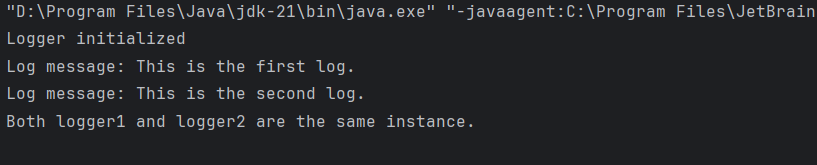
**Exercise 1: Implementing the Singleton Pattern**

**Program:**

package singleton;  
  
public 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: " + message);  
 }  
}

package singleton;  
  
public class Main {  
 public static void main(String[] args) {  
 Logger logger1 = Logger.*getInstance*();  
 Logger logger2 = Logger.*getInstance*();  
  
 logger1.log("This is the first log.");  
 logger2.log("This is the second log.");  
  
 if (logger1 == logger2) {  
 System.*out*.println("Both logger1 and logger2 are the same instance.");  
 } else {  
 System.*out*.println("Different logger instances.");  
 }  
 }  
}



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

**Program:**

public class Main {  
 public static void main(String[] args) {  
 DocumentFactory factory;  
  
 factory = new WordDocumentFactory();  
 Document doc1 = factory.createDocument();  
 doc1.open();  
  
 factory = new PdfDocumentFactory();  
 Document doc2 = factory.createDocument();  
 doc2.open();  
  
 factory = new ExcelDocumentFactory();  
 Document doc3 = factory.createDocument();  
 doc3.open();  
 }  
}

public interface Document {  
 void open();  
}

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

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

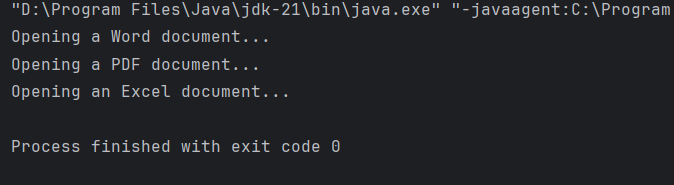
public class ExcelDocument implements Document {  
 public void open() {  
 System.*out*.println("Opening an Excel document...");  
 }  
}

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

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

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

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

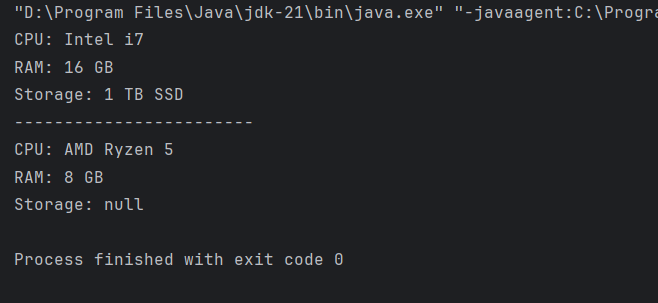


**Exercise 3: Implementing the Builder Pattern**

**Program:**

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 void showSpecs() {  
 System.*out*.println("CPU: " + CPU);  
 System.*out*.println("RAM: " + RAM);  
 System.*out*.println("Storage: " + 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 class Main {  
 public static void main(String[] args) {  
 Computer comp1 = new Computer.Builder()  
 .setCPU("Intel i7")  
 .setRAM("16 GB")  
 .setStorage("1 TB SSD")  
 .build();  
  
 comp1.showSpecs();  
  
 System.*out*.println("------------------------");  
  
 Computer comp2 = new Computer.Builder()  
 .setCPU("AMD Ryzen 5")  
 .setRAM("8 GB")  
 .build(); // Storage not set  
  
 comp2.showSpecs();  
 }  
}

****

**Exercise 4: Implementing the Adapter Pattern**

**Program:**

public class Main {  
 public static void main(String[] args) {  
 PaymentProcessor paypal = new PayPalAdapter();  
 paypal.processPayment(1500.00);  
  
 PaymentProcessor stripe = new StripeAdapter();  
 stripe.processPayment(2000.00);  
  
