S.Y.M.Sc(Comp. Sci.) Practical Examination

Practical Paper(CS-604-MJP) :

Lab Course on CS-601-MJ and CS-603-MJ

(Software Architecture & Design Pattern and Internet of Things)

Software Architecture And Design pattern or IOT Practical slips programs

Slip 1 :

Q.1 Write a Java Program to implement I/O Decorator for converting uppercase letters to lower case letters

```
import java.io.*;

// Decorator class that converts uppercase letters to lowercase
class LowerCaseInputStream extends InputStreamReader {
    private Reader reader;

    public LowerCaseInputStream(Reader reader) {
        super(reader);
        this.reader = reader;
    }
```

```
    @Override
    public int read() throws IOException {
        int data = reader.read();
        if (data == -1) {
            return -1; // End of stream
        }
        // Convert to lowercase if it's an uppercase letter
        return Character.toLowerCase((char) data);
    }

    @Override
    public int read(char[] cbuf, int off, int len) throws IOException {
        int numCharsRead = reader.read(cbuf, off, len);
        for (int i = off; i < off + numCharsRead; i++) {
            cbuf[i] = Character.toLowerCase(cbuf[i]);
        }
        return numCharsRead;
    }
}

public class IODecoratorExample {
    public static void main(String[] args) {
        try {
```

```
        // Wrapping System.in with a BufferedReader and then LowerCaseInputStream

        Reader reader = new LowerCaseInputStream(new BufferedReader(new
InputStreamReader(System.in)));

        BufferedReader br = new BufferedReader(reader);



        System.out.println("Enter some text (uppercase will be converted to lowercase):");



        String line;

        while ((line = br.readLine()) != null) {

            System.out.println("Converted text: " + line);

        }

      } catch (IOException e) {

        e.printStackTrace();

      }

   }

}
```

Q.2 Write a program to sense the available networks using Arduino

```
#include <WiFi.h>  // For ESP32. For ESP8266, use <ESP8266WiFi.h>

void setup() {
  // Start the serial communication to see the output
  Serial.begin(115200);

  // Connect to Wi-Fi (no credentials needed for scanning)
  WiFi.mode(WIFI_STA);  // Set the Wi-Fi mode to station (client)
  Serial.println("Scanning for Wi-Fi networks...");
```

```
  // Start the scan for Wi-Fi networks
  int networkCount = WiFi.scanNetworks(); // This function returns the number of networks
found

  Serial.println("Scan complete.");

  // If networks were found, print the list
  if (networkCount == 0) {
    Serial.println("No networks found.");
  } else {
    Serial.print(networkCount);
    Serial.println(" networks found:");
    for (int i = 0; i < networkCount; i++) {
      // Print the SSID (network name), RSSI (signal strength), and Encryption type
      Serial.print(i + 1);
      Serial.print(": ");
      Serial.print(WiFi.SSID(i));        // Network name (SSID)
      Serial.print(" | Signal Strength: ");
      Serial.print(WiFi.RSSI(i));        // Signal strength (in dBm)
      Serial.print(" dBm | Encryption: ");
      Serial.println(WiFi.encryptionType(i)); // Encryption type
    }
  }
}

void loop() {
  // Nothing to do here as we only need to scan once
}
```

Slip 2 :

Q.1 Write a Java Program to implement Singleton pattern for multithreading

```
// Singleton class with thread-safety in a multithreaded environment
public class Singleton {

    // Declare the instance as volatile to ensure proper synchronization in a multithreaded
environment
    private static volatile Singleton instance;

    // Private constructor to prevent instantiation
    private Singleton() {
        // Simulating time-consuming initialization, e.g., database connection setup, etc.
```

```
        try {
            Thread.sleep(100);  // Simulate some delay in instance creation
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    // Public method to provide access to the instance
    public static Singleton getInstance() {
        // First check (without locking)
        if (instance == null) {
            synchronized (Singleton.class) {
                // Second check (with locking)
                if (instance == null) {
                    instance = new Singleton();  // Create the instance
                }
            }
        }
        return instance;
    }

    public void showMessage() {
        System.out.println("Hello, Singleton instance: " + this);
    }
}

// Thread class to test Singleton instance access in a multithreaded environment
class SingletonTestThread extends Thread {
    @Override
    public void run() {
        Singleton singleton = Singleton.getInstance();
        singleton.showMessage();
    }
}

public class SingletonTest {

    public static void main(String[] args) {
        // Create multiple threads to test Singleton pattern in a multithreaded environment
        SingletonTestThread thread1 = new SingletonTestThread();
        SingletonTestThread thread2 = new SingletonTestThread();
        SingletonTestThread thread3 = new SingletonTestThread();
        SingletonTestThread thread4 = new SingletonTestThread();

        // Start the threads
        thread1.start();
```

```
        thread2.start();
        thread3.start();
        thread4.start();

        try {
            // Wait for all threads to complete
            thread1.join();
            thread2.join();
            thread3.join();
            thread4.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Q.2 Write a program to measure the distance using ultrasonic sensor and make LED blink using Arduino.

```
// Define the pins for the ultrasonic sensor
const int trigPin = 9;
const int echoPin = 10;

// Define the pin for the LED
const int ledPin = 13;

// Define variables for measuring distance
long duration;
int distance;

void setup() {
  // Start the serial communication for debugging
  Serial.begin(9600);

  // Set the trigPin as an output and echoPin as an input
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);

  // Set the LED pin as an output
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // Send a pulse to trigger the ultrasonic sensor
```

```
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);  // Wait for a brief moment to ensure a clean trigger
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);  // Send a 10-microsecond pulse to trigger the sensor
  digitalWrite(trigPin, LOW);

  // Measure the duration of the pulse from the echoPin
  duration = pulseIn(echoPin, HIGH);  // Measure the time the pulse travels

  // Calculate the distance in centimeters (using speed of sound: 343 m/s)
  distance = duration * 0.0344 / 2;  // Time * speed of sound / 2 (for round-trip)

  // Print the distance to the Serial Monitor
  Serial.print("Distance: ");
  Serial.print(distance);
  Serial.println(" cm");

  // Blink the LED if the distance is less than a threshold (e.g., 10 cm)
  if (distance < 10) {
    digitalWrite(ledPin, HIGH);  // Turn the LED ON
    delay(500);  // Wait for 500 milliseconds
    digitalWrite(ledPin, LOW);   // Turn the LED OFF
    delay(500);  // Wait for 500 milliseconds
  } else {
    digitalWrite(ledPin, LOW);   // Ensure the LED is OFF if distance is greater than 10 cm
  }

  delay(100);  // Small delay before the next measurement
}
```

Slip 3 :

Q.1 Write a JAVA Program to implement built-in support (java.util.Observable) Weather station with members temperature, humidity, pressure and methods mesurmentsChanged(), setMesurment(), getTemperature(), getHumidity(), getPressure()

Code Implementation:

1. `WeatherStation.java` (Observable Class)

```
java
Copy code
import java.util.Observable;
```

```
public class WeatherStation extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    // Method to notify observers when measurements change
    public void measurementsChanged() {
        setChanged(); // Marks the observable as changed
        notifyObservers(); // Notifies all observers
    }

    // Set new measurements and notify observers
    public void setMeasurements(float temperature, float humidity, float
pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged(); // Notify observers when new measurements are
set
    }

    // Getters for weather measurements
    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}
```

2. `WeatherDisplay.java` (Observer Class)

```
java
Copy code
import java.util.Observable;
import java.util.Observer;

public class WeatherDisplay implements Observer {

    private float temperature;
    private float humidity;
    private float pressure;

    // Update method to receive notifications from the Observable
    @Override
    public void update(Observable o, Object arg) {
        if (o instanceof WeatherStation) {
            WeatherStation weatherStation = (WeatherStation) o;
            this.temperature = weatherStation.getTemperature();
            this.humidity = weatherStation.getHumidity();
```

```
            this.pressure = weatherStation.getPressure();
            display(); // Display the updated weather information
        }
    }

    // Display the updated weather information
    public void display() {
        System.out.println("Weather Data Updated: ");
        System.out.println("Temperature: " + temperature + "°C");
        System.out.println("Humidity: " + humidity + "%");
        System.out.println("Pressure: " + pressure + " hPa");
        System.out.println("-----------------------------");
    }
}
```

3. `Main.java` (Testing the Program)

```
java
Copy code
public class Main {
    public static void main(String[] args) {
        // Create a WeatherStation instance (Observable)
        WeatherStation weatherStation = new WeatherStation();

        // Create a WeatherDisplay instance (Observer)
        WeatherDisplay weatherDisplay = new WeatherDisplay();

        // Add the WeatherDisplay as an observer to the WeatherStation
        weatherStation.addObserver(weatherDisplay);

        // Simulate changing weather measurements
        weatherStation.setMeasurements(25.0f, 65.0f, 1013.0f); // Temp: 25°C,
Humidity: 65%, Pressure: 1013 hPa
        weatherStation.setMeasurements(28.0f, 70.0f, 1010.0f); // Temp: 28°C,
Humidity: 70%, Pressure: 1010 hPa
        weatherStation.setMeasurements(22.0f, 60.0f, 1020.0f); // Temp: 22°C,
Humidity: 60%, Pressure: 1020 hPa
    }
}
```

Q. 2 Write a program to detects the vibration of an object with sensor using Arduino.

```
// Define the pin for the vibration sensor
const int vibrationPin = 7;  // Pin connected to the OUT pin of the SW-420
sensor
const int ledPin = 13;       // Pin connected to the LED

void setup() {
  // Set vibration sensor pin as input
  pinMode(vibrationPin, INPUT);

  // Set the LED pin as output
  pinMode(ledPin, OUTPUT);
```

```
  // Start the serial communication for debugging
  Serial.begin(9600);
}

void loop() {
  // Read the state of the vibration sensor
  int vibrationState = digitalRead(vibrationPin);

  // Check if the sensor detects vibration (HIGH state)
  if (vibrationState == HIGH) {
    // If vibration is detected, turn on the LED
    digitalWrite(ledPin, HIGH);
    Serial.println("Vibration detected!");
  } else {
    // If no vibration, turn off the LED
    digitalWrite(ledPin, LOW);
    Serial.println("No vibration detected.");
  }

  // Small delay to debounce
  delay(100);
}
```

Slip 4 :

Q.1 Write a Java Program to implement Factory method for Pizza Store with createPizza(), orederPizza(), prepare(), Bake(), cut(), box(). Use this to create variety of pizza's like NyStyleCheesePizza, ChicagoStyleCheesePizza etc.

1. `Pizza` Interface

```
java
Copy code
public interface Pizza {
    void prepare();
    void bake();
    void cut();
    void box();
}
```

2. Concrete Pizza Classes

Each specific pizza will implement the `Pizza` interface.

`NyStyleCheesePizza.java`

```
java
Copy code
public class NyStyleCheesePizza implements Pizza {
```

```
    @Override
    public void prepare() {
        System.out.println("Preparing New York Style Cheese Pizza...");
    }

    @Override
    public void bake() {
        System.out.println("Baking New York Style Cheese Pizza...");
    }

    @Override
    public void cut() {
        System.out.println("Cutting New York Style Cheese Pizza...");
    }

    @Override
    public void box() {
        System.out.println("Boxing New York Style Cheese Pizza...");
    }
}
```

ChicagoStyleCheesePizza.java

```
java
Copy code
public class ChicagoStyleCheesePizza implements Pizza {
    @Override
    public void prepare() {
        System.out.println("Preparing Chicago Style Cheese Pizza...");
    }

    @Override
    public void bake() {
        System.out.println("Baking Chicago Style Cheese Pizza...");
    }

    @Override
    public void cut() {
        System.out.println("Cutting Chicago Style Cheese Pizza...");
    }

    @Override
    public void box() {
        System.out.println("Boxing Chicago Style Cheese Pizza...");
    }
}
```

3. PizzaStore Class with Factory Method

The `PizzaStore` class will have a `createPizza()` method that is overridden by subclasses to create different types of pizzas.

```
java
Copy code
public abstract class PizzaStore {
```

```
    // The orderPizza() method is the factory method that creates a pizza
    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);  // Factory method
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }

    // This method will be overridden by subclasses to create specific pizza
types
    protected abstract Pizza createPizza(String type);
}
```

4. Concrete Pizza Store Subclasses

We can now create specific pizza stores (like `NyPizzaStore` and `ChicagoPizzaStore`) that will implement the `createPizza()` method to create specific types of pizzas.

NyPizzaStore.java

```
java
Copy code
public class NyPizzaStore extends PizzaStore {
    @Override
    protected Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new NyStyleCheesePizza();
        } else {
            // Add other pizza types as needed
            return null;
        }
    }
}
```

ChicagoPizzaStore.java

```
java
Copy code
public class ChicagoPizzaStore extends PizzaStore {
    @Override
    protected Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new ChicagoStyleCheesePizza();
        } else {
            // Add other pizza types as needed
            return null;
        }
    }
}
```

5. Main Class to Test

Now, let's implement a `Main` class to test the pizza store ordering process.

```
java
Copy code
public class Main {
    public static void main(String[] args) {
        // Create a New York Pizza Store
        PizzaStore nyPizzaStore = new NyPizzaStore();
        // Order a Cheese Pizza from New York Pizza Store
        Pizza nyPizza = nyPizzaStore.orderPizza("cheese");

        System.out.println("\n-----\n");

        // Create a Chicago Pizza Store
        PizzaStore chicagoPizzaStore = new ChicagoPizzaStore();
        // Order a Cheese Pizza from Chicago Pizza Store
        Pizza chicagoPizza = chicagoPizzaStore.orderPizza("cheese");
    }
}
```

Output:

```
mathematica
Copy code
Preparing New York Style Cheese Pizza...
Baking New York Style Cheese Pizza...
Cutting New York Style Cheese Pizza...
Boxing New York Style Cheese Pizza...

