DESIGN PATTERNS

Exercise 1: Implementing the Singleton Pattern

```
interface LoggerInterface {
    void log(String message);
}

// Logger class following Singleton Design Pattern and SOLID Principles
class Logger implements LoggerInterface {
    // Eager initialization of the single instance
    private static final Logger instance = new Logger();

    // Private constructor to prevent external instantiation
    private Logger() {
        System.out.println("Logger instance created.");
    }

    // Public static method to provide access to the single instance
    public static Logger getInstance() {
        return instance;
    }

    // Method to perform logging
    @Override
    public void log(String message) {
        System.out.println("[LOG]: " + message);
    }
}

// Test class to verify Singleton behavior
public class SingletonPatternExample {
    public static void main(String[] args) {
        LoggerInterface logger1 = Logger.getInstance();
        LoggerInterface logger2 = Logger.getInstance();

        logger1.log("Logging from logger1.");
        logger2.log("Logging from logger2.");
```

```
        // Verifying both references point to the same instance
        if (logger1 == logger2) {
            System.out.println("Both logger1 and logger2 are the same
instance. Singleton works!");
        } else {
            System.out.println(" Different instances detected. Singleton
failed.");
        }
    }
}
```

OUTPUT:

```
.SingletonPatternExample'
Logger instance created.
[LOG]: Logging from logger1.
[LOG]: Logging from logger2.
Both logger1 and logger2 are the same instance. Singleton works!
PS C:\Users\nehar\OneDrive\vs java>
```

Exercise 2: Implementing the Factory Method Pattern

```
interface Document {
    void open();
}

// 2. Concrete Document Classes (SRP, LSP)
class WordDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening a Word Document.");
    }
}

class PdfDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening a PDF Document.");
    }
}
```

```java
class ExcelDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening an Excel Document.");
    }
}

// 3. Abstract Factory Class (OCP, DIP)
abstract class DocumentFactory {
    // Factory Method
    public abstract Document createDocument();
}

// 4. Concrete Factories for each Document Type (OCP)
class WordDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new WordDocument();
    }
}

class PdfDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new PdfDocument();
    }
}

class ExcelDocumentFactory extends DocumentFactory {
    @Override
    public Document createDocument() {
        return new ExcelDocument();
    }
}

// 5. Test class (Client Code)
public class FactoryMethodPatternExample {
    public static void main(String[] args) {
        // Word document creation
        DocumentFactory wordFactory = new WordDocumentFactory();
```

```
        Document wordDoc = wordFactory.createDocument();
        wordDoc.open();

        // PDF document creation
        DocumentFactory pdfFactory = new PdfDocumentFactory();
        Document pdfDoc = pdfFactory.createDocument();
        pdfDoc.open();

        // Excel document creation
        DocumentFactory excelFactory = new ExcelDocumentFactory();
        Document excelDoc = excelFactory.createDocument();
        excelDoc.open();

        // Verifying SOLID behavior
        System.out.println("\n Factory Method Pattern with SOLID
Principles Executed Successfully!");
    }
}
```

OUTPUT:

```
.FactoryMethodPatternExample'
Opening a Word Document.
Opening a PDF Document.
Opening an Excel Document.

 Factory Method Pattern with SOLID Principles Executed Successfully!
```

Exercise 3: Implementing the Builder Pattern

```
interface ComputerPlan {
    String getConfiguration();
}

class Computer implements ComputerPlan {
    private final String CPU;
    private final String RAM;
    private final String storage;
    private final String graphicsCard;
```

```
    private final String operatingSystem;

    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.storage = builder.storage;
        this.graphicsCard = builder.graphicsCard;
        this.operatingSystem = builder.operatingSystem;
    }

    public static class Builder {
        private String CPU;
        private String RAM;
        private String storage;
        private String graphicsCard;
        private String operatingSystem;

        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 Builder setOperatingSystem(String operatingSystem) {
            this.operatingSystem = operatingSystem;
            return this;
```

```
        }

        public Computer build() {
            return new Computer(this);
        }
    }

    @Override
    public String getConfiguration() {
        return "CPU: " + CPU + ", RAM: " + RAM + ", Storage: " + storage +
               ", Graphics: " + graphicsCard + ", OS: " + operatingSystem;
    }
}

public class BuilderPatternExample {
    public static void main(String[] args) {
        Computer basicPC = new Computer.Builder()
                .setCPU("Intel i3")
                .setRAM("8GB")
                .setStorage("256GB SSD")
                .setOperatingSystem("Windows 10")
                .build();

        Computer gamingPC = new Computer.Builder()
                .setCPU("Intel i9")
                .setRAM("32GB")
                .setStorage("1TB NVMe")
                .setGraphicsCard("NVIDIA RTX 4090")
                .setOperatingSystem("Windows 11")
                .build();

        Computer linuxServer = new Computer.Builder()
                .setCPU("AMD Ryzen 9")
                .setRAM("64GB")
                .setStorage("2TB SSD")
                .setOperatingSystem("Ubuntu Server")
                .build();

        System.out.println("Basic PC: " + basicPC.getConfiguration());
        System.out.println("Gaming PC: " + gamingPC.getConfiguration());
```

```
        System.out.println("Linux Server: " +
linuxServer.getConfiguration());
    }
}
```

