '" + byteContent + "' to "
+ byteFileName + "\n");

            // Read bytes
            byte[] readBytes = new byte[byteContent.length()];
            fis.read(readBytes);
            String readByteContent = new String(readBytes);
            outputArea.append("Byte Stream: Read '" + readByteContent + "'
from " + byteFileName + "\n");

        } catch (IOException e) {
            outputArea.append("Byte Stream Error: " + e.getMessage() + "\n");
```

```
            e.printStackTrace();
        }

        // --- Character Stream Demo (Writing and Reading Text) ---
        String charFileName = "char_data.txt";
        String charContent = "This is text from Character Stream.";
        try (FileWriter fw = new FileWriter(charFileName);
             FileReader fr = new FileReader(charFileName);
             BufferedReader br = new BufferedReader(fr)) { // Using
BufferedReader for efficient reading

            // Write characters
            fw.write(charContent);
            outputArea.append("\nCharacter Stream: Wrote '" + charContent +
"' to " + charFileName + "\n");
            fw.flush(); // Ensure data is written to file immediately

            // Read characters
            String readCharContent = br.readLine();
            outputArea.append("Character Stream: Read '" + readCharContent +
"' from " + charFileName + "\n");

        } catch (IOException e) {
            outputArea.append("Character Stream Error: " + e.getMessage() +
"\n");
            e.printStackTrace();
        }
    }

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

Input

User interaction: Clicking the "Perform File I/O" button in the GUI.

Expected Output

A GUI window will appear, demonstrating:

- **Top Panel (Layout Managers):** Two sub-panels, one using `FlowLayout` (buttons centered) and another using `BorderLayout` (buttons at North, South, East, West, and a label in Center).
- **Bottom Panel (File I/O):** A `TextArea` to display messages and a "Perform File I/O" button.

When the "Perform File I/O" button is clicked, the `TextArea` will update with messages indicating:

- Successful writing of "Hello from Byte Stream!" to `byte_data.dat` and reading it back.
- Successful writing of "This is text from Character Stream." to `char_data.txt` and reading it back.

Additionally, two files (`byte_data.dat` and `char_data.txt`) will be created in the same directory where the Java program is run.