**DESIGN PRINCIPLE AND PATTERN**

**-Harini Baskar(6396726)**

**Exercise 1: Implementing the Singleton Pattern**

**Code:**

package singleton;

public class Logger {

private static Logger *instance*;

private Logger() {

System.***out***.println("Logger Created");

}

public static Logger getInstance() {

if (*instance* == null) {

*instance* = new Logger();

}

return *instance*;

}

public void log(String message) {

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

}

}

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

logger1.log("First message");

Logger logger2 = Logger.*getInstance*();

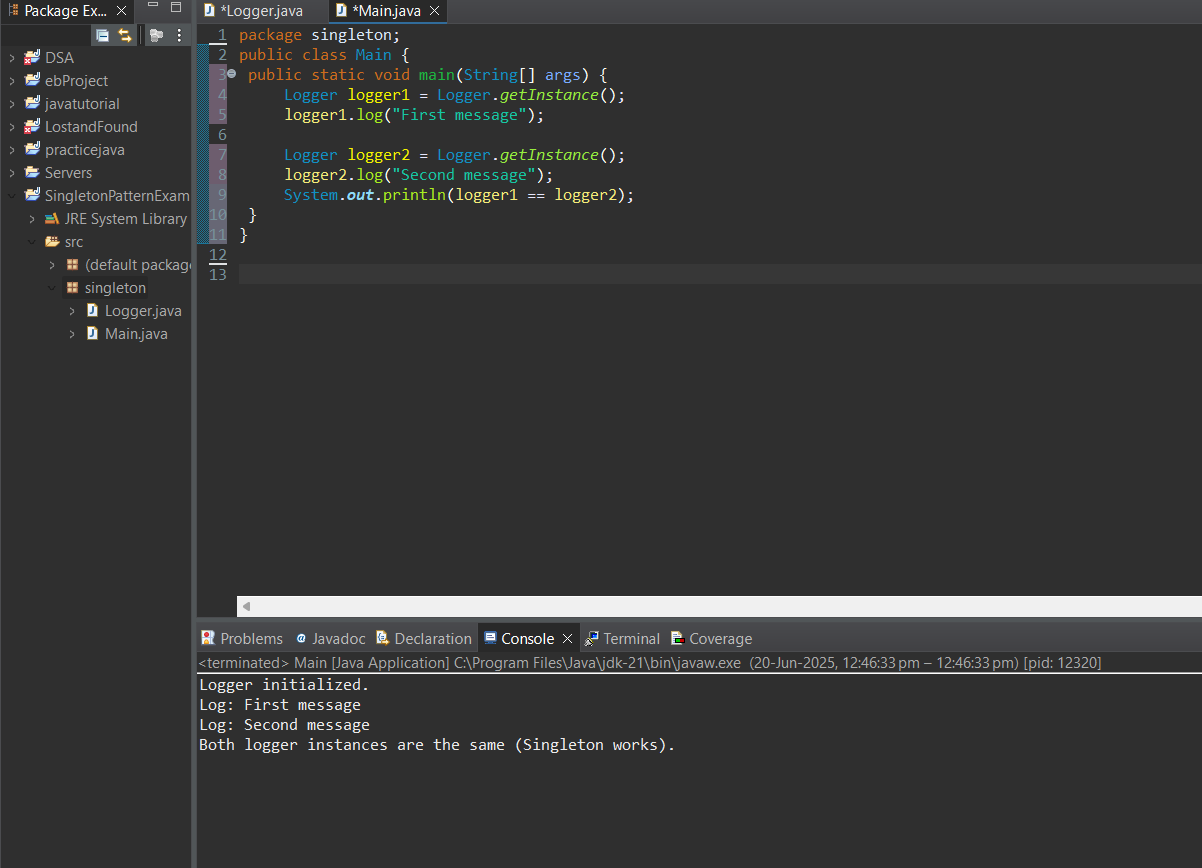
logger2.log("Second message");

System.***out***.println(logger1 == logger2);

}

}

**Output:**

****

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

**Code:**

// 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 {

public void open() {

System.out.println("Opening PDF Document");

}

}

// ExcelDocument.java

public class ExcelDocument implements Document {

public void open() {

System.out.println("Opening Excel Document");

}

}

// DocumentFactory.java

public abstract class DocumentFactory {

public abstract Document createDocument();

}

// WordFactory.java

public class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

// PdfFactory.java

public class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

// ExcelFactory.java

public class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

// Main.java

public class Main {

public static void main(String[] args) {

DocumentFactory factory;

factory = new WordFactory();

Document doc1 = factory.createDocument();

doc1.open();

factory = new PdfFactory();

Document doc2 = factory.createDocument();

doc2.open();

factory = new ExcelFactory();

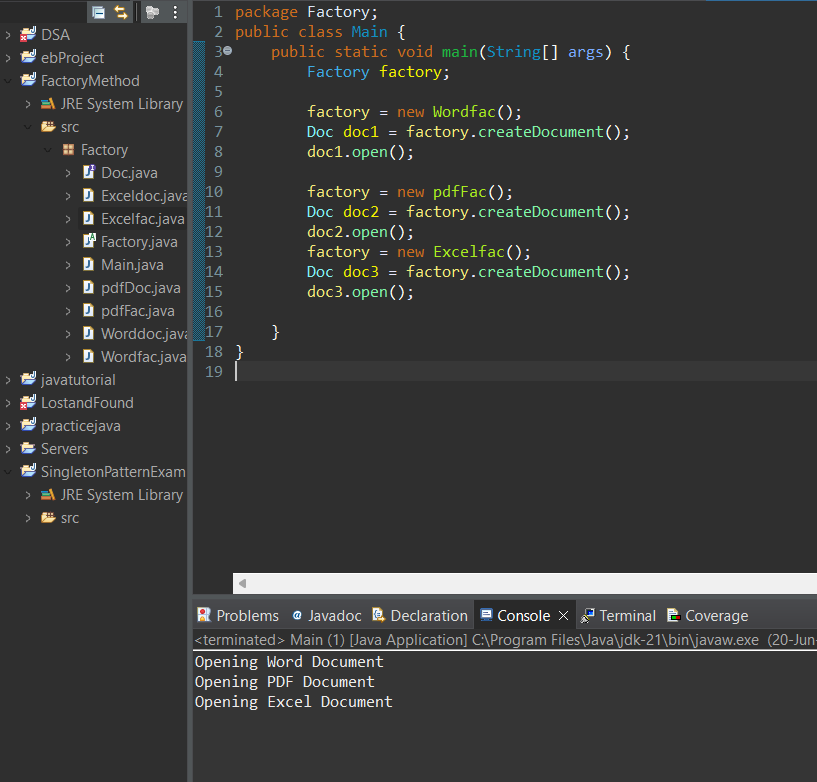
Document doc3 = factory.createDocument();

doc3.open();

}

}

**Output:**

****

**Exercise 3: Implementing the Builder Pattern**

**Code:**

// Computer.java

public class Computer {

private String CPU;

private String RAM;

private String storage;

// Private constructor

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);

}

// Static nested Builder class

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);

}

}

}

// Main.java

public class Main {

public static void main(String[] args) {

// Creating a computer with all parts

Computer myComputer = new Computer.Builder()

.setCPU("Intel i7")

.setRAM("16GB")

.setStorage("512GB SSD")

.build();

myComputer.showSpecs();

// Creating another computer with different specs

Computer basicComputer = new Computer.Builder()

.setCPU("Intel i3")

.setRAM("8GB")

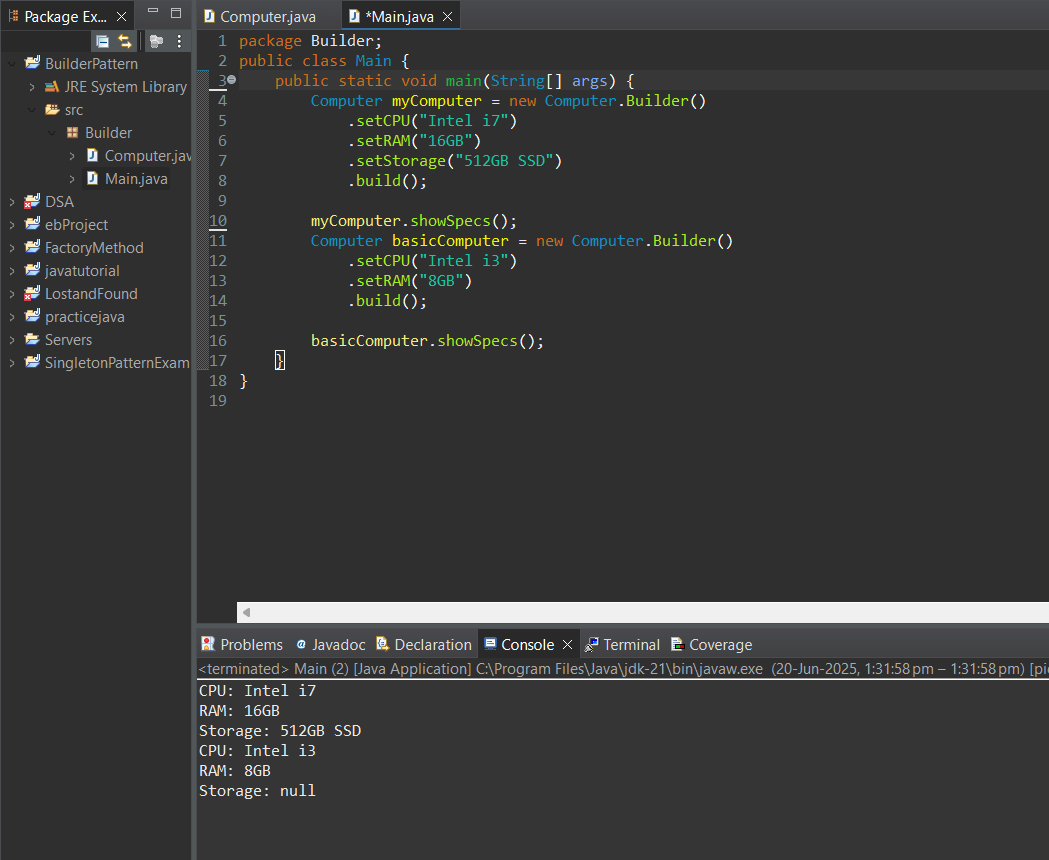
.build();

basicComputer.showSpecs();

}

}

**Output:**

****

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

public interface PaymentProcessor {

void processPayment(double amount);

}

class LegacyBankGateway {

public void makePayment(String accountNumber, double amount) {

System.out.println("Payment of ₹" + amount + " made from account " + accountNumber + " using Legacy Bank.");

}

}

class LegacyBankAdapter implements PaymentProcessor {

private LegacyBankGateway legacy;

public LegacyBankAdapter() {

this.legacy = new LegacyBankGateway();

}

public void processPayment(double amount) {

legacy.makePayment("ACC123456", amount);

}

}

public class AdapterTest {

public static void main(String[] args) {

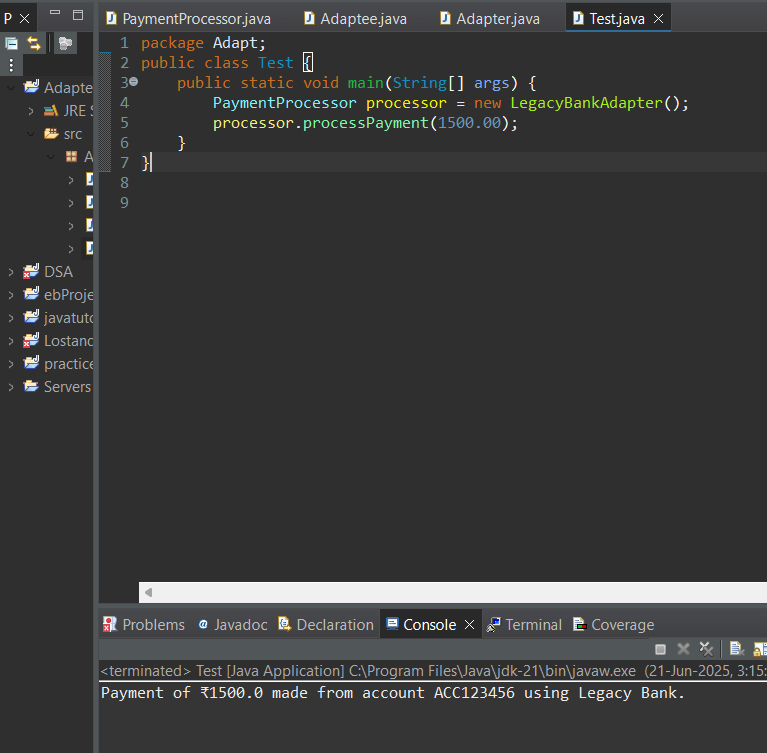
PaymentProcessor processor = new LegacyBankAdapter();

