WEEK-1 SINGLETON PATTERN EXAMPLE

Logger.java

public 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);

    }

}

Main.java

public class Main {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("First log message");

        Logger logger2 = Logger.getInstance();

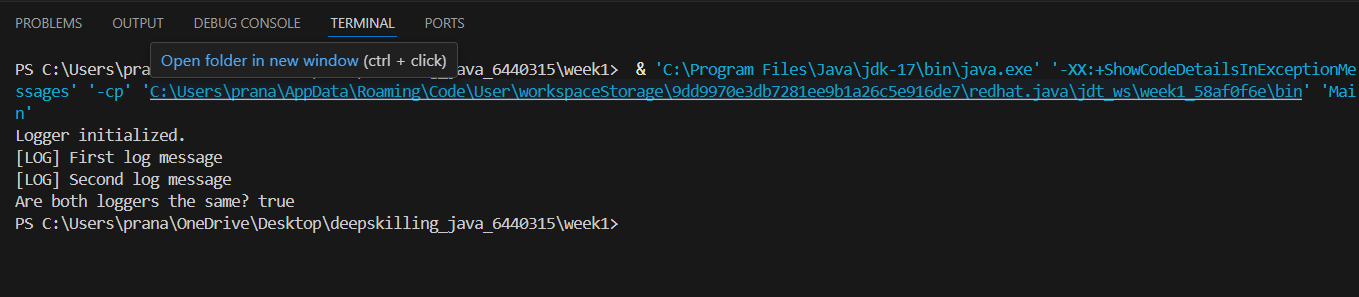
        logger2.log("Second log message");

        System.out.println("Are both loggers the same? " + (logger1 == logger2));

    }

}

OUTPUT:



WEEK 1-FACTORYMETHOD PATTERN EXAMPLE

Document.java

package FactoryMethodPatternExample;

interface Document {

    void open();

}

class WordDocument implements Document {

    public void open() {

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

    }

}

class PdfDocument implements Document {

    public void open() {

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

    }

}

class ExcelDocument implements Document {

    public void open() {

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

    }

}

DocumentApp.java

package FactoryMethodPatternExample;

public class DocumentApp {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

DocumentFactory.java

package FactoryMethodPatternExample;

abstract class DocumentFactory {

    public abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new WordDocument();

    }

}

class PdfDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new PdfDocument();

    }

}

class ExcelDocumentFactory extends DocumentFactory {

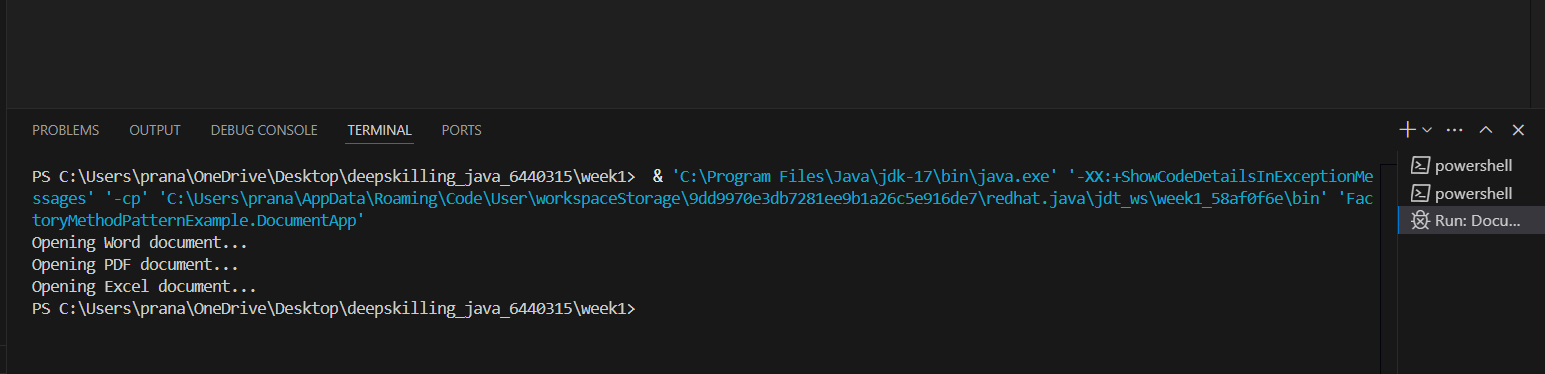
    public Document createDocument() {

        return new ExcelDocument();

    }

}

OUTPUT:



**Exercise 3: Builder Pattern**

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 display() {

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

}

}

public class Main {

public static void main(String[] args) {

Computer comp1 = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

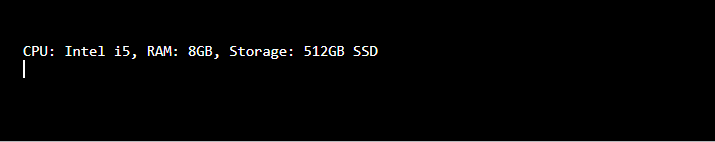
.setStorage("512GB SSD")

.build();

comp1.display();

}

}



**Exercise 4: Adapter Pattern**

interface PaymentProcessor {

void processPayment(double amount);

}

class PayPal {

public void sendMoney(double amount) {

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

}

}

class Stripe {

public void makePayment(double amount) {

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

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPal paypal = new PayPal();

public void processPayment(double amount) {

paypal.sendMoney(amount);

}

}

class StripeAdapter implements PaymentProcessor {

private Stripe stripe = new Stripe();

public void processPayment(double amount) {

stripe.makePayment(amount);

}

}

public class PaymentAdapterDemo {

public static void main(String[] args) {

PaymentProcessor paypal = new PayPalAdapter();

paypal.processPayment(1000);

PaymentProcessor stripe = new StripeAdapter();

stripe.processPayment(2000);

}

}



**Exercise 5: Decorator Pattern**

interface Notifier {

void send(String message);

}

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

public class NotificationDecoratorDemo {

public static void main(String[] args) {

Notifier notifier = new SlackNotifierDecorator(

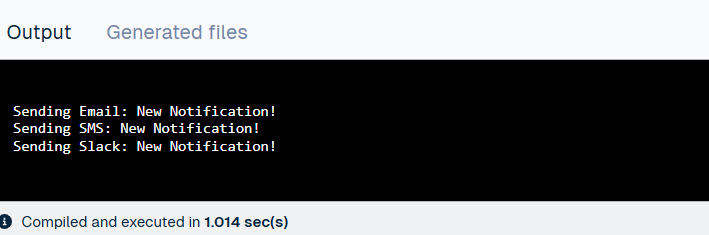
new SMSNotifierDecorator(

new EmailNotifier()));

notifier.send("New Notification!");

}

}



**Exercise 6: Proxy Pattern**

interface Image {

void display();

}

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromDisk();

}

private void loadFromDisk() {

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

}

public void display() {

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

}

}

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 class ImageProxyDemo {

public static void main(String[] args) {

Image img1 = new ProxyImage("photo1.jpg");

img1.display();

img1.display();

}

}



**Exercise 7: Observer Pattern**

import java.util.\*;

interface Observer {

void update(String stockName, 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 stockName;

private double price;

public void setStock(String stockName, double price) {

this.stockName = stockName;

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(stockName, price);

}

}

}

class MobileApp implements Observer {

public void update(String stockName, double price) {

System.out.println("MobileApp - " + stockName + ": " + price);

}

}

class WebApp implements Observer {

public void update(String stockName, double price) {

System.out.println("WebApp - " + stockName + ": " + price);

}

}

public class StockObserverDemo {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

market.register(mobile);

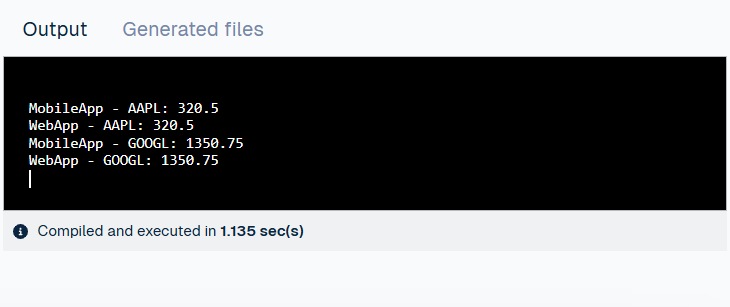
market.register(web);

market.setStock("AAPL", 320.50);

market.setStock("GOOGL", 1350.75);

}

}



**Exercise 8: Strategy Pattern**

interface PaymentStrategy {

void pay(int amount);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(int amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(int amount) {

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

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(int amount) {

strategy.pay(amount);

}

}

public class PaymentStrategyDemo {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment());

context.pay(1000);

context.setPaymentStrategy(new PayPalPayment());

context.pay(2000);

}

}



**Exercise 9: Command Pattern**

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 {

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();

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

public class CommandPatternDemo {

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();

}

}



**Exercise 10: MVC Pattern**

class Student {

private String name;

private int id;

private String grade;

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

this.name = name;

this.id = id;

this.grade = grade;

}

public String getName() { return name; }

public int getId() { return id; }

public String getGrade() { return grade; }

public void setName(String name) { this.name = name; }

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

}

class StudentView {

public void displayStudentDetails(Student student) {

System.out.println("Student: " + student.getName() +

", ID: " + student.getId() +

", Grade: " + student.getGrade());

}

}

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void updateView() {

view.displayStudentDetails(model);

}

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

}

public class MVCPatternDemo {

public static void main(String[] args) {

Student student = new Student("Alice", 1, "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

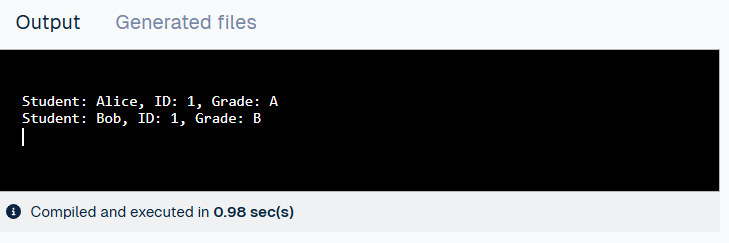
controller.setStudentName("Bob");

controller.setStudentGrade("B");

controller.updateView();

}

}



**Exercise 11: Dependency Injection**

interface CustomerRepository {

String findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer#" + id + ": John Doe";

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomer(int id) {

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

}

}

public class DependencyInjectionDemo {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.displayCustomer(101);

} }

