Design Patterns and Principles

Exercise-1: **Implementing the Singleton Pattern**

**ABMain.java**

class ABMain {

    public static void main(String[] args){

        Logger l1 = Logger.getInstance();

        Logger l2 = Logger.getInstance();

        l1.log("First message");

        l2.log("Second message");

        System.out.println(l1 == l2);

    }

}

**Logger.java**

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;

    }

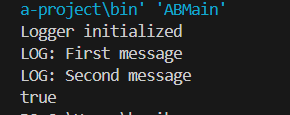
    public void log(String msg){

        System.out.println("LOG: " + msg);

    }

**}**

**Output:**

****

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

interface Document{

void open();

}

class WordDocument implements Document{

public void open()

{

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

}

}

class PdfDocument implements Document {

public void open() {

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

}

}

abstract class DocumentFactory{

public abstract Document createDocument();

}

class WordFactory extends DocumentFactory

{

public Document createDocument() {

return new WordDocument();

}

}

class PdfFactory extends DocumentFactory

{

public Document createDocument(){

return new PdfDocument();

}

}

class Main{

public static void main(String[] args)

{

DocumentFactory wordFactory = new WordFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfFactory();

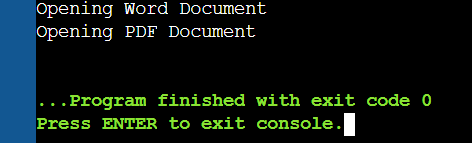
Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

}

}

Output:



Exercise-3: **Implementing the Builder Pattern**

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 void showConfig() {

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

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

        System.out.println("Storage: " + (storage != null ? storage : "Not Added"));

        System.out.println("Graphics Card: " + (graphicsCard != null ? graphicsCard : "Not Added"));

    }

    public static class Builder {

        private String CPU;

        private String RAM;

        private String storage;

        private String graphicsCard;

        public Builder(String CPU, String RAM){

            this.CPU = CPU;

            this.RAM = RAM;

        }

        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);

        }

    }

}

public class BuilderMain{

    public static void main(String[] args){

        // Creating computer with only required fields

        Computer basicComputer = new Computer.Builder("Intel i5", "8GB").build();

        System.out.println("Basic Configuration:");

        basicComputer.showConfig();

        System.out.println();

        Computer gamingPC = new Computer.Builder("AMD Ryzen 9", "32GB")

                                .setStorage("1TB SSD")

                                .setGraphicsCard("NVIDIA RTX 4070")

                                .build();

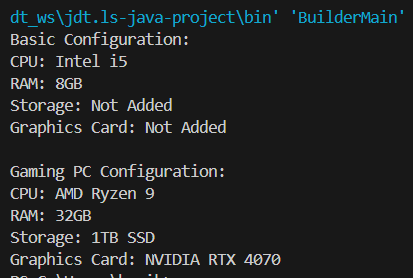
        System.out.println("Gaming PC Configuration:");

        gamingPC.showConfig();

    }

}

Output:



**Exercise 4: Implementing the Adapter Pattern**

interface PayX {

    void pay(double amt, String forWhat);}

class PP {

    void doTransfer(double amt, String note) {

        System.out.println("PP ₹" + amt + " done for: " + note);}

}

class ST {

    void charge(String desc, double val){

        System.out.println("ST ₹" + val + " charged for: " + desc);

    }

}

class PPAdapter implements PayX{

    PP p = new PP();

    public void pay(double amt, String msg) {

        p.doTransfer(amt, msg);

    }

}

class STAdapter implements PayX{

    ST s = new ST();

    public void pay(double amt, String msg) {

        s.charge(msg, amt);

    }}

public class PaymentDemo{

    public static void main(String[] args){

        PayX p1 = new PPAdapter();

        PayX p2 = new STAdapter();

        run(p1, 750.0, "cloud bill");

        run(p2, 2100.5, "design fee");

