**1. Write a function to remove duplicates from a List while preserving the order of elements.**

**Hint: Use a LinkedHashSet which maintains the insertion order and automatically handles duplicates.**

**Pseudocode:**

BEGIN

DEFINE function removeDuplicates(list):

// Create a LinkedHashSet to store elements and automatically handle duplicates

INITIALIZE linkedHashSet with new LinkedHashSet()

// Iterate through the elements in the list

FOR each element in list:

ADD element to linkedHashSet (duplicates will be ignored)

// Convert the LinkedHashSet back to a List

INITIALIZE resultList with new ArrayList(linkedHashSet)

RETURN resultList

FUNCTION main:

// Define an example list with duplicates

INITIALIZE list with [1, 2, 3, 2, 4, 5, 3, 6, 4, 7]

// Call the removeDuplicates function

INITIALIZE uniqueList with removeDuplicates(list)

// PRINT the unique list with duplicates removed

PRINT uniqueList

END

**Java Code:**

import java.util.ArrayList;

import java.util.LinkedHashSet;

import java.util.List;

public class RemoveDuplicates {

// Function to remove duplicates from a List while preserving order

public static List<Integer> removeDuplicates(List<Integer> list) {

// Create a LinkedHashSet to preserve order and handle duplicates

LinkedHashSet<Integer> linkedHashSet = new LinkedHashSet<>(list);

// Convert the LinkedHashSet back to a List

List<Integer> resultList = new ArrayList<>(linkedHashSet);

return resultList;

}

public static void main(String[] args) {

// Define an example list with duplicates

List<Integer> list = new ArrayList<>();

list.add(1);

list.add(2);

list.add(3);

list.add(2);

list.add(4);

list.add(5);

list.add(3);

list.add(6);

list.add(4);

list.add(7);

// Call the removeDuplicates function

List<Integer> uniqueList = removeDuplicates(list);

// Print the unique list with duplicates removed

System.out.println("List after removing duplicates: " + uniqueList);

}

}

**2. Write a function to rotate a list to the right by k places. Hint: To rotate a list, we can divide the list into two parts and then swap their positions**

Pseudocode:

BEGIN

DEFINE function rotateList(list, k):

// Get the size of the list

INITIALIZE n as size of list

// If the list is empty or k is zero, no rotation needed

IF n is 0 OR k is 0:

RETURN list

// Adjust k if it's greater than the size of the list

SET k to k % n

// Split the list into two parts

INITIALIZE part1 as sublist from index n-k to n (last k elements)

INITIALIZE part2 as sublist from index 0 to n-k (first n-k elements)

// Concatenate part1 and part2

INITIALIZE rotatedList as part1 + part2

RETURN rotatedList

FUNCTION main:

// Define an example list

INITIALIZE list with [1, 2, 3, 4, 5, 6, 7]

// Set rotation steps

INITIALIZE k as 3

// Call the rