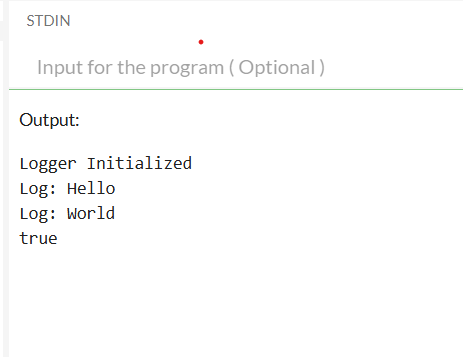
Java Design Patterns - CTS Preparation

## 1. Singleton Pattern - Logger

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 msg) {  
 System.out.println("Log: " + msg);  
 }  
  
 public static void main(String[] args) {  
 Logger logger1 = Logger.getInstance();  
 Logger logger2 = Logger.getInstance();  
  
 logger1.log("Hello");  
 logger2.log("World");  
  
 System.out.println(logger1 == logger2); // true  
 }  
}

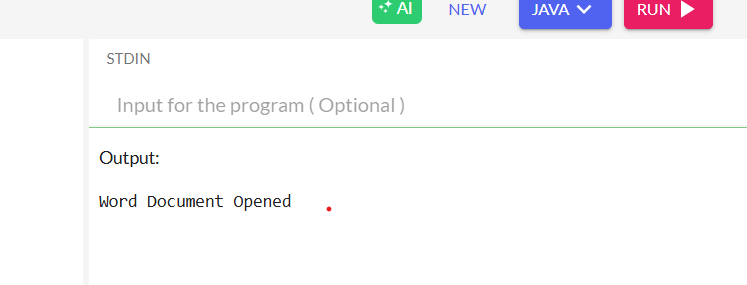
**OUTPUT**



## 2. Factory Method Pattern - Document

interface Document {  
 void open();  
}  
  
class WordDocument implements Document {  
 public void open() {  
 System.out.println("Word Document Opened");  
 }  
}  
  
class PdfDocument implements Document {  
 public void open() {  
 System.out.println("PDF Document Opened");  
 }  
}  
  
abstract class DocumentFactory {  
 abstract Document createDocument();  
}  
  
class WordFactory extends DocumentFactory {  
 Document createDocument() {  
 return new WordDocument();  
 }  
}  
  
class PdfFactory extends DocumentFactory {  
 Document createDocument() {  
 return new PdfDocument();  
 }  
}  
  
class Main {  
 public static void main(String[] args) {  
 DocumentFactory factory = new WordFactory();  
 Document doc = factory.createDocument();  
 doc.open();  
 }  
}

**OUTPUT**



**3.Builder Pattern**

class Computer {

private String cpu, ram, storage;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

}

public static class Builder {

private String cpu, ram, 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 showConfig() {

System.out.println(cpu + " | " + ram + " | " + storage);

}

}

class Main {

public static void main(String[] args) {

Computer pc = new Computer.Builder()

.setCpu("Intel i7")

.setRam("16GB")

.setStorage("1TB SSD")

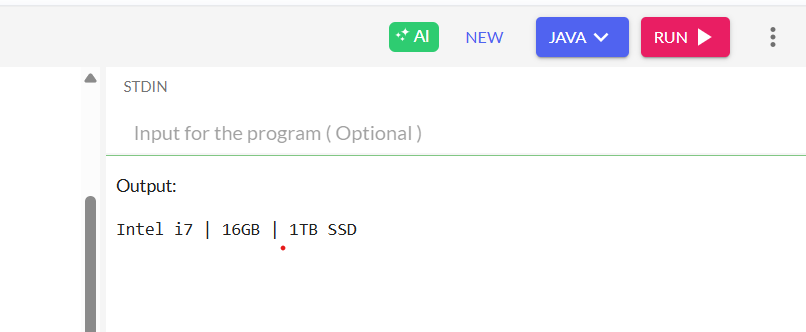
.build();

pc.showConfig();

}

}

**OUTPUT**



**4. Adapter Pattern – PaymentProcessor**

interface PaymentProcessor {

void processPayment();

}

class StripeGateway {

void makeStripePayment() {

System.out.println("Paid using Stripe");

}

}

class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment() {

stripe.makeStripePayment();

}

}

class Main {

public static void main(String[] args) {

StripeGateway stripe = new StripeGateway();

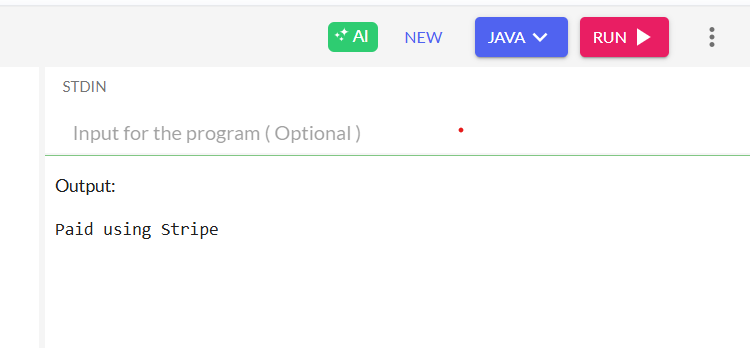
PaymentProcessor processor = new StripeAdapter(stripe);

processor.processPayment();

}

}

**OUTPUT**



## 5. Decorator Pattern – Notifier

interface Notifier {

void send();

}

class EmailNotifier implements Notifier {

public void send() {

System.out.println("Email sent");

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send() {

notifier.send();

}

}

class SMSNotifier extends NotifierDecorator {

public SMSNotifier(Notifier notifier) {

super(notifier);

}

public void send() {

super.send();

System.out.println("SMS sent");

}

}

class Main {

public static void main(String[] args) {

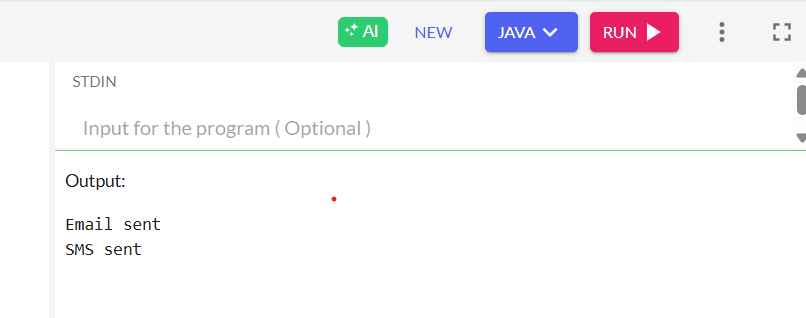
Notifier notifier = new SMSNotifier(new EmailNotifier());

notifier.send();

}

}

**OUTPUT**



1. **Proxy Pattern – Image**

interface Image {

void display();

}

class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromDisk();

}

private void loadFromDisk() {

System.out.println("Loading " + fileName);

}

public void display() {

System.out.println("Displaying " + fileName);

}

}

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();

}

}

class Main {

public static void main(String[] args) {

Image image = new ProxyImage("cat.png");

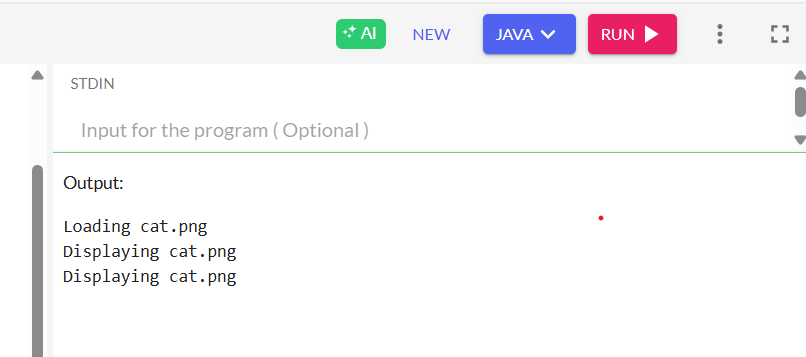
image.display(); // loads and displays

image.display(); // just displays

}

}

**OUTPUT**



1. **Observer Pattern – Stock**

import java.util.\*;

interface Observer {

void update(float price);

}

interface Stock {

void register(Observer o);

void remove(Observer o);

void notifyAllObservers();

}

class StockMarket implements Stock {

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

private float price;

public void setPrice(float price) {

this.price = price;

notifyAllObservers();

}

public void register(Observer o) {

observers.add(o);

}

public void remove(Observer o) {

observers.remove(o);

}

public void notifyAllObservers() {

for (Observer o : observers) {

o.update(price);

}

}

}

class MobileApp implements Observer {

public void update(float price) {

System.out.println("Mobile App Updated: Price = " + price);

}

}

class Main {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer app1 = new MobileApp();

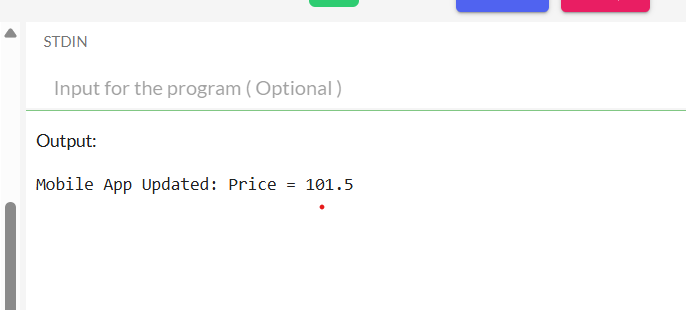
market.register(app1);

market.setPrice(101.5f);

}

}

**OUTPUT**



1. **Strategy Pattern – Payment**

interface PaymentStrategy {

void pay(int amount);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(int amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(int amount) {

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

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(int amount) {

strategy.pay(amount);

}

}

class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setStrategy(new CreditCardPayment());

context.executePayment(500);

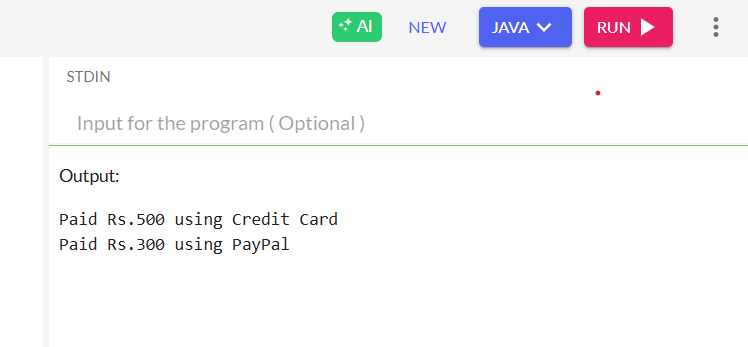
context.setStrategy(new PayPalPayment());

context.executePayment(300);

}

}

**OUTPUT**



1. **Command Pattern – RemoteControl**

interface Command {

void execute();

}

class Light {

void on() {

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

}

void off() {

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

}

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.on();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.off();

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

class Main {

public static void main(String[] args) {

Light light = new Light();

Command on = new LightOnCommand(light);

Command off = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(on);

remote.pressButton();

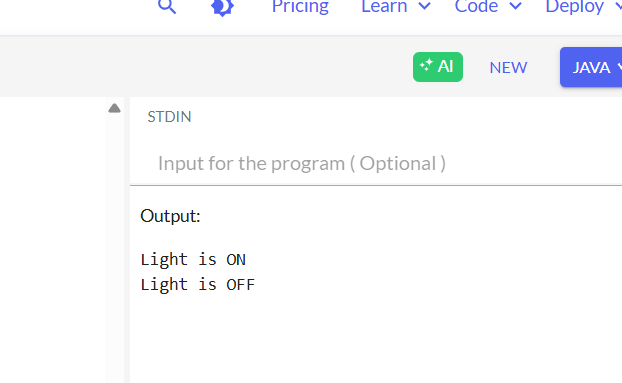
remote.setCommand(off);

remote.pressButton();

}

}

**OUTPUT**



1. **MVC Pattern – Student App**

class Student {

private String name;

private String id;

public Student(String name, String id) {

this.name = name;

this.id = id;

}

public String getName() { return name; }

public String getId() { return id; }

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

}

class StudentView {

public void displayStudent(String name, String id) {

System.out.println("Student: " + name + ", ID: " + id);

}

}

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void updateView() {

view.displayStudent(model.getName(), model.getId());

}

public void setStudentName(String name) {

model.setName(name);

}

}

class Main {

public static void main(String[] args) {

Student model = new Student("Dhivya", "CTS1025");

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

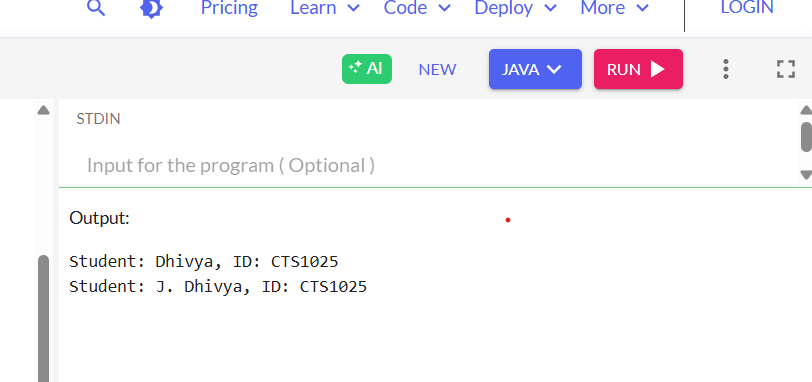
controller.setStudentName("J. Dhivya");

controller.updateView();

}

}

**OUTPUT**



1. **Dependency Injection – CustomerService**

interface CustomerRepository {

String findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer Found: ID = " + id;

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomer(String id) {

String result = repository.findCustomerById(id);

System.out.println(result);

}

}

class Main {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.displayCustomer("C123");

}

}

**OUTPUT**