    }

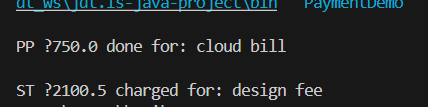
    static void run(PayX p, double a, String msg){

        System.out.println();

        p.pay(a, msg);

    }}

Output:



**Exercise 5: Implementing the Decorator Pattern**

interface Notify{

    void send(String msg);

}

class Email implements Notify

 {

    public void send(String msg)

    {

        System.out.println("📧: " + msg);

    }

}

abstract class Wrap implements Notify

{

    Notify n;

    Wrap(Notify n) { this.n = n; }}

class SMS extends Wrap {

    SMS(Notify n) { super(n); }

    public void send(String msg) {

        n.send(msg);

        System.out.println("📱: " + msg);}

}

class Slack extends Wrap {

    Slack(Notify n) { super(n); }

    public void send(String msg)

    {

        n.send(msg);

        System.out.println("💬: " + msg);

    }

}

public class NotifyDemo{

    public static void main(String[] args)

     {

        Notify basic = new Email();

        Notify combo1 = new SMS(basic);

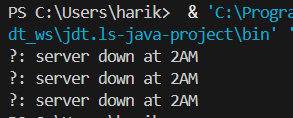
        Notify combo2 = new Slack(combo1);

        combo2.send("server down at 2AM");

    }

}

Output:



**Exercise 6: Implementing the Proxy Pattern**

interface Image {

    void display();

}

class RealImage implements Image {

    private String file;

    RealImage(String file) {

        this.file = file;

        load();

    }

    private void load() {

        System.out.println("Loading " + file + " from server...");

    }

    public void display() {

        System.out.println("Showing " + file);

    }

}

class ProxyImage implements Image{

    private RealImage img;

    private String file;

    ProxyImage(String file){

        this.file = file;

    }

    public void display(){

        if (img == null) img = new RealImage(file);

        img.display();

    }

}

public class ProxyImageDemo{

    public static void main(String[] args){

        Image a = new ProxyImage("pic1.jpg");

        Image b = new ProxyImage("pic2.png");

        a.display();

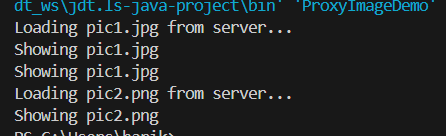
        a.display();

        b.display();

    }

}

Output:



**Exercise 7: Implementing the Observer Pattern**

import java.util.\*;

interface Stock{

void addObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers(double price);

}

interface Observer{

void update(double price);

}

class StockMarket implements Stock{

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

private double price;

public void addObserver(Observer o) {

obs.add(o);

}

public void removeObserver(Observer o){

obs.remove(o);

}

public void notifyObservers(double price){

for (Observer o : obs)

o.update(price);

}

public void setPrice(double p) {

this.price = p;

notifyObservers(p);

}

}

class MobileApp implements Observer {

private String name;

MobileApp(String name){

this.name = name;

}

public void update(double price){

System.out.println(name + " (Mobile) got update: ₹" + price);

}

}

class WebApp implements Observer{

private String name;

WebApp(String name){

this.name = name;

}

public void update(double price) {

System.out.println(name + " (Web) got update: ₹" + price);

}

}

public class Main {

public static void main(String[] args) {

StockMarket sm = new StockMarket();

Observer mob1 = new MobileApp("AppX");

Observer web1 = new WebApp("SiteY");

sm.addObserver(mob1);

sm.addObserver(web1);

sm.setPrice(1200.50);

sm.setPrice(1225.75);

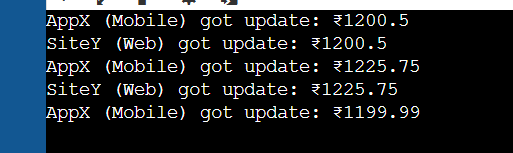
sm.removeObserver(web1);

sm.setPrice(1199.99);