 PaymentProcessor razor = new RazorAdapter();  
 razor.processPayment(2500.00);  
 }  
}

public class RazorAdapter implements PaymentProcessor {  
 private RazorPayGateway razor = new RazorPayGateway();  
  
 public void processPayment(double amount) {  
 razor.tryPay(amount);  
 }  
}

public class StripeAdapter implements PaymentProcessor {  
 private StripeGateway stripe = new StripeGateway();  
  
 public void processPayment(double amount) {  
 stripe.doPay(amount);  
 }  
}

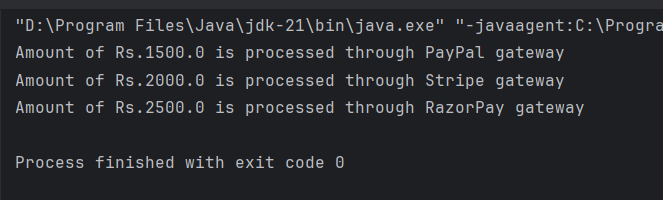
public class PayPalAdapter implements PaymentProcessor {  
 private PayPalGateway paypal = new PayPalGateway();  
  
 public void processPayment(double amount) {  
 paypal.sendPay(amount);  
 }  
}

public class RazorPayGateway {  
 public void tryPay(double amount) {  
 System.*out*.println("Amount of Rs." + amount + " is processed through RazorPay gateway");  
 }  
}

public class StripeGateway {  
 public void doPay(double amount) {  
 System.*out*.println("Amount of Rs." + amount + " is processed through Stripe gateway");  
 }  
}

public class PayPalGateway {  
 public void sendPay(double amount) {  
 System.*out*.println("Amount of Rs." + amount + " is processed through PayPal gateway");  
 }  
}

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



**Exercise 5: Implementing the Decorator Pattern**

**Program:**

interface Notifier {

void send(String message);

}

class EmailNotifier implements Notifier {

public void send(String message) {

System.out.println("Sending Email: " + message);

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

public class DecoratorPatternExample {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

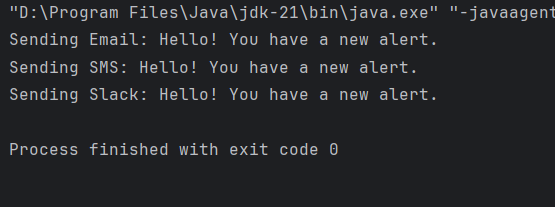
notifier = new SMSNotifierDecorator(notifier);

notifier = new SlackNotifierDecorator(notifier);

notifier.send("Hello! You have a new alert.");

}

}



**Exercise 6: Implementing the Proxy Pattern**

**Program:**

public class ProxyPatternExample {

interface Image {

void display();

}

static class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

System.out.println("Loading image from remote server: " + 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 image1 = new ProxyImage("scenery1.jpg");

Image image2 = new ProxyImage("scenery2.jpg");

image1.display();

System.out.println();

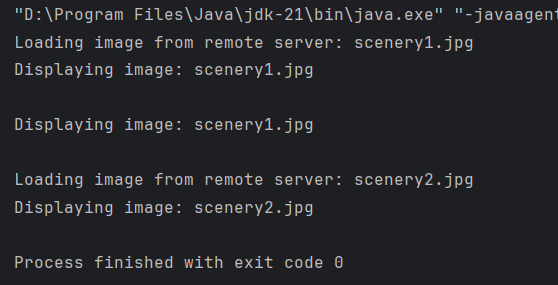
image1.display();

System.out.println();

image2.display();

}

}

****

**Exercise 7: Implementing the Observer Pattern**

**Program:**

import java.util.\*;

public class ObserverPatternExample {

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

interface Observer {

void update(int stockPrice);

}

static class StockMarket implements Stock {

private List<Observer> observers = new ArrayList<>();

private int stockPrice;

public void setStockPrice(int price) {

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(stockPrice);

}

}

}

static class MobileApp implements Observer {

public void update(int stockPrice) {

System.out.println("MobileApp: Stock price is Rs." + stockPrice);

}

}

static class WebApp implements Observer {

public void update(int stockPrice) {

System.out.println("WebApp: Stock price is Rs." + stockPrice);

}

}

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.setStockPrice(150);

