JAVA PROGRAMMING

EXP1 : class and object

Aim:

The aim of this program is to demonstrate the concept of classes and objects in Java by creating a **Person** class with attributes such as name and age. It creates objects of the **Person** class and displays information about each person.

Algorithm:

1. Start
2. Define a class named **Person** with the following attributes:
    - **name** (String): to store the name of the person.
    - **age** (int): to store the age of the person.
3. Define a constructor in the **Person** class to initialize the **name** and **age** attributes.
4. Define a method named **displayInfo()** in the **Person** class to display the information about the person, including their name and age.
5. Define a public class named **Main**.
6. Inside the **Main** class, define the **main** method as the entry point of the program.
7. Inside the **main** method:
    - Create an object **person1** of the **Person** class with the name "John" and age 30.
    - Call the **displayInfo()** method on **person1** to display information about the person.
    - Create another object **person2** of the **Person** class with the name "Alice" and age 25.
    - Call the **displayInfo()** method on **person2** to display information about the person.
8. End

CODE

```
// Define a class named 'Person'
class Person {
    // Instance variables
    String name;
    int age;

    // Constructor
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Method to display information about the person
    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an object of Person class
        Person person1 = new Person("John", 30);

        // Call displayInfo() method to display information about person1
        person1.displayInfo();

        // Create another object of Person class
        Person person2 = new Person("Alice", 25);

        // Call displayInfo() method to display information about person2
        person2.displayInfo();
    }
}
```

EXP2: Inheritance

Single inheritance

Aim:

The aim of this program is to illustrate single inheritance in Java. It demonstrates how a subclass can inherit attributes and methods from a single superclass.

Algorithm:

```
Start:
Define a class named Parent.
Inside the Parent class:
Define a method parentMethod() to print "This is the parent method."
Define a class named Child that inherits from the Parent class.
Inside the Child class:
Define a method childMethod() to print "This is the child method."
Define a public class named Main.
Inside the Main class, define the main method as the entry point of the program.
Inside the main method:
Create an object childObj of Child class.
Call the parentMethod() and childMethod() using the childObj object.
End
```

```
// Parent class
class Parent {
    void parentMethod() {
        System.out.println("This is the parent method.");
    }
}

// Child class inheriting from Parent class
class Child extends Parent {
    void childMethod() {
        System.out.println("This is the child method.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an object of Child class
        Child childObj = new Child();

        // Call methods from both Parent and Child class
        childObj.parentMethod();
        childObj.childMethod();
    }
}
```

Multiple Inheritance

Aim:

The aim of this program is to demonstrate multilevel inheritance in Java. It illustrates how classes can inherit properties and behaviors from multiple levels of inheritance hierarchy.

Algorithm:

Start:

Define a class named Grandparent.

Inside the Grandparent class:

Define an instance variable grandparentVar.

Define a method grandparentMethod() to print "This is the grandparent method."

Define a class named Parent that inherits from Grandparent.

Inside the Parent class:

Define an instance variable parentVar.

Define a method parentMethod() to print "This is the parent method."

Define a class named Child that inherits from Parent.

Inside the Child class:

Define an instance variable childVar.

Define a method childMethod() to print "This is the child method."

Define a public class named Main.

Inside the Main class, define the main method as the entry point of the program.

Inside the main method:

Create an object childObj of Child class.

Access and print grandparentVar, parentVar, and childVar.

Call grandparentMethod(), parentMethod(), and childMethod() using the childObj object.