}

}

Output:



**Exercise 8: Implementing the Strategy Pattern**

interface PaymentStrategy{

void pay(double amt);

}

class CreditCardPayment implements PaymentStrategy{

public void pay(double amt) {

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

}

}

class PayPalPayment implements PaymentStrategy{

public void pay(double amt) {

System.out.println("Paid ₹" + amt + " via PayPal.");

}

}

class PaymentContext {

private PaymentStrategy method;

PaymentContext(PaymentStrategy method) {

this.method = method;

}

public void process(double amt) {

method.pay(amt);

}

}

public class Main {

public static void main(String[] args) {

PaymentContext ctx1 = new PaymentContext(new CreditCardPayment());

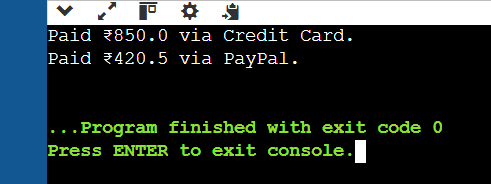
PaymentContext ctx2 = new PaymentContext(new PayPalPayment());

ctx1.process(850.0);

ctx2.process(420.5);

}

}

Output:  


**Exercise 9: Implementing the Command Pattern**

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

{

    Light l;

    LightOnCommand(Light l){

        this.l = l;

    }

    public void execute(){

        l.on();

    }

}

class LightOffCommand implements Command{

    Light l;

    LightOffCommand(Light l){

        this.l = l;

    }

    public void execute(){

        l.off();

    }

}

class RemoteControl{

    Command c;

    void set(Command c){

        this.c = c;

    }

    void press(){

        c.execute();

    }

}

public class HomeCmd{

    public static void main(String[] args){

        Light bulb = new Light();

        Command on = new LightOnCommand(bulb);

        Command off = new LightOffCommand(bulb);

        RemoteControl rc = new RemoteControl();

        rc.set(on);

        rc.press();

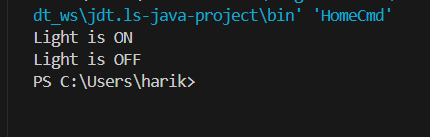
        rc.set(off);

        rc.press();

    }

}

Output:



**Exercise 10: Implementing the MVC Pattern**

class Student{

private String name;

private int id;

private String grade;

Student(String name, int id, String grade){

this.name = name;

this.id = id;

this.grade = grade;

}

String getName(){

return name;

}

void setName(String name){

this.name = name;

}

int getId(){

return id;

}

void setId(int id){

this.id = id;

}

String getGrade(){

return grade;

}

void setGrade(String grade){

this.grade = grade;

}

}

class StudentView{

void displayStudentDetails(String name, int id, String grade){

System.out.println("Name: " + name);

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

System.out.println("Grade: " + grade);

}

}

class StudentController{

private Student s;

private StudentView v;

StudentController(Student s, StudentView v){

this.s = s;

this.v = v;

}

void setName(String name){

s.setName(name);

}

void setGrade(String grade){

s.setGrade(grade);

}

void updateView(){

v.displayStudentDetails(s.getName(), s.getId(), s.getGrade());

}

}

public class Main{

public static void main(String[] args){

Student s = new Student("Hari", 101, "A");

StudentView v = new StudentView();

StudentController c = new StudentController(s, v);

c.updateView();

c.setName("Harish");

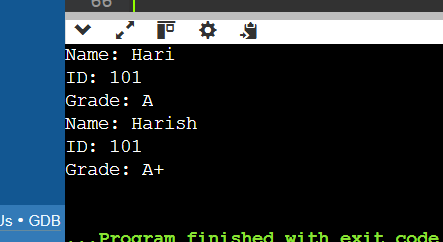
c.setGrade("A+");

c.updateView();

}

}

Output:



**Exercise 11: Implementing Dependency Injection**

interface CustomerRepository{

    String findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository{

    public String findCustomerById(int id){

        return "Customer#" + id + " - Hari Kumar";

    }

}

class CustomerService{

    private CustomerRepository repo;

    CustomerService(CustomerRepository repo){

        this.repo = repo;

    }

    void getCustomer(int id){

        String data = repo.findCustomerById(id);

        System.out.println(data);

    }

}

public class CustomerApp{

    public static void main(String[] args){

        CustomerRepository repo = new CustomerRepositoryImpl();

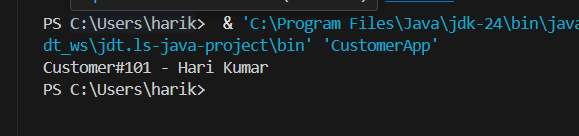
        CustomerService service = new CustomerService(repo);

        service.getCustomer(101);

    }

}

Output:



**Algorithms\_Data Structures**

**Exercise 1: Inventory Management System**

import java.util.\*;

class Product{

int pid;

String pname;

int qty;

double price;

Product(int pid, String pname, int qty, double price){

this.pid = pid;

this.pname = pname;

this.qty = qty;

this.price = price;

}

public void display(){

System.out.println("ID: " + pid + ", Name: " + pname + ", Qty: " + qty + ", Price: ₹" + price);

}

}

class Inventory{

HashMap<Integer, Product> inv = new HashMap<>();

public void addProd(Product p){

if(inv.containsKey(p.pid)){

System.out.println("Prod ID already exists.");

}else{

inv.put(p.pid, p);

System.out.println("Prod added.");

}

}

public void updateProd(int pid, int qty, double price){

if(inv.containsKey(pid)){

Product p = inv.get(pid);

p.qty = qty;

p.price = price;

System.out.println("Prod updated.");

}else{

System.out.println("Prod not found.");

}

}

public void delProd(int pid){

if(inv.containsKey(pid)){

inv.remove(pid);

System.out.println("Prod deleted.");

}else{

System.out.println("Prod not found.");

}

}

public void showAll(){

for(Product p : inv.values()){

p.display();

}

}

}

public class Main{

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

Inventory inv = new Inventory();

while(true){

System.out.println("\n1.Add 2.Update 3.Delete 4.Show 5.Exit");

int ch = sc.nextInt();

if(ch==1){

System.out.print("Enter id name qty price: ");

int id = sc.nextInt();

String name = sc.next();

int q = sc.nextInt();

double pr = sc.nextDouble();

Product p = new Product(id, name, q, pr);

inv.addProd(p);

}else if(ch==2){

System.out.print("Enter id to update, new qty & price: ");

int id = sc.nextInt();

int q = sc.nextInt();

double pr = sc.nextDouble();

inv.updateProd(id, q, pr);

}else if(ch==3){

System.out.print("Enter id to delete: ");

int id = sc.nextInt();

inv.delProd(id);

}else if(ch==4){

inv.showAll();

}else if(ch==5){

System.out.println("Exit.");

break;

}else{

System.out.println("Invalid.");

}

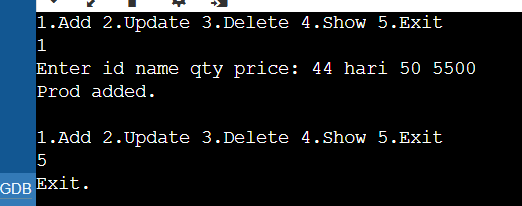
}

sc.close();

}

}

Output:



**Exercise 2: E-commerce Platform Search Function**

**What is Big O Notation?**  
Big O notation is used to describe how fast or slow an algorithm runs as the size of input increases. It tells us the worst-case time complexity of a program. This helps in comparing which algorithm is better in terms of speed and performance.

**best, Average, and Worst Case in Search:**

* **best case**: The item is found at the first position.
* **Average case**: The item is found somewhere in the middle.
* **Worst case**: The item is not found or at the last position.  
  For example, in linear search, best case is O(1), and worst case is O(n). In binary search, best case is O(1), but worst case is O(log n).

