| **Exercise 1: Implementing the Singleton Pattern**  **Scenario:**  You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.  **Solution:**  **Logger.java**  public class Logger {  private static Logger singleInstance;  private Logger() {  System.out.println("Logger initialized");  }  public static Logger getInstance() {  if (singleInstance == null) {  singleInstance = new Logger();  }  return singleInstance;  }  public void log(String message) {  System.out.println("Log: " + message);  }  }  **TestLogger.java**  public class TestLogger {  public static void main(String[] args) {  Logger logger1 = Logger.getInstance();  logger1.log("First log message");  Logger logger2 = Logger.getInstance();  logger2.log("Second log message");  if (logger1 == logger2) {  System.out.println("Both logger instances are the same (Singleton works )");  } else {  System.out.println("Different instances (Singleton failed )");  }  }  }  **output console:**    **Exercise 2: Implementing the Factory Method Pattern**  **Scenario:**  You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.  **Solution:**  **Java Project: FactoryMethodPatternExample**  **Document.java**  public interface Document {  void open();  }  **WordDocument.java**  public class WordDocument implements Document {    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");  }  }  **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();  }  }  **TestDocumentFactory.java**  public class TestDocumentFactory {  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**    **Exercise 3: Implementing the Builder Pattern**  **Scenario:**  You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.  **Solution:**  **Java Project :BuilderPatternExample**  **Computer.java**  public 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 static class Builder {  // Required attributes  private String CPU;  private String RAM;  private String storage;  private String graphicsCard;  public Builder(String CPU, String RAM) {  this.CPU = CPU;  this.RAM = RAM;  }  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 void displayConfiguration() {  System.out.println("CPU: " + CPU);  System.out.println("RAM: " + RAM);  System.out.println("Storage: " + (storage != null ? storage : "Not included"));  System.out.println("Graphics Card: " + (graphicsCard != null ? graphicsCard : "Not included"));  System.out.println("----------------------------------");  }  }  **TestBuilderPattern.java**  public class TestBuilderPattern {  public static void main(String[] args) {  Computer basicComputer = new Computer.Builder("Intel i5", "8GB").build();  Computer gamingComputer = new Computer.Builder("Intel i9", "32GB")  .setStorage("1TB SSD")  .setGraphicsCard("NVIDIA RTX 4090")  .build();  Computer officeComputer = new Computer.Builder("AMD Ryzen 5", "16GB")  .setStorage("512GB SSD")  .build();  System.out.println("Basic Computer:");  basicComputer.displayConfiguration();  System.out.println("Gaming Computer:");  gamingComputer.displayConfiguration();  System.out.println("Office Computer:");  officeComputer.displayConfiguration();  }  }  **Output:**    **Exercise 4: Implementing the Adapter Pattern**  **Scenario:**  You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.  **Solution:**  **Java Project :AdapterPatternExample**  **PaymentProcessor.java**  public interface PaymentProcessor {  void processPayment(double amount);  }  **PayPalGateway.java**  public class PayPalGateway {  public void makePayment(double amountInDollars) {  System.out.println("Processing PayPal payment of $" + amountInDollars);  }  }  **StripeGateway.java**  public class StripeGateway {  public void sendPayment(double money) {  System.out.println("Processing Stripe payment of $" + money);  }  }  **PayPalAdapter.java**  public class PayPalAdapter implements PaymentProcessor {  private PayPalGateway paypal;  public PayPalAdapter(PayPalGateway paypal) {  this.paypal = paypal;  }  @Override  public void processPayment(double amount) {  paypal.makePayment(amount);  }  }  **StripeAdapter.java**  public class StripeAdapter implements PaymentProcessor {  private StripeGateway stripe;  public StripeAdapter(StripeGateway stripe) {  this.stripe = stripe;  }  @Override  public void processPayment(double amount) {  stripe.sendPayment(amount);  }  }  **PaymentTest.java**  public class PaymentTest {  public static void main(String[] args) {  PayPalGateway paypal = new PayPalGateway();  PaymentProcessor paypalAdapter = new PayPalAdapter(paypal);  paypalAdapter.processPayment(250.0);  StripeGateway stripe = new StripeGateway();  PaymentProcessor stripeAdapter = new StripeAdapter(stripe);  stripeAdapter.processPayment(400.0);  }  }  **Output:**    **Exercise 5: Implementing the Decorator Pattern**  **Scenario:**  You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.  **Solution:**  **Java Project:DecoratorPatternExample**  **Notifier.java**  public interface Notifier {  void send(String message);  }  **EmailNotifier.java**  public class EmailNotifier implements Notifier {  @Override  public void send(String message) {  System.out.println("Sending Email: " + message);  }  }  **NotifierDecorator.java**  public abstract class NotifierDecorator implements Notifier {  protected Notifier wrappee;  public NotifierDecorator(Notifier notifier) {  this.wrappee = notifier;  }  @Override  public void send(String message) {  wrappee.send(message);  }  }  **SMSNotifierDecorator.java**  public class SMSNotifierDecorator extends NotifierDecorator {  public SMSNotifierDecorator(Notifier notifier) {  super(notifier);  }  @Override  public void send(String message) {  super.send(message);  sendSMS(message);  }  private void sendSMS(String message) {  System.out.println("Sending SMS: " + message);  }  }  **SlackNotifierDecorator.java**  public class SlackNotifierDecorator extends NotifierDecorator {  public SlackNotifierDecorator(Notifier notifier) {  super(notifier);  }  @Override  public void send(String message) {  super.send(message);  sendSlack(message);  }  private void sendSlack(String message) {  System.out.println("Sending Slack: " + message);  }  }  **NotificationTest.java**  public class NotificationTest {  public static void main(String[] args) {  Notifier basicNotifier = new EmailNotifier();  Notifier smsNotifier = new SMSNotifierDecorator(basicNotifier);  Notifier fullNotifier = new SlackNotifierDecorator(smsNotifier);  fullNotifier.send("System Update: New features released!");  }  }  **Output:**    **Exercise 6: Implementing the Proxy Pattern**  **Scenario:**  You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.  **Solution:**  **Java Project:ProxyPatternExample**  **Image.java**  public interface Image {  void display();  }  **RealImage.java**  public 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...");  }  @Override  public void display() {  System.out.println("Displaying " + fileName);  }  }  **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); // Lazy loading  } else {  System.out.println(fileName + " already loaded. Using cached image.");  }  realImage.display();  }  }  **ProxyTest.java**  public class ProxyTest {  public static void main(String[] args) {  Image image1 = new ProxyImage("photo1.jpg");  Image image2 = new ProxyImage("photo2.jpg");  image1.display();  image1.display();  image2.display();  image2.display();  }  }  **Output:**    **Exercise 7: Implementing the Observer Pattern**  **Scenario:**  You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.  **Solution:**  **Java Project:ObserverPatternExample**  **Stock.java**  public interface Stock {  void registerObserver(Observer o);  void removeObserver(Observer o);  void notifyObservers();  }  **Observer.java**  public interface Observer {  void update(String stockName, double price);  }  **StockMarket.java**  import java.util.ArrayList;  import java.util.List;  public 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();  }  @Override  public void registerObserver(Observer o) {  observers.add(o);  }  @Override  public void removeObserver(Observer o) {  observers.remove(o);  }  @Override  public void notifyObservers() {  for (Observer o : observers) {  o.update(stockName, price);  }  }  }  **MobileApp.java**  public class MobileApp implements Observer {  private String user;  public MobileApp(String user) {  this.user = user;  }  @Override  public void update(String stockName, double price) {  System.out.println(user + "'s Mobile App: " + stockName + " is now ₹" + price);  }  }  **WebApp.java**  public class WebApp implements Observer {  private String user;  public WebApp(String user) {  this.user = user;  }  @Override  public void update(String stockName, double price) {  System.out.println(user + "'s Web App: " + stockName + " is now ₹" + price);  }  }  **ObserverPatternTest.java**  public class ObserverPatternTest {  public static void main(String[] args) {  StockMarket stockMarket = new StockMarket();  Observer mobileHarini = new MobileApp("Harini");  Observer webAbinaya = new WebApp("Abinaya");  stockMarket.registerObserver(mobileHarini);  stockMarket.registerObserver(webAbinaya);  // First stock price change  stockMarket.setStock("TCS", 3750.00);  stockMarket.setStock("Infosys", 1555.00);  stockMarket.removeObserver(webAbinaya);  stockMarket.setStock("Wipro", 456.75);  }  }  **Output:**    **Exercise 8: Implementing the Strategy Pattern**  **Scenario:**  You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.  **Solution:**  **Java Project:StrategyPatternExample**  **PaymentStrategy.java**  public interface PaymentStrategy {  void pay(double amount);  }  **CreditCardPayment.java**  public class CreditCardPayment implements PaymentStrategy {  private String cardNumber;  private String cardHolderName;  public CreditCardPayment(String cardNumber, String cardHolderName) {  this.cardNumber = cardNumber;  this.cardHolderName = cardHolderName;  }  @Override  public void pay(double amount) {  System.out.println("Paid ₹" + amount + " using Credit Card [" + cardHolderName + "]");  }  }  **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("Paid ₹" + amount + " using PayPal [" + email + "]");  }  }  **PaymentContext.java**  public class PaymentContext {  private PaymentStrategy strategy;  public void setPaymentStrategy(PaymentStrategy strategy) {  this.strategy = strategy;  }  public void processPayment(double amount) {  if (strategy != null) {  strategy.pay(amount);  } else {  System.out.println("Payment strategy not set!");  }  }  }  **StrategyTest.java**  public class StrategyTest {  public static void main(String[] args) {  PaymentContext context = new PaymentContext();  context.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432", "Harini G"));  context.processPayment(1500.0);  context.setPaymentStrategy(new PayPalPayment("harini.pay@example.com"));  context.processPayment(800.0);  }  } **Output:**   **Exercise 9: Implementing the Command Pattern**  **Scenario:**  You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.  **Solution:**  **Java Project:CommandPatternExample**  **Command.java**  public interface Command {  void execute();  }  **Light.java**  public class Light {  public void turnOn() {  System.out.println("Light is ON");  }  public void turnOff() {  System.out.println("Light is OFF");  }  }  **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();  }  }  **RemoteControl.java**  public 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 set.");  }  }  }  **CommandPatternTest.java**  public class CommandPatternTest {  public static void main(String[] args) {  Light livingRoomLight = new Light();  Command lightOn = new LightOnCommand(livingRoomLight);  Command lightOff = new LightOffCommand(livingRoomLight);  RemoteControl remote = new RemoteControl();  remote.setCommand(lightOn);  remote.pressButton();  remote.setCommand(lightOff);  remote.pressButton();  }  }  **Output:**    **Exercise 10: Implementing the MVC Pattern**  **Scenario:**  You are developing a simple web application for managing student records using the MVC pattern.  **Solution:**  **Java Project:MVCPatternExample**  **Student.java**  public class Student {  private String name;  private String id;  private String grade;  public Student(String name, String id, String grade) {  this.name = name;  this.id = id;  this.grade = 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; }  }  **StudentView.java**  public 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);  System.out.println("-------------------------");  }  }  **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(String id) {  model.setId(id);  }  public String 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());  }  }  **MVCTest.java**  public class MVCTest {  public static void main(String[] args) {  // Create the model  Student student = new Student("Harini G", "CSE1023", "A");  // Create the view  StudentView view = new StudentView();  // Create the controller  StudentController controller = new StudentController(student, view);  // Display initial data  controller.updateView();  // Update model via controller  controller.setStudentGrade("A+");  controller.setStudentName("Harini Govindasamy");  // Display updated data  controller.updateView();  }  }  **Output:**    **Exercise 11: Implementing Dependency Injection**  **Scenario:**  You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.  **Solution:**  **Java Project:DependencyInjectionExample**  **Customer.java**  public class Customer {  private int id;  private String name;  public Customer(int id, String name) {  this.id = id;  this.name = name;  }  public int getId() { return id; }  public String getName() { return name; }  @Override  public String toString() {  return "Customer [ID: " + id + ", Name: " + name + "]";  }  }  **CustomerRepository.java**  public interface CustomerRepository {  Customer findCustomerById(int id);  }  **CustomerRepositoryImpl.java**  import java.util.HashMap;  import java.util.Map;  public class CustomerRepositoryImpl implements CustomerRepository {  private Map<Integer, Customer> customers = new HashMap<>();  public CustomerRepositoryImpl() {  customers.put(1, new Customer(1, "Harini"));  customers.put(2, new Customer(2, "Abinaya"));  }  @Override  public Customer findCustomerById(int id) {  return customers.getOrDefault(id, null);  }  }  **CustomerService.java**  public class CustomerService {  private CustomerRepository customerRepository;  // Constructor injection  public CustomerService(CustomerRepository customerRepository) {  this.customerRepository = customerRepository;  }  public void displayCustomerById(int id) {  Customer customer = customerRepository.findCustomerById(id);  if (customer != null) {  System.out.println("Customer Found: " + customer);  } else {  System.out.println("Customer Not Found with ID: " + id);  }  }  }  **DependencyInjectionTest.java**  public class DependencyInjectionTest {  public static void main(String[] args) {  // Create repository implementation  CustomerRepository repository = new CustomerRepositoryImpl();  // Inject repository into service  CustomerService service = new CustomerService(repository);  // Use the service  service.displayCustomerById(1);  service.displayCustomerById(2);  service.displayCustomerById(3); // Not found  }  }  **Output:** |
| --- |