**Exercise 1: Implementing the Singleton Pattern**

**Main.java**  
  
//Singleton pattern

public class Main {

public static void main(String[] args) {

//Logger obj = new Logger();

// we cannot call the Logger obj because the constructor is private

Logger obj1 = Logger.*getObject*();

Logger obj2 = Logger.*getObject*();

System.***out***.println(obj1+"\n"+obj2);

if(obj1.equals(obj2)) {

System.***out***.println("true");

}

}

}

**Logger.java**

//Singleton class

public class Logger {

private static Logger *object*;

private Logger();

public static Logger getObject() {

if(*object* == null) {

*object* = new Logger();

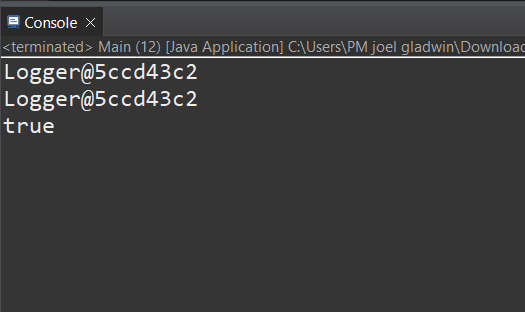
}

return *object*;

}

}

**Output:**

****

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

**Document.java**

public interface Document {

default void open() {

System.***out***.println("Default Document");

}

}

**DocumentFactory.java**

public interface DocumentFactory {

Document createDocument();

}

**WordDocument.java**

public class WordDocument implements Document{

*@Override*

public void open() {

System.***out***.println("Word Document!");

}

}

**PdfDocument.java**

public class PdfDocument implements Document{

*@Override*

public void open() {

System.***out***.println("Pdf Document!");

}

}

**ExcelDocument.java**

public class ExcelDocument implements Document{

*@Override*

public void open() {

System.***out***.println("Excel Document!");

}

}

**WordDocumentFactory.java**

public class WordDocumentFactory implements DocumentFactory{

*@Override*

public Document createDocument() {

return new WordDocument();

}

}

**PdfDocumentFactory.java**

public class PdfDocumentFactory implements DocumentFactory{

*@Override*

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelDocumentFactory.java**

public class ExcelDocumentFactory implements DocumentFactory{

*@Override*

public Document createDocument() {

return new ExcelDocument();

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

DocumentFactory wdf = new WordDocumentFactory();

Document wd = wdf.createDocument();

wd.open();

DocumentFactory pdf = new PdfDocumentFactory();

Document pd = pdf.createDocument();

pd.open();

DocumentFactory edf = new WordDocumentFactory();

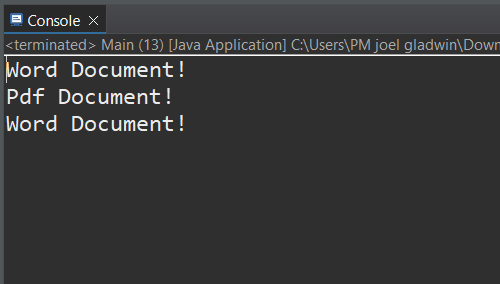
Document ed = edf.createDocument();

ed.open();

}

}

**OUTPUT:**

****

**Exercise 3: Implementing the Builder Pattern**

**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);

}

}

public void displayDetails() {

System.***out***.println("CPU: "+ CPU + " RAM: "+RAM+" STORAGE: "+storage);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

// **TODO** Auto-generated method stub

Computer pc1 = new Computer.Builder()

.setCPU("Intel i5 10th gen")

.setRAM("8GB")

.setStorage("512gb")

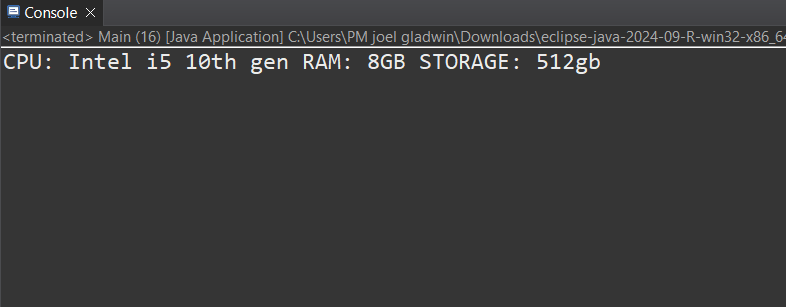
.build();

pc1.displayDetails();

