**SUPERSET ID :6416838**

Exercise 1: Implementing the Singleton Pattern

PROGRAM:

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.java

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("Application started");

Logger logger2 = Logger.getInstance();

logger2.log("Another message");

if (logger1 == logger2) {

System.out.println("Both logger instances are the same (Singleton works!)");

} else {

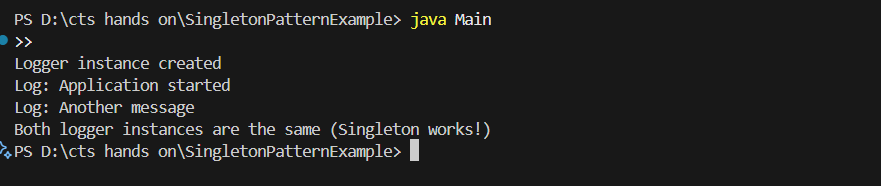
System.out.println("Different instances (Singleton failed)");

}

}

}

Output:



Exercise 2: Implementing the Factory Method Pattern

PROGRAM:

Main.java

// Main.java

interface Notification {

void notifyUser();

}

class EmailNotification implements Notification {

@Override

public void notifyUser() {

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

}

}

class SMSNotification implements Notification {

@Override

public void notifyUser() {

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

}

}

class PushNotification implements Notification {

@Override

public void notifyUser() {

System.out.println("Sending a Push Notification");

}

}

class NotificationFactory {

public Notification createNotification(String type) {

if (type == null || type.isEmpty()) return null;

switch (type.toLowerCase()) {

case "sms":

return new SMSNotification();

case "email":

return new EmailNotification();

case "push":

return new PushNotification();

default:

throw new IllegalArgumentException("Unknown notification type: " + type);

}

}

}

public class Main {

public static void main(String[] args) {

NotificationFactory factory = new NotificationFactory();

Notification n1 = factory.createNotification("sms");

n1.notifyUser();

Notification n2 = factory.createNotification("email");

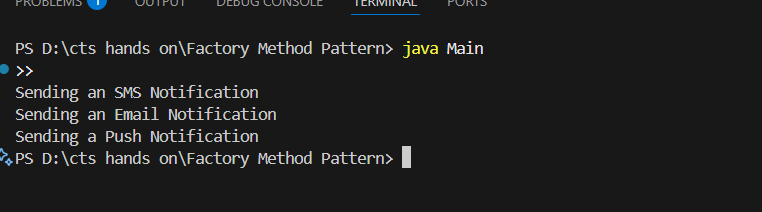
n2.notifyUser();

Notification n3 = factory.createNotification("push");

n3.notifyUser();

}

}

Output:

**ADAPTER PATTERN**

PROGRAM:

// Main.java

interface PaymentProcessor {

void processPayment(double amount);

}

class PayPalGateway {

void sendPayment(double amount) {

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

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal = new PayPalGateway();

public void processPayment(double amount) {

paypal.sendPayment(amount);

}

}

public class Main {

public static void main(String[] args) {

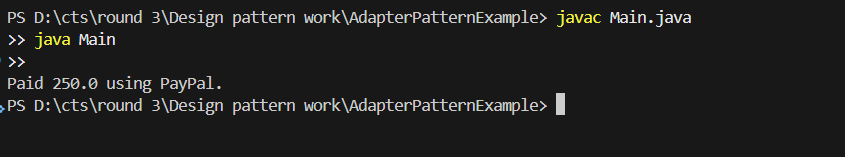
PaymentProcessor processor = new PayPalAdapter();

processor.processPayment(250.0);

}

}

Output:



**BUILDER PATTERN**

PROGRAM:

Computer.java

public class Computer {

private String CPU;

private String RAM;

private String storage;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

}

public static class Builder {

private String CPU;

private String RAM;

private String storage;

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 Computer build() {

return new Computer(this);

}

}

public void showSpecs() {

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

}

}

Main.java

public class Main {

public static void main(String[] args) {

Computer comp = new Computer.Builder()

.setCPU("Intel i7")

.setRAM("16GB")

.setStorage("512GB SSD")

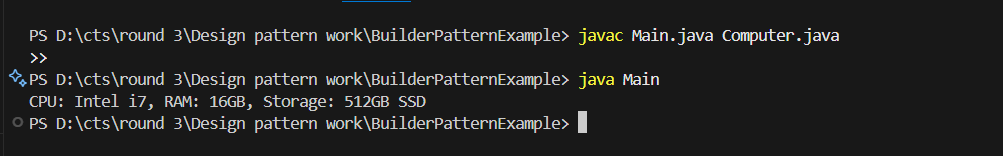
.build();

comp.showSpecs();

}

}

Output:



**COMMAND PATTERN**

PROGRAM:

interface Command {

void execute();

}

class Light {

void on() {

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

}

void off() {

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

}

}

class LightOnCommand implements Command {

Light light;

LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.on();

}

}

class LightOffCommand implements Command {

Light light;

LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.off();

}

}

class RemoteControl {

Command command;

void setCommand(Command command) {

this.command = command;

}

void pressButton() {

command.execute();

}

}

public class Main {

public static void main(String[] args) {

Light light = new Light();

RemoteControl remote = new RemoteControl();

remote.setCommand(new LightOnCommand(light));

remote.pressButton();

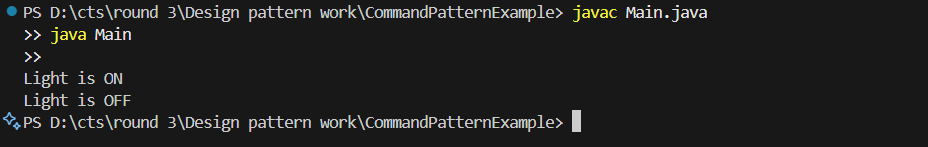
remote.setCommand(new LightOffCommand(light));

remote.pressButton();

