**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.io.PrintStream;

import java.util.ArrayList;

import java.util.LinkedList;

import java.util.Queue;

import static com.company.ObserverPattern.so;

public class ObserverPattern {

static PrintStream so = System.out;

public static void main(String[] args) {

so.println("OBSERVER\_PATTERN");

WeatherData wd = new WeatherData();

CurrentConditionsDisplay ccd = new CurrentConditionsDisplay(wd);

StatisticsDisplay sd = new StatisticsDisplay(wd);

ForecastDisplay fd = new ForecastDisplay(wd);

ThirdPartyDisplay third = new ThirdPartyDisplay(wd);

wd.setMeasurements( 80, 65, 30.4);

wd.setMeasurements( 82, 70, 29.2);

wd.setMeasurements( 78, 90, 29.2);

}

}

interface Subject {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

interface Observer { void update(double temperature, double humidity, double pressure); }

interface DisplayElement { void display(); }

// concrete classes

// concrete subject class

class WeatherData implements Subject {

private ArrayList<Observer> observers;

private double temperature, humidity, 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 (Object o : observers) {

Observer obs = (Observer)o;

obs.update(temperature, humidity, pressure);

}

}

public double getTemperature() { return temperature; }

public double getHumidity() { return humidity; }

public double getPressure() { return pressure; }

public void setMeasurements(double temperature, double humidity, double pressure) {

this.temperature = temperature;

this.humidity = humidity;

this.pressure = pressure;

measurementsChanged();

}

public void measurementsChanged() { notifyObservers(); }

}

// concrete observer and display classes

class CurrentConditionsDisplay implements Observer, DisplayElement {

private double temperature, humidity;

private Subject weatherData;

public CurrentConditionsDisplay(Subject weatherData) {

this.weatherData = weatherData;

weatherData.registerObserver(this);

}

public void update(double temperature, double humidity, double pressure) {

this.temperature = temperature;

this.humidity = humidity;

display();

}

public void display() {

so.println(String.format("\nCurrent conditions: %.2f F degrees and %.2f %% humidity", temperature, humidity));

}

}

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

public class changeCase {

    public static void main(String[] args) {

        String str1="Great Power";

        StringBuffer newStr=new StringBuffer(str1);

        for(int i = 0; i < str1.length(); i++) {

            //Checks for lower case character

            if(Character.isLowerCase(str1.charAt(i))) {

                //Convert it into upper case using toUpperCase() function

                newStr.setCharAt(i, Character.toUpperCase(str1.charAt(i)));

            }

            //Checks for upper case character

            else if(Character.isUpperCase(str1.charAt(i))) {

                //Convert it into upper case using toLowerCase() function

                newStr.setCharAt(i, Character.toLowerCase(str1.charAt(i)));

            }

        }

        System.out.println("String after case conversion : " + newStr);

    }

}

**3.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.**

Step 1: Create an abstract class called Pizza which abstract pizza-related data:

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");

}

}

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

public class SingletonDesignPatternInMultiThreadedEnvironment {

    // Step 1: private static variable of INSTANCE variable

    private static volatile

        SingletonDesignPatternInMultiThreadedEnvironment INSTANCE;

    // Step 2: private constructor

    private SingletonDesignPatternInMultiThreadedEnvironment() {

    }

    // Step 3: Provide public static getInstance() method

    // returning INSTANCE after checking

    public static SingletonDesignPatternInMultiThreadedEnvironment

        getInstance() {

        // synchronized block

        synchronized

          (SingletonDesignPatternInMultiThreadedEnvironment.class){

          if(null == INSTANCE){

              INSTANCE =

                new

                SingletonDesignPatternInMultiThreadedEnvironment();

          }

          return INSTANCE;

        }

    }

}

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

// A simple Java program to demonstrate

// implementation of Command Pattern using

// a remote control example.

// An interface for command

interface Command

{

public void execute();

}

// Light class and its corresponding command

// classes

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;

// The constructor is passed the light it

// is going to control.

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();

}

}

// Stereo and its command classes

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)

{

// code to set the 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);

}

}

// A Simple remote control with one button

class SimpleRemoteControl

{

Command slot; // only one button

public SimpleRemoteControl()

{

}

public void setCommand(Command command)

{

// set the command the remote will

// execute

slot = command;

}

public void buttonWasPressed()

{

slot.execute();

}

}

// Driver class

class RemoteControlTest

{

public static void main(String[] args)

{

SimpleRemoteControl remote =

new SimpleRemoteControl();

Light light = new Light();

Stereo stereo = new Stereo();

// we can change command dynamically

remote.setCommand(new

LightOnCommand(light));

remote.buttonWasPressed();

remote.setCommand(new

StereoOnWithCDCommand(stereo));

remote.buttonWasPressed();

remote.setCommand(new

StereoOffCommand(stereo));

remote.buttonWasPressed();

}

}

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

public 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;

String location;

int speed;

public CeilingFan(String location)

{

this.location = location;

speed = OFF;

}

public void high()

{

speed = HIGH;

System.out.println(location + " ceiling fan is on high");

}

public void medium()

{

speed = MEDIUM;

System.out.println(location + " ceiling fan is on medium");

}

public void low()

{

speed = LOW; System.out.println(location + " ceiling fan is on low");

}

public void off()

{

speed = OFF; System.out.println(location + " ceiling fan is off");

}

public int getSpeed()

{

return speed;

}

}

public class CeilingFanHighCommand implements Command

{

CeilingFan ceilingFan;

int prevSpeed;

public CeilingFanHighCommand(CeilingFan ceilingFan)

{

this.ceilingFan = ceilingFan;

}

public void execute()

{

prevSpeed = ceilingFan.getSpeed();

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 if (prevSpeed == CeilingFan.OFF)

{

ceilingFan.off();

}

}

}

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

public class MyEnumeratorAdapter<Object> implements Enumeration<Object> {

private Iterator<Object> adaptee;

public MyEnumeratorAdapter(Iterator<Object> it) {

this.adaptee = it;

}

@Override

public boolean hasMoreElements() {

return adaptee.hasNext();

}

@Override

public Object nextElement() {

return adaptee.next();

}

}

and my arraylist class is:

public class MyArrayList<Object> extends ArrayList<Object> {

public MyArrayList() {

this.enumerator = new MyEnumeratorAdapter<Object>(this.iterator());

}

public Enumeration<Object> enumerator() {

return this.enumerator;

}

public boolean hasMoreElements() {

return this.enumerator.hasMoreElements();

}

public Object nextElement() {

return this.enumerator.nextElement();

}

private static final long serialVersionUID = 1L;

private Enumeration<Object> enumerator;

}

public static void main(String[] args) {

MyArrayList<String> names = new MyArrayList<String>();

names.add("jim");

names.add("jack");

names.add("jai");

for (Enumeration<String> iterator = names.enumerator(); iterator

.hasMoreElements();) {

String name = (String) iterator.nextElement();

System.out.println(name);

}

}

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

Base object

public interface Menu {

public Iterator<MenuItem> createIterator();

}

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 boolean isVegetarian() {

return vegetarian;

}

public double getPrice() {

return price;

}

@Override

public String toString() {

return "MenuItem{" +

"name='" + name + '\'' +

", description='" + description + '\'' +

", vegetarian=" + vegetarian +

", price=" + price +

'}';

}

}

Concrete Class

public class PancakeHouseMenu implements Menu {

ArrayList<MenuItem> menuItems;

public PancakeHouseMenu() {

menuItems = new ArrayList<>();

addItem("K&B Pancake set",

"scrambled egg & toast pancake", true, 2.99);

addItem("Blueberry Pancake",

"Blueberry Pancake", true, 3.49);

}

public void addItem(String name, String description, boolean vegetarian, double price) {

menuItems.add(

new MenuItem(name, description, vegetarian, price)

);

}

public Iterator<MenuItem> createIterator() {

return menuItems.iterator();

}

}

public class DinerMenu implements Menu {

static final int MAX\_ITEMS = 6;

int numberOfItems = 0;

MenuItem[] menuItems;

public DinerMenu() {

menuItems = new MenuItem[MAX\_ITEMS];

addItem("Vegetarian BLT",

"Vegetarian BLT", true, 2.99);

addItem("BLT",

"BLT", false, 2.99);

addItem("Today's soup",

"Soup with potato salad", false, 3.29);

addItem("Hotdog",

"Hotdog with cheese", false, 3.29);

}

public void addItem(String name, String description, boolean vegetarian, double price) {

MenuItem menuItem = new MenuItem(name, description, vegetarian, price);

if (MAX\_ITEMS <= numberOfItems) {

System.out.println("Sorry, menu is full. It cannot add new item");

} else {

menuItems[numberOfItems] = menuItem;

numberOfItems += 1;

}

}

public Iterator<MenuItem> createIterator() {

return new DinerMenuIterator(menuItems);

}

}

public class CafeMenu implements Menu {

Hashtable<String, MenuItem> menuItems = new Hashtable<>();

public CafeMenu() {

addItem("Burger & Fries",

"Burger & potato fries", true, 3.99);

addItem("Today's soup",

"Soup with salad", false, 3.69);

}

public void addItem(String name, String description, boolean vegetarian, double price) {

MenuItem menuItem = new MenuItem(name, description, vegetarian, price);

menuItems.put(menuItem.getName(), menuItem);

}

@Override

public Iterator<MenuItem> createIterator() {

return menuItems.values().iterator();

}

}

Concrete Iterator

public class DinerMenuIterator implements Iterator<MenuItem> {

MenuItem[] items;

int position = 0;

public DinerMenuIterator(MenuItem[] items) {

this.items = items;

}

@Override

public boolean hasNext() {

if (items.length <= position || items[position] == null) {

return false;

} else {

return true;

}

}

@Override

public MenuItem next() {

MenuItem menuItem = items[position];

position += 1;

return menuItem;

}

}

Client

public class Waitress {

List<Menu> menus;

public Waitress(List<Menu> menus) {

this.menus = menus;

}

public void printMenu() {

System.out.println("Menus : ");

menus.forEach(menu -> {

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

printMenu(menu.createIterator());

});

}

private void printMenu(Iterator<MenuItem> iterator) {

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

}

}

Test

public class MenuTest {

public static void main(String[] args) {

List<Menu> menus = Arrays.asList(new PancakeHouseMenu(), new DinerMenu(), new CafeMenu());

Waitress waitress = new Waitress(menus);

waitress.printMenu();

}

}

**9.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. For actions we need to implement methods to insert a quarter, remove a quarter, turning the crank and display gumball.**

**package headfirst.designpatterns.state.gumball;**

public class GumballMachine {

final static int SOLD\_OUT = 0;

final static int NO\_QUARTER = 1;

final static int HAS\_QUARTER = 2;

final static int SOLD = 3;

int state = SOLD\_OUT;

int count = 0;

public GumballMachine(int count) {

this.count = count;

if (count > 0) {

state = NO\_QUARTER;

}

}

public void insertQuarter() {

if (state == HAS\_QUARTER) {

System.out.println("You can't insert another quarter");

} else if (state == NO\_QUARTER) {

state = HAS\_QUARTER;

System.out.println("You inserted a quarter");

} else if (state == SOLD\_OUT) {

System.out.println("You can't insert a quarter, the machine is sold out");

} else if (state == SOLD) {

System.out.println("Please wait, we're already giving you a gumball");

}

}

public void ejectQuarter() {

if (state == HAS\_QUARTER) {

System.out.println("Quarter returned");

state = NO\_QUARTER;

} else if (state == NO\_QUARTER) {

System.out.println("You haven't inserted a quarter");

} else if (state == SOLD) {

System.out.println("Sorry, you already turned the crank");

} else if (state == SOLD\_OUT) {

System.out.println("You can't eject, you haven't inserted a quarter yet");

}

}

public void turnCrank() {

if (state == SOLD) {

System.out.println("Turning twice doesn't get you another gumball!");

} else if (state == NO\_QUARTER) {

System.out.println("You turned but there's no quarter");

} else if (state == SOLD\_OUT) {

System.out.println("You turned, but there are no gumballs");

} else if (state == HAS\_QUARTER) {

System.out.println("You turned...");

state = SOLD;

dispense();

}

}

private void dispense() {

if (state == SOLD) {

System.out.println("A gumball comes rolling out the slot");

count = count - 1;

if (count == 0) {

System.out.println("Oops, out of gumballs!");

state = SOLD\_OUT;

} else {

state = NO\_QUARTER;

}

} else if (state == NO\_QUARTER) {

System.out.println("You need to pay first");

} else if (state == SOLD\_OUT) {

System.out.println("No gumball dispensed");

} else if (state == HAS\_QUARTER) {

System.out.println("No gumball dispensed");

}

}

public void refill(int numGumBalls) {

this.count = numGumBalls;

state = NO\_QUARTER;

}

public String toString() {

StringBuffer result = new StringBuffer();

result.append("\nMighty Gumball, Inc.");

result.append("\nJava-enabled Standing Gumball Model #2004\n");

result.append("Inventory: " + count + " gumball");

if (count != 1) {

result.append("s");

}

result.append("\nMachine is ");

if (state == SOLD\_OUT) {

result.append("sold out");

} else if (state == NO\_QUARTER) {

result.append("waiting for quarter");

} else if (state == HAS\_QUARTER) {

result.append("waiting for turn of crank");

} else if (state == SOLD) {

result.append("delivering a gumball");

}

result.append("\n");

return result.toString();

}

}

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

11. Design simple HR Application using Spring Framework.