**Exercise 1: Implementing the Singleton Pattern**

package package1;

public class SingletonDemo {

static 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 static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

logger1.log("App started.");

Logger logger2 = Logger.*getInstance*();

logger2.log("User logged in.");

if (logger1 == logger2) {

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

}

}

}

**Output**

|  |
| --- |
|  |

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

package package1;

public class Factorypatterndemo {

interface Document {

void open();

}

static class WordDocument implements Document {

public void open() {

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

}

}

static class PdfDocument implements Document {

public void open() {

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

}

}

static class ExcelDocument implements Document {

public void open() {

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

}

}

static abstract class DocumentFactory {

public abstract Document createDocument();

}

static class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

static class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

static class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open(); // Output: Opening Word document.

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open(); // Output: Opening PDF document.

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Output:**

|  |
| --- |
|  |

**Exercise 3: Implementing the Builder Pattern**

package package1;

public class BuilderDemo {

static class Computer {

private final String cpu;

private final int ram;

private final int storage;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

}

static class Builder {

private String cpu;

private int ram;

private int storage;

public Builder setCpu(String cpu) {

this.cpu = cpu;

return this;

}

public Builder setRam(int ram) {

this.ram = ram;

return this;

}

public Builder setStorage(int storage) {

this.storage = storage;

return this;

}

public Computer build() {

return new Computer(this);

}

}

public void displayConfig() {

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

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

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

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

}

}

public static void main(String[] args) {

Computer computer = new Computer.Builder()

.setCpu("Intel i5")

.setRam(8)

.setStorage(512)

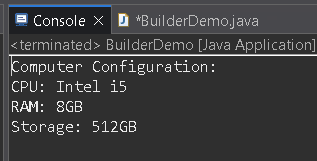
.build();

computer.displayConfig();

}

}

**Output:**



**Exercise 4: Implementing the Adapter Pattern**

package package1;

public class AdapterDemo {

interface PaymentProcessor {

void processPayment(double amount);

}

static class PayPalGateway {

public void makePayment(double amount) {

System.***out***.println("Payment of ₹" + amount + " processed using PayPal.");

}

}

static class StripeGateway {

public void executeTransaction(double amount) {

System.***out***.println("Payment of ₹" + amount + " processed using Stripe.");

}

}

static class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPal;

public PayPalAdapter() {

this.payPal = new PayPalGateway();

}

*@Override*

public void processPayment(double amount) {

payPal.makePayment(amount);

}

}

static class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter() {

this.stripe = new StripeGateway();

}

*@Override*

public void processPayment(double amount) {

stripe.executeTransaction(amount);

}

}

public static void main(String[] args) {

PaymentProcessor payPalProcessor = new PayPalAdapter();

payPalProcessor.processPayment(1500.00);

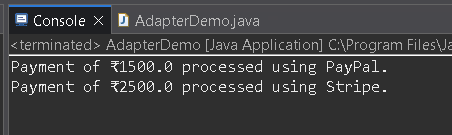
PaymentProcessor stripeProcessor = new StripeAdapter();

stripeProcessor.processPayment(2500.00);

}

}

**Output:**



**Exercise 5: Implementing the Decorator Pattern**

package package1;

public class DecoratorDemo {

interface Notifier {

void send();

}

static class EmailNotifier implements Notifier {

*@Override*

public void send() {

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

}

}

static abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send() {

notifier.send();

}

}

static class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

*@Override*

public void send() {

super.send();

System.***out***.println("Sending SMS Notification.");

}

}

static class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

*@Override*

public void send() {

super.send();

System.***out***.println("Sending Slack Notification.");

}

}

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(notifier);

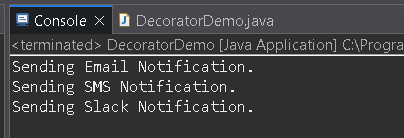
Notifier slackAndSmsNotifier = new SlackNotifierDecorator(smsNotifier);

slackAndSmsNotifier.send();

}

}

**Output:**



**Exercise 6: Implementing the Proxy Pattern**

package package1;

public class ProxyDemo {

interface Image {

void display();

}

static 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);

}

}

static 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);

}

realImage.display();

}

}

public static void main(String[] args) {

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

image1.display();

System.***out***.println();

image1.display();

System.***out***.println();

