**Week 1**

**Design Pattern and Principles- Hands on (All problems finished)**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Logger.java:**

public class Logger{

    private static Logger instance;

    private Logger(){

        System.out.print("Logger initialized");}

    public static Logger getInstance(){

        if(instance==null){

            instance = new Logger(); }

        return instance;}

    public void log(String message){

        System.out.println("Logging"+ message);}}

**Main.java**

public class Main{

    public static void main(String[] args){

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        logger1.log("Start");

        logger2.log("stop");

        if(logger1==logger2){

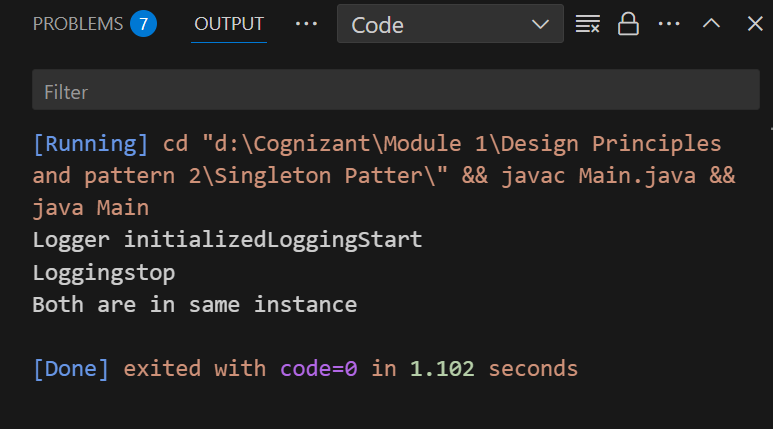
            System.out.println("Both are in same instance");

        }

        else{

            System.out.println("Different instances");}

**Output:**



**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Document.java:**

public interface Document{

    void open();

}

**DocumentFactory.java**

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

**ExcelDocument.java**

public class ExcelDocument implements Document{

    public void open(){

        System.out.println("Opening Excel Document"); } }

**ExcelDocumentFactory.java**

public class ExcelDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new ExcelDocument(); } }

**WordDocument.java**

public class WordDocument implements Document{

    public void open(){

        System.out.println("Opening word document"); } }

**WordDocumentFactory.java**

public class WordDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new WordDocument(); }

**PdfDocumentFactory.java**

public class PdfDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new PdfDocument(); } }

**PdfDocument.java**

public class PdfDocument implements Document{

    public void open(){

        System.out.println("Opening PDF Document"); } }

**Main.java**

public class Main {

    public static void main(String[] args) {

        DocumentFactory factory;

        factory = new WordDocumentFactory();

        Document doc1 = factory.createDocument();

        doc1.open();

        factory = new PdfDocumentFactory();

        Document doc2 = factory.createDocument();

        doc2.open();

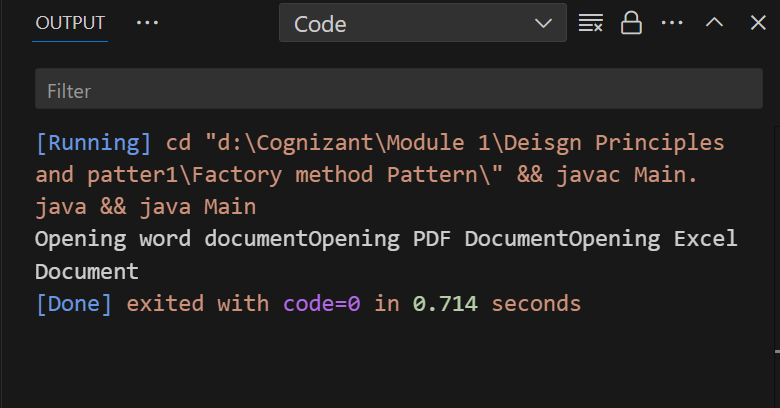
        factory = new ExcelDocumentFactory();

        Document doc3 = factory.createDocument();

        doc3.open();

    } }

**Output:**

****

**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Computer.java**

public class Computer {

    private String cpu;

    private String ram;

    private String storage;

    private String graphics;

    private Computer(Builder builder) {

        this.cpu = builder.cpu;

        this.ram = builder.ram;

        this.storage = builder.storage;

        this.graphics = builder.graphics;}

    public static class Builder {

        private String cpu;

        private String ram;

        private String storage;

        private String graphics;

        public Builder(String cpu, String ram) {

            this.cpu = cpu;

            this.ram = ram;}

        public Builder setStorage(String storage) {

            this.storage = storage;

            return this; }

        public Builder setGraphics(String graphics) {

            this.graphics = graphics;

            return this;}

        public Computer build(){

            return new Computer(this); } }

    public void displayConfig() {

        System.out.println("CPU: "+ cpu);

        System.out.println("RAM: "+ ram);

        System.out.println("Storage: "+ storage);

        System.out.println("Graphics: "+ graphics); } }

**Main.java**

public class Main {

    public static void main(String[] args) {

        Computer basic = new Computer.Builder("BBB", "8GB") .build();

         Computer gaming = new Computer.Builder("AAA", "16GB").setStorage("AAAA").setGraphics("GG").build();

        System.out.println("Basic Computer:");

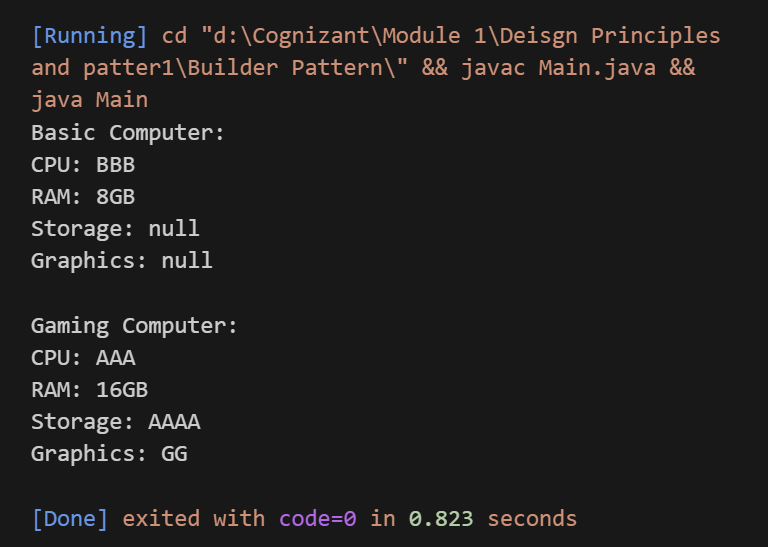
        basic.displayConfig();

        System.out.println("\nGaming Computer:");

        gaming.displayConfig();

    } }

**Output:**

****

**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**PaymentProcessor.java**

public interface PaymentProcessor {

    void processPayment(double amount); }

**PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway payPal;

    public PayPalAdapter(PayPalGateway payPal) {

        this.payPal = payPal; }

    public void processPayment(double amount) {

        payPal.sendPayment(amount);} }

**PayPalGateway.java**

public class PayPalGateway {

    public void sendPayment(double amount) {

        System.out.println("Payment is " + amount);} }

**public class Main {**

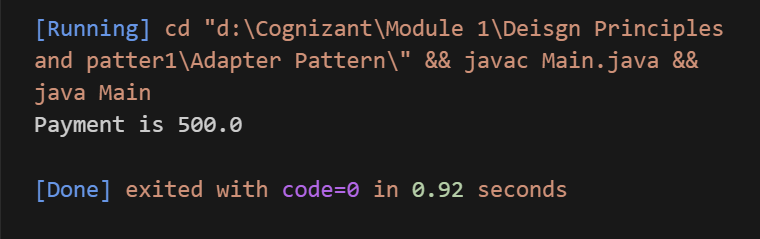
public static void main(String[] args) {

       PaymentProcessor payPalProcessor = new PayPalAdapter(new PayPalGateway());

        payPalProcessor.processPayment(500.0);

} }

**Output:**



**Exercise 5: Implementing the Decorator Pattern**

**Scenario:**

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

**EmailNotifier.java**

public class EmailNotifier implements Notifier {

    public void send(String message) {

        System.out.println("Email sent: " + message);

    }

}

**Notifier.java**

public interface Notifier {

    void send(String message);}

**NotifierDecorator.java**

public abstract class NotifierDecorator implements Notifier {

    protected Notifier wrappee;

    public NotifierDecorator(Notifier notifier) {

        this.wrappee = notifier; }

    public void send(String message) {

        wrappee.send(message); } }

**SlackNotifierDocorator.java**

public class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }public void send(String message) {

        super.send(message);

        System.out.println("Slack : " + message); } }

**SMSNotifierDecorator.java**

public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier); }

    public void send(String message) {

        super.send(message);

        System.out.println("SMS : " + message); } }

**Main.java**

public class Main {

    public static void main(String[] args) {

        Notifier email = new EmailNotifier();

        Notifier emailAndSms = new SMSNotifierDecorator(email);

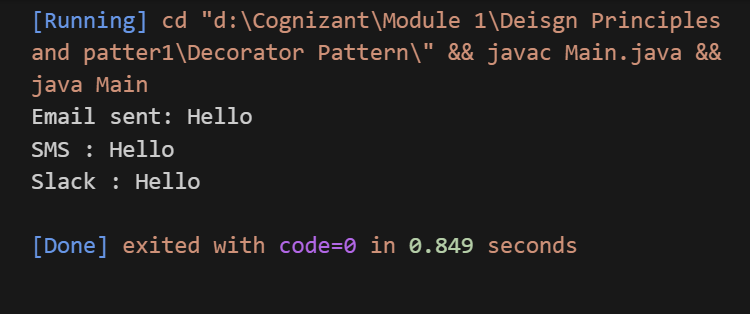
        Notifier fullNotifier = new SlackNotifierDecorator(emailAndSms);

        fullNotifier.send("Hello");

    }

}

**Output:**



**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**Image.java**

public interface Image {

    void display(); }

**ProxyImage.java**

public class ProxyImage implements Image {

    private String filename;

    private RealImage realImage;

    public ProxyImage(String filename) {

        this.filename = filename;}

    public void display() {

        if (realImage == null) {

            realImage = new RealImage(filename);  }

        realImage.display(); } }

**RealImage.java**

public class RealImage implements Image {

    private String filename;

    public RealImage(String filename) {

        this.filename = filename;

        loadFromServer();}

     private void loadFromServer() {

        System.out.println("Loading " + filename);}

    public void display() {

        System.out.println("image: " + filename); } }

**Main.java**

public class Main {

    public static void main(String[] args) {

        Image image1 = new ProxyImage("one.jpg");

        image1.display();

        image1.display();

        System.out.println();

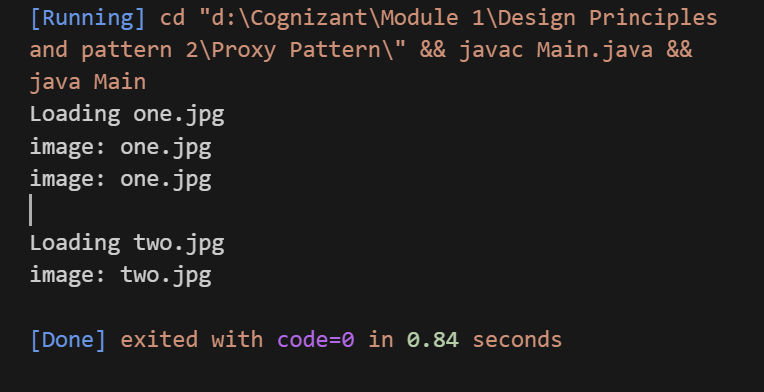
        Image image2 = new ProxyImage("two.jpg");

        image2.display();

    }

}

**Output:**



**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**WebApp.java**

public class WebApp implements Observer {

    public void update(double newPrice) {

        System.out.println("Web App Price: " + newPrice); } }

**StockMarket.java**

import java.util.\*;

public class StockMarket implements Stock {

    private List<Observer> observers = new ArrayList<>();

    private double stockPrice;

public void setStockPrice(double price) {

        this.stockPrice = price;

        notifyObservers();}

public double getStockPrice() {

        return stockPrice;}

public void registerObserver(Observer o) {

        observers.add(o);}

public void removeObserver(Observer o) {

        observers.remove(o);}

public void notifyObservers() {

        for (Observer o : observers) {

            o.update(stockPrice); } } }

**Stock.java**

public interface Stock {

    void registerObserver(Observer o);

    void removeObserver(Observer o);

    void notifyObservers();}

**Observer.java**

public interface Observer {

    void update(double newPrice);}

**MobileApp.java**

public class MobileApp implements Observer {

    public void update(double newPrice) {

        System.out.println("Mobile App: Price " + newPrice); } }

**Main.java**

public class Main {

    public static void main(String[] args) {

        StockMarket market = new StockMarket();

 Observer mobile = new MobileApp();

        Observer web = new WebApp();

market.registerObserver(mobile);

        market.registerObserver(web);

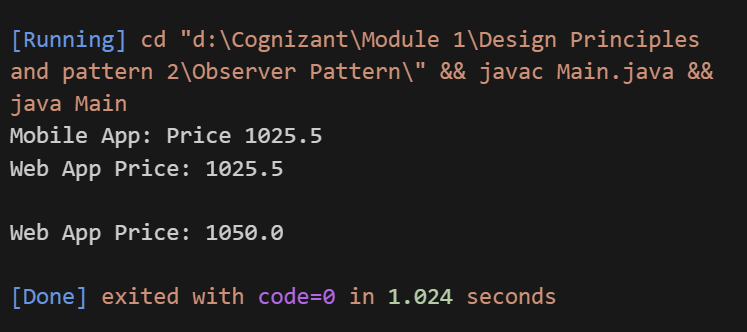
market.setStockPrice(1025.5);

System.out.println();

market.removeObserver(mobile);

        market.setStockPrice(1050.0); } }

**Output**:



**Exercise 8: Implementing the Strategy Pattern**

**Scenario:**

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

**PayPalPayment.java**

public class PayPalPayment implements PaymentStrategy {

    public void pay(double amount) {

        System.out.println("Paid "+amount+" - PayPal.");}}

**PaymentStrategy.java**

public interface PaymentStrategy {

    void pay(double amount);

}

**PaymentContext.java**

public class PaymentContext {

    private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;}

public void payAmount(double amount) {

        if (strategy == null) {

            System.out.println("No payment method selected.");

        } else {

            strategy.pay(amount);} } }

**CreditCardPayment.java**

public class CreditCardPayment implements PaymentStrategy {

    public void pay(double amount) {

        System.out.println("Paid " +amount+"- Credit Card.");

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        context.setPaymentStrategy(new CreditCardPayment());

        context.payAmount(1500.00);

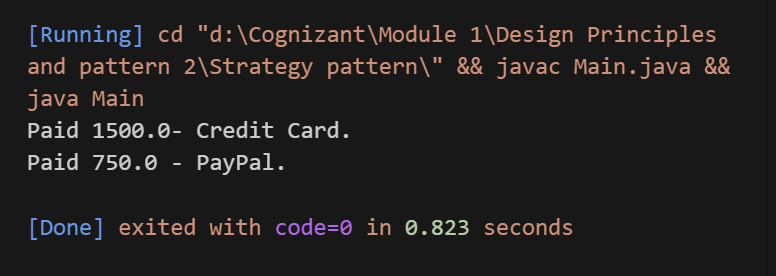
        context.setPaymentStrategy(new PayPalPayment());

        context.payAmount(750.00);

    }

}

**Output:**



**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

**RemoteControl.java**

public class RemoteControl {

    private Command command;

public void setCommand(Command command) {

        this.command = command;

    }

public void pressButton() {

        command.execute();} }

**LightOnCommand.java**

public class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }public void execute() {

        light.turnOn();} }

**LightOffCommand.java**

public class LightOffCommand implements Command {

    private Light light;

public LightOffCommand(Light light) {

        this.light = light;}

public void execute() {

        light.turnOff();} }

**Light.java**

public class Light {

    public void turnOn() {

        System.out.println("Light ON");}

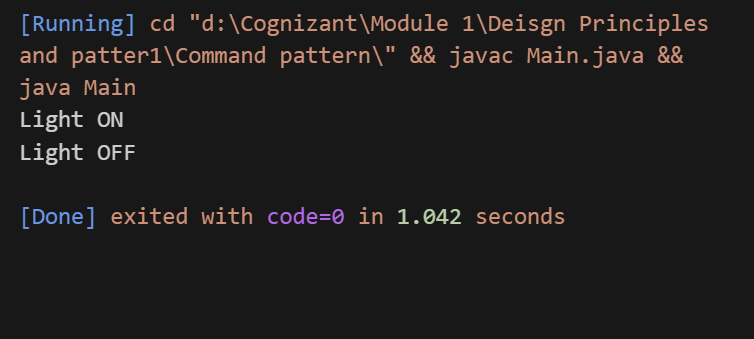
    public void turnOff() {

        System.out.println("Light OFF");} } }

public interface Command {

    void execute(); }

**Output:**



**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern

**StudentView.java**

public class StudentView {

    public void displayStudentDetails(String name, String id, String grade) {

        System.out.println("Student Details:");

        System.out.println("Name: " + name);

        System.out.println("ID: " + id);

        System.out.println("Grade: " + grade);} }

**StudentController.java**

public class StudentController {

    private Student model;

    private StudentView view;

public StudentController(Student model, StudentView view) {

        this.model = model;

        this.view = view;

    }public void setStudentName(String name) {

        model.setName(name);}

 public void setStudentId(String id) {

        model.setId(id);

    }public void setStudentGrade(String grade) {

        model.setGrade(grade);

    } public void updateView() {

        view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());}}

**Student.java**

public class Student {

    private String name;

    private String id;

    private String grade;

public String getName() {

        return name;}

public void setName(String name) { this.name = name;}

public String getId() { return id;}

 public void setId(String id) { this.id = id;}

public String getGrade() { return grade;}  
 public void setGrade(String grade) {this.grade = grade;} }

**Main.java**

public class Main {

    public static void main(String[] args) {

        Student student = new Student();

        student.setName("Dharshini");

        student.setId("101");

        student.setGrade("A+");

StudentView view = new StudentView();

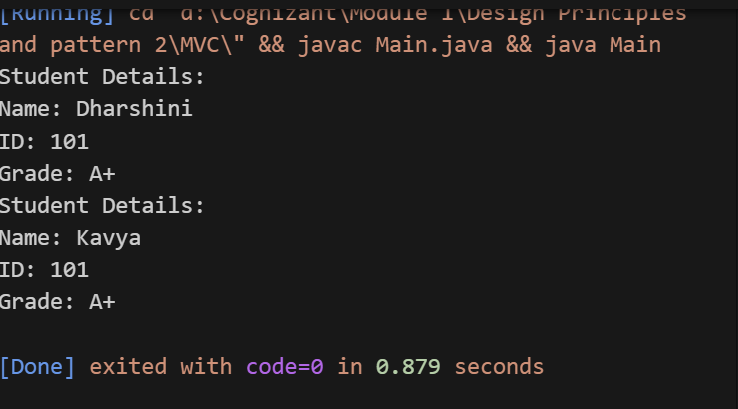
StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Kavya");

        controller.updateView();} }

**Output:**

****

**Exercise 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

**Customer.java**

public class CustomerService {

    private final CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;}

    public void getCustomerDetails(String id) {

        String customer = customerRepository.findCustomerById(id);

        System.out.println("Customer Details: " + customer);}}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

    public String findCustomerById(String id) {

        return "ID: "+id+", Name: Dharshini";}}

**CustomerRepository.java**

public interface CustomerRepository {

    String findCustomerById(String id);

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        CustomerRepository repository = new CustomerRepositoryImpl();

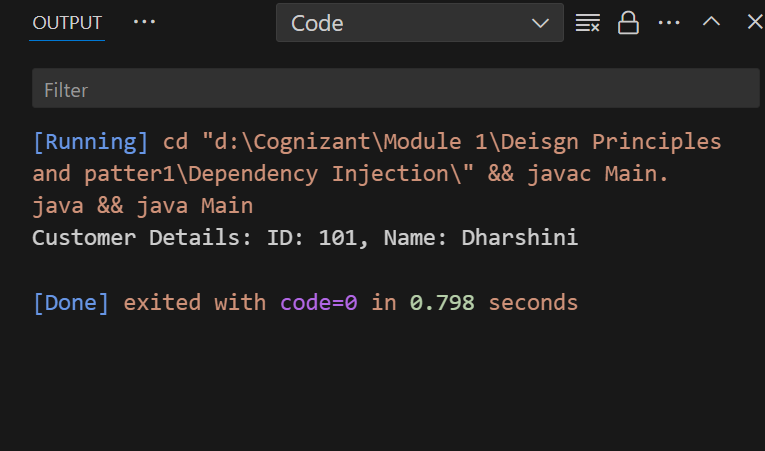
       CustomerService service = new CustomerService(repository);

service.getCustomerDetails("101");

    }

}

**Output:**

****