OUTPUT:

```
Basic PC: CPU: Intel i3, RAM: 8GB, Storage: 256GB SSD, Graphics: null, OS: Windows 10
Gaming PC: CPU: Intel i9, RAM: 32GB, Storage: 1TB NVMe, Graphics: NVIDIA RTX 4090, OS: Windows 11
Linux Server: CPU: AMD Ryzen 9, RAM: 64GB, Storage: 2TB SSD, Graphics: null, OS: Ubuntu Server
```

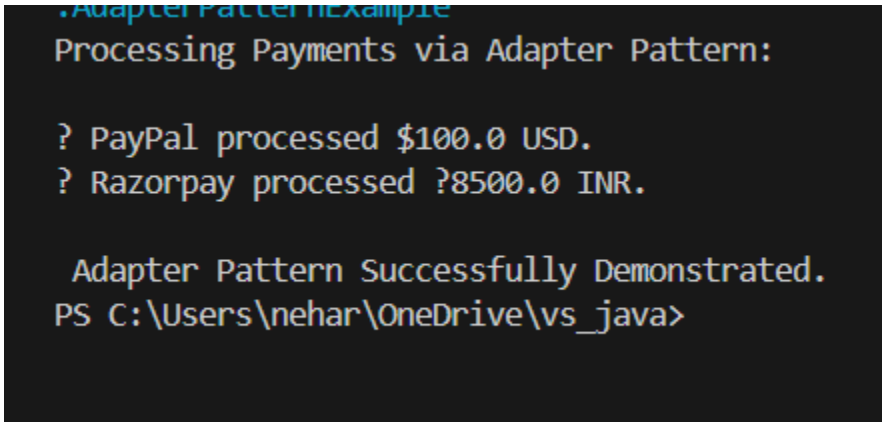
Exercise 4: Implementing the Adapter Pattern

```
// Target Interface
interface PaymentProcessor {
    void processPayment(double amount);
}

// Adaptee 1 - PayPal (Incompatible Interface)
class PayPalGateway {
    public void sendPayment(double amountInUSD) {
        System.out.println("✅ PayPal processed $" + amountInUSD + "
USD.");
    }
}

// Adaptee 2 - Razorpay (Incompatible Interface)
class RazorpayGateway {
    public void makePaymentInINR(double amountInINR) {
        System.out.println("✅ Razorpay processed ₹" + amountInINR + "
INR.");
    }
}

// Adapter 1 - PayPal Adapter
class PayPalAdapter implements PaymentProcessor {
    private PayPalGateway paypal;

    public PayPalAdapter(PayPalGateway paypal) {
        this.paypal = paypal;
```

```
    }

    @Override
    public void processPayment(double amount) {
        paypal.sendPayment(amount); // Delegating to PayPal-specific
method
    }
}

// Adapter 2 - Razorpay Adapter
class RazorpayAdapter implements PaymentProcessor {
    private RazorpayGateway razorpay;

    public RazorpayAdapter(RazorpayGateway razorpay) {
        this.razorpay = razorpay;
    }

    @Override
    public void processPayment(double amount) {
        razorpay.makePaymentInINR(amount); // Delegating to
Razorpay-specific method
    }
}

// Client code - Test class
public class AdapterPatternExample {
    public static void main(String[] args) {
        PaymentProcessor paypalProcessor = new PayPalAdapter(new
PayPalGateway());
        PaymentProcessor razorpayProcessor = new RazorpayAdapter(new
RazorpayGateway());

        System.out.println("Processing Payments via Adapter Pattern:\n");

        paypalProcessor.processPayment(100.0);
        razorpayProcessor.processPayment(8500.0);

        System.out.println("\n Adapter Pattern Successfully
Demonstrated.");
    }
```

```
}
```

