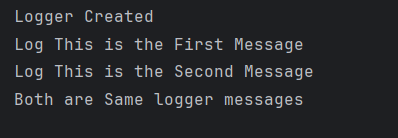
**Exercise 1: Implementing the Singleton Pattern**

**Code :**

public class Logger {  
 private static Logger singleInstance;  
 private Logger(){  
 System.out.println("Logger Created");  
 }  
 public static Logger getInstance(){  
 if(singleInstance == null) singleInstance = new Logger();  
 return singleInstance;  
 }  
 public void log(String message){  
 System.out.println("Log "+message);  
 }  
}  
class Main{  
 public static void main(String[] args) {  
 Logger logger1 = Logger.getInstance();  
 Logger logger2 = Logger.getInstance();  
 logger1.log("This is the First Message");  
 logger2.log("This is the Second Message");  
  
 if (logger1 == logger2){  
 System.out.println("Both are Same logger messages");  
 }  
 else{  
 System.out.println("Different instances (wrong!)");  
 }  
 }  
}

**Output :**



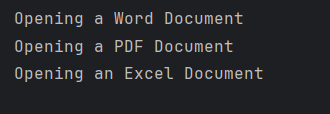
**Exercise 2: Implementing the Factory Method Pattern**

**Code :**

interface Document {  
 void open();  
}

class WordDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening a Word Document");  
 }  
}  
class PdfDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening a PDF Document");  
 }  
}  
class ExcelDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening an Excel Document");  
 }  
}  
abstract class DocumentFactory {  
 public abstract Document createDocument();  
}  
class WordDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new WordDocument();  
 }  
}  
class PdfDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new PdfDocument();  
 }  
}  
class ExcelDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new ExcelDocument();  
 }  
}  
public class FactoryMethodPatternExample {  
 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 :**

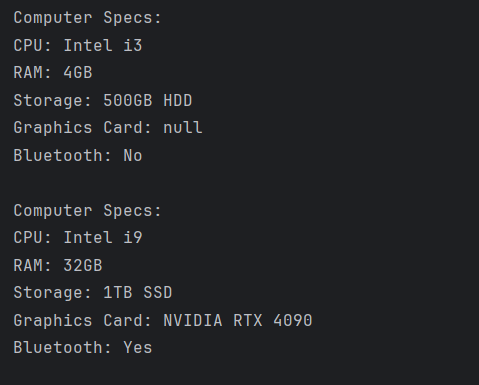


**Exercise 3: Implementing the Builder Pattern**

**Code:**

class Computer {  
 private String CPU;  
 private String RAM;  
 private String storage;  
 private String graphicsCard;  
 private boolean hasBluetooth;  
  
 private Computer(Builder builder) {  
 this.CPU = builder.CPU;  
 this.RAM = builder.RAM;  
 this.storage = builder.storage;  
 this.graphicsCard = builder.graphicsCard;  
 this.hasBluetooth = builder.hasBluetooth;  
 }  
  
 public static class Builder {  
 private String CPU;  
 private String RAM;  
 private String storage;  
 private String graphicsCard;  
 private boolean hasBluetooth;  
  
 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 Builder setBluetooth(boolean hasBluetooth) {  
 this.hasBluetooth = hasBluetooth;  
 return this;  
 }  
  
 public Computer build() {  
 return new Computer(this);  
 }  
 }  
  
 public void showSpecs() {  
 System.*out*.println("Computer Specs:");  
 System.*out*.println("CPU: " + CPU);  
 System.*out*.println("RAM: " + RAM);  
 System.*out*.println("Storage: " + storage);  
 System.*out*.println("Graphics Card: " + graphicsCard);  
 System.*out*.println("Bluetooth: " + (hasBluetooth ? "Yes" : "No"));  
 System.*out*.println();  
 }  
}  
  
public class BuilderPatternExample {  
 public static void main(String[] args) {  
 Computer basicComputer = new Computer.Builder()  
 .setCPU("Intel i3")  
 .setRAM("4GB")  
 .setStorage("500GB HDD")  
 .build();  
  