}

}

Output:



**DECORATE PATTERN**

PROGRAM:

interface Notifier {

void send();

}

class EmailNotifier implements Notifier {

public void send() {

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

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send() {

notifier.send();

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

}

}

public class Main {

public static void main(String[] args) {

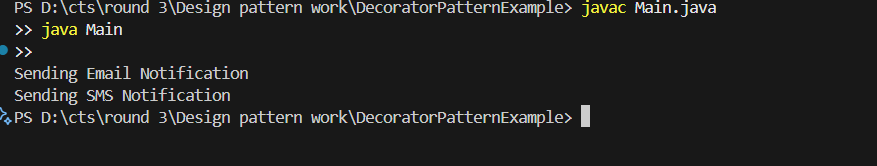
Notifier notifier = new SMSNotifierDecorator(new EmailNotifier());

notifier.send();

}

}

Output:



**DEPENDENCY INJECTION**

PROGRAM:

interface CustomerRepository {

String findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer with ID " + id;

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomer(String id) {

System.out.println(repository.findCustomerById(id));

}

}

public class Main {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

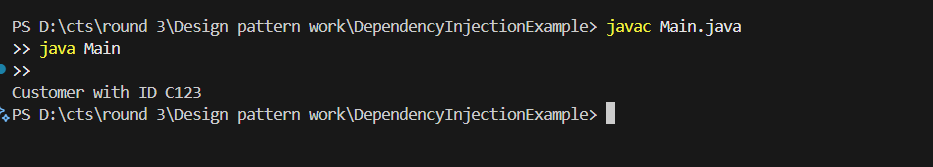
CustomerService service = new CustomerService(repo);

service.displayCustomer("C123");

}

}

Output:



**MVC PATTERN**

PROGRAM:

class Student {

private String name;

private String id;

private String grade;

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

this.name = name;

this.id = id;

this.grade = grade;

}

public String getName() { return name; }

public String getId() { return id; }

public String getGrade() { return grade; }

public void setGrade(String grade) { this.grade = grade; }

}

class StudentView {

void displayStudentDetails(Student student) {

System.out.println("Student: " + student.getName() + ", ID: " + student.getId() + ", Grade: " + student.getGrade());

}

}

class StudentController {

private Student student;

private StudentView view;

public StudentController(Student student, StudentView view) {

this.student = student;

this.view = view;

}

public void updateGrade(String grade) {

student.setGrade(grade);

}

public void display() {

view.displayStudentDetails(student);

}

}

public class Main {

public static void main(String[] args) {

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

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.display();

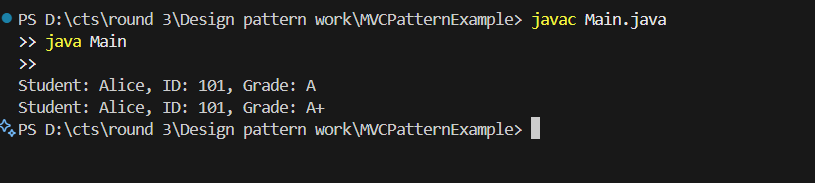
controller.updateGrade("A+");

controller.display();

}

}

Output:



**OBSERVER PATTERN**

PROGRAM:

import java.util.\*;

interface Observer {

void update(String stockSymbol, double price);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

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

private String symbol;

private double price;

public void setStock(String symbol, double price) {

this.symbol = symbol;

this.price = price;

notifyObservers();

}

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(symbol, price);

}

}

}

class MobileApp implements Observer {

public void update(String stockSymbol, double price) {

System.out.println("MobileApp: " + stockSymbol + " is now " + price);

}

}

public class Main {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

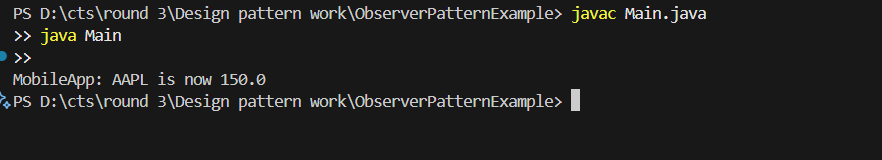
market.register(mobile);

market.setStock("AAPL", 150.0);

}

}

Output:



**PROXY PATTERN**

PROGRAM:

interface Image {

void display();

}

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadImageFromDisk();

}

private void loadImageFromDisk() {

System.out.println("Loading " + filename);

}

public void display() {

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

}

}

class ProxyImage implements Image {

private RealImage realImage;

private String filename;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

public class Main {

public static void main(String[] args) {

Image img = new ProxyImage("test.jpg");

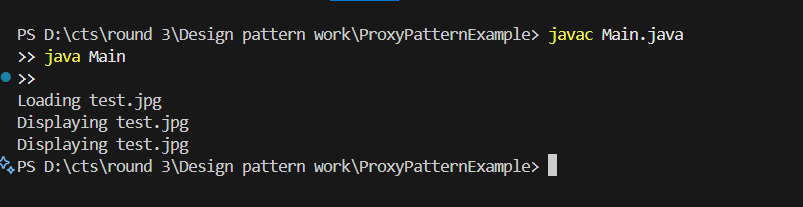
img.display();

img.display();

}

}

Output:



**STRATEGY PATTERN**

PROGRAM:

interface PaymentStrategy {

void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(double amount) {

strategy.pay(amount);

}

}

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setStrategy(new CreditCardPayment());

context.pay(100.0);

context.setStrategy(new PayPalPayment());

context.pay(200.0);

}

}

Output:

