**Exercise 1: Implementing the Singleton Pattern**

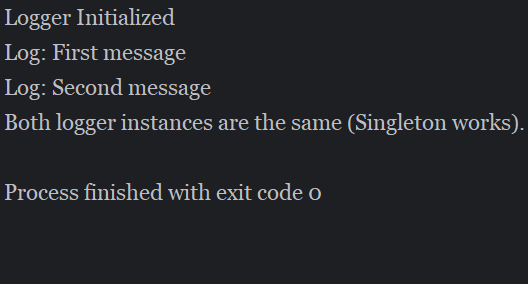
**Code:**

Logger.java

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);  
 }  
}

Main.java

public class Main {  
 public static void main(String[] args) {  
 Logger logger1 = Logger.*getInstance*();  
 logger1.log("First message");  
 Logger logger2 = Logger.*getInstance*();  
 logger2.log("Second message");  
 if (logger1 == logger2) {  
 System.*out*.println("Both logger instances are the same (Singleton works).");  
 } else {  
 System.*out*.println("Different instances (Singleton failed).");  
 }  
 }  
}

**Output:**

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

**Code:**

Document.java

public interface Document {  
 void open();  
}

WordDocument.java

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

PdfDocument.java

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

ExcelDocument.java

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

DocumentFactory.java

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

WordDocumentFactory.java

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

PdfDocumentFactory.java

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

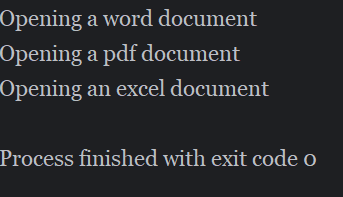
ExcelDocumentFactory.java

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

Main.java

public class Main {  
 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:**

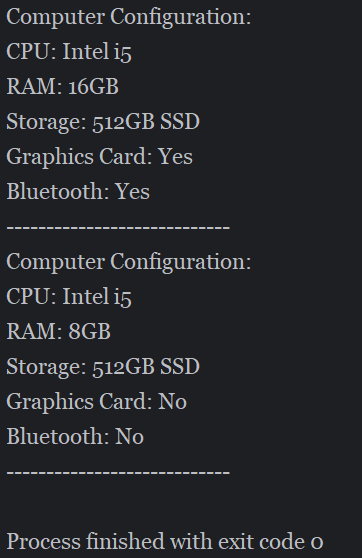
Computer.java

public class Computer {  
 private final String CPU;  
 private final String RAM;  
 private final String storage;  
 private final boolean hasGraphicsCard;  
 private final boolean hasBluetooth;  
  
 private Computer(Builder builder) {  
 this.CPU = builder.CPU;  
 this.RAM = builder.RAM;  
 this.storage = builder.storage;  
 this.hasGraphicsCard = builder.hasGraphicsCard;  
 this.hasBluetooth = builder.hasBluetooth;  
 }  
  
 public static class Builder {  
 private final String CPU;  
 private final String RAM;  
 private final String storage;  
  
 private boolean hasGraphicsCard = false;  
 private boolean hasBluetooth = false;  
  
 public Builder(String CPU, String RAM, String storage) {  
 this.CPU = CPU;  
 this.RAM = RAM;  
 this.storage = storage;  
 }  
 public Builder withGraphicsCard(boolean value) {  
 this.hasGraphicsCard = value;  
 return this;  
 }  
 public Builder withBluetooth(boolean value) {  
 this.hasBluetooth = value;  
 return this;  
 }  
  
 public Computer build() {  
 return new Computer(this);  
 }  
 }  
 public void displayConfiguration() {  
 System.*out*.println("Computer Configuration:");  
 System.*out*.println("CPU: " + CPU);  
 System.*out*.println("RAM: " + RAM);  
 System.*out*.println("Storage: " + storage);  
 System.*out*.println("Graphics Card: " + (hasGraphicsCard ? "Yes" : "No"));  
 System.*out*.println("Bluetooth: " + (hasBluetooth ? "Yes" : "No"));  
 System.*out*.println("----------------------------");  
 }  
}