OUTPUT:



Exercise 5: Implementing the Decorator Pattern

```
//  Component Interface (Defines base notification behavior)
interface Notifier {
    void send(String message);
}

//  Concrete Component (Sends notification via Email)
class EmailNotifier implements Notifier {
    @Override
    public void send(String message) {
        System.out.println("📧 Email sent: " + message);
    }
}

// Abstract Decorator (Wraps a Notifier and allows dynamic enhancements)
abstract class NotifierDecorator implements Notifier {
    protected Notifier notifier; // Reference to another Notifier

    public NotifierDecorator(Notifier notifier) {
        this.notifier = notifier;
    }

    @Override
    public void send(String message) {
```

```
        notifier.send(message); // Delegate base notification
    }
}

//  Concrete Decorator - SMS
class SMSNotifierDecorator extends NotifierDecorator {
    public SMSNotifierDecorator(Notifier notifier) {
        super(notifier);
    }

    @Override
    public void send(String message) {
        super.send(message); // Send original notification
        sendSMS(message);    // Add SMS functionality
    }

    private void sendSMS(String message) {
        System.out.println("📱 SMS sent: " + message);
    }
}

// Concrete Decorator - Slack
class SlackNotifierDecorator extends NotifierDecorator {
    public SlackNotifierDecorator(Notifier notifier) {
        super(notifier);
    }

    @Override
    public void send(String message) {
        super.send(message);   // Send original + previous notifications
        sendSlack(message);    // Add Slack functionality
    }

    private void sendSlack(String message) {
        System.out.println("💬 Slack message sent: " + message);
    }
}

//  Test Class
public class DecoratorPatternExample {
```

```
    public static void main(String[] args) {
        // Base notifier: Email only
        Notifier emailNotifier = new EmailNotifier();

        // Email + SMS
        Notifier smsEmailNotifier = new
SMSNotifierDecorator(emailNotifier);

        // Email + SMS + Slack
        Notifier fullNotifier = new
SlackNotifierDecorator(smsEmailNotifier);

        // Test: send a notification
        System.out.println(" Sending notification through all
channels:\n");
        fullNotifier.send("Your order #12345 has been shipped.");

        System.out.println("\n Decorator Pattern implemented with SOLID
principles.");
    }
}
```

OUTPUT:

```
  Sending notification through all channels:

 ? Email sent: Your order #12345 has been shipped.
 ? SMS sent: Your order #12345 has been shipped.
 ? Slack message sent: Your order #12345 has been shipped.

  Decorator Pattern implemented with SOLID principles.
```

Exercise 6: Implementing the Proxy Pattern

```
// Step 2: Subject Interface
interface Image {
    void display();
}
```

```java
// Step 3: Real Subject - Loads image from remote server
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);
    }

    @Override
    public void display() {
        System.out.println(" Displaying image: " + filename);
    }
}

// Step 4: Proxy Class - Adds lazy loading and caching
class ProxyImage implements Image {
    private String filename;
    private RealImage realImage;

    public ProxyImage(String filename) {
        this.filename = filename;
    }

    @Override
    public void display() {
        if (realImage == null) {
            System.out.println(" Image not loaded yet. Loading now...");
            realImage = new RealImage(filename); // Lazy loading
        } else {
            System.out.println(" Image already loaded. Using cache...");
        }
        realImage.display(); // Delegate to real image
    }
```

```
}

// Step 5: Test Class
public class ProxyPatternExample {
    public static void main(String[] args) {
        Image image1 = new ProxyImage("nature.png");

        System.out.println("\n[First display call]");
        image1.display(); // Loads from server

        System.out.println("\n[Second display call]");
        image1.display(); // Uses cached image

        System.out.println("\n Proxy Pattern executed with lazy loading
and caching.");
    }
}
```

OUTPUT:

```
[First display call]
 Image not loaded yet. Loading now...
 Loading image from remote server: nature.png
 Displaying image: nature.png

[Second display call]
 Image already loaded. Using cache...
 Displaying image: nature.png

 Proxy Pattern executed with lazy loading and caching.
```