-----

Preparing Chicago Style Cheese Pizza...
Baking Chicago Style Cheese Pizza...
Cutting Chicago Style Cheese Pizza...
Boxing Chicago Style Cheese Pizza...
```

Q.2 Write a program to sense a finger when it is placed on the board Arduino. [15 M]

```
// Define the pin for the capacitive touch sensor
const int touchPin = 7;  // Pin connected to the SIG (signal) of the TTP223 sensor
const int ledPin = 13;    // Pin connected to the LED

void setup() {
  // Set the touch sensor pin as input
  pinMode(touchPin, INPUT);

  // Set the LED pin as output
  pinMode(ledPin, OUTPUT);
```

```
  // Start serial communication for debugging
  Serial.begin(9600);
}

void loop() {
  // Read the state of the touch sensor
  int touchState = digitalRead(touchPin);

  // If the sensor detects a touch (HIGH state)
  if (touchState == HIGH) {
    digitalWrite(ledPin, HIGH);  // Turn on the LED
    Serial.println("Finger detected! LED ON");
  } else {
    digitalWrite(ledPin, LOW);   // Turn off the LED
    Serial.println("No finger detected. LED OFF");
  }

  // Small delay for stability
  delay(100);
}
```

Slip 5 :

Q.1 Write a Java Program to implement Adapter pattern for Enumeration iterator

```
import java.util.Enumeration;
import java.util.Iterator;
import java.util.Vector;

// Adapter class that adapts Enumeration to Iterator
class EnumerationIteratorAdapter<T> implements Iterator<T> {
   private Enumeration<T> enumeration;

   // Constructor
   public EnumerationIteratorAdapter(Enumeration<T> enumeration) {
      this.enumeration = enumeration;
   }

   // hasNext() method from Iterator
   @Override
   public boolean hasNext() {
      return enumeration.hasMoreElements();
   }

   // next() method from Iterator
```

```
    @Override
    public T next() {
        return enumeration.nextElement();
    }

    // remove() method from Iterator is unsupported for Enumeration
    @Override
    public void remove() {
        throw new UnsupportedOperationException("Remove not supported.");
    }
}

public class AdapterPatternExample {
    public static void main(String[] args) {
        // Creating a Vector and populating it with some data
        Vector<String> vector = new Vector<>();
        vector.add("Element1");
        vector.add("Element2");
        vector.add("Element3");

        // Getting an Enumeration from the Vector
        Enumeration<String> enumeration = vector.elements();

        // Adapting Enumeration to Iterator
        Iterator<String> iterator = new EnumerationIteratorAdapter<>(enumeration);

        // Using the adapted Iterator
        System.out.println("Using Enumeration adapted to Iterator:");
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

Q.2 Write a program to connect with the available Wi-Fi using Arduino.

```
#include <ESP8266WiFi.h> // Use <WiFi.h> for ESP32

// Replace with your network credentials
const char* ssid = "your_SSID";
const char* password = "your_PASSWORD";

void setup() {
```

```
  Serial.begin(115200);
  delay(10);

  // Connect to Wi-Fi network
  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("Wi-Fi connected successfully.");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void loop() {
  // Your code here (e.g., HTTP requests, etc.)
}
```

Slip 6 :

Q.1 Write a Java Program to implement command pattern to test Remote Control

1. Define the Command Interface

The `Command` interface defines a single method `execute()` that all concrete commands must implement.

```
java
Copy code
// Command interface
interface Command {
    void execute();
```

```
}
```

2. Create Concrete Command Classes

Each concrete command will implement the `Command` interface and perform an action on the receiver. Here, let's assume we have `Light` and `Fan` devices to control.

Light Commands

```
java
Copy code
// Receiver class: Light
class Light {
    public void on() {
        System.out.println("Light is ON");
    }

    public void off() {
        System.out.println("Light is OFF");
    }
}

// Command to turn the light on
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.on();
    }
}

// Command to turn the light off
class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.off();
    }
}
```

Fan Commands

```
java
Copy code
```

```
// Receiver class: Fan
class Fan {
    public void start() {
        System.out.println("Fan is running");
    }

    public void stop() {
        System.out.println("Fan is stopped");
    }
}

// Command to start the fan
class FanStartCommand implements Command {
    private Fan fan;

    public FanStartCommand(Fan fan) {
        this.fan = fan;
    }

    @Override
    public void execute() {
        fan.start();
    }
}

// Command to stop the fan
class FanStopCommand implements Command {
    private Fan fan;

    public FanStopCommand(Fan fan) {
        this.fan = fan;
    }

    @Override
    public void execute() {
        fan.stop();
    }
}
```

3. Create the Remote Control (Invoker)

The `RemoteControl` class has buttons (slots) that execute commands.

```
java
Copy code
// Invoker class: RemoteControl
class RemoteControl {
    private Command[] buttons;

    public RemoteControl() {
        buttons = new Command[4];
    }

    public void setCommand(int slot, Command command) {
        buttons[slot] = command;
```

```
    }

    public void pressButton(int slot) {
        if (buttons[slot] != null) {
            buttons[slot].execute();
        } else {
            System.out.println("No command assigned to this button");
        }
    }
}
```

4. Test the Remote Control

Now, we can test the Remote Control by assigning commands to different buttons and pressing them.

```
java
Copy code
public class RemoteControlTest {
    public static void main(String[] args) {
        RemoteControl remote = new RemoteControl();

        // Creating receivers
        Light livingRoomLight = new Light();
        Fan ceilingFan = new Fan();

        // Creating commands
        Command lightOn = new LightOnCommand(livingRoomLight);
        Command lightOff = new LightOffCommand(livingRoomLight);
        Command fanStart = new FanStartCommand(ceilingFan);
        Command fanStop = new FanStopCommand(ceilingFan);

        // Setting commands to remote control buttons
        remote.setCommand(0, lightOn);
        remote.setCommand(1, lightOff);
        remote.setCommand(2, fanStart);
        remote.setCommand(3, fanStop);

        // Testing the remote control
        System.out.println("Testing Remote Control:");
        remote.pressButton(0); // Light ON
        remote.pressButton(1); // Light OFF
        remote.pressButton(2); // Fan ON
        remote.pressButton(3); // Fan OFF
    }
}
```

Q.2 Write a program to get temperature notification using Arduino.

#include <DHT.h>

```
// Define DHT sensor type and pin
#define DHTPIN 2         // Pin connected to the DHT sensor
#define DHTTYPE DHT11    // DHT11 or DHT22

// Threshold temperature for notifications
const float TEMP_THRESHOLD = 30.0;  // Temperature threshold in Celsius

// Notification output pin (for LED or Buzzer)
#define NOTIFICATION_PIN 3

// Initialize the DHT sensor
DHT dht(DHTPIN, DHTTYPE);

void setup() {
  Serial.begin(9600);
  dht.begin();

  // Initialize notification pin
  pinMode(NOTIFICATION_PIN, OUTPUT);
  digitalWrite(NOTIFICATION_PIN, LOW); // Start with LED/Buzzer OFF
}

void loop() {
  // Read temperature and humidity
  float temperature = dht.readTemperature();
  float humidity = dht.readHumidity();

  // Check if the readings are valid
  if (isnan(temperature) || isnan(humidity)) {
    Serial.println("Failed to read from DHT sensor!");
    return;
  }

  // Print temperature and humidity to Serial Monitor
  Serial.print("Temperature: ");
  Serial.print(temperature);
  Serial.print(" °C\tHumidity: ");
  Serial.print(humidity);
  Serial.println(" %");

  // Check if temperature exceeds threshold
  if (temperature > TEMP_THRESHOLD) {
    Serial.println("Warning: Temperature exceeded threshold!");

    // Activate LED/Buzzer as a notification
    digitalWrite(NOTIFICATION_PIN, HIGH);
```

```
   delay(500); // Keep the notification ON for 500 ms
   digitalWrite(NOTIFICATION_PIN, LOW);
  }

  delay(2000); // Wait for 2 seconds before reading again
}
```

Slip 7 :

Q.1 Write a Java Program to implement undo command to test Ceiling fan. [15 M]

Steps

1. Define a `Command` interface with `execute()` and `undo()` methods.
2. Create a `CeilingFan` receiver class that has various speed settings and an off state.
3. Implement concrete command classes (`CeilingFanLowCommand`, `CeilingFanMediumCommand`, `CeilingFanHighCommand`, `CeilingFanOffCommand`) to set different fan speeds.
4. Each command will store the previous speed of the fan, allowing the `undo` method to revert to the previous state.
5. Test the functionality by changing fan speeds and using the `undo` feature.

Code Implementation

Step 1: Command Interface

```
java
Copy code
// Command interface
interface Command {
    void execute();
    void undo();
}
```

Step 2: CeilingFan Receiver Class

The `CeilingFan` class will have methods to set different speeds and a variable to keep track of its current speed.

```
java
Copy code
// Receiver class: CeilingFan
class CeilingFan {
```

```
    public static final int HIGH = 3;
    public static final int MEDIUM = 2;
    public static final int LOW = 1;
    public static final int OFF = 0;

    private int speed;

    public CeilingFan() {
        speed = OFF;
    }

    public void high() {
        speed = HIGH;
        System.out.println("Ceiling fan is on high");
    }

    public void medium() {
        speed = MEDIUM;
        System.out.println("Ceiling fan is on medium");
    }

    public void low() {
        speed = LOW;
        System.out.println("Ceiling fan is on low");
    }

    public void off() {
        speed = OFF;
        System.out.println("Ceiling fan is off");
    }

    public int getSpeed() {
        return speed;
    }
}
```

Step 3: Concrete Command Classes

Each command class sets a specific speed for the fan and stores the previous speed so that it can be reverted with `undo()`.

```
java
Copy code
// Command to set ceiling fan to high
class CeilingFanHighCommand implements Command {
    private CeilingFan ceilingFan;
    private int prevSpeed;

    public CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    @Override
    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
```

```
        ceilingFan.high();
    }

    @Override
    public void undo() {
        setPreviousSpeed();
    }

    private void setPreviousSpeed() {
        if (prevSpeed == CeilingFan.HIGH) ceilingFan.high();
        else if (prevSpeed == CeilingFan.MEDIUM) ceilingFan.medium();
        else if (prevSpeed == CeilingFan.LOW) ceilingFan.low();
        else ceilingFan.off();
    }
}

// Command to set ceiling fan to medium
class CeilingFanMediumCommand implements Command {
    private CeilingFan ceilingFan;
    private int prevSpeed;

    public CeilingFanMediumCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    @Override
    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.medium();
    }

    @Override
    public void undo() {
        setPreviousSpeed();
    }

    private void setPreviousSpeed() {
        if (prevSpeed == CeilingFan.HIGH) ceilingFan.high();
        else if (prevSpeed == CeilingFan.MEDIUM) ceilingFan.medium();
        else if (prevSpeed == CeilingFan.LOW) ceilingFan.low();
        else ceilingFan.off();
    }
}

// Command to set ceiling fan to low
class CeilingFanLowCommand implements Command {
    private CeilingFan ceilingFan;
    private int prevSpeed;

    public CeilingFanLowCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    @Override
    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.low();
```

```
    }

    @Override
    public void undo() {
        setPreviousSpeed();
    }

    private void setPreviousSpeed() {
        if (prevSpeed == CeilingFan.HIGH) ceilingFan.high();
        else if (prevSpeed == CeilingFan.MEDIUM) ceilingFan.medium();
        else if (prevSpeed == CeilingFan.LOW) ceilingFan.low();
        else ceilingFan.off();
    }
}

// Command to turn off the ceiling fan
class CeilingFanOffCommand implements Command {
    private CeilingFan ceilingFan;
    private int prevSpeed;

    public CeilingFanOffCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    @Override
    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.off();
    }

    @Override
    public void undo() {
        setPreviousSpeed();
    }

    private void setPreviousSpeed() {
        if (prevSpeed == CeilingFan.HIGH) ceilingFan.high();
        else if (prevSpeed == CeilingFan.MEDIUM) ceilingFan.medium();
        else if (prevSpeed == CeilingFan.LOW) ceilingFan.low();
        else ceilingFan.off();
    }
}
```

Step 4: Remote Control (Invoker) Class with Undo Functionality

```
java
Copy code
// RemoteControl class to invoke commands
class RemoteControl {
    private Command slot;
    private Command lastCommand;

    public void setCommand(Command command) {
        slot = command;
    }
```

```
    public void pressButton() {
        slot.execute();
        lastCommand = slot;
    }

    public void pressUndo() {
        if (lastCommand != null) {
            lastCommand.undo();
        }
    }
}
```

Step 5: Testing the Ceiling Fan with Undo Functionality

```
java
Copy code
public class CeilingFanTest {
    public static void main(String[] args) {
        RemoteControl remote = new RemoteControl();
        CeilingFan ceilingFan = new CeilingFan();

        // Create commands for different speeds and off state
        Command ceilingFanHigh = new CeilingFanHighCommand(ceilingFan);
        Command ceilingFanMedium = new CeilingFanMediumCommand(ceilingFan);
        Command ceilingFanLow = new CeilingFanLowCommand(ceilingFan);
        Command ceilingFanOff = new CeilingFanOffCommand(ceilingFan);

        // Test turning fan to high and undoing
        remote.setCommand(ceilingFanHigh);
        remote.pressButton();
        remote.pressUndo();

        // Test turning fan to medium and undoing
        remote.setCommand(ceilingFanMedium);
        remote.pressButton();
        remote.pressUndo();

        // Test turning fan to low and undoing
        remote.setCommand(ceilingFanLow);
        remote.pressButton();
        remote.pressUndo();

        // Test turning fan off and undoing
        remote.setCommand(ceilingFanOff);
        remote.pressButton();
        remote.pressUndo();
    }
}
```

Explanation

1. **Command Interface**: Each command has an `execute` and an `undo` method.
2. **Concrete Commands**: Commands store the previous speed before executing a new one, allowing them to revert with `undo()`.