End

Here's the Java code implementing the algorithm:

java

Copy code

```java
// Grandparent class
class Grandparent {
    String grandparentVar = "Grandparent variable";

    void grandparentMethod() {
        System.out.println("This is the grandparent method.");
    }
```

```
}

// Parent class inheriting from Grandparent class
class Parent extends Grandparent {
    String parentVar = "Parent variable";

    void parentMethod() {
        System.out.println("This is the parent method.");
    }
}

// Child class inheriting from Parent class
class Child extends Parent {
    String childVar = "Child variable";

    void childMethod() {
        System.out.println("This is the child method.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an object of Child class
        Child childObj = new Child();

        // Access and print variables from all levels of inheritance
        System.out.println("Grandparent variable: " + childObj.grandparentVar);
        System.out.println("Parent variable: " + childObj.parentVar);
        System.out.println("Child variable: " + childObj.childVar);
        // Call methods from all levels of inheritance
        childObj.grandparentMethod();
        childObj.parentMethod();
        childObj.childMethod();
    }
}
```

Hierarchical Inheritance

Aim:

The aim of this program is to illustrate hierarchical inheritance in Java. It shows how multiple classes can inherit properties and behaviors from a single base class.

Algorithm:

Start:

Define a class named Animal.

Inside the Animal class:

Define an instance variable animalType.

Define a method eat() to print "Animal is eating."

Define a class named Dog that inherits from Animal.

Inside the Dog class:

Define a method bark() to print "Dog is barking."

Define a class named Cat that also inherits from Animal.

Inside the Cat class:

Define a method meow() to print "Cat is meowing."

Define a public class named Main.

Inside the Main class, define the main method as the entry point of the program.

Inside the main method:

Create an object dogObj of Dog class.

Create an object catObj of Cat class.

Call eat() and bark() methods using dogObj.

Call eat() and meow() methods using catObj.

End

Here's the Java code implementing the algorithm:

java

Copy code

```java
// Animal class
class Animal {
    String animalType;

    void eat() {
        System.out.println("Animal is eating.");
    }
}

// Dog class inheriting from Animal class
class Dog extends Animal {
    void bark() {
        System.out.println("Dog is barking.");
    }
}

// Cat class inheriting from Animal class
class Cat extends Animal {
    void meow() {
        System.out.println("Cat is meowing.");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        // Create an object of Dog class
        Dog dogObj = new Dog();
        // Call eat() and bark() methods using dogObj
        dogObj.eat();
        dogObj.bark();

        // Create an object of Cat class
        Cat catObj = new Cat();
        // Call eat() and meow() methods using catObj
        catObj.eat();
        catObj.meow();
    }
}
```

EXP3: Packages

Wildcard(*)

Aim:

The aim of this program is to demonstrate the usage of the wildcard (*) in Java to import all classes from a package. This allows for easier access to all classes within a package without specifying each class individually.

Algorithm:

Start:

Define a package named myPackage with multiple classes (Class1, Class2, Class3, ...).

Write Java classes (Class1, Class2, Class3, ...) inside the myPackage package.

Define a public class named Main outside the myPackage package.

Inside the Main class, define the main method as the entry point of the program.

Import all classes from the myPackage package using import myPackage.*;.

Inside the main method:

Access and use the classes (Class1, Class2, Class3, ...) from the myPackage package.

End

Here's the Java code implementing the algorithm:

java

Copy code

```
// Class1.java

package myPackage;

public class Class1 {

   public void display() {

      System.out.println("This is Class1.");

   }

}
```

```
// Class2.java
package myPackage;
public class Class2 {
    public void display() {
        System.out.println("This is Class2.");
    }
}

// Class3.java
package myPackage;
public class Class3 {
    public void display() {
        System.out.println("This is Class3.");
    }
}

// Main.java
import myPackage.*;

public class Main {
    public static void main(String[] args) {
        // Create objects of classes from the myPackage package
        Class1 obj1 = new Class1();
        Class2 obj2 = new Class2();
        Class3 obj3 = new Class3();

        // Call display() method of each class
        obj1.display();
        obj2.display();
        obj3.display();
    }
}
```

Specify the Class

Aim:

The aim of this program is to demonstrate the usage of importing specific classes from a package in Java. By importing only the required classes, it optimizes memory usage and improves code readability.

Algorithm:

Start:

Define a package named myPackage with multiple classes (Class1, Class2, Class3, ...).

Write Java classes (Class1, Class2, Class3, ...) inside the myPackage package.

