**WEEK-1**

**DESIGN PATTERNS AND PRINCIPLES**

HANDS ON EXERCISE

**1.Implementing the Singleton Pattern**

Scenario:

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

Steps:

1. Create a New Java Project:

o Create a new Java project named SingletonPatternExample.

2. Define a Singleton Class:

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

o Ensure the constructor of Logger is private.

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

3. Implement the Singleton Pattern:

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

4. Test the Singleton Implementation:

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

**CODE:**

class Logger {

private static Logger instance;

private Logger() {}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

Logger1.log(“Hello”);

System.out.println(logger1 == logger2); // true

}

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**2. Implementing the Factory Method Pattern**

Scenario:

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

Steps:

1. Create a New Java Project:

o Create a new Java project named FactoryMethodPatternExample.

2. Define Document Classes:

o Create interfaces or abstract classes for different document types such as WordDocument, PdfDocument, and ExcelDocument.

3. Create Concrete Document Classes:

o Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.

4. Implement the Factory Method:

o Create an abstract class DocumentFactory with a method createDocument().

o Create concrete factory classes for each document type that extends DocumentFactory and implements the createDocument() method.

5. Test the Factory Method Implementation:

o Create a test class to demonstrate the creation of different document types using the factory method.

**CODE:**

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");

}

}

class ExcelDocument implements Document {

public void open() {

System.out.println("Opening Excel 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 ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public static void main(String[] args) {

DocumentFactory factory = new PdfFactory();

Document doc = factory.createDocument();

DocumentFactory fac=new ExcelFactory();

Document doc1 = fac.createDocument();

doc.open();

System.out.println("Opening next file");

doc1.open();

System.out.println("Opening next file");

DocumentFactory fact=new WordFactory();

Document doc2 = fact.createDocument();

doc2.open();

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**3.Implementing the Builder Pattern**

**Scenario:**

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

**Steps:**

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

**CODE:**

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 spwecs() {

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

}

public static void main(String[] args) {

Computer pc = new Computer.Builder()

.setCPU("Intel i7").setRAM("16GB").setStorage("512GB SSD").build();

pc.specs();

Computer pc1 = new Computer.Builder()

.setCPU("Intel i5").setRAM("32GB").setStorage("1TB SSD").build();

pc1.specs();

}

}

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**4.Implementing the Adapter Pattern**

**Scenario:**

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

**Steps:**

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

**CODE:**

interface PaymentProcessor {

void processPayment(double amount);

class OldGateway {

public void makePayment(double amount) {

System.out.println("Paid " + amount + " via Old Gateway");

}

public void receivePayment(double amount) {

System.out.println("Received " + amount + " via Old Gateway");

}

}

class OldGatewayAdapter implements PaymentProcessor {

private OldGateway gateway = new OldGateway();

public void processPayment(double amount) {

gateway.makePayment(amount);

}

}

class RazorPay {

public void makePayment(double amount) {

System.out.println("Paid " + amount + " via RazorPay");

}

public void receivePayment(double amount) {

System.out.println("Received " + amount + " via RazorPay");

}

}

class RazorPayAdapter implements PaymentProcessor {

private RazorPay gateway1 = new RazorPay();

public void processPayment(double amount) {

gateway1.receivePayment(amount);

}

}

public static void main(String[] args) {

System.out.println("Paying Money via Old Gateway");

PaymentProcessor processor = new OldGatewayAdapter();

processor.processPayment(1000);

System.out.println("Receiving money via RazorPay");

PaymentProcessor processor1 = new RazorPayAdapter();

processor1.processPayment(1000);

}

}

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**DATA STRUCTURE AND ALOGRITHMS**

**Exercise** **1: Inventory Management System**

Scenario:

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

Steps:

1. Understand the Problem:
   * Explain why data structures and algorithms are essential in handling large inventories.
   * Discuss the types of data structures suitable for this problem.
2. Setup:
   * Create a new project for the inventory management system.
3. Implementation:
   * Define a class Product with attributes like productId, productName, quantity, and price.
   * Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   * Implement methods to add, update, and delete products from the inventory.
4. Analysis:
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   * Discuss how you can optimize these operations.

For this problem ArrayList and HashMap are suitable.

**CODE:**

import java.util.HashMap;

import java.util.Scanner;

class Product {

int productId;

String productName;

int quantity;

double price;

public Product(int productId, String productName, int quantity, double price) {

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public String toString() {

return "Product ID: " + productId + ", Name: " + productName +

", Quantity: " + quantity + ", Price: $" + price;

}

}

public class InventoryManagementSystem {

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

public void addProduct(Product p) {

inventory.put(p.productId, p);

System.out.println("Product added successfully.");

}

public void updateProduct(int id, Product updated) {

if (inventory.containsKey(id)) {

inventory.put(id, updated);

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

} else {

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

}

}

public void deleteProduct(int id) {

if (inventory.containsKey(id)) {

inventory.remove(id);

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

} else {

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

}

}

public void displayInventory() {

if (inventory.isEmpty()) {

System.out.println("Inventory is empty.");

return;

}

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

System.out.println(p);

}

}

public static void main(String[] args) {

InventoryManagementSystem ims = new InventoryManagementSystem();

Scanner scanner = new Scanner(System.in);

int choice;

do {

System.out.println("\n1. Add Product");

System.out.println("2. Update Product");

System.out.println("3. Delete Product");

System.out.println("4. Display Inventory");

System.out.println("5. Exit");

System.out.print("Enter your choice: ");

choice = scanner.nextInt();

int id, qty;

double price;

String name;

switch (choice) {

case 1:

System.out.print("Enter ID: ");

id = scanner.nextInt();

scanner.nextLine();

System.out.print("Enter Name: ");

name = scanner.nextLine();

System.out.print("Enter Quantity: ");

qty = scanner.nextInt();

System.out.print("Enter Price: ");

price = scanner.nextDouble();

ims.addProduct(new Product(id, name, qty, price));

break;

case 2:

System.out.print("Enter ID to update: ");

id = scanner.nextInt();

scanner.nextLine();

System.out.print("Enter New Name: ");

name = scanner.nextLine();

System.out.print("Enter New Quantity: ");

qty = scanner.nextInt();

System.out.print("Enter New Price: ");

price = scanner.nextDouble();

ims.updateProduct(id, new Product(id, name, qty, price));

break;

case 3:

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

id = scanner.nextInt();

ims.deleteProduct(id);

break;

case 4:

ims.displayInventory();

break;

case 5:

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

break;

default:

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

}

} while (choice != 5);

scanner.close();

}

}

| **Operation** | **Time Complexity (Average)** | **Worst Case** |
| --- | --- | --- |
| Add Product | O(1) | O(n) (rare) |
| Update Product | O(1) | O(n) (rare) |
| Delete Product | O(1) | O(n) (rare) |
| Search Product | O(1) | O(n) (rare) |

In HashMap, we search for the products using a unique attribute which is productid. By using the productid it is easy to do operations (add, delete, update) in HashMap.

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**2.E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

What is Big O Notation?

Big O notation describes the **upper bound** of an algorithm’s time or space complexity in terms of input size n. It tells you how your algorithm performs as the input grows large.

**Time Complexity of Searching Algorithms**

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) | O(n/2) = O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

Where n=number of inputs

**Linear Search** scans elements one by one — best when data is unsorted.

**Binary Search** repeatedly divides sorted data — best for sorted large lists.

**CODE:**

import java.util.Arrays;

import java.util.Comparator;

class Product {

    int productId;

    String productName;

    String category;

    public Product(int productId, String productName, String category) {

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

    public String toString() {

        return productId + ": " + productName + " (" + category + ")";

    }

    public static int linearSearch(Product[] products, String targetName) {

        for (int i = 0; i < products.length; i++) {

            if (products[i].productName.equalsIgnoreCase(targetName)) {

                return i;

            }

        }

        return -1;

    }

    public static int binarySearch(Product[] products, String targetName) {

        int left = 0, right = products.length - 1;

        while (left <= right) {

            int mid = (left + right) / 2;

            int cmp = products[mid].productName.compareToIgnoreCase(targetName);

            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(101, "Laptop", "Electronics"),

            new Product(102, "Camera", "Electronics"),

            new Product(103, "Shoes", "Footwear"),

            new Product(104, "Watch", "Accessories"),

            new Product(105, "Phone", "Electronics")

        };

        // For binary search, sort the array by productName

        Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

        // Display sorted products

        System.out.println("Sorted Product List:");

        for (Product p : products) System.out.println(p);

        // Test searches

        String searchName = "Phone";

        int linearResult = linearSearch(products, searchName);

        int binaryResult = binarySearch(products, searchName);

        System.out.println("\nLinear Search Result:");

        System.out.println(linearResult != -1 ? products[linearResult] : "Not found");

        System.out.println("\nBinary Search Result:");

        System.out.println(binaryResult != -1 ? products[binaryResult] : "Not found");

    }

}

We use Binary Search Algorithm when there is a large dataset. For the E-Commerce Website the Binary Search Algorithm is suitable for searching the product .

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**3.Sorting Customer Orders**

**Scenario:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**Steps:**

1. **Understand Sorting Algorithms:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.

**CODE:**

class Order {

int orderId;

String customerName;

double totalPrice;

public Order(int orderId, String customerName, double totalPrice) {

this.orderId = orderId;

this.customerName = customerName;

this.totalPrice = totalPrice;

}

public String toString() {

return orderId + ": " + customerName + " - $" + totalPrice;

}

}

public class OrderSorting {

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[] orders, int low, int high) {

double pivot = orders[high].totalPrice;

int i = (low - 1);

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

if (orders[j].totalPrice < pivot) {

i++;

Order temp = orders[i];

orders[i] = orders[j];

orders[j] = temp;

}

}

Order temp = orders[i + 1];

orders[i + 1] = orders[high];

orders[high] = temp;

return i + 1;

}

public static void printOrders(Order[] orders) {

for (Order o : orders) System.out.println(o);

}

public static void main(String[] args) {

Order[] orders = {

new Order(1, "Alice", 1500.0),

new Order(2, "Bob", 2200.0),

new Order(3, "Charlie", 800.0)

};

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

printOrders(orders);

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

System.out.println("\nAfter QuickSort:");

printOrders(orders);

}

}

**Time Complexity Comparison:**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Bubble Sort** | O(n) | O(n²) | O(n²) |
| **Quick Sort** | O(n log n) | O(n log n) | O(n²) |

The quick Sort Algorithm is suitable because it is follows Divide And Conquer Approach and uses recursion and partitioning approach to reduce comparison count.