**Code:**

import java.util.\*;

class Product{

int pid;

String pname;

String cat;

Product(int pid, String pname, String cat){

this.pid = pid;

this.pname = pname;

this.cat = cat;

}

void show(){

System.out.println("ID: " + pid + ", Name: " + pname + ", Cat: " + cat);

}

}

class SearchFunc{

public static Product linearSearch(Product[] arr, String key){

for(Product p : arr){

if(p.pname.equalsIgnoreCase(key)){

return p;

}

}

return null;

}

public static Product binarySearch(Product[] arr, String key){

int l=0, r=arr.length-1;

while(l<=r){

int mid=(l+r)/2;

int cmp = arr[mid].pname.compareToIgnoreCase(key);

if(cmp==0) return arr[mid];

else if(cmp<0) l=mid+1;

else r=mid-1;

}

return null;

}

}

public class Main{

public static void main(String[] args){

Product[] prods = {

new Product(101,"Laptop","Electronics"),

new Product(102,"Shirt","Fashion"),

new Product(103,"Phone","Electronics"),

new Product(104,"Shoes","Footwear"),

new Product(105,"Watch","Accessories")

};

Product[] sorted = Arrays.copyOf(prods, prods.length);

Arrays.sort(sorted, (a,b) -> a.pname.compareToIgnoreCase(b.pname));

Scanner sc = new Scanner(System.in);

System.out.print("Enter prod name to search: ");

String key = sc.nextLine();

System.out.println("Linear Search:");

Product res1 = SearchFunc.linearSearch(prods, key);

if(res1!=null) res1.show(); else System.out.println("Not found.");

System.out.println("Binary Search:");

Product res2 = SearchFunc.binarySearch(sorted, key);

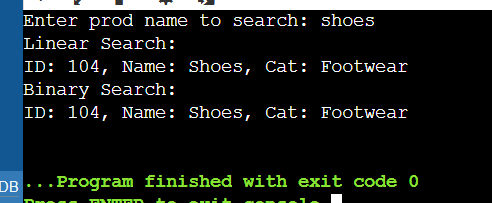
if(res2!=null) res2.show(); else System.out.println("Not found.");

sc.close();

}

}

Output:



**Analysis:**

Time Complexity Comparison:

* **Linear Search**:  
   best - O(1), Average cse is- O(n/2), Worst - O(n)
* **Binary Search**:  
   best - O(1), Average case- O(log n), Worst - O(log n)  
  (Only works on sorted data I assume)

**Which One Is Better and Why?**  
As far as I am concerned binary search is faster and better when data is sorted because it reduces the number of checks using divide and conquer. Linear search is simple but slow for large lists. In e-commerce, where speed matters, binary search is more suitable for product searching.

**Exercise 3: Sorting Customer Orders**

**Bubble Sort:**  
In this algorithm, we compare two elements next to each other and swap them if they are in the wrong order. We keep doing this until the list is sorted. It is easy to write but not efficient for big data.  
**Time Complexity:**

* best: O(n)
* Worst: O(n²)

**Insertion Sort:**  
We take one element from the unsorted part and insert it in the correct place in the sorted part. It is good for small data.  
**Time Complexity:**

* best: O(n)
* Worst: O(n²)

**Quick Sort:**  
This is a divide and conquer algorithm. It picks a pivot and places all smaller elements on one side and bigger ones on the other. It works fast for large data.  
**Time Complexity:**

* best/Average: O(n log n)
* Worst: O(n²)

**Merge Sort:**  
Also divide and conquer. It splits the array into halves, sorts each half, and merges them back. It uses more memory but is very efficient.  
**Time Complexity:** O(n log n)

import java.util.\*;

class Order{

int oid;

String cname;

double price;

Order(int oid, String cname, double price){

this.oid = oid;

this.cname = cname;

this.price = price;

}

void show(){

System.out.println("Order ID: " + oid + ", Name: " + cname + ", Price: ₹" + price);

}

}

class SortAlgo{

public static void bubblesort(Order[] arr){

int n = arr.length;

for(int i=0;i<n-1;i++){

for(int j=0;j<n-i-1;j++){

if(arr[j].price > arr[j+1].price){

Order temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

}

public static void quickSort(Order[] arr, int low, int high){

if(low<high){

int pi = partition(arr, low, high);

quickSort(arr, low, pi-1);

quickSort(arr, pi+1, high);

}

}

private static int partition(Order[] arr, int low, int high){

double pivot = arr[high].price;

int i = low - 1;

for(int j=low;j<high;j++){

if(arr[j].price < 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 class Main{

public static void main(String[] args){

Order[] orders = {

new Order(201,"Ravi",1500.75),

new Order(202,"Neha",950.50),

new Order(203,"Amit",2300.00),

new Order(204,"Sara",1200.00)

};

System.out.println("Before Sorting:");

for(Order o : orders) o.show();

// bubble sort

SortAlgo.bubbleSort(orders);

System.out.println("\nAfter Bubble Sort (Low to High):");

for(Order o : orders) o.show();

orders = new Order[]{

new Order(201,"Ravi",1500.75),

new Order(202,"Neha",950.50),

new Order(203,"Amit",2300.00),

new Order(204,"Sara",1200.00)

};

// quick sort

SortAlgo.quickSort(orders, 0, orders.length-1);

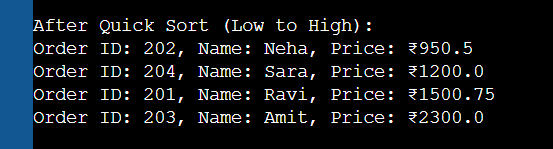
System.out.println("\nAfter Quick Sort (Low to High):");

for(Order o : orders) o.show();

}

}

Output:



Quick sort wokss much faster for large iputs because it reduces the number of comparison by dividng the array. Buble sort is only good for very small arrays and is not used in real projects..

**Exercise 4: Employee Management System**

Arrays are a collection of elements stored at contiguous memory locations. Each element is accessed using an index, which starts from 0.

**Advantages of arrays:**

* Fixed size, so memory is allocated in advance.
* Fast access using index (O(1)).
* Easy to use and manage small datasets

import java.util.Scanner;

class Employee{

int empId;

String name;

String pos;

double sal;

Employee(int empId, String name, String pos, double sal){

this.empId = empId;

this.name = name;

this.pos = pos;

this.sal = sal;

}

void show(){

System.out.println("ID: " + empId + ", Name: " + name + ", Pos: " + pos + ", Salary: ₹" + sal);

}

}

class EmpSystem{

Employee[] arr = new Employee[100];

int count = 0;

void addEmp(Employee e){

if(count < arr.length){

arr[count++] = e;

System.out.println("Employee added.");

}else{

System.out.println("Array is full.");

}

}

void searchEmp(int id){

for(int i=0;i<count;i++){

if(arr[i].empId == id) {

arr[i].show();

return;

}

}

System.out.println("Employee not found.");

}

void deleteEmp(int id){

for(int i=0;i<count;i++){

if(arr[i].empId == id) {

for(int j=i;j<count-1;j++) {

arr[j] = arr[j+1];

}

arr[--count] = null;

System.out.println("Employee deleted.");

return;

}

}

System.out.println("Employee not found.");

}

void listAll() {

if(count == 0){

System.out.println("No employees.");

return;

}

for(int i=0;i<count;i++) {

arr[i].show();

}

}

}

public class Main {

public static void main(String[] args) {

EmpSystem sys = new EmpSystem();

