**DESIGN PATTERNS**

1. **Singleton Pattern**

**Code:**

**// 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 method to test singleton behavior**

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("Starting application");

        Logger logger2 = Logger.getInstance();

        logger2.log("Continuing application");

        if (logger1 == logger2) {

            System.out.println("Both logger1 and logger2 are the same instance.");

        } else {

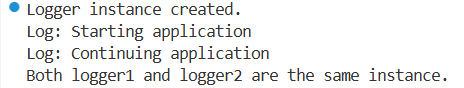
            System.out.println("Different instances exist! Singleton not working.");

        }

    }

}

**Output:**



1. **Factory Method Pattern**

**Code:**

**// Document.java**

public interface Document {

void open();

}

**// WordDocument.java**

public class WordDocument implements Document {

@Override

public void open() {

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

}

}

**// PdfDocument.java**

public class PdfDocument implements Document {

@Override

public void open() {

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

}

}

**// ExcelDocument.java**

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

**// DocumentFactory.java**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**// WordFactory.java**

public class WordFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

**// PdfFactory.java**

public class PdfFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

**// ExcelFactory.java**

public class ExcelFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**// Main.java**

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelFactory();

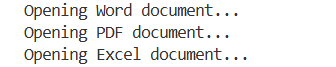
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Output:**



1. **Builder Pattern**

**Code:**

**// Computer.java**

package Week1.DesignPatterns.BuilderPatternExample;

public class Computer {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

private String operatingSystem;

// Private constructor: only accessible from the Builder

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.operatingSystem = builder.operatingSystem;

}

// Static nested Builder class

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

private String operatingSystem;

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 setOperatingSystem(String operatingSystem) {

this.operatingSystem = operatingSystem;

return this;

}

// Final step - return the constructed Computer

public Computer build() {

return new Computer(this);

}

}

// For printing the Computer configuration

public void showConfig() {

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

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

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

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

System.out.println("Graphics Card: " + graphicsCard);

System.out.println("Operating System: " + operatingSystem);

System.out.println();

}

}

**//Main.java**

package Week1.DesignPatterns.BuilderPatternExample;

public class Main {

public static void main(String[] args) {

// Basic computer configuration

Computer basicComputer = new Computer.Builder()

.setCpu("Intel i3")

.setRam("8GB")

.setStorage("256GB SSD")

.build();

basicComputer.showConfig();

// High-end gaming configuration

Computer gamingComputer = new Computer.Builder()

.setCpu("Intel i9")

.setRam("32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4090")

.setOperatingSystem("Windows 11 Pro")

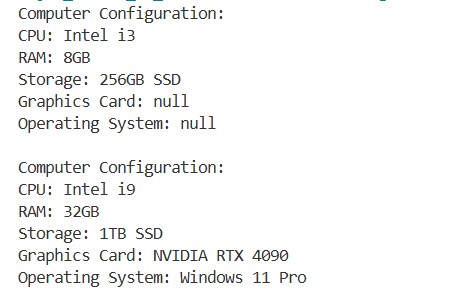
.build();

gamingComputer.showConfig();

}

}

**Output:**



1. **Adapter Pattern**

**Code:**

**// PaymentProcessor.java**

package Week1.DesignPatterns.AdapterPatternExample;

public interface PaymentProcessor {

void processPayment(double amount);

}

**// PayPal.java**

package Week1.DesignPatterns.AdapterPatternExample;

public class PayPal {

public void makePayment(double amount) {

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

}

}

**// Stripe.java**

package Week1.DesignPatterns.AdapterPatternExample;

public class Stripe {

public void pay(double amountInRupees) {

System.out.println("Paid ₹" + amountInRupees + " using Stripe.");

}

}

**// Razorpay.java**

package Week1.DesignPatterns.AdapterPatternExample;

public class Razorpay {

public void executeTransaction(double value) {

System.out.println("Paid ₹" + value + " using Razorpay.");

}

}

**// PayPalAdapter.java**

package Week1.DesignPatterns.AdapterPatternExample;

public class PayPalAdapter implements PaymentProcessor {

private PayPal paypal = new PayPal();

@Override

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

**// StripeAdapter.java**

package Week1.DesignPatterns.AdapterPatternExample;

public class StripeAdapter implements PaymentProcessor {

private Stripe stripe = new Stripe();

@Override

public void processPayment(double amount) {

stripe.pay(amount);

}

}

**// RazorpayAdapter.java**

package Week1.DesignPatterns.AdapterPatternExample;

public class RazorpayAdapter implements PaymentProcessor {

private Razorpay razorpay = new Razorpay();

@Override

public void processPayment(double amount) {

razorpay.executeTransaction(amount);

}

}

**// Main.java**

package Week1.DesignPatterns.AdapterPatternExample;

public class Main {

public static void main(String[] args) {

PaymentProcessor paypalProcessor = new PayPalAdapter();

paypalProcessor.processPayment(500.00);

PaymentProcessor stripeProcessor = new StripeAdapter();

stripeProcessor.processPayment(1000.00);

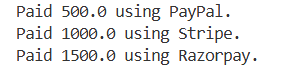
PaymentProcessor razorpayProcessor = new RazorpayAdapter();

razorpayProcessor.processPayment(1500.00);

}

}

**Output:**



1. **Decorator Pattern**

**Code:**

**// Notifier.java**

package Week1.DesignPatterns.DecoratorPatternExample;

public interface Notifier {

void send(String message);

}

**// EmailNotifier.java**

package Week1.DesignPatterns.DecoratorPatternExample;

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

System.out.println("Sending EMAIL with message: " + message);

}

}

**// NotifierDecorator.java**

package Week1.DesignPatterns.DecoratorPatternExample;

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

@Override

public void send(String message) {

notifier.send(message);

}

}

**// SMSNotifierDecorator.java**

package Week1.DesignPatterns.DecoratorPatternExample;

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

private void sendSMS(String message) {

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

}

@Override

public void send(String message) {

super.send(message); // First, send base notifier message

sendSMS(message); // Then, send additional SMS

}

}

**// SlackNotifierDecorator.java**

package Week1.DesignPatterns.DecoratorPatternExample;

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

private void sendSlack(String message) {

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

}

@Override

public void send(String message) {

super.send(message); // Base + extended

sendSlack(message);

}

}

**// NotificationTest.java**

package Week1.DesignPatterns.DecoratorPatternExample;

public class NotificationTest {

public static void main(String[] args) {

// Base notifier

Notifier emailNotifier = new EmailNotifier();

// Add SMS functionality

Notifier smsEmailNotifier = new SMSNotifierDecorator(emailNotifier);

// Add Slack on top of Email + SMS

Notifier fullNotifier = new SlackNotifierDecorator(smsEmailNotifier);

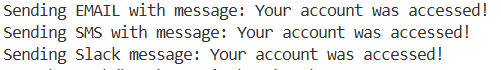
// Send the notification

fullNotifier.send("Your account was accessed!");

}

}

**Output:**



1. **Proxy Pattern**

**Code:**

**// Image.java**

package Week1.DesignPatterns.ProxyPatternExample;

public interface Image {

void display();

}

**// RealImage.java**

package Week1.DesignPatterns.ProxyPatternExample;

public class RealImage implements Image {

private final String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

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

// Simulate a network delay or expensive operation

try {

Thread.sleep(1000); // Simulate loading time

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

@Override

public void display() {

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

}

}

**// ProxyImage.java**

package Week1.DesignPatterns.ProxyPatternExample;

public class ProxyImage implements Image {

private final String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(filename); // Lazy initialization

}

realImage.display(); // Use cached image

}

}

**// ProxyPatternTest.java**

package Week1.DesignPatterns.ProxyPatternExample;

public class ProxyPatternTest {

public static void main(String[] args) {

Image image1 = new ProxyImage("cat\_photo.jpg");

System.out.println("First call to display():");

image1.display(); // Loads and displays

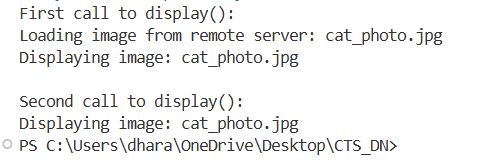
System.out.println("\nSecond call to display():");

image1.display(); // Uses cached image (no loading)

}

}

**Output:**



1. **Observer Pattern**

**Code:**

**// Stock.java**

package Week1.DesignPatterns.ObserverPatternExample;

public interface Stock {

void registerObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObservers();

}

**// StockMarket.java**

package Week1.DesignPatterns.ObserverPatternExample;

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

private String stockSymbol;

private double price;

public StockMarket(String stockSymbol, double initialPrice) {

this.stockSymbol = stockSymbol;

this.price = initialPrice;

}

public void setPrice(double newPrice) {

System.out.println("\nStock Price Updated: " + stockSymbol + " = " + newPrice);

this.price = newPrice;

notifyObservers();

}

public double getPrice() {

return price;

}

public String getStockSymbol() {

return stockSymbol;

}

@Override

public void registerObserver(Observer observer) {

observers.add(observer);

}

@Override

public void removeObserver(Observer observer) {

observers.remove(observer);

}

@Override

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockSymbol, price);

}

}

}

**// Observer.java**

package Week1.DesignPatterns.ObserverPatternExample;

public interface Observer {

void update(String stockSymbol, double newPrice);

}

**// MobileApp.java**

package Week1.DesignPatterns.ObserverPatternExample;

public class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(String stockSymbol, double newPrice) {

System.out.println(name + " (Mobile App) received update: " + stockSymbol + " = " + newPrice);

}

}

**// WebApp.java**

package Week1.DesignPatterns.ObserverPatternExample;

public class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(String stockSymbol, double newPrice) {

System.out.println(name + " (Web App) received update: " + stockSymbol + " = " + newPrice);

}

}

**// ObserverPatternTest.java**

package Week1.DesignPatterns.ObserverPatternExample;

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket teslaStock = new StockMarket("TSLA", 620.0);

Observer mobileClient = new MobileApp("Alice");

Observer webClient = new WebApp("Bob");

teslaStock.registerObserver(mobileClient);

teslaStock.registerObserver(webClient);

teslaStock.setPrice(625.50);

teslaStock.setPrice(630.00);

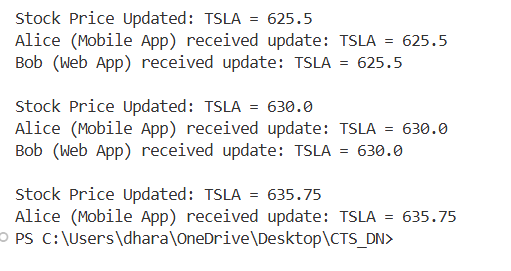
teslaStock.removeObserver(webClient);

teslaStock.setPrice(635.75);

}

}

**Output:**



1. **Strategy Pattern**

**Code:**

**// PaymentStrategy.java**

public interface PaymentStrategy {

void pay(double amount);

}

**// CreditCardPayment.java**

public 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: " + cardNumber + " (Holder: " + cardHolder + ")");

}

}

**// PayPalPayment.java**

public 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);

}

}

**// PaymentContext.java**

public class PaymentContext {

private PaymentStrategy strategy;

// Set the payment strategy dynamically

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

// Execute the selected strategy

public void processPayment(double amount) {

if (strategy == null) {

System.out.println("Payment strategy not set.");

} else {

strategy.pay(amount);

}

}

}

**// StrategyPatternTest.java**

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Pay using Credit Card

PaymentStrategy creditCard = new CreditCardPayment("1234-5678-9876-5432", "Alice");

context.setPaymentStrategy(creditCard);

context.processPayment(2500.00);

// Switch to PayPal

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

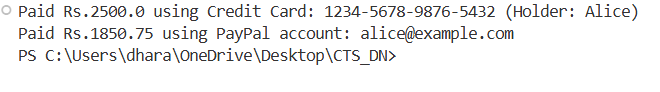
context.setPaymentStrategy(paypal);

context.processPayment(1850.75);

}

}

**Output:**



1. **Command Pattern**

**Code:**

**// Command.java**

public interface Command {

void execute();

}

**// LightOnCommand.java**

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

**// LightOffCommand.java**

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

**// RemoteControl.java**

public class RemoteControl {

private Command command;

// Set the command dynamically

public void setCommand(Command command) {

this.command = command;

}

// Invoke the command

public void pressButton() {

if (command != null) {

command.execute();

} else {

System.out.println("No command set.");

}

}

}

**// Light.java**

public class Light {

private String room;

public Light(String room) {

this.room = room;

}

public void turnOn() {

System.out.println(room + " light is ON");

}

public void turnOff() {

System.out.println(room + " light is OFF");

}

}

**// CommandPatternTest.java**

public class CommandPatternTest {

public static void main(String[] args) {

Light livingRoomLight = new Light("Living Room");

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

// Turn on the light

remote.setCommand(lightOn);

remote.pressButton();

// Turn off the light

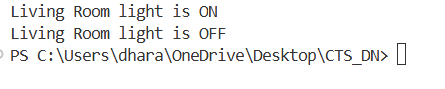
remote.setCommand(lightOff);

remote.pressButton();

}

}

**Output:**



1. **MVC Pattern**

**Code:**

**// Student.java**

public class Student {

private String id;

private String name;

private String grade;

// Constructor

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

this.id = id;

this.name = name;

this.grade = grade;

}

// Getters and setters

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

**// StudentView.java**

public class StudentView {

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

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

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

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

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

System.out.println();

}

}

**// StudentController.java**

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

// Setters

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

// Getters

public String getStudentName() {

return model.getName();

}

public String getStudentId() {

return model.getId();

}

public String getStudentGrade() {

return model.getGrade();

}

// Update the view

public void updateView() {

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

}

}

**// MVCPatternTest.java**

public class MVCPatternTest {

public static void main(String[] args) {

// Create model

Student student = new Student("101", "Alice", "A");

// Create view

StudentView view = new StudentView();

// Create controller

StudentController controller = new StudentController(student, view);

// Display initial student details

controller.updateView();

// Update student details

controller.setStudentName("Alicia");

controller.setStudentGrade("A+");

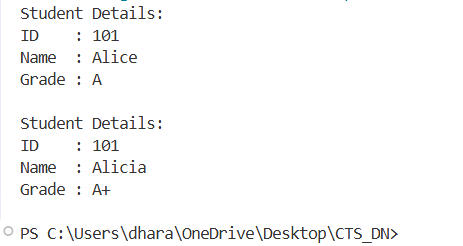
// Display updated details

controller.updateView();

}

}

**Output:**



1. **Dependency Injection**

**Code:**

**// CustomerRepository.java**

public interface CustomerRepository {

Customer findCustomerById(String customerId);

}

**// CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public Customer findCustomerById(String customerId) {

// In real apps, this would access a database

return new Customer(customerId, "Alice", "alice@example.com");

}

}

**// Customer.java**

public class Customer {

private String id;

private String name;

private String email;

public Customer(String id, String name, String email) {

this.id = id;

this.name = name;

this.email = email;

}

// Getters

public String getId() {

return id;

}

public String getName() {

return name;

}

public String getEmail() {

return email;

}

}

**// CustomerService.java**

public class CustomerService {

private final CustomerRepository customerRepository;

// Constructor Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void showCustomerDetails(String customerId) {

Customer customer = customerRepository.findCustomerById(customerId);

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

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

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

System.out.println("Email : " + customer.getEmail());

}

}

**// DependencyInjectionTest.java**

public class DependencyInjectionTest {

public static void main(String[] args) {

// Manually create dependencies

CustomerRepository repository = new CustomerRepositoryImpl();

// Inject the repository into the service

CustomerService service = new CustomerService(repository);

// Use the service

service.showCustomerDetails("C101");

}

}

**Output:**

