## Slip 1: Java Program to Implement I/O Decorator for Uppercase to Lowercase Conversion

```
import java.io.*;

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 IOExample {
   public static void main(String[] args) throws IOException {
      InputStream in = null;
      try {
         in = new LowerCaseInputStream(new BufferedInputStream(new
FileInputStream("test.txt")));
         int c;
         while((c = in.read()) >= 0) {
            System.out.print((char)c);
         }
      } finally {
         if (in != null) in.close();
      }
   }
}
```

## Slip 2: Java Program to Implement Singleton Pattern for Multithreading

```
class Singleton {
   private static Singleton instance;

   private Singleton() {}

   public static synchronized Singleton getInstance() {
      if (instance == null) {
         instance = new Singleton();
      }
      return instance;
   }

   public void showMessage() {
      System.out.println("Hello, Singleton Pattern!");
   }
}

public class SingletonPattern {
   public static void main(String[] args) {
      Singleton obj = Singleton.getInstance();
      obj.showMessage();
   }
}
```

## Slip 3: Java Program for Weather Station using java.util.Observable

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

class WeatherData extends Observable {
   private float temperature;
   private float humidity;
   private float pressure;

   public void measurementsChanged() {
      setChanged();
      notifyObservers();
   }

   public void setMeasurement(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; }
}

class CurrentConditionsDisplay implements Observer {
   private float temperature;
   private float humidity;

   public CurrentConditionsDisplay(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");
   }
}

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

      weatherData.setMeasurement(80, 65, 30.4f);
      weatherData.setMeasurement(82, 70, 29.2f);
   }
}
```

## Slip 4: Java Program to Implement Factory Method for Pizza Store

```
abstract class Pizza {
   String name;

   void prepare() { System.out.println("Preparing " + name); }
   void bake() { System.out.println("Baking " + name); }
   void cut() { System.out.println("Cutting " + name); }
   void box() { System.out.println("Boxing " + name); }
}

class NYStyleCheesePizza extends Pizza {
   NYStyleCheesePizza() { name = "NY Style Cheese Pizza"; }
}

class ChicagoStyleCheesePizza extends Pizza {
   ChicagoStyleCheesePizza() { name = "Chicago Style Cheese Pizza"; }
}

abstract class PizzaStore {
   abstract Pizza createPizza(String type);

   public Pizza orderPizza(String type) {
      Pizza pizza = createPizza(type);
      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
   }
}

class NYPizzaStore extends PizzaStore {
   Pizza createPizza(String type) {
      if (type.equals("cheese")) {
         return new NYStyleCheesePizza();
      } else return null;
   }
}

class ChicagoPizzaStore extends PizzaStore {
   Pizza createPizza(String type) {
      if (type.equals("cheese")) {
         return new ChicagoStyleCheesePizza();
      } else return null;
   }
}

public class PizzaTestDrive {
   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.name + "\n");

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

## Slip 5: Java Program to Implement Adapter Pattern for Enumeration Iterator

```
import java.util.*;

class EnumerationIterator implements Iterator<Object> {
   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 EnumerationAdapterTest {
   public static void main(String[] args) {
      Vector<String> vector = new Vector<>();
      vector.add("Item1");
      vector.add("Item2");
      Enumeration<String> enumeration = vector.elements();
      Iterator<Object> iterator = new EnumerationIterator(enumeration);

      while (iterator.hasNext()) {
         System.out.println(iterator.next());
      }
   }
}
```

## Slip 6: Java Program to Implement Command Pattern for Remote Control

```
interface Command {
    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 RemoteControl {
    Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}

public class RemoteControlTest {
    public static void main(String[] args) {
        RemoteControl remote = new RemoteControl();
        Light light = new Light();
        LightOnCommand lightOn = new LightOnCommand(light);

        remote.setCommand(lightOn);
        remote.pressButton();
    }
}
```

## Slip 7: Java Program to Implement Undo Command for Ceiling Fan

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

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 ceilingFan;

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

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

   public void undo() {
      ceilingFan.off();
   }
}

class RemoteControl {
   Command command;

   public void setCommand(Command command) {
      this.command = command;
   }

   public void pressButton() {
      command.execute();
   }

   public void pressUndo() {
      command.undo();
   }
}

public class CeilingFanTest {
   public static void main(String[] args) {
      RemoteControl remote = new RemoteControl();
      CeilingFan ceilingFan = new CeilingFan();
      CeilingFanOnCommand ceilingFanOn = new CeilingFanOnCommand(ceilingFan);

      remote.setCommand(ceilingFanOn);
      remote.pressButton();
      remote.pressUndo();
   }
}
```

## Slip 8: Java Program to Implement State Pattern for Gumball Machine

```
interface State {
   void insertQuarter();
   void ejectQuarter();
   void turnCrank();
   void dispense();
}

class GumballMachine {
   State soldOutState;
   State noQuarterState;
   State hasQuarterState;
   State soldState;

   State state = soldOutState;
   int count = 0;

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

      if (count > 0) {
         state = noQuarterState;
      }
   }

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

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

   public int getCount() { return count; }
}

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.hasQuarterState);
   }

   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");
    }
}

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.noQuarterState);
    }

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

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

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.noQuarterState);
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.soldOutState);
        }
```

```
   }
}

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 class GumballMachineTestDrive {
   public static void main(String[] args) {
      GumballMachine gumballMachine = new GumballMachine(5);

      System.out.println(gumballMachine);

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

      System.out.println(gumballMachine);

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

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

## Slip 9: Java Program to Design HR Application using Spring Framework

(As this requires the Spring Framework, a simplified version without Spring is provided as a placeholder)

```
class Employee {
   private String name;
   private String position;

   public Employee(String name, String position) {
      this.name = name;
      this.position = position;
   }

   public String getName() {
      return name;
   }

   public String getPosition() {
      return position;
   }

   public void setPosition(String position) {
      this.position = position;
   }
}

public class HRApplication {
   public static void main(String[] args) {
      Employee employee = new Employee("John Doe", "Developer");
      System.out.println("Employee: " + employee.getName() + ", Position: " +
employee.getPosition());
   }
}
```

## Slip 10: Java Program to Implement Strategy Pattern for Duck Behavior

```
interface FlyBehavior {
   void fly();
}

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

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

interface QuackBehavior {
   void quack();
}

class Quack implements QuackBehavior {
   public void quack() {
      System.out.println("Quack");
   }
}

class MuteQuack implements QuackBehavior {
   public void quack() {
      System.out.println("<< Silence >>");
   }
}

abstract class Duck {
   FlyBehavior flyBehavior;
   QuackBehavior quackBehavior;

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

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

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

   public void setFlyBehavior(FlyBehavior fb) {
      flyBehavior = fb;
   }

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

class MallardDuck extends Duck {
   public MallardDuck() {
      quackBehavior = new Quack();
      flyBehavior = new FlyWithWings();
   }

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

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

## Slip 11: Java Program to Implement Adapter Pattern for Heart Model to Beat Model

```
class HeartModel {
   public void startBeat() {
      System.out.println("Heart is beating...");
   }

   public void stopBeat() {
      System.out.println("Heart has stopped beating...");
   }
}

interface BeatModelInterface {
   void start();
   void stop();
}

class HeartAdapter implements BeatModelInterface {
   private HeartModel heart;

   public HeartAdapter(HeartModel heart) {
      this.heart = heart;
   }

   public void start() {
      heart.startBeat();
   }

   public void stop() {
      heart.stopBeat();
   }
}

public class HeartModelTest {
   public static void main(String[] args) {
      HeartModel heart = new HeartModel();
      BeatModelInterface beatModel = new HeartAdapter(heart);

      beatModel.start();
      beatModel.stop();
   }
}
```

## Slip 12: Java Program to Implement Decorator Pattern for Car (Sports Car and Luxury Car)

```
interface Car {
   void assemble();
}

class BasicCar implements Car {
   public void assemble() {
      System.out.println("Basic Car.");
   }
}

class CarDecorator implements Car {
   protected Car car;

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

   public void assemble() {
      this.car.assemble();
   }
}

class SportsCar extends CarDecorator {
   public SportsCar(Car c) {
      super(c);
   }

   public void assemble() {
      super.assemble();
      System.out.print(" Adding features of Sports Car.");
   }
}

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

   public void assemble() {
      super.assemble();
      System.out.print(" Adding features of Luxury Car.");
   }
}

public class DecoratorPatternTest {
   public static void main(String[] args) {
      Car sportsCar = new SportsCar(new BasicCar());
      sportsCar.assemble();
      System.out.println("\n");

      Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));
      sportsLuxuryCar.assemble();
   }
}
```

## Slip 13: Java Program to Implement Adapter Pattern for Mobile Charger

```
class Volt {
   private int volts;

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

   public int getVolts() {
      return volts;
   }
}

class Socket {
   public Volt getVolt() {
      return new Volt(120);
   }
}

interface MobileAdapter {
   Volt get3Volt();
   Volt get12Volt();
   Volt get120Volt();
}

class SocketAdapter implements MobileAdapter {
   private Socket socket;

   public SocketAdapter(Socket socket) {
      this.socket = socket;
   }

   public Volt get3Volt() {
      return convertVolt(socket.getVolt(), 40);
   }

   public Volt get12Volt() {
      return convertVolt(socket.getVolt(), 10);
   }

   public Volt get120Volt() {
      return socket.getVolt();
   }

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

public class AdapterPatternTest {
   public static void main(String[] args) {
      Socket socket = new Socket();
      MobileAdapter adapter = new SocketAdapter(socket);

      System.out.println("3 volts : " + adapter.get3Volt().getVolts());
      System.out.println("12 volts : " + adapter.get12Volt().getVolts());
      System.out.println("120 volts : " + adapter.get120Volt().getVolts());
   }
```

**Slip 14: Java Program to Implement Command Pattern for Command Interface (Light and Garage Door)**

```
interface Command {
   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 GarageDoor {
   public void up() {
      System.out.println("Garage Door is Open");
   }

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

class GarageDoorUpCommand implements Command {
   GarageDoor garageDoor;

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

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

class RemoteControl {
   Command command;

   public void setCommand(Command command) {
      this.command = command;
   }

   public void pressButton() {
      command.execute();
   }
}

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

      Light light = new Light();
      GarageDoor garageDoor = new GarageDoor();

      LightOnCommand lightOn = new LightOnCommand(light);
      GarageDoorUpCommand garageUp = new
GarageDoorUpCommand(garageDoor);

      remote.setCommand(lightOn);
      remote.pressButton();

      remote.setCommand(garageUp);
      remote.pressButton();
   }
}
```

## Slip 15: Java Program to Implement Facade Design Pattern for Home Theater

```
class Amplifier {
  public void on() {
    System.out.println("Amplifier is on");
  }

  public void off() {
    System.out.println("Amplifier is off");
  }
}
class DvdPlayer {
  public void on() {
    System.out.println("DVD Player is on");
  }
  public void play(String movie) {
    System.out.println("Playing movie: " + movie);
  }
  public void off() {
    System.out.println("DVD Player is off");
  }
}
class Projector {
  public void on() {
    System.out.println("Projector is on");
  }
  public void off() {
    System.out.println("Projector is off");
  }
}

class HomeTheaterFacade {
  Amplifier amp;
  DvdPlayer dvd;
  Projector projector;
  public HomeTheaterFacade(Amplifier amp, DvdPlayer dvd, Projector projector) {
    this.amp = amp;
    this.dvd = dvd;
    this.projector = projector;
  }
  public void watchMovie(String movie) {
    System.out.println("Get ready to watch a movie...");
    amp.on();
    projector.on();
    dvd.on();
    dvd.play(movie);
  }
  public void endMovie() {
    System.out.println("Shutting movie theater down...");
    dvd.off();
    projector.off();
    amp.off();
  }
}
public class HomeTheaterTest {
  public static void main(String[] args) {
    Amplifier amp = new Amplifier();
    DvdPlayer dvd = new DvdPlayer();
    Projector projector = new Projector();
    HomeTheaterFacade homeTheater = new HomeTheaterFacade(amp, dvd, projector);
    homeTheater.watchMovie("Inception");
    homeTheater.endMovie();
  }
}
```

## Slip 16: Observer Design Pattern for Number Conversion (Decimal to Hexadecimal, Octal, Binary):

```
import java.util.ArrayList;
import java.util.List;
// Observer interface
interface Observer {
  void update(int number);
}
// Concrete Observer for Binary conversion
class BinaryObserver implements Observer {
  @Override
  public void update(int number) {
    System.out.println("Binary: " + Integer.toBinaryString(number));
  }
}
// Concrete Observer for Octal conversion
class OctalObserver implements Observer {
  @Override
  public void update(int number) {
    System.out.println("Octal: " + Integer.toOctalString(number));
  }
}
// Concrete Observer for Hexadecimal conversion
class HexObserver implements Observer {
  @Override
  public void update(int number) {
    System.out.println("Hexadecimal: " + Integer.toHexString(number).toUpperCase());
  }
}
// Subject class that notifies observers of number changes
class NumberSubject {
  private List<Observer> observers = new ArrayList<>();
  private int number;
  public void setNumber(int number) {
    this.number = number;
    notifyAllObservers();
  }
  public void addObserver(Observer observer) {
    observers.add(observer);
  }
  public void notifyAllObservers() {
    for (Observer observer : observers) {
      observer.update(number);
    }
  }
}
// Main class to demonstrate Observer Pattern
public class ObserverPatternNumberConversion {
  public static void main(String[] args) {
    NumberSubject numberSubject = new NumberSubject();
    // Attach observers
    numberSubject.addObserver(new BinaryObserver());
    numberSubject.addObserver(new OctalObserver());
    numberSubject.addObserver(new HexObserver());
    System.out.println("Enter a number in Decimal form:");
    numberSubject.setNumber(10);  // Example number: 10

    System.out.println("\nUpdating number to 255:");
    numberSubject.setNumber(255);
  }}
```

## Slip 17: Java Program to implement Abstract Factory Pattern for Shape interface.

```
//Step 1: Define the Shape interface
public interface Shape {
   void draw();
}

//Step 2: Implement concrete classes for Shape
public class Circle implements Shape {
   @Override
   public void draw() {
      System.out.println("Drawing a Circle");
   }
}
public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Drawing a Rectangle");
   }
}
public class Square implements Shape {
   @Override
   public void draw() {
      System.out.println("Drawing a Square");
   }
}

//Step 3: Define the ShapeFactory interface
public interface ShapeFactory {
   Shape createShape();
}

//Step 4: Implement concrete factories for each shape
public class CircleFactory implements ShapeFactory {
   @Override
   public Shape createShape() {
      return new Circle();
   }
}
public class RectangleFactory implements ShapeFactory {
   @Override
   public Shape createShape() {
      return new Rectangle();
   }
}
public class SquareFactory implements ShapeFactory {
   @Override
   public Shape createShape() {
      return new Square();
   }
}
//Step 5: Create the FactoryProducer to get the factories
public class FactoryProducer {
    public static ShapeFactory getFactory(String shapeType) {
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new CircleFactory();
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new RectangleFactory();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new SquareFactory();
        }
        return null;
    }
}

//Step 6: Client code to test the Abstract Factory Pattern
public class Main {
    public static void main(String[] args) {
        ShapeFactory circleFactory = FactoryProducer.getFactory("CIRCLE");
        Shape circle = circleFactory.createShape();
        circle.draw();

        ShapeFactory rectangleFactory =
FactoryProducer.getFactory("RECTANGLE");
        Shape rectangle = rectangleFactory.createShape();
        rectangle.draw();

        ShapeFactory squareFactory = FactoryProducer.getFactory("SQUARE");
        Shape square = squareFactory.createShape();
        square.draw();
    }
}
```

**Slip 18: Java Program to implement built-in support (java.util.Observable) Weather station with members temperature, humidity, pressure and methods mesurmentsChanged(), setMesurment(), getTemperature(), getHumidity(), getPressure()**

**//Step 1: Create the WeatherStation class**

```
import java.util.Observable;

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

   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;
   }
}
```

**//Step 2: Create the WeatherDisplay class**

```
//This class will implement the Observer interface to display the weather data.

import java.util.Observable;
import java.util.Observer;

public class WeatherDisplay implements Observer {
   @Override
   public void update(Observable o, Object arg) {
      if (o instanceof WeatherStation) {
         WeatherStation station = (WeatherStation) o;
         System.out.println("Current Weather Measurements:");
```

```
            System.out.println("Temperature: " + station.getTemperature());
            System.out.println("Humidity: " + station.getHumidity());
            System.out.println("Pressure: " + station.getPressure());
            System.out.println();
        }
    }
}
```

**//Step 3: Create the Main class**

//This class will tie everything together and demonstrate the functionality.

```
public class Main {
    public static void main(String[] args) {
        WeatherStation weatherStation = new WeatherStation();
        WeatherDisplay display = new WeatherDisplay();

        // Register the display as an observer
        weatherStation.addObserver(display);

        // Simulating new weather measurements
        weatherStation.setMeasurements(25.0f, 65.0f, 1013.0f);
        weatherStation.setMeasurements(22.0f, 70.0f, 1010.0f);
        weatherStation.setMeasurements(28.0f, 60.0f, 1015.0f);
    }
}
```

**Slip 19: 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: Define the Pizza abstract class**

```
public abstract class Pizza {
   String name;
   String dough;
   String sauce;

   public void prepare() {
      System.out.println("Preparing " + name);
      System.out.println("Tossing dough... " + dough);
      System.out.println("Adding sauce... " + sauce);
   }
   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 getName() {
      return name;
   }
}
```

**//Step 2: Create concrete pizza classes**

***//NYStyleCheesePizza***

```
public class NYStyleCheesePizza extends Pizza {
   public NYStyleCheesePizza() {
      name = "NY Style Cheese Pizza";
      dough = "Thin Crust Dough";
      sauce = "Marinara Sauce";
   }
}
```

***//ChicagoStyleCheesePizza***

```
public class ChicagoStyleCheesePizza extends Pizza {
   public ChicagoStyleCheesePizza() {
      name = "Chicago Style Cheese Pizza";
      dough = "Deep Dish Dough";
      sauce = "Plum Tomato Sauce";
   }

   @Override
   public void cut() {
      System.out.println("Cutting " + name + " into square slices");
   }
}
```

```
//Step 3: Define the PizzaStore abstract class
public abstract class PizzaStore {
  public Pizza orderPizza(String type) {
    Pizza pizza = createPizza(type);

    if (pizza != null) {
      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
    }
    return pizza;
  }
  protected abstract Pizza createPizza(String type);
}
//Step 4: Implement concrete pizza store classes
//NYPizzaStore
public class NYPizzaStore extends PizzaStore {
  @Override
  protected Pizza createPizza(String type) {
    if (type.equalsIgnoreCase("cheese")) {
      return new NYStyleCheesePizza();
    }
    return null; // Add more types as needed
  }
}
//ChicagoPizzaStore
public class ChicagoPizzaStore extends PizzaStore {
  @Override
  protected Pizza createPizza(String type) {
    if (type.equalsIgnoreCase("cheese")) {
      return new ChicagoStyleCheesePizza();
    }
    return null; // Add more types as needed
  }
}
//Step 5: Create the Main class to test the implementation
public class Main {
  public static void main(String[] args) {
    PizzaStore nyStore = new NYPizzaStore();
    PizzaStore chicagoStore = new ChicagoPizzaStore();

    System.out.println("Ordering a Cheese Pizza from NY Store:");
    nyStore.orderPizza("cheese");

    System.out.println("\nOrdering a Cheese Pizza from Chicago Store:");
    chicagoStore.orderPizza("cheese");
  }
}
```

**Slip 20: Java Program to implement I/O Decorator for converting uppercase letters to lower case letters.**

```
import java.io.*;

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 IOExample {
   public static void main(String[] args) throws IOException {
      InputStream in = null;
      try {
         in = new LowerCaseInputStream(new BufferedInputStream(new
FileInputStream("test.txt")));
         int c;
         while((c = in.read()) >= 0) {
            System.out.print((char)c);
         }
      } finally {
         if (in != null) in.close();
      }
   }
}
```