Define a public class named Main outside the myPackage package.

Inside the Main class, define the main method as the entry point of the program.

Import specific classes (Class1, Class2, Class3, ...) from the myPackage package using import myPackage.Classname;.

Inside the main method:

Access and use the imported classes (Class1, Class2, Class3, ...) from the myPackage package.

End

Here's the Java code implementing the algorithm:

java

Copy code

```
// Class1.java
package myPackage;
public class Class1 {
    public void display() {
        System.out.println("This is Class1.");
    }
}

// Class2.java
package myPackage;
public class Class2 {
    public void display() {
        System.out.println("This is Class2.");
    }
}

// Class3.java
package myPackage;
public class Class3 {
    public void display() {
        System.out.println("This is Class3.");
    }
}

// Main.java
import myPackage.Class1;
import myPackage.Class2;
import myPackage.Class3;

public class Main {
    public static void main(String[] args) {
        // Create objects of specific classes from the myPackage package
        Class1 obj1 = new Class1();
        Class2 obj2 = new Class2();
        Class3 obj3 = new Class3();


        // Call display() method of each class
        obj1.display();
        obj2.display();
        obj3.display();
    }
}
```

Fully Qualified Name

Aim:

The aim of this program is to illustrate the concept of fully qualified names in Java. It shows how to use the fully qualified names to refer to classes in different packages without importing them.

Algorithm:

Start:

Define a package named package1 with a class Class1.

Define another package named package2 with a class Class2.

Define a public class named Main outside both packages.

Inside the Main class, define the main method as the entry point of the program.

Use fully qualified names to refer to Class1 and Class2 from package1 and package2 respectively.

Inside the main method:

Create objects of Class1 and Class2 using fully qualified names.

End

```
// package1/Class1.java
package package1;
public class Class1 {
    public void display() {
        System.out.println("This is Class1 from package1.");
    }
}

// package2/Class2.java
package package2;
public class Class2 {
    public void display() {
        System.out.println("This is Class2 from package2.");
    }
}

// Main.java
public class Main {
    public static void main(String[] args) {
        // Create objects of Class1 and Class2 using fully qualified names
        package1.Class1 obj1 = new package1.Class1();
        package2.Class2 obj2 = new package2.Class2();

        // Call display() method of each class
        obj1.display();
        obj2.display();
    }
}
```

Exp4: I/P Stream class

Aim:

The aim of this program is to illustrate the usage of input and output stream classes in Java. It demonstrates how to read data from an input stream and write data to an output stream.

Algorithm:

Start:

Define a class named IOStreams.

Inside the IOStreams class, define the main method as the entry point of the program.

Create an instance of InputStream to read data.

Create an instance of OutputStream to write data.

Read data from the input stream and write it to the output stream.

Close both input and output streams.

End

```
import java.io.*;


public class IOStreams {
   public static void main(String[] args) {
      // Define input and output streams
      InputStream inputStream = null;
      OutputStream outputStream = null;


      try {
         // Create input and output stream objects
         inputStream = new FileInputStream("input.txt");
         outputStream = new FileOutputStream("output.txt");


         // Read data from input stream and write it to output stream
```

```
            int data;
            while ((data = inputStream.read()) != -1) {
                outputStream.write(data);
            }
            System.out.println("Data has been copied from input.txt to output.txt.");
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        } finally {
            // Close input and output streams
            try {
                if (inputStream != null) {
                    inputStream.close();
                }
                if (outputStream != null) {
                    outputStream.close();
                }
            } catch (IOException e) {
                System.out.println("An error occurred while closing streams: " + e.getMessage());
            }
        }
    }
}
```

EXP 5: Read Write Classes

Aim:

The aim of this program is to illustrate the usage of Reader and Writer classes in Java. It demonstrates how to read data from a text file using a Reader and write data to a text file using a Writer.

Algorithm:

Start:

Define a class named FileReadWrite.

Inside the FileReadWrite class, define the main method as the entry point of the program.

Create instances of Reader and Writer to read from and write to a file.

