**SUPERSET ID :** **6416829**

**Exercise 1: Implementing the Singleton Pattern**

**PROGRAM:**

Logger.java

public class Logger {

private static Logger instance;

private Logger() {

System.out.println("Logger instance created");

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

System.out.println("[LOG]: " + message);

}

}

Main.java

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("First message");

logger2.log("Second message");

if (logger1 == logger2) {

System.out.println("Only one Logger instance exists.");

} else {

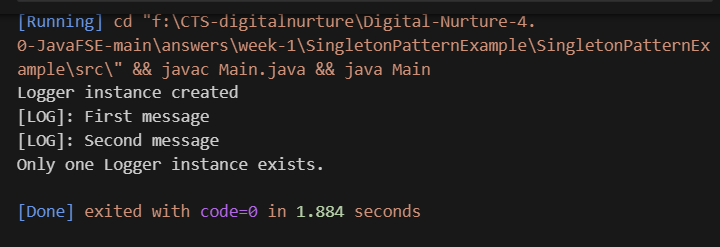
System.out.println("Different Logger instances exist.");

}

}

}

Output:



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

**PROGRAM:**

// DocumentFactoryPattern.java

interface Document {

void open();

}

class WordDocument implements Document {

public void open() { System.out.println("Opening Word Document..."); }

}

class PdfDocument implements Document {

public void open() { System.out.println("Opening PDF Document..."); }

}

class ExcelDocument implements Document {

public void open() { System.out.println("Opening Excel Document..."); }

}

abstract class DocumentFactory {

public abstract Document createDocument();

}

class WordFactory extends DocumentFactory {

public Document createDocument() { return new WordDocument(); }

}

class PdfFactory extends DocumentFactory {

public Document createDocument() { return new PdfDocument(); }

}

class ExcelFactory extends DocumentFactory {

public Document createDocument() { return new ExcelDocument(); }

}

public class DocumentFactoryPattern {

public static void main(String[] args) {

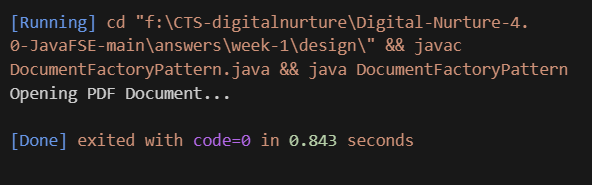
DocumentFactory factory = new PdfFactory();

Document doc = factory.createDocument();

doc.open();

}

}

Output: 

**Exercise 3: Implementing the Builder Pattern**

PROGRAM:

// Computer.java

class Computer {

private String CPU, RAM, storage;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

}

static class Builder {

private String CPU, RAM, storage;

public Builder setCPU(String cpu) { this.CPU = cpu; return this; }

public Builder setRAM(String ram) { this.RAM = ram; return this; }

public Builder setStorage(String storage) { this.storage = storage; return this; }

public Computer build() { return new Computer(this); }

}

public String toString() {

return "Computer[CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage + "]";

}

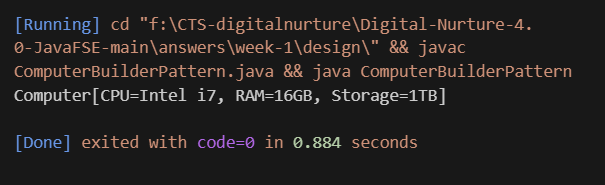
public static void main(String[] args) {

Computer comp = new Builder().setCPU("Intel i7").setRAM("16GB").setStorage("1TB").build();

System.out.println(comp);

}

}

Output: 

**Exercise 4: Implementing the Adapter Pattern**

// PaymentAdapterPattern.java

interface PaymentProcessor {

void processPayment();

}

class PayPalGateway {

public void makePayPalPayment() {

System.out.println("Payment processed via PayPal.");

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPalGateway gateway = new PayPalGateway();

public void processPayment() { gateway.makePayPalPayment(); }

}

public class PaymentAdapterPattern {

public static void main(String[] args) {

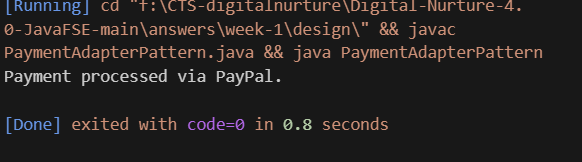
PaymentProcessor processor = new PayPalAdapter();

processor.processPayment();

}

}

Output:



**Exercise 5: Implementing the Decorator Pattern**

PROGRAM:

// NotificationDecoratorPattern.java

interface Notifier {

void send(String message);

}

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

}

class SMSNotifier extends NotifierDecorator {

public SMSNotifier(Notifier notifier) { super(notifier); }

public void send(String message) {

notifier.send(message);

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

}

}

public class NotificationDecoratorPattern {

public static void main(String[] args) {

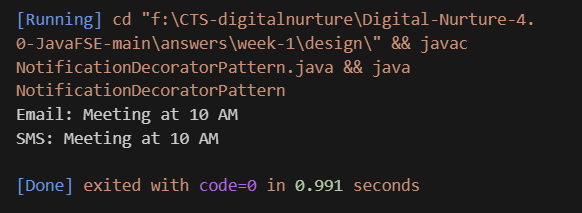
Notifier notifier = new SMSNotifier(new EmailNotifier());

notifier.send("Meeting at 10 AM");

}

}

Output:



**Exercise 6: Implementing the Proxy Pattern**

PROGRAM:

// ImageProxyPattern.java

interface Image {

void display();

}

class RealImage implements Image {

private String file;

public RealImage(String file) {

this.file = file;

loadFromDisk();

}

private void loadFromDisk() {

System.out.println("Loading image: " + file);

}

public void display() {

System.out.println("Displaying: " + file);

}

}

class ProxyImage implements Image {

private RealImage realImage;

private String file;

public ProxyImage(String file) {

this.file = file;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(file);

}

realImage.display();

}

}

public class ImageProxyPattern {

public static void main(String[] args) {

Image img = new ProxyImage("photo.jpg");

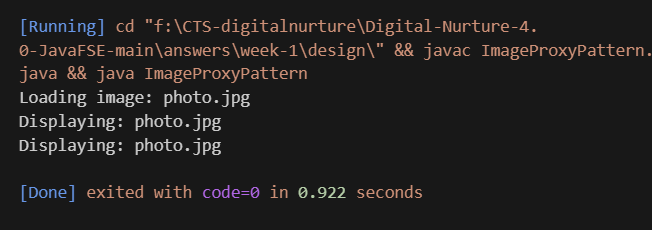
img.display(); // loads and displays

img.display(); // only displays

}

}

Output:



**Exercise 7: Implementing the Observer Pattern**

PROGRAM:

// StockObserverPattern.java

import java.util.\*;

interface Observer {

void update(float price);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

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

private float price;

public void register(Observer o) { observers.add(o); }

public void deregister(Observer o) { observers.remove(o); }

public void setPrice(float price) {

this.price = price;

notifyObservers();

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(price);

}

}

}

class MobileApp implements Observer {

public void update(float price) {

System.out.println("Mobile App - New price: " + price);

}

}

public class StockObserverPattern {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobApp = new MobileApp();

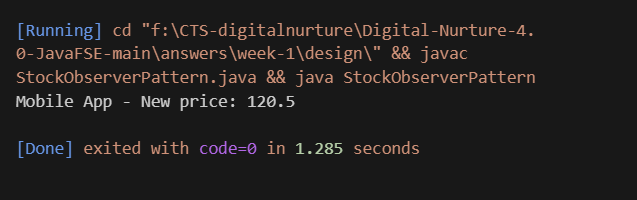
market.register(mobApp);

market.setPrice(120.5f);

}

}

Output:



**Exercise 8: Implementing the Strategy Pattern**

PROGRAM:

// PaymentStrategyPattern.java

interface PaymentStrategy {

void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using PayPal.");

}

}

class PaymentContext {

private PaymentStrategy strategy;

public PaymentContext(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

strategy.pay(amount);

}

public static void main(String[] args) {

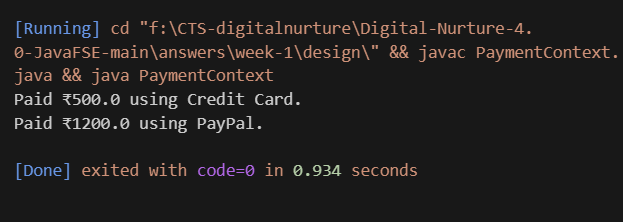
new PaymentContext(new CreditCardPayment()).executePayment(500);

new PaymentContext(new PayPalPayment()).executePayment(1200);

}

}

Output:



**Exercise 9: Implementing the Command Pattern**

PROGRAM:

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 {

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();

}

}

public class CommandPatternExample {

public static void main(String[] args) {

Light light = new Light();

Command on = new LightOnCommand(light);

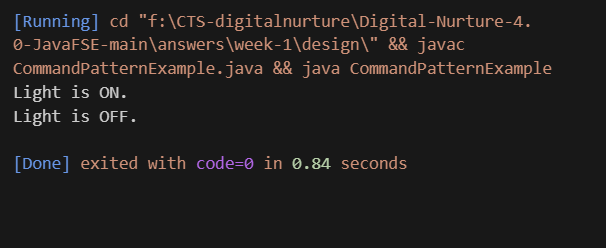
Command off = new LightOffCommand(light);

on.execute();

off.execute();

}

}Output:



**Exercise 10: Implementing the MVC Pattern**

PROGRAM:

// MVCPatternExample.java

class Student {

private String name;

private int id;

private String grade;

public Student(String name, int id, String grade) {

this.name = name; this.id = id; this.grade = grade;

}

public String getName() { return name; }

public int getId() { return id; }

public String getGrade() { return grade; }

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

}

class StudentView {

public void displayStudentDetails(Student student) {

System.out.println("Student: " + student.getName() + ", ID: " + student.getId() + ", Grade: " + student.getGrade());

}

}

class StudentController {

private Student student;

private StudentView view;

public StudentController(Student student, StudentView view) {

this.student = student; this.view = view;

}

public void updateGrade(String grade) { student.setGrade(grade); }

public void showStudent() { view.displayStudentDetails(student); }

public static void main(String[] args) {

Student stu = new Student("Asha", 1, "A");

StudentView view = new StudentView();

StudentController ctrl = new StudentController(stu, view);

ctrl.showStudent();

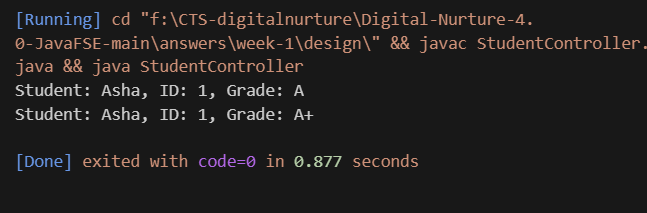
ctrl.updateGrade("A+");

ctrl.showStudent();

}

}

Output:



**Exercise 11: Implementing Dependency Injection**

PROGRAM:

// DependencyInjectionExample.java

interface CustomerRepository {

String findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer#" + id + " found.";

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomer(int id) {

System.out.println(repository.findCustomerById(id));

}

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.getCustomer(101);

}

}

Output:

