```
//Slip 1
import java.io.*;
import java.util.Scanner;

public class LowerCaseDecoratorExample {
   public static void main(String[] args) throws IOException {
      // Take input from user
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter text: ");
      String input = scanner.nextLine();

      // Convert the input to a stream and wrap it with a BufferedReader
      InputStream inputStream = new ByteArrayInputStream(input.getBytes());
      BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));

      // Output stream with OutputStreamWriter to write to the console
      OutputStreamWriter writer = new OutputStreamWriter(System.out);

      int ch;
      // Read each character, convert to lowercase and write to output
      while ((ch = reader.read()) != -1) {
         writer.write(Character.toLowerCase(ch));
      }

      // Flush the writer to ensure output is displayed
      writer.flush();

      // Close resources
      reader.close();
      writer.close();
   }
}
```

```
//Slip2

class Singleton {
    private static Singleton instance;

    // Private constructor to prevent instantiation
    private Singleton() {}

    // Synchronized method to ensure thread safety
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    // Example method
    public void showMessage() {
        System.out.println("Singleton instance accessed by " + Thread.currentThread().getName());
    }
}

public class SingletonDemo {
    public static void main(String[] args) {
        // Creating two threads that access the Singleton instance
        Thread t1 = new Thread(() -> {
            Singleton singleton = Singleton.getInstance();
            singleton.showMessage();
        });

        Thread t2 = new Thread(() -> {
            Singleton singleton = Singleton.getInstance();
```

```
            singleton.showMessage();
        });

        t1.start();
        t2.start();
    }
}
```

```
//Slip 3

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

// WeatherStation class that extends Observable
class WeatherStation extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    // Method to set measurements and notify observers
    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    // Method to notify observers of changes
    public void measurementsChanged() {
```

```
        setChanged();  // Marks the observable as having been changed
        notifyObservers();  // Notify all observers
    }

    // Getter methods for temperature, humidity, and pressure
    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}

// Display class that implements Observer
class Display implements Observer {
    @Override
    public void update(Observable o, Object arg) {
        if (o instanceof WeatherStation) {
            WeatherStation ws = (WeatherStation) o;
            System.out.println("Updated weather data: ");
            System.out.println("Temperature: " + ws.getTemperature());
            System.out.println("Humidity: " + ws.getHumidity());
            System.out.println("Pressure: " + ws.getPressure());
        }
    }
}
```

```
// Main class to test the WeatherStation
public class WeatherStationDemo {
    public static void main(String[] args) {
        // Create weather station and display objects
        WeatherStation weatherStation = new WeatherStation();
        Display display = new Display();

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

        // Simulate new weather measurements
        weatherStation.setMeasurements(25.5f, 65.0f, 1013.1f);
        weatherStation.setMeasurements(30.0f, 70.0f, 1012.5f);
    }
}
```

```
//Slip 4

//Base Pizza class
abstract class Pizza {
 public void prepare() {
    System.out.println("Preparing " + this.getClass().getSimpleName());
 }

 public void bake() {
    System.out.println("Baking " + this.getClass().getSimpleName());
 }

 public void cut() {
    System.out.println("Cutting " + this.getClass().getSimpleName());
```

```
 }

 public void box() {
    System.out.println("Boxing " + this.getClass().getSimpleName());
 }
}

//NyStyleCheesePizza class
class NyStyleCheesePizza extends Pizza {}

//ChicagoStyleCheesePizza class
class ChicagoStyleCheesePizza extends Pizza {}

//PizzaStore class with the factory method
abstract class PizzaStore {
 // Factory Method: to be implemented by subclasses
 public abstract Pizza createPizza(String type);

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

//NyPizzaStore class implementing factory method
class NyPizzaStore extends PizzaStore {
 public Pizza createPizza(String type) {
    if (type.equals("cheese")) {
```

```
        return new NyStyleCheesePizza();
    }
    return null;
 }
}

//ChicagoPizzaStore class implementing factory method
class ChicagoPizzaStore extends PizzaStore {
 public Pizza createPizza(String type) {
    if (type.equals("cheese")) {
        return new ChicagoStyleCheesePizza();
    }
    return null;
 }
}

//Main class to test the program
public class PizzaStoreDemo {
 public static void main(String[] args) {
    PizzaStore nyStore = new NyPizzaStore();
    PizzaStore chicagoStore = new ChicagoPizzaStore();

    // Order pizzas from different stores
    nyStore.orderPizza("cheese");
    chicagoStore.orderPizza("cheese");
 }
}



//Slip 5
import java.util.Enumeration;
```

```
import java.util.Iterator;
import java.util.Scanner;
import java.util.Vector;

// Adapter class to convert Enumeration to Iterator
class EnumerationIteratorAdapter implements Iterator<Object> {
    private Enumeration<?> enumeration;

    public EnumerationIteratorAdapter(Enumeration<?> enumeration) {
        this.enumeration = enumeration;
    }

    // Check if more elements are available
    @Override
    public boolean hasNext() {
        return enumeration.hasMoreElements();
    }

    // Get the next element
    @Override
    public Object next() {
        return enumeration.nextElement();
    }
}

// Main class to test the adapter
public class AdapterPatternDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Vector<String> vector = new Vector<>();

        // Taking input from user
```

```
        System.out.println("Enter elements for the vector (type 'done' to finish):");
        while (true) {
            String input = scanner.nextLine();
            if (input.equalsIgnoreCase("done")) {
                break;
            }
            vector.add(input);  // Add input to vector
        }

        Enumeration<String> enumeration = vector.elements();

        // Use the adapter to treat Enumeration as Iterator
        Iterator<Object> iterator = new EnumerationIteratorAdapter(enumeration);

        // Iterate using Iterator methods
        System.out.println("Iterating through the vector elements:");
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        scanner.close();
    }
}



//Slip 6

//Command Interface
interface Command {
 void execute();
}
```

```
//Light class with on and off actions
class Light {
 public void on() {
    System.out.println("Light is ON");
 }

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

//Concrete Command to turn on the light
class LightOnCommand implements Command {
 private Light light;

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

 @Override
 public void execute() {
    light.on();
 }
}

//Concrete Command to turn off the light
class LightOffCommand implements Command {
 private Light light;

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

```
 }

 @Override
 public void execute() {
    light.off();
 }
}

//Simple Remote Control that triggers commands
class RemoteControl {
 private Command command;

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

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

//Main class to test the Remote Control with Commands
public class CommandPatternDem {
 public static void main(String[] args) {
    // Create a Light object (Receiver)
    Light light = new Light();

    // Create commands for turning the light on and off
    Command lightOn = new LightOnCommand(light);
    Command lightOff = new LightOffCommand(light);

    // Create a RemoteControl (Invoker)
```

```
    RemoteControl remote = new RemoteControl();

    // Test turning the light on
    remote.setCommand(lightOn);
    remote.pressButton();

    // Test turning the light off
    remote.setCommand(lightOff);
    remote.pressButton();
 }
}
```

```
//slip8

//State interface
interface State {
 void insertCoin();
 void ejectCoin();
 void turnCrank();
 void dispense();
}

//Gumball Machine class
class GumballMachine {
 State noCoinState;
 State hasCoinState;
 State soldOutState;
 State currentState;

 public GumballMachine(int numberOfGumballs) {
```

```
    noCoinState = new NoCoinState(this);
    hasCoinState = new HasCoinState(this);
    soldOutState = new SoldOutState(this);

    currentState = (numberOfGumballs > 0) ? noCoinState : soldOutState;
}

public void setCurrentState(State state) {
    currentState = state;
}

public void insertCoin() {
    currentState.insertCoin();
}

public void ejectCoin() {
    currentState.ejectCoin();
}

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

public void releaseGumball() {
    System.out.println("A gumball comes rolling out the slot.");
}

public State getNoCoinState() {
    return noCoinState;
}
```

```
 public State getHasCoinState() {
    return hasCoinState;
 }

 public State getSoldOutState() {
    return soldOutState;
 }
}

//State Implementations
class NoCoinState implements State {
 GumballMachine gumballMachine;

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

 public void insertCoin() {
    System.out.println("Coin inserted.");
    gumballMachine.setCurrentState(gumballMachine.getHasCoinState());
 }

 public void ejectCoin() {
    System.out.println("No coin to eject.");
 }

 public void turnCrank() {
    System.out.println("You need to insert a coin first.");
 }

 public void dispense() {
    System.out.println("Insert coin first.");
```

```
 }
}

class HasCoinState implements State {
 GumballMachine gumballMachine;

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

 public void insertCoin() {
    System.out.println("Coin already inserted.");
 }

 public void ejectCoin() {
    System.out.println("Coin returned.");
    gumballMachine.setCurrentState(gumballMachine.getNoCoinState());
 }

 public void turnCrank() {
    System.out.println("Crank turned.");
    gumballMachine.releaseGumball();
    gumballMachine.setCurrentState(gumballMachine.getNoCoinState());
 }

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

class SoldOutState implements State {
 GumballMachine gumballMachine;
```

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

 public void insertCoin() {
    System.out.println("Machine is sold out.");
 }

 public void ejectCoin() {
    System.out.println("No coin to eject.");
 }

 public void turnCrank() {
    System.out.println("Machine is sold out.");
 }

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

public class GumballMachineDemo {
        public static void main(String[] args) {
      GumballMachine gumballMachine = new GumballMachine(1); // 1 gumball

      gumballMachine.insertCoin(); // Insert coin
      gumballMachine.turnCrank();   // Turn crank

      gumballMachine.insertCoin(); // Insert coin again
      gumballMachine.ejectCoin();   // Eject coin
      gumballMachine.turnCrank();   // Try to turn crank without inserting coin
```

```
    }
}



//Slip 10

//FlyingBehavior Interface
interface FlyingBehavior {
 void fly();
}

//QuackingBehavior Interface
interface QuackingBehavior {
 void quack();
}

//Different Flying Behaviors
class FlyWithWings implements FlyingBehavior {
 public void fly() {
    System.out.println("I can fly!");
 }
}

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

//Different Quacking Behaviors
class Quack implements QuackingBehavior {
```

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

class Squeak implements QuackingBehavior {
 public void quack() {
    System.out.println("Squeak!");
 }
}

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

//Duck Class
abstract class Duck {
 FlyingBehavior flyingBehavior;
 QuackingBehavior quackingBehavior;

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

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

 public void swim() {
    System.out.println("All ducks float!");
```

```
 }

 // Setters for behaviors
 public void setFlyingBehavior(FlyingBehavior fb) {
    flyingBehavior = fb;
 }

 public void setQuackingBehavior(QuackingBehavior qb) {
    quackingBehavior = qb;
 }
}

//MallardDuck Class
class MallardDuck extends Duck {
 public MallardDuck() {
    flyingBehavior = new FlyWithWings();
    quackingBehavior = new Quack();
 }
}

//RubberDuck Class
class RubberDuck extends Duck {
 public RubberDuck() {
    flyingBehavior = new FlyNoWay();
    quackingBehavior = new Squeak();
 }
}

//Main Class to Test Duck Behaviors
public class DuckSimulator {
 public static void main(String[] args) {
    Duck mallard = new MallardDuck();
```

```
    Duck rubberDuck = new RubberDuck();

    System.out.println("Mallard Duck:");
    mallard.performFly();       // Output: I can fly!
    mallard.performQuack();     // Output: Quack!

    System.out.println("\nRubber Duck:");
    rubberDuck.performFly();    // Output: I can't fly.
    rubberDuck.performQuack();  // Output: Squeak!
 }
}


//slip 11

//HeartModel class simulating the heart's behavior
class HeartModel {
 public void pump() {
    System.out.println("Heart is pumping!");
 }

 public void setHeartRate(int rate) {
    System.out.println("Heart rate set to: " + rate);
 }
}

//BeatModel interface defining the behavior for a beat
interface BeatModel {
 void start();
 void stop();
 void setHeartRate(int rate);
}
```

```
//Adapter class that adapts HeartModel to BeatModel
class HeartModelAdapter implements BeatModel {
 private HeartModel heartModel;

 public HeartModelAdapter(HeartModel heartModel) {
    this.heartModel = heartModel;
 }

 @Override
 public void start() {
    heartModel.pump(); // Call the heart's pump method
 }

 @Override
 public void stop() {
    System.out.println("Heart stopped beating.");
 }

 @Override
 public void setHeartRate(int rate) {
    heartModel.setHeartRate(rate);
 }
}

public class AdapterPatternHeartModel {
        public static void main(String[] args) {
            HeartModel heartModel = new HeartModel();
            BeatModel heartAdapter = new HeartModelAdapter(heartModel);

            // Using the adapter to interact with the heart model
            heartAdapter.start();           // Start beating
```

```
        heartAdapter.setHeartRate(75);    // Set heart rate to 75
        heartAdapter.stop();              // Stop beating
    }
}


//Slip 12

//Car Interface
interface Car {
 String assemble();
}

//BasicCar Class
class BasicCar implements Car {
 public String assemble() {
    return "Basic Car";
 }
}

//Abstract CarDecorator Class
abstract class CarDecorator implements Car {
 protected Car car;

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

 public String assemble() {
    return car.assemble();
 }
}
```

```
//SportsCar Decorator
class SportsCar extends CarDecorator {
 public SportsCar(Car car) {
    super(car);
 }

 public String assemble() {
    return super.assemble() + ", with Sports Features";
 }
}

//LuxuryCar Decorator
class LuxuryCar extends CarDecorator {
 public LuxuryCar(Car car) {
    super(car);
 }

 public String assemble() {
    return super.assemble() + ", with Luxury Features";
 }
}

//Main Class to Test Decorator Pattern
public class DecoratorPatternDemo {
 public static void main(String[] args) {
    Car basicCar = new BasicCar();
    System.out.println(basicCar.assemble()); // Output: Basic Car

    Car sportsCar = new SportsCar(basicCar);
    System.out.println(sportsCar.assemble()); // Output: Basic Car, with Sports Features
```

```
    Car luxuryCar = new LuxuryCar(basicCar);
    System.out.println(luxuryCar.assemble()); // Output: Basic Car, with Luxury Features

    Car sportsLuxuryCar = new LuxuryCar(new SportsCar(basicCar));
    System.out.println(sportsLuxuryCar.assemble()); // Output: Basic Car, with Sports Features, with Luxury Features
 }
}
```

```
//Slip 13
package p;
//Volt class to measure voltage
class Volt {
 private int volts;

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

 public int getVolts() {
    return volts;
 }
}

//Socket class that produces a constant voltage of 120V
class Socket {
 public Volt getVolt() {
    return new Volt(120);
 }
}
```

```
//Adapter class to convert voltage levels
class VoltageAdapter extends Socket {
 private Volt volt;

 public VoltageAdapter(Volt volt) {
    this.volt = volt;
 }

 // Method to get the desired voltage
 public Volt getAdaptedVolt(int desiredVolts) {
    switch (desiredVolts) {
       case 3:
          return new Volt(3);
       case 12:
          return new Volt(12);
       default:
          return volt; // Return default 120V
    }
 }
}
public class AdapterPatternMobileCharger {
            public static void main(String[] args) {
                  Socket socket = new Socket();
                  VoltageAdapter adapter = new VoltageAdapter(socket.getVolt());

                  // Getting different voltage outputs
                  System.out.println("Voltage from Socket: " + adapter.getAdaptedVolt(120).getVolts() +
"V"); // Default 120V
                  System.out.println("Voltage from Adapter (3V): " +
adapter.getAdaptedVolt(3).getVolts() + "V");
                  System.out.println("Voltage from Adapter (12V): " +
adapter.getAdaptedVolt(12).getVolts() + "V");
            }
```

```
}



//Slip 14

//Command Interface
interface Command {
 void execute();
}

//Light Class
class Light {
 public void on() {
    System.out.println("The light is ON");
 }

 public void off() {
    System.out.println("The light is OFF");
 }
}

//GarageDoor Class
class GarageDoor {
 public void up() {
    System.out.println("The garage door is UP");
 }

 public void down() {
    System.out.println("The garage door is DOWN");
 }
}
```

```
//Stereo Class
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("CD is set in the stereo");
 }
}

//Concrete Command Classes
class LightOnCommand implements Command {
 private Light light;

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

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

class LightOffCommand implements Command {
 private Light light;
```

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

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

class GarageDoorUpCommand implements Command {
 private GarageDoor garageDoor;

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

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

class StereoOnWithCDCommand implements Command {
 private Stereo stereo;

 public StereoOnWithCDCommand(Stereo stereo) {
    this.stereo = stereo;
 }

 public void execute() {
    stereo.on();
    stereo.setCD();
 }
```

```
}

//Main Class to Test Command Pattern
public class CommandPatternDemo14 {
 public static void main(String[] args) {
    // Create the objects
    Light light = new Light();
    GarageDoor garageDoor = new GarageDoor();
    Stereo stereo = new Stereo();

    // Create command objects
    Command lightOn = new LightOnCommand(light);
    Command lightOff = new LightOffCommand(light);
    Command garageDoorUp = new GarageDoorUpCommand(garageDoor);
    Command stereoOnWithCD = new StereoOnWithCDCommand(stereo);

    // Execute the commands
    lightOn.execute();            // Output: The light is ON
    lightOff.execute();           // Output: The light is OFF
    garageDoorUp.execute();       // Output: The garage door is UP
    stereoOnWithCD.execute();     // Output: Stereo is ON, CD is set in the stereo
 }
}




//Slip 15

//Subsystems
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");
 }
}

//Facade
```

```
class HomeTheaterFacade {
 private Amplifier amp;
 private DVDPlayer dvd;
 private 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();
    dvd.on();
    projector.on();
    dvd.play(movie);
 }

 public void endMovie() {
    System.out.println("Shutting down the home theater...");
    projector.off();
    dvd.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
import java.util.ArrayList;
import java.util.List;

// Observer interface
interface Observer {
    void update(int number);
}

// Concrete Observers
class HexObserver implements Observer {
    public void update(int number) {
        System.out.println("Hexadecimal: " + Integer.toHexString(number).toUpperCase());
    }
}

class OctalObserver implements Observer {
    public void update(int number) {
        System.out.println("Octal: " + Integer.toOctalString(number));
    }
}
```

```
class BinaryObserver implements Observer {
    public void update(int number) {
        System.out.println("Binary: " + Integer.toBinaryString(number));
    }
}

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

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void setNumber(int number) {
        this.number = number;
        notifyObservers();
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(number);
        }
    }
}

public class NumberConversionTest {
            public static void main(String[] args) {
        NumberSubject numberSubject = new NumberSubject();
```

```
        numberSubject.addObserver(new HexObserver());
        numberSubject.addObserver(new OctalObserver());
        numberSubject.addObserver(new BinaryObserver());

        System.out.println("Setting number to 15:");
        numberSubject.setNumber(15);

        System.out.println("\nChanging number to 32:");
        numberSubject.setNumber(32);
    }
}



//Slip 17

//Shape interface
interface Shape {
 void draw();
}

//Concrete classes implementing Shape interface
class Circle implements Shape {
 public void draw() {
    System.out.println("Drawing Circle");
 }
}

class Rectangle implements Shape {
 public void draw() {
    System.out.println("Drawing Rectangle");
 }
```

```
}

//Abstract Factory interface
interface AbstractFactory {
 Shape getShape();
}

//Concrete Factory classes
class CircleFactory implements AbstractFactory {
 public Shape getShape() {
    return new Circle();
 }
}

class RectangleFactory implements AbstractFactory {
 public Shape getShape() {
    return new Rectangle();
 }
}

//Factory Creator
class FactoryCreator {
 public static AbstractFactory getFactory(String shapeType) {
    if (shapeType.equalsIgnoreCase("CIRCLE")) {
       return new CircleFactory();
    } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
       return new RectangleFactory();
    }
    return null;
 }
}
```

```
public class AbstractFactoryPatternDemo {
        public static void main(String[] args) {
            AbstractFactory shapeFactory = FactoryCreator.getFactory("CIRCLE");
            Shape shape1 = shapeFactory.getShape();
            shape1.draw();

            shapeFactory = FactoryCreator.getFactory("RECTANGLE");
            Shape shape2 = shapeFactory.getShape();
            shape2.draw();
          }
}
```

//Slip 18

Same (Weather station)

//Slip 19

Same(Pizza Store)

//Slip 20

Same(Uppercase to lowercase)