**WEEK-1**

**1) Implementing the Singleton Pattern**

**CODE:**

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);

}

}

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("Application started.");

Logger logger2 = Logger.getInstance();

logger2.log("Second log message.");

if (logger1 = = logger2) {

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

} else {

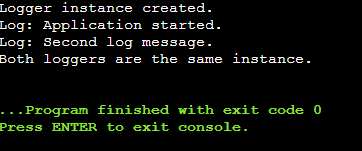
System.out.println("Different logger instances.");

}

}

}

**OUTPUT:**

****

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

**CODE:**

import java.util.Scanner;

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

System.out.println("Opening a Word document.");

}

}

class PdfDocument implements Document {

public void open() {

System.out.println("Opening a PDF document.");

}

}

class ExcelDocument implements Document {

public void open() {

System.out.println("Opening an Excel document.");

}

}

interface DocumentFactory {

Document createDocument();

}

class WordFactory implements DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfFactory implements DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelFactory implements DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

class DocumentService {

private final DocumentFactory factory;

public DocumentService(DocumentFactory factory) {

this.factory = factory;

}

public void handleOpen() {

Document doc = factory.createDocument();

doc.open();

}

}

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.println("Choose Document Type:");

System.out.println("1. Word");

System.out.println("2. PDF");

System.out.println("3. Excel");

System.out.print("Enter your choice: ");

int choice = scanner.nextInt();

DocumentFactory factory;

switch (choice) {

case 1 -> factory = new WordFactory();

case 2 -> factory = new PdfFactory();

case 3 -> factory = new ExcelFactory();

default -> {

System.out.println("Invalid choice.");

return;

}

}

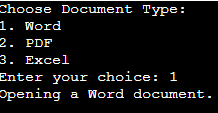
DocumentService service = new DocumentService(factory);

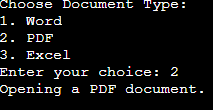
service.handleOpen();

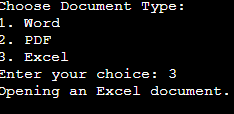
}

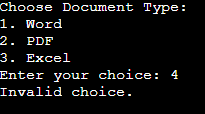
}

**OUTPUT:**

****

****

****

****

**3) Implementing the Builder Pattern**

**CODE:**

class Computer {

private final String cpu;

private final String ram;

private final String storage;

private final String graphicsCard;

private final String motherboard;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.motherboard = builder.motherboard;

}

public String toString() {

return "Computer Configuration:\n" +

"CPU: " + cpu + "\n" +

"RAM: " + ram + "\n" +

"Storage: " + storage + "\n" +

"Graphics Card: " + graphicsCard + "\n" +

"Motherboard: " + motherboard;

}

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

private String motherboard;

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 Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder setMotherboard(String motherboard) {

this.motherboard = motherboard;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

public class Main {

public static void main(String[] args) {

Computer officeComputer = new Computer.Builder()

.setCpu("Intel i5")

.setRam("8GB")

.setStorage("512GB SSD")

.build();

Computer gamingComputer = new Computer.Builder()

.setCpu("AMD Ryzen 9")

.setRam("32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.setMotherboard("ASUS ROG")

.build();

System.out.println(officeComputer);

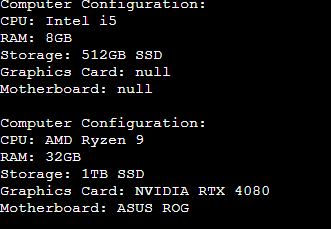
System.out.println();

System.out.println(gamingComputer);

}

}

**OUTPUT:**



**4) Implementing the Adapter Pattern**

**CODE:**

interface PaymentProcessor {

void processPayment(double amount);

}

class PayPalGateway {

public void makePayment(double amount) {

System.out.println("Payment of Rs" + amount + " processed through PayPal.");

}

}

class PhonePeGateway {

public void upiTransfer(double amount) {

System.out.println("Payment of Rs" + amount + " processed through PhonePe.");

}

}

class GPayGateway {

public void payUsingUpi(double amount) {

System.out.println("Payment of Rs" + amount + " processed through GPay.");

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

class PhonePeAdapter implements PaymentProcessor {

private PhonePeGateway phonePe;

public PhonePeAdapter(PhonePeGateway phonePe) {

this.phonePe = phonePe;

}

public void processPayment(double amount) {

phonePe.upiTransfer(amount);

}

}

class GPayAdapter implements PaymentProcessor {

private GPayGateway gpay;

public GPayAdapter(GPayGateway gpay) {

this.gpay = gpay;

}

public void processPayment(double amount) {

gpay.payUsingUpi(amount);

}

}

public class Main {

public static void main(String[] args) {

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

PaymentProcessor phonePeProcessor = new PhonePeAdapter(new PhonePeGateway());

PaymentProcessor gpayProcessor = new GPayAdapter(new GPayGateway());

paypalProcessor.processPayment(600);

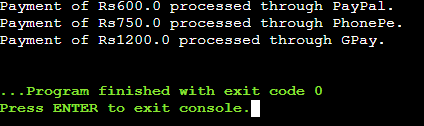
phonePeProcessor.processPayment(750);

gpayProcessor.processPayment(1200);

}

}

**OUTPUT:**



**5)Implementing the Decorator Pattern**

**CODE:**

interface Notifier {

void send();

}

class EmailNotifier implements Notifier {

public void send() {

System.out.println("Sending notification via Email.");

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send() {

notifier.send();

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send() {

super.send();

System.out.println("Sending notification via SMS.");

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send() {

super.send();

System.out.println("Sending notification via Slack.");

}

}

public class Main {

public static void main(String[] args) {

Notifier baseNotifier = new EmailNotifier();

Notifier smsDecorator = new SMSNotifierDecorator(baseNotifier);

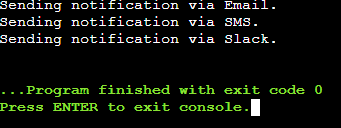
Notifier slackDecorator = new SlackNotifierDecorator(smsDecorator);

slackDecorator.send();

}

}

**OUTPUT:**

****

**6)Implementing the Proxy Pattern**

**CODE:**

interface Image {

void display();

}

class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromServer();

}

private void loadFromServer() {

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: " + fileName);

}

realImage.display();

}

}

public class Main {

public static void main(String[] args) {

Image image1 = new ProxyImage("photo1.jpg");

Image image2 = new ProxyImage("photo2.jpg");

image1.display();

System.out.println();

image1.display();

System.out.println();

image2.display();

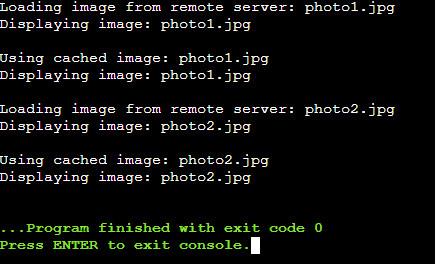
System.out.println();

image2.display();

}

}

**OUTPUT:**

****

1. **Implementing the Observer Pattern**

**CODE:**

import java.util.\*;

interface Observer {

void update(String stockName, double price);

}

interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers(String stockName, double price);

}

class StockMarket implements Stock {

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

public void registerObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers(String stockName, double price) {

for (Observer o : observers) {

o.update(stockName, price);

}

}

public void setStockPrice(String stockName, double price) {

System.out.println("Stock Update: " + stockName + " is now ₹" + price);

notifyObservers(stockName, price);

}

}

class MobileApp implements Observer {

public void update(String stockName, double price) {

System.out.println("MobileApp received update: " + stockName + " is ₹" + price);

}

}

class WebApp implements Observer {

public void update(String stockName, double price) {

System.out.println("WebApp received update: " + stockName + " is ₹" + price);

}

}

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("TCS", 4000);

System.out.println();

market.setStockPrice("Infosys", 5000);

System.out.println();

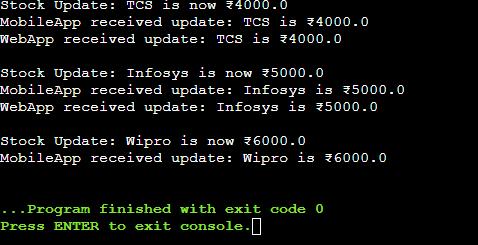
market.removeObserver(web);

market.setStockPrice("Wipro", 6000);

}

}

**OUTPUT:**

****

1. **Implementing the Strategy Pattern**

**CODE:**

import java.util.\*;

interface PaymentStrategy {

void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

public void pay(double amount) {

System.out.println("Paid Rs " + amount + " using Credit Card ending with " + cardNumber.substring(cardNumber.length() - 4));

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

public void pay(double amount) {

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

}

}

class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void executePayment(double amount) {

if (paymentStrategy != null) {

paymentStrategy.pay(amount);

} else {

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

}

}

}

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

PaymentStrategy creditCard = new CreditCardPayment("1234567890125789");

context.setPaymentStrategy(creditCard);

context.executePayment(1500);

System.out.println();

PaymentStrategy paypal = new PayPalPayment("hello@example.com");

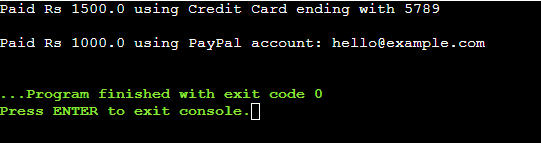
context.setPaymentStrategy(paypal);

context.executePayment(1000);

}

}

**OUTPUT:**



1. **Implementing the Command Pattern**

**CODE:**

interface Command {

void execute();

}

class Light {

public void turnOn() {

System.out.println("The light is ON.");

}

public void turnOff() {

System.out.println("The light is OFF.");

}

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

if (command != null) {

command.execute();

}

}

}

public class Main {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

remote.pressButton();

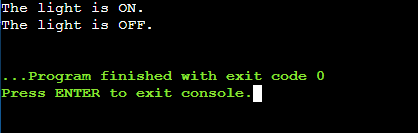
remote.setCommand(lightOff);

remote.pressButton();

}

}

**OUTPUT:**



1. **Implementing the MVC Pattern**

**CODE:**

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;

}

}

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);

}

}

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());

}

}

public class Main {

public static void main(String[] args) {

Student model = new Student();

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.setStudentName("Indu");

controller.setStudentId("12345");

controller.setStudentGrade("B");

controller.updateView();

controller.setStudentGrade("B+");

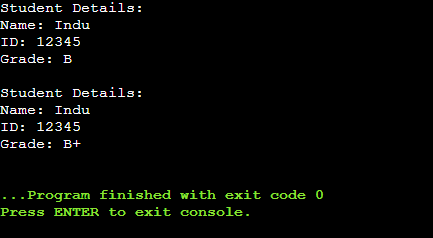
System.out.println();

controller.updateView();

}

}

**OUTPUT:**

****

1. **Implementing Dependency Injection**

**CODE:**

interface CustomerRepository {

String findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

if (id.equals("C001")) {

return "Sony";

} else if (id.equals("C002")) {

return "Sreeja";

} else {

return "Customer not found";

}

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomerDetails(String id) {

String customer = repository.findCustomerById(id);

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

}

}

public class Main {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.getCustomerDetails("C001");//since id is equal it returns

service.getCustomerDetails("C003");//id is not equal so it returns customer not found

}

}

**OUTPUT:**