}

}

**Output:**



**Exercise 4: Implementing the Adapter Pattern**

**PaymentProcessor.java**

public interface PaymentProcessor {

public void payment(int amount);

}

**Gateway1.java**

public class Gateway1 {

public void makePayment(int amount) {

System.***out***.println("Payment sent: " + amount);

}

}

**Gateway2.java**

public class Gateway2 {

public void sendPayment(int amountInINR) {

System.***out***.println("Payment sent: " + amountInINR);

}

}

**Gateway1Adapter.java**

public class Gateway1Adapter implements PaymentProcessor{

private Gateway1 googlepay;

public Gateway1Adapter(Gateway1 googlepay) {

this.googlepay = googlepay;

}

*@Override*

public void payment(int amount) {

googlepay.makePayment(amount);

}

}

**Gateway2Adapter.java**

public class Gateway2Adapter implements PaymentProcessor{

private Gateway2 paytm;

public Gateway2Adapter(Gateway2 paytm) {

this.paytm = paytm;

}

*@Override*

public void payment(int amount) {

// **TODO** Auto-generated method stub

paytm.sendPayment(amount);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Gateway1 googlepay = new Gateway1();

PaymentProcessor gpay = new Gateway1Adapter(googlepay);

gpay.payment(2000);

Gateway2 paytm = new Gateway2();

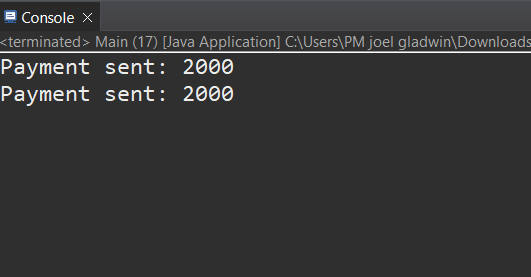
PaymentProcessor paytmp = new Gateway2Adapter(paytm);

paytmp.payment(3000);

}

}

**OUTPUT:**

****

**Exercise 5: Implementing the Decorator Pattern**

**Notifier.java**

public interface Notifier {

void send(String message);

}

**NotifierDecorator.java**

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

**EmailNotifier.java**

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

**SMSNotfierDecorator.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 baseNotifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(baseNotifier);

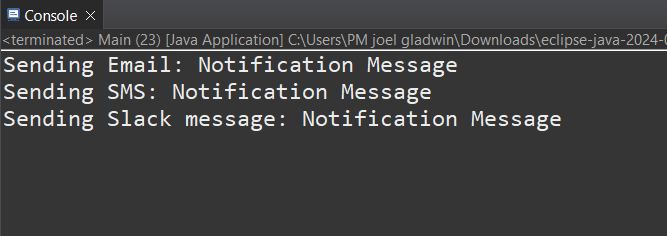
Notifier slackAndSmsNotifier = new SlackNotifierDecorator(smsNotifier);

slackAndSmsNotifier.send("Notification Message");

}

}

**Output:**

****

**Exercise 6: Implementing the Proxy Pattern**

**Image.java**

public interface Image {

void display();

}

**ProxyImage.java**

public 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();

}

}

**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 image from remote server: " + filename);

}

public void display() {

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

}

}

**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();

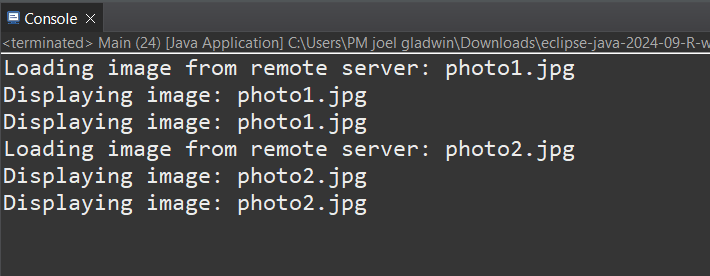
image2.display();

image2.display();

}

}

**OUTPUT:**

****

**Exercise 7: Implementing the Observer Pattern**

**Observer.java**

public interface Observer {

void update(double price);

}

**Stock.java**

public interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

void setPrice(double price);

}

**StockMarket.java**

import java.util.\*;

public class StockMarket implements Stock {

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

private double stockPrice;

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);

}

}

public void setPrice(double price) {

stockPrice = price;

notifyObservers();

}

}

**WebApp.java**

public class WebApp implements Observer {

private String appName;

public WebApp(String name) {

this.appName = name;

}

public void update(double price) {

System.***out***.println(appName + " received updated stock price: " + price);

}

}

**MobileApp.java**

public class MobileApp implements Observer {

private String appName;

public MobileApp(String name) {

this.appName = name;

}

public void update(double price) {

System.***out***.println(appName + " received updated stock price: " + price);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobile = new MobileApp("MobileApp");

Observer web = new WebApp("WebApp");

stockMarket.register(mobile);

stockMarket.register(web);

stockMarket.setPrice(101.25);

stockMarket.setPrice(105.60);

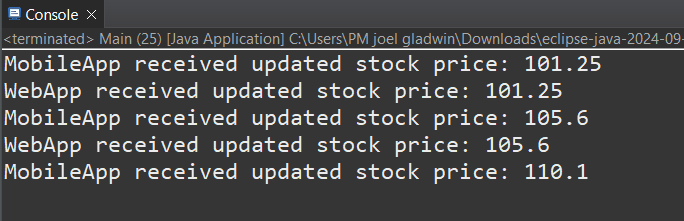
stockMarket.deregister(web);

stockMarket.setPrice(110.10);

}

}

**OUTPUT:**

****

**Exercise 8: Implementing the Strategy Pattern**

**PaymentStrategy.java**

public interface PaymentStrategy {

void pay(double amount);

}

**PaymentContext.java**

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void processPayment(double amount) {

strategy.pay(amount);

}

}

**PayPalPayment.java**

public 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: " + email);

}

}

**CreditCardPayment.java**

public 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);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));

context.processPayment(250.0);

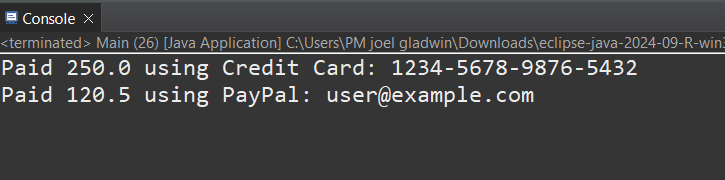
context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.processPayment(120.5);

}

}

**OUTPUT:**



**Exercise 9: Implementing the Command Pattern**

**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;

}

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() {

command.execute();

}

}

**Main.java**

public class Main {

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();

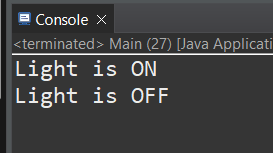
remote.setCommand(lightOff);

remote.pressButton();

}

}

**OUTPUT:**

****

**Exercise 10: Implementing the MVC Pattern**

**Student.java**

public class Student {

public String name;

public String id;

public String grade;

}

**StudentController.java**

public class StudentController {

public Student model;

public StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) {

model.name = name;

}

public void setStudentId(String id) {

model.id = id;

}

public void setStudentGrade(String grade) {

model.grade = grade;

}

public void updateView() {

view.displayStudentDetails(model.name, model.id, model.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);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Student student = new Student();

student.name = "Joel Gladwin";

student.id = "107";

student.grade = "O";

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Joseph");

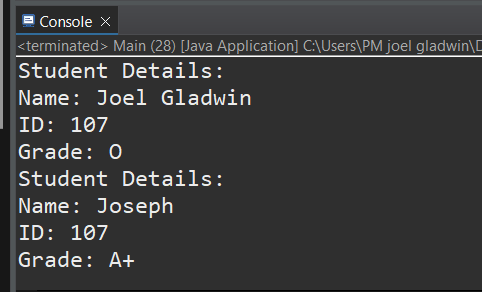
controller.setStudentGrade("A+");

controller.updateView();

}

}

**OUTPUT:**

****

**Exercise 11: Implementing Dependency Injection**

**CustomerRepository.java**

public interface CustomerRepository {

String findCustomerById(String id);

}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer[ID: " + id + ", Name: Joel Gladwin]";

}

}

**CustomerService.java**

public class CustomerService {

public CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomer(String id) {

String customer = repository.findCustomerById(id);

System.***out***.println(customer);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

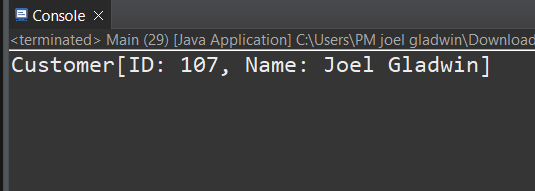
CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.getCustomer("107");

}

}

**OUTPUT:  
  
**