**OUTPUT:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**5.Task Management System**

Scenario:

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

Steps:

1. Understand Linked Lists:
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. Setup:
   * Create a class Task with attributes like taskId, taskName, and status.
3. Implementation:
   * Implement a singly linked list to manage tasks.
   * Implement methods to add, search, traverse, and delete tasks in the linked list.
4. Analysis:
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

**CODE:**

Types of Linked Lists

1. Singly Linked List:
   * Each node points to the next node.
   * Memory-efficient.
   * Only forward traversal is possible.
2. Doubly Linked List:
   * Each node points to both previous and next.
   * Allows forward and backward traversal.
   * Requires more memory per node.

class Task {

    int taskId;

    String taskName;

    String status;

    Task next;

    public Task(int taskId, String taskName, String status) {

        this.taskId = taskId;

        this.taskName = taskName;

        this.status = status;

        this.next = null;

    }

    public String toString() {

        return taskId + ": " + taskName + " [" + status + "]";

    }

}

public class TaskManagementSystem {

    Task head;

    public void addTask(int id, String name, String status) {

        Task newTask = new Task(id, name, status);

        if (head == null) {

            head = newTask;

        } else {

            Task temp = head;

            while (temp.next != null) temp = temp.next;

            temp.next = newTask;

        }

    }

    public Task searchTask(int id) {

        Task temp = head;

        while (temp != null) {

            if (temp.taskId == id) return temp;

            temp = temp.next;

        }

        return null;

    }

    public void deleteTask(int id) {

        if (head == null) return;

        if (head.taskId == id) {

            head = head.next;

            return;

        }

        Task temp = head;

        while (temp.next != null && temp.next.taskId != id) temp = temp.next;

        if (temp.next != null) temp.next = temp.next.next;

    }

    public void traverseTasks() {

        Task temp = head;

        while (temp != null) {

            System.out.println(temp);

            temp = temp.next;

        }

    }

    public static void main(String[] args) {

        TaskManagementSystem tms = new TaskManagementSystem();

        tms.addTask(1, "Design UI", "Pending");

        tms.addTask(2, "Develop Backend", "In Progress");

        tms.traverseTasks();

        tms.deleteTask(1);

        tms.traverseTasks();

    }

}

**OUTPUT:**

Analysis of Time Complexity

| **Operation** | **Time Complexity** |
| --- | --- |
| Add | O(n) |
| Search | O(n) |
| Delete | O(n) |
| Traverse | O(n) |

**ADVANTAGES OF LINKEDLIST OVER ARRAYS**

* Arrays are of Fixed size where as LinkedList are of Dynamic Size
* LinkedList allocates memory dynamically where as Arrays allocates memory statically

A screenshot of a computer program

AI-generated content may be incorrect.

**7.Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**CODE:**

**What is Recursion?**

* Recursion is a programming technique where a method calls itself to solve smaller instances of a problem.
* It's particularly useful for problems that have repetitive sub-problems, such as calculations over time or divide-and-conquer algorithms.

Example: Financial forecasting is recursive because each year builds upon the previous year's value.

import java.util.Scanner;

public class FinancialForecasting {

// Recursive method to forecast future value

public static double forecastRecursive(double currentValue, double growthRate, int years) {

if (years == 0) return currentValue;

return forecastRecursive(currentValue \* (1 + growthRate), growthRate, years - 1);

}

// Optimized version using iteration (to avoid deep recursion)

public static double forecastIterative(double currentValue, double growthRate, int years) {

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

currentValue \*= (1 + growthRate);

}

return currentValue;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

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

double currentValue = sc.nextDouble();

System.out.print("Enter annual growth rate (e.g., 0.10 for 10%): ");

double growthRate = sc.nextDouble();

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

int years = sc.nextInt();

double futureRecursive = forecastRecursive(currentValue, growthRate, years);

double futureIterative = forecastIterative(currentValue, growthRate, years);

System.out.printf("\n Recursive Forecast after %d years: %.2f", years, futureRecursive);

System.out.printf("\n Iterative Forecast after %d years: %.2f\n", years, futureIterative);

sc.close();

}

}

**OUTPUT:**

The Time Complexity of this recursive algorithm is O(n).

To **optimize a recursive solution and avoid excessive computation**, especially in problems we can replace recursion with **iteration** whenever possible to avoid deep call stacks. For numeric predictions, instead of recursion by using a **mathematical formula** we can avoid excessive computations.

A screenshot of a computer program

AI-generated content may be incorrect.