Q1) Write a Java Program to implement I/O Decorator for converting uppercase letters tolower 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();
 }
 }
}
```

Q2) Write a Java Program to implement Singleton pattern for multithreading.

```
public class Main {
   public static void main(String ar[]) {
      Test1 t = new Test1();
      Test1  t2  = new  Test1();
      Test1 t3 = new Test1();
      Thread tt = new Thread(t);
      Thread tt2 = new Thread(t2);
      Thread tt3 = new Thread(t3);
      Thread tt4 = new Thread(t);
      Thread tt5 = new Thread(t);
      tt.start();
      tt2.start();
      tt3.start();
      tt4.start();
      tt5.start();

   }
}

      final class Test1 implements Runnable {                              //Override
   public void run() {
      for (int i = 0; i < 5; i++) {
         System.out.println(Thread.currentThread().getName() + " : " +
Single.getInstance().hashCode());
      }
   }

}

class Single {

      private final static Single sing = new Single();
      private Single() {
      }
      public static Single getInstance() {
         return sing;
      }
   }
```

Q.3) Write a JAVA Program to implement built-in support (java.util.Observable) Weatherstation with members temperature, humidity, pressure and methods mesurmentsChanged(), setMesurment(), getTemperature(), getHumidity(), getPressure() Observer Pattern

```
import java.util.Observable;import
java.util.Observer;
class CurrentConditionsDisplay implements Observer, DisplayElement {Observable
        observable;
        private float temperature;private
        float humidity;
    public CurrentConditionsDisplay(Observable observable) {this.observable = observable;
            observable.addObserver(this);
        }
  public void update(Observable obs, Object arg) {if
            (obs instanceof WeatherData) {
                WeatherData weatherData = (WeatherData)obs;
                this.temperature = weatherData.getTemperature();
              this.humidity = weatherData.getHumidity();
                                display();
            }
        }

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

}

interface DisplayElement {
        public void display();
}

class ForecastDisplay implements Observer, DisplayElement {private
        float currentPressure = 29.92f;
        private float lastPressure;
         public ForecastDisplay(Observable observable) {observable.addObserver(this);
        }
        public void update(Observable observable, Object arg) {if
            (observable instanceof WeatherData) {
                WeatherData weatherData = (WeatherData)observable;
                lastPressure = currentPressure;
                currentPressure = weatherData.getPressure();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");
                }
            }
  }

   class HeatIndexDisplay implements Observer, DisplayElement {float
            heatIndex = 0.0f;

            public HeatIndexDisplay(Observable observable) {observable.addObserver(this);
            }
             public void update(Observable observable, Object arg) {if
                    (observable instanceof WeatherData) {
                        WeatherData weatherData = (WeatherData)observable;float t =
               weatherData.getTemperature();
          float rh = weatherData.getHumidity();heatIndex
                    = (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)));
                        display();
                }
            }

             public void display() {

                    System.out.println("Heat index is " + heatIndex);

            }
}

   class StatisticsDisplay implements Observer, DisplayElement {
            private float maxTemp = 0.0f;
            private float minTemp = 200;
            private float tempSum= 0.0f;
            private int numReadings;


             public StatisticsDisplay(Observable observable) {observable.addObserver(this);
            }

            public void update(Observable observable, Object arg) {if
                    (observable instanceof WeatherData) {
                        WeatherData weatherData = (WeatherData)observable;float temp
         = weatherData.getTemperature();
```

```
                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);
          }
}

 class WeatherData extends Observable {
          private float temperature;
          private float humidity;
          private float pressure;
                    public  WeatherData()  { } public
          void measurementsChanged() {

                    setChanged();
                    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 Main {
          public static void main(String[] args) {
                    WeatherData weatherData = new WeatherData();
CurrentConditionsDisplay currentConditions = 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);
            }
}
```

Q4) 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.

```
//program for ny and Chicago cheese pizza
import java.util.ArrayList;
class ChicagoPizzaStore extends PizzaStore
 {
        Pizza createPizza(String item)
        {
        if (item.equals("cheese"))
          {
      return new ChicagoStyleCheesePizza();
          }
        else return null;
        }
}
class ChicagoStyleCheesePizza extends Pizza
 {
        public ChicagoStyleCheesePizza()
         {
        name = "Chicago Style Deep Dish Cheese Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";
        toppings.add("Shredded  Mozzarella  Cheese");
        }
        void cut()
        {
        System.out.println("Cutting the pizza into square slices");
        }
}
class DependentPizzaStore
{       public Pizza createPizza(String style, String type)
   {   Pizza pizza = null;
            if (style.equals("NY"))
            {
              if (type.equals("cheese"))
                {
                pizza = new NYStyleCheesePizza();
           }
            }

         if (style.equals("Chicago"))
              {
                if (type.equals("cheese"))
```

```
                {
                pizza = new ChicagoStyleCheesePizza();
                }
                }

            else
            {
              System.out.println("Error: invalid type of pizza");     return null;
            }
          pizza.prepare();
          pizza.bake();
          pizza.cut();
          pizza.box();
          return pizza;
          }
}

class NYPizzaStore extends PizzaStore
 {
   Pizza createPizza(String item)
          {
          if (item.equals("cheese"))
            {
            return new NYStyleCheesePizza();
            }
   else return null;
 }
}
class NYStyleCheesePizza extends Pizza
{
          public  NYStyleCheesePizza()
          {
          name = "NY Style Sauce and Cheese Pizza";
          dough = "Thin Crust Dough";
          sauce = "Marinara Sauce";
          toppings.add("Grated  Reggiano  Cheese");
          }}
abstract class Pizza
{
          String name;
          String dough;
          String sauce;
          ArrayList toppings = new ArrayList();
          void prepare()
          {
            System.out.println("Preparing " + name);
            System.out.println("Tossing dough...");
            System.out.println("Adding sauce...");
            System.out.println("Adding toppings: ");
            for (int i = 0; i < toppings.size(); i++)
```

```
            {
            System.out.println("   " + toppings.get(i));
            }
        }
        void bake()
        {
        System.out.println("Bake for 25 minutes at 350");
        }
        void cut()
        {
        System.out.println("Cutting the pizza into diagonal slices");
        }
        void box()
        {
        System.out.println("Place pizza in official PizzaStore box");
        }
        public String getName()
        {
        return name;
        }
        public String toString()
        {
        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();
        }
}
abstract class PizzaStore
        {
        abstract Pizza createPizza(String item);
        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;
        }
        }
public class Main
{
        public static void main(String[] args)
```

```
        {
        PizzaStore nyStore =  new  NYPizzaStore();
        PizzaStore chicagoStore = new ChicagoPizzaStore();
        Pizza pizza = nyStore.orderPizza("cheese");

        System.out.println("Poonam ordered a " + pizza.getName() + "\n");
        pizza = chicagoStore.orderPizza("cheese");

        System.out.println("Kadambari ordered a " + pizza.getName() + "\n");
        pizza = nyStore.orderPizza("cheese");
                        }

}

/* program for all pizza types*/
import java.util.ArrayList;
class ChicagoPizzaStore extends PizzaStore
 {
        Pizza createPizza(String item)
        {
        if (item.equals("cheese"))
          {
     return new ChicagoStyleCheesePizza();
          }
        else if (item.equals("veggie"))
          {
        return new ChicagoStyleVeggiePizza();
        }
        else if (item.equals("clam"))
          {
         return new ChicagoStyleClamPizza();
          }
        else if (item.equals("pepperoni"))
          {
        return new ChicagoStylePepperoniPizza();
          }
        else return null;
        }
}
class ChicagoStyleCheesePizza extends Pizza
 {
        public ChicagoStyleCheesePizza()
         {
        name = "Chicago Style Deep Dish Cheese Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";
        toppings.add("Shredded  Mozzarella  Cheese");
        }
        void cut()
        {
```

```
        System.out.println("Cutting the pizza into square slices");
        }
}
class ChicagoStyleClamPizza extends Pizza
{
        public ChicagoStyleClamPizza()
         {
        name = "Chicago Style Clam Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";
        toppings.add("Shredded Mozzarella Cheese");
   toppings.add("Frozen Clams from Chesapeake Bay");
        }
        void cut()
        {
        System.out.println("Cutting the pizza into square slices");
        }
}
class ChicagoStylePepperoniPizza extends Pizza
{
        public ChicagoStylePepperoniPizza()
        {
        name = "Chicago Style Pepperoni Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";
        toppings.add("Shredded Mozzarella Cheese");
        toppings.add("Black Olives");
        toppings.add("Spinach");
        toppings.add("Eggplant");
        toppings.add("Sliced Pepperoni");
        }
        void cut()
        {
        System.out.println("Cutting the pizza into square slices");
        }
}
class ChicagoStyleVeggiePizza extends Pizza
{
        public ChicagoStyleVeggiePizza()
        {
        name = "Chicago Deep Dish Veggie Pizza";
        dough = "Extra Thick Crust Dough";
        sauce = "Plum Tomato Sauce";
        toppings.add("Shredded Mozzarella Cheese");
   toppings.add("Black Olives");
        toppings.add("Spinach");
        toppings.add("Eggplant");
        }
        void cut()
        {
```

```
        System.out.println("Cutting the pizza into square slices");
        }
}
class DependentPizzaStore
{       public Pizza createPizza(String style, String type)
  {   Pizza pizza = null;
          if (style.equals("NY"))
          {
            if (type.equals("cheese"))
              {
              pizza = new NYStyleCheesePizza();
         }
       else if (type.equals("veggie"))
            {
            pizza = new NYStyleVeggiePizza();
            }
          else if (type.equals("clam"))
            {
            pizza = new NYStyleClamPizza();
            }
          else if (type.equals("pepperoni"))
              {
              pizza = new NYStylePepperoniPizza();
              }
            }
       else if (style.equals("Chicago"))
            {
              if (type.equals("cheese"))
              {
              pizza = new ChicagoStyleCheesePizza();
              }
              else if (type.equals("veggie"))
              {
              pizza = new ChicagoStyleVeggiePizza();
              }else if (type.equals("clam"))
              {
                 pizza = new ChicagoStyleClamPizza();
              }
              else if (type.equals("pepperoni"))
              {
                 pizza = new ChicagoStylePepperoniPizza();
              }
            }
          else
          {
            System.out.println("Error: invalid type of pizza");     return null;
          }
         pizza.prepare();
         pizza.bake();
         pizza.cut();
```

```
        pizza.box();
        return pizza;
        }
}

class NYPizzaStore extends PizzaStore
 {
   Pizza createPizza(String item)
        {
        if  (item.equals("cheese"))
          {
          return new NYStyleCheesePizza();
          }
        else if (item.equals("veggie"))
          {
          return new NYStyleVeggiePizza();
          }
        else if (item.equals("clam"))
          {
            return new NYStyleClamPizza();
              }
        else if (item.equals("pepperoni"))
          {
            return new NYStylePepperoniPizza();
          }
   else return null;
 }
}
class NYStyleCheesePizza extends Pizza
{
        public  NYStyleCheesePizza()
        {
        name = "NY Style Sauce and Cheese Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";
        toppings.add("Grated  Reggiano  Cheese");
        }
}
class NYStyleClamPizza extends Pizza
        {
        public NYStyleClamPizza()
        {
        name = "NY Style Clam Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";
        toppings.add("Grated Reggiano Cheese");
        toppings.add("Fresh Clams from Long Island Sound");
        }
}
```

```
class NYStylePepperoniPizza extends Pizza
{
        public NYStylePepperoniPizza()
        {
        name = "NY Style Pepperoni Pizza";dough =
        "Thin Crust Dough";
        sauce = "Marinara Sauce";
        toppings.add("Grated  Reggiano  Cheese");
        toppings.add("Sliced Pepperoni");
        toppings.add("Garlic");
        toppings.add("Onion");
        toppings.add("Mushrooms");
        toppings.add("Red Pepper");
        }
}
class NYStyleVeggiePizza extends Pizza
{
public NYStyleVeggiePizza()
        {
        name =              "NY Style Veggie Pizza";
        dough = "Thin Crust Dough";
        sauce = "Marinara Sauce";
        toppings.add("Grated  Reggiano  Cheese");
        toppings.add("Garlic");
        toppings.add("Onion");
        toppings.add("Mushrooms");
        toppings.add("Red Pepper");
        }
}
abstract class Pizza
{
        String name;
        String dough;
        String sauce;
        ArrayList toppings = new ArrayList();
        void prepare()
        {
           System.out.println("Preparing " + name);
           System.out.println("Tossing dough...");
           System.out.println("Adding sauce...");
           System.out.println("Adding toppings: "); for
           (int i = 0; i < toppings.size(); i++)
              {
              System.out.println("   " + toppings.get(i));
              }
        }
        void bake()
        {
        System.out.println("Bake for 25 minutes at 350");
        }
        void cut()
        {
        System.out.println("Cutting the pizza into diagonal slices");
```

```
        }
        void box()
        {
        System.out.println("Place pizza in official PizzaStore box");
        }
        public String getName()
        {
        return name;
        }
        public String toString()
        {
        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();
        }
}
abstract class PizzaStore
        {
        abstract Pizza createPizza(String item);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;
        }
        }
public class Main
{
        public static void main(String[] args)
        {
        PizzaStore nyStore =  new  NYPizzaStore(); PizzaStore
        chicagoStore = new ChicagoPizzaStore();Pizza pizza =
        nyStore.orderPizza("cheese");
        System.out.println("Ethan ordered a " + pizza.getName() + "\n");pizza
        = chicagoStore.orderPizza("cheese"); System.out.println("Joel ordered a
        " + pizza.getName() + "\n"); pizza = nyStore.orderPizza("clam");
        System.out.println("Ethan ordered a " + pizza.getName() + "\n");pizza
        =  chicagoStore.orderPizza("clam"); System.out.println("Joel ordered a "
        + pizza.getName() + "\n");
                        pizza = nyStore.orderPizza("pepperoni"); System.out.println("Ethan
        ordered a " + pizza.getName() + "\n");
                pizza = chicagoStore.orderPizza("pepperoni");
        System.out.println("Joel ordered a " + pizza.getName() + "\n");
                        pizza = nyStore.orderPizza("veggie");
```

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

Q5) 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());
                }
        }
}
```

Q6) 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 l1;

        public LightOnCommand(Light a) {
                this.l1 = a;
        }

        public void execute() {
                l1.on();
        }

}
 class LightOffCommand implements Command {
Light l1;
public LightOffCommand(Light a) {
this.l1 = a;
}
public void execute() {
l1.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 r1 = new SimpleRemoteControl();
                Light l1 = new Light();

                LightOnCommand lo = new LightOnCommand(l1);
                r1.setCommand(lo);
                r1.buttonWasPressed();

                LightOffCommand lO = new LightOffCommand(l1);
                r1.setCommand(lO);
                r1.buttonWasPressed();

          }
}
```

Q7) 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();


   }
}
```

Q8) 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, StereoOnWithCDCommand.

```
interface Command {
        public void execute();
}class Stereo { public void On(){
     System.out.println("Stereo is on");
  }
}
class GarageDoor {
  public void Up(){
     System.out.println("Garage Door is Up");
  }
}
 class GarageDoorUpCommand implements Command {
        GarageDoor c;

        public GarageDoorUpCommand(GarageDoor l) {
                this.c = l;
        }

        public void execute() {
                 c.Up();
        }

}
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  StereoOn implements Command {
Stereo s;
public StereoOn(Stereo l) {
this.s = l;
}
public void execute() {
s.On();
}
}
 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();

Light light = new Light();
LightOnCommand lightOn = new LightOnCommand(light);
remote.setCommand(lightOn);
remote.buttonWasPressed();
LightOffCommand lightOff = new LightOffCommand(light);
remote.setCommand(lightOff);
remote.buttonWasPressed();


GarageDoor garageDoor = new GarageDoor();
GarageDoorUpCommand garageDoorUp = new GarageDoorUpCommand(garageDoor);
 remote.setCommand(garageDoorUp);
remote.buttonWasPressed();

 Stereo s1=new Stereo();
StereoOn s2 =new StereoOn(s1);
remote.setCommand(s2);
remote.buttonWasPressed();

   }
```

Q8) Write a Java Program to implement State Pattern for Gumball Machine. Create instancevariable that holds current state from there, we just need to handle all actions, behaviors and state transition that can happen

```
interface State {

        public void insertQuarter();
        public void ejectQuarter();
        public void turnCrank();
        public void dispense();

        public void refill();
}
 class NoQuarterState implements State {
   GumballMachine gumballMachine;

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

        public void insertQuarter() {
                System.out.println("You inserted a quarter");
                gumballMachine.setState(gumballMachine.getHasQuarterState());
        }

        public void ejectQuarter() {
                System.out.println("You haven't inserted a quarter");
        }

        public void turnCrank() {
                System.out.println("You turned, but there's no quarter");
         }

        public void dispense() {
                System.out.println("You need to pay first");
        }

        public void refill() { }

        public String toString() {
                return "waiting for quarter";
        }
}

 class GumballMachine {

        State soldOutState;
        State noQuarterState;
```

```
        State hasQuarterState;
        State soldState;

        State state;
        int count = 0;

        public GumballMachine(int numberGumballs) {
                soldOutState = new SoldOutState(this);
                noQuarterState = new NoQuarterState(this);
                hasQuarterState = new HasQuarterState(this);
                soldState = new SoldState(this);

                this.count = numberGumballs;
                if (numberGumballs > 0) {
                        state = noQuarterState;
                } else {
                        state = soldOutState;
                }
        }

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

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

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

        void releaseBall() {
                System.out.println("A gumball comes rolling out the slot...");
                if (count != 0) {
                        count = count - 1;
                }
        }

        int getCount() {
                return count;
        }

        void refill(int count) {
                this.count += count;
                System.out.println("The gumball machine was just refilled; it's new count
is: " + this.count);
                state.refill();
        }
```

```
        void setState(State state) {
                this.state = state;
        }
    public State getState() {
      return state;
    }

    public State getSoldOutState() {
      return soldOutState;
    }

    public State getNoQuarterState() {
      return noQuarterState;
    }

    public State getHasQuarterState() {
      return hasQuarterState;
    }

    public State getSoldState() {
      return soldState;
    }

        public String toString() {
                StringBuffer result = new StringBuffer();
                result.append("\nMighty Gumball, Inc.");
                result.append("\nJava-enabled Standing Gumball Model #2004");
                result.append("\nInventory: " + count + " gumball");
                if (count != 1) {
                        result.append("s");
                }
                result.append("\n");
                result.append("Machine is " + state + "\n");
                return result.toString();
        }
}


 class HasQuarterState implements State {
        GumballMachine gumballMachine;

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

        public void insertQuarter() {
                System.out.println("You can't insert another quarter");
        }
```

```
        public void ejectQuarter() {
                System.out.println("Quarter returned");
                gumballMachine.setState(gumballMachine.getNoQuarterState());
        }

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

    public void dispense() {
      System.out.println("No gumball dispensed");
    }

    public void refill() { }

        public String toString() {
                return "waiting for turn of crank";
        }
}

 class SoldState implements State {

    GumballMachine gumballMachine;

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

        public void insertQuarter() {
                System.out.println("Please wait, we're already giving you a gumball");
        }

        public void ejectQuarter() {
                System.out.println("Sorry, you already turned the crank");
        }

        public void turnCrank() {
                System.out.println("Turning twice doesn't get you another gumball!");
        }

        public void dispense() {
                gumballMachine.releaseBall();
                if (gumballMachine.getCount() > 0) {
                        gumballMachine.setState(gumballMachine.getNoQuarterState());
                } else {
                        System.out.println("Oops, out of gumballs!");
                        gumballMachine.setState(gumballMachine.getSoldOutState());
                }
        }
```

```
        public void refill() { }

        public String toString() {
            return "dispensing a gumball";
        }
}

 class SoldOutState implements State {
   GumballMachine gumballMachine;

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

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

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

        public void turnCrank() {
            System.out.println("You turned, but there are no gumballs");
        }
        public void dispense() {
            System.out.println("No gumball dispensed");
        }

        public void refill() {
             gumballMachine.setState(gumballMachine.getNoQuarterState());
        }

        public String toString() {
            return "sold out";
        }

}
public class Main {

        public static void main(String[] args) {
            GumballMachine gumballMachine = new GumballMachine(2);

            System.out.println(gumballMachine);

            gumballMachine.insertQuarter();
            gumballMachine.turnCrank();

            System.out.println(gumballMachine);
```

```
            gumballMachine.insertQuarter();
            gumballMachine.turnCrank();
            gumballMachine.insertQuarter();
            gumballMachine.turnCrank();

            gumballMachine.refill(5);
            gumballMachine.insertQuarter();
            gumballMachine.turnCrank();

            System.out.println(gumballMachine);
        }
}
```

Q9) 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

```
abstract class Duck {
 FlyBehaviour flyBehaviour;
 QuackBehaviour quackBehaviour;

 public Duck() {
 }

 public abstract void display();

 public void performFly() {
  flyBehaviour.fly();
 }

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

 public void swim() {
  System.out.println("All ducks float even decoys");
 }

 public void setFlyBehaviour(FlyBehaviour fb) {
  flyBehaviour = fb;
 }

 public void setQuackBehaviour(QuackBehaviour qb) {
  QuackBehaviour q;
 }
}

class MallardDuck extends Duck {

 public MallardDuck() {
  quackBehaviour = new Quack();
  flyBehaviour = new FlyWithWings();
 }

 public void display() {
  System.out.println("I'm a real Mallard duck");
 }
}

interface FlyBehaviour {
 public void fly();
}
```

```
 interface QuackBehaviour {
  public void quack() {
  System.out.println("Quack");
  }
}

 class Quack implements QuackBehaviour {
  public void quack() {
  System.out.println("Quack");
  }
}
 class FlyWithWings implements FlyBehaviour {
  public void fly() {
   System.out.println("I'm flying!!");
  }
}


public class Main {
  public static void main(String[] args) {
    Duck mallard = new MallardDuck();
    mallard.performQuack();
    mallard.performFly();
  }
}

/*Simple Programme*/
interface DuckB
{
   public void oper();
}

class Fly implements DuckB
{
   public void oper()
   {
      System.out.println("Duck Flies");
   }
}
 class Quack implements DuckB
{
   public void oper()
   {
      System.out.println("Duck Sounds Quack Quack");
   }
}
class Context
{
```

```
   private DuckB s1;
    public Context(DuckB p)
    {
       this.s1=p;
    }
    public void est()
    {
        s1.oper();
    }
}


public class Main
{
public static void main(String[] args) {
Context c1=new Context(new Fly());
System.out.println("Duck Behaviour");
c1.est();
c1=new Context(new Quack());
System.out.println("Duck Behaviour ");
c1.est();
          }
}
```

Q10) Write a java program to implement Adapter pattern to design Heart Model to BeatModel

```
interface BeatModelInterface {
void initialize();
void on();
void off();
void setBPM(int bpm);
int getBPM();
void  registerObserver(BeatObserver  o);
void  removeObserver(BeatObserver  o);
void  registerObserver(BPMObserver  o);
void  removeObserver(BPMObserver  o);
}
 class BeatModel implements BeatModelInterface, MetaEventListener {
Sequencer sequencer;
ArrayList beatObservers = new ArrayList();
ArrayList bpmObservers = new ArrayList();
int bpm = 90;
// other instance variables here
public void initialize() {
setUpMidi();
buildTrackAndStart();
}
public  void  on()  {
sequencer.start();
setBPM(90);
}
public void off() {
setBPM(0);
sequencer.stop();
}
public void  setBPM(int  bpm)  {
this.bpm = bpm;
sequencer.setTempoInBPM(getBPM());
notifyBPMObservers();
}
public int getBPM() {
return bpm;
}
void beatEvent() {
notifyBeatObservers();
}
// Code to register and notify observers
// Lots of MIDI code to handle the beat
}
 class DJView implements ActionListener, BeatObserver, BPMObserver {
BeatModelInterface model;
ControllerInterface controller;
JFrame viewFrame;
JPanel viewPanel;
BeatBar beatBar;
```

```
JLabel bpmOutputLabel;
public DJView(ControllerInterface controller, BeatModelInterface model) {
this.controller = controller;
this.model = model;
model.registerObserver((BeatObserver)this);
model.registerObserver((BPMObserver)this);
public void createView() {
// Create all Swing components here
}
public void updateBPM() {
int bpm = model.getBPM();
if (bpm == 0) {
bpmOutputLabel.setText("offline");
} else {
bpmOutputLabel.setText("Current BPM: " + model.getBPM());
}
}
public void updateBeat() {
beatBar.setValue(100);
}
}

 interface ControllerInterface {
void start();
void stop();
void increaseBPM();
void decreaseBPM();
void setBPM(int bpm);
}

 class BeatController implements ControllerInterface {
BeatModelInterface model;
DJView view;
public BeatController(BeatModelInterface model) {
this.model = model;
view = new DJView(this, model);
view.createView();
view.createControls();
view.disableStopMenuItem();
view.enableStartMenuItem();
model.initialize();
}
public void start() {
model.on();
view.disableStartMenuItem();
view.enableStopMenuItem();
}
public void stop() {
model.off();
view.disableStopMenuItem();
view.enableStartMenuItem();
```

```
}
public void increaseBPM() {
int bpm = model.getBPM();
model.setBPM(bpm + 1);
public void decreaseBPM() {
int bpm = model.getBPM();
model.setBPM(bpm - 1);
}
public void setBPM(int bpm) {
model.setBPM(bpm);
}
}

public class Main {
public static void main (String[] args) {
BeatModelInterface model = new BeatModel();
ControllerInterface controller = new BeatController(model);
}
}
```

Q11) Write a Java Program to implement Decorator Pattern for interface Car to define theassemble() method and then decorate it to Sports car and Luxury Car

```
 interface Car {
public void assemble();
}
 class BasicCar implements Car {        @Override
public void assemble() {
System.out.print("Basic Car.");
}
}
 class CarDecorator implements Car {
protected Car car;
public CarDecorator(Car c){
this.car=c;
}
@Override
public void assemble() {
this.car.assemble();
}
}

class SportsCar extends CarDecorator {
public SportsCar(Car c) {
super(c);
}
@Override
public void assemble(){
car.assemble();
System.out.print(" Adding features of Sports Car.");
}
}
class LuxuryCar extends CarDecorator {
public LuxuryCar(Car c) {
super(c);
}
public void assemble(){
car.assemble();
System.out.print(" Adding features of Luxury Car.");
}
}
public class Main {
public static void main(String[] args) {
Car s1 = new SportsCar(new BasicCar());
s1.assemble();
Car s2 = new LuxuryCar(new BasicCar());
s2.assemble();
}
}
```

Q12) Write a Java Program to implement an Adapter design pattern in mobile charger. Definetwo 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

```
class  Volt  {
   private int volts;
   public Volt(int v) { this.volts=v; }
   public int getVolts() { return volts; }
   public void setVolts(int volts) { this.volts = volts; }
}
 class Socket {
   public Volt getVolt(){ return new Volt(120); }
}
 interface SocketAdapter {
   public Volt get120Volt();
   public Volt get12Volt();
   public Volt get3Volt();
}
 class SocketClassAdapterImpl extends Socket implements SocketAdapter {
   @Override
   public Volt get120Volt() {
      return getVolt();
   }

   @Override
   public Volt get12Volt() {
      Volt v = getVolt();
      return convertVolt(v,10);
   }

   @Override
   public Volt get3Volt() {
      Volt v = getVolt();
      return convertVolt(v,40);
   }

   private Volt convertVolt(Volt v, int i) {
      return new Volt(v.getVolts()/i);
   }
}
 class SocketObjectAdapterImpl implements SocketAdapter {
   // using composition for adapter pattern
   private Socket sock = new Socket();

   @Override
   public Volt get120Volt() {
```

```
    return sock.getVolt();
  }

  @Override
  public Volt get12Volt() {
    Volt v = sock.getVolt();
    return convertVolt(v,10);
  }

  @Override
  public Volt get3Volt() {
    Volt v = sock.getVolt();
    return convertVolt(v,40);
  }

  private Volt convertVolt(Volt v, int i) {
    return new Volt(v.getVolts()/i);
  }
}

public class Main {
  public static void main(String[] args) {
    testClassAdapter();
    testObjectAdapter();
  }

  private static void testObjectAdapter() {
    SocketAdapter sockAdapter = new SocketObjectAdapterImpl();Volt v3
    = getVolt(sockAdapter,3);
    Volt v12 = getVolt(sockAdapter,12); Volt
    v120 = getVolt(sockAdapter,120);
    System.out.println("v3 volts using Object Adapter="+v3.getVolts());
    System.out.println("v12 volts using Object Adapter="+v12.getVolts());
    System.out.println("v120 volts using Object Adapter="+v120.getVolts());
  }
  private static void testClassAdapter() {
    SocketAdapter sockAdapter = new SocketClassAdapterImpl();Volt v3
    = getVolt(sockAdapter,3);
    Volt v12 = getVolt(sockAdapter,12); Volt
    v120 = getVolt(sockAdapter,120);
    System.out.println("v3 volts using Class Adapter="+v3.getVolts());
    System.out.println("v12 volts using Class Adapter="+v12.getVolts());
    System.out.println("v120 volts using Class Adapter="+v120.getVolts());
  }

  private static Volt getVolt(SocketAdapter sockAdapter, int i) {switch
    (i){
      case 3: return sockAdapter.get3Volt(); case
      12: return sockAdapter.get12Volt();
      case 120: return sockAdapter.get120Volt();
      default: return sockAdapter.get120Volt();
    }
  }}
```

Q13) Write a Java Program to implement Facade Design Pattern for HomeTheater

```
package headfirst.facade.hometheater;

class   Amplifier   {
String description;
//Tuner  tuner;
DvdPlayer  dvd;
CdPlayer cd;

public Amplifier(String description) {
 this.description = description;
}

public void on() {
 System.out.println(description + " on");
}

public void off() {
 System.out.println(description + " off");
}

public void setStereoSound() {
 System.out.println(description + " stereo mode on");
}

public void setSurroundSound() {
 System.out.println(description + " surround sound on (5 speakers, 1 subwoofer)");
}
public void setVolume(int level) {
 System.out.println(description + " setting volume to " + level);
}

public void setTuner(Tuner tuner) {
 System.out.println(description + " setting tuner to " + dvd);
 this.tuner = tuner;
}

public void setDvd(DvdPlayer dvd) {
 System.out.println(description + " setting DVD player to " + dvd);
 this.dvd = dvd;
}

public void setCd(CdPlayer  cd)  {
 System.out.println(description + " setting CD player to " + cd);
 this.cd = cd;
}

public String toString() {
```

```
 return description;
 }
}

class CdPlayer {
String description;
int currentTrack;
Amplifier amplifier;
String title;

public CdPlayer(String description, Amplifier amplifier) {
 this.description = description;
 this.amplifier = amplifier;
}

public void on() {
 System.out.println(description + " on");
}

public void off() {
 System.out.println(description + " off");
}

public void eject() {
 title = null;
 System.out.println(description + " eject");
}

public void play(String title) {
 this.title = title;
 currentTrack = 0;
 System.out.println(description + " playing \"" + title + "\"");
}

public void play(int track) {
 if (title == null) {
  System.out.println(description + " can't play track " + currentTrack +
    ", no cd inserted");
 } else {
  currentTrack = track;
  System.out.println(description + " playing track " + currentTrack);
 }
}

public void stop() {
 currentTrack = 0;
 System.out.println(description + " stopped");
}

public void pause() {
 System.out.println(description + " paused \"" + title + "\"");
```

```
 }

 public String toString() {
  return description;
 }
}

class DvdPlayer {
 String description;
 int currentTrack;
 Amplifier amplifier;
 String movie;

 public DvdPlayer(String description, Amplifier amplifier) {
  this.description = description;
  this.amplifier = amplifier;
 }

 public void on() {
  System.out.println(description + " on");
 }

 public void off() {
  System.out.println(description + " off");
 }

      public void eject() {
  movie = null;
            System.out.println(description + " eject");
      }

 public void play(String movie) {
  this.movie = movie;
  currentTrack = 0;
  System.out.println(description + " playing \"" + movie + "\"");
 }

 public void play(int track) {
  if (movie == null) {
   System.out.println(description + " can't play track " + track + " no dvd inserted");
  } else {
   currentTrack = track;
   System.out.println(description + " playing track " + currentTrack + " of \"" + movie +
"\"");
  }
 }

 public void stop() {
  currentTrack = 0;
  System.out.println(description + " stopped \"" + movie + "\"");
 }
```

```
 public void pause() {
  System.out.println(description + " paused \"" + movie + "\"");
 }

 public void setTwoChannelAudio() {
  System.out.println(description + " set two channel audio");
 }

 public void setSurroundAudio() {
  System.out.println(description + " set surround audio");
 }

 public String toString() {
  return description;
 }
}

 class Projector {
 String description;
 DvdPlayer dvdPlayer;

 public Projector(String description, DvdPlayer dvdPlayer) {
  this.description = description;
  this.dvdPlayer = dvdPlayer;
 }

 public void on() {
  System.out.println(description + " on");
 }

 public void off() {
  System.out.println(description + " off");
 }

 public void wideScreenMode() {
  System.out.println(description + " in widescreen mode (16x9 aspect ratio)");
 }

 public void tvMode() {
  System.out.println(description + " in tv mode (4x3 aspect ratio)");
 }

      public String toString() {
            return description;
      }
}

 class TheaterLights {
 String description;
```

```
 public TheaterLights(String description) {
  this.description = description;
 }
 public void on() {
  System.out.println(description + " on");
 }

 public void off() {
  System.out.println(description + " off");
 }

 public void dim(int level) {
  System.out.println(description + " dimming to " + level + "%");
 }

       public String toString() {
             return description;
       }
}

class Screen {
 String description;

 public Screen(String description) {
  this.description = description;
 }
 public void up() {
  System.out.println(description + " going up");
 }

 public void down() {
  System.out.println(description + " going down");
 }

        public String toString() {
             return description;
       }
}

 class PopcornPopper {
 String description;

 public PopcornPopper(String description) {
  this.description = description;
 }

 public void on() {
  System.out.println(description + " on");
 }
```

```
 public void off() {
  System.out.println(description + " off");
 }

 public void pop() {
  System.out.println(description + " popping popcorn!");
 }


      public String toString() {
            return description;
      }
}

 class HomeTheaterFacade {
          Amplifier  amp;
          Tuner tuner;
          DvdPlayer dvd;
          CdPlayer cd;
          Projector projector;
          TheaterLights lights;
          Screen screen;
          PopcornPopper popper;

          public HomeTheaterFacade(Amplifier amp,
                                   Tuner tuner,
                                   DvdPlayer dvd,
                                   CdPlayer cd,
                                   Projector projector,
                                   Screen screen,
                                   TheaterLights lights,
                                   PopcornPopper popper) {

                this.amp = amp;
             this.tuner = tuner;
                this.dvd = dvd;
                this.cd = cd;
                this.projector = projector;
                this.screen = screen;
                this.lights = lights;
                this.popper = popper;
          }

          public void watchMovie(String movie) {
                System.out.println("Get ready to watch a movie...");
                popper.on();
                popper.pop();
                lights.dim(10);
                screen.down();
                projector.on();
```

```
            projector.wideScreenMode();
            amp.on();
            amp.setDvd(dvd);
            amp.setSurroundSound();
            amp.setVolume(5);
            dvd.on();
            dvd.play(movie);
        }


        public void endMovie() {
            System.out.println("Shutting movie theater down...");
            popper.off();
            lights.on();
            screen.up();
            projector.off();
            amp.off();
            dvd.stop();
            dvd.eject();
            dvd.off();
        }

public void listenToCd(String cdTitle) {
System.out.println("Get ready for an audiopile experence...");
            lights.on();
            amp.on();
            amp.setVolume(5);
            amp.setCd(cd);
            amp.setStereoSound();
            cd.on();
            cd.play(cdTitle);
        }

        public void endCd() {
            System.out.println("Shutting down CD...");
            amp.off();
            amp.setCd(cd);
            cd.eject();
            cd.off();
        }

        public void listenToRadio(double frequency) {
            System.out.println("Tuning in the airwaves...");
            tuner.on();
            tuner.setFrequency(frequency);
            amp.on();
            amp.setVolume(5);
            amp.setTuner(tuner);
        }

        public void endRadio() {
```

```
                System.out.println("Shutting down the tuner...");
                tuner.off();
                amp.off();
        }
}

public class Main {
        public static void main(String[] args) {
                Amplifier amp = new Amplifier("Top-O-Line Amplifier");
                Tuner tuner = new Tuner("Top-O-Line AM/FM Tuner", amp);
                DvdPlayer dvd = new DvdPlayer("Top-O-Line DVD Player", amp);
                CdPlayer cd = new CdPlayer("Top-O-Line CD Player", amp);
                Projector projector = new Projector("Top-O-Line Projector", dvd);
                TheaterLights lights = new TheaterLights("Theater Ceiling Lights");
                Screen screen = new Screen("Theater Screen");
                PopcornPopper popper = new PopcornPopper("Popcorn Popper");

                HomeTheaterFacade homeTheater =
                                new HomeTheaterFacade(amp, dvd, cd,
                                                projector, screen, lights, popper);

                homeTheater.watchMovie("Raiders of the Lost Ark");
                homeTheater.endMovie();
        }
}
```

Q15) 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

```
import java.util.ArrayList;
import java.util.List;

class Subject
{
	private List<Observer>observers=new ArrayList<Observer>();
	private int state;

	public int getState()
	{
	return state;
	}

	public void setState(int s)
	{
	this.state=s;
	notifyAllObservers();
	}

	public void attach (Observer o1)
	{
	observers.add(o1);
	}

	public void notifyAllObservers()
	{
		for(Observer o1: observers)
		{
		o1.update();
		}
	}
}
abstract class Observer
{
protected Subject s1;
public abstract void update();
}
class BinaryObserver extends Observer
	{
	public BinaryObserver(Subject s)
	{
	this.s1=s;
	this.s1.attach(this);
	}
	public void update()
```

```
        {
        System.out.println("Binary String:" +Integer.toBinaryString(s1.getState()));
        }
}
class OctalObserver extends Observer
        {
        public OctalObserver(Subject s)
        {
        this.s1=s;
        this.s1.attach(this);
        }
        public void update()
        {
        System.out.println("Octal String:" +Integer.toOctalString(s1.getState()));
        }
}

class HexaObserver extends Observer
        {
        public HexaObserver(Subject s)
        {
        this.s1=s;
        this.s1.attach(this);
        }
        public void update()
        {
        System.out.println("Heaxdeciamal String:" +Integer.toHexString(s1.getState()));
        }
}

public class Main
{
        public static void main(String [] args)
        {
        Subject s1=new Subject();
        new BinaryObserver(s1);
        new OctalObserver(s1);
        new HexaObserver(s1);
        System.out.println("First state Change:15");
        s1.setState(15);
        System.out.println("Second state Change:10");
        s1.setState(10);
        }
}
```

Q16) Write a Java Program to implement Abstract Factory Pattern for Shape interface

```
interface Shape
{
void draw();
}
class RoundedRectangle implements Shape
{
        public void draw()
        {
        System.out.println(" Inside RR");
        }
}
class RoundedSquare implements Shape
{
        public void draw()
        {
        System.out.println(" Inside RS");
        }
}
class Rectangle implements Shape
{
        public void draw()
        {
        System.out.println(" Inside Simple R");
        }
}

class Square implements Shape
{
        public void draw()
        {
        System.out.println(" Inside Simple Sq");
        }
}

abstract class AbstractFactory
{
        abstract Shape getShape( String st);
}

class ShapeFactory extends AbstractFactory
{
        public Shape getShape( String st)
        {
                if(st.equalsIgnoreCase("Rectangle"))
                { return new Rectangle();}
                else if(st.equalsIgnoreCase("Square"))
                { return new Square();}
        return null;
```

```
        }
}

class RoundedShapeFactory extends AbstractFactory
{
        public Shape getShape( String st)
        {
                if(st.equalsIgnoreCase("Rectangle"))
                { return new RoundedRectangle();}
                else if(st.equalsIgnoreCase("Square"))
                { return new RoundedSquare();}
        return null;
        }
}

class FactoryProducer
{
        public static AbstractFactory getFactory(boolean rounded)
        {
         if (rounded)
        { return new RoundedShapeFactory();}
        else
        { return new ShapeFactory();}
        }
}

public class Main
{
        public static void main(String[]args)
        {
        AbstractFactory shapeFactory=FactoryProducer.getFactory(false);
        Shape  shape1=shapeFactory.getShape("Rectangle");
        shape1.draw();

        Shape shape2=shapeFactory.getShape("SQuare");
        shape2.draw();

        AbstractFactory shapeFactory1=FactoryProducer.getFactory(true);
        Shape shape3=shapeFactory1.getShape("REctangle");
        shape3.draw();

        Shape shape4=shapeFactory1.getShape("square");
        shape4.draw();
        }

}
```