Exercise 7: Implementing the Observer Pattern

```
import java.util.*;

// Step 2: Subject Interface
interface Stock {
```

```
    void register(Observer o);
    void deregister(Observer o);
    void notifyObservers();
    void setPrice(double price);
}

// Step 3: Concrete Subject - StockMarket
class StockMarket implements Stock {
    private List<Observer> observers = new ArrayList<>();
    private double stockPrice;

    @Override
    public void register(Observer o) {
        observers.add(o);
        System.out.println("✅ Observer registered: " + o.getName());
    }

    @Override
    public void deregister(Observer o) {
        observers.remove(o);
        System.out.println("❌ Observer deregistered: " + o.getName());
    }

    @Override
    public void notifyObservers() {
        for (Observer o : observers) {
            o.update(stockPrice);
        }
    }

    @Override
    public void setPrice(double price) {
        this.stockPrice = price;
        System.out.println("\n💹 Stock price updated to $" + price);
        notifyObservers();
    }
}

// Step 4: Observer Interface
interface Observer {
```

```
    void update(double price);
    String getName();
}

// Step 5: Concrete Observer - MobileApp
class MobileApp implements Observer {
    private String name;

    public MobileApp(String name) {
        this.name = name;
    }

    @Override
    public void update(double price) {
        System.out.println("📱 " + name + " received update: New stock
price is $" + price);
    }

    @Override
    public String getName() {
        return name;
    }
}

// Step 5: Concrete Observer - WebApp
class WebApp implements Observer {
    private String name;

    public WebApp(String name) {
        this.name = name;
    }

    @Override
    public void update(double price) {
        System.out.println("💻 " + name + " received update: New stock
price is $" + price);
    }

    @Override
    public String getName() {
```

```
        return name;
    }
}

// Step 6: Test Class
public class ObserverPatternExample {
    public static void main(String[] args) {
        StockMarket stockMarket = new StockMarket();

        Observer mobileApp = new MobileApp("Mobile Client");
        Observer webApp = new WebApp("Web Client");

        stockMarket.register(mobileApp);
        stockMarket.register(webApp);

        stockMarket.setPrice(102.5);
        stockMarket.setPrice(108.3);

        stockMarket.deregister(mobileApp);
        stockMarket.setPrice(115.0);

        System.out.println("\n✅ Observer Pattern implemented
successfully.");
    }
}
```

OUTPUT:

```
.ObserverPatternExample
? Observer registered: Mobile Client
? Observer registered: Web Client

? Stock price updated to $102.5
? Mobile Client received update: New stock price is $102.5
? Web Client received update: New stock price is $102.5

? Stock price updated to $108.3
? Mobile Client received update: New stock price is $108.3
? Web Client received update: New stock price is $108.3
? Observer deregistered: Mobile Client

? Stock price updated to $115.0
? Web Client received update: New stock price is $115.0

? Observer Pattern implemented successfully.
```

Exercise 8: Implementing the Strategy Pattern

```java
// Step 2: Strategy Interface
interface PaymentStrategy {
    void pay(double amount);
}

// Step 3: Concrete Strategy - Credit Card
class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;

    public CreditCardPayment(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    @Override
    public void pay(double amount) {
        System.out.println(" Paid $" + amount + " using Credit Card (****
" + cardNumber.substring(cardNumber.length() - 4) + ")");
```

```
    }
}

// Step 3: Concrete Strategy - PayPal
class PayPalPayment implements PaymentStrategy {
    private String email;

    public PayPalPayment(String email) {
        this.email = email;
    }

    @Override
    public void pay(double amount) {
        System.out.println(" Paid $" + amount + " using PayPal account: "
+ email);
    }
}

// Step 4: Context Class
class PaymentContext {
    private PaymentStrategy strategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
        System.out.println(" Payment strategy switched to: " +
strategy.getClass().getSimpleName());
    }

    public void executePayment(double amount) {
        if (strategy != null) {
            strategy.pay(amount);
        } else {
            System.out.println(" No payment strategy selected.");
        }
    }
}

// Step 5: Test Class
public class StrategyPatternExample {
    public static void main(String[] args) {
```

```
        PaymentContext context = new PaymentContext();

        // Use Credit Card Payment
        context.setPaymentStrategy(new
CreditCardPayment("1234567812345678"));
        context.executePayment(150.0);

        // Switch to PayPal Payment
        context.setPaymentStrategy(new PayPalPayment("user@example.com"));
        context.executePayment(89.99);

        System.out.println("\n Strategy Pattern implemented
successfully.");
    }
}
```

OUTPUT:

```
.StrategyPatternExample
 Payment strategy switched to: CreditCardPayment
 Paid $150.0 using Credit Card (**** 5678)
 Payment strategy switched to: PayPalPayment
 Paid $89.99 using PayPal account: user@example.com

 Strategy Pattern implemented successfully.
```

Exercise 9: Implementing the Command Pattern

```
interface Command {
    void execute();
}

// Step 5: Receiver Class - Light
class Light {
    public void turnOn() {
        System.out.println(" The light is ON.");
    }

    public void turnOff() {
        System.out.println(" The light is OFF.");
    }
}
```

```
// Step 3: Concrete Command - Turn Light ON
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }
}

// Step 3: Concrete Command - Turn Light OFF
class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOff();
    }
}

// Step 4: Invoker Class - RemoteControl
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
        System.out.println(" Command set to: " +
command.getClass().getSimpleName());
    }

    public void pressButton() {
```

```
        if (command != null) {
            command.execute();
        } else {
            System.out.println(" No command set.");
        }
    }
}

// Step 6: Test Class
public class CommandPatternExample {
    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();

        System.out.println("\n Command Pattern implemented
successfully.");
    }
}
```

OUTPUT:

```
Command set to: LightOnCommand
The light is ON.
Command set to: LightOffCommand
The light is OFF.

Command Pattern implemented successfully.
```

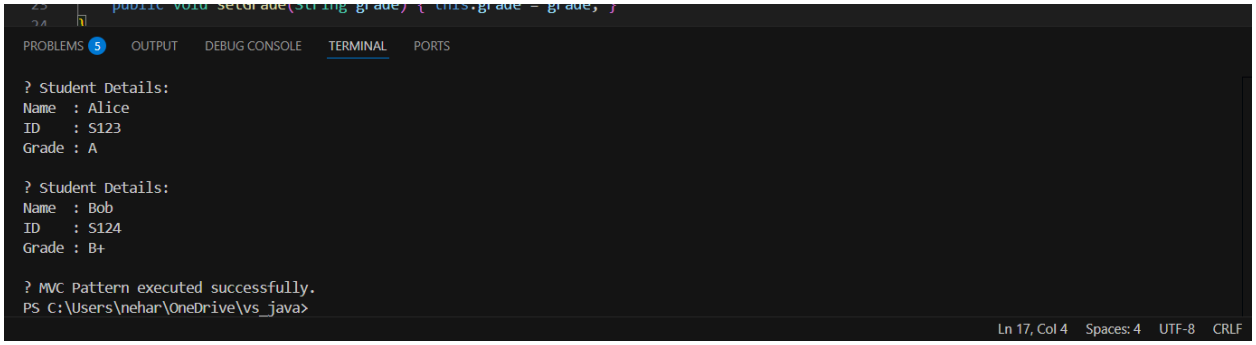
Exercise 10: Implementing the MVC Pattern

```
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;
    }

    // Getters
    public String getName() { return name; }
    public String getId() { return id; }
    public String getGrade() { return grade; }

    // Setters
    public void setName(String name) { this.name = name; }
    public void setId(String id) { this.id = id; }
    public void setGrade(String grade) { this.grade = grade; }
}

// Step 3: View Class
class StudentView {
    public void displayStudentDetails(String name, String id, String
grade) {
        System.out.println("\n📘 Student Details:");
        System.out.println("Name  : " + name);
        System.out.println("ID    : " + id);
        System.out.println("Grade : " + grade);
    }
}

// Step 4: Controller Class
class StudentController {
    private Student model;
    private StudentView view;
```

```
    public StudentController(Student model, StudentView view) {
        this.model = model;
        this.view = view;
    }

    // Update Model
    public void setStudentName(String name) {
        model.setName(name);
    }

    public void setStudentId(String id) {
        model.setId(id);
    }

    public void setStudentGrade(String grade) {
        model.setGrade(grade);
    }

    // Display using View
    public void updateView() {
        view.displayStudentDetails(model.getName(), model.getId(),
model.getGrade());
    }
}

// Step 5: Test Class (Main)
public class MVCPatternExample {
    public static void main(String[] args) {
        // Create Model
        Student student = new Student("Alice", "S123", "A");

        // Create View
        StudentView view = new StudentView();

        // Create Controller
        StudentController controller = new StudentController(student,
view);

        // Initial Display
        controller.updateView();
```

```
        // Update details through controller
        controller.setStudentName("Bob");
        controller.setStudentId("S124");
        controller.setStudentGrade("B+");

        // Updated Display
        controller.updateView();

        System.out.println("\n✅ MVC Pattern executed successfully.");
    }
}
```