Scanner sc = new Scanner(System.in);

while(true) {

System.out.println("\n1.Add 2.Search 3.Delete 4.Show All 5.Exit");

int ch = sc.nextInt();

if(ch==1){

System.out.print("Enter id, name, position, salary: ");

int id = sc.nextInt();

String name = sc.next();

String pos = sc.next();

double sal = sc.nextDouble();

Employee e = new Employee(id, name, pos, sal);

sys.addEmp(e);

}else if(ch==2){

System.out.print("Enter id to search: ");

int id = sc.nextInt();

sys.searchEmp(id);

}else if(ch==3){

System.out.print("Enter id to delete: ");

int id = sc.nextInt();

sys.deleteEmp(id);

}else if(ch==4){

sys.listAll();

}else if(ch==5){

System.out.println("Exiting...");

break;

}else{

System.out.println("Invalid choice.");

}

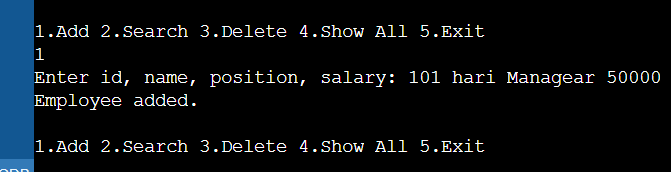
}

sc.close();

}

}

Output:



**Exercise 5: Task Management System**

A linked list is a linear data structure where elements aslo called as nodes are connected using pointers.

**Types:**

* **Singly Linked List**: Each node has data and a pointer to the next node only.
* **Doubly Linked List**: Each node has pointers to both the next and previous nodes.

In this exercise, we use singly linked list as it is simple and uses less memory.

import java.util.Scanner;

class Task{

int id;

String name;

String status;

Task next;

Task(int id, String name, String status){

this.id = id;

this.name = name;

this.status = status;

this.next = null;

}

void show(){

System.out.println("ID: " + id + ", Name: " + name + ", Status: " + status);

}

}

class TaskList{

Task head = null;

void addTask(Task t){

if(head == null){

head = t;

}else{

Task cur = head;

while(cur.next != null){

cur = cur.next;

}

cur.next = t;

}

System.out.println("Task added.");

}

void searchTask(int id){

Task cur = head;

while(cur != null){

if(cur.id == id){

cur.show();

return;

}

cur = cur.next;

}

System.out.println("Task not found.");

}

void deleteTask(int id){

if(head == null){

System.out.println("No tasks.");

return;

}

if(head.id == id){

head = head.next;

System.out.println("Task deleted.");

return;

}

Task cur = head;

while(cur.next != null){

if(cur.next.id == id){

cur.next = cur.next.next;

System.out.println("Task deleted.");

return;

}

cur = cur.next;

}

System.out.println("Task not found.");

}

void showAll(){

if(head == null){

System.out.println("No tasks to show.");

return;

}

Task cur = head;

while(cur != null){

cur.show();

cur = cur.next;

}

}

}

public class Main{

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

TaskList tl = new TaskList();

while(true){

System.out.println("\n1.Add 2.Search 3.Delete 4.Show All 5.Exit");

int ch = sc.nextInt();

if(ch==1){

System.out.print("Enter id, name, status: ");

int id = sc.nextInt();

String name = sc.next();

String stat = sc.next();

Task t = new Task(id, name, stat);

tl.addTask(t);

}else if(ch==2){

System.out.print("Enter id to search: ");

int id = sc.nextInt();

tl.searchTask(id);

}else if(ch==3){

System.out.print("Enter id to delete: ");

int id = sc.nextInt();

tl.deleteTask(id);

}else if(ch==4){

tl.showAll();

}else if(ch==5){

System.out.println("Exiting...");

break;

}else{

System.out.println("Invalid choice.");

}

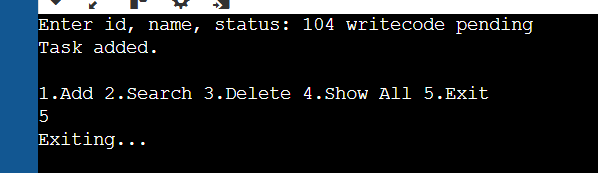
}

sc.close();

}

}

Output:



Advantages of Linked Lists over Arrays

* Linked lists are dynamic in size, so no need to pre-define size.
* Easier to insert or delete in between nodes.
* In arrays, shifting is needed after delete; in linked lists, just pointer update is enough.

