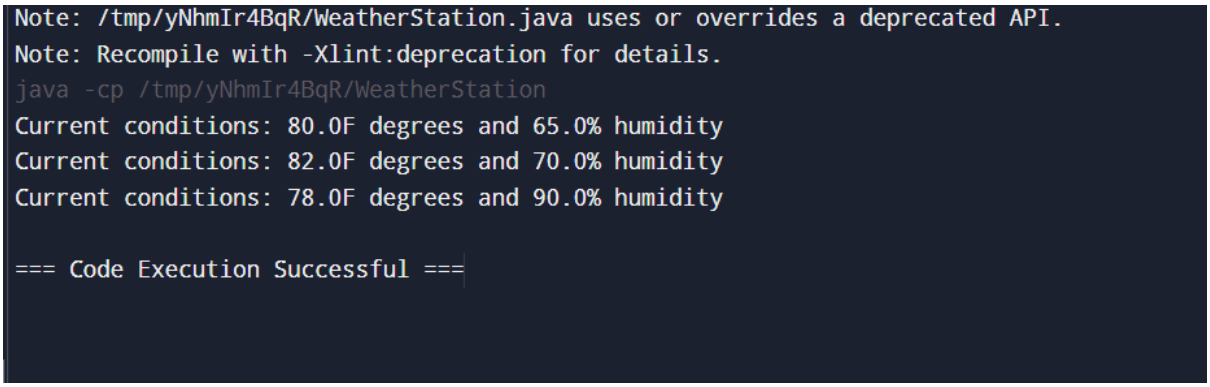
1. **Write a JAVA Program to implement built-in support (java.util.Observable) Weather station with members temperature, humidity, pressure and methods - mesurmentsChanged(), setMesurment(), getTemperature(), getHumidity(), getPressure()**

```
import java.util.Observable;
import java.util.Observer;
// WeatherData class extending Observable
class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;
    // Method to set new measurements
    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
    // Notify observers that measurements have changed
    private void measurementsChanged() {
        setChanged();  // Mark this Observable as changed
        notifyObservers();  // Notify all registered observers
    }
    // Getter methods
    public float getTemperature() {
        return temperature;
    }
    public float getHumidity() {
        return humidity;
    }
    public float getPressure() {
        return pressure;
    }
}
// DisplayElement interface
interface DisplayElement {
    void display();
}

// CurrentConditionsDisplay class implementing Observer and DisplayElement
class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Observable observable;
    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }
    // Update method called by Observable
    @Override
    public void update(Observable o, Object arg) {
        if (o instanceof WeatherData) {
            WeatherData weatherData = (WeatherData) o;
            this.temperature = weatherData.getTemperature();
```

```
            this.humidity = weatherData.getHumidity();
            display();  // Call display method to show updated conditions
        }
    }
    // Display the current conditions
    @Override
    public void display() {
        System.out.println("Current conditions: " + temperature + "F degrees and " + humidity +
"% humidity");
    }
}
// Main class to run the program
public class WeatherStation {
    public static void main(String[] args) {
        // Create WeatherData instance
        WeatherData weatherData = new WeatherData();

        // Create CurrentConditionsDisplay and pass WeatherData instance
        CurrentConditionsDisplay currentDisplay = new
CurrentConditionsDisplay(weatherData);

        // Set new measurements
        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

Output:-



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

```
import java.io.BufferedReader;
import java.io.FilterReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Reader;
class LowerCaseReader extends FilterReader {
    protected LowerCaseReader(Reader in) {
        super(in);
    }
    @Override
    public int read() throws IOException {
        int c = super.read();
        if (c == -1) {
            return c;  // End of stream
        }
        return Character.toLowerCase(c);  // Convert uppercase to lowercase
    }
    @Override
    public int read(char[] cbuf, int off, int len) throws IOException {
        int numChars = super.read(cbuf, off, len);
        if (numChars == -1) {
            return numChars;  // End of stream
        }
        for (int i = off; i < off + numChars; i++) {
            cbuf[i] = Character.toLowerCase(cbuf[i]);
        }
        return numChars;
    }
}
public class LowerCaseDecoratorExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(
```

```
            new LowerCaseReader(new InputStreamReader(System.in))) {
        System.out.println("Enter text (press Enter to convert to lowercase):");
        String line = br.readLine();
        while (line != null && !line.isEmpty()) {
            System.out.println("Converted to lowercase: " + line);
            line = br.readLine();  // Read next line
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

Output :-

```
java -cp /tmp/3mbw9cxXIt/LowerCaseDecoratorExample
Enter text (press Enter to convert to lowercase):
Hritik
Converted to lowercase: hritik
hritik
Converted to lowercase: hritik
```

3. Write a Java Program to implement Factory method for Pizza Store with createPizza(), orderPizza(),prepare(), bake(), cut(), box(). Use this to create variety of pizza's like NyStyleCheesePizza, ChicagoStyleCheesePizzaetc.

```
abstract class Pizza {
  String name;
  void prepare() {
    System.out.println("Preparing " + name);
  }
  void bake() {
    System.out.println("Baking " + name);
  }
  void cut() {
    System.out.println("Cutting " + name);
  }
  void box() {
```

```
        System.out.println("Boxing " + name);
    }
    String getName() {
        return name;
    }
}
class NYStyleCheesePizza extends Pizza {
    public NYStyleCheesePizza() {
        name = "NY Style Cheese Pizza";
    }
}
class ChicagoStyleCheesePizza extends Pizza {
    public ChicagoStyleCheesePizza() {
        name = "Chicago Style Cheese Pizza";
    }
    @Override
    void cut() {
        System.out.println("Cutting the pizza into square slices (Chicago Style)");
    }
}
abstract class PizzaStore {
    // Factory Method
    public abstract Pizza createPizza(String type);
    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        System.out.println("--- Making a " + pizza.getName() + " ---");
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
class NYPizzaStore extends PizzaStore {
```

```
    @Override
    public Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new NYStyleCheesePizza();
        }
        return null;
    }
}
class ChicagoPizzaStore extends PizzaStore {
    @Override
    public Pizza createPizza(String type) {
        if (type.equals("cheese")) {
            return new ChicagoStyleCheesePizza();
        }
        // You can add more pizza types here for Chicago Style
        return null;
    }
}
public class PizzaFactoryMethodExample {
    public static void main(String[] args) {
        // Create NY pizza store
        PizzaStore nyStore = new NYPizzaStore();
        Pizza pizza1 = nyStore.orderPizza("cheese");
        System.out.println("Ordered a " + pizza1.getName() + "\n");
        PizzaStore chicagoStore = new ChicagoPizzaStore();
        Pizza pizza2 = chicagoStore.orderPizza("cheese");
        System.out.println("Ordered a " + pizza2.getName() + "\n");
    }
}
```

Output :-

```
java -cp /tmp/FmpzV76PPg/PizzaFactoryMethodExample
--- Making a NY Style Cheese Pizza ---
Preparing NY Style Cheese Pizza
Baking NY Style Cheese Pizza
Cutting NY Style Cheese Pizza
Boxing NY Style Cheese Pizza
Ordered a NY Style Cheese Pizza

--- Making a Chicago Style Cheese Pizza ---
Preparing Chicago Style Cheese Pizza
Baking Chicago Style Cheese Pizza
Cutting the pizza into square slices (Chicago Style)
Boxing Chicago Style Cheese Pizza
Ordered a Chicago Style Cheese Pizza


=== Code Execution Successful ===
```

4. Write a Java Program to implement Singleton pattern for multithreading

```
class Singleton {
    // Volatile ensures visibility of changes to variables across threads
    private static volatile Singleton instance = null;
    // Private constructor to prevent instantiation
    private Singleton() {
        System.out.println("Singleton instance created.");
    }
    // Double-checked locking to ensure thread safety and avoid unnecessary synchronization
    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();  // Only one thread will create the instance
                }
            }
        }
        return instance;
    }
```

```
}
public class SingletonMultithreadedTest {
    public static void main(String[] args) {
        // Create multiple threads trying to get the Singleton instance
        Thread t1 = new Thread(() -> {
            Singleton singleton1 = Singleton.getInstance();
            System.out.println("Thread 1: Singleton instance hashcode: " + singleton1.hashCode());
        });
        Thread t2 = new Thread(() -> {
            Singleton singleton2 = Singleton.getInstance();
            System.out.println("Thread 2: Singleton instance hashcode: " + singleton2.hashCode());
        });
        Thread t3 = new Thread(() -> {
            Singleton singleton3 = Singleton.getInstance();
            System.out.println("Thread 3: Singleton instance hashcode: " + singleton3.hashCode());
        });
        // Start the threads
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output :-

```
java -cp /tmp/CjwtTY2Fux/SingletonMultithreadedTest
Singleton instance created.
Thread 3: Singleton instance hashcode: 1288577530
Thread 1: Singleton instance hashcode: 1288577530
Thread 2: Singleton instance hashcode: 1288577530

=== Code Execution Successful ===
```

5. Write a Java Program to implement command pattern to test Remote Control Book

```
// Command interface
interface Command {
    void execute();
}
// Receiver class
class Light {
    public void turnOn() {
        System.out.println("The light is ON");
    }
    public void turnOff() {
        System.out.println("The light is OFF");
    }
}
// Receiver class
class Fan {
    public void start() {
        System.out.println("The fan is ON");
    }
    public void stop() {
        System.out.println("The fan is OFF");
    }
}
// Concrete command classes
class LightOnCommand implements Command {
    private Light light;
    public LightOnCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
```

```
        light.turnOn();
    }
}
class LightOffCommand implements Command {
    private Light light;
    public LightOffCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
        light.turnOff();
    }
}

class FanStartCommand implements Command {
    private Fan fan;
    public FanStartCommand(Fan fan) {
        this.fan = fan;
    }
    @Override
    public void execute() {
        fan.start();
    }
}
class FanStopCommand implements Command {
    private Fan fan;
    public FanStopCommand(Fan fan) {
        this.fan = fan;
    }
    @Override
    public void execute() {
        fan.stop();
```

```
    }
}
// Invoker class
class RemoteControl {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void pressButton() {
        command.execute();
    }
}
// Main class
public class RemoteControlBook {
    public static void main(String[] args) {
        Light livingRoomLight = new Light();
        Fan ceilingFan = new Fan();
        Command lightOn = new LightOnCommand(livingRoomLight);
        Command lightOff = new LightOffCommand(livingRoomLight);
        Command fanStart = new FanStartCommand(ceilingFan);
        Command fanStop = new FanStopCommand(ceilingFan);
        RemoteControl remote = new RemoteControl();
        // Testing Light Commands
        remote.setCommand(lightOn);
        remote.pressButton();  // The light is ON
        remote.setCommand(lightOff);
        remote.pressButton();  // The light is OFF
        // Testing Fan Commands
        remote.setCommand(fanStart);
        remote.pressButton();  // The fan is ON
        remote.setCommand(fanStop);
        remote.pressButton();  // The fan is OFF
```

```
    }
}
// Command interface
interface Command {
    void execute();
}
// Receiver class
class Light {
    public void turnOn() {
        System.out.println("The light is ON");
    }
    public void turnOff() {
        System.out.println("The light is OFF");
    }
}

// Receiver class
class Fan {
    public void start() {
        System.out.println("The fan is ON");
    }
    public void stop() {
        System.out.println("The fan is OFF");
    }
}
// Concrete command classes
class LightOnCommand implements Command {
    private Light light;
    public LightOnCommand(Light light) {
        this.light = light;
    }
    @Override
```

```
    public void execute() {
        light.turnOn();
    }
}
class LightOffCommand implements Command {
    private Light light;
    public LightOffCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
        light.turnOff();
    }
}
class FanStartCommand implements Command {
    private Fan fan;
    public FanStartCommand(Fan fan) {
        this.fan = fan;
    }
    @Override
    public void execute() {
        fan.start();
    }
}
class FanStopCommand implements Command {
    private Fan fan;
    public FanStopCommand(Fan fan) {
        this.fan = fan;
    }
    @Override
    public void execute() {
        fan.stop();
```

```
    }
}
// Invoker class
class RemoteControl {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void pressButton() {
        command.execute();
    }
}
// Main class
public class RemoteControlBook {
    public static void main(String[] args) {
        Light livingRoomLight = new Light();
        Fan ceilingFan = new Fan();
        Command lightOn = new LightOnCommand(livingRoomLight);
        Command lightOff = new LightOffCommand(livingRoomLight);
        Command fanStart = new FanStartCommand(ceilingFan);
        Command fanStop = new FanStopCommand(ceilingFan);
        RemoteControl remote = new RemoteControl();
        // Testing Light Commands
        remote.setCommand(lightOn);
        remote.pressButton();  // The light is ON
        remote.setCommand(lightOff);
        remote.pressButton();  // The light is OFF
        // Testing Fan Commands
        remote.setCommand(fanStart);
        remote.pressButton();  // The fan is ON
        remote.setCommand(fanStop);
        remote.pressButton();  // The fan is OFF
```

```
    }
}
// Command interface
interface Command {
    void execute();
}
// Receiver class
class Light {
    public void turnOn() {
        System.out.println("The light is ON");
    }
    public void turnOff() {
        System.out.println("The light is OFF");
    }
}
// Receiver class
class Fan {
    public void start() {
        System.out.println("The fan is ON");
    }
    public void stop() {
        System.out.println("The fan is OFF");
    }
}
// Concrete command classes
class LightOnCommand implements Command {
    private Light light;
    public LightOnCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
```

```
        light.turnOn();
    }
}
class LightOffCommand implements Command {
    private Light light;
    public LightOffCommand(Light light) {
        this.light = light;
    }
    @Override
    public void execute() {
        light.turnOff();
    }
}
class FanStartCommand implements Command {
    private Fan fan;
    public FanStartCommand(Fan fan) {
        this.fan = fan;
    }
    @Override
    public void execute() {
        fan.start();
    }
}
class FanStopCommand implements Command {
    private Fan fan;
    public FanStopCommand(Fan fan) {
        this.fan = fan;
    }
    @Override
    public void execute() {
        fan.stop();
    }
```

```
}
// Invoker class
class RemoteControl {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void pressButton() {
        command.execute();
    }
}
// Main class
public class RemoteControlBook {
    public static void main(String[] args) {
        Light livingRoomLight = new Light();
        Fan ceilingFan = new Fan();
        Command lightOn = new LightOnCommand(livingRoomLight);
        Command lightOff = new LightOffCommand(livingRoomLight);
        Command fanStart = new FanStartCommand(ceilingFan);
        Command fanStop = new FanStopCommand(ceilingFan);
        RemoteControl remote = new RemoteControl();
        // Testing Light Commands
        remote.setCommand(lightOn);
        remote.pressButton();  // The light is ON
        remote.setCommand(lightOff);
        remote.pressButton();  // The light is OFF
        // Testing Fan Commands
        remote.setCommand(fanStart);
        remote.pressButton();  // The fan is ON
        remote.setCommand(fanStop);
        remote.pressButton();  // The fan is OFF
    }
```

}

Output :-

```
java -cp /tmp/D5EVzwDuZO/RemoteControlBook
The light is ON
The light is OFF
The fan is ON
The fan is OFF

=== Code Execution Successful ===
```

6. Write a Java Program to implement undo command to test Ceilingfan.

```
// CeilingFan class (Receiver)
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;  // Initially, the fan is 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;
    }
}
// Command interface
interface Command {
    void execute();
    void undo();
}
// Command to set the ceiling fan to HIGH speed
class CeilingFanHighCommand implements Command {
    private CeilingFan ceilingFan;
    private int prevSpeed;
    public CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    public void execute() {
        prevSpeed = ceilingFan.getSpeed();  // Save previous speed for undo
        ceilingFan.high();
    }
    public void undo() {
        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 the ceiling fan to MEDIUM speed
class CeilingFanMediumCommand implements Command {
    private CeilingFan ceilingFan;
    private int prevSpeed;
    public CeilingFanMediumCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }
    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.medium();
    }

    public void undo() {
        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 the ceiling fan to LOW speed
class CeilingFanLowCommand implements Command {
    private CeilingFan ceilingFan;
    private int prevSpeed;
    public CeilingFanLowCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }
    public void execute() {
```

```
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.low();
    }
    public void undo() {
        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;
    }
    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.off();
    }
    public void undo() {
        if (prevSpeed == CeilingFan.HIGH) {
            ceilingFan.high();
        } else if (prevSpeed == CeilingFan.MEDIUM) {
            ceilingFan.medium();
        } else if (prevSpeed == CeilingFan.LOW) {
            ceilingFan.low();
        } else {
```

```
            ceilingFan.off();
        }
    }
}
// RemoteControl class - Invoker
class RemoteControl {
    private Command[] onCommands;
    private Command lastCommand;
    public RemoteControl() {
        onCommands = new Command[4];  // To handle fan speeds (OFF, LOW, MEDIUM, HIGH)
    }
    public void setCommand(int slot, Command command) {
        onCommands[slot] = command;
    }
    public void pressButton(int slot) {
        onCommands[slot].execute();
        lastCommand = onCommands[slot];
    }
    public void pressUndoButton() {
        if (lastCommand != null) {
            lastCommand.undo();
        }
    }
}
public class CeilingFanTest {
    public static void main(String[] args) {
        RemoteControl remote = new RemoteControl();
        // Create the ceiling fan (receiver)
        CeilingFan ceilingFan = new CeilingFan();
        // Create commands for ceiling fan speeds
        CeilingFanHighCommand ceilingFanHigh = new CeilingFanHighCommand(ceilingFan);
        CeilingFanMediumCommand ceilingFanMedium = new
CeilingFanMediumCommand(ceilingFan);
        CeilingFanLowCommand ceilingFanLow = new CeilingFanLowCommand(ceilingFan);
```

```
        CeilingFanOffCommand ceilingFanOff = new CeilingFanOffCommand(ceilingFan);
        // Set commands in the remote control
        remote.setCommand(0, ceilingFanHigh);  // Slot 0 for HIGH
        remote.setCommand(1, ceilingFanMedium);  // Slot 1 for MEDIUM
        remote.setCommand(2, ceilingFanLow);  // Slot 2 for LOW
        remote.setCommand(3, ceilingFanOff);  // Slot 3 for OFF
        // Test the ceiling fan commands
        System.out.println("--- Testing Ceiling Fan ---");
        remote.pressButton(2);  // Set to LOW
        remote.pressButton(1);  // Set to MEDIUM
        remote.pressUndoButton();  // Undo (should go back to LOW)
        remote.pressButton(0);  // Set to HIGH
        remote.pressUndoButton();  // Undo (should go back to MEDIUM)
        remote.pressButton(3);  // Turn off
        remote.pressUndoButton();  // Undo (should go back to HIGH)
    }
}
```

Output :-

```
java -cp /tmp/yOs0TStURS/CeilingFanTest
--- Testing Ceiling Fan ---
Ceiling fan is on LOW
Ceiling fan is on MEDIUM
Ceiling fan is on LOW
Ceiling fan is on HIGH
Ceiling fan is on LOW
Ceiling fan is OFF
Ceiling fan is on LOW

=== Code Execution Successful ===
```

7. Write a Java Program to implement Iterator Pattern forDesigning Menu like Breakfast, Lunch or DinnerMenu.

```
import java.util.ArrayList;
// MenuItem class
class MenuItem {
    private String name;
    private String description;
    private boolean vegetarian;
    private double price;
    public MenuItem(String name, String description, boolean vegetarian, double price) {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }
    public String getName() {
        return name;
    }
    public String getDescription() {
        return description;
    }
    public boolean isVegetarian() {
        return vegetarian;
    }
    public double getPrice() {
        return price;
    }
}
// Iterator interface for menu items
interface Iterator {
    boolean hasNext();
    MenuItem next();
}
// Menu interface to create an iterator
```

```
interface Menu {
    Iterator createIterator();
}
// Breakfast Menu class - uses an array to store menu items
class BreakfastMenu implements Menu {
    private static final int MAX_ITEMS = 6;
    private int numberOfItems = 0;
    private MenuItem[] menuItems;
    public BreakfastMenu() {
        menuItems = new MenuItem[MAX_ITEMS];
        addItem("Pancakes", "Pancakes with syrup", true, 2.99);
        addItem("Waffles", "Waffles with blueberries", true, 3.59);
    }
    public void addItem(String name, String description, boolean vegetarian, double price) {
        if (numberOfItems >= MAX_ITEMS) {
            System.out.println("Menu is full! Can't add more items.");
        } else {
            menuItems[numberOfItems] = new MenuItem(name, description, vegetarian, price);
            numberOfItems++;
        }
    }
    public Iterator createIterator() {
        return new BreakfastMenuIterator(menuItems);
    }
}
// Iterator for Breakfast Menu (using an array)
class BreakfastMenuIterator implements Iterator {
    private MenuItem[] items;
    private int position = 0;
    public BreakfastMenuIterator(MenuItem[] items) {
        this.items = items;
    }
    public boolean hasNext() {
        return position < items.length && items[position] != null;
```

```
    }

    public MenuItem next() {
        MenuItem menuItem = items[position];
        position++;
        return menuItem;
    }
}
// Lunch Menu class - uses an ArrayList to store menu items
class LunchMenu implements Menu {
    private ArrayList<MenuItem> menuItems;
    public LunchMenu() {
        menuItems = new ArrayList<>();
        addItem("Burger", "A juicy beef burger", false, 5.99);
        addItem("Salad", "A fresh garden salad", true, 3.99);
    }
    public void addItem(String name, String description, boolean vegetarian, double price) {
        menuItems.add(new MenuItem(name, description, vegetarian, price));
    }
    public Iterator createIterator() {
        return new LunchMenuIterator(menuItems);
    }
}
// Iterator for Lunch Menu (using ArrayList)
class LunchMenuIterator implements Iterator {
    private ArrayList<MenuItem> items;
    private int position = 0;

    public LunchMenuIterator(ArrayList<MenuItem> items) {
        this.items = items;
    }
    public boolean hasNext() {
        return position < items.size();
    }
```

```
    public MenuItem next() {
        MenuItem menuItem = items.get(position);
        position++;
        return menuItem;
    }
}
// Dinner Menu class - uses a fixed array to store menu items
class DinnerMenu implements Menu {
    private static final int MAX_ITEMS = 5;
    private MenuItem[] menuItems;
    private int numberOfItems = 0;
    public DinnerMenu() {
        menuItems = new MenuItem[MAX_ITEMS];
        addItem("Steak", "Grilled steak with vegetables", false, 15.99);
        addItem("Pasta", "Spaghetti with marinara sauce", true, 7.99);
    }
    public void addItem(String name, String description, boolean vegetarian, double price) {
        if (numberOfItems >= MAX_ITEMS) {
            System.out.println("Menu is full! Can't add more items.");
        } else {
            menuItems[numberOfItems] = new MenuItem(name, description, vegetarian, price);
            numberOfItems++;
        }
    }
    public Iterator createIterator() {
        return new DinnerMenuIterator(menuItems);
    }
}
// Iterator for Dinner Menu
class DinnerMenuIterator implements Iterator {
    private MenuItem[] items;
    private int position = 0;
    public DinnerMenuIterator(MenuItem[] items) {
        this.items = items;
```

```
    }
    public boolean hasNext() {
        return position < items.length && items[position] != null;
    }
    public MenuItem next() {
        MenuItem menuItem = items[position];
        position++;
        return menuItem;
    }
}
// Waiter class - Client
class Waiter {
    private Menu breakfastMenu;
    private Menu lunchMenu;
    private Menu dinnerMenu;
    public Waiter(Menu breakfastMenu, Menu lunchMenu, Menu dinnerMenu) {
        this.breakfastMenu = breakfastMenu;
        this.lunchMenu = lunchMenu;
        this.dinnerMenu = dinnerMenu;
    }
    public void printMenu() {
        Iterator breakfastIterator = breakfastMenu.createIterator();
        Iterator lunchIterator = lunchMenu.createIterator();
        Iterator dinnerIterator = dinnerMenu.createIterator();
        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(breakfastIterator);
        System.out.println("\nLUNCH");
        printMenu(lunchIterator);
        System.out.println("\nDINNER");
        printMenu(dinnerIterator);
    }
    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
```

```
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
    }
}
public class MenuTest {
    public static void main(String[] args) {
        BreakfastMenu breakfastMenu = new BreakfastMenu();
        LunchMenu lunchMenu = new LunchMenu();
        DinnerMenu dinnerMenu = new DinnerMenu();
        Waiter waiter = new Waiter(breakfastMenu, lunchMenu, dinnerMenu);
        waiter.printMenu();
    }
}
```

Output :-

```
java -cp /tmp/hUkAggxM6R/MenuTest
MENU
----
BREAKFAST
Pancakes, 2.99 -- Pancakes with syrup
Waffles, 3.59 -- Waffles with blueberries

LUNCH
Burger, 5.99 -- A juicy beef burger
Salad, 3.99 -- A fresh garden salad

DINNER
Steak, 15.99 -- Grilled steak with vegetables
Pasta, 7.99 -- Spaghetti with marinara sauce

=== Code Execution Successful ===
```