Java Design Patterns Implementation

# Exercise 1: Singleton Pattern

## Java Code:

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 class Main {  
 public static void main(String[] args) {  
 Logger log1 = Logger.getInstance();  
 Logger log2 = Logger.getInstance();  
 System.out.println(log1 == log2);  
 }  
}

## Output:

Logger initialized  
true

# Exercise 2: Factory Method Pattern

## Java Code:

interface Document {  
 void open();  
}  
class WordDocument implements Document {  
 public void open() {  
 System.out.println("Word document opened.");  
 }  
}  
class PdfDocument implements Document {  
 public void open() {  
 System.out.println("PDF document opened.");  
 }  
}  
abstract class DocumentFactory {  
 public abstract Document createDocument();  
}  
class WordFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new WordDocument();  
 }  
}  
class PdfFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new PdfDocument();  
 }  
}  
public class Main {  
 public static void main(String[] args) {  
 DocumentFactory factory = new WordFactory();  
 Document doc = factory.createDocument();  
 doc.open();  
 }  
}

## Output:

Word document opened.

# Exercise 3: Builder Pattern

## Java Code:

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 showConfig() {  
 System.out.println("CPU: " + CPU + ", RAM: " + RAM + ", Storage: " + storage);  
 }  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Computer comp = new Computer.Builder()  
 .setCPU("Intel i5")  
 .setRAM("8GB")  
 .setStorage("512GB SSD")  
 .build();  
 comp.showConfig();  
 }  
}

## Output:

CPU: Intel i5, RAM: 8GB, Storage: 512GB SSD

# Exercise 4: Adapter Pattern

## Java Code:

interface PaymentProcessor {  
 void processPayment();  
}  
class PayPalGateway {  
 public void makePayment() {  
 System.out.println("Payment made through PayPal");  
 }  
}  
class PayPalAdapter implements PaymentProcessor {  
 private PayPalGateway gateway = new PayPalGateway();  
 public void processPayment() {  
 gateway.makePayment();  
 }  
}  
public class Main {  
 public static void main(String[] args) {  
 PaymentProcessor processor = new PayPalAdapter();  
 processor.processPayment();  
 }  
}

## Output:

Payment made through PayPal

# Exercise 5: Decorator Pattern

## Java Code:

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:

Sending Email Notification  
Sending SMS Notification

# Exercise 6: Proxy Pattern

## Java Code:

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 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 image = new ProxyImage("test.jpg");  
 image.display();  
 image.display();  
 }  
}

## Output:

Loading test.jpg  
Displaying test.jpg  
Displaying test.jpg

# Exercise 7: Observer Pattern

## Java Code:

import java.util.\*;  
  
interface Observer {  
 void update(float price);  
}  
interface Stock {  
 void register(Observer o);  
 void remove(Observer o);  
 void notifyObservers();  
}  
class StockMarket implements Stock {  
 private List<Observer> observers = new ArrayList<>();  
 private float price;  
  
 public void register(Observer o) {  
 observers.add(o);  
 }  
 public void remove(Observer o) {  
 observers.remove(o);  
 }  
 public void setPrice(float price) {  
 this.price = price;  
 notifyObservers();  
 }  
 public void notifyObservers() {  
 for (Observer o : observers) {  
 o.update(price);  
 }  
 }  
}  
class MobileApp implements Observer {  
 public void update(float price) {  
 System.out.println("MobileApp: New stock price: " + price);  
 }  
}  
public class Main {  
 public static void main(String[] args) {  
 StockMarket stockMarket = new StockMarket();  
 Observer app = new MobileApp();  
 stockMarket.register(app);  
 stockMarket.setPrice(105.5f);  
 }  
}

## Output:

MobileApp: New stock price: 105.5

# Exercise 8: Strategy Pattern

## Java Code:

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 PaymentContext(PaymentStrategy strategy) {  
 this.strategy = strategy;  
 }  
 public void pay(int amount) {  
 strategy.pay(amount);  
 }  
}  
public class Main {  
 public static void main(String[] args) {  
 PaymentContext context = new PaymentContext(new CreditCardPayment());  
 context.pay(500);  
 }  
}

## Output:

Paid 500 using Credit Card

# Exercise 9: Command Pattern

## Java Code:

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 Main {  
 public static void main(String[] args) {  
 Light light = new Light();  
 RemoteControl remote = new RemoteControl();  
 remote.setCommand(new LightOnCommand(light));  
 remote.pressButton();  
 }  
}

## Output:

Light is ON

# Exercise 10: MVC Pattern

## Java Code:

class Student {  
 private String name;  
 private String id;  
  
 public Student(String name, String id) {  
 this.name = name;  
 this.id = id;  
 }  
 public String getName() { return name; }  
 public void setName(String name) { this.name = name; }  
 public String getId() { return id; }  
}  
class StudentView {  
 public void displayStudentDetails(String name, String id) {  
 System.out.println("Student: " + name + ", ID: " + id);  
 }  
}  
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());  
 }  
}  
public class Main {  
 public static void main(String[] args) {  
 Student student = new Student("Alice", "101");  
 StudentView view = new StudentView();  
 StudentController controller = new StudentController(student, view);  
 controller.updateView();  
 }  
}

## Output:

Student: Alice, ID: 101

# Exercise 11: Dependency Injection

## Java Code:

interface CustomerRepository {  
 String findCustomerById(String id);  
}  
class CustomerRepositoryImpl implements CustomerRepository {  
 public String findCustomerById(String id) {  
 return "Customer " + id;  
 }  
}  
class CustomerService {  
 private CustomerRepository repository;  
  
 public CustomerService(CustomerRepository repository) {  
 this.repository = repository;  
 }  
 public void showCustomer(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.showCustomer("123");  
 }  
}

## Output:

Customer 123