### **Exercise 1: Implementing the Singleton Pattern**

#### **1. Create a New Java Project**

Create a new Java project named SingletonPatternExample.

#### **2. Define a Singleton Class**

**Logger.java**:

public class Logger {  
 private static Logger instance;  
  
 // Private constructor to prevent instantiation  
 private Logger() {}  
  
 // Public method to provide access to the single instance  
 public static Logger getInstance() {  
 if (instance == null) {  
 synchronized (Logger.class) {  
 if (instance == null) {  
 instance = new Logger();  
 }  
 }  
 }  
 return instance;  
 }  
  
 public void log(String message) {  
 System.out.println(message);  
 }  
}

#### **3. Implement the Singleton Pattern**

The Logger class follows the Singleton design pattern by ensuring only one instance is created. The getInstance() method provides access to this instance, and synchronization ensures thread safety.

#### **4. Test the Singleton Implementation**

**SingletonTest.java**:

public class SingletonTest {  
 public static void main(String[] args) {  
 Logger logger1 = Logger.getInstance();  
 Logger logger2 = Logger.getInstance();  
  
 System.out.println("Are both logger instances the same? " + (logger1 == logger2));  
  
 logger1.log("This is a log message.");  
 logger2.log("This is another log message.");  
 }  
}

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

#### **1. Create a New Java Project**

Create a new Java project named FactoryMethodPatternExample.

#### **2. Define Document Classes**

**Document.java**:

public interface Document {  
 void open();  
}

**WordDocument.java**:

public class WordDocument implements Document {  
 @Override  
 public void open() {  
 System.out.println("Opening Word document.");  
 }  
}

**PdfDocument.java**:

public class PdfDocument implements Document {  
 @Override  
 public void open() {  
 System.out.println("Opening PDF document.");  
 }  
}

**ExcelDocument.java**:

public class ExcelDocument implements Document {  
 @Override  
 public void open() {  
 System.out.println("Opening Excel document.");  
 }  
}

#### **3. Implement the Factory Method**

**DocumentFactory.java**:

public abstract class DocumentFactory {  
 public abstract Document createDocument();  
}

**WordDocumentFactory.java**:

public class WordDocumentFactory extends DocumentFactory {  
 @Override  
 public Document createDocument() {  
 return new WordDocument();  
 }  
}

**PdfDocumentFactory.java**:

public class PdfDocumentFactory extends DocumentFactory {  
 @Override  
 public Document createDocument() {  
 return new PdfDocument();  
 }  
}

**ExcelDocumentFactory.java**:

public class ExcelDocumentFactory extends DocumentFactory {  
 @Override  
 public Document createDocument() {  
 return new ExcelDocument();  
 }  
}

#### **4. Test the Factory Method Implementation**

**FactoryMethodTest.java**:

public class FactoryMethodTest {  
 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();  
 }  
}

### **Exercise 3: Implementing the Builder Pattern**

#### **1. Create a New Java Project**

Create a new Java project named BuilderPatternExample.

#### **2. Define a Product Class**

**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);  
 }  
 }  
  
 @Override  
 public String toString() {  
 return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", storage=" + storage + "]";  
 }  
}

#### **3. Implement the Builder Class**

The Builder class is a static nested class inside Computer, providing methods to set each attribute and a build() method to return an instance of Computer.

#### **4. Test the Builder Implementation**

**BuilderTest.java**:

public class BuilderTest {  
 public static void main(String[] args) {  
 Computer computer = new Computer.Builder()  
 .setCPU("Intel i7")  
 .setRAM("16GB")  
 .setStorage("512GB SSD")  
 .build();  
  
 System.out.println(computer);  
 }  
}

### **Exercise 4: Implementing the Adapter Pattern**

#### **1. Create a New Java Project**

Create a new Java project named AdapterPatternExample.

#### **2. Define Target Interface**

**PaymentProcessor.java**:

public interface PaymentProcessor {  
 void processPayment(double amount);  
}

#### **3. Implement Adaptee Classes**

**OldPaymentGateway.java**:

public class OldPaymentGateway {  
 public void pay(double amount) {  
 System.out.println("Processing payment of " + amount + " using Old Payment Gateway.");  
 }  
}

**NewPaymentGateway.java**:

public class NewPaymentGateway {  
 public void executePayment(double amount) {  
 System.out.println("Processing payment of " + amount + " using New Payment Gateway.");  
 }  
}

#### **4. Implement the Adapter Class**

**OldPaymentGatewayAdapter.java**:

public class OldPaymentGatewayAdapter implements PaymentProcessor {  
 private OldPaymentGateway oldPaymentGateway;  
  
 public OldPaymentGatewayAdapter(OldPaymentGateway oldPaymentGateway) {  
 this.oldPaymentGateway = oldPaymentGateway;  
 }  
  
 @Override  
 public void processPayment(double amount) {  
 oldPaymentGateway.pay(amount);  
 }  
}

**NewPaymentGatewayAdapter.java**:

public class NewPaymentGatewayAdapter implements PaymentProcessor {  
 private NewPaymentGateway newPaymentGateway;  
  
 public NewPaymentGatewayAdapter(NewPaymentGateway newPaymentGateway) {  
 this.newPaymentGateway = newPaymentGateway;  
 }  
  
 @Override  
 public void processPayment(double amount) {  
 newPaymentGateway.executePayment(amount);  
 }  
}

#### **5. Test the Adapter Implementation**

**AdapterTest.java**:

public class AdapterTest {  
 public static void main(String[] args) {  
 PaymentProcessor oldGateway = new OldPaymentGatewayAdapter(new OldPaymentGateway());  
 oldGateway.processPayment(100.0);  
  
 PaymentProcessor newGateway = new NewPaymentGatewayAdapter(new NewPaymentGateway());  
 newGateway.processPayment(200.0);  
 }  
}

### **Exercise 5: Implementing the Decorator Pattern**

#### **1. Create a New Java Project**

Create a new Java project named DecoratorPatternExample.

#### **2. Define Component Interface**

**Notifier.java**:

public interface Notifier {  
 void send(String message);  
}

#### **3. Implement Concrete Component**

**EmailNotifier.java**:

public class EmailNotifier implements Notifier {  
 @Override  
 public void send(String message) {  
 System.out.println("Sending Email: " + message);  
 }  
}

#### **4. Implement Decorator Classes**

**NotifierDecorator.java**:

public abstract class NotifierDecorator implements Notifier {  
 protected Notifier decoratedNotifier;  
  
 public NotifierDecorator(Notifier decoratedNotifier) {  
 this.decoratedNotifier = decoratedNotifier;  
 }  
  
 @Override  
 public void send(String message) {  
 decoratedNotifier.send(message);  
 }  
}

**SMSNotifierDecorator.java**:

public class SMSNotifierDecorator extends NotifierDecorator {  
 public SMSNotifierDecorator(Notifier decoratedNotifier) {  
 super(decoratedNotifier);  
 }  
  
 @Override  
 public void send(String message) {  
 decoratedNotifier.send(message);  
 System.out.println("Sending SMS: " + message);  
 }  
}

**SlackNotifierDecorator.java**:

public class SlackNotifierDecorator extends NotifierDecorator {  
 public SlackNotifierDecorator(Notifier decoratedNotifier) {  
 super(decoratedNotifier);  
 }  
  
 @Override  
 public void send(String message) {  
 decoratedNotifier.send(message);  
 System.out.println("Sending Slack message: " + message);  
 }  
}

#### **5. Test the Decorator Implementation**

**DecoratorTest.java**:

public class DecoratorTest {  
 public static void main(String[] args) {  
 Notifier emailNotifier = new EmailNotifier();  
 Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);  
 Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);  
  
 slackNotifier.send("Hello World!");  
 }  
}

### **Exercise 6: Implementing the Proxy Pattern**

#### **1. Create a New Java Project**

Create a new Java project named ProxyPatternExample.

#### **2. Define Subject Interface**

**Image.java**:

public interface Image {  
 void display();  
}

#### **3. Implement Real Subject Class**

**RealImage.java**:

public class RealImage implements Image {  
 private String filename;  
  
 public RealImage(String filename) {  
 this.filename = filename;  
 loadImageFromDisk();  
 }  
  
 private void loadImageFromDisk() {  
 System.out.println("Loading " + filename);  
 }  
  
 @Override  
 public void display() {  
 System.out.println("Displaying " + filename);  
 }  
}

#### **4. Implement Proxy Class**

**ProxyImage.java**:

public class ProxyImage implements Image {  
 private RealImage realImage;  
 private String filename;  
  
 public ProxyImage(String filename) {  
 this.filename = filename;  
 }  
  
 @Override  
 public void display() {  
 if (realImage == null) {  
 realImage = new RealImage(filename);  
 }  
 realImage.display();  
 }  
}

#### **5. Test the Proxy Implementation**

**ProxyTest.java**:

public class ProxyTest {  
 public static void main(String[] args) {  
 Image image1 = new ProxyImage("test\_image1.jpg");  
 Image image2 = new ProxyImage("test\_image2.jpg");  
  
 image1.display(); // Load and display  
 image1.display(); // Display without loading  
  
 image2.display(); // Load and display  
 }  
}

### **Exercise 7: Implementing the Observer Pattern**

#### **1. Create a New Java Project**

Create a new Java project named ObserverPatternExample.

#### **2. Define Subject Interface**

**Stock.java**:

import java.util.ArrayList;  
import java.util.List;  
  
public interface Stock {  
 void registerObserver(Observer observer);  
 void deregisterObserver(Observer observer);  
 void notifyObservers();  
}

#### **3. Implement Concrete Subject**

**StockMarket.java**:

import java.util.ArrayList;  
import java.util.List;  
  
public class StockMarket implements Stock {  
 private List<Observer> observers;  
 private double price;  
  
 public StockMarket() {  
 observers = new ArrayList<>();  
 }  
  
 @Override  
 public void registerObserver(Observer observer) {  
 observers.add(observer);  
 }  
  
 @Override  
 public void deregisterObserver(Observer observer) {  
 observers.remove(observer);  
 }  
  
 @Override  
 public void notifyObservers() {  
 for (Observer observer : observers) {  
 observer.update(price);  
 }  
 }  
  
 public void setPrice(double price) {  
 this.price = price;  
 notifyObservers();  
 }  
}

#### **4. Define Observer Interface**

**Observer.java**:

public interface Observer {  
 void update(double price);  
}

#### **5. Implement Concrete Observers**

**MobileApp.java**:

public class MobileApp implements Observer {  
 @Override  
 public void update(double price) {  
 System.out.println("MobileApp: Stock price updated to " + price);  
 }  
}

**WebApp.java**:

public class WebApp implements Observer {  
 @Override  
 public void update(double price) {  
 System.out.println("WebApp: Stock price updated to " + price);  
 }  
}

#### **6. Test the Observer Implementation**

**ObserverTest.java**:

public class ObserverTest {  
 public static void main(String[] args) {  
 StockMarket stockMarket = new StockMarket();  
 Observer mobileApp = new MobileApp();  
 Observer webApp = new WebApp();  
  
 stockMarket.registerObserver(mobileApp);  
 stockMarket.registerObserver(webApp);  
  
 stockMarket.setPrice(150.0);  
 stockMarket.setPrice(200.0);  
 }  
}

### **Exercise 8: Implementing the Strategy Pattern**

#### **1. Create a New Java Project**

Create a new Java project named StrategyPatternExample.

#### **2. Define Strategy Interface**

**PaymentStrategy.java**:

public interface PaymentStrategy {  
 void pay(double amount);  
}

#### **3. Implement Concrete Strategies**

**CreditCardPayment.java**:

public class CreditCardPayment implements PaymentStrategy {  
 private String cardNumber;  
  
 public CreditCardPayment(String cardNumber) {  
 this.cardNumber = cardNumber;  
 }  
  
 @Override  
 public void pay(double amount) {  
 System.out.println("Paying " + amount + " using Credit Card " + cardNumber);  
 }  
}

**PayPalPayment.java**:

public class PayPalPayment implements PaymentStrategy {  
 private String email;  
  
 public PayPalPayment(String email) {  
 this.email = email;  
 }  
  
 @Override  
 public void pay(double amount) {  
 System.out.println("Paying " + amount + " using PayPal account " + email);  
 }  
}

#### **4. Implement Context Class**

**PaymentContext.java**:

public class PaymentContext {  
 private PaymentStrategy paymentStrategy;  
  
 public PaymentContext(PaymentStrategy paymentStrategy) {  
 this.paymentStrategy = paymentStrategy;  
 }  
  
 public void executePayment(double amount) {  
 paymentStrategy.pay(amount);  
 }  
}

#### **5. Test the Strategy Implementation**

**StrategyTest.java**:

public class StrategyTest {  
 public static void main(String[] args) {  
 PaymentStrategy creditCard = new CreditCardPayment("1234-5678-9876-5432");  
 PaymentStrategy payPal = new [PayPalPayment("user@example.com](mailto:PayPalPayment("user@example.com)");  
  
 PaymentContext context = new PaymentContext(creditCard);  
 context.executePayment(100.0);  
  
 context = new PaymentContext(payPal);  
 context.executePayment(200.0);  
 }  
}

### **Exercise 9: Implementing the Command Pattern**

#### **1. Create a New Java Project**

Create a new Java project named CommandPatternExample.

#### **2. Define Command Interface**

**Command.java**:

public interface Command {  
 void execute();  
}

#### **3. Implement Concrete Commands**

**LightOnCommand.java**:

public class LightOnCommand implements Command {  
 private Light light;  
  
 public LightOnCommand(Light light) {  
 this.light = light;  
 }  
  
 @Override  
 public void execute() {  
 light.turnOn();  
 }  
}

**LightOffCommand.java**:

public class LightOffCommand implements Command {  
 private Light light;  
  
 public LightOffCommand(Light light) {  
 this.light = light;  
 }  
  
 @Override  
 public void execute() {  
 light.turnOff();  
 }  
}

#### **4. Implement Invoker Class**

**RemoteControl.java**:

public class RemoteControl {  
 private Command command;  
  
 public void setCommand(Command command) {  
 this.command = command;  
 }  
  
 public void pressButton() {  
 command.execute();  
 }  
}

#### **5. Implement Receiver Class**

**Light.java**:

public class Light {  
 public void turnOn() {  
 System.out.println("Light is ON");  
 }  
  
 public void turnOff() {  
 System.out.println("Light is OFF");  
 }  
}

#### **6. Test the Command Implementation**

**CommandTest.java**:

public class CommandTest {  
 public static void main(String[] args) {  
 Light light = new Light();  
 Command lightOn = new LightOnCommand(light);  
 Command lightOff = new LightOffCommand(light);  
  
 RemoteControl remote = new RemoteControl();  
 remote.setCommand(lightOn);  
 remote.pressButton();  
  
 remote.setCommand(lightOff);  
 remote.pressButton();  
 }  
}

### **Exercise 10: Implementing the MVC Pattern**

#### **1. Create a New Java Project**

Create a new Java project named MVCPatternExample.

#### **2. Define Model Class**

**Student.java**:

public 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;  
 }  
  
 // Getters and setters  
 public String getName() { return name; }  
 public void setName(String name) { this.name = name; }  
  
 public int getId() { return id; }  
 public void setId(int id) { this.id = id; }  
  
 public String getGrade() { return grade; }  
 public void setGrade(String grade) { this.grade = grade; }  
}

#### **3. Define View Class**

**StudentView.java**:

public class StudentView {  
 public void displayStudentDetails(String name, int id, String grade) {  
 System.out.println("Student Details:");  
 System.out.println("Name: " + name);  
 System.out.println("ID: " + id);  
 System.out.println("Grade: " + grade);  
 }  
}

#### **4. Define Controller Class**

**StudentController.java**:

public class StudentController {  
 private Student model;  
 private StudentView view;  
  
 public StudentController(Student model, StudentView view) {  
 this.model = model;  
 this.view = view;  
 }  
  
 public void setStudentName(String name) {  
 model.setName(name);  
 }  
  
 public String getStudentName() {  
 return model.getName();  
 }  
  
 public void setStudentId(int id) {  
 model.setId(id);  
 }  
  
 public int getStudentId() {  
 return model.getId();  
 }  
  
 public void setStudentGrade(String grade) {  
 model.setGrade(grade);  
 }  
  
 public String getStudentGrade() {  
 return model.getGrade();  
 }  
  
 public void updateView() {  
 view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());  
 }  
}

#### **5. Test the MVC Implementation**

**MVCTest.java**:

public class MVCTest {  
 public static void main(String[] args) {  
 Student model = new Student("John Doe", 1, "A");  
 StudentView view = new StudentView();  
 StudentController controller = new StudentController(model, view);  
  
 controller.updateView();  
  
 controller.setStudentName("Jane Doe");  
 controller.setStudentGrade("B");  
 controller.updateView();  
 }  
}

### **Exercise 11: Implementing Dependency Injection**

#### **1. Create a New Java Project**

Create a new Java project named DependencyInjectionExample.

#### **2. Define Repository Interface**

**CustomerRepository.java**:

public interface CustomerRepository {  
 Customer findCustomerById(int id);  
}

#### **3. Implement Concrete Repository**

**CustomerRepositoryImpl.java**:

public class CustomerRepositoryImpl implements CustomerRepository {  
 @Override  
 public Customer findCustomerById(int id) {  
 // Simulate fetching a customer from a database  
 return new Customer("John Doe", id);  
 }  
}

#### **4. Define Service Class**

**CustomerService.java**:

public class CustomerService {  
 private CustomerRepository repository;  
  
 public CustomerService(CustomerRepository repository) {  
 this.repository = repository;  
 }  
  
 public Customer getCustomer(int id) {  
 return repository.findCustomerById(id);  
 }  
}

#### **5. Implement Dependency Injection**

Dependency injection is implemented through constructor injection in CustomerService.

#### **6. Test the Dependency Injection Implementation**

**DependencyInjectionTest.java**:

public class DependencyInjectionTest {  
 public static void main(String[] args) {  
 CustomerRepository repository = new CustomerRepositoryImpl();  
 CustomerService service = new CustomerService(repository);  
  
 Customer customer = service.getCustomer(1);  
 System.out.println("Customer Name: " + customer.getName());  
 }  
}

**Customer.java**:

public class Customer {  
 private String name;  
 private int id;  
  
 public Customer(String name, int id) {  
 this.name = name;  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public int getId() {  
 return id;  
 }  
}