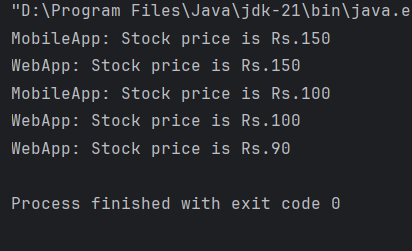
market.setStockPrice(100);

market.deregister(mobile);

market.setStockPrice(90);

}

}

****

**Exercise 8: Implementing the Strategy Pattern**

**Program:**

public class StrategyPatternExample {

interface PaymentStrategy {

void pay();

}

static class CreditCardPayment implements PaymentStrategy {

public void pay() {

System.out.println("Paying using credit card");

}

}

static class PayPalPayment implements PaymentStrategy {

public void pay() {

System.out.println("Paying using paypal");

}

}

static class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void startPayment() {

strategy.pay();

}

}

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setStrategy(new CreditCardPayment());

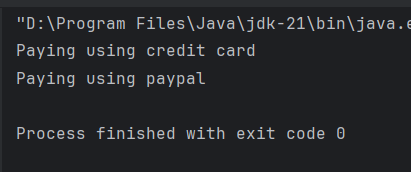
context.startPayment();

context.setStrategy(new PayPalPayment());

context.startPayment();

}

}



**Exercise 9: Implementing the Command Pattern**

**Program:**

public class CommandPatternExample {

interface Command {

void execute();

}

static class Light {

public void turnOn() {

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

}

public void turnOff() {

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.turnOn();

}

}

static class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

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 onCommand = new LightOnCommand(light);

Command offCommand = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(onCommand);

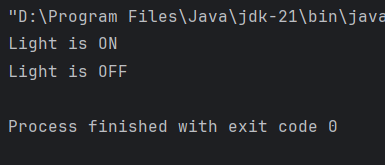
remote.pressButton();

remote.setCommand(offCommand);

remote.pressButton();

}

}



**Exercise 10: Implementing the MVC Pattern**

**Program:**

public class MVCPatternExample {

static class Student {

private String name;

private int id;

private char grade;

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 char getGrade() {

return grade;

}

public void setGrade(char grade) {

this.grade = grade;

}

}

static class StudentView {

public void displayStudentDetails(String name, int id, char grade) {

System.out.println("Student Details:");

System.out.println("Name: " + name);

System.out.println("Id: " + id);

System.out.println("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 setStudentName(String name) {

model.setName(name);

}

public void setStudentId(int id) {

model.setId(id);

}

public void setGrade(char grade) {

model.setGrade(grade);

}

public void updateView() {

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

}

}

public static void main(String[] args) {

Student student = new Student();

student.setName("Amit");

student.setId(1);

student.setGrade('A');

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Raj");

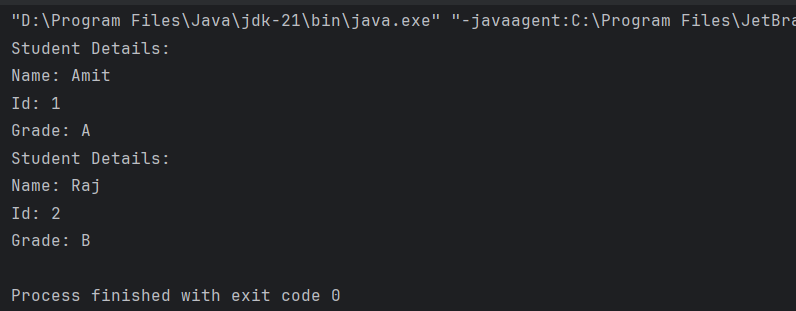
controller.setStudentId(2);

controller.setGrade('B');

controller.updateView();

}

}



**Exercise 11: Implementing Dependency Injection**

**Program:**

public class DependencyInjectionExample {

interface CustomerRepository {

void findCustomerById(int id);

}

static class CustomerRepositoryImpl implements CustomerRepository {

public void findCustomerById(int id) {

System.out.println("Customer with ID " + id + " is found in the database.");

}

}

static class CustomerService {

private CustomerRepository customerRepository;

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void getCustomer(int id) {

customerRepository.findCustomerById(id);

}

}

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.getCustomer(11);

}

}