 Computer gamingComputer = new Computer.Builder()  
 .setCPU("Intel i9")  
 .setRAM("32GB")  
 .setStorage("1TB SSD")  
 .setGraphicsCard("NVIDIA RTX 4090")  
 .setBluetooth(true)  
 .build();  
  
 basicComputer.showSpecs();  
 gamingComputer.showSpecs();  
 }  
}

**Output:**

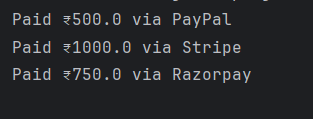
****

**Exercise 4: Implementing the Adapter Pattern**

**Code :**

interface PaymentProcessor {  
 void processPayment(double amount);  
}  
  
class PayPalGateway {  
 public void sendMoney(double amount) {  
 System.*out*.println("Paid ₹" + amount + " via PayPal");  
 }  
}  
class StripeGateway {  
 public void makePayment(double amount) {  
 System.*out*.println("Paid ₹" + amount + " via Stripe");  
 }  
}  
class RazorpayGateway {  
 public void doTransaction(double amount) {  
 System.*out*.println("Paid ₹" + amount + " via Razorpay");  
 }  
}  
  
class PayPalAdapter implements PaymentProcessor {  
 private PayPalGateway payPal;  
  
 public PayPalAdapter() {  
 payPal = new PayPalGateway();  
 }  
  
 public void processPayment(double amount) {  
 payPal.sendMoney(amount);  
 }  
}  
  
class StripeAdapter implements PaymentProcessor {  
 private StripeGateway stripe;  
  
 public StripeAdapter() {  
 stripe = new StripeGateway();  
 }  
  
 public void processPayment(double amount) {  
 stripe.makePayment(amount);  
 }  
}  
  
class RazorpayAdapter implements PaymentProcessor {  
 private RazorpayGateway razorpay;  
  
 public RazorpayAdapter() {  
 razorpay = new RazorpayGateway();  
 }  
  
 public void processPayment(double amount) {  
 razorpay.doTransaction(amount);  
 }  
}  
public class AdapterPatternExample {  
 public static void main(String[] args) {  
 PaymentProcessor payPal = new PayPalAdapter();  
 PaymentProcessor stripe = new StripeAdapter();  
 PaymentProcessor razorpay = new RazorpayAdapter();  
  
 payPal.processPayment(500.00);  
 stripe.processPayment(1000.00);  
 razorpay.processPayment(750.00);  
 }  
}

**Output :**

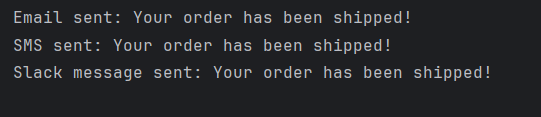
****

**Exercise 5: Implementing the Decorator Pattern**

**Code:**

interface Notifier {  
 void send(String message);  
}  
class EmailNotifier implements Notifier {  
 public void send(String message) {  
 System.*out*.println("Email sent: " + message);  
 }  
}  
abstract class NotifierDecorator implements Notifier {  
 protected Notifier wrappedNotifier;  
 public NotifierDecorator(Notifier notifier) {  
 this.wrappedNotifier = notifier;  
 }  
 public void send(String message) {  
 wrappedNotifier.send(message);   
 }  
}  
class SMSNotifierDecorator extends NotifierDecorator {  
 public SMSNotifierDecorator(Notifier notifier) {  
 super(notifier);  
 }  
 public void send(String message) {  
 super.send(message);  
 sendSMS(message);  
 }  
 private void sendSMS(String message) {  
 System.*out*.println("SMS sent: " + message);  
 }  
}  
  
class SlackNotifierDecorator extends NotifierDecorator {  
 public SlackNotifierDecorator(Notifier notifier) {  
 super(notifier);  
 }  
 public void send(String message) {  
 super.send(message);  
 sendSlack(message);  
 }  
 private void sendSlack(String message) {  
 System.*out*.println("Slack message sent: " + message);  
 }  
}  
  
public class DecoratorPatternExample {  
 public static void main(String[] args) {  
 Notifier emailNotifier = new EmailNotifier();  
 Notifier emailAndSMS = new SMSNotifierDecorator(emailNotifier);  
 Notifier multiChannel = new SlackNotifierDecorator(emailAndSMS);  
 multiChannel.send("Your order has been shipped!");  
 }  
}