Use a BufferedReader to read text from an input file.

Use a BufferedWriter to write text to an output file.

Read data from the input file and write it to the output file.

Close both Reader and Writer objects.

End

Here's the Java code implementing the algorithm:

java

Copy code

```
import java.io.*;

public class FileReadWrite {
    public static void main(String[] args) {
        // Define Reader and Writer objects
        Reader reader = null;
        Writer writer = null;

        try {
```

```
    // Create Reader and Writer objects
    reader = new FileReader("input.txt");
    writer = new FileWriter("output.txt");

    // Use BufferedReader to read text from input file
    BufferedReader bufferedReader = new BufferedReader(reader);
    // Use BufferedWriter to write text to output file
    BufferedWriter bufferedWriter = new BufferedWriter(writer);

    // Read data from input file and write it to output file
    String line;
    while ((line = bufferedReader.readLine()) != null) {
        bufferedWriter.write(line);
        bufferedWriter.newLine(); // Add newline character
    }
    System.out.println("Data has been copied from input.txt to output.txt.");
} catch (IOException e) {
    System.out.println("An error occurred: " + e.getMessage());
} finally {
    // Close Reader and Writer objects
    try {
        if (reader != null) {
            reader.close();
        }
        if (writer != null) {
            writer.close();
        }
    } catch (IOException e) {
        System.out.println("An error occurred while closing streams: " + e.getMessage());
```

```
            }
        }
    }
}
```

EXP 6 : Data Input and output Stream

Aim:

The aim of this program is to illustrate the usage of DataInputStream and DataOutputStream classes in Java. It demonstrates how to read and write primitive data types from/to a file using these classes.

Algorithm:

Start:

Define a class named DataStreamExample.

Inside the DataStreamExample class, define the main method as the entry point of the program.

Create instances of DataInputStream and DataOutputStream to read from and write to a file.

Write primitive data types (int, double, boolean) to the output file using DataOutputStream.

Read the written data from the input file using DataInputStream.

Print the read data to the console.

Close both DataInputStream and DataOutputStream.

End

Here's the Java code implementing the algorithm:

java

Copy code

```
import java.io.*;

public class DataStreamExample {
    public static void main(String[] args) {
        // Define DataInputStream and DataOutputStream objects
        DataInputStream dataInputStream = null;
        DataOutputStream dataOutputStream = null;

        try {
            // Create DataOutputStream object
            dataOutputStream = new DataOutputStream(new
FileOutputStream("data_output.txt"));

            // Write data to the output file
            dataOutputStream.writeInt(123);
            dataOutputStream.writeDouble(456.789);
            dataOutputStream.writeBoolean(true);

            System.out.println("Data has been written to data_output.txt.");

            // Create DataInputStream object
            dataInputStream = new DataInputStream(new FileInputStream("data_output.txt"));

            // Read data from the input file
            int intValue = dataInputStream.readInt();
            double doubleValue = dataInputStream.readDouble();
            boolean booleanValue = dataInputStream.readBoolean();

            // Print the read data
            System.out.println("Int Value: " + intValue);
            System.out.println("Double Value: " + doubleValue);
            System.out.println("Boolean Value: " + booleanValue);
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        } finally {
            // Close DataInputStream and DataOutputStream objects
            try {
                if (dataInputStream != null) {
                    dataInputStream.close();
                }
                if (dataOutputStream != null) {
                    dataOutputStream.close();
                }
            } catch (IOException e) {
                System.out.println("An error occurred while closing streams: " + e.getMessage());
            }
        }
    }
}
```

EXP 7 : JAVA FX or J2ME

Aim:

To develop a basic Calculator application using **JavaFX** in **IntelliJ IDEA** that can perform standard arithmetic operations such as **Addition, Subtraction, Multiplication, and Division** with a user-friendly graphical interface.

Algorithm:

1. **Start the JavaFX Project:**
   - Open IntelliJ IDEA.
   - Create a new JavaFX project (ensure JavaFX libraries are configured).
2. **Design the GUI:**
   - Create a layout using GridPane or VBox.
   - Add display (TextField or Label) to show input and result.
   - Add buttons for digits 0-9, operations +, -, *, /, ., =, and C (clear).
3. **Handle Button Clicks:**
   - Use setOnAction() to attach event handlers to each button.
   - Update the display as user inputs digits/operators.
4. **Implement the Logic:**
   - Store the first operand, operator, and second operand.
   - When = is clicked:
     - Parse the operands.
     - Perform the corresponding arithmetic operation.
     - Display the result.
5. **Add Clear Function:**
   - Reset all stored values and clear the display when C is pressed.
6. **Handle Errors:**
   - Add basic error handling for division by zero and invalid input.
7. **Test the App:**
   - Run the app and test with different combinations to ensure correctness.

8. **End the Program.**

CODES

FXML CODE:

```
<?xml version="1.0" encoding="UTF-8"?>


<?import javafx.geometry.Insets?>

<?import javafx.scene.control.Button?>

<?import javafx.scene.control.Label?>

<?import javafx.scene.layout.ColumnConstraints?>

<?import javafx.scene.layout.GridPane?>

<?import javafx.scene.layout.RowConstraints?>


<GridPane hgap="10" prefHeight="232.0" prefWidth="315.0" vgap="10"
xmlns:fx="http://javafx.com/fxml/1" xmlns="http://javafx.com/javafx/23.0.1"
fx:controller="com.pradeep.myjavafx.HelloController">

   <!-- Display label for the calculator -->

   <Label fx:id="display" alignment="center" style="-fx-font-size: 24;"
GridPane.columnSpan="4" />


   <!-- Row 1 -->

   <Button onAction="#onDigitButtonClick" prefHeight="25.0" prefWidth="47.0" text="7"
GridPane.columnIndex="0" GridPane.rowIndex="1" />

   <Button onAction="#onDigitButtonClick" prefHeight="25.0" prefWidth="62.0" text="8"
GridPane.columnIndex="1" GridPane.rowIndex="1" />

   <Button onAction="#onDigitButtonClick" prefHeight="25.0" prefWidth="48.0" text="9"
GridPane.columnIndex="2" GridPane.rowIndex="1" />

   <Button onAction="#onOperatorButtonClick" prefHeight="25.0" prefWidth="86.0"
text="/" GridPane.columnIndex="3" GridPane.rowIndex="1" />


   <!-- Row 2 -->
```

```
  <Button onAction="#onDigitButtonClick" prefHeight="25.0" prefWidth="48.0" text="4"
GridPane.columnIndex="0" GridPane.rowIndex="2" />
  <Button onAction="#onDigitButtonClick" prefHeight="25.0" prefWidth="63.0" text="5"
GridPane.columnIndex="1" GridPane.rowIndex="2" />
  <Button onAction="#onDigitButtonClick" prefHeight="25.0" prefWidth="48.0" text="6"
GridPane.columnIndex="2" GridPane.rowIndex="2" />
  <Button onAction="#onOperatorButtonClick" prefHeight="25.0" prefWidth="85.0"
text="*" GridPane.columnIndex="3" GridPane.rowIndex="2" />


  <!-- Row 3 -->
  <Button onAction="#onDigitButtonClick" prefHeight="25.0" prefWidth="49.0" text="1"
GridPane.columnIndex="0" GridPane.rowIndex="3" />
  <Button onAction="#onDigitButtonClick" prefHeight="25.0" prefWidth="61.0" text="2"
GridPane.columnIndex="1" GridPane.rowIndex="3" />
  <Button onAction="#onDigitButtonClick" prefHeight="25.0" prefWidth="48.0" text="3"
GridPane.columnIndex="2" GridPane.rowIndex="3" />
  <Button onAction="#onOperatorButtonClick" prefHeight="25.0" prefWidth="84.0" text="-
" GridPane.columnIndex="3" GridPane.rowIndex="3" />


  <!-- Row 4 -->
  <Button onAction="#onDigitButtonClick" prefHeight="25.0" prefWidth="49.0" text="0"
GridPane.rowIndex="4" />
  <Button onAction="#onClearButtonClick" prefHeight="35.0" prefWidth="60.0" text="C"
GridPane.columnIndex="1" GridPane.rowIndex="4" />
  <Button onAction="#onEqualsButtonClick" prefHeight="36.0" prefWidth="47.0" text="="
GridPane.columnIndex="2" GridPane.rowIndex="4" />
  <Button onAction="#onOperatorButtonClick" prefHeight="40.0" prefWidth="80.0"
text="+" GridPane.columnIndex="3" GridPane.rowIndex="4" />
 <columnConstraints>
   <ColumnConstraints />
   <ColumnConstraints />
   <ColumnConstraints />
   <ColumnConstraints />
 </columnConstraints>
 <rowConstraints>
   <RowConstraints />
   <RowConstraints />
   <RowConstraints />
   <RowConstraints />
   <RowConstraints />
 </rowConstraints>
 <padding>
   <Insets bottom="20.0" left="40.0" right="20.0" top="20.0" />
 </padding>
</GridPane>
```