OUTPUT:

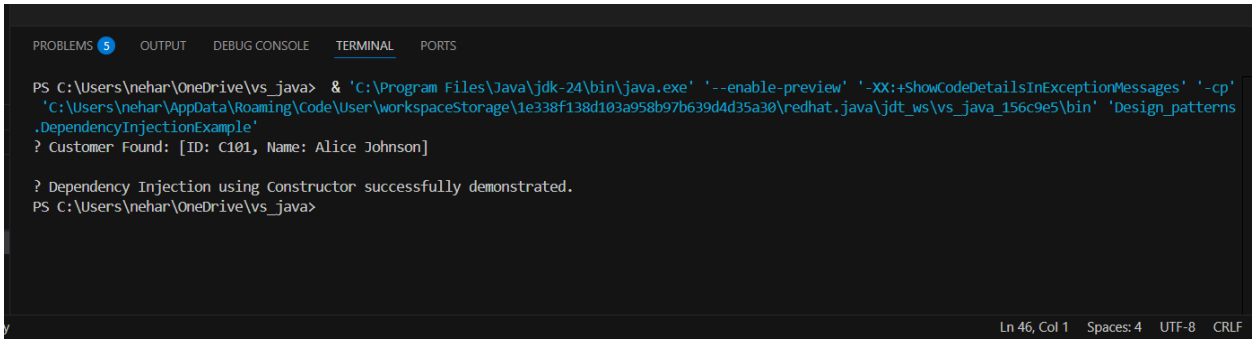


Exercise 11: Implementing Dependency Injection

```
// Step 2: Repository Interface
interface CustomerRepository {
    String findCustomerById(String id);
}

// Step 3: Concrete Repository Implementation
class CustomerRepositoryImpl implements CustomerRepository {
    @Override
    public String findCustomerById(String id) {
        return " Customer Found: [ID: " + id + ", Name: Alice Johnson]";
    }
}
```

```java
// Step 4: Service Class that depends on Repository
class CustomerService {
    private final CustomerRepository repository;

    // Step 5: Constructor Injection
    public CustomerService(CustomerRepository repository) {
        this.repository = repository;
    }

    public void displayCustomer(String id) {
        String customer = repository.findCustomerById(id);
        System.out.println(customer);
    }
}

// Step 6: Main Class to Test DI
public class DependencyInjectionExample {
    public static void main(String[] args) {
        // Create repository (dependency)
        CustomerRepository repository = new CustomerRepositoryImpl();

        // Inject repository into service
        CustomerService service = new CustomerService(repository);

        // Use service
        service.displayCustomer("C101");

        System.out.println("\n Dependency Injection using Constructor
successfully demonstrated.");
    }
}
```

OUTPUT:



ALGORITHMS AND DATA STRUCTURE

Exercise 1: Inventory Management System

```java
// Exercise 1: Inventory Management System
import java.util.*;

class Product {
    int productId;
    String productName;
    int quantity;
    double price;

    public Product(int id, String name, int qty, double price) {
        this.productId = id;
        this.productName = name;
        this.quantity = qty;
        this.price = price;
    }

    public String toString() {
        return productId + ": " + productName + " - Qty: " + quantity + ", Price: " + price;
    }
}
```

```java
public class InventoryManagementSystem {
    Map<Integer, Product> inventory = new HashMap<>();

    public void addProduct(Product p) {
        inventory.put(p.productId, p);
        System.out.println("Added product: " + p);
    }

    public void updateProduct(int id, int qty, double price) {
        if (inventory.containsKey(id)) {
            Product p = inventory.get(id);
            p.quantity = qty;
            p.price = price;
            System.out.println("Updated product: " + p);
        }
    }

    public void deleteProduct(int id) {
        if (inventory.containsKey(id)) {
            System.out.println("Deleted product: " + inventory.get(id));
            inventory.remove(id);
        }
    }

    public static void main(String[] args) {
        InventoryManagementSystem ims = new InventoryManagementSystem();
        ims.addProduct(new Product(101, "Mouse", 50, 499.99));
        ims.updateProduct(101, 40, 459.99);
        ims.deleteProduct(101);
    }
}
```

OUTPUT:

```
Added product: 101: Mouse - Qty: 50, Price: 499.99
Updated product: 101: Mouse - Qty: 40, Price: 459.99
Deleted product: 101: Mouse - Qty: 40, Price: 459.99
```

