**Mandotary Handson**

**1.Implementing the Singleton Pattern**

package singleton;

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;

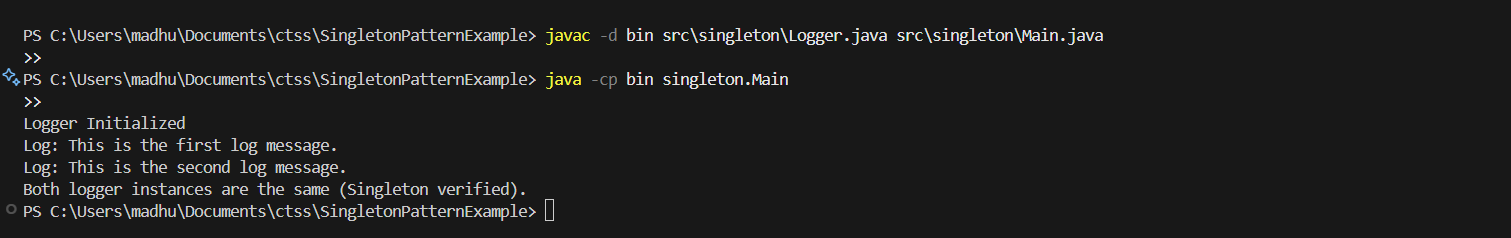
} public void log(String message) {

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

}

}

**OUTPUT:**



**2.Implementing the Factory Method Pattern**

**Document.java**

package factory;

public interface Document {

    void open();

}

**DocumentFactory.java**

package factory;

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

**ExcelDocument.java**

package factory;

public class ExcelDocument implements Document {

    public void open() {

        System.out.println("Opening an Excel document.");

    }

}

**ExcelFactory.java**

package factory;

public class ExcelFactory extends DocumentFactory {

    public Document createDocument() {

        return new ExcelDocument();

    }

}

**Main.java**

package factory;

public class Main {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

DocumentFactory pdfFactory = new PdfFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

DocumentFactory excelFactory = new ExcelFactory();

        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

**PdfDocument.java**

package factory;

public class PdfDocument implements Document {

    public void open() {

        System.out.println("Opening a PDF document.");

    }

}

**PdfFactory.java**

package factory;

public class PdfFactory extends DocumentFactory {

    public Document createDocument() {

        return new PdfDocument();

    }

}

**WordDcoument.java**

package factory;

public class WordDocument implements Document {

    public void open() {

        System.out.println("Opening a Word document.");

    }

}

**WordFactroy.java**

package factory;

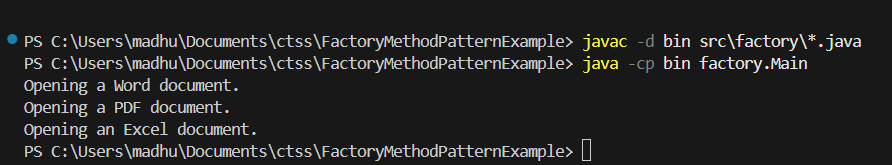
public class WordFactory extends DocumentFactory {

    public Document createDocument() {

        return new WordDocument();

    }}

**OUTPUT:**



**Aditional :**

**3.Implementing the Builder Pattern**

**Computer.java**

public class Computer {

    private String CPU;

    private String RAM;

    private String storage;

    private String graphicsCard;

private Computer(Builder builder) {

        this.CPU = builder.CPU;

        this.RAM = builder.RAM;

        this.storage = builder.storage;

        this.graphicsCard = builder.graphicsCard;

    }

public String getCPU() { return CPU; }

    public String getRAM() { return RAM; }

    public String getStorage() { return storage; }

    public String getGraphicsCard() { return graphicsCard; }

    public String toString() {

        return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage + ", Graphics Card=" + graphicsCard + "]";

    }

    public static class Builder {

        private String CPU;

        private String RAM;

        private String storage;

        private String graphicsCard;

        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 setGraphicsCard(String graphicsCard) {

            this.graphicsCard = graphicsCard;

            return this;}

public Computer build() {

            return new Computer(this);

        }}}

**TestBuilderPattern.java**

public class TestBuilderPattern {

    public static void main(String[] args) {

        Computer basicComputer = new Computer.Builder()

                .setCPU("Intel i3")

                .setRAM("8GB")

                .setStorage("256GB SSD")

                .build();

        Computer gamingComputer = new Computer.Builder()

                .setCPU("Intel i9")

                .setRAM("32GB")

                .setStorage("1TB SSD")

                .setGraphicsCard("NVIDIA RTX 4090")

                .build();

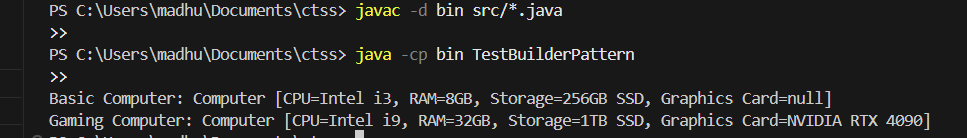
System.out.println("Basic Computer: " + basicComputer);