3. **RemoteControl (Invoker)**: Tracks the last executed command and calls `undo()` on it when the undo button is pressed.
4. **Testing**: This program will print each fan speed change, and when `undo()` is called, it will revert to the previous speed.

Example Output

```
vbnet
Copy code
Ceiling fan is on high
Ceiling fan is off
Ceiling fan is on medium
Ceiling fan is off
Ceiling fan is on low
Ceiling fan is off
Ceiling fan is off
```
Ceiling fan is on low

Q.2 Write a program for LDR to vary the light intensity of LED using Arduino

```
// Define the pins
const int ldrPin = A0;     // Analog pin connected to the LDR
const int ledPin = 9;      // PWM pin connected to the LED

void setup() {
  Serial.begin(9600);      // Start serial communication for debugging
  pinMode(ledPin, OUTPUT); // Set LED pin as output
}

void loop() {
  // Read the analog value from the LDR
  int ldrValue = analogRead(ldrPin);

  // Map the LDR value (0-1023) to PWM range (0-255)
  int ledBrightness = map(ldrValue, 0, 1023, 0, 255);

  // Set the brightness of the LED
  analogWrite(ledPin, ledBrightness);

  // Print values to Serial Monitor for debugging
  Serial.print("LDR Value: ");
  Serial.print(ldrValue);
  Serial.print(" -> LED Brightness: ");
  Serial.println(ledBrightness);

  delay(100); // Small delay for stability
```

}

Slip 8 :

Q. 1 Write a Java Program to implement State Pattern for Gumball Machine. Create instance variable that holds current state from there, we just need to handle all actions, behaviors and state transition that can happen

Steps

1. Define a `State` interface with methods for actions such as inserting a coin, ejecting a coin, turning the crank, and dispensing.
2. Create concrete state classes for each of the machine's states.
3. In the `GumballMachine` class, keep a reference to the current state and delegate actions to it, allowing for state transitions.

Code Implementation

Step 1: `State` Interface

The `State` interface declares actions available on the gumball machine.

```
java
Copy code
interface State {
    void insertCoin();
    void ejectCoin();
    void turnCrank();
    void dispense();
}
```

Step 2: Concrete State Classes

Each concrete state class implements `State` and handles actions accordingly.

```
java
Copy code
// State when there is no coin inserted
class NoCoinState implements State {
    private GumballMachine gumballMachine;

    public NoCoinState(GumballMachine gumballMachine) {
```

```
        this.gumballMachine = gumballMachine;
    }

    @Override
    public void insertCoin() {
        System.out.println("Coin inserted.");
        gumballMachine.setState(gumballMachine.getHasCoinState());
    }

    @Override
    public void ejectCoin() {
        System.out.println("No coin to eject.");
    }

    @Override
    public void turnCrank() {
        System.out.println("Insert a coin first.");
    }

    @Override
    public void dispense() {
        System.out.println("Insert a coin to get a gumball.");
    }
}

// State when there is a coin inserted
class HasCoinState implements State {
    private GumballMachine gumballMachine;

    public HasCoinState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    @Override
    public void insertCoin() {
        System.out.println("Coin already inserted.");
    }

    @Override
    public void ejectCoin() {
        System.out.println("Coin ejected.");
        gumballMachine.setState(gumballMachine.getNoCoinState());
    }

    @Override
    public void turnCrank() {
        System.out.println("Crank turned...");
        gumballMachine.setState(gumballMachine.getSoldState());
    }

    @Override
    public void dispense() {
        System.out.println("Turn the crank to get a gumball.");
    }
}

// State when the gumball is being dispensed
```

```
class SoldState implements State {
    private GumballMachine gumballMachine;

    public SoldState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    @Override
    public void insertCoin() {
        System.out.println("Please wait, dispensing gumball.");
    }

    @Override
    public void ejectCoin() {
        System.out.println("Cannot eject, crank already turned.");
    }

    @Override
    public void turnCrank() {
        System.out.println("Turning twice won’t get you another gumball!");
    }

    @Override
    public void dispense() {
        gumballMachine.releaseGumball();
        if (gumballMachine.getCount() > 0) {
            gumballMachine.setState(gumballMachine.getNoCoinState());
        } else {
            System.out.println("Out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }
}

// State when the gumball machine is sold out
class SoldOutState implements State {
    private GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

    @Override
    public void insertCoin() {
        System.out.println("Out of gumballs, can't insert coin.");
    }

    @Override
    public void ejectCoin() {
        System.out.println("No coin to eject.");
    }

    @Override
    public void turnCrank() {
        System.out.println("No gumballs to dispense.");
    }
```

```
    @Override
    public void dispense() {
        System.out.println("No gumballs available.");
    }
}
```

Step 3: `GumballMachine` Class

The `GumballMachine` class manages the states and transitions.

```
java
Copy code
class GumballMachine {
    private State soldOutState;
    private State noCoinState;
    private State hasCoinState;
    private State soldState;

    private State state;
    private int count = 0;

    public GumballMachine(int numberOfGumballs) {
        soldOutState = new SoldOutState(this);
        noCoinState = new NoCoinState(this);
        hasCoinState = new HasCoinState(this);
        soldState = new SoldState(this);

        this.count = numberOfGumballs;
        state = (count > 0) ? noCoinState : soldOutState;
    }

    public void insertCoin() {
        state.insertCoin();
    }

    public void ejectCoin() {
        state.ejectCoin();
    }

    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }

    void setState(State state) {
        this.state = state;
    }

    void releaseGumball() {
        if (count > 0) {
            count--;
            System.out.println("A gumball comes rolling out...");
        }
    }
```

```
    public int getCount() {
        return count;
    }

    public State getSoldOutState() {
        return soldOutState;
    }

    public State getNoCoinState() {
        return noCoinState;
    }

    public State getHasCoinState() {
        return hasCoinState;
    }

    public State getSoldState() {
        return soldState;
    }
}
```

Step 4: Testing the Gumball Machine

```
java
Copy code
public class GumballMachineTest {
    public static void main(String[] args) {
        GumballMachine gumballMachine = new GumballMachine(3);

        // Test various interactions
        gumballMachine.insertCoin();
        gumballMachine.turnCrank();

        System.out.println("-----");

        gumballMachine.insertCoin();
        gumballMachine.ejectCoin();
        gumballMachine.turnCrank();

        System.out.println("-----");

        gumballMachine.insertCoin();
        gumballMachine.turnCrank();
        gumballMachine.insertCoin();
        gumballMachine.turnCrank();

        System.out.println("-----");

        gumballMachine.insertCoin();
        gumballMachine.turnCrank();
    }
}
```

Explanation of the Output

1. **Insert Coin**: The machine transitions from `NoCoinState` to `HasCoinState`.
2. **Turn Crank**: This transitions to `SoldState` and dispenses a gumball.
3. **Out of Gumballs**: When the count of gumballs reaches zero, the machine goes into `SoldOutState`.

Sample Output

```
markdown
Copy code
Coin inserted.
Crank turned...
A gumball comes rolling out...
-----

Coin inserted.
Coin ejected.
Insert a coin first.
-----

Coin inserted.
Crank turned...
A gumball comes rolling out...
Coin inserted.
Crank turned...
A gumball comes rolling out...
Out of gumballs!
-----

Out of gumballs, can't insert coin.
```

```
Q.2 Start Raspberry Pi and execute various Linux commands in command terminal
window:
ls, cd, touch, mv, rm, man, mkdir, rmdir, tar, gzip, cat, more, less, ps,
sudo, cron, chown,
chgrp, pingetc.
```

Step 1: Power on and Set Up the Raspberry Pi

1. **Connect** the Raspberry Pi to a monitor, keyboard, and mouse.
2. **Insert the microSD card** with Raspberry Pi OS (or another Linux OS) installed.
3. **Power on** the Raspberry Pi by connecting it to a power source.

Once booted, log into the Raspberry Pi desktop environment or terminal.

Step 2: Open the Terminal

- Open the terminal application from the desktop or press `Ctrl + Alt + T`.

Step 3: Execute Common Linux Commands

You can try executing the following commands, which are commonly used for file and system management in Linux. Here's a brief explanation and example usage of each:

1. **`ls`** - List files and directories in the current directory.

```
bash
Copy code
ls
ls -la  # List all files with details
```

2. **`cd`** - Change the directory.

```
bash
Copy code
cd /home/pi   # Navigate to the /home/pi directory
cd ..         # Move up one directory
```

3. **`touch`** - Create a new, empty file.

```
bash
Copy code
touch example.txt
```

4. **`mv`** - Move or rename a file or directory.

```
bash
Copy code
mv example.txt new_example.txt  # Rename
mv new_example.txt /home/pi/Documents  # Move to another directory
```

5. **`rm`** - Remove files or directories.

```
bash
Copy code
rm new_example.txt  # Remove a file
rm -r /home/pi/Documents/old_folder  # Remove a directory and its
contents
```

6. **`man`** - Display the manual for a command.

```
bash
Copy code
man ls  # Show manual for ls command
```

7. **`mkdir`** - Create a new directory.

```
bash
Copy code
mkdir new_folder
```

8. **`rmdir`** - Remove an empty directory.

```
bash
Copy code
rmdir new_folder
```

9. **`tar`** - Archive files.

```
bash
Copy code
tar -cvf archive.tar /home/pi/Documents  # Create an archive
tar -xvf archive.tar  # Extract the archive
```

10. **`gzip`** - Compress a file with gzip.

```
bash
Copy code
gzip archive.tar  # Compress the file
gunzip archive.tar.gz  # Decompress the file
```

11. **`cat`** - Display contents of a file.

```
bash
Copy code
cat example.txt
```

12. **`more`** and **`less`** - Display file contents page by page.

```
bash
Copy code
more example.txt
less example.txt
```

13. **`ps`** - Display currently running processes.

```
bash
Copy code
ps aux  # Show all processes
```

14. **`sudo`** - Execute a command with superuser privileges.

```
bash
Copy code
sudo apt update  # Update package lists as root
```

15. **`cron`** - Schedule recurring tasks (edit with `crontab`).

```
bash
Copy code
crontab -e  # Edit crontab to schedule a task
```

16. **`chown`** - Change ownership of a file or directory.

```
bash
Copy code
sudo chown pi:pi example.txt  # Change owner to user 'pi'
```

17. **`chgrp`** - Change the group ownership of a file or directory.

```
bash
Copy code
sudo chgrp staff example.txt  # Change group to 'staff'
```

18. **`ping`** - Check network connectivity.

```
bash
Copy code
ping google.com  # Ping Google to check internet connection
```

Tips for Running Commands

- **Run `man <command>`** to learn more about each command and its options.
- **Use `sudo`** carefully as it grants administrator privileges.
- **Practice** with caution when using commands like `rm`, `chown`, or `chmod`, as they can alter system files.

Slip 9 :

Q.1 Design simple HR Application using Spring Framework [15 M]

Step 1: Set Up Spring Boot Project

Create a Spring Boot project using Spring Initializr or your IDE and include these dependencies:

- Spring Web
- Spring Data JPA
- H2 Database
- Thymeleaf

Step 2: Define the Project Structure

The structure could look like this:

```
css
Copy code
src/main/java
└── com
    └── example
        └── hrapp
            ├── controller
            │   └── EmployeeController.java
            ├── model
            │   └── Employee.java
            ├── repository
            │   └── EmployeeRepository.java
            ├── service
            │   └── EmployeeService.java
            ├── HrApplication.java
src/main/resources
└── templates
    ├── employees.html
    ├── add_employee.html
    └── edit_employee.html
application.properties
```

Step 3: Create the Model

Define an `Employee` entity with fields for ID, name, department, and email.

```
java
Copy code
package com.example.hrapp.model;

import javax.persistence.*;

@Entity
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String department;
    private String email;

    // Getters and setters
}
```

Step 4: Create Repository Interface

Define an interface `EmployeeRepository` that extends `JpaRepository`.

```
java
Copy code
package com.example.hrapp.repository;
```

```
import com.example.hrapp.model.Employee;
import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {
}
```

Step 5: Create Service Class

The `EmployeeService` will handle the business logic and communicate with the repository.

```
java
Copy code
package com.example.hrapp.service;

import com.example.hrapp.model.Employee;
import com.example.hrapp.repository.EmployeeRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    public List<Employee> getAllEmployees() {
        return employeeRepository.findAll();
    }

    public Employee getEmployeeById(Long id) {
        return employeeRepository.findById(id).orElse(null);
    }

    public Employee saveEmployee(Employee employee) {
        return employeeRepository.save(employee);
    }

    public void deleteEmployee(Long id) {
        employeeRepository.deleteById(id);
    }
}
```

Step 6: Create Controller

The `EmployeeController` will handle HTTP requests and direct to appropriate views.

```
java
Copy code
package com.example.hrapp.controller;

import com.example.hrapp.model.Employee;
import com.example.hrapp.service.EmployeeService;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

@Controller
@RequestMapping("/employees")
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @GetMapping
    public String getAllEmployees(Model model) {
        model.addAttribute("employees", employeeService.getAllEmployees());
        return "employees";
    }

    @GetMapping("/add")
    public String addEmployeeForm(Model model) {
        model.addAttribute("employee", new Employee());
        return "add_employee";
    }

    @PostMapping("/save")
    public String saveEmployee(@ModelAttribute Employee employee) {
        employeeService.saveEmployee(employee);
        return "redirect:/employees";
    }

    @GetMapping("/edit/{id}")
    public String editEmployeeForm(@PathVariable Long id, Model model) {
        model.addAttribute("employee", employeeService.getEmployeeById(id));
        return "edit_employee";
    }

    @PostMapping("/update/{id}")
    public String updateEmployee(@PathVariable Long id, @ModelAttribute
Employee employee) {
        employee.setId(id);
        employeeService.saveEmployee(employee);
        return "redirect:/employees";
    }

    @GetMapping("/delete/{id}")
    public String deleteEmployee(@PathVariable Long id) {
        employeeService.deleteEmployee(id);
        return "redirect:/employees";
    }
}
```

Step 7: Configure H2 Database (application.properties)

```
properties
Copy code
# application.properties
```

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.h2.console.enabled=true
spring.jpa.hibernate.ddl-auto=update
```

Step 8: Create Thymeleaf Templates

`employees.html` - Display List of Employees

```
html
Copy code
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Employees</title>
</head>
<body>
    <h2>Employee List</h2>
    <a href="/employees/add">Add Employee</a>
    <table border="1">
        <tr>

<th>ID</th><th>Name</th><th>Department</th><th>Email</th><th>Actions</th>
        </tr>
        <tr th:each="employee : ${employees}">
            <td th:text="${employee.id}"></td>
            <td th:text="${employee.name}"></td>
            <td th:text="${employee.department}"></td>
            <td th:text="${employee.email}"></td>
            <td>
                <a
th:href="@{/employees/edit/{id}(id=${employee.id})}">Edit</a> |
                <a
th:href="@{/employees/delete/{id}(id=${employee.id})}">Delete</a>
            </td>
        </tr>
    </table>
</body>
</html>
```

`add_employee.html` - Add Employee Form

```
html
Copy code
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Add Employee</title>
</head>
<body>
    <h2>Add Employee</h2>
    <form th:action="@{/employees/save}" th:object="${employee}"
method="post">
```

```
        <label>Name:</label><input type="text" th:field="*{name}" /><br/>
        <label>Department:</label><input type="text" th:field="*{department}"
/><br/>
        <label>Email:</label><input type="text" th:field="*{email}" /><br/>
        <button type="submit">Save</button>
    </form>
</body>
</html>
```

`edit_employee.html` - Edit Employee Form

Similar to `add_employee.html`, but with fields pre-filled using employee data.

Step 9: Run the Application

Run the Spring Boot application and open the browser at `http://localhost:8080/employees` to see the list of employees and perform CRUD operations.

Q.2 Write python programs on Pi : a) Read your name and print Hello message with name b) Read two numbers and print their sum, difference, product and division. c) Word and character count of a given string. d) Area of a given shape (rectangle, triangle and circle) reading shape and appropriate values from standard input.

a) Read Your Name and Print a Hello Message

```
python
Copy code
# Read name from input and display a hello message
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

---

b) Read Two Numbers and Print Their Sum, Difference, Product, and Division

```
python
Copy code
# Read two numbers from input
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))

# Calculate and display the sum, difference, product, and division
print(f"Sum: {num1 + num2}")
print(f"Difference: {num1 - num2}")
```

```
print(f"Product: {num1 * num2}")
if num2 != 0:
    print(f"Division: {num1 / num2}")
else:
    print("Division: Cannot divide by zero")
```

---

c) Word and Character Count of a Given String

```
python
Copy code
# Read a string from input
text = input("Enter a string: ")

# Calculate word and character count
word_count = len(text.split())
char_count = len(text)

print(f"Word count: {word_count}")
print(f"Character count: {char_count}")
```

---

d) Area of a Given Shape (Rectangle, Triangle, Circle)

```
python
Copy code
import math

# Read the shape type
shape = input("Enter the shape (rectangle, triangle, circle):
").strip().lower()

# Calculate area based on shape
if shape == "rectangle":
    length = float(input("Enter the length: "))
    width = float(input("Enter the width: "))
    area = length * width
    print(f"Area of rectangle: {area}")

elif shape == "triangle":
    base = float(input("Enter the base: "))
    height = float(input("Enter the height: "))
    area = 0.5 * base * height
    print(f"Area of triangle: {area}")

elif shape == "circle":
    radius = float(input("Enter the radius: "))
    area = math.pi * radius * radius
    print(f"Area of circle: {area}")

else:
```

print("Invalid shape entered.")

Slip 10 :

Q.1 Write a Java Program to implement Strategy Pattern for Duck Behavior. Create instance variable that holds current state of Duck from there, we just need to handle all Flying Behaviors and Quack Behavior

Step 1: Define the FlyBehavior Interface

```
java
Copy code
public interface FlyBehavior {
    void fly();
}
```

Step 2: Define Concrete Classes for FlyBehavior

```
java
Copy code
// Concrete Flying Behaviors
public class FlyWithWings implements FlyBehavior {
    @Override
    public void fly() {
        System.out.println("I am flying with wings!");
    }
}

public class FlyNoWay implements FlyBehavior {
    @Override
    public void fly() {
        System.out.println("I can't fly!");
    }
}

public class FlyRocketPowered implements FlyBehavior {
    @Override
    public void fly() {
        System.out.println("I am flying with a rocket!");
    }
}
```

Step 3: Define the QuackBehavior Interface

```
java
Copy code
public interface QuackBehavior {
    void quack();
}
```

Step 4: Define Concrete Classes for QuackBehavior

```
java
Copy code
// Concrete Quacking Behaviors
public class Quack implements QuackBehavior {
    @Override
    public void quack() {
        System.out.println("Quack! Quack!");
    }
}

public class Squeak implements QuackBehavior {
    @Override
    public void quack() {
        System.out.println("Squeak! Squeak!");
    }
}

public class MuteQuack implements QuackBehavior {
    @Override
    public void quack() {
        System.out.println("... (no sound)");
    }
}
```

Step 5: Create the Duck Class

The **Duck** class will use the **FlyBehavior** and **QuackBehavior** interfaces and delegate the behavior to the appropriate strategy.

```
java
Copy code
public abstract class Duck {
    // Instance variables for holding current behaviors
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    // Duck behavior methods
    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }

    // Common Duck method
```

```
    public void swim() {
        System.out.println("All ducks float on water!");
    }

    // Setter methods to change behaviors dynamically
    public void setFlyBehavior(FlyBehavior fb) {
        flyBehavior = fb;
    }

    public void setQuackBehavior(QuackBehavior qb) {
        quackBehavior = qb;
    }

    // Abstract method for display (each type of duck will implement it)
    public abstract void display();
}
```

Step 6: Create Concrete Duck Classes

Now, we can create specific types of ducks, each having a different combination of behaviors.

```
java
Copy code
// Mallard Duck
public class MallardDuck extends Duck {
    public MallardDuck() {
        flyBehavior = new FlyWithWings();
        quackBehavior = new Quack();
    }

    @Override
    public void display() {
        System.out.println("I am a Mallard Duck!");
    }
}

// Model Duck
public class ModelDuck extends Duck {
    public ModelDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new MuteQuack();
    }

    @Override
    public void display() {
        System.out.println("I am a Model Duck!");
    }
}
```

Step 7: Create the DuckSimulator to Test the Strategy Pattern

In the **DuckSimulator** class, we can simulate different ducks and behaviors.

```
java
```

```
Copy code
public class DuckSimulator {
    public static void main(String[] args) {
        // Create a Mallard Duck
        Duck mallard = new MallardDuck();
        mallard.display();
        mallard.performFly();  // Fly with wings
        mallard.performQuack();  // Quack

        System.out.println("\n");

        // Create a Model Duck
        Duck model = new ModelDuck();
        model.display();
        model.performFly();  // Cannot fly
        model.performQuack();  // Mute quack

        // Change behaviors dynamically
        System.out.println("\nChanging Model Duck's behavior...");
        model.setFlyBehavior(new FlyRocketPowered());
        model.setQuackBehavior(new Squeak());

        model.performFly();  // Fly with rocket
        model.performQuack();  // Squeak
    }
}
```

Explanation:

- The **Duck** class is abstract and defines common behaviors (`performFly`, `performQuack`, `swim`), but the actual flying and quacking behaviors are delegated to the **FlyBehavior** and **QuackBehavior** interfaces, respectively.
- **Concrete duck types** (like `MallardDuck` and `ModelDuck`) initialize their behaviors through the constructor.
- The behaviors can be changed dynamically at runtime using the `setFlyBehavior` and `setQuackBehavior` methods.

Output:

```
css
Copy code
I am a Mallard Duck!
I am flying with wings!
Quack! Quack!

I am a Model Duck!
I can't fly!
... (no sound)

Changing Model Duck's behavior...
I am flying with a rocket!
Squeak! Squeak!
```

Q.2 Write python programs on Pi like:
a) Print a name 'n' times, where name and n are read from standard input, using for and
while loops.
b) Handle Divided by Zero Exception.
c) Print current time for 10 times with an interval of10seconds.
d) Read a fileline byline and print the word count of each line

a) Print a Name 'n' Times, Where Name and n are Read from Standard Input, Using For and While Loops

```
python
Copy code
# Using for loop
name = input("Enter the name: ")
n = int(input("Enter the number of times to print the name: "))

# For loop
print("\nUsing for loop:")
for _ in range(n):
    print(name)

# Using while loop
print("\nUsing while loop:")
count = 0
while count < n:
    print(name)
    count += 1
```

---

b) Handle Divided by Zero Exception

```
python
Copy code
try:
    numerator = float(input("Enter numerator: "))
    denominator = float(input("Enter denominator: "))
    result = numerator / denominator
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Division by zero is not allowed!")
except ValueError:
    print("Error: Please enter valid numeric values!")
```

---

c) Print Current Time for 10 Times with an Interval of 10 Seconds

```
python
Copy code
import time
from datetime import datetime

# Loop 10 times with an interval of 10 seconds
for i in range(10):
    current_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    print(f"Current Time: {current_time}")
    time.sleep(10)  # Wait for 10 seconds before the next print
```

---

d) Read a File Line by Line and Print the Word Count of Each Line

```
python
Copy code
# Make sure there is a file named 'sample.txt' in the same directory
filename = input("Enter the file name: ")

try:
    with open(filename, 'r') as file:
        # Read each line from the file
        for line_num, line in enumerate(file, 1):
            word_count = len(line.split())  # Split line into words and count
them
            print(f"Line {line_num}: Word Count = {word_count}")
except FileNotFoundError:
    print("Error: The file does not exist!")
```

Explanation:

1. **Part a**: This program reads a name and the number of times (n) to print that name. It demonstrates both **for** and **while** loops.
2. **Part b**: This program attempts to perform division and handles the ZeroDivisionError exception gracefully, printing an error message if the denominator is zero.
3. **Part c**: The program prints the current time every 10 seconds, using the time.sleep() function to pause for 10 seconds between prints.
4. **Part d**: This program reads a file line by line and prints the word count for each line. It handles FileNotFoundError in case the file doesn't exist.

Slip 11:

Q.1 Write a java program to implement Adapter pattern to design Heart Model to Beat Model

Step-by-Step Solution:

Step 1: Define the BeatMode Interface

This will be the interface that we want to adapt the **HeartModel** to.

```
java
Copy code
public interface BeatMode {
    void beat();
}
```

Step 2: Define the HeartModel Class

This class represents the heart and has its own way of beating.

```
java
Copy code
public class HeartModel {
    // HeartModel's internal beat method
    public void startBeating() {
        System.out.println("Heart is beating...");
    }
}
```

Step 3: Create the HeartAdapter Class

The **HeartAdapter** will adapt the **HeartModel** to the **BeatMode** interface.

```
java
Copy code
public class HeartAdapter implements BeatMode {
    private HeartModel heartModel;

    // Constructor that takes the HeartModel instance
    public HeartAdapter(HeartModel heartModel) {
        this.heartModel = heartModel;
    }

    // Implement the beat method to call the HeartModel's startBeating
    @Override
    public void beat() {
        heartModel.startBeating();
    }
}
```

Step 4: Create the Client Code to Test the Adapter

The client will use the **BeatMode** interface without knowing about the underlying **HeartModel** class.

```
java
Copy code
public class HeartClient {
    public static void main(String[] args) {
        // Create a HeartModel object
        HeartModel heart = new HeartModel();

        // Create a HeartAdapter that adapts the HeartModel to the BeatMode
interface
        BeatMode beatMode = new HeartAdapter(heart);

        // Use the beat method via the BeatMode interface
        beatMode.beat();
    }
}
```

Explanation:

1. **BeatMode Interface**: This is the target interface that defines the method `beat()`. The client code expects objects of this type to perform the beating action.
2. **HeartModel Class**: This is the existing class with a method `startBeating()` that is not compatible with the `BeatMode` interface. It represents a heart in a specific mode that beats.
3. **HeartAdapter Class**: This class acts as the adapter. It takes an instance of `HeartModel` and delegates the `beat()` method call to the `startBeating()` method of `HeartModel`. This makes the `HeartModel` compatible with the `BeatMode` interface.
4. **HeartClient**: This is the client code that uses the `BeatMode` interface without knowing the underlying implementation. It works with the `HeartAdapter` to interact with the `HeartModel`.

Output:

```
csharp
Copy code
Heart is beating...
```

Q.2 Run some python programs on Pi like a) Light an LED through Python program b) Get input from two switches and switch on corresponding LEDs c) Flash an LED at a given on time and off time cycle, where the two times are taken from a file

Setup:

Before running these programs, ensure you have:

- The **GPIO library** installed. You can install it using:

  ```
  bash
  Copy code
  sudo apt-get install python3-rpi.gpio
  ```

- Proper connections for LEDs and switches. You can use a breadboard to connect the GPIO pins to LEDs and switches, ensuring you know which GPIO pins are connected to them.

---

a) Light an LED Through Python Program

This program will turn on an LED connected to a GPIO pin (e.g., GPIO17) when the program runs.

```
python
Copy code
import RPi.GPIO as GPIO
import time

# Set up GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

# Set GPIO pin 17 as output (LED pin)
LED_PIN = 17
GPIO.setup(LED_PIN, GPIO.OUT)

# Turn on the LED
GPIO.output(LED_PIN, GPIO.HIGH)
print("LED is ON")

# Wait for 5 seconds
time.sleep(5)

# Turn off the LED
GPIO.output(LED_PIN, GPIO.LOW)
print("LED is OFF")

# Clean up GPIO settings
GPIO.cleanup()
```

Explanation:

- This program uses GPIO pin 17 to control the LED.
- It turns the LED on for 5 seconds, then turns it off.
- The `GPIO.cleanup()` ensures that the GPIO settings are cleared when the program ends.

b) Get Input from Two Switches and Switch On Corresponding LEDs

This program will read input from two switches and turn on the corresponding LEDs based on the state of the switches.

```
python
Copy code
import RPi.GPIO as GPIO
import time

# Set up GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

# Define GPIO pins for switches and LEDs
SWITCH1_PIN = 18
SWITCH2_PIN = 23
LED1_PIN = 17
LED2_PIN = 24

# Set up the switches as inputs and LEDs as outputs
GPIO.setup(SWITCH1_PIN, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(SWITCH2_PIN, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(LED1_PIN, GPIO.OUT)
GPIO.setup(LED2_PIN, GPIO.OUT)

try:
    while True:
        # Read switch states
        switch1_state = GPIO.input(SWITCH1_PIN)
        switch2_state = GPIO.input(SWITCH2_PIN)

        # Control LEDs based on switch states
        if switch1_state == GPIO.LOW:  # Button pressed (active low)
            GPIO.output(LED1_PIN, GPIO.HIGH)
        else:
            GPIO.output(LED1_PIN, GPIO.LOW)

        if switch2_state == GPIO.LOW:
            GPIO.output(LED2_PIN, GPIO.HIGH)
        else:
            GPIO.output(LED2_PIN, GPIO.LOW)

        time.sleep(0.1)  # Debounce delay

except KeyboardInterrupt:
    print("Program terminated")

finally:
    # Clean up GPIO settings
    GPIO.cleanup()
```

Explanation:

- The program monitors two switches (connected to GPIO pins 18 and 23) and two LEDs (connected to GPIO pins 17 and 24).
- If a switch is pressed (input is `LOW`), the corresponding LED will turn on. If the switch is not pressed (`HIGH`), the LED will turn off.
- The program continues to monitor the switches in a loop, and a `try-except` block is used to cleanly handle the program termination (e.g., via `Ctrl+C`).

---

c) Flash an LED at a Given On Time and Off Time Cycle, Where the Two Times Are Taken from a File

This program will read on and off times from a file (e.g., `times.txt`) and flash an LED based on those times.

Sample File (`times.txt`):

```
Copy code
2
1
```

This means the LED will stay on for 2 seconds and off for 1 second.

```
python
Copy code
import RPi.GPIO as GPIO
import time

# Set up GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

# Set GPIO pin for the LED
LED_PIN = 17
GPIO.setup(LED_PIN, GPIO.OUT)

# Read on and off times from a file
try:
    with open('times.txt', 'r') as file:
        on_time = float(file.readline().strip())  # Read the 'on' time
        off_time = float(file.readline().strip())  # Read the 'off' time
except FileNotFoundError:
    print("Error: times.txt file not found!")
    GPIO.cleanup()
    exit()

# Flash the LED based on the times read from the file
try:
    while True:
```

```
        GPIO.output(LED_PIN, GPIO.HIGH)  # Turn on the LED
        print(f"LED ON for {on_time} seconds")
        time.sleep(on_time)  # Wait for 'on' time

        GPIO.output(LED_PIN, GPIO.LOW)  # Turn off the LED
        print(f"LED OFF for {off_time} seconds")
        time.sleep(off_time)  # Wait for 'off' time

except KeyboardInterrupt:
    print("Program terminated")

finally:
    GPIO.cleanup()
```

Explanation:

- The program reads the on-time and off-time values from a file (`times.txt`).
- It flashes the LED on for the specified `on_time` and off for the specified `off_time` in an infinite loop.
- The `try-except` block ensures that the program can be stopped safely using `Ctrl+C`, and the GPIO pins are cleaned up afterward.

Slip 12 :

Q.1 Write a Java Program to implement Decorator Pattern for interface Car to define the assemble() method and then decorate it to Sports car and Luxury Car

Step-by-Step Solution:

Step 1: Define the `Car` Interface

This interface will have a single method `assemble()`.

```
java
Copy code
public interface Car {
    void assemble();
}
```

Step 2: Concrete Implementation of `Car` — Basic Car

```
java
Copy code
public class BasicCar implements Car {
    @Override
    public void assemble() {
        System.out.println("Basic Car.");
    }
}
```

Step 3: Decorator Class — CarDecorator

This will be a base class for all decorators. It implements the `Car` interface and wraps a `Car` object.

```
java
Copy code
public class CarDecorator implements Car {
    protected Car decoratedCar;

    public CarDecorator(Car car) {
        this.decoratedCar = car;
    }

    @Override
    public void assemble() {
        this.decoratedCar.assemble();
    }
}
```

Step 4: SportsCar Decorator

This decorator class will add additional features specific to a sports car.

```
java
Copy code
public class SportsCar extends CarDecorator {
    public SportsCar(Car car) {
        super(car);
    }

    @Override
    public void assemble() {
        super.assemble();  // Call the base assemble method
        System.out.println("Adding features of Sports Car.");
    }
}
```

Step 5: LuxuryCar Decorator

This decorator class will add additional features specific to a luxury car.

```
java
Copy code
```

```
public class LuxuryCar extends CarDecorator {
    public LuxuryCar(Car car) {
        super(car);
    }

    @Override
    public void assemble() {
        super.assemble();  // Call the base assemble method
        System.out.println("Adding features of Luxury Car.");
    }
}
```

Step 6: Client Code — Testing the Decorator Pattern

The client will create a basic car and then dynamically decorate it with different features.

```
java
Copy code
public class DecoratorPatternTest {
    public static void main(String[] args) {
        // Create a basic car
        Car sportsCar = new SportsCar(new BasicCar());
        sportsCar.assemble();
        System.out.println("\n*****");

        // Create a luxury sports car
        Car sportsLuxuryCar = new LuxuryCar(new SportsCar(new BasicCar()));
        sportsLuxuryCar.assemble();
    }
}
```

Output:

```
markdown
Copy code
Basic Car.
Adding features of Sports Car.

*****
Basic Car.
Adding features of Sports Car.
Adding features of Luxury Car.
```

Q.2 Write a program to sense the available networks using Arduino [15 M]

```
#include <ESP8266WiFi.h>  // Include the ESP8266 Wi-Fi library

void setup() {
  // Start the Serial communication at 115200 baud rate
  Serial.begin(115200);
```

```
  // Connect to Wi-Fi (not necessary for scanning, but useful for testing)
  // WiFi.begin("your-SSID", "your-password");

  // Print a message to the Serial Monitor
  Serial.println("Scanning for Wi-Fi networks...");

  // Start the Wi-Fi scan
  int numberOfNetworks = WiFi.scanNetworks();  // Returns the number of
networks found

  // Check if any networks were found
  if (numberOfNetworks == 0) {
    Serial.println("No networks found.");
  } else {
    Serial.print(numberOfNetworks);
    Serial.println(" networks found:");
  }

  // Loop through all the networks and print their details
  for (int i = 0; i < numberOfNetworks; i++) {
    // Print network SSID
    Serial.print(i + 1);
    Serial.print(": ");
    Serial.print(WiFi.SSID(i));

    // Print signal strength in dBm
    Serial.print("  Signal Strength (RSSI): ");
    Serial.print(WiFi.RSSI(i));
    Serial.println(" dBm");

    // Print encryption type (Security)
    Serial.print("  Encryption: ");
    switch (WiFi.encryptionType(i)) {
      case WIFI_AUTH_OPEN:
        Serial.println("Open");
        break;
      case WIFI_AUTH_WEP:
        Serial.println("WEP");
        break;
      case WIFI_AUTH_WPA_PSK:
        Serial.println("WPA-PSK");
        break;
      case WIFI_AUTH_WPA2_PSK:
        Serial.println("WPA2-PSK");
        break;
      case WIFI_AUTH_WPA_WPA2_PSK:
        Serial.println("WPA/WPA2-PSK");
        break;
      default:
        Serial.println("Unknown");
        break;
    }
    Serial.println();  // Print a newline for separation between networks
  }
}

void loop() {
```

```
  // The loop doesn't do anything in this case
}
```

Slip 13 :

Q.1 Write a Java Program to implement an Adapter design pattern in mobile charger. Define two classes – Volt (to measure volts) and Socket (producing constant volts of 120V). Build an adapter that can produce 3 volts, 12 volts and default 120 volts. Implements Adapter pattern using Class Adapter

Step 1: Define the Volt Class

The `Volt` class represents the voltage and will have methods to get the voltage in different units.

```
java
Copy code
public class Volt {
    private int volts;

    public Volt(int volts) {
        this.volts = volts;
    }

    public int getVolts() {
        return volts;
    }
}
```

Step 2: Define the Socket Class

The `Socket` class represents a socket that provides 120V (constant voltage).

```
java
Copy code
public class Socket {
    public Volt getVolt() {
        return new Volt(120); // Socket provides 120V by default
    }
}
```

Step 3: Define the Adapter Class

In the **Class Adapter** design pattern, the adapter extends the existing class (`Socket`) to modify the behavior. We will create an adapter that can return different voltages (3V, 12V, or 120V).

```
java
```

```
Copy code
public class SocketAdapter extends Socket {

    // Convert 120V to 3V
    public Volt get3Volt() {
        Volt v = getVolt();  // Get 120V from the Socket
        return convertVolt(v, 40);  // 120V / 40 = 3V
    }

    // Convert 120V to 12V
    public Volt get12Volt() {
        Volt v = getVolt();  // Get 120V from the Socket
        return convertVolt(v, 10);  // 120V / 10 = 12V
    }

    // No conversion, just return 120V
    public Volt get120Volt() {
        return getVolt();  // Return 120V
    }

    // Helper method to convert voltage
    private Volt convertVolt(Volt v, int divider) {
        return new Volt(v.getVolts() / divider);
    }
}
```

Step 4: Test the Adapter Pattern

Finally, we will create a **main class** to test the functionality of the `Adapter` pattern by creating a `SocketAdapter` and using it to get 3V, 12V, and 120V.

```
java
Copy code
public class AdapterPatternTest {

    public static void main(String[] args) {
        SocketAdapter socketAdapter = new SocketAdapter();

        // Getting 120V from the socket
        System.out.println("Voltage obtained from socket: " +
socketAdapter.get120Volt().getVolts() + "V");

        // Getting 12V using the adapter
        System.out.println("Voltage after conversion to 12V: " +
socketAdapter.get12Volt().getVolts() + "V");

        // Getting 3V using the adapter
        System.out.println("Voltage after conversion to 3V: " +
socketAdapter.get3Volt().getVolts() + "V");
    }
}
```

Output:

```
vbnet
Copy code
Voltage obtained from socket: 120V
Voltage after conversion to 12V: 12V
Voltage after conversion to 3V: 3V
```

Explanation:

1. **Volt Class**: This class represents a voltage and has a method `getVolts()` that returns the voltage.
2. **Socket Class**: This class produces a constant voltage of 120V by default, which is returned by the `getVolt()` method.
3. **SocketAdapter Class**: This class extends `Socket` and provides methods to convert the 120V to 3V, 12V, or 120V. It uses a helper method `convertVolt()` to adjust the voltage based on the required value.
4. **AdapterPatternTest**: The test class creates an instance of `SocketAdapter` and calls its methods to get voltages of 3V, 12V, and 120V.

Q.2 Write a program to measure the distance using ultrasonic sensor and make LED blink using Arduino.

```
#define TRIG_PIN 9    // Trigger pin of HC-SR04
#define ECHO_PIN 10   // Echo pin of HC-SR04
#define LED_PIN 13    // LED pin

void setup() {
  // Initialize the serial communication
  Serial.begin(9600);

  // Initialize the pins
  pinMode(TRIG_PIN, OUTPUT);
  pinMode(ECHO_PIN, INPUT);
  pinMode(LED_PIN, OUTPUT);

  // Make sure the LED is off initially
  digitalWrite(LED_PIN, LOW);
}

void loop() {
  // Send a pulse to trigger the ultrasonic sensor
  digitalWrite(TRIG_PIN, LOW);  // Ensure the trigger pin is LOW
  delayMicroseconds(2);
  digitalWrite(TRIG_PIN, HIGH);  // Send the pulse
  delayMicroseconds(10);
  digitalWrite(TRIG_PIN, LOW);   // Stop the pulse
```

```
  // Measure the time for the pulse to return
  long duration = pulseIn(ECHO_PIN, HIGH);

  // Calculate the distance in cm (Speed of sound = 343 m/s = 0.0343 cm/µs)
  long distance = duration * 0.0343 / 2;

  // Print the distance to the Serial Monitor
  Serial.print("Distance: ");
  Serial.print(distance);
  Serial.println(" cm");

  // If the distance is less than 20 cm, blink the LED
  if (distance < 20) {
    digitalWrite(LED_PIN, HIGH);  // Turn LED on
    delay(500);                   // Wait for 500ms
    digitalWrite(LED_PIN, LOW);   // Turn LED off
    delay(500);                   // Wait for 500ms
  } else {
    digitalWrite(LED_PIN, LOW);   // Keep LED off if distance > 20 cm
  }

  // Add a small delay before taking the next reading
  delay(100);
}
```

Slip 14 :

Q.1 Write a Java Program to implement Command Design Pattern for Command Interface with execute() . Use this to create variety of commands for LightOnCommand, LightOffCommand, GarageDoorUpCommand, StereoOnWithCDComman.

1. Command Interface

```
java
Copy code
// Command Interface
public interface Command {
    void execute();
}
```

2. Receiver Classes

```
java
Copy code
// Light Class (Receiver)
public class Light {
    public void on() {
        System.out.println("Light is ON");
    }

    public void off() {
        System.out.println("Light is OFF");
    }
}

// GarageDoor Class (Receiver)
public class GarageDoor {
    public void up() {
        System.out.println("Garage Door is UP");
    }

    public void down() {
        System.out.println("Garage Door is DOWN");
    }
}

// Stereo Class (Receiver)
public class Stereo {
    public void on() {
        System.out.println("Stereo is ON");
    }

    public void off() {
        System.out.println("Stereo is OFF");
    }

    public void setCD() {
        System.out.println("CD is set in Stereo");
    }

    public void setVolume(int volume) {
        System.out.println("Stereo volume is set to " + volume);
    }
}
```

3. Concrete Command Classes

```
java
Copy code
// LightOnCommand (Concrete Command)
public class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }
```

```
    @Override
    public void execute() {
        light.on();
    }
}

// LightOffCommand (Concrete Command)
public class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.off();
    }
}

// GarageDoorUpCommand (Concrete Command)
public class GarageDoorUpCommand implements Command {
    private GarageDoor garageDoor;

    public GarageDoorUpCommand(GarageDoor garageDoor) {
        this.garageDoor = garageDoor;
    }

    @Override
    public void execute() {
        garageDoor.up();
    }
}

// StereoOnWithCDCommand (Concrete Command)
public class StereoOnWithCDCommand implements Command {
    private Stereo stereo;

    public StereoOnWithCDCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    @Override
    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);  // Setting a default volume level
    }
}
```

4. The Remote Control (Invoker)

```
java
Copy code
// RemoteControl Class (Invoker)
```

```
public class RemoteControl {
    private Command[] commands;

    public RemoteControl() {
        commands = new Command[4]; // You can add more commands here
    }

    // Set the command at a specific position
    public void setCommand(int slot, Command command) {
        commands[slot] = command;
    }

    // Press the button to execute the command
    public void pressButton(int slot) {
        commands[slot].execute();
    }
}
```

5. Main Class to Test the Command Pattern

```
java
Copy code
public class CommandPatternTest {
    public static void main(String[] args) {
        // Creating receivers
        Light livingRoomLight = new Light();
        GarageDoor garageDoor = new GarageDoor();
        Stereo stereo = new Stereo();

        // Creating concrete commands
        Command lightOn = new LightOnCommand(livingRoomLight);
        Command lightOff = new LightOffCommand(livingRoomLight);
        Command garageDoorUp = new GarageDoorUpCommand(garageDoor);
        Command stereoOnWithCD = new StereoOnWithCDCommand(stereo);

        // Creating the remote control (Invoker)
        RemoteControl remote = new RemoteControl();

        // Setting commands to the remote control
        remote.setCommand(0, lightOn);
        remote.setCommand(1, lightOff);
        remote.setCommand(2, garageDoorUp);
        remote.setCommand(3, stereoOnWithCD);

        // Pressing the buttons to execute the commands
        System.out.println("Pressing button 0 (Light On):");
        remote.pressButton(0);

        System.out.println("\nPressing button 1 (Light Off):");
        remote.pressButton(1);

        System.out.println("\nPressing button 2 (Garage Door Up):");
        remote.pressButton(2);

        System.out.println("\nPressing button 3 (Stereo On with CD):");
        remote.pressButton(3);
```

```
    }
}
```

Explanation:

1. **Command Interface**: Defines the `execute()` method which will be implemented by all concrete command classes.
2. **Receiver Classes**: The `Light`, `GarageDoor`, and `Stereo` classes represent the devices being controlled. They perform the actions when their corresponding commands are executed.
3. **Concrete Command Classes**: These implement the `Command` interface and delegate the action to the appropriate method in the receiver class. For example, `LightOnCommand` calls the `on()` method of the `Light` class.
4. **Invoker**: The `RemoteControl` class holds a list of commands and can execute the appropriate command when the button is pressed. The `setCommand()` method binds a command to a button, and `pressButton()` executes the command.
5. **Main**: In the `CommandPatternTest` class, we create instances of receivers and commands, set them in the remote control, and then simulate pressing buttons on the remote control to execute various actions.

Output:

```
vbnet
Copy code
Pressing button 0 (Light On):
Light is ON

Pressing button 1 (Light Off):
Light is OFF

Pressing button 2 (Garage Door Up):
Garage Door is UP

Pressing button 3 (Stereo On with CD):
Stereo is ON
CD is set in Stereo
Stereo volume is set to 11
```

Q.2 Write a program to detects the vibration of an object with sensor using Arduino.

```
#define VIBRATION_SENSOR_PIN 8 // Pin for vibration sensor
#define LED_PIN 13             // Pin for LED (optional)

void setup() {
  // Initialize the vibration sensor pin as input
  pinMode(VIBRATION_SENSOR_PIN, INPUT);
```

```
  // Initialize the LED pin as output
  pinMode(LED_PIN, OUTPUT);

  // Start the Serial Monitor
  Serial.begin(9600);
}

void loop() {
  // Read the state of the vibration sensor
  int sensorState = digitalRead(VIBRATION_SENSOR_PIN);

  // Check if vibration is detected
  if (sensorState == LOW) { // Sensor output LOW indicates vibration detected
    Serial.println("Vibration Detected!");
    digitalWrite(LED_PIN, HIGH);  // Turn on LED (if connected)
    delay(200);  // Keep LED on for 200ms
    digitalWrite(LED_PIN, LOW);   // Turn off LED
  } else {
    Serial.println("No Vibration");
  }

  // Delay before the next reading
  delay(100);  // Delay for 100ms before checking again
}
```

Slip 15 :

Q.1 Write a Java Program to implement Facade Design Pattern for Home Theater

1. Subsystems (Components)

```
java
Copy code
// TV Class (Subsystem)
public class TV {
    public void on() {
        System.out.println("Turning on the TV.");
    }
```

```
    public void off() {
        System.out.println("Turning off the TV.");
    }
}

// SoundSystem Class (Subsystem)
public class SoundSystem {
    public void on() {
        System.out.println("Turning on the sound system.");
    }

    public void off() {
        System.out.println("Turning off the sound system.");
    }

    public void setVolume(int volume) {
        System.out.println("Setting sound system volume to " + volume);
    }
}

// Lights Class (Subsystem)
public class Lights {
    public void dim() {
        System.out.println("Dimming the lights.");
    }

    public void on() {
        System.out.println("Turning on the lights.");
    }
}

// DVDPlayer Class (Subsystem)
public class DVDPlayer {
    public void on() {
        System.out.println("Turning on the DVD player.");
    }

    public void off() {
        System.out.println("Turning off the DVD player.");
    }

    public void play() {
        System.out.println("Playing the DVD.");
    }

    public void stop() {
        System.out.println("Stopping the DVD.");
    }
}
```

2. Facade Class

```
java
Copy code
// HomeTheaterFacade Class (Facade)
public class HomeTheaterFacade {
```

```
    private TV tv;
    private SoundSystem soundSystem;
    private Lights lights;
    private DVDPlayer dvdPlayer;

    public HomeTheaterFacade(TV tv, SoundSystem soundSystem, Lights lights,
DVDPlayer dvdPlayer) {
        this.tv = tv;
        this.soundSystem = soundSystem;
        this.lights = lights;
        this.dvdPlayer = dvdPlayer;
    }

    public void watchMovie() {
        System.out.println("Get ready to watch a movie...");
        lights.dim();
        tv.on();
        soundSystem.on();
        soundSystem.setVolume(10);
        dvdPlayer.on();
        dvdPlayer.play();
    }

    public void endMovie() {
        System.out.println("Shutting down the movie...");
        dvdPlayer.stop();
        dvdPlayer.off();
        soundSystem.off();
        tv.off();
        lights.on();
    }
}
```

3. Client Code (Test)

```
java
Copy code
// Client Code (Main)
public class FacadePatternTest {
    public static void main(String[] args) {
        // Creating instances of subsystems (components)
        TV tv = new TV();
        SoundSystem soundSystem = new SoundSystem();
        Lights lights = new Lights();
        DVDPlayer dvdPlayer = new DVDPlayer();

        // Creating the facade
        HomeTheaterFacade homeTheater = new HomeTheaterFacade(tv,
soundSystem, lights, dvdPlayer);

        // Using the facade to simplify the process of watching a movie
        homeTheater.watchMovie();

        System.out.println("\n--- Movie finished ---\n");

        // Using the facade to end the movie and turn off the systems
```

```
        homeTheater.endMovie();
    }
}
```

Explanation:

1. **Subsystem Classes**: These classes (`TV`, `SoundSystem`, `Lights`, and `DVDPlayer`) represent the individual components of the home theater system. Each class has methods to control its respective functionality, such as turning on/off, adjusting volume, or playing a DVD.
2. **Facade Class** (`HomeTheaterFacade`): This class provides a simplified interface to control all the components of the home theater system. It has methods like `watchMovie()` and `endMovie()` that internally call the appropriate methods on the subsystem objects. The facade hides the complexity of interacting with each component.
3. **Client Code**: In the `FacadePatternTest` class, we create instances of the subsystems (TV, SoundSystem, Lights, and DVDPlayer) and pass them to the `HomeTheaterFacade`. The client can then simply call `watchMovie()` or `endMovie()` without needing to deal with the individual components of the system.

Output:

```
vbnet
Copy code
Get ready to watch a movie...
Dimming the lights.
Turning on the TV.
Turning on the sound system.
Setting sound system volume to 10
Turning on the DVD player.
Playing the DVD.

--- Movie finished ---

Stopping the DVD.
Turning off the DVD player.
Turning off the sound system.
Turning off the TV.
Turning on the lights.
```

Q.2 Write a program to sense a finger when it is placed on the board Arduino. [15 M]

```
#define TOUCH_SENSOR_PIN 8  // Pin for the capacitive touch sensor
#define LED_PIN 13          // Pin for the LED (optional)

void setup() {
  // Initialize the touch sensor pin as input
  pinMode(TOUCH_SENSOR_PIN, INPUT);
```

```
  // Initialize the LED pin as output
  pinMode(LED_PIN, OUTPUT);

  // Start the Serial Monitor
  Serial.begin(9600);
}

void loop() {
  // Read the state of the touch sensor
  int sensorState = digitalRead(TOUCH_SENSOR_PIN);

  // Check if a finger is placed on the sensor
  if (sensorState == LOW) { // TTP223 sensor returns LOW when touched
    Serial.println("Finger detected!");
    digitalWrite(LED_PIN, HIGH);  // Turn on LED (if connected)
  } else {
    Serial.println("No finger detected.");
    digitalWrite(LED_PIN, LOW);   // Turn off LED
  }

  // Delay before the next reading
  delay(100);  // Delay for 100ms before checking again
}
```

Slip 16 :

Q.1 Write a Java Program to implement Observer Design Pattern for number conversion. Accept a number in Decimal form and represent it in Hexadecimal, Octal and Binary. Change the Number and it reflects in other forms also

1. Observer Interface

```
java
Copy code
// Observer Interface
public interface Observer {
    void update(int decimalNumber);
}
```

2. Concrete Observers (Hexadecimal, Octal, Binary)

```
java
Copy code
// Concrete Observer for Hexadecimal format
public class HexadecimalObserver implements Observer {
    @Override
    public void update(int decimalNumber) {
        System.out.println("Hexadecimal: " +
Integer.toHexString(decimalNumber).toUpperCase());
    }
}

// Concrete Observer for Octal format
public class OctalObserver implements Observer {
    @Override
    public void update(int decimalNumber) {
        System.out.println("Octal: " + Integer.toOctalString(decimalNumber));
    }
}

// Concrete Observer for Binary format
public class BinaryObserver implements Observer {
    @Override
    public void update(int decimalNumber) {
        System.out.println("Binary: " +
Integer.toBinaryString(decimalNumber));
    }
}
```

3. Subject (DecimalNumber)

```
java
Copy code
// Subject that holds the decimal number and notifies observers
import java.util.ArrayList;
import java.util.List;

public class DecimalNumber {
    private int decimalNumber;
    private List<Observer> observers = new ArrayList<>();

    // Attach an observer
    public void attach(Observer observer) {
        observers.add(observer);
    }

    // Detach an observer
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    // Set the decimal number and notify observers
    public void setDecimalNumber(int decimalNumber) {
        this.decimalNumber = decimalNumber;
        notifyObservers();
    }
```

```
    // Notify all observers about the change
    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(decimalNumber);
        }
    }
}
```

4. Main Program (Client Code)

```
java
Copy code
import java.util.Scanner;

public class ObserverPatternTest {
    public static void main(String[] args) {
        // Create the Subject (DecimalNumber)
        DecimalNumber decimalNumber = new DecimalNumber();

        // Create the Observers
        HexadecimalObserver hexObserver = new HexadecimalObserver();
        OctalObserver octObserver = new OctalObserver();
        BinaryObserver binObserver = new BinaryObserver();

        // Attach observers to the subject
        decimalNumber.attach(hexObserver);
        decimalNumber.attach(octObserver);
        decimalNumber.attach(binObserver);

        // Accept input from user
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a decimal number: ");
        int number = scanner.nextInt();

        // Set the decimal number and update all observers
        decimalNumber.setDecimalNumber(number);

        // Allow the user to change the number
        System.out.print("\nEnter a new decimal number: ");
        number = scanner.nextInt();

        // Update observers with the new number
        decimalNumber.setDecimalNumber(number);

        scanner.close();
    }
}
```

Q.2 Write a program to connect with the available Wi-Fi using Arduino. [15 M]

```
#include <ESP8266WiFi.h>  // Include the Wi-Fi library for ESP8266
```

```
// Replace these with your network credentials
const char* ssid = "your-SSID";       // Wi-Fi SSID (name of the Wi-Fi network)
const char* password = "your-PASSWORD"; // Wi-Fi Password

void setup() {
  // Start the Serial communication
  Serial.begin(115200);

  // Connect to Wi-Fi
  Serial.println("Connecting to Wi-Fi...");
  WiFi.begin(ssid, password);  // Start the connection using SSID and password

  // Wait for the connection to establish
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  // Once connected, print the local IP address
  Serial.println("");
  Serial.println("Connected to Wi-Fi!");
  Serial.print("IP Address: ");
  Serial.println(WiFi.localIP());  // Print the IP address assigned to the ESP8266
}

void loop() {
  // You can add additional functionality here if needed
}
```

Slip 17 :

Q.1 Write a Java Program to implement Abstract Factory Pattern for Shape interface.

Code Implementation:

1. Shape Interface

```
java
Copy code
// Abstract Product: Shape interface
public interface Shape {
    void draw();
}
```

2. Concrete Products (Circle, Square, Rectangle)

```
java
Copy code
// Concrete Product: Circle
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}

// Concrete Product: Square
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Square");
    }
}

// Concrete Product: Rectangle
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle");
    }
}
```

3. Abstract Factory

```
java
Copy code
// Abstract Factory: ShapeFactory
public interface ShapeFactory {
    Shape createShape(String shapeType);
}
```

4. Concrete Factories (2D and 3D)

```
java
Copy code
// Concrete Factory: 2DShapeFactory
public class TwoDShapeFactory implements ShapeFactory {
    @Override
    public Shape createShape(String shapeType) {
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new Square();
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        }
        return null;
    }
}
```

```
// Concrete Factory: 3DShapeFactory (can create 3D shapes like Sphere,
Cuboid, etc.)
public class ThreeDShapeFactory implements ShapeFactory {
    // In this example, we are assuming that 3D shapes like Sphere and Cuboid
could be created
    // But for simplicity, we will keep it similar to 2D factory structure.
    @Override
    public Shape createShape(String shapeType) {
        if (shapeType.equalsIgnoreCase("SPHERE")) {
            // Return a 3D shape like Sphere (for now, it's a placeholder).
            return new Circle();  // Just using Circle here as an example of
a 3D object.
        } else if (shapeType.equalsIgnoreCase("CUBOID")) {
            // Return a 3D shape like Cuboid (for now, it's a placeholder).
            return new Square();  // Just using Square here as an example of
a 3D object.
        }
        return null;
    }
}
```

5. Client Code

```
java
Copy code
public class AbstractFactoryPatternDemo {
    public static void main(String[] args) {
        // Create 2D Shape Factory
        ShapeFactory shapeFactory2D = new TwoDShapeFactory();

        // Create shapes using the 2D Shape Factory
        Shape shape1 = shapeFactory2D.createShape("CIRCLE");
        shape1.draw();

        Shape shape2 = shapeFactory2D.createShape("SQUARE");
        shape2.draw();

        Shape shape3 = shapeFactory2D.createShape("RECTANGLE");
        shape3.draw();

        // Create 3D Shape Factory
        ShapeFactory shapeFactory3D = new ThreeDShapeFactory();

        // Create shapes using the 3D Shape Factory
        Shape shape4 = shapeFactory3D.createShape("SPHERE");
        shape4.draw();

        Shape shape5 = shapeFactory3D.createShape("CUBOID");
        shape5.draw();
    }
}
```

Explanation of the Code:

1. **Shape Interface**: This defines the `draw()` method that all concrete shapes (e.g., Circle, Square, Rectangle) must implement.
2. **Concrete Shapes**:
    - `Circle`, `Square`, and `Rectangle` implement the `Shape` interface and provide their respective `draw()` method implementations.
3. **Abstract Factory** (`ShapeFactory`): This interface defines a `createShape()` method, which will be used by concrete factories to create specific shapes based on input.
4. **Concrete Factories**:
    - `TwoDShapeFactory` is responsible for creating **2D shapes** like Circle, Square, and Rectangle.
    - `ThreeDShapeFactory` can be extended to create **3D shapes** like Sphere and Cuboid (though for simplicity, we're using placeholders in this example).
5. **Client Code**: The client (in this case, `AbstractFactoryPatternDemo`) interacts with the abstract factories to create shapes. It can use the `TwoDShapeFactory` to create 2D shapes and `ThreeDShapeFactory` to create 3D shapes.

Q.2 Write a program to get temperature notification using Arduino.

```
#include <DHT.h>

// Define the pin connected to the DHT sensor
#define DHTPIN 2  // Pin where the DHT sensor data pin is connected

// Define the sensor type (DHT11 or DHT22)
#define DHTTYPE DHT11  // Use DHT11 or DHT22 depending on your sensor

// Initialize the DHT sensor
DHT dht(DHTPIN, DHTTYPE);

void setup() {
  // Start the serial communication
  Serial.begin(9600);

  // Initialize the DHT sensor
  dht.begin();

  Serial.println("Temperature Notification System Initialized");
}

void loop() {
  // Wait a few seconds between measurements
  delay(2000);
```

```
  // Read the temperature in Celsius
  float tempC = dht.readTemperature();

  // Check if the reading failed and exit early
  if (isnan(tempC)) {
    Serial.println("Failed to read from DHT sensor!");
    return;
  }

  // Print the temperature to the Serial Monitor
  Serial.print("Current Temperature: ");
  Serial.print(tempC);
  Serial.println("°C");

  // If temperature is higher than a certain threshold, send notification
  if (tempC > 30.0) {
    Serial.println("Temperature is too high! Sending notification...");
    sendTemperatureNotification(tempC);
  } else if (tempC < 10.0) {
    Serial.println("Temperature is too low! Sending notification...");
    sendTemperatureNotification(tempC);
  }

  // Delay to prevent continuous checking
  delay(1000);
}

void sendTemperatureNotification(float temperature) {
  // Simulating sending a notification (you can integrate with email/SMS here)
  Serial.print("Notification: ");
  Serial.print("Temperature Alert! Current Temperature is ");
  Serial.print(temperature);
  Serial.println("°C");
  // You can integrate with services like IFTTT, email, or SMS here
}
```

Slip 18 :

Q.1 Write a JAVA Program to implement built-in support (java.util.Observable) Weather station with members temperature, humidity, pressure and methods mesurmentsChanged(), setMesurment(), getTemperature(), getHumidity(), getPressure()

Weather Station Implementation:

1. WeatherStation (Observable)

```
java
Copy code
import java.util.Observable;

public class WeatherStation extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    // Constructor
    public WeatherStation() {}

    // Set the measurements and notify observers
    public void setMeasurements(float temperature, float humidity, float
pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();  // Notify observers
    }

    // Notify observers that measurements have changed
    private void measurementsChanged() {
        setChanged();  // Marks the Observable as changed
        notifyObservers();  // Notifies all observers
    }

    // Getters for the measurements
    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}
```

2. WeatherDisplay (Observer)

Now, let's create an observer class `WeatherDisplay` that will observe the **WeatherStation** object for changes in temperature, humidity, and pressure.

```
java
Copy code
import java.util.Observer;
import java.util.Observable;

public class WeatherDisplay implements Observer {
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherDisplay(Observable weatherStation) {
        weatherStation.addObserver(this);  // Register this object as an
observer
    }

    // This method will be called whenever the observable object notifies its
observers
    @Override
    public void update(Observable observable, Object arg) {
        if (observable instanceof WeatherStation) {
            WeatherStation weatherStation = (WeatherStation) observable;
            this.temperature = weatherStation.getTemperature();
            this.humidity = weatherStation.getHumidity();
            this.pressure = weatherStation.getPressure();
            display();  // Display updated values
        }
    }

    // Display the weather information
    public void display() {
        System.out.println("Weather Update: ");
        System.out.println("Temperature: " + temperature + "°C");
        System.out.println("Humidity: " + humidity + "%");
        System.out.println("Pressure: " + pressure + " hPa");
        System.out.println("----------------------------");
    }
}
```

3. Main Program (Test the WeatherStation and WeatherDisplay)

```
java
Copy code
public class WeatherStationApp {
    public static void main(String[] args) {
        // Create a WeatherStation object
        WeatherStation weatherStation = new WeatherStation();

        // Create a WeatherDisplay object (Observer)
        WeatherDisplay weatherDisplay = new WeatherDisplay(weatherStation);

        // Set measurements and notify the observers
        System.out.println("Setting measurements to: ");
```

```
        weatherStation.setMeasurements(30.5f, 65.0f, 1013.0f);

        // Set new measurements and notify again
        System.out.println("Setting new measurements to: ");
        weatherStation.setMeasurements(25.0f, 70.0f, 1010.0f);
    }
}
```

Explanation:

1. **WeatherStation Class**:
   - Inherits from **`Observable`**.
   - Contains private fields for **temperature**, **humidity**, and **pressure**.
   - Provides the `setMeasurements()` method to set values for these fields and then call `measurementsChanged()` to notify observers.
   - The `measurementsChanged()` method marks the object as changed and notifies all observers by calling `notifyObservers()`.
2. **WeatherDisplay Class**:
   - Implements the **`Observer`** interface.
   - When the observable (WeatherStation) calls `notifyObservers()`, the `update()` method is triggered in this class.
   - The `update()` method retrieves the new measurements from the **WeatherStation** object and calls `display()` to show the updated values.
3. **Main Program (WeatherStationApp)**:
   - Creates a **WeatherStation** object and a **WeatherDisplay** object.
   - Sets measurements on the **WeatherStation**, which triggers the `update()` method in the **WeatherDisplay**, displaying the updated weather information.

Sample Output:

```
yaml
Copy code
Setting measurements to:
Weather Update:
Temperature: 30.5°C
Humidity: 65.0%
Pressure: 1013.0 hPa
--------------------------
Setting new measurements to:
Weather Update:
Temperature: 25.0°C
Humidity: 70.0%
Pressure: 1010.0 hPa
--------------------------
```

Q.2 Write a program for LDR to vary the light intensity of LED using Arduino.

```
// Pin Definitions
int ldrPin = A0;     // LDR connected to Analog pin A0
int ledPin = 9;      // LED connected to PWM pin 9

void setup() {
  pinMode(ledPin, OUTPUT);  // Set the LED pin as an output
  Serial.begin(9600);  // Start serial communication to monitor LDR values
}

void loop() {
  // Read the value from the LDR (light intensity)
  int ldrValue = analogRead(ldrPin);

  // Map the LDR value (0 to 1023) to PWM range (0 to 255)
  int brightness = map(ldrValue, 0, 1023, 0, 255);

  // Set the brightness of the LED based on the LDR value
  analogWrite(ledPin, brightness);

  // Print the LDR value and brightness to the Serial Monitor
  Serial.print("LDR Value: ");
  Serial.print(ldrValue);
  Serial.print(" -> LED Brightness: ");
  Serial.println(brightness);

  // Add a small delay to avoid excessive serial printing
  delay(100);
}
```

Slip 19 :

Q.1 Write a Java Program to implement Factory method for Pizza Store with createPizza(), orederPizza(), prepare(), Bake(), cut(), box(). Use this to create variety of pizza's like NyStyleCheesePizza, ChicagoStyleCheesePizza etc

Java Program to Implement the Factory Method Pattern:

1. Pizza Class (Abstract Base Class)

This class defines the methods that all pizzas will share.

```
java
Copy code
```

```
public abstract class Pizza {
    protected String name;

    // Common pizza methods that all pizza types will use
    public void prepare() {
        System.out.println("Preparing " + name);
    }

    public void bake() {
        System.out.println("Baking " + name);
    }

    public void cut() {
        System.out.println("Cutting " + name);
    }

    public void box() {
        System.out.println("Boxing " + name);
    }

    // Get pizza name
    public String getName() {
        return name;
    }
}
```

2. Concrete Pizza Classes

Each of these classes represents a different style of pizza.

NYStyleCheesePizza Class:

```
java
Copy code
public class NYStyleCheesePizza extends Pizza {
    public NYStyleCheesePizza() {
        name = "New York Style Cheese Pizza";
    }
}
```

ChicagoStyleCheesePizza Class:

```
java
Copy code
public class ChicagoStyleCheesePizza extends Pizza {
    public ChicagoStyleCheesePizza() {
        name = "Chicago Style Cheese Pizza";
    }
}
```

3. PizzaStore Class (Factory Method)

The `PizzaStore` class will define the abstract method `createPizza()` that must be implemented by subclasses to return the specific type of pizza.

```
java
Copy code
public abstract class PizzaStore {

    // Factory Method
    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);  // Create pizza based on the type

        pizza.prepare();  // Prepare the pizza
        pizza.bake();     // Bake the pizza
        pizza.cut();      // Cut the pizza
        pizza.box();      // Box the pizza

        return pizza;
    }

    // Abstract Factory Method, to be implemented by concrete stores
    protected abstract Pizza createPizza(String type);
}
```

4. Concrete Pizza Stores

Each store subclass will implement the `createPizza()` method, which will return the appropriate pizza based on the input type.

NYPizzaStore Class:

```
java
Copy code
public class NYPizzaStore extends PizzaStore {

    @Override
    protected Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new NYStyleCheesePizza();
        }
        // Add more types here if necessary, like "veggie", "clam", etc.

        return pizza;
    }
}
```

ChicagoPizzaStore Class:

```
java
Copy code
public class ChicagoPizzaStore extends PizzaStore {

    @Override
    protected Pizza createPizza(String type) {
        Pizza pizza = null;
```

```
        if (type.equals("cheese")) {
            pizza = new ChicagoStyleCheesePizza();
        }
        // Add more types here if necessary, like "veggie", "clam", etc.

        return pizza;
    }
}
```

5. Main Class to Test the Program

In the `PizzaTestDrive` class, we will create instances of different pizza stores and order pizzas.

```
java
Copy code
public class PizzaTestDrive {

    public static void main(String[] args) {
        // Create a New York Pizza Store
        PizzaStore nyStore = new NYPizzaStore();

        // Order a Cheese Pizza from NY Pizza Store
        Pizza pizza = nyStore.orderPizza("cheese");
        System.out.println("Ethan ordered a " + pizza.getName() + "\n");

        // Create a Chicago Pizza Store
        PizzaStore chicagoStore = new ChicagoPizzaStore();

        // Order a Cheese Pizza from Chicago Pizza Store
        pizza = chicagoStore.orderPizza("cheese");
        System.out.println("Joel ordered a " + pizza.getName() + "\n");
    }
}
```

Output:

```
mathematica
Copy code
Preparing New York Style Cheese Pizza
Baking New York Style Cheese Pizza
Cutting New York Style Cheese Pizza
Boxing New York Style Cheese Pizza
Ethan ordered a New York Style Cheese Pizza

Preparing Chicago Style Cheese Pizza
Baking Chicago Style Cheese Pizza
Cutting Chicago Style Cheese Pizza
Boxing Chicago Style Cheese Pizza
Joel ordered a Chicago Style Cheese Pizza
```

Explanation:

1. **Pizza Class**:

    - The `Pizza` class is an abstract base class that defines the common behavior (prepare, bake, cut, and box) for all types of pizzas.
    - The `name` field is set in each subclass to specify the type of pizza.
2. **Concrete Pizza Classes (NYStyleCheesePizza, ChicagoStyleCheesePizza)**:
    - These classes are specific implementations of pizza styles. They set the `name` of the pizza to reflect the style.
3. **PizzaStore Class**:
    - The `PizzaStore` class is the core of the Factory Method.
      The `orderPizza()` method calls `createPizza()` to get the pizza and then prepares, bakes, cuts, and boxes it.
    - `createPizza()` is an abstract method, which is implemented by the concrete pizza store classes (`NYPizzaStore`, `ChicagoPizzaStore`).
4. **Concrete Pizza Stores (NYPizzaStore, ChicagoPizzaStore)**:
    - Each store implements `createPizza()` to return the correct type of pizza based on the order.
    - For example, `NYPizzaStore` will return a `NYStyleCheesePizza` when `"cheese"` is ordered.
5. **PizzaTestDrive Class**:
    - The main class simulates the pizza ordering process from different pizza stores. It demonstrates the **Factory Method Pattern** by showing how the creation of different pizza types is abstracted through the `PizzaStore`'s `createPizza()` method.

Q.2 Start Raspberry Pi and Execute various Linux commands in command terminal window: ls, cd, touch, mv, rm, man, mkdir, rmdir, tar, gzip, cat, more, less, ps, sudo, cron, chown, chgrp, pingetc.

Step 1: Start Raspberry Pi

1. **Power on the Raspberry Pi**.
2. Make sure it is connected to a **monitor**, **keyboard**, and **mouse**. Alternatively, you can connect to the Raspberry Pi using **SSH** if it is connected to your local network.

Step 2: Open the Command Terminal

Once the Raspberry Pi boots up:

- If you're using a **GUI** interface, open the **Terminal** by clicking on the Terminal icon or pressing `Ctrl + Alt + T`.
- If you're using SSH, open your SSH client (e.g., **PuTTY**) and connect to your Raspberry Pi by using its IP address and login credentials.

Step 3: Execute the Various Linux Commands

Here is a list of common **Linux commands** and their usage. Execute these commands one by one in the terminal window.

1. `ls` – List directory contents

```
bash
Copy code
ls
```

This will list all the files and directories in the current working directory.

2. `cd` – Change directory

```
bash
Copy code
cd /path/to/directory
```

Use this command to navigate to a different directory. Replace `/path/to/directory` with the actual path.

3. `touch` – Create an empty file

```
bash
Copy code
touch myfile.txt
```

This will create a new empty file called `myfile.txt` in the current directory.

4. `mv` – Move or rename files

```
bash
Copy code
mv myfile.txt /path/to/new/directory
```

This will move the file `myfile.txt` to the new directory. You can also use this command to rename files:

```
bash
Copy code
mv oldname.txt newname.txt
```

5. `rm` – Remove files or directories

```
bash
Copy code
rm myfile.txt
```

This will delete the file `myfile.txt`.

To delete a directory (and its contents):

```
bash
Copy code
rm -r mydirectory
```

6. `man` – View manual for commands

```
bash
Copy code
man ls
```

This will display the manual page for the `ls` command. You can exit the manual by pressing `q`.

7. `mkdir` – Create a directory

```
bash
Copy code
mkdir mydirectory
```

This will create a new directory called `mydirectory`.

8. `rmdir` – Remove an empty directory

```
bash
Copy code
rmdir mydirectory
```

This will remove an empty directory called `mydirectory`.

9. `tar` – Archive files

```
bash
Copy code
tar -cvf archive.tar directory
```

This will create an archive called `archive.tar` of the specified directory. The options used:

- `-c`: Create a new archive.
- `-v`: Verbose mode (shows files being archived).
- `-f`: Specifies the archive file name.

To extract the archive:

```
bash
Copy code
tar -xvf archive.tar
```

10. `gzip` – Compress files

```
bash
Copy code
gzip myfile.txt
```

This will compress the file `myfile.txt` into `myfile.txt.gz`.

To decompress a `.gz` file:

```
bash
Copy code
gzip -d myfile.txt.gz
```

11. `cat` – Concatenate and display file content

```
bash
Copy code
cat myfile.txt
```

This will display the contents of `myfile.txt` on the terminal.

12. `more` – View file contents page by page

```
bash
Copy code
more myfile.txt
```

This will allow you to view the contents of `myfile.txt` one page at a time.

13. `less` – View file contents interactively

```
bash
Copy code
less myfile.txt
```

This is similar to `more`, but allows you to scroll up and down interactively.

14. `ps` – Display running processes

```
bash
Copy code
ps aux
```

This will show a list of all running processes. `ps aux` displays all processes, including those from other users.

15. `sudo` – Execute commands with root privileges

```
bash
Copy code
sudo apt update
```

This command updates the package list on your Raspberry Pi. `sudo` is used to execute commands with elevated privileges.

16. `cron` – Schedule tasks to run at specified times

```
bash
Copy code
crontab -e
```

This will open the cron jobs editor. You can use it to schedule tasks to run at specific times. A cron job entry might look like this:

```
bash
Copy code
0 5 * * * /path/to/script.sh
```

This would run `/path/to/script.sh` every day at 5:00 AM.

17. `chown` – Change file owner and group

```
bash
Copy code
sudo chown pi:pi myfile.txt
```

This will change the owner and group of `myfile.txt` to `pi`.

18. `chgrp` – Change group ownership

```
bash
Copy code
sudo chgrp pi myfile.txt
```

This will change the group ownership of `myfile.txt` to `pi`.

19. `ping` – Send ICMP Echo Request

```
bash
Copy code
ping google.com
```

This will send a ping request to `google.com` to check the network connectivity. To stop the ping process, press `Ctrl+C`.

20. `etc` – Directory for configuration files

In Linux, `/etc` is a directory that holds configuration files for the system. You can list files in `/etc` with:

```
bash
Copy code
ls /etc
```

Slip 20 :

Q.1 Write a Java Program to implement I/O Decorator for converting uppercase letters to lower case letters

Steps:

1. **InputStream**: We will use `InputStream` or `Reader` as the base interface for reading data.
2. **Decorator**: We will create a decorator class that extends `InputStreamReader` or `BufferedReader`, which will modify the input data (convert it to lowercase) while reading.
3. **Concrete Component**: The concrete class will read input normally.
4. **Client**: The client will use the decorator to read the input with modified behavior (in this case, converting all uppercase letters to lowercase).

Java Program:

1. I/O Decorator Pattern for Lowercase Conversion

```
java
Copy code
import java.io.*;

public class LowerCaseInputStream extends FilterInputStream {

    // Constructor that passes the input stream to the super class
    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    // Override the read() method to convert characters to lowercase
    @Override
    public int read() throws IOException {
```

```
        int character = super.read();  // Read a byte
        if (character == -1) {
            return -1;  // End of stream
        }
        return Character.toLowerCase((char) character);  // Convert to
lowercase
    }

    // Override the read() method that reads a portion of the input stream
    @Override
    public int read(byte[] b, int off, int len) throws IOException {
        int bytesRead = super.read(b, off, len);  // Read the bytes into the
array
        if (bytesRead == -1) {
            return -1;  // End of stream
        }

        // Convert all the characters in the byte array to lowercase
        for (int i = 0; i < bytesRead; i++) {
            b[off + i] = (byte) Character.toLowerCase((char) b[off + i]);
        }
        return bytesRead;
    }
}
```

2. Test Class for Using the Lowercase Decorator

```
java
Copy code
import java.io.*;

public class IOTest {
    public static void main(String[] args) {
        String input = "Hello World! This is a TEST message.";

        // Creating a ByteArrayInputStream to simulate user input (as if
typed in the console)
        ByteArrayInputStream byteArrayInputStream = new
ByteArrayInputStream(input.getBytes());

        // Wrap the byte array input stream with our custom
LowerCaseInputStream
        LowerCaseInputStream lowerCaseInputStream = new
LowerCaseInputStream(byteArrayInputStream);

        // Read the data from the lower case input stream
        try {
            int character;
            while ((character = lowerCaseInputStream.read()) != -1) {
                System.out.print((char) character);  // Print the character
in lowercase
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
}
```

Explanation:

1. **`LowerCaseInputStream` Class**:
   - It extends `FilterInputStream`, which is a subclass of `InputStream`. The `FilterInputStream` class is used to decorate an existing input stream.
   - The `read()` method is overridden to convert the character read to lowercase using `Character.toLowerCase()` before returning it.
   - The second `read(byte[] b, int off, int len)` method is also overridden to convert the entire byte array of characters to lowercase.
2. **`IOTest` Class**:
   - A test case where we simulate user input by creating a `ByteArrayInputStream` from a string (`input`).
   - The `LowerCaseInputStream` decorator wraps the `ByteArrayInputStream` and converts all input characters to lowercase.
   - We then read and print the data from the stream. As expected, all characters are converted to lowercase.

Output:

```
kotlin
Copy code
hello world! this is a test message.
```

```
Q.2 Write python programs on Pi like:
a) Read your name and print Hello message with name
b) Read two numbers and print their sum, difference, product and division.
c) Word and character count of a given string.
d) Area of a given shape (rectangle, triangle and circle) reading shape and
appropriate
values from standard input.
```

a) Read your name and print Hello message with name

```
python
Copy code
# Program to read your name and print a hello message
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

- This program prompts the user to input their name and then prints a greeting message using the input name.

b) Read two numbers and print their sum, difference, product, and division

```
python
Copy code
# Program to read two numbers and print their sum, difference, product, and
division
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

sum_result = num1 + num2
difference_result = num1 - num2
product_result = num1 * num2

# Checking if division by zero occurs
if num2 != 0:
    division_result = num1 / num2
else:
    division_result = "Undefined (division by zero)"

print(f"Sum: {sum_result}")
print(f"Difference: {difference_result}")
print(f"Product: {product_result}")
print(f"Division: {division_result}")
```

- This program takes two numbers as input and prints their sum, difference, product, and division. It also checks for division by zero.

c) Word and character count of a given string

```
python
Copy code
# Program to count the words and characters in a given string
input_string = input("Enter a string: ")

# Count characters (excluding spaces)
char_count = len(input_string.replace(" ", ""))

# Count words (splitting by spaces)
word_count = len(input_string.split())

print(f"Word count: {word_count}")
print(f"Character count (excluding spaces): {char_count}")
```

- This program counts the number of words and characters (excluding spaces) in the given string. The string is split by spaces to count the words, and spaces are removed to count the characters.

d) Area of a given shape (rectangle, triangle, and circle) reading shape and appropriate values from standard input

```
python
```

```
Copy code
# Program to calculate the area of a given shape
shape = input("Enter the shape (rectangle, triangle, or circle): ").lower()

if shape == "rectangle":
    length = float(input("Enter the length: "))
    width = float(input("Enter the width: "))
    area = length * width
    print(f"Area of the rectangle: {area}")

elif shape == "triangle":
    base = float(input("Enter the base: "))
    height = float(input("Enter the height: "))
    area = 0.5 * base * height
    print(f"Area of the triangle: {area}")

elif shape == "circle":
    radius = float(input("Enter the radius: "))
    area = 3.14159 * radius * radius
    print(f"Area of the circle: {area}")

else:
    print("Invalid shape entered!")
```

- This program prompts the user to enter a shape (rectangle, triangle, or circle) and then asks for the appropriate dimensions (length, width, base, height, or radius). Based on the input shape, it calculates and prints the area.

How to run these programs on your Raspberry Pi:

1. Open the terminal on your Raspberry Pi or connect to it via SSH.
2. Create a new Python file, e.g., program.py:

   ```
   bash
   Copy code
   nano program.py
   ```

3. Copy and paste any of the above programs into the file.
4. Save the file (Ctrl + O), then exit the editor (Ctrl + X).
5. Run the program:

   ```
   bash
   Copy code
   python3 program.py
   ```

END