**Exercise 1: Implementing the Singleton Pattern**

**Code:**

package basic;

// Step 2: Define a Singleton Class

// Logger class ensures only one instance exists across the application

class Logger {

    // Step 3: Implement Singleton Pattern

    // Static instance created eagerly (eager initialization)

    private static Logger instance = new Logger();

    // Private constructor prevents external instantiation

    private Logger() {

        System.out.println("Singleton Logger Class Instantiated");

    }

    // Public static method to return the single instance

    public static Logger getInstance() {

        return instance;

    }

    // Method to simulate logging functionality

    public void log(String message) {

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

    }

}

// Step 4: Test the Singleton Implementation

public class LoggerTest {

    public static void main(String[] args) {

        // Get Logger instance using getInstance()

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        // Logging using both references

        logger1.log("Log message of Logger1");

        logger2.log("Log message of Logger2");

        // Check if both references point to the same object

        if (logger1 == logger2) {

            System.out.println("Both instances are same, Logger is a Singleton Pattern Class");

        } else {

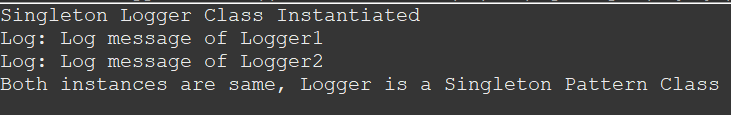
            System.out.println("Both instances are not same, Logger is not a Singleton Pattern Class");

        }

    }

}

**Output:**

****

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

**Code:**

package basic;

// Step 2: Define Document Interface

// This interface will be implemented by all document types

interface Document {

public void open();

}

// Step 3: Create Concrete Document Classes

// Concrete class for Word document

class WordDocument implements Document {

public WordDocument() {

System.***out***.println("Word Document Created.");

}

*@Override*

public void open() {

System.***out***.println("Opening Word Document...");

}

}

// Concrete class for PDF document

class PdfDocument implements Document {

public PdfDocument() {

System.***out***.println("Pdf Document Created.");

}

*@Override*

public void open() {

System.***out***.println("Opening Pdf Document...");

}

}

// Concrete class for Excel document

class ExcelDocument implements Document {

public ExcelDocument() {

System.***out***.println("Excel Document Created.");

}

*@Override*

public void open() {

System.***out***.println("Opening Excel Document...");

}

}

// Step 4: Implement the Factory Method

// Abstract factory class with a factory method to create documents

abstract class DocumentFactory {

public abstract Document createDocument();

}

// Factory for creating Word documents

class WordDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

return new WordDocument();

}

}

// Factory for creating PDF documents

class PdfDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

return new PdfDocument();

}

}

// Factory for creating Excel documents

class ExcelDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

return new ExcelDocument();

}

}

// Step 5: Test the Factory Method Implementation

public class DocumentFactoryTest {

public static void main(String[] args) {

// Create and open a Word document using its factory

DocumentFactory wordFactory = new WordDocumentFactory();

Document word = wordFactory.createDocument();

word.open();

// Create and open a PDF document using its factory

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

// Create and open an Excel document using its factory

DocumentFactory excelFactory = new ExcelDocumentFactory();

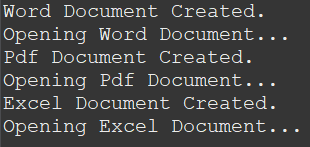
Document excel = excelFactory.createDocument();

excel.open();

}

}

**Output:**



**Exercise 3: Implementing the Builder Pattern**

**Code:**

package basic;

// Step 2: Define a Product Class

// Computer class represents the product being built

class Computer {

private String cpu, ram, storage;

// Step 4: Implement Builder Pattern

// Private constructor accepts a builder object to set attributes

public Computer(ComputerBuilder computerBuilder) {

this.cpu = computerBuilder.cpu;

this.ram = computerBuilder.ram;

this.storage = computerBuilder.storage;

}

// Step 3: Implement the Builder Class

// Static nested class - the Builder

static class ComputerBuilder {

// Required and optional fields

private String cpu; // required

private String ram; // optional

private String storage; // optional

// Constructor with required field

public ComputerBuilder(String cpu) {

this.cpu = cpu;

}

// Setter method for RAM

public ComputerBuilder setRam(String ram) {

this.ram = ram;

return this; // return builder for chaining

}

// Setter method for Storage

public ComputerBuilder setStorage(String storage) {

this.storage = storage;

return this; // return builder for chaining

}

// build() method creates the final Computer object

public Computer build() {

return new Computer(this);

}

}

// Getters for Computer attributes

public String getCpu() {

return cpu;

}

public String getRam() {

return ram;

}

public String getStorage() {

return storage;

}

}

// Step 5: Test the Builder Implementation

public class ComputerTest {

public static void main(String[] args) {

// Build first computer configuration using builder

Computer computer1 = new Computer.ComputerBuilder("Intel Core i5 13th Gen").setRam("8GB").setStorage("1TB").build();

System.***out***.println("Computer1 Configuration:\nCPU: " + computer1.getCpu() + "\nRAM: " + computer1.getRam() + "\nStorage: " + computer1.getStorage());

// Build second computer configuration using builder

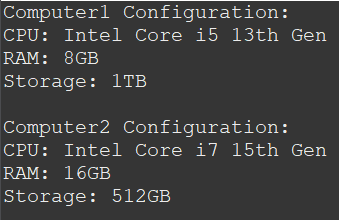
Computer computer2 = new Computer.ComputerBuilder("Intel Core i7 15th Gen").setRam("16GB").setStorage("512GB").build();

System.***out***.println("\nComputer2 Configuration:\nCPU: " + computer2.getCpu() + "\nRAM: " + computer2.getRam() + "\nStorage: " + computer2.getStorage());

}

}

**Output:**

****

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

package basic;

// Step 2: Define the Target Interface

// All payment processors should implement this common interface

interface PaymentProcessor {

public void processPayment(); // Unified method expected by the system

}

// Step 3: Implement Adaptee Classes

// These are existing or third-party payment gateway classes with their own method names

class CreditPaymentProcessor {

public void processCreditPayment() {

System.***out***.println("Processing Credit Payment...");

}

}

class DebitPaymentProcessor {

public void processDebitPayment() {

System.***out***.println("Processing Debit Payment...");

}

}

class RazorPayPaymentProcessor {

public void processRazorPayPayment() {

System.***out***.println("Processing RazorPay Payment...");

}

}

// Step 4: Implement Adapter Classes

// Each adapter implements the Target Interface and wraps an adaptee class

// Adapter for CreditPaymentProcessor

class CreditPaymentAdapter implements PaymentProcessor {

private CreditPaymentProcessor creditPaymentProcessor;

public CreditPaymentAdapter(CreditPaymentProcessor creditPaymentProcessor) {

this.creditPaymentProcessor = creditPaymentProcessor;

}

*@Override*

public void processPayment() {

System.***out***.println("Credit Payment Adapter called");

creditPaymentProcessor.processCreditPayment();

}

}

// Adapter for DebitPaymentProcessor

class DebitPaymentAdapter implements PaymentProcessor {

private DebitPaymentProcessor debitPaymentProcessor;

public DebitPaymentAdapter(DebitPaymentProcessor debitPaymentProcessor) {

this.debitPaymentProcessor = debitPaymentProcessor;

}

*@Override*

public void processPayment() {

System.***out***.println("Debit Payment Adapter called");

debitPaymentProcessor.processDebitPayment();

}

}

// Adapter for RazorPayPaymentProcessor

class RazorPayPaymentAdapter implements PaymentProcessor {

private RazorPayPaymentProcessor razorPayPaymentProcessor;

public RazorPayPaymentAdapter(RazorPayPaymentProcessor razorPayPaymentProcessor) {

this.razorPayPaymentProcessor = razorPayPaymentProcessor;

}

*@Override*

public void processPayment() {

System.***out***.println("Razor Pay Payment Adapter called");

razorPayPaymentProcessor.processRazorPayPayment();

}

}

// Step 5: Test the Adapter Implementation

public class PaymentProcessorTest {

public static void main(String[] args) {

// Using RazorPay via adapter

PaymentProcessor razorPay = new RazorPayPaymentAdapter(new RazorPayPaymentProcessor());

razorPay.processPayment();

// Using Credit Card Payment via adapter

PaymentProcessor credit = new CreditPaymentAdapter(new CreditPaymentProcessor());

credit.processPayment();

// Using Debit Card Payment via adapter

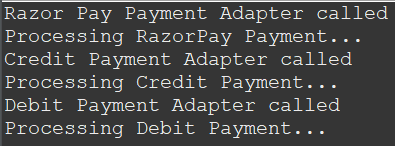
PaymentProcessor debit = new DebitPaymentAdapter(new DebitPaymentProcessor());

debit.processPayment();

}

}

**Output:**



**Exercise 5: Implementing the Decorator Pattern**

**Code:**

package basic;

// Step 2: Define Component Interface

// This interface defines a common method for all notifiers.

interface Notifier {

public void send();

}

// Step 3: Implement Concrete Component

// A basic implementation of Notifier that sends email notifications.

class EmailNotifier implements Notifier {

public void send() {

System.***out***.println("Sending Email...");

}

}

// Step 4: Implement Decorator Classes

// Abstract Decorator Class

// This class implements Notifier and holds a reference to a Notifier object.

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier; // Holds the component to be decorated

// Force subclasses to implement the send method

public abstract void send();

}

// Concrete Decorator 1: Adds SMS functionality

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super.notifier = notifier; // Set the component to wrap

}

*@Override*

public void send() {

System.***out***.println("Sending from SMS Notifier");

notifier.send(); // Delegate to the wrapped notifier

}

}

// Concrete Decorator 2: Adds Slack functionality

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super.notifier = notifier;

}

*@Override*

public void send() {

System.***out***.println("Sending from Slack Notifier");

notifier.send(); // Delegate to the wrapped notifier

}

}

// Step 5: Test the Decorator Implementation

public class NotifierDecoratorTest {

public static void main(String[] args) {

// Sending notifications via SMS and then Email

System.***out***.println("=== SMS + Email Notification ===");

NotifierDecorator smsNotifierDecorator = new SMSNotifierDecorator(new EmailNotifier());

smsNotifierDecorator.send();

// Sending notifications via Slack and then Email

System.***out***.println("\n=== Slack + Email Notification ===");

NotifierDecorator slackNotifierDecorator = new SlackNotifierDecorator(new EmailNotifier());

slackNotifierDecorator.send();

// Composing multiple decorators: Slack -> SMS -> Email

System.***out***.println("\n=== Slack + SMS + Email Notification ===");

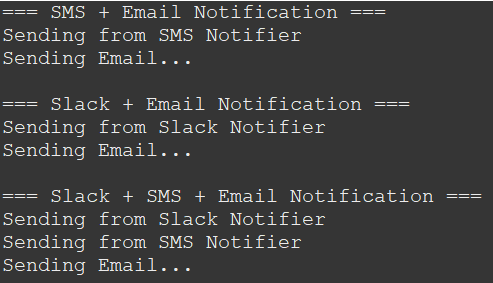
Notifier multiChannelNotifier = new SlackNotifierDecorator( new SMSNotifierDecorator( new EmailNotifier()));

multiChannelNotifier.send();

}

}

**Output:**



**Exercise 6: Implementing the Proxy Pattern**

**Code:**

package basic;

// Step 2: Define Subject Interface

// Common interface for RealImage and ProxyImage

interface Image {

void display();

}

// Step 3: Implement Real Subject Class

// Loads the actual image (simulating heavy operation)

class RealImage implements Image {

private String filename;

// Constructor performs the heavy loading operation

public RealImage(String filename) {

this.filename = filename;

loadImageFromDisk(); // Expensive operation

}

// Simulate image loading from disk or remote server

private void loadImageFromDisk() {

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

}

// Displays the image

public void display() {

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

}

}

// Step 4: Implement Proxy Class

// Controls access to RealImage and adds lazy loading and caching

class ProxyImage implements Image {

private RealImage realImage; // Reference to the real object

private String filename;

// Constructor just stores the filename, does not load image yet

public ProxyImage(String filename) {

this.filename = filename;

}

// Display method with lazy initialization

public void display() {

if (realImage == null) {

// RealImage is only created when needed (lazy loading)

realImage = new RealImage(filename);

System.***out***.println("Loaded Image");

} else {

// Image already loaded, use cached version

System.***out***.println("Loading from Cache");

}

realImage.display(); // Delegate display to RealImage

}

}

// Step 5: Test the Proxy Implementation

public class ProxyTest {

public static void main(String[] args) {

// Using ProxyImage instead of RealImage directly

Image image = new ProxyImage("example.jpg");

// First call: Image is loaded and displayed

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

image.display();

// Second call: Image is not loaded again, uses cache

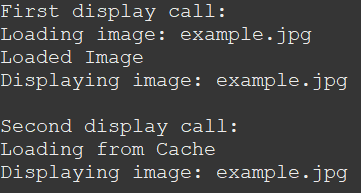
System.***out***.println("\nSecond display call:");

image.display();

}

}

**Output:**

****

**Exercise 7: Implementing the Observer Pattern**

**Code:**

package basic;

import java.util.ArrayList;

// Step 2: Define Subject Interface

// Represents the observable (Stock Market) that allows observers to register, deregister, and get notified.

interface Stock {

void register(Observer observer); // Register an observer

void deRegister(Observer observer); // Remove an observer

void notifyObservers(String s); // Notify all observers with a message

}

// Step 3: Implement Concrete Subject

// Implements the Stock interface and maintains a list of registered observers

class StockMarket implements Stock {

private ArrayList<Observer> list = new ArrayList<Observer>();

// Add a new observer to the list

public void register(Observer observer) {

list.add(observer);

System.***out***.println("Observer " + observer.getName() + " added");

}

// Remove an observer from the list

public void deRegister(Observer observer) {

list.remove(observer);

System.***out***.println("Observer " + observer.getName() + " removed");

}

// Notify all registered observers with a message

public void notifyObservers(String message) {

for (Observer observer : list) {

observer.update(message);

}

}

}

// Step 4: Define Observer Interface

// Represents the observer which needs to be notified of updates

interface Observer {

String getName(); // Return observer name

void update(String message); // Update method called when subject notifies

}

// Step 5: Implement Concrete Observers

// MobileApp observer implementation

class MobileApp implements Observer {

String name;

public MobileApp(String name) {

this.name = name;

}

public String getName() {

return name;

}

// Displays the received notification

public void update(String message) {

System.***out***.println("Notification received to " + name + " : " + message);

}

}

// WebApp observer implementation

class WebApp implements Observer {

String name;

public WebApp(String name) {

this.name = name;

}

public String getName() {

return name;

}

// Displays the received notification

public void update(String message) {

System.***out***.println("Notification received to " + name + " : " + message);

}

}

// Step 6: Test the Observer Implementation

public class Observertest {

public static void main(String args[]) {

// Create observers

Observer mobileApp = new MobileApp("MobileAppObserver1");

Observer webApp = new WebApp("WebAppObserver1");

// Create the subject (StockMarket)

Stock stockMarket = new StockMarket();

// Register observers to the stock market

stockMarket.register(mobileApp);

stockMarket.register(webApp);

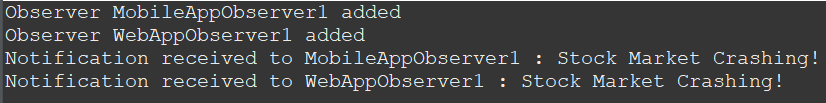
// Notify all observers of a stock market update

stockMarket.notifyObservers("Stock Market Crashing!");

}

}

**Output:**

****

**Exercise 8: Implementing the Strategy Pattern**

**Code:**

package basic;

// Step 2: Define Strategy Interface

// This interface declares the pay() method all payment strategies must implement

interface PaymentStrategy {

void pay();

}

// Step 3: Implement Concrete Strategies

// A concrete strategy for Credit Card payment

class CreditCardPayment implements PaymentStrategy {

public void pay() {

System.***out***.println("Paying with Credit Card...");

}

}

// A concrete strategy for PayPal payment

class PayPalPayment implements PaymentStrategy {

public void pay() {

System.***out***.println("Paying with PayPal...");

}

}

// Step 4: Implement Context Class

// This class maintains a reference to the PaymentStrategy and uses it to process payments

class PaymentContext {

PaymentStrategy paymentStrategy;

// Method to change the strategy at runtime

void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

// Executes the selected payment strategy

void pay() {

paymentStrategy.pay();

}

}

// Step 5: Test the Strategy Implementation

public class StartegyPatterTest {

public static void main(String[] args) {

PaymentContext paymentContext = new PaymentContext();

// Use CreditCardPayment strategy

paymentContext.setPaymentStrategy(new CreditCardPayment());

paymentContext.pay();

// Switch to PayPalPayment strategy

paymentContext.setPaymentStrategy(new PayPalPayment());

paymentContext.pay();

