**WEEK 1:DESIGN PATTERNS AND PRINCIPLES**

**MANDATORY**

**Exercise 1: Implementing the Singleton Pattern**

**CODE:**

package weekone;

public class SingletonPatternExample {

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*();

Logger logger2 = Logger.*getInstance*();

logger1.log("First message");

logger2.log("Second message");

if (logger1 == logger2) {

System.*out*.println("Same instance is used for both loggers.");

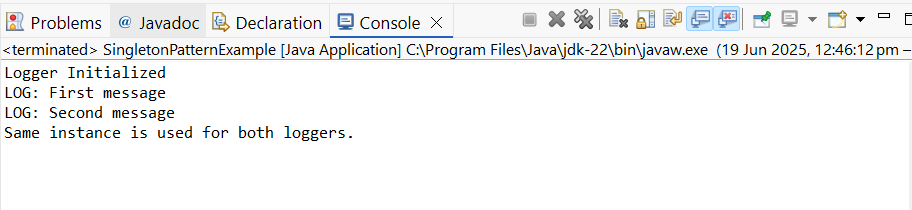
} else {

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

}

}

}

**Console output for Exercise 1:**

**MANDATORY**

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

**CODE:**

package weekone;

public class FactoryMethodPatternExample {

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();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

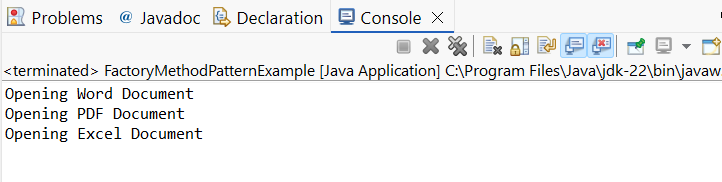
DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Console output for Exercise 2:**

**HANDS ON**

**Exercise 3: Implementing the Builder Pattern**

**SOLUTION:**

package weekone;

public class BuilderPatternExample {

static 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);

}

}

@Override

public String toString() {

return "Computer [CPU=" + cpu + ", RAM=" + ram + ", Storage=" + storage + "]";

}

}

public static void main(String[] args) {

Computer gamingPC = new Computer.Builder()

.setCpu("Intel i9")

.setRam("32GB")

.setStorage("1TB SSD")

.build();

Computer officePC = new Computer.Builder()

.setCpu("Intel i5")

.setRam("16GB")

.setStorage("512GB SSD")

.build();

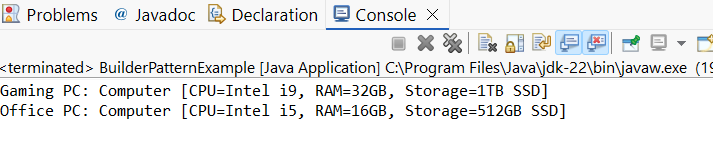
System.*out*.println("Gaming PC: " + gamingPC);

System.*out*.println("Office PC: " + officePC);

}

}

**Console output for Exercise 3:**

****

**Exercise 4: Implementing the Adapter Pattern**

**SOLUTION:**

package weekone;

public class AdapterPatternExample {

interface PaymentProcessor {

void processPayment(double amount);

}

static class PayPal {

public void makePayment(double amount) {

System.*out*.println("PayPal processed payment of $" + amount);

}

}

static class PayPalAdapter implements PaymentProcessor {

private PayPal paypal;

public PayPalAdapter(PayPal paypal) {

this.paypal = paypal;

}

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

public static void main(String[] args) {

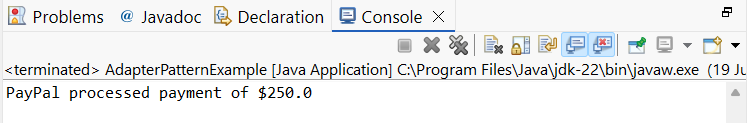
PaymentProcessor processor = new PayPalAdapter(new PayPal());

processor.processPayment(250.0);

}

}

**Console output for Exercise 4:**

****

**Exercise 5: Implementing the Decorator Pattern**

**SOLUTION:**

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 sent: " + message);

}

}

static abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

static class SMSNotifier extends NotifierDecorator {

public SMSNotifier(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

static class SlackNotifier extends NotifierDecorator {

public SlackNotifier(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

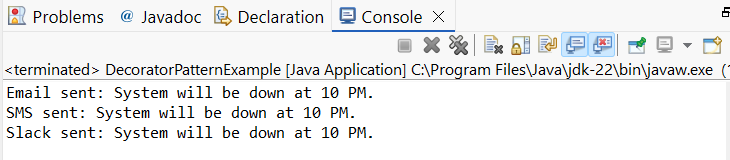
public static void main(String[] args) {

Notifier notifier = new SlackNotifier(new SMSNotifier(new EmailNotifier()));

notifier.send("System will be down at 10 PM.");

}

}

**Console output for Exercise 5:**

**Exercise 6: Implementing the Proxy Pattern**

**SOLUTION:**

package weekone;

public class ProxyPatternExample {

interface Image {

void display();

}

static class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromDisk();

}

private void loadFromDisk() {

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

}

public void display() {

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

}

}

static 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 static void main(String[] args) {

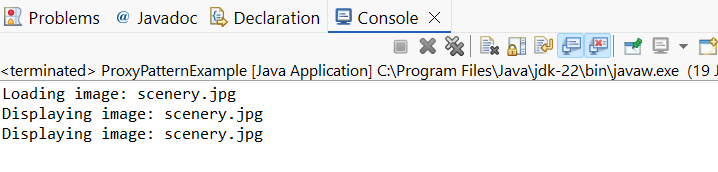
Image image = new ProxyImage("scenery.jpg");

image.display();

image.display();

}

}

**Console output for Exercise 6:**

**Exercise 7: Implementing the Observer Pattern**

**SOLUTION:**

package weekone;

import java.util.\*;

public class ObserverPatternExample {

interface Observer {

void update(String stock, 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 setPrice(String stock, double price) {

this.stockName = stock;

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 stock, double price) {

System.*out*.println("MobileApp - " + stock + ": $" + price);

}

}

static class WebApp implements Observer {

public void update(String stock, double price) {

System.*out*.println("WebApp - " + stock + ": $" + price);

}

}

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

market.register(mobile);

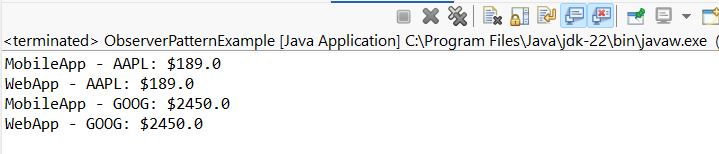
market.register(web);

market.setPrice("AAPL", 189.0);

market.setPrice("GOOG", 2450.0);

}

}

**Console output for Exercise 7:**

**Exercise 8: Implementing the Strategy Pattern**

**SOLUTION:**

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 $" + 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 setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void payAmount(double amount) {

strategy.pay(amount);

}

}

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setStrategy(new CreditCardPayment());

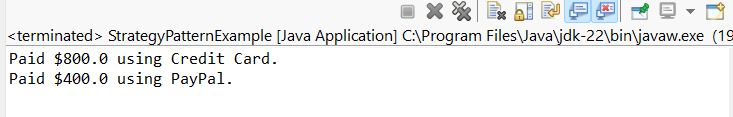
context.payAmount(800);

context.setStrategy(new PayPalPayment());

context.payAmount(400);

}

}

**Console output for Exercise 8:**

**Exercise 9: Implementing the Command Pattern**

package weekone;

public class CommandPatternExample {

interface Command {

void execute();

}

static class Light {

public void on() {

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

}

public void off() {

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.on();

}

}

static class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.off();

}

}

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 on = new LightOnCommand(light);

Command off = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(on);

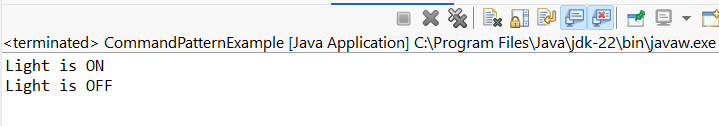
remote.pressButton();

remote.setCommand(off);

remote.pressButton();

}

}

**Console output for Exercise 9:**

**Exercise 10: Implementing the MVC Pattern**

**SOLUTION:**

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: " + name + ", ID: " + id + ", 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 updateView() {

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

}

}

public static void main(String[] args) {

Student student = new Student();

student.setName("Subhiksha S");

student.setId("6422802");

student.setGrade("A");

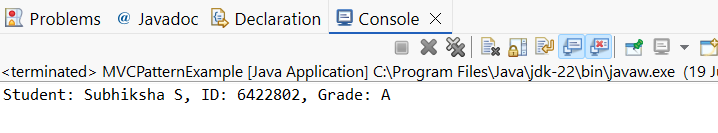
StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

}

}

**Console output for Exercise 10:**

**Exercise 11: Implementing Dependency Injection**

**SOLUTION:**

package weekone;

public class DependencyInjectionExample {

interface CustomerRepository {

String findCustomerById(int id);

}

static class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer ID: " + id + " - Name: Subhiksha S";

}

}

static class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void showCustomer(int id) {

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

}

}

public static void main(String[] args) {

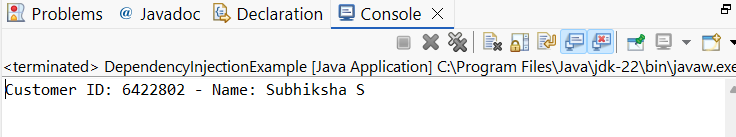
CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.showCustomer(6422802);

}

}

**Console output for Exercise 11:**