CONTROLLER CODE:

```
package com.pradeep.myjavafx;


import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.Label;


public class HelloController {


    @FXML
    private Label display;


    private String currentInput = "";
    private double result = 0;
    private String lastOperator = "";
    private boolean isOperatorPressed = false;
```

```
// Called when a digit button is clicked
@FXML
protected void onDigitButtonClick(javafx.event.ActionEvent event) {
    if (isOperatorPressed) {
        currentInput = "";
        isOperatorPressed = false;
    }

    Button button = (Button) event.getSource();
    currentInput += button.getText();
    display.setText(currentInput);
}

// Called when an operator button is clicked
@FXML
protected void onOperatorButtonClick(javafx.event.ActionEvent event) {
    Button button = (Button) event.getSource();
    String operator = button.getText();

    if (!currentInput.isEmpty()) {
        result = Double.parseDouble(currentInput);
    }

    lastOperator = operator;
    isOperatorPressed = true;
}

// Called when the equal button is clicked
```

```
@FXML
protected void onEqualsButtonClick(javafx.event.ActionEvent event) {
    if (!currentInput.isEmpty() && !lastOperator.isEmpty()) {
        double secondOperand = Double.parseDouble(currentInput);
        switch (lastOperator) {
            case "+":
                result += secondOperand;
                break;
            case "-":
                result -= secondOperand;
                break;
            case "*":
                result *= secondOperand;
                break;
            case "/":
                result /= secondOperand;
                break;
        }

        display.setText(String.valueOf(result));
        currentInput = String.valueOf(result);
        lastOperator = "";
    }
}

// Called when the clear button is clicked
@FXML
protected void onClearButtonClick(javafx.event.ActionEvent event) {
    currentInput = "";
```

```
        result = 0;

        display.setText("0");

    }

}
```

Launch code:

```
package com.pradeep.myjavafx;

import javafx.application.Application;

import javafx.fxml.FXMLLoader;

import javafx.scene.Scene;

import javafx.stage.Stage;

import java.io.IOException;

public class HelloApplication extends Application {

    @Override

    public void start(Stage stage) throws IOException {

        FXMLLoader fxmlLoader = new FXMLLoader(HelloApplication.class.getResource("hello-
view.fxml"));

        Scene scene = new Scene(fxmlLoader.load(), 320, 240);

        stage.setTitle("Calculator");

        stage.setScene(scene);

        stage.show();

    }

    public static void main(String[] args) {

        launch();

    }

}
```

EXP 8: GPIO CONFIG

Aim:

The aim of this program is to demonstrate how to configure GPIO pins as input and output on a Raspberry Pi using Java and the Pi4J library.

Algorithm:

Start:

Import necessary libraries (com.pi4j.io.gpio.*) for GPIO manipulation.

Set up a GPIO pin as input or output using the GpioFactory and GpioController.