Main.java

public class Main {  
 public static void main(String[] args) {  
 Computer gamingPC = new Computer.Builder("Intel i5", "16GB", "512GB SSD")  
 .withGraphicsCard(true)  
 .withBluetooth(true)  
 .build();  
 Computer officePC = new Computer.Builder("Intel i5", "8GB", "512GB SSD")  
 .build();  
 gamingPC.displayConfiguration();  
 officePC.displayConfiguration();  
 }  
}

**Output:**

****

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

PaymentProcessor.java

public interface PaymentProcessor {

void processPayment(double amount);

}

StripeGateway.java

public class StripeGateway {

public void makeStripePayment(double amount) {

System.out.println("Processing payment of ₹" + amount + " via Stripe.");

}

}

PaypalGateway.java

public class PayPalGateway {

public void sendPayment(double amount) {

System.out.println("Processing payment of ₹" + amount + " via PayPal.");

}

}

StripeAdapter.java

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripeGateway;

public StripeAdapter(StripeGateway stripeGateway) {

this.stripeGateway = stripeGateway;

}

public void processPayment(double amount) {

stripeGateway.makeStripePayment(amount);

}

}

PayPalAdapter.java

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPalGateway;

public PayPalAdapter(PayPalGateway payPalGateway) {

this.payPalGateway = payPalGateway;

}

public void processPayment(double amount) {

payPalGateway.sendPayment(amount);

}

}

Main.java

public class Main {

public static void main(String[] args) {

PaymentProcessor stripe = new StripeAdapter(new StripeGateway());

stripe.processPayment(1500.00);

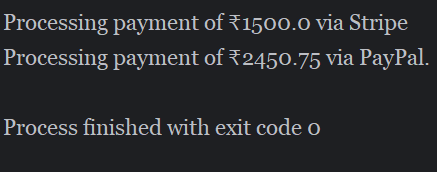
PaymentProcessor paypal = new PayPalAdapter(new PayPalGateway());

paypal.processPayment(2450.75);

}

}

**Output:**

****

**Exercise 5: Implementing the Decorator Pattern**

**Code:**

**Notifier.java**

public interface Notifier {

void send(String message);

}

**EmailNotifier.java**

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

**NotifierDecorator.java**

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send(String message) {

wrappedNotifier.send(message);

}

}

**SMSNotifierDecorator.java**

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

**SlackNotifierDecorator.java**

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(notifier);

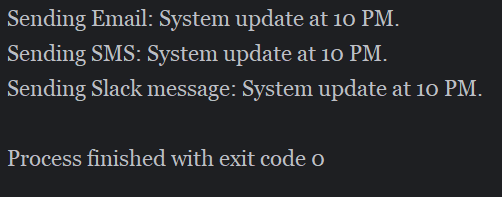
Notifier fullNotifier = new SlackNotifierDecorator(smsNotifier);

fullNotifier.send("System update at 10 PM.");

}

}

**Output:**

****

**Exercise 6: Implementing the Proxy Pattern**

**Code:**

**Image.java**

public interface Image {

void display();

}

**RealImage.java**

public class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromServer();

}

private void loadFromServer() {

System.out.println("Loading image from server: " + filename);

}

public void display() {

System.out.println("Displaying image: " + filename);

}

}

**ProxyImage.java**

public class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename); // Lazy loading

}

realImage.display();

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Image image1 = new ProxyImage("photo1.jpg");

Image image2 = new ProxyImage("photo2.jpg");

image1.display();

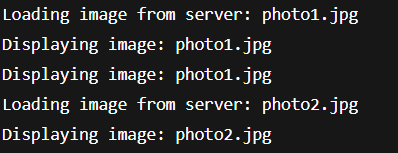
image1.display();

image2.display();

}

}

**Output:**

****

**Exercise 7: Implementing the Observer Pattern**

**Code:**

**Observer.java**

public interface Observer {

void update(String stockName, double price);

}

**Stock.java**

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

**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 stockPrice;

public void setStockData(String stockName, double price) {

this.stockName = stockName;

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 observer : observers) {

observer.update(stockName, stockPrice);

}

}

}

**MobileApp.java**

public class MobileApp implements Observer {

public void update(String stockName, double price) {

System.out.println("Mobile App: " + stockName + " is now ₹" + price);

}

}

**WebApp.java**

public class WebApp implements Observer {

public void update(String stockName, double price) {

System.out.println("Web App: " + stockName + " is now ₹" + price);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

market.registerObserver(mobile);

market.registerObserver(web);

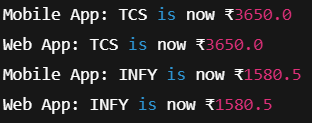
market.setStockData("TCS", 3650.00);

market.setStockData("INFY", 1580.50);

}

}

**Output:**

****

**Exercise 8: Implementing the Strategy Pattern**

**Code:**

**PaymentStrategy.java**

public interface PaymentStrategy {

void pay(double amount);

}

**CreditCardPayment.java**

public class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using Credit Card.");

}

}

**PayPalPayment.java**

public class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using PayPal.");

}

}

**PaymentContext.java**

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

if (strategy == null) {

System.out.println("No payment strategy selected.");

} else {

strategy.pay(amount);

}

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment());

context.executePayment(1000.00);

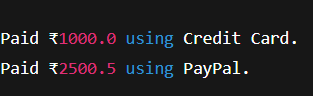
context.setPaymentStrategy(new PayPalPayment());

context.executePayment(2500.50);

}

}

**Output:**

****

**Exercise 9: Implementing the Command Pattern**

**Code:**

**Command.java**

public interface Command {

void execute();

}

**Light.java**

public class Light {

public void turnOn() {

System.out.println("The light is ON.");

}

public void turnOff() {

System.out.println("The light is OFF.");

}

}

**LightOnCommand.java**

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

**LightOffCommand.java**

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

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.");

}

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightsOn = new LightOnCommand(livingRoomLight);

Command lightsOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightsOn);

remote.pressButton();

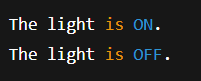
remote.setCommand(lightsOff);

remote.pressButton();

}

}

**Output:**

****

**Exercise 10: Implementing the MVC Pattern**

**Code:**

**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 String getId() { return id; }

public String getGrade() { return grade; }

public void setName(String name) { this.name = name; }

public void setId(String id) { this.id = id; }

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 void setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public void updateView() {

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Student student = new Student("John Doe", "S123", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Jane Smith");

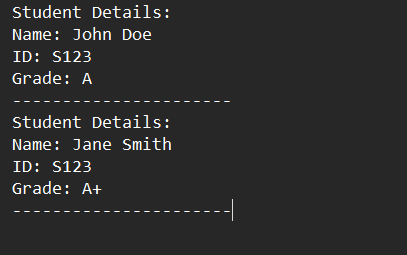
controller.setStudentGrade("A+");

controller.updateView();

}

}

**Output:**

****

**Exercise 11: Implementing Dependency Injection**

**Code:**

**CustomerRepository.java**

public interface CustomerRepository {

String findCustomerById(String customerId);

}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String customerId) {

return "Customer[id=" + customerId + ", name=John Doe]";

}

}

**CustomerService.java**

public class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomerDetails(String customerId) {

String customer = repository.findCustomerById(customerId);

System.out.println("Retrieved: " + customer);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl(); // Dependency created

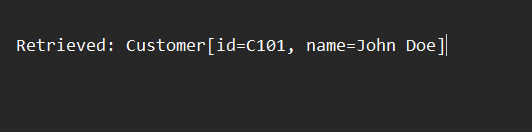
CustomerService service = new CustomerService(repository); // Injected into service

service.getCustomerDetails("C101");

}

}

**Output:**

****