Index

Q1) 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()

```
public interface Observer
{
	public void update(float temp, float humidity, float pressure);
}

public interface DisplayElement
{
	public void display();
}
public interface Subject
 {
	public void registerObserver(Observer o);
	public void removeObserver(Observer o);
	public void notifyObservers();
}

import java.util.*;

public class WeatherData implements Subject
 {
	private ArrayList<Observer> observers;
	private float temperature;
	private float humidity;
	private float pressure;
```

```
public WeatherData()
 {
        observers = new ArrayList<>();
}

public void registerObserver(Observer o)
{
        observers.add(o);
}

public void removeObserver(Observer o)
{
        int i = observers.indexOf(o);
        if (i >= 0) {
        observers.remove(i);
}
}

public void notifyObservers()
 {
        for (int i = 0; i < observers.size(); i++) {
        Observer observer = (Observer)observers.get(i);
        observer.update(temperature, humidity, pressure);
}
}

public void measurementsChanged()
{
        notifyObservers();
}

public void setMeasurements(float temperature, float humidity, float pressure)
 {
```

```
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
}

public float getTemperature()
 {
        return temperature;
}
public float getHumidity()
 {
        return humidity;
}
public float getPressure()
 {
        return pressure;
}
}
```

public class ForecastDisplay implements Observer, DisplayElement

```
{
        private float currentPressure = 29.92f;
        private float lastPressure;
        private WeatherData weatherData;

public ForecastDisplay(WeatherData weatherData)
 {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
}
```

```
public void update(float temp, float humidity, float pressure)
{
	lastPressure = currentPressure;
	currentPressure = pressure;

	display();
}

public void display()
 {
	System.out.print("Forecast: ");
	if (currentPressure > lastPressure)
	{
		System.out.println("Improving weather on the way!");
	} else if (currentPressure == lastPressure)
	{
		System.out.println("More of the same");
	} else if (currentPressure < lastPressure)
	 {
		System.out.println("Watch out for cooler, rainy weather");
	}
}
}
```
public class HeatIndexDisplay implements Observer, DisplayElement
```
{
	float heatIndex = 0.0f;
	private WeatherData weatherData;

public HeatIndexDisplay(WeatherData weatherData)
 {
	this.weatherData = weatherData;
	weatherData.registerObserver(this);
}
```

```
public void update(float t, float rh, float pressure)
{
	heatIndex = computeHeatIndex(t, rh);
	display();
}

private float computeHeatIndex(float t, float rh)
 {
float index = (float)((16.923 + (0.185212 * t) + (5.37941 * rh) - (0.100254 * t * rh)
+ (0.00941695 * (t * t)) + (0.00728898 * (rh * rh))
+ (0.000345372 * (t * t * rh)) - (0.000814971 * (t * rh * rh)) +
(0.0000102102 * (t * t * rh * rh)) - (0.000038646 * (t * t * t)) + (0.0000291583 *
(rh * rh * rh)) + (0.00000142721 * (t * t * t * rh)) +
(0.000000197483 * (t * rh * rh * rh)) - (0.0000000218429 * (t * t * t * rh * rh)) +
0.000000000843296 * (t * t * rh * rh * rh)) -
(0.0000000000481975 * (t * t * t * rh * rh * rh)));
return index;
}

public void display()
{
	System.out.println("Heat index is " + heatIndex);
}
}
```

Step 6: Create third Observer StatisticsDisplay class.

```
public class StatisticsDisplay implements Observer, DisplayElement
 {
	private float maxTemp = 0.0f;
	private float minTemp = 200;
```

```
        private float tempSum= 0.0f;
        private int numReadings;
        private WeatherData weatherData;

public StatisticsDisplay(WeatherData weatherData)
 {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
}

public void update(float temp, float humidity, float pressure)
{
        tempSum += temp;
        numReadings++;

        if (temp > maxTemp) {
        maxTemp = temp;
}

        if (temp < minTemp) {
        minTemp = temp;
        }

        display();
        }

public void display()
 {
System.out.println("Avg/Max/Min temperature = " + (tempSum / numReadings)
+ "/" + maxTemp + "/" + minTemp);
}
}
```

```
public class CurrentConditionsDisplay implements Observer, DisplayElement
 {
        private float temperature;
        private float humidity;
        private Subject weatherData;

public CurrentConditionsDisplay(Subject weatherData)
 {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
}

public void update(float temperature, float humidity, float pressure)
{
        this.temperature = temperature;
        this.humidity = humidity;
        display();
}

public void display()
 {
        System.out.println("Current conditions: " + temperature
        + "F degrees and " + humidity + "% humidity");
}
}
```

```
public class WeatherStation
{
        public static void main(String[] args)
{
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
        new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

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

Output

```
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
```

Q2. Write a Java Program to implement I/O Decorator for converting uppercase letters to lower case letters.

```
import java.io.*;
import java.util.*;

 class LowerCaseInputStream extends FilterInputStream
 {
 public LowerCaseInputStream(InputStream in)
{
        super(in);
 }

 public int read() throws IOException
 {
 int c = super.read();
 return (c == -1 ? c : Character.toLowerCase((char)c));
 }

 public int read(byte[] b, int offset, int len) throws IOException
{
 int result = super.read(b, offset, len);

 for (int i = offset; i < offset+result; i++)
 {
        b[i] = (byte)Character.toLowerCase((char)b[i]);
 }
 return result;
 }
}
```

```
public class Main
{
 public static void main(String[] args) throws IOException
{
 int c;
 try
{
 InputStream in =  new LowerCaseInputStream(
          new BufferedInputStream( new FileInputStream("test.txt")));
 while((c = in.read()) >= 0)
{
         System.out.print((char)c);
 }
 in.close();
 } catch (IOException e)
 {
          e.printStackTrace();
 }
 }
}
```

Q3) 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.

```
import java.util.ArrayList;

abstract public class Pizza {
   String name;
   String dough;
   String sauce;
   ArrayList toppings = new ArrayList();

   public String getName() {
      return name;
   }

   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);
    }

    public String toString() {
        // code to display pizza name and ingredients
        StringBuffer display = new StringBuffer();
        display.append("---- " + name + " ----\n");
        display.append(dough + "\n");
        display.append(sauce + "\n");
        for (int i = 0; i < toppings.size(); i++) {
            display.append((String) toppings.get(i) + "\n");
        }
        return display.toString();
    }
}
```

Step 2: Create Concrete Pizza classes which extends abstract Pizza class - CheesePizza, ClamPizza, VeggiePizza, and PepperoniPizza class:

```
public class CheesePizza extends Pizza {
    public CheesePizza() {
        name = "Cheese Pizza";
```

```
        dough = "Regular Crust";
        sauce = "Marinara Pizza Sauce";
        toppings.add("Fresh Mozzarella");
        toppings.add("Parmesan");
    }
}

public class ClamPizza extends Pizza {
    public ClamPizza() {
        name = "Clam Pizza";
        dough = "Thin crust";
        sauce = "White garlic sauce";
        toppings.add("Clams");
        toppings.add("Grated parmesan cheese");
    }
}

public class VeggiePizza extends Pizza {
    public VeggiePizza() {
        name = "Veggie Pizza";
        dough = "Crust";
        sauce = "Marinara sauce";
        toppings.add("Shredded mozzarella");
        toppings.add("Grated parmesan");
        toppings.add("Diced onion");
        toppings.add("Sliced mushrooms");
```

```
        toppings.add("Sliced red pepper");
        toppings.add("Sliced black olives");
    }
}

public class PepperoniPizza extends Pizza {
    public PepperoniPizza() {
        name = "Pepperoni Pizza";
        dough = "Crust";
        sauce = "Marinara sauce";
        toppings.add("Sliced Pepperoni");
        toppings.add("Sliced Onion");
        toppings.add("Grated parmesan cheese");
    }
}
```

Step 3: Create a SimplePizzaFactory class which produces pizza object based on the type of the pizza - SimplePizzaFactory java class.