For an input pin:

Create a GpioPinDigitalInput instance using GpioFactory.getInstance().provisionDigitalInputPin().

Optionally, add listeners for events such as pin state changes.

For an output pin:

Create a GpioPinDigitalOutput instance using GpioFactory.getInstance().provisionDigitalOutputPin().

Optionally, set initial state (high or low) using setState() method.

Perform desired operations with the GPIO pins, such as reading input state or setting output state.

Release GPIO resources when done.

End

Here's the Java code implementing the algorithm:

java

```
import com.pi4j.io.gpio.*;


public class GPIOExample {

    public static void main(String[] args) throws InterruptedException {

        // Create a GPIO controller

        final GpioController gpio = GpioFactory.getInstance();
        // Configure pin 1 as input
        final GpioPinDigitalInput inputPin = gpio.provisionDigitalInputPin(RaspiPin.GPIO_01,
PinPullResistance.PULL_DOWN);

        // Configure pin 2 as output
        final GpioPinDigitalOutput outputPin =
gpio.provisionDigitalOutputPin(RaspiPin.GPIO_02, "MyLED", PinState.LOW);

        // Read the state of the input pin
        System.out.println("Input Pin State: " + inputPin.getState());

        // Toggle the state of the output pin every second
        while (true) {
            outputPin.toggle();
            Thread.sleep(1000);
        }

        // Release GPIO resources when done
        // gpio.shutdown();
    }
}
```

EXP 9 :read values from MC and display them in console (Serial comm)

Aim:

The aim of this program is to develop an application that reads the environment temperature using a temperature sensor connected to a microcontroller or development board and displays the temperature value on a display or console interface.

Algorithm:

Start:

Connect the temperature sensor to the microcontroller or development board.

Import necessary libraries for interfacing with the temperature sensor and controlling GPIO pins.

Initialize the temperature sensor and configure it for temperature reading.

Read the temperature value from the sensor.

Display the temperature value on a display (such as an LCD) or console interface.

Repeat steps 5-6 at regular intervals to continuously monitor the temperature.

Optionally, add error handling to handle communication errors or sensor failures.

End

Java CODE:

```
import oshi.SystemInfo;

import oshi.hardware.CentralProcessor;

import oshi.hardware.HardwareAbstractionLayer;

import oshi.hardware.Sensors;


public class CpuTemperatureReader {


    public static void main(String[] args) {

        // Create an instance of SystemInfo to access hardware data

        SystemInfo systemInfo = new SystemInfo();

        HardwareAbstractionLayer hal = systemInfo.getHardware();
        Sensors sensors = hal.getSensors();

        try {
            while (true) {
                // Get the CPU temperature (returns NaN if not available)
                double cpuTemperature = sensors.getCpuTemperature();

                if (!Double.isNaN(cpuTemperature)) {
                    // Print the temperature value in Celsius
                    System.out.printf("Temperature: %.2f °C\n", cpuTemperature);
                } else {
                    System.out.println("CPU temperature data is not available or unsupported on this
system.");
                }

                // Wait for 5 seconds before reading again
                Thread.sleep(5000);
            }
        } catch (InterruptedException e) {
            System.out.println("\nTemperature reading stopped by user.");
            Thread.currentThread().interrupt();
        }
    }
}
```

Dependencies:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.pradeep</groupId>
    <artifactId>CpuTemperatureReader</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- OSHI dependency -->
        <dependency>
            <groupId>com.github.oshi</groupId>
            <artifactId>oshi-core</artifactId>
            <version>6.7.1</version>
        </dependency>

        <!-- SLF4J API -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>2.0.17</version>
        </dependency>

        <!-- Logback Classic implementation -->
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.4.7</version>
        </dependency>
        <!-- Logback Core (required by logback-classic) -->
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-core</artifactId>
            <version>1.4.7</version>
        </dependency>
    </dependencies>
    <properties>
        <maven.compiler.source>23</maven.compiler.source>
        <maven.compiler.target>23</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