processor.processPayment(1500.00);

}

}

**Output:**

****

**Exercise 5: Implementing the Decorator Pattern**

**Code:**

public 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 wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send(String message) {

wrappedNotifier.send(message);

}

}

class SMSNotifier extends NotifierDecorator {

public SMSNotifier(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

class SlackNotifier extends NotifierDecorator {

public SlackNotifier(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

public class DecoratorTest {

public static void main(String[] args) {

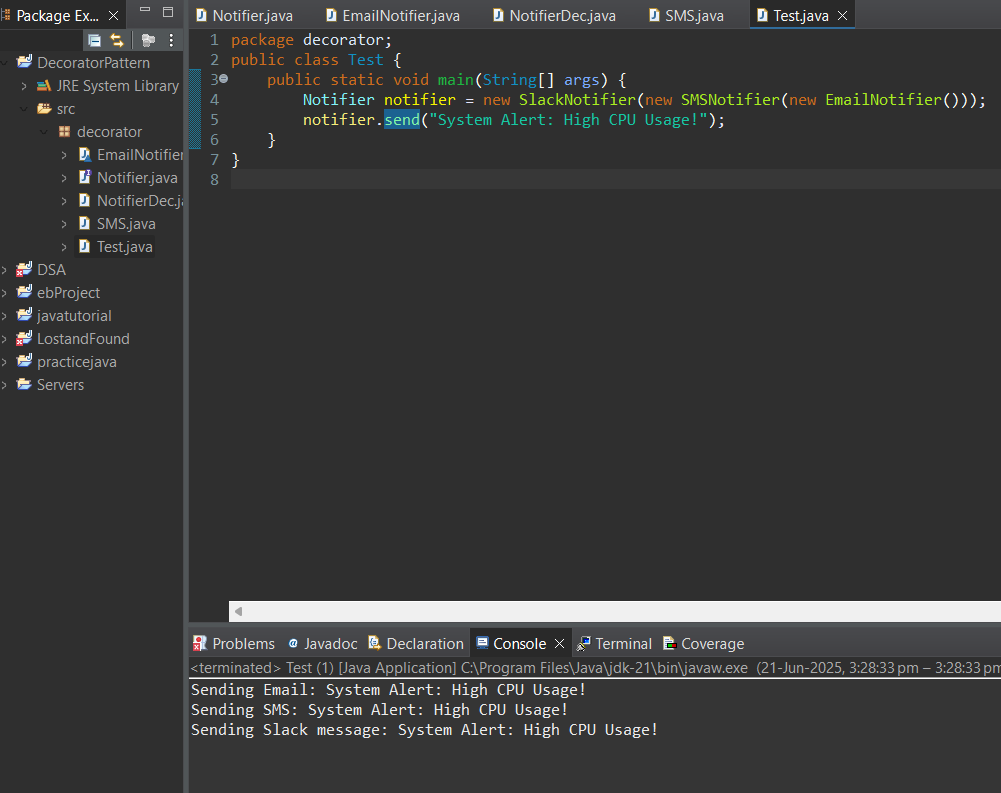
Notifier notifier = new SlackNotifier(new SMSNotifier(new EmailNotifier()));

notifier.send("System Alert: High CPU Usage!");

}

}

**Output:**

****

**Exercise 8: Implementing the Strategy Pattern**

**Code:**

public interface PaymentStrategy {

void pay(double amount);

}

public class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

public class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

public class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void payAmount(double amt) {

strategy.pay(amt);

}

}

public class StrategyTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setStrategy(new CreditCardPayment());

context.payAmount(1200);

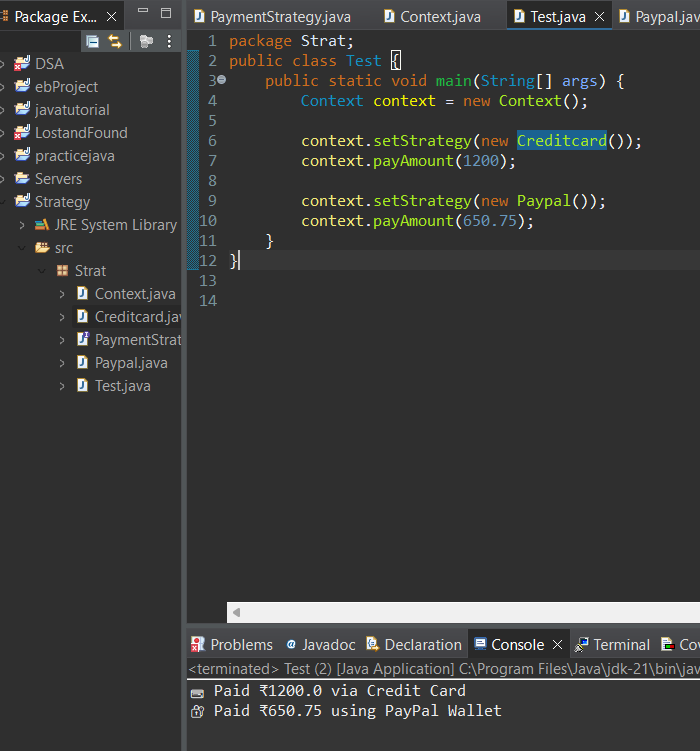
context.setStrategy(new PayPalPayment());

context.payAmount(650.75);

}

}

**Output:**

****

**Exercise 11: Implementing Dependency Injection**

**Code:**

public interface CustomerRepository {

String findCustomerById(int id);

}

public class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer#" + id + " found in database.";

}

}

public class CustomerService {

private CustomerRepository repo;

public CustomerService(CustomerRepository repo) {

this.repo = repo;

}

public void showCustomer(int id) {

String result = repo.findCustomerById(id);

System.out.println("🔍 Service says: " + result);

}

}

public class DIExample {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

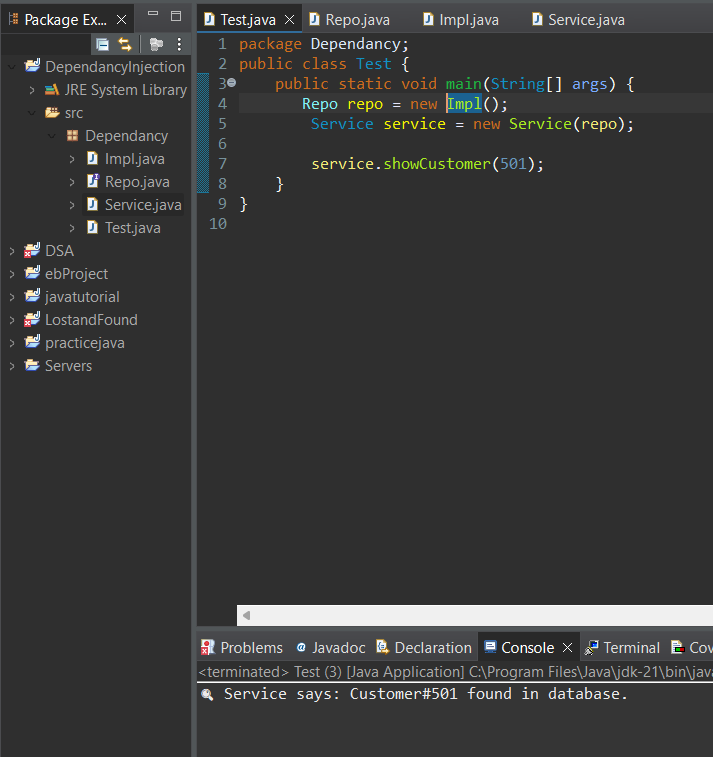
CustomerService service = new CustomerService(repo);

service.showCustomer(501);

}

}

**Output:**

****