Exercise 2: E-commerce Platform Search Function

```
import java.util.Arrays;
import java.util.Comparator;

class ECommerceSearch {
    static class Product {
        int productId;
        String productName;
        String category;

        Product(int id, String name, String cat) {
            productId = id;
            productName = name;
            category = cat;
        }
    }

    public static int linearSearch(Product[] products, String name) {
        for (int i = 0; i < products.length; i++) {
            if (products[i].productName.equals(name))
                return i;
        }
        return -1;
    }

    public static int binarySearch(Product[] products, String name) {
        int left = 0, right = products.length - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            int cmp = products[mid].productName.compareTo(name);
            if (cmp == 0) return mid;
            else if (cmp < 0) left = mid + 1;
            else right = mid - 1;
        }
        return -1;
    }

    public static void main(String[] args) {
```

```
        Product[] products = {
            new Product(1, "Laptop", "Electronics"),
            new Product(2, "Phone", "Electronics"),
            new Product(3, "Tablet", "Electronics")
        };
        Arrays.sort(products,
Comparator.comparing((ECommerceSearch.Product p) -> p.productName));

        System.out.println("Linear Search Index: " +
linearSearch(products, "Phone"));
        System.out.println("Binary Search Index: " +
binarySearch(products, "Phone"));
    }
}
```

OUTPUT:

```
Linear Search Index: 1
Binary Search Index: 1
```

Exercise 3: Sorting Customer Orders

```
class CustomerOrderSorting {
    static class Order {
        int orderId;
        String customerName;
        double totalPrice;

        Order(int id, String name, double price) {
            orderId = id;
            customerName = name;
            totalPrice = price;
        }
    }

    public static void bubbleSort(Order[] orders) {
        int n = orders.length;
        for (int i = 0; i < n - 1; i++) {
```

```
            for (int j = 0; j < n - i - 1; j++) {
                if (orders[j].totalPrice > orders[j + 1].totalPrice) {
                    Order temp = orders[j];
                    orders[j] = orders[j + 1];
                    orders[j + 1] = temp;
                }
            }
        }
    }

    public static void quickSort(Order[] orders, int low, int high) {
        if (low < high) {
            int pi = partition(orders, low, high);
            quickSort(orders, low, pi - 1);
            quickSort(orders, pi + 1, high);
        }
    }

    private static int partition(Order[] arr, int low, int high) {
        double pivot = arr[high].totalPrice;
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (arr[j].totalPrice <= pivot) {
                i++;
                Order temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        Order temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        return i + 1;
    }

    public static void main(String[] args) {
        Order[] orders = {
            new Order(1, "Alice", 500.0),
            new Order(2, "Bob", 1500.0),
            new Order(3, "Charlie", 800.0)
```

```
        };
        quickSort(orders, 0, orders.length - 1);
        System.out.println("Sorted Orders by Total Price:");
        for (Order o : orders)
            System.out.println(o.customerName + ": " + o.totalPrice);
    }
}
```

OUTPUT:

```
Sorted Orders by Total Price:
Alice: 500.0
Charlie: 800.0
Bob: 1500.0
```

Exercise 4: Employee Management System

```
class EmployeeManagement {
    static class Employee {
        int employeeId;
        String name;
        String position;
        double salary;

        Employee(int id, String name, String pos, double sal) {
            employeeId = id;
            this.name = name;
            position = pos;
            salary = sal;
        }
    }

    Employee[] employees = new Employee[100];
    int size = 0;

    public void addEmployee(Employee emp) {
        employees[size++] = emp;
        System.out.println("Added employee: " + emp.name);
    }
```

```
    public Employee searchEmployee(int id) {
        for (int i = 0; i < size; i++) {
            if (employees[i].employeeId == id) return employees[i];
        }
        return null;
    }

    public void deleteEmployee(int id) {
        for (int i = 0; i < size; i++) {
            if (employees[i].employeeId == id) {
                for (int j = i; j < size - 1; j++) {
                    employees[j] = employees[j + 1];
                }
                size--;
                System.out.println("Deleted employee with ID: " + id);
                break;
            }
        }
    }

    public void traverseEmployees() {
        System.out.println("All Employees:");
        for (int i = 0; i < size; i++) {
            System.out.println(employees[i].name);
        }
    }

    public static void main(String[] args) {
        EmployeeManagement em = new EmployeeManagement();
        em.addEmployee(new Employee(101, "Neha", "Engineer", 60000));
        em.traverseEmployees();
    }
}
```

OUTPUT:

```
Added employee: Neha
All Employees:
Neha
```

Exercise 5: Task Management System

```
class TaskManagementSystem {
    static class Task {
        int taskId;
        String taskName;
        String status;
        Task next;

        Task(int id, String name, String status) {
            taskId = id;
            taskName = name;
            this.status = status;
            next = null;
        }
    }

    Task head = null;

    public void addTask(Task task) {
        task.next = head;
        head = task;
        System.out.println("Added task: " + task.taskName);
    }

    public Task searchTask(int id) {
        Task current = head;
        while (current != null) {
            if (current.taskId == id) return current;
            current = current.next;
        }
        return null;
    }

    public void deleteTask(int id) {
        Task current = head, prev = null;
        while (current != null) {
            if (current.taskId == id) {
                if (prev == null) head = current.next;
                else prev.next = current.next;
```

```
                System.out.println("Deleted task with ID: " + id);
                return;
            }
            prev = current;
            current = current.next;
        }
    }

    public void traverse() {
        System.out.println("All Tasks:");
        Task current = head;
        while (current != null) {
            System.out.println(current.taskName);
            current = current.next;
        }
    }

    public static void main(String[] args) {
        TaskManagementSystem tms = new TaskManagementSystem();
        tms.addTask(new Task(1, "Complete Assignment", "Pending"));
        tms.traverse();
    }
}
```

OUTPUT:

```
Added task: Complete Assignment
All Tasks:
Complete Assignment
```

Exercise 6: Library Management System

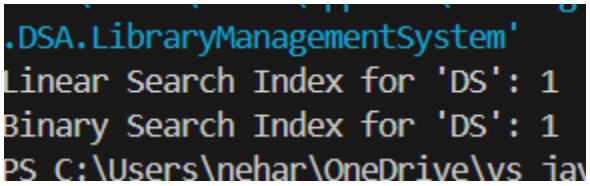
```
import java.util.Arrays;

import java.util.Comparator;
```

```
class LibraryManagementSystem {
    static class Book {
        int bookId;
        String title;
        String author;

        Book(int id, String title, String author) {
            bookId = id;
            this.title = title;
            this.author = author;
        }
    }

    public static int linearSearch(Book[] books, String title) {
        for (int i = 0; i < books.length; i++) {
            if (books[i].title.equals(title)) return i;
        }
        return -1;
    }

    public static int binarySearch(Book[] books, String title) {
        int left = 0, right = books.length - 1;
```

```
        while (left <= right) {
            int mid = (left + right) / 2;
            int cmp = books[mid].title.compareTo(title);
            if (cmp == 0) return mid;
            else if (cmp < 0) left = mid + 1;
            else right = mid - 1;
        }
        return -1;
    }

    public static void main(String[] args) {
        Book[] books = {
            new Book(1, "AI", "Russell"),
            new Book(2, "DS", "Tanenbaum"),
            new Book(3, "OS", "Silberschatz")
        };

        Arrays.sort(books,
Comparator.comparing((LibraryManagementSystem.Book b) -> b.title));
        System.out.println("Linear Search Index for 'DS': " +
linearSearch(books, "DS"));
        System.out.println("Binary Search Index for 'DS': " +
binarySearch(books, "DS"));
    }
```

```
}
```

OUTPUT:



Exercise 7: Financial Forecasting

```
import java.util.Scanner;



class FinancialForecasting {

   public static double calculateFutureValue(double baseValue, double
growthRate, int years) {



    if (years == 0) {



    return baseValue;



}
```

```
return calculateFutureValue(baseValue, growthRate, years - 1) * (1 +
growthRate);



}



public static void main(String[] args) {


Scanner sc = new Scanner(System.in);




System.out.print("Enter initial value: ");


double baseValue = sc.nextDouble();




System.out.print("Enter annual growth rate(in percentage): ");


double growthRate = sc.nextDouble();




System.out.print("Enter number of years to forecast: ");


int years = sc.nextInt();
```

```java
double futureValue = calculateFutureValue(baseValue, growthRate, years);



System.out.printf("Future value after %d years: %.2f\n", years,
futureValue);


}


}
```

OUTPUT:

```
Enter initial value: 5
Enter annual growth rate(in percentage): 5
Enter number of years to forecast: 3
Future value after 3 years: 1080.00
```