**Exercise 6: Library Management System**

import java.util.\*;

class Book{

int id;

String title;

String author;

Book(int id, String title, String author){

this.id = id;

this.title = title;

this.author = author;

}

void show(){

System.out.println("ID: " + id + ", Title: " + title + ", Author: " + author);

}

}

class Search{

public static Book linearSearch(Book[] arr, String key){

for(Book b : arr){

if(b.title.equalsIgnoreCase(key)){

return b;

}

}

return null;

}

public static Book binarySearch(Book[] arr, String key){

int l=0, r=arr.length-1;

while(l<=r){

int mid = (l+r)/2;

int cmp = arr[mid].title.compareToIgnoreCase(key);

if(cmp==0) return arr[mid];

else if(cmp<0) l=mid+1;

else r=mid-1;

}

return null;

}

}

public class Main{

public static void main(String[] args){

Book[] books = {

new Book(1,"Data Structures","Mark Allen"),

new Book(2,"Algorithms","Robert Sedgewick"),

new Book(3,"Clean Code","Robert Martin"),

new Book(4,"Java Basics","James Gosling")

};

Book[] sorted = Arrays.copyOf(books, books.length);

Arrays.sort(sorted, (a,b) -> a.title.compareToIgnoreCase(b.title));

Scanner sc = new Scanner(System.in);

System.out.print("Enter book title to search: ");

String key = sc.nextLine();

System.out.println("\nUsing Linear Search:");

Book res1 = Search.linearSearch(books, key);

if(res1 != null) res1.show();

else System.out.println("Book not found.");

System.out.println("\nUsing Binary Search:");

Book res2 = Search.binarySearch(sorted, key);

if(res2 != null) res2.show();

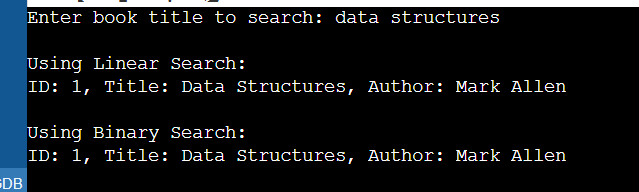
else System.out.println("Book not found.");

sc.close();

}

}

Output:



**Exercise 7: Financial Forecasting**

Recursion is a programming method where a function calls itself to solve smaller parts of a problem.It helps to break down big problems into easier sub-problems. It's often used in problems like factorials, Fibonacci series, or any repetitive calculation that follows a pattern.

import java.util.Scanner;

class Forecast{

double predict(double val, double rate, int years){

if(years == 0){

return val;

}

return predict(val \* (1 + rate), rate, years - 1);

}

}

public class Main{

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

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

double v = sc.nextDouble();

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

double r = sc.nextDouble() / 100;

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

int y = sc.nextInt();

Forecast f = new Forecast();

double res = f.predict(v, r, y);

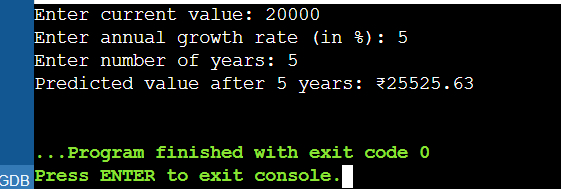
System.out.printf("Predicted value after %d years: ₹%.2f\n", y, res);

sc.close();

}

}

Output:



* This recursive function has **O(n)** time complexity, where n is the number of years.
* It makes one call per year until years == 0.