**Output :**

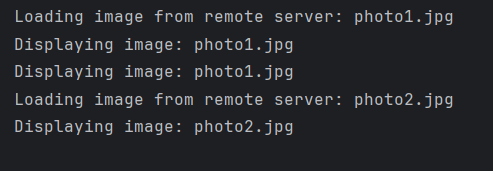
****

**Exercise 6: Implementing the Proxy Pattern**

**Code :**

interface Image {  
 void display();  
}  
class RealImage implements Image {  
 private String fileName;  
 public RealImage(String fileName) {  
 this.fileName = fileName;  
 loadFromRemoteServer(); // simulate heavy loading  
 }  
 private void loadFromRemoteServer() {  
 System.*out*.println("Loading image from remote server: " + fileName);  
 }  
 public void display() {  
 System.*out*.println("Displaying image: " + 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 ProxyPatternExample {  
 public static void main(String[] args) {  
 Image image1 = new ProxyImage("photo1.jpg");  
 Image image2 = new ProxyImage("photo2.jpg");  
 image1.display();  
  
 image1.display();  
 image2.display();  
 }  
}

**Output :**

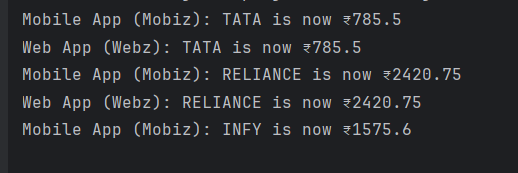
****

**Exercise 7: Implementing the Observer Pattern**

**Code :**

import java.util.\*;  
  
interface Stock {  
 void registerObserver(Observer o);  
 void removeObserver(Observer o);  
 void notifyObservers();  
}  
  
interface Observer {  
 void update(String stockName, double price);  
}  
class StockMarket implements Stock {  
 private List<Observer> observers = new ArrayList<>();  
 private String stockName;  
 private double stockPrice;  
  
 public void setStockData(String name, double price) {  
 this.stockName = name;  
 this.stockPrice = price;  
 notifyObservers();  
 }  
 public void registerObserver(Observer o) {  
 observers.add(o);  
 }  
 public void removeObserver(Observer o) {  
 observers.remove(o);  
 }  
 public void notifyObservers() {  
 for (Observer o : observers) {  
 o.update(stockName, stockPrice);  
 }  
 }  
}  
class MobileApp implements Observer {  
 private String user;  
 public MobileApp(String user) {  
 this.user = user;  
 }  
 public void update(String stockName, double price) {  
 System.*out*.println("Mobile App (" + user + "): " + stockName + " is now ₹" + price);  
 }  
}  
  
class WebApp implements Observer {  
 private String user;  
 public WebApp(String user) {  
 this.user = user;  
 }  
 public void update(String stockName, double price) {  
 System.*out*.println("Web App (" + user + "): " + stockName + " is now ₹" + price);  
 }  
}  
public class ObserverPatternExample {  
 public static void main(String[] args) {  
 StockMarket stockMarket = new StockMarket();  
  
 Observer mobileUser = new MobileApp("Mobiz");  
 Observer webUser = new WebApp("Webz");  
  
 stockMarket.registerObserver(mobileUser);  
 stockMarket.registerObserver(webUser);  
  
 stockMarket.setStockData("TATA", 785.50);  
 stockMarket.setStockData("RELIANCE", 2420.75);  
  
 stockMarket.removeObserver(webUser);  
 stockMarket.setStockData("INFY", 1575.60);  
 }  
}

**Output :**

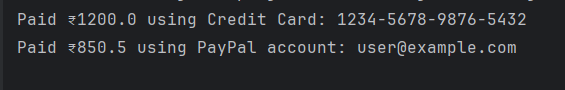
****

**Exercise 8: Implementing the Strategy Pattern**

**Code :**

interface PaymentStrategy {  
 void pay(double amount);  
}  
  
class CreditCardPayment implements PaymentStrategy {  
 private String cardNumber;  
 public CreditCardPayment(String cardNumber) {  
 this.cardNumber = cardNumber;  
 }  
 public void pay(double amount) {  
 System.*out*.println("Paid ₹" + amount + " using Credit Card: " + cardNumber);  
 }  
}  
  
