**WEEK 1 : DESIGN PATTERNS AND PRINCIPLES**

**MANDATORY**

Exercise 1: Implementing the Singleton Pattern

package weekone;

public class SingletonPatternExample {

static class Logger {

private static Logger *instance*;

private Logger() {

System.*out*.println("Logger Object Created");

}

public static Logger getInstance() {

if (*instance* == null) {

*instance* = new Logger();

}

return *instance*;

}

public void logMessage(String message) {

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

}

}

public static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

Logger logger2 = Logger.*getInstance*();

logger1.logMessage("First log");

logger2.logMessage("Second log");

if (logger1 == logger2) {

System.*out*.println("Both are Same instance.");

} else {

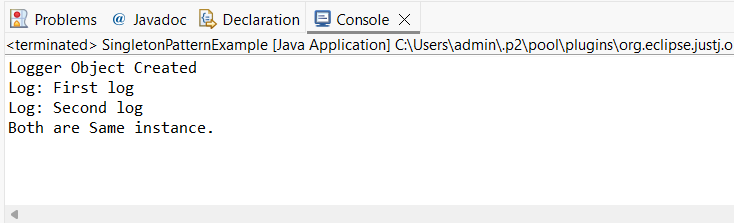
System.*out*.println("Different instances Not a Singleton");

}

}

}

OUPUT :



Exercise 2: Implementing the Factory Method Pattern

package weekone;

public class FactoryMethodPatternExample {

interface Document {

void open();

}

static class WordDocument implements Document {

public void open() {

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

}

}

static class PdfDocument implements Document {

public void open() {

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

}

}

static class ExcelDocument implements Document {

public void open() {

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

}

}

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();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

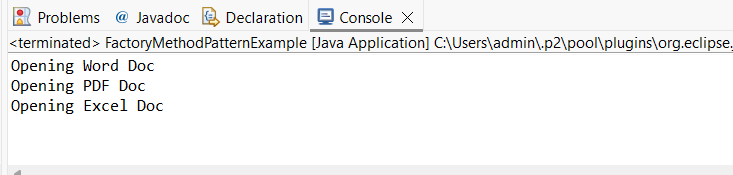
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

OUTPUT:



**ADDITIONAL EX**

Exercise 3: Implementing the Builder Pattern

package weekone;

public class BuilderPatternExample {

static class Computer {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

}

public String toString() {

return "CPU: " + cpu + ", RAM: " + ram + ", Storage: " + storage + ", Graphics: " + graphicsCard;

}

static class Builder {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

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 Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

public static void main(String[] args) {

Computer basicComputer = new Computer.Builder()

.setCPU("Intel i3")

.setRAM("8GB")

.setStorage("256GB SSD")

.build();

Computer gamingComputer = new Computer.Builder()

.setCPU("Intel i7")

.setRAM("16GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA GTX 1660")

.build();

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

System.*out*.println(basicComputer);

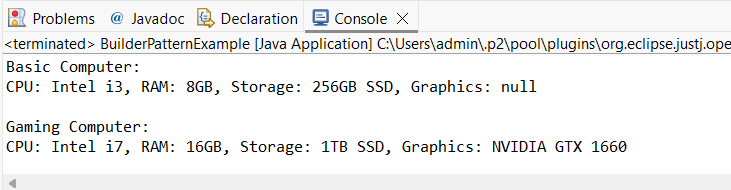
System.*out*.println("\nGaming Computer:");

System.*out*.println(gamingComputer);

}

}

OUTPUT:



Exercise 4: Implementing the Adapter Pattern

package weekone;

public class AdapterPatternExample {

interface PaymentProcessor {

void processPayment(double amount);

}

static class PayPalGateway {

public void sendMoney(double amount) {

System.*out*.println("PayPal processed: Rs." + amount);

}

}

static class StripeGateway {

public void makePayment(double amount) {

System.*out*.println("Stripe processed: Rs." + amount);

}

}

static class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPal;

public PayPalAdapter(PayPalGateway gateway) {

this.payPal = gateway;

}

public void processPayment(double amount) {

payPal.sendMoney(amount);

}

}

static class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway gateway) {

this.stripe = gateway;

}

public void processPayment(double amount) {

stripe.makePayment(amount);

}

}

public static void main(String[] args) {

PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());

paypalProcessor.processPayment(100.0);

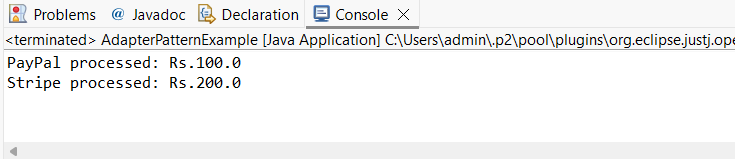
PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

stripeProcessor.processPayment(200.0);

}

}

OUTPUT:



Exercise 5: Implementing the Decorator Pattern

package weekone;

public class DecoratorPatternExample {

interface Notifier {

void send(String message);

}

static class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

static abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send(String message) {

wrappedNotifier.send(message);

}

}

static class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

static class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(notifier);

