**Week 01 superset ID:6364190**

**Design pattern and principles**

.

**Exercise 1: Implementing the Singleton Pattern**

Code :

class Logger

{

private static Logger instance;

private Logger()

{

System.out.println("Logger initialized.");

}

public static Logger getInstance()

{

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message)

{

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

}

}

public class TestLogger

{

public static void main(String[] args)

{

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("Application started");

logger2.log("Performing some operation");

if (logger1 == logger2) {

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

} else

{

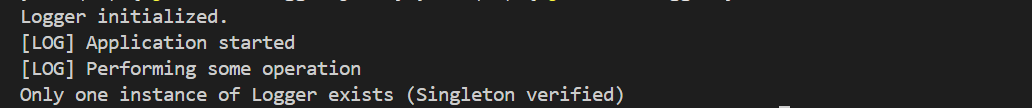
System.out.println("Multiple instances found (Singleton failed)");

}

}

}

Output :



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

CODE:

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 void createDocument();

}

class WordDocumentFactory extends DocumentFactory {

public void createDocument() {

Document doc = new WordDocument();

doc.open();

}

}

class PdfDocumentFactory extends DocumentFactory {

public void createDocument() {

Document doc = new PdfDocument();

doc.open();

}

}

class ExcelDocumentFactory extends DocumentFactory {

public void createDocument() {

Document doc = new ExcelDocument();

doc.open();

}

}

public class TestDocumentFactory {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

DocumentFactory pdfFactory = new PdfDocumentFactory();

DocumentFactory excelFactory = new ExcelDocumentFactory();

wordFactory.createDocument();

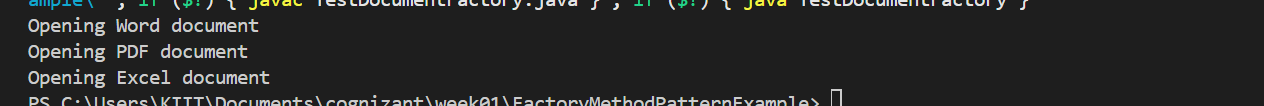
pdfFactory.createDocument();

excelFactory.createDocument();

}

}

Output :



**Exercise 3: Implementing the Builder Pattern**

CODE:

class Computer {

private String CPU;

private String RAM;

private String storage;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

}

static class Builder {

private String CPU;

private String RAM;

private String storage;

public Builder(String CPU, String RAM) {

this.CPU = CPU;

this.RAM = RAM;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Computer build() {

return new Computer(this);

}

}

// Display

public void showSpecs() {

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

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

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

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

System.out.println();

}

}

public class Testclass {

public static void main(String[] args) {

// Build a basic computer

Computer basicComputer = new Computer.Builder("Intel i5", "8GB")

.setStorage("256GB SSD")

.build();

// Show configurations

basicComputer.showSpecs();

}

}

Output :



**Exercise 4: Implementing the Adapter Pattern**

CODE :

interface payme {

void processPayment(double amount);

}

// Adaptee 1

class PayPalGateway {

public void sendPayment(double amount) {

System.out.println("Processing PayPal payment of $" + amount);

}

}

// Adaptee 2

class StripeGateway {

public void makeStripePayment(double amount) {

System.out.println("Processing Stripe payment of $" + amount);

}

}

// Adaptee 3

class RazorpayGateway {

public void pay(double money) {

System.out.println("Processing Razorpay payment of $" + money);

}

}

// Adapter 1

class PayPalAdapter implements payme {

private PayPalGateway paypal = new PayPalGateway();

public void processPayment(double amount) {

paypal.sendPayment(amount);

}

}

// Adapter 2

class StripeAdapter implements payme {

private StripeGateway stripe = new StripeGateway();

public void processPayment(double amount) {

stripe.makeStripePayment(amount);

}

}

// Adapter 3

class RazorpayAdapter implements payme {

private RazorpayGateway razorpay = new RazorpayGateway();

public void processPayment(double amount) {

razorpay.pay(amount);

}

}

public class Testadapter {

public static void main(String[] args) {

payme paypal = new PayPalAdapter();

payme stripe = new StripeAdapter();

payme razorpay = new RazorpayAdapter();

paypal.processPayment(100.00);

stripe.processPayment(250.75);

razorpay.processPayment(500.50);

}

}

Output;



**Exercise 6: Implementing the Proxy Pattern**

CODE:

interface Image {

void display();

}

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

System.out.println("Loading image from remote server: " + filename);

}

public void display() {

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

}

}

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);

} else {

System.out.println("Using cached image for: " + filename);

}

realImage.display();

}

}

public class testexample {

public static void main(String[] args) {

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

Image image2 = new ProxyImage("portrait.png");

image1.display();

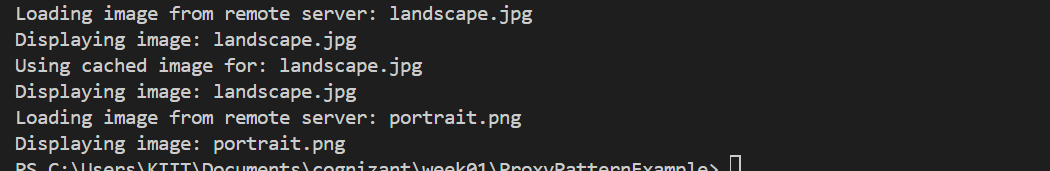
image1.display();

image2.display();

}

}

OUtput ;



**Exercise 7: Implementing the Observer Pattern**

Code:

import java.util.\*;

interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public StockMarket(String stockName) {

this.stockName = stockName;

}

public void setPrice(double newPrice) {

System.out.println("\nStock price updated: " + stockName + " = $" + newPrice);

this.stockPrice = newPrice;

notifyObservers();

}

public double getPrice() {

return stockPrice;

}

public String getStockName() {

return stockName;

}

@Override

public void registerObserver(Observer o) {

observers.add(o);

}

@Override

public void removeObserver(Observer o) {

observers.remove(o);

}

@Override

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

}

interface Observer {

void update(String stockName, double newPrice);

}

class MobileApp implements Observer {

private String user;

public MobileApp(String user) {

this.user = user;

}

@Override

public void update(String stockName, double newPrice) {

System.out.println("MobileApp [" + user + "] - " + stockName + " is now $" + newPrice);

}

}

class WebApp implements Observer {

private String user;

public WebApp(String user) {

this.user = user;

}

@Override

public void update(String stockName, double newPrice) {

System.out.println("WebApp [" + user + "] - " + stockName + " updated to $" + newPrice);

}

}

public class Testclass {

public static void main(String[] args) {

StockMarket teslaStock = new StockMarket("TSLA");

Observer user1 = new MobileApp("Alice");

Observer user2 = new WebApp("Bob");

teslaStock.registerObserver(user1);

teslaStock.registerObserver(user2);

teslaStock.setPrice(720.50);

teslaStock.setPrice(735.75);

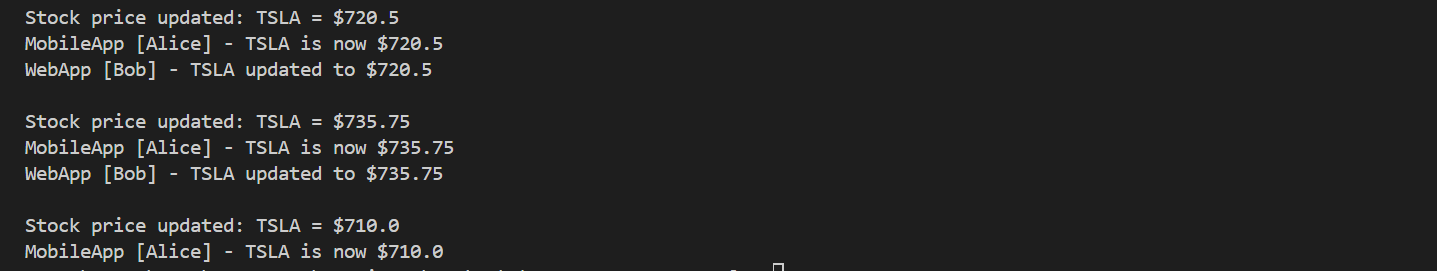
teslaStock.removeObserver(user2);

teslaStock.setPrice(710.00);

}

}

Output



**Exercise 8: Implementing the Strategy Pattern**

interface PaymentStrategy {

void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolder;

public CreditCardPayment(String cardNumber, String cardHolder) {

this.cardNumber = cardNumber;

this.cardHolder = cardHolder;

}

@Override

public void pay(double amount) {

System.out.println("Paid $" + amount + " using Credit Card: " + cardHolder);

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(double amount) {

System.out.println("Paid $" + amount + " using PayPal account: " + email);

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

if (strategy == null) {

System.out.println("Payment method not selected.");

} else {

strategy.pay(amount);

}

}

}

public class TestClass {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment("1234", "india"));

context.executePayment(120.00);

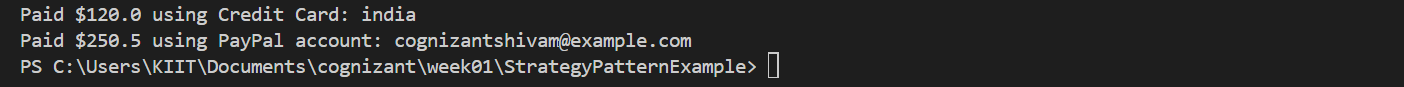
context.setPaymentStrategy(new PayPalPayment("cognizantshivam@example.com"));

context.executePayment(250.50);

}

}

Output :



**Exercise 9: Implementing the Command Pattern**

Code :

interface Command {

void execute();

}

class Light {

void turnOn() {

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

}

void turnOff() {

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

}

}

class LightOnCommand implements Command {

private Light light;

LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

class LightOffCommand implements Command {

private Light light;

LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

class RemoteControl {

private Command command;

void setCommand(Command command) {

this.command = command;

}

void pressButton() {

command.execute();

}

}

public class CommandPatternExample {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command onCommand = new LightOnCommand(livingRoomLight);

Command offCommand = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(onCommand);

remote.pressButton();

remote.setCommand(offCommand);

remote.pressButton();

}

}

Output :



**Exercise 10: Implementing the MVC Pattern**

Code :

class Student {

private String name;

private String id;

private String grade;

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

this.name = name;

this.id = id;

this.grade = grade;

}

public String getName() { return name; }

public String getId() { return id; }

public String getGrade() { return grade; }

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

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

}

// View

class StudentView {

public void displayStudentDetails(Student s) {

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

System.out.println("Name: " + s.getName());

System.out.println("ID: " + s.getId());

System.out.println("Grade: " + s.getGrade());

}

}

// Controller

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void updateStudent(String name, String grade) {

model.setName(name);

model.setGrade(grade);

}

public void displayStudent() {

view.displayStudentDetails(model);

}

}

// Main

class MVCPatternExample {

public static void main(String[] args) {

Student s = new Student("Shivam singh", “66", "A");

StudentView v = new StudentView();

StudentController c = new StudentController(s, v);

c.displayStudent();

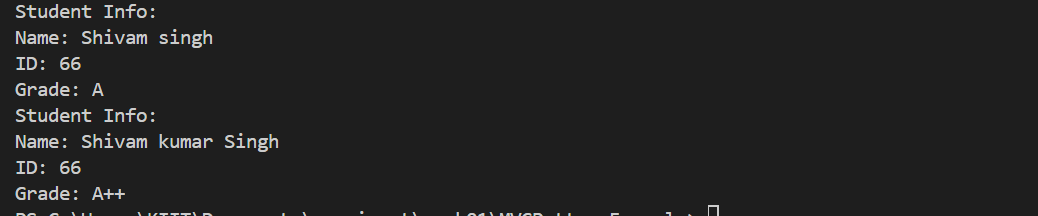
c.updateStudent("shivam kumar singh", "A++");

c.displayStudent();

}

}

Output :



**Exercise 11: Implementing Dependency Injection**

Code :

// Repository Interface

interface CustomerRepository {

String findCustomerById(int id);

}

// Concrete Repository

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer #" + id + ": Riya Sharma";

}

}

// Service Class

class CustomerService {

private CustomerRepository repo;

// Constructor Injection

public CustomerService(CustomerRepository repo) {

this.repo = repo;

}

public void printCustomer(int id) {

String customer = repo.findCustomerById(id);

System.out.println("Retrieved: " + customer);

}

}

// Main Class

class DependencyInjectionExample {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.printCustomer(1001); // Output: Retrieved: Customer #1001: Riya Sharma

}

}

Output :

