**Exercise 1: Implementing the Singleton Pattern**

* **Logger.java-Singleton Class**

public class Logger {

// Private static instance of Logger

private static Logger *instance*;

// Private constructor to prevent external instantiation

private Logger() {

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

}

// Public method to provide access to the single instance

public static Logger getInstance() {

if (*instance* == null) {

*instance* = new Logger(); // Lazy initialization

}

return *instance*;

}

// Sample logging method

public void log(String message) {

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

}

}

* **SingletonTest.java- Test Class**

// File: SingletonTest.java

public class SingletonTest {

public static void main(String[] args) {

// Retrieve the Logger instance

Logger logger1 = Logger.*getInstance*();

logger1.log("First log message.");

// Try to retrieve another Logger instance

Logger logger2 = Logger.getInstance();

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

// Check if both logger references point to the same object

if (logger1 == logger2) {

System.***out***.println("Both logger1 and logger2 refer to the same instance.");

} else {

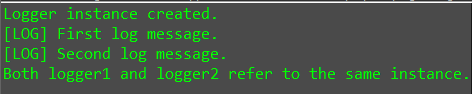
System.***out***.println("Different instances detected!");

}

}

}

**Output**



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

1. **Document.java – Common Interface**

package com.factory.example;

public interface Document {

void open();

}

1. **Concrete Document Classes** 
   * WordDocument.java

package com.factory.example;

public class WordDocument implements Document {

@Override

public void open() {

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

}

}

* + PdfDocument.java

package com.factory.example;

public class PdfDocument implements Document {

@Override

public void open() {

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

}

}

* + ExcelDocument.java

package com.factory.example;

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

1. **Abstract Factory Class – DocumentFactory.java**

package com.factory.example;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

1. **Concrete Factory Classes**
   * WordFactory.java

package com.factory.example;

public class WordFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

* + PdfFactory.java

package com.factory.example;

public class PdfFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

* + ExcelFactory.java

package com.factory.example;

public class ExcelFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

1. **Test Class – FactoryPatternTest.java**

package com.factory.example;

public class FactoryPatternTest {

public static void main(String[] args) {

// Word document factory

DocumentFactory wordFactory = new WordFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

// PDF document factory

DocumentFactory pdfFactory = new PdfFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

// Excel document factory

DocumentFactory excelFactory = new ExcelFactory();

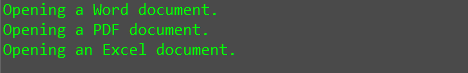
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Output (in Eclipse Console):**

****

**Exercise 3: Implementing the Builder Pattern**

1. **Computer.java – Product + Builder**

package com.builder.example;

public class Computer {

// Required parameters

private String cpu;

private String ram;

// Optional parameters

private String storage;

private String graphicsCard;

private String keyboard;

private String monitor;

// Private constructor

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.keyboard = builder.keyboard;

this.monitor = builder.monitor;

}

// Static nested Builder class

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

private String keyboard;

private String monitor;

// Builder constructor with required params

public Builder(String cpu, String ram) {

this.cpu = cpu;

this.ram = ram;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder setKeyboard(String keyboard) {

this.keyboard = keyboard;

return this;

}

public Builder setMonitor(String monitor) {

this.monitor = monitor;

return this;

}

public Computer build() {

return new Computer(this);

}

}

public void displayConfiguration() {

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

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

System.out.println("Storage: " + (storage != null ? storage : "Not Included"));

System.out.println("Graphics Card: " + (graphicsCard != null ? graphicsCard : "Not Included"));

System.out.println("Keyboard: " + (keyboard != null ? keyboard : "Not Included"));

System.out.println("Monitor: " + (monitor != null ? monitor : "Not Included"));

System.out.println("--------------------------------------------------");

}

}

1. **BuilderPatternTest.java – Test Class**

package com.builder.example;

public class BuilderPatternTest {

public static void main(String[] args) {

// Basic configuration

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

.build();

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

basicComputer.displayConfiguration();

// Advanced configuration

Computer gamingComputer = new Computer.Builder("AMD Ryzen 9", "32GB")

.setGraphicsCard("NVIDIA RTX 4080")

.setStorage("2TB SSD")

.setKeyboard("Mechanical RGB Keyboard")

.setMonitor("32-inch 4K Monitor")

.build();

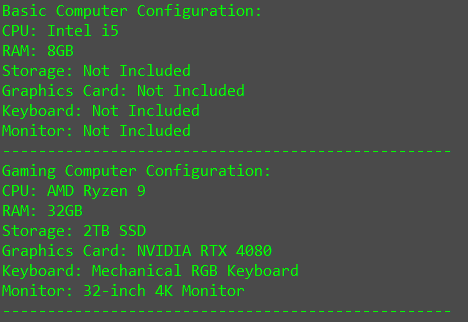
System.out.println("Gaming Computer Configuration:");

gamingComputer.displayConfiguration();

}

}

**Output**



**Exercise 4: Implementing the Adapter Pattern**

1. **PaymentProcessor.java – Target Interface**

package com.adapter.example;

public interface PaymentProcessor {

void processPayment(double amount);

}

1. **Adaptee Classes (Third-party payment gateways)**
   * PayPalGateway.java

package com.adapter.example;

public class PayPalGateway {

public void makePayment(double amount) {

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

}

}

* + StripeGateway.java

package com.adapter.example;

public class StripeGateway {

public void chargeCard(double amount) {

System.out.println("Charging ₹" + amount + " via Stripe.");

}

}

1. **Adapter Classes (Convert to Target Interface)**
   * PayPalAdapter.java

package com.adapter.example;

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter() {

this.paypal = new PayPalGateway();

}

@Override

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

* + StripeAdapter.java

package com.adapter.example;

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter() {

this.stripe = new StripeGateway();

}

@Override

public void processPayment(double amount) {

stripe.chargeCard(amount);

}

}

1. **AdapterPatternTest.java – Test Class**

package com.adapter.example;

public class AdapterPatternTest {

public static void main(String[] args) {

// Using PayPal through adapter

PaymentProcessor paypalProcessor = new PayPalAdapter();

paypalProcessor.processPayment(1500.00);

// Using Stripe through adapter

PaymentProcessor stripeProcessor = new StripeAdapter();

stripeProcessor.processPayment(2500.00);

}

}

**Output**

****

**Exercise 5: Implementing the Decorator Pattern**

1. **Notifier.java – Component Interface**

package com.decorator.example;

public interface Notifier {

void send(String message);

}

1. **EmailNotifier.java – Concrete Component**

package com.decorator.example;

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

1. **NotifierDecorator.java – Abstract Decorator**

package com.decorator.example;

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

@Override

public void send(String message) {

wrappedNotifier.send(message); // Delegate to wrapped notifier

}

}

1. **Concrete Decorators**
   * SMSNotifierDecorator.java

package com.decorator.example;

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message); // Send base notification

sendSMS(message); // Add SMS functionality

}

private void sendSMS(String message) {

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

}

}

* + SlackNotifierDecorator.java

package com.decorator.example;

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message); // Send previous notifications

sendSlack(message); // Add Slack functionality

}

private void sendSlack(String message) {

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

}

}

1. **DecoratorPatternTest.java – Test Class**

package com.decorator.example;

public class DecoratorPatternTest {

public static void main(String[] args) {

// Base notifier: Email

Notifier emailNotifier = new EmailNotifier();

// Add SMS functionality

Notifier smsEmailNotifier = new SMSNotifierDecorator(emailNotifier);

// Add Slack on top of Email + SMS

Notifier fullNotifier = new SlackNotifierDecorator(smsEmailNotifier);

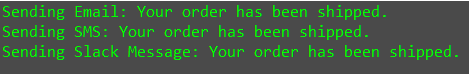
// Send notification through all channels

fullNotifier.send("Your order has been shipped.");

}

}

**Output**

****

**Exercise 6: Implementing the Proxy Pattern**

1. **Image.java – Subject Interface**

package com.proxy.example;

public interface Image {

void display();

}

1. **RealImage.java – Real Subject**

package com.proxy.example;

public 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);

}

@Override

public void display() {

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

}

}

1. **ProxyImage.java – Proxy Class with Lazy Loading & Caching**

package com.proxy.example;

public class ProxyImage implements Image {

private 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(); // Delegates to RealImage

}

}

1. **ProxyPatternTest.java – Test Class**

package com.proxy.example;

public class ProxyPatternTest {

public static void main(String[] args) {

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

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

// Image will be loaded from remote server

image1.display();

// Image is cached; real image won't be loaded again

image1.display();

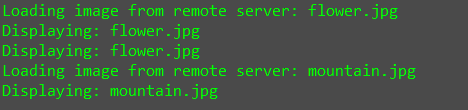
// Another image is lazily loaded

image2.display();

}

}

**Output**

****

**Exercise 7: Implementing the Observer Pattern**

1. **Stock.java – Subject Interface**

package com.observer.example;

public interface Stock {

void registerObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObservers();

}

1. **Observer.java – Observer Interface**

package com.observer.example;

public interface Observer {

void update(String stockName, double price);

}

1. **StockMarket.java – Concrete Subject**

package com.observer.example;

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

private String stockName;

private double stockPrice;

private List<Observer> observers;

public StockMarket(String stockName) {

this.stockName = stockName;

this.observers = new ArrayList<>();

}

@Override

public void registerObserver(Observer observer) {

observers.add(observer);

}

@Override

public void removeObserver(Observer observer) {

observers.remove(observer);

}

@Override

public void notifyObservers() {

for (Observer obs : observers) {

obs.update(stockName, stockPrice);

}

}

// Method to simulate price update

public void setPrice(double newPrice) {

System.out.println("\nUpdating price of " + stockName + " to ₹" + newPrice);

this.stockPrice = newPrice;

notifyObservers();

}

}

1. **MobileApp.java – Concrete Observer**

package com.observer.example;

public class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

@Override

public void update(String stockName, double price) {

System.out.println("[" + appName + " - Mobile] " + stockName + " updated to ₹" + price);

}

}

1. **WebApp.java – Concrete Observer**

package com.observer.example;

public class WebApp implements Observer {

private String appName;

public WebApp(String appName) {

this.appName = appName;

}

@Override

public void update(String stockName, double price) {

System.out.println("[" + appName + " - Web] " + stockName + " updated to ₹" + price);

}

}

1. **ObserverPatternTest.java – Test Class**

package com.observer.example;

public class ObserverPatternTest {

public static void main(String[] args) {

// Create stock subject

StockMarket tcsStock = new StockMarket("TCS");

// Create observers

Observer mobileApp = new MobileApp("GrowFast");

Observer webApp = new WebApp("GrowFast");

// Register observers

tcsStock.registerObserver(mobileApp);

tcsStock.registerObserver(webApp);

// Simulate stock price updates

tcsStock.setPrice(3545.50);

tcsStock.setPrice(3590.00);

// Unsubscribe web app

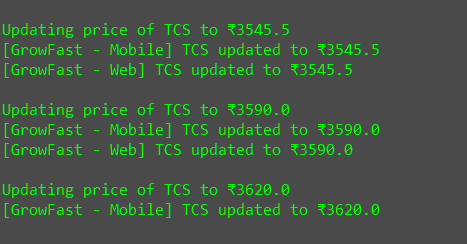
tcsStock.removeObserver(webApp);

tcsStock.setPrice(3620.00);

}

}

**Ouput**

****

**Exercise 8: Implementing the Strategy Pattern**

1. **PaymentStrategy.java – Strategy Interface**

package com.strategy.example;

public interface PaymentStrategy {

void pay(double amount);

}

1. **CreditCardPayment.java – Concrete Strategy**

package com.strategy.example;

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 (" + cardHolder + ").");

}

}

1. **PayPalPayment.java – Concrete Strategy**

package com.strategy.example;

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 (" + email + ").");

}

}

1. **PaymentContext.java – Context Class**

package com.strategy.example;

public class PaymentContext {

private PaymentStrategy paymentStrategy;

// Set the payment method at runtime

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void processPayment(double amount) {

if (paymentStrategy == null) {

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

} else {

paymentStrategy.pay(amount);

}

}

}

1. **StrategyPatternTest.java – Test Class**

package com.strategy.example;

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Using Credit Card

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456", "Vinayak P."));

context.processPayment(499.99);

// Switch to PayPal

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

context.processPayment(1499.00);

// No payment method selected

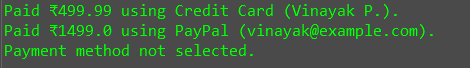
context.setPaymentStrategy(null);

context.processPayment(300.00);

}

}

**Output**

****

**Exercise 9: Implementing the Command Pattern**

1. **Command.java – Command Interface**

package com.command.example;

public interface Command {

void execute();

}

1. **Light.java – Receiver**

package com.command.example;

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

1. **LightOnCommand.java – Concrete Command**

package com.command.example;

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

1. **LightOffCommand.java – Concrete Command**

package com.command.example;

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

1. **RemoteControl.java – Invoker**

package com.command.example;

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

if (command != null) {

command.execute();

} else {

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

}

}

}

1. **CommandPatternTest.java – Test Class**

package com.command.example;

public class CommandPatternTest {

public static void main(String[] args) {

Light livingRoomLight = new Light(); // Receiver

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl(); // Invoker

// Turn the light ON

remote.setCommand(lightOn);

remote.pressButton();

// Turn the light OFF

remote.setCommand(lightOff);

remote.pressButton();

}

}

**Output**

****

**Exercise 10: Implementing the MVC Pattern**

1. **Student.java – Model**

package com.mvc.example;

public class Student {

private String id;

private String name;

private String grade;

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;

}

}

1. **StudentView.java – View**

package com.mvc.example;

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("----------------------------");

}

}

1. **StudentController.java – Controller**

package com.mvc.example;

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 setStudentGrade(String grade) {

model.setGrade(grade);

}

public void updateView() {

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

}

}

1. **MVCPatternTest.java – Test / Main Class**

package com.mvc.example;

public class MVCPatternTest {

public static void main(String[] args) {

// Create model

Student student = new Student("S101", "Vinayak", "A");

// Create view

StudentView view = new StudentView();

// Create controller

StudentController controller = new StudentController(student, view);

// Display initial data

controller.updateView();

// Modify student details

controller.setStudentName("Vinayak P.");

controller.setStudentGrade("A+");

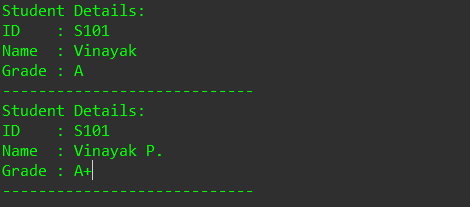
// Display updated data

controller.updateView();

}

}

**Output**



**Exercise 11: Implementing Dependency Injection**

1. **Customer.java – POJO class**

package com.di.example;

public class Customer {

private String id;

private String name;

public Customer(String id, String name) {

this.id = id;

this.name = name;

}

// Getters

public String getId() {

return id;

}

public String getName() {

return name;

}

}

1. **CustomerRepository.java – Repository Interface**

package com.di.example;

public interface CustomerRepository {

Customer findCustomerById(String id);

}

1. **CustomerRepositoryImpl.java – Concrete Repository**

package com.di.example;

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public Customer findCustomerById(String id) {

// Dummy data for example

return new Customer(id, "Vinayak Puitandy");

}

}

1. **CustomerService.java – Service Class (uses DI)**

package com.di.example;

public class CustomerService {

private CustomerRepository customerRepository;

// Constructor-based Dependency Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void getCustomerDetails(String id) {

Customer customer = customerRepository.findCustomerById(id);

if (customer != null) {

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

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

} else {

System.out.println("Customer not found.");

}

}

}

**5. DIExampleTest.java – Main/Test Class**

package com.di.example;

public class DIExampleTest {

public static void main(String[] args) {

// Create repository implementation

CustomerRepository repo = new CustomerRepositoryImpl();

// Inject dependency into service via constructor

CustomerService service = new CustomerService(repo);

// Use the service

service.getCustomerDetails("C101");

}

}

**Output**

****