class PayPalPayment implements PaymentStrategy {  
 private String email;  
 public PayPalPayment(String email) {  
 this.email = email;  
 }  
 public void pay(double amount) {  
 System.*out*.println("Paid ₹" + amount + " using PayPal account: " + email);  
 }  
}  
  
class PaymentContext {  
 private PaymentStrategy strategy;  
 public void setPaymentStrategy(PaymentStrategy strategy) {  
 this.strategy = strategy;  
 }  
 public void executePayment(double amount) {  
 if (strategy == null) {  
 System.*out*.println("Payment method not set!");  
 } else {  
 strategy.pay(amount);  
 }  
 }  
}  
public class StrategyPatternExample {  
 public static void main(String[] args) {  
 PaymentContext context = new PaymentContext();  
  
 context.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));  
 context.executePayment(1200.00);  
  
 context.setPaymentStrategy(new PayPalPayment("user@example.com"));  
 context.executePayment(850.50);  
 }  
}

**Output :**

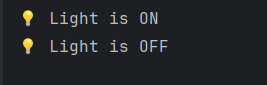
****

**Exercise 9: Implementing the Command Pattern**

**Code :**

interface Command {  
 void execute();  
}  
  
class Light {  
 public void turnOn() {  
 System.*out*.println("💡 Light is ON");  
 }  
 public void turnOff() {  
 System.*out*.println("💡 Light is OFF");  
 }  
}  
  
class LightOnCommand implements Command {  
 private Light light;  
 public LightOnCommand(Light light) {  
 this.light = light;  
 }  
 public void execute() {  
 light.turnOn();  
 }  
}  
  
class LightOffCommand implements Command {  
 private Light light;  
 public LightOffCommand(Light light) {  
 this.light = light;  
 }  
 public void execute() {  
 light.turnOff();  
 }  
}  
  
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!");  
 }  
 }  
}  
  
public class CommandPatternExample {  
 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 :**

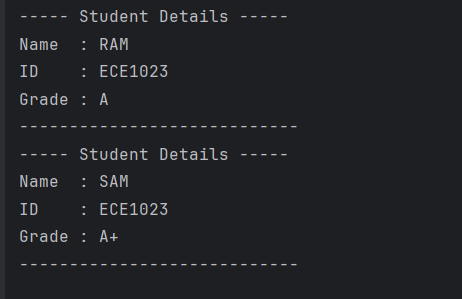
****

**Exercise 10: Implementing the MVC Pattern**

**Code :**

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; }  
}  
  
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("----------------------------");  
 }  
}  
  
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 String getStudentName() { return student.getName(); }  
 public String getStudentId() { return student.getId(); }  
 public String getStudentGrade() { return student.getGrade(); }  
  
 public void updateView() {  
 view.displayStudentDetails(student.getName(), student.getId(), student.getGrade());  
 }  
}  
  
public class MVCPatternExample {  
 public static void main(String[] args) {  
 Student student = new Student("RAM", "ECE1023", "A");  
  
 StudentView view = new StudentView();  
  
 StudentController controller = new StudentController(student, view);  
  
 controller.updateView();  
  
 controller.setStudentName("SAM");  
 controller.setStudentGrade("A+");  
  
 controller.updateView();  
 }  
}

**Output :**

****

**Exercise 11: Implementing Dependency Injection**

**Code :**

interface CustomerRepository {  
 String findCustomerById(String id);  
}  
  
class CustomerRepositoryImpl implements CustomerRepository {  
 public String findCustomerById(String id) {  
 return "Customer{id='" + id + "', name='Santhosh Kannan'}";  
 }  
}  
  
class CustomerService {  
 private final CustomerRepository repository;  
 public CustomerService(CustomerRepository repository) {  
 this.repository = repository;  
 }  
 public void displayCustomer(String id) {  
 String customer = repository.findCustomerById(id);  
 System.*out*.println("Fetched: " + customer);  
 }  
}  
  
public class DependencyInjectionExample {  
 public static void main(String[] args) {  
 CustomerRepository repo = new CustomerRepositoryImpl();  
 CustomerService service = new CustomerService(repo);  
  
 service.displayCustomer("CUST001");  
 }  
}

**Output :**

****