}

}

**Output:**



**Exercise 9: Implementing the Command Pattern**

**Code:**

package basic;

// Step 2: Define Command Interface

interface Command {

void execute();

}

// Step 5: Implement Receiver Class

// The actual object that performs the action

class Light {

public void on() {

System.***out***.println("Light turned on");

}

public void off() {

System.***out***.println("Light turned off");

}

}

// Step 3: Implement Concrete Commands

// Each command binds a receiver (Light) and calls one of its methods

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();

}

}

// Step 4: Implement Invoker Class

// The invoker just knows how to execute the command

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

// Step 6: Test the Command Implementation

public class CommandPatternTest {

public static void main(String[] args) {

// Create receiver

Light livingRoomLight = new Light();

// Create command objects, binding them to the receiver

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

// Create invoker

RemoteControl remote = new RemoteControl();

// Turn on the light

remote.setCommand(lightOn);

remote.pressButton();

// Turn off the light

remote.setCommand(lightOff);

remote.pressButton();

}

}

**Output:**



**Exercise 10: Implementing the MVC Pattern**

**Code:**

package basic;

// Model: Represents the student entity with properties and getters/setters

class Student {

private String name, id, grade;

// Constructor to initialize student details

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

this.name = name;

this.id = id;

this.grade = grade;

}

// Getter and Setter methods for student attributes

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;

}

}

// View: Responsible for displaying student information to the console (UI layer)

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);

}

}

// Controller: Acts as a mediator between the Model and View

class StudentController {

Student student; // Model

StudentView studentView; // View

// Constructor to bind model and view

public StudentController(Student student, StudentView studentView) {

this.student = student;

this.studentView = studentView;

}

// Methods to update model data through the controller

public String getName() {

return student.getName();

}

public void setName(String name) {

student.setName(name);

}

public String getId() {

return student.getId();

}

public void setId(String id) {

student.setId(id);

}

public String getGrade() {

return student.getGrade();

}

public void setGrade(String grade) {

student.setGrade(grade);

}

// Updates the view with the current state of the model

public void updateView() {

studentView.displayStudentDetails(student.getName(), student.getId(), student.getGrade());

}

}

// Test class to demonstrate the MVC pattern in action

public class MVCTest {

public static void main(String[] args) {

// Create model object

Student student = new Student("Ram", "22f24f2", "A+");

// Create view object

StudentView studentView = new StudentView();

// Create controller and associate it with model and view

StudentController studentController = new StudentController(student, studentView);

// Display initial student details

studentController.updateView();

// Update student name via controller

studentController.setName("Ram Nallan");

System.***out***.println("\nAfter Update:\n");

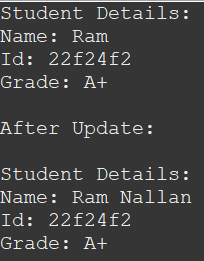
// Display updated student details

studentController.updateView();

}

}

**Output:**



**Exercise 11: Implementing Dependency Injection**

**Code:**

package basic;

// Repository interface to abstract customer data access

interface CustomerRepository {

// Method to find a customer by their ID

Customer findCustomerById(String customerId);

}

// Concrete implementation of the CustomerRepository interface

class CustomerRepositoryImpl implements CustomerRepository {

*@Override*

public Customer findCustomerById(String customerId) {

// In a real application, data would come from a database

// Here we're returning a dummy customer for demonstration

return new Customer(customerId, "Ram", "ram@gmail.com");

}

}

// Model class representing a customer

class Customer {

private String id;

private String name;

private String email;

// Constructor to initialize customer object

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;

}

}

// Service class that depends on the CustomerRepository

class CustomerService {

private CustomerRepository customerRepository;

// Constructor Injection - dependency is injected through the constructor

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

// Method to retrieve and display customer information

public void displayCustomer(String customerId) {

Customer customer = customerRepository.findCustomerById(customerId);

// Display customer details

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

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

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

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

}

}

// Main class to test the Dependency Injection implementation

public class Main {

public static void main(String[] args) {

// Create repository implementation (dependency)

CustomerRepository repository = new CustomerRepositoryImpl();

// Inject the repository into the service class

CustomerService service = new CustomerService(repository);

// Use the service to display a customer by ID

service.displayCustomer("C101");

}

}

**Output:**