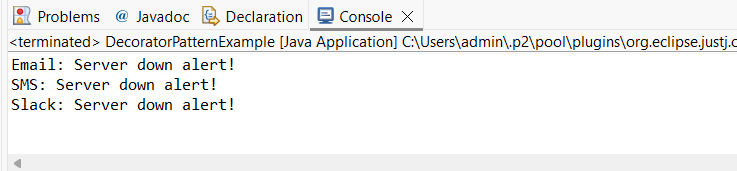
Notifier fullNotifier = new SlackNotifierDecorator(smsNotifier);

fullNotifier.send("Server down alert!");

}

}

OUTPUT:



Exercise 6: Implementing the Proxy Pattern

package weekone;

public class ProxyPatternExample {

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 " + fileName + " from remote server");

}

public void display() {

System.*out*.println("Displaying " + 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 img1 = new ProxyImage("image1.jpg");

Image img2 = new ProxyImage("image2.jpg");

img1.display();

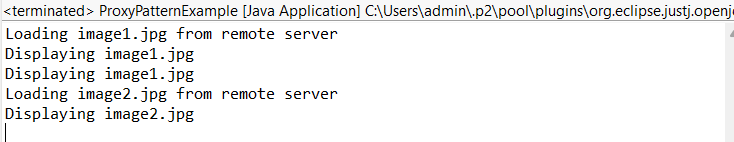
img1.display();

img2.display();

}

}

OUTPUT:



Exercise 7: Implementing the Observer Pattern

package weekone;

import java.util.\*;

public class ObserverPatternExample {

interface Observer {

void update(String stockName, double price);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

static class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public void setStock(String name, double price) {

this.stockName = name;

this.stockPrice = 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, stockPrice);

}

}

}

static class MobileApp implements Observer {

public void update(String stockName, double price) {

System.*out*.println("MobileApp: " + stockName + " is now $" + price);

}

}

static class WebApp implements Observer {

public void update(String stockName, double price) {

System.*out*.println("WebApp: " + stockName + " is now $" + 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.setStock("TCS", 3200.50);

market.setStock("INFY", 1485.75);

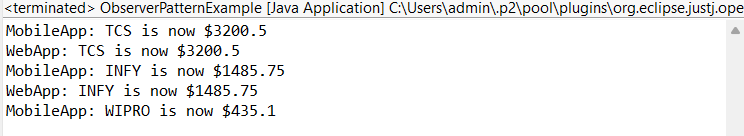
market.deregister(web);

market.setStock("WIPRO", 435.10);

}

}

OUTPUT:



Exercise 8: Implementing the Strategy Pattern

package weekone;

public class StrategyPatternExample {

interface PaymentStrategy {

void pay(double amount);

}

static class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

static class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

static class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void payAmount(double amount) {

if (strategy != null) {

strategy.pay(amount);

} else {

System.*out*.println("No payment strategy selected");

}

}

}

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment());

context.payAmount(500.0);

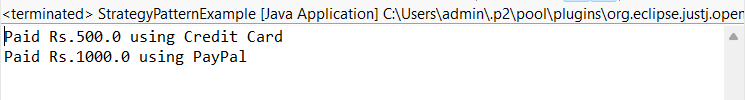
context.setPaymentStrategy(new PayPalPayment());

context.payAmount(1000.0);

}

}

OUTPUT:



Exercise 9: Implementing the Command Pattern

package weekone;

public class CommandPatternExample {

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() {

if (command != null) {

command.execute();

} else {

System.*out*.println("No command assigned");

}

}

}

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command onCommand = new LightOnCommand(livingRoomLight);

Command offCommand = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(onCommand);

remote.pressButton();

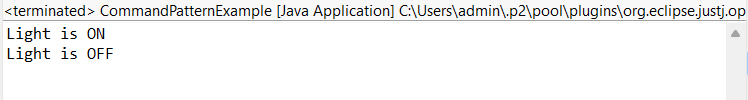
remote.setCommand(offCommand);

remote.pressButton();

}

}

OUTPUT:



Exercise 10: Implementing the MVC Pattern

package weekone;

public class MVCPatternExample {

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 student;

private StudentView view;

public StudentController(Student student, StudentView view) {

this.student = student;

this.view = view;

}

public void setStudentName(String name) {

student.setName(name);

}

public void setStudentId(String id) {

student.setId(id);

}

public void setStudentGrade(String grade) {

student.setGrade(grade);

}

public void updateView() {

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

}

}

public static void main(String[] args) {

Student student = new Student();

student.setName("Aiswarya Raja");

student.setId("22UIT006");

student.setGrade("A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

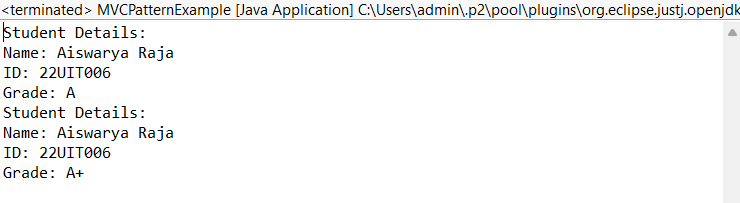
controller.setStudentGrade("A+");

controller.updateView();

}

}

OUTPUT:



Exercise 11: Implementing Dependency Injection

package weekone;

public class DependencyInjectionExample {

interface CustomerRepository {

String findCustomerById(String id);

}

static class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer ID: " + id + "\nName: Aiswarya Raja";

}

}

static class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomerDetails(String id) {

String details = repository.findCustomerById(id);

System.*out*.println(details);

}

}

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.getCustomerDetails("C101");

}

}

OUTPUT:

