DESIGN PATTERN EXERCISE SOLUTION

Exercise 1: Implementing the Singleton Pattern

Scenario:

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

Steps:

1. Create a New Java Project:

Create a new Java project named SingletonPatternExample.

2. Define a Singleton Class:

Create a class named Logger that has a private static instance of itself.

Ensure the constructor of Logger is private.

Provide a public static method to get the instance of the Logger class.

3. Implement the Singleton Pattern:

Write code to ensure that the Logger class follows the Singleton design pattern.

4. Test the Singleton Implementation:

Create a test class to verify that only one instance of Logger is created and used across the application.

CODE:-

Logger.java:-

package q1;

public class Logger {

    private static Logger instance;

    private Logger() {

    }

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

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

    }

}

LoggerTest.java:-

package q1;

public class LoggerTest {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        logger1.log("This is the first log message.");

        logger2.log("This is the second log message.");

        if (logger1 == logger2) {

            System.out.println("Both logger instances are the same.");

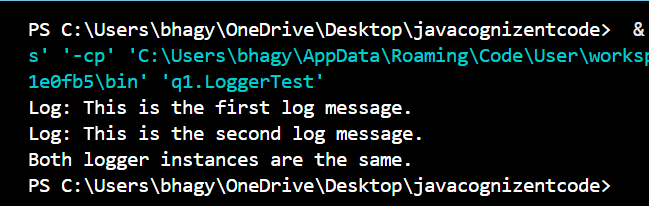
        } else {

            System.out.println("Different logger instances exist.");

        }

    }

}



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

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

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();

}

**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();

}

}

**FactoryTest.java**

public class FactoryTest {

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();

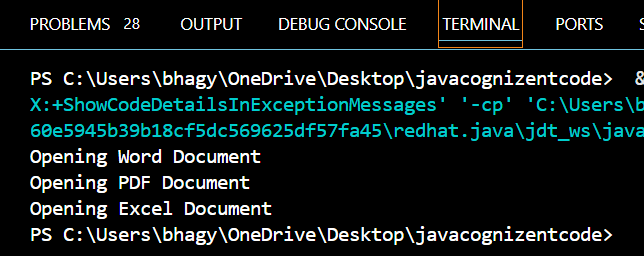
DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}



**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   * Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   * Create a static nested Builder class inside Computer with methods to set each attribute.
   * Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   * Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   * Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

CODE:-

Computer.java

public class Computer {

private String cpu;

private String ram;

private String storage;

private String gpu;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.gpu = builder.gpu;

}

public void showConfig() {

System.out.println("CPU: " + cpu);

System.out.println("RAM: " + ram);

System.out.println("Storage: " + storage);

System.out.println("GPU: " + gpu);

}

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String gpu;

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 setGpu(String gpu) {

this.gpu = gpu;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

BuilderTest.java

public class BuilderTest {

public static void main(String[] args) {

Computer gamingPc = new Computer.Builder()

.setCpu("Intel i9")

.setRam("32GB")

.setStorage("1TB SSD")

.setGpu("NVIDIA RTX 4080")

.build();

Computer officePc = new Computer.Builder()

.setCpu("Intel i5")

.setRam("16GB")

.setStorage("512GB SSD")

.build();

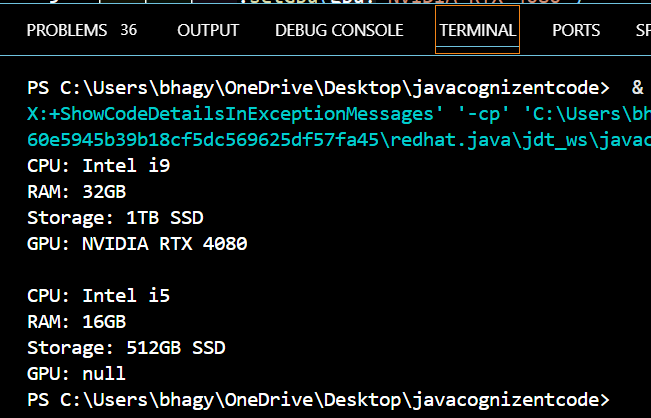
gamingPc.showConfig();

System.out.println();

officePc.showConfig();

}

}



**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **AdapterPatternExample**.
2. **Define Target Interface:**
   * Create an interface **PaymentProcessor** with methods like **processPayment()**.
3. **Implement Adaptee Classes:**
   * Create classes for different payment gateways with their own methods.
4. **Implement the Adapter Class:**
   * Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.
5. **Test the Adapter Implementation:**
   * Create a test class to demonstrate the use of different payment gateways through the adapter.

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 through Stripe: ₹" + amount);

}

}

PayPalGateway.java

public class PayPalGateway {

public void sendPayment(double amount) {

System.out.println("Processing payment through PayPal: ₹" + amount);

}

}

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);

}

}

AdapterTest.java

public class AdapterTest {

public static void main(String[] args) {

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

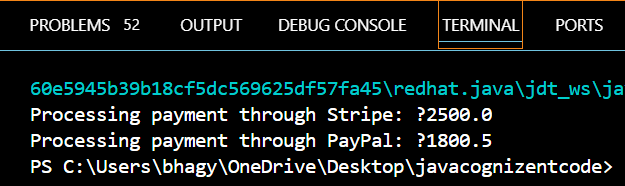
stripePayment.processPayment(2500.00);

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

paypalPayment.processPayment(1800.50);

}

}



**Exercise 5: Implementing the Decorator Pattern**

**Scenario:**

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DecoratorPatternExample**.
2. **Define Component Interface:**
   * Create an interface **Notifier** with a method **send()**.
3. **Implement Concrete Component:**
   * Create a class **EmailNotifier** that implements Notifier.
4. **Implement Decorator Classes:**
   * Create abstract decorator class **NotifierDecorator** that implements **Notifier** and holds a reference to a **Notifier** object.
   * Create concrete decorator classes like **SMSNotifierDecorator**, **SlackNotifierDecorator** that extend **NotifierDecorator**.
5. **Test the Decorator Implementation:**
   * Create a test class to demonstrate sending notifications via multiple channels using decorators.

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 notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.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);

}

}

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);

}

}

DecoratorTest.java

public class DecoratorTest {

public static void main(String[] args) {

Notifier notifier = new SlackNotifierDecorator(

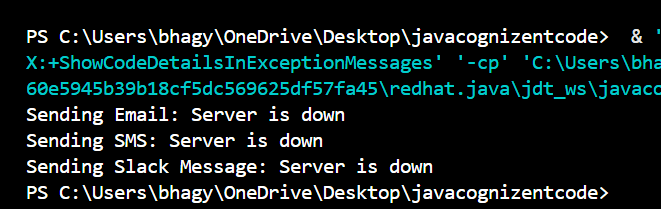
new SMSNotifierDecorator(

new EmailNotifier()));

notifier.send("Server is down");

}

}



**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ProxyPatternExample**.
2. **Define Subject Interface:**
   * Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
   * Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
   * Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   * Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**
   * Create a test class to demonstrate the use of **ProxyImage** to load and display images.

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;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

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

}

public void display() {

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

}

}

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();

}

}

ProxyTest.java

public class ProxyTest {

public static void main(String[] args) {

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

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

image1.display();

System.out.println();

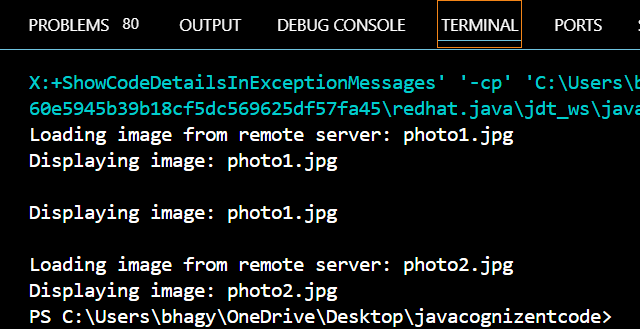
image1.display();

System.out.println();

image2.display();

}

}



**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   * Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   * Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   * Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   * Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   * Create a test class to demonstrate the registration and notification of observers.

CODE:-

Stock.java

public interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

Observer.java

public interface Observer {

void update(String stockName, double price);

}

StockMarket.java

import java.util.\*;

public class StockMarket implements Stock {

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

private String stockName;

private double price;

public void setStock(String stockName, double price) {

this.stockName = stockName;

this.price = 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(stockName, price);

}

}

}

MobileApp.java

public class MobileApp implements Observer {

public void update(String stockName, double price) {

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

}

}

WebApp.java

public class WebApp implements Observer {

public void update(String stockName, double price) {

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

}

}

ObserverTest.java

public class ObserverTest {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

stockMarket.register(mobile);

stockMarket.register(web);

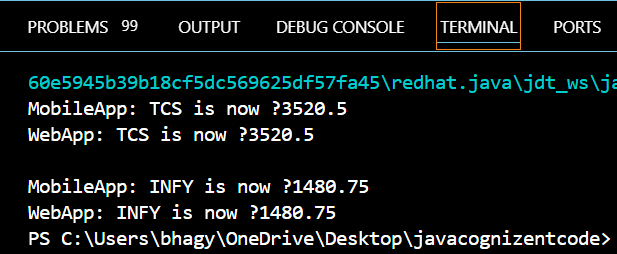
stockMarket.setStock("TCS", 3520.50);

System.out.println();

stockMarket.setStock("INFY", 1480.75);

}

}



**Exercise 8: Implementing the Strategy Pattern**

**Scenario:**

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **StrategyPatternExample**.
2. **Define Strategy Interface:**
   * Create an interface PaymentStrategy with a method **pay()**.
3. **Implement Concrete Strategies:**
   * Create classes **CreditCardPayment**, **PayPalPayment** that implement **PaymentStrategy**.
4. **Implement Context Class:**
   * Create a class **PaymentContext** that holds a reference to **PaymentStrategy** and a method to execute the strategy.
5. **Test the Strategy Implementation:**
   * Create a test class to demonstrate selecting and using different payment strategies.

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) {

strategy.pay(amount);

}

}

StrategyTest.java

public class StrategyTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment());

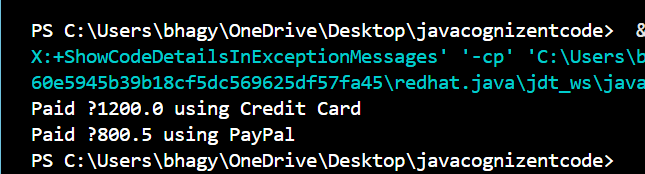
context.executePayment(1200.00);

context.setPaymentStrategy(new PayPalPayment());

context.executePayment(800.50);

}

}



**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **CommandPatternExample**.
2. **Define Command Interface:**
   * Create an interface Command with a method **execute()**.
3. **Implement Concrete Commands:**
   * Create classes **LightOnCommand**, **LightOffCommand** that implement Command.
4. **Implement Invoker Class:**
   * Create a class **RemoteControl** that holds a reference to a Command and a method to execute the command.
5. **Implement Receiver Class:**
   * Create a class **Light** with methods to turn on and off.
6. **Test the Command Implementation:**
   * Create a test class to demonstrate issuing commands using the **RemoteControl**.

CODE:-

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();

}

}

CommandTest.java

public class CommandTest {

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);

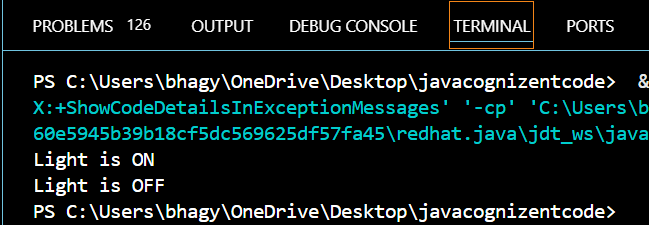
remote.pressButton();

remote.setCommand(lightOff);

remote.pressButton();

}

}



**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **MVCPatternExample**.
2. **Define Model Class:**
   * Create a class **Student** with attributes like **name, id, and grade**.
3. **Define View Class:**
   * Create a class **StudentView** with a method **displayStudentDetails()**.
4. **Define Controller Class:**
   * Create a class **StudentController** that handles the communication between the model and the view.
5. **Test the MVC Implementation:**
   * Create a main class to demonstrate creating a **Student**, updating its details using **StudentController**, and displaying them using **StudentView**.

CODE:-

Student.java

public class Student {

private String name;

private String id;

private String 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;

}

}

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);

}

}

StudentController.java

public 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());

}

}

MVCTest.java

public class MVCTest {

public static void main(String[] args) {

Student student = new Student();

student.setName("Bhagyashree");

student.setId("S123");

student.setGrade("A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

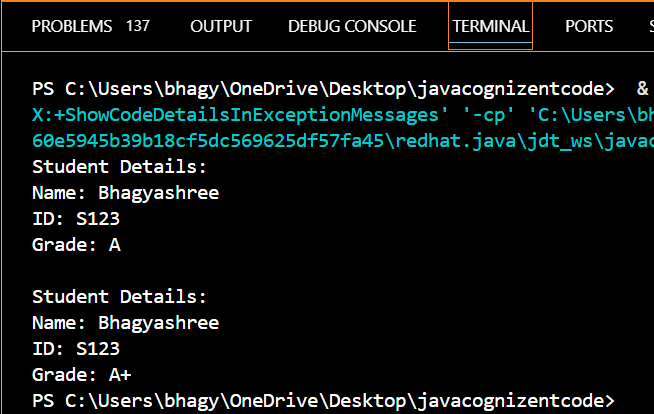
controller.setStudentGrade("A+");

System.out.println();

controller.updateView();

}

}



**Exercise 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DependencyInjectionExample**.
2. **Define Repository Interface:**
   * Create an interface **CustomerRepository** with methods like **findCustomerById()**.
3. **Implement Concrete Repository:**
   * Create a class **CustomerRepositoryImpl** that implements **CustomerRepository**.
4. **Define Service Class:**
   * Create a class **CustomerService** that depends on **CustomerRepository**.
5. **Implement Dependency Injection:**
   * Use constructor injection to inject **CustomerRepository** into **CustomerService**.
6. **Test the Dependency Injection Implementation:**
   * Create a main class to demonstrate creating a **CustomerService** with **CustomerRepositoryImpl** and using it to find a customer.

CODE:-

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=Bhagyashree Panda]";

}

}

CustomerService.java

public class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomer(String id) {

String customer = repository.findCustomerById(id);

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

}

}

DIAppTest.java

public class DIAppTest {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.displayCustomer("C101");

}

}