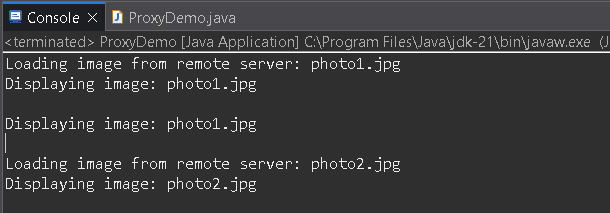
Image image2 = new ProxyImage("photo2.jpg");

image2.display();

}

}

**Output:**



**Exercise 7: Implementing the Observer Pattern**

import java.util.ArrayList;

import java.util.List;

public class ObserverDemo {

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

void setPrice(double price);

}

interface Observer {

void update(double price);

}

static class StockMarket implements Stock {

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

private double price;

public void register(Observer o) {

observers.add(o);

}

public void deregister(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(price);

}

}

public void setPrice(double price) {

this.price = price;

notifyObservers();

}

}

static class MobileApp implements Observer {

public void update(double price) {

System.***out***.println("Mobile App - Stock Price Updated: ₹" + price);

}

}

static class WebApp implements Observer {

public void update(double price) {

System.***out***.println("Web App - Stock Price Updated: ₹" + price);

}

}

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

market.register(mobile);

market.register(web);

market.setPrice(1200.50);

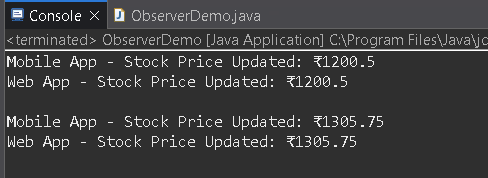
System.***out***.println();

market.setPrice(1305.75);

}

}

**Output:**



**Exercise 8: Implementing the Strategy Pattern**

package package1;

public class StrategyDemo {

interface PaymentStrategy {

void pay(double amount);

}

static class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

System.***out***.println("Paid ₹" + amount + " using Credit Card.");

}

}

static class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

static class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

strategy.pay(amount);

}

}

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment());

context.executePayment(1500.00);

System.***out***.println();

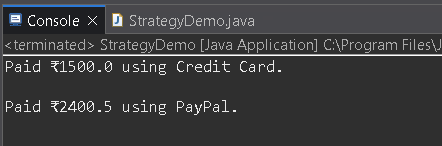
context.setPaymentStrategy(new PayPalPayment());

context.executePayment(2400.50);

}

}

**Output:**



**Exercise 9: Implementing the Command Pattern**

package package1;

public class CommandDemo {

interface Command {

void execute();

}

static class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

static class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

static class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

static class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

public static void main(String[] args) {

Light light = new Light();

Command lightOn = new LightOnCommand(light);

Command lightOff = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

remote.pressButton();

System.***out***.println();

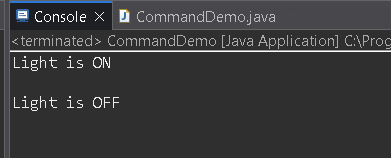
remote.setCommand(lightOff);

remote.pressButton();

}

}

**Output:**



**Exercise 10: Implementing the MVC Pattern**

package package1;

public class MVCDemo {

static 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;

}

}

static 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);

}

}

static 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 String getStudentName() {

return model.getName();

}

public void setStudentId(String id) {

model.setId(id);

}

public String getStudentId() {

return model.getId();

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

public static void main(String[] args) {

Student model = new Student();

model.setName("Varshitha");

model.setId("S101");

model.setGrade("A");

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

System.***out***.println();

controller.setStudentName("Kavya");

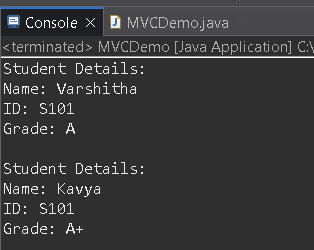
controller.setStudentGrade("A+");

controller.updateView();

}

}

**Output:**



**Exercise 11: Implementing Dependency Injection**

package package1;

public class DependencyInjectionDemo {

interface CustomerRepository {

String findCustomerById(String id);

}

static class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer[id=" + id + ", name=Varshitha]";

}

}

static class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomer(String id) {

String customer = repository.findCustomerById(id);

System.***out***.println(customer);

}

}

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.getCustomer("C101");

}

}

**Output:**