</project>
```

EXP 10: develop an application that read data from Arduino boards

Aim:

The aim of this program is to develop a Java application that reads sensor values from a light sensor connected to an Arduino board and communicates with the Arduino board via USB serial communication from a Raspberry Pi.

Algorithm:

Start:

Connect the light sensor to the analog input pins of the Arduino board.

Upload the Arduino sketch to read analog sensor values and send them over serial communication.

Connect the Arduino board to the Raspberry Pi via USB cable.

Import the necessary Java libraries for serial communication (SerialPort from jSerialComm).

Open a serial connection to the Arduino board from the Raspberry Pi using the appropriate port and baud rate.

Read sensor values sent by the Arduino board over the serial connection.

Process and display the sensor readings as required.

Repeat steps 7-8 at regular intervals to continuously monitor the sensor values.

Optionally, handle errors and exceptions related to serial communication or sensor readings.

End

Java code:

```
import com.fazecast.jSerialComm.SerialPort;

import java.io.InputStream;

import java.io.FileWriter;

import java.io.IOException;


public class TestRXTX {

  public static void main(String[] args) {

    // Get all available serial ports

    SerialPort[] ports = SerialPort.getCommPorts();


    // Display available ports

    for (SerialPort port : ports) {

      System.out.println("Port: " + port.getSystemPortName());

    }


    // Choose the first available port (you can modify this if needed)

    if (ports.length > 0) {

      SerialPort chosenPort = ports[0]; // Use the first available port

      System.out.println("Opening port: " + chosenPort.getSystemPortName());


      // Open the port
```

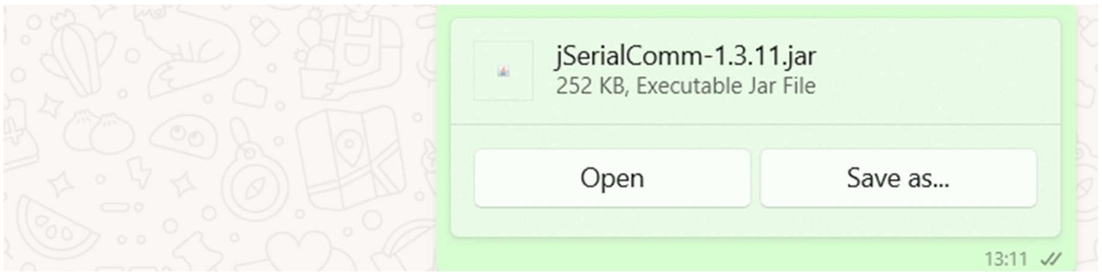
```
            chosenPort.openPort();


            // Create a FileWriter to write the data into a text file
            try (FileWriter fileWriter = new FileWriter("sensor_data.txt", true)) { // true for
append mode
                // Set up the input stream to read data from the serial port
                InputStream inputStream = chosenPort.getInputStream();
                byte[] readBuffer = new byte[1024];
                int numBytes;


                // Continuously read data from the input stream
                while (true) {
                    // Check if there is data available to read
                    if (inputStream.available() > 0) {
                        numBytes =  inputStream.read(readBuffer);
                        String data = new String(readBuffer, 0, numBytes);


                        // Write the received data to the text file
                        fileWriter.write(data);  // Write the data to the file
                        fileWriter.flush();  // Ensure the data is written immediately
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                // Close the port when done
                chosenPort.closePort();
            }
        } else {
            System.out.println("No serial ports found.");
        }
    }
}
```

JAR Files:



jSerialComm-1.3.11.jar
252 KB, Executable Jar File

Arduino Code:

```
void setup() {
  // Initialize serial communication at 9600 baud rate
  Serial.begin(9600);
}

void loop() {
  // Read analog sensor value from analog pin A0
  int sensorValue = analogRead(A0);

  // Print sensor value to serial port
  Serial.print("Light sensor value:");
  Serial.println(sensorValue);

  // Delay for a short period before reading again
  delay(1000);
}
```