        System.out.println("Gaming Computer: " + gamingComputer);

    }

}

**OUTPUT:**



**Exercise 4: Implementing the Adapter Pattern**

**Main.java**

public class Main {

    public static void main(String[] args) {

        PayPalGateway payPal = new PayPalGateway();

        PaymentProcessor paypalProcessor = new PayPalAdapter(payPal);

        paypalProcessor.processPayment(250.00);

        StripeGateway stripe = new StripeGateway();

        PaymentProcessor stripeProcessor = new StripeAdapter(stripe);

        stripeProcessor.processPayment(120.50);

    }

}

**PaymentProcessor.java**

public interface PaymentProcessor {

    void processPayment(double 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);

    }

}

**PayPalGateway.java**

public class PayPalGateway {

    public void sendPayment(double amountInDollars) {

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

    }

}

**StripeAdapter.java**

public class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripeGateway;

    public StripeAdapter(StripeGateway stripeGateway) {

        this.stripeGateway = stripeGateway;

    }

    public void processPayment(double amount) {

        int amountInCents = (int)(amount \* 100);

        stripeGateway.makePayment(amountInCents);

    }

}

**StripeGateway.java**

public class StripeGateway {

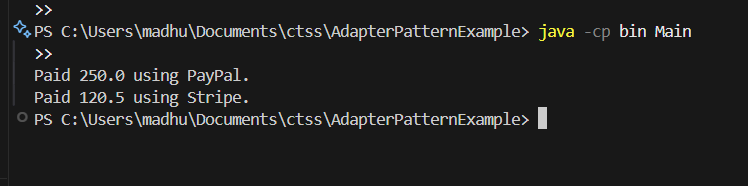
    public void makePayment(int amountInCents) {

        System.out.println("Paid " + (amountInCents / 100.0) + " using Stripe.");

    }

}

OUTPUT:



**Exercise 5: Implementing the Decorator Pattern**

**EmailNotifier.class**

public class EmailNotifier implements Notifier {

   public EmailNotifier() {

   }

   public void send(String var1) {

      System.out.println("Sending Email: " + var1);

   }

}

**EmailNotifier.java**

public class EmailNotifier implements Notifier {

    public void send(String message) {

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

    }

}

**Main.class**

public class Main {

   public Main() {

   }

   public static void main(String[] var0) {

      SlackNotifierDecorator var1 = new SlackNotifierDecorator(new SMSNotifierDecorator(new EmailNotifier()));

      var1.send("Hello!");

   }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        Notifier notifier = new SlackNotifierDecorator(new SMSNotifierDecorator(new EmailNotifier()));

        notifier.send("Hello!");

    }

}

**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;

    }

}

**SlackNotifierDEcorator.java**

public class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    public void send(String message) {

        notifier.send(message);

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

    }

}

**SMSNotifierDecorator.java**

public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

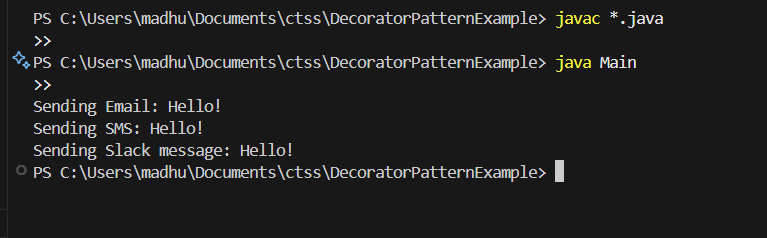
    public void send(String message) {

        notifier.send(message);

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

    }

}

**OUTPUT:**

**Exercise 6: Implementing the Proxy Pattern**

**Image.class**

public interface Image {

   void display();

}

**Image.java**

public interface Image {

    void display();

}

**Main.class**

public class Main {

   public Main() {

   }

public static void main(String[] var0) {

      ProxyImage var1 = new ProxyImage("test.jpg");

      var1.display();

      var1.display();

   }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        Image image = new ProxyImage("test.jpg");

        image.display();

        image.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();

    }

}

**Realmage.java**

public class RealImage implements Image {

    private String filename;

    public RealImage(String filename) {

        this.filename = filename;

        loadFromDisk();

    }

    private void loadFromDisk() {

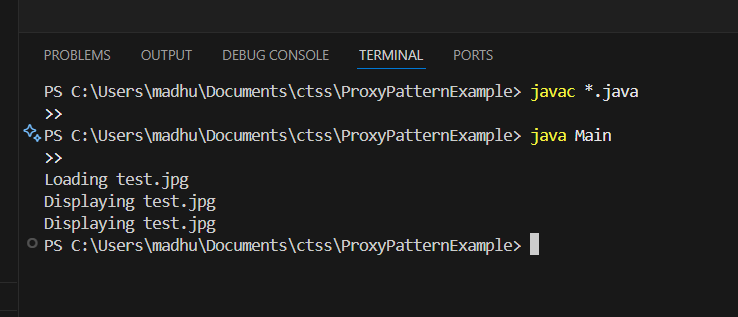
        System.out.println("Loading " + filename);

    }

    public void display() {

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

    }}

**OUTPUT:**

**Exercise 7: Implementing the Observer Pattern**

**OUTPUT:**

**Exercise 8: Implementing the Strategy Pattern**

**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("The light is ON.");

    }

    public void turnOff() {

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

    }

}

**Lightoffcommand.java**

public class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.turnOff();

    }

}

**Lightoncommand.java**

public class LightOnCommand implements Command {

    private Light light;

public LightOnCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.turnOn();

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        TestCommandPattern.main(args);

    }

}

**RemoteControl.java**

public class RemoteControl {

    private Command command;

    public void setCommand(Command command) {

        this.command = command;

    }

    public void pressButton() {

        if (command != null) {

            command.execute();

        }

    }

}

**TestCommandPatter.java**

public class TestCommandPattern {

    public static void main(String[] args) {

        Light light = new Light();

        Command lightOn = new LightOnCommand(light);

        Command lightOff = new LightOffCommand(light);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(lightOn);

        remote.pressButton();

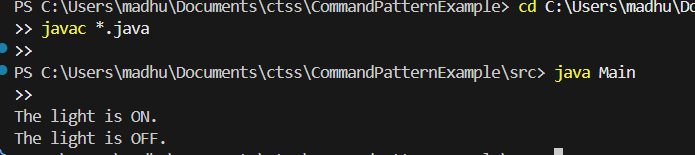
        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

**OUTPUT:**

****

**Exercise 10: Implementing the MVC Pattern**

**Main.java**

public class Main {

    public static void main(String[] args) {

        TestMVCPattern.main(args);

    }

}

**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 setGrade(String grade) { this.grade = grade; }

}

**StudentControler.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 setStudentGrade(String grade) {

        model.setGrade(grade);

    }

    public void updateView() {

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

    }

}

**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);

    }

}

**TestMVCPattern.java**

public class TestMVCPattern {

public static void main(String[] args) {

        Student student = new Student("Alice", "S001", "A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        controller.setStudentName("Alice Cooper");

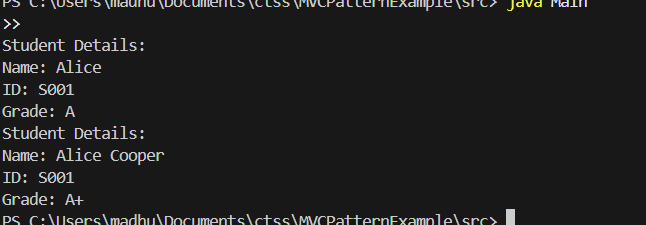
        controller.setStudentGrade("A+");

        controller.updateView();

    }

}

**OUTPUT:**

****

**Exercise 11: Implementing the DependencyInjectionProgram**

**Customer.java**

public class Customer {

    private String id;

    private String name;

    public Customer(String id, String name) {

        this.id = id;

        this.name = name;

    }

    public String getId() { return id; }

    public String getName() { return name; }

}

**CustomerRepository.java**

public interface CustomerRepository {

    Customer findCustomerById(String id);

}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

    public Customer findCustomerById(String id) {

        return new Customer(id, "John Doe");

    }

}

**CustomerService.java**

public class CustomerService {

    private CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public void printCustomerDetails(String id) {

        Customer customer = customerRepository.findCustomerById(id);

        System.out.println("Customer ID: " + customer.getId());

        System.out.println("Customer Name: " + customer.getName());

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        TestDI.main(args);

    }

}

**TestDI.java**

public class TestDI {

    public static void main(String[] args) {

        CustomerRepository repository = new CustomerRepositoryImpl();

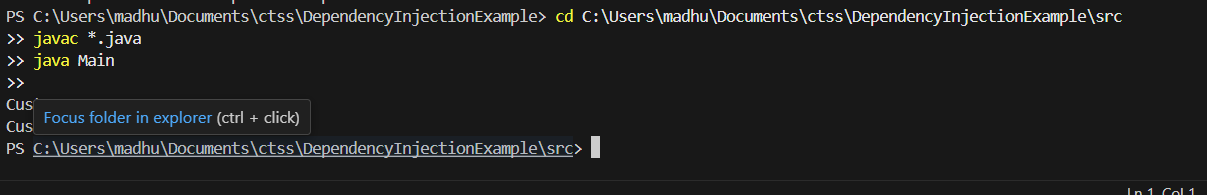
        CustomerService service = new CustomerService(repository);

service.printCustomerDetails("CUST123");

    }

}

Output:

****