```
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {
        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
```

```
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

Step 4: Let's create PizzaStore to order the Pizza:

```
package com.ramesh.gof.factory.pizzas;

public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;
```

```
        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }
}
```

Step 5: Let's test the Factory Pattern with below PizzaTestDrive:

```
public class PizzaTestDrive {

    public static void main(String[] args) {
        SimplePizzaFactory factory = new SimplePizzaFactory();
        PizzaStore store = new PizzaStore(factory);

        Pizza pizza = store.orderPizza("cheese");
        System.out.println("We ordered a " + pizza.getName() + "\n");

        pizza = store.orderPizza("veggie");
        System.out.println("We ordered a " + pizza.getName() + "\n");
    }
```

}

Output :

Preparing Cheese Pizza

Baking Cheese Pizza

Cutting Cheese Pizza

Boxing Cheese Pizza

We ordered a Cheese Pizza

Preparing Veggie Pizza

Baking Veggie Pizza

Cutting Veggie Pizza

Boxing Veggie Pizza

We ordered a Veggie Pizza

Q4. Write a Java Program to implement Singleton pattern for multithreading

```
package in.bench.resources.singleton.design.pattern;

public class SingletonDesignPatternInMultiThreadedEnvironment
{
private static volatile
SingletonDesignPatternInMultiThreadedEnvironment INSTANCE;

private SingletonDesignPatternInMultiThreadedEnvironment()
 {}
public static SingletonDesignPatternInMultiThreadedEnvironment
getInstance()
{
synchronized
(SingletonDesignPatternInMultiThreadedEnvironment.class)
{
if(null == INSTANCE)
{
INSTANCE =new SingletonDesignPatternInMultiThreadedEnvironment();
}
        return INSTANCE;
}
}
}
```

Output

```
LazySingleton was created 0 ms ago
EagerSingleton was created 1001 ms ago
```

Q5. Write a Java Program to implement command pattern to test Remote Control.

```
interface Command
{
	public void execute();
}

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

class LightOnCommand implements Command
{
	Light light;

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

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

class LightOffCommand implements Command

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

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

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("Stereo is set " +"for CD input");
	}
	public void setDVD()
	{
		System.out.println("Stereo is set"+" for DVD input");
	}
	public void setRadio()
	{
		System.out.println("Stereo is set" +" for Radio");
	}
	public void setVolume(int volume)
	{
	System.out.println("Stereo volume set"+ " to " + volume);
	}
}
```

class StereoOffCommand implements Command

```
{
	Stereo stereo;
	public StereoOffCommand(Stereo stereo)
	{
		this.stereo = stereo;
	}
	public void execute()
	{
	stereo.off();
	}
}
```

```
class StereoOnWithCDCommand implements Command
{
	Stereo stereo;
	public StereoOnWithCDCommand(Stereo stereo)
	{
		this.stereo = stereo;
	}
	public void execute()
	{
		stereo.on();
		stereo.setCD();
		stereo.setVolume(11);
	}
}

class SimpleRemoteControl
{
	Command slot;

	public SimpleRemoteControl()
	{
	}

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

	public void buttonWasPressed()
	{
		slot.execute();
```

```
    }

class RemoteControlTest
{
    public static void main(String[] args)
    {
        SimpleRemoteControl remote =new SimpleRemoteControl();

        Light light = new Light();

        Stereo stereo = new Stereo();

        remote.setCommand(newLightOnCommand(light));
        remote.buttonWasPressed();

        remote.setCommand(newStereoOnWithCDCommand(stereo));
        remote.buttonWasPressed();

        remote.setCommand(newStereoOffCommand(stereo));
        remote.buttonWasPressed();
    }
}
```

Output

```
Light is on
Stereo is on
Stereo is set for CD input
Stereo volume set to 11
Stereo is off
```

Q6. Write a Java Program to implement undo command to test Ceiling fan.

```
interface Command
{
 public void execute();
}

class CeilingFan
 {
 public void on()
{
          System.out.println("Ceiling Fan is on");
 }
 public void off()
 {
         System.out.println("Ceiling Fan is off");
 }
}

 class CeilingFanOnCommand implements Command
{
 CeilingFan c;

 public CeilingFanOnCommand(CeilingFan l)
 {
          this.c = l;
 }

 public void execute() {
 c.on();
```

```
 } }
class CeilingFanOffCommand implements Command
{
CeilingFan c;

public CeilingFanOffCommand(CeilingFan l)
 {
        this.c = l;
}
public void execute()
 {
        c.off();
}
}

 class SimpleRemoteControl
{
 Command slot;

 public SimpleRemoteControl() {}

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

 public void buttonWasPressed()
 {
         slot.execute();
 }

}
```

```
public class Main
 {
 public static void main(String[] args)
 {
 SimpleRemoteControl remote = new SimpleRemoteControl();

 CeilingFan ceilingFan=new CeilingFan();

 CeilingFanOnCommand ceilingFanOn =new CeilingFanOnCommand(ceilingFan);

 remote.setCommand(ceilingFanOn);
 remote.buttonWasPressed();

CeilingFanOffCommand ceilingFanOff =new CeilingFanOffCommand(ceilingFan);

 remote.setCommand(ceilingFanOff);
 remote.buttonWasPressed();

 }
}
```

Output

```
Ceiling Fan is on
Ceiling Fan is off
```

Q7. Write a Java Program to implement Adapter pattern for Enumeration iterator

```
import java.util.*;

 class EnumerationIterator implements Iterator
 {
 Enumeration enumeration;

 public EnumerationIterator(Enumeration enumeration)
 {
        this.enumeration = enumeration;
 }

 public boolean hasNext()
 {
        return enumeration.hasMoreElements();
 }

 public Object next()
 {
        return enumeration.nextElement();
 }

public void remove()
{
        throw new UnsupportedOperationException();
 }
}
```

```
public class Main
 {
 public static void main (String args[])
 {
 Vector v = new Vector(Arrays.asList(args));
 Iterator iterator = new EnumerationIterator(v.elements());
 while (iterator.hasNext())
{
 System.out.println(iterator.next());
 }
 }
}
```

Q8) Write a Java Program to implement Iterator Pattern for Designing Menu like Breakfast, Lunch or Dinner Menu.

```
import java.util.Iterator;
public interface Menu {
	public Iterator<?> createIterator();
	String name;
	public String getName() {
		return name;
	}
}

public class MenuItem
{
	String name;
	String description;
	boolean vegetarian;
	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 double getPrice()
 {
        return price;
}
public boolean isVegetarian()
 {
        return vegetarian;
```

```
}
}
public class PancakeHouseMenu implements Menu
{
        ArrayList<MenuItem> menuItems;

public PancakeHouseMenu()
 {
        name = "BREAKFAST";
        menuItems = new ArrayList<MenuItem>();

        addItem("K&B's Pancake Breakfast",
        "Pancakes with scrambled eggs, and toast",
        true,
        2.99);

        addItem("Regular Pancake Breakfast",
        "Pancakes with fried eggs, sausage",
        false,
        2.99);

        addItem("Blueberry Pancakes",
```

```
        "Pancakes made with fresh blueberries, and blueberry syrup",
        true,
        3.49);

        addItem("Waffles",
        "Waffles, with your choice of blueberries or strawberries",
        true,
        3.59);
        }

public void addItem(String name, String description,
boolean vegetarian, double price)
{
        MenuItem menuItem = new MenuItem(name, description, vegetarian,
price);
        menuItems.add(menuItem);
}

public ArrayList<MenuItem> getMenuItems()
{
        return menuItems;
}
```

```
public Iterator<MenuItem> createIterator()
{
        return menuItems.iterator();
}

// other menu methods here
}
```

```
import java.util.Iterator;

public class DinerMenu implements Menu
{
        static final int MAX_ITEMS = 6;
        int numberOfItems = 0;
        MenuItem[] menuItems;

public DinerMenu()
 {
        name = "LUNCH";
        menuItems = new MenuItem[MAX_ITEMS];

        addItem("Vegetarian BLT",
        "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99);
        addItem("BLT",
        "Bacon with lettuce & tomato on whole wheat", false, 2.99);
        addItem("Soup of the day",
        "Soup of the day, with a side of potato salad", false, 3.29);
        addItem("Hotdog",
        "A hot dog, with saurkraut, relish, onions, topped with cheese",
```

```
        false, 3.05);
        addItem("Steamed Veggies and Brown Rice",
        "Steamed vegetables over brown rice", true, 3.99);
        addItem("Pasta",
        "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
        true, 3.89);
}

public void addItem(String name, String description,
boolean vegetarian, double price)
{
MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
        if (numberOfItems >= MAX_ITEMS) {
        System.err.println("Sorry, menu is full!  Can't add item to menu");
        } else {
        menuItems[numberOfItems] = menuItem;
        numberOfItems = numberOfItems + 1;
}
}

public MenuItem[] getMenuItems()
{
        return menuItems;
}

public Iterator<MenuItem> createIterator()
{
        return new DinerMenuIterator(menuItems);
        //return new AlternatingDinerMenuIterator(menuItems);
}

public
```

```
// other menu methods here
}
```

Also DinerMenu returns its concrete implementation of the Iterator<MenuItem> interface, DinerMenuIterator:

```
import java.util.Iterator;

public class DinerMenuIterator implements Iterator<MenuItem>
{
        MenuItem[] list;
        int position = 0;

public DinerMenuIterator(MenuItem[] list)
 {
        this.list = list;
}

public MenuItem next()
{
        MenuItem menuItem = list[position];
        position = position + 1;
        return menuItem;
}

public boolean hasNext()
{
        if (position >= list.length || list[position] == null) {
        return false;
} else {
        return true;
}
```

```
}

public void remove()
{
        if (position <= 0) {
        throw new IllegalStateException
        ("You can't remove an item until you've done at least one next()");
}
        if (list[position-1] != null)
         {
        for (int i = position-1; i < (list.length-1); i++) {
        list[i] = list[i+1];
        }
        list[list.length-1] = null;
}
}
}
```

public class Waitress

```
{
        ArrayList<Menu> menus;


        public Waitress(ArrayList<Menu> menus) {
        this.menus = menus;
}

public void printMenu()
{
        Iterator<?> menuIterator = menus.iterator();

        System.out.print(MENU\n----\n);
        while(menuIterator.hasNext()) {
```

```
        Menu menu = (Menu)menuIterator.next();
        System.out.print("\n" + menu.getName() + "\n");
        printMenu(menu.createIterator());
}
}

void printMenu(Iterator<?> iterator)
 {
        while (iterator.hasNext()) {
        MenuItem menuItem = (MenuItem)iterator.next();
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + " -- ");
        System.out.println(menuItem.getDescription());
}
}
}
```

To test this program we use the following snippet:

```
public class MenuTestDrive
 {
        public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        ArrayList<Menu> menus = new ArrayList<Menu>();
        menus.add(pancakeHouseMenu);
        menus.add(dinerMenu);
        Waitress waitress = new Waitress(menus);
        waitress.printMenu();

}
}
```

The output printing the menus is:

```
$ java MenuTestDrive
MENU
----
BREAKFAST
K&B's Pancake Breakfast, 2.99 -- Pancakes with scrambled eggs, and toast
Regular Pancake Breakfast, 2.99 -- Pancakes with fried eggs, sausage
Blueberry Pancakes, 3.49 -- Pancakes made with fresh blueberries, and blueberry
syrup
Waffles, 3.59 -- Waffles, with your choice of blueberries or strawberries

LUNCH
Vegetarian BLT, 2.99 -- (Fakin') Bacon with lettuce & tomato on whole wheat
BLT, 2.99 -- Bacon with lettuce & tomato on whole wheat
Soup of the day, 3.29 -- Soup of the day, with a side of potato salad
Hotdog, 3.05 -- A hot dog, with saurkraut, relish, onions, topped with cheese
Steamed Veggies and Brown Rice, 3.99 -- Steamed vegetables over brown rice
Pasta, 3.89 -- Spaghetti with Marinara Sauce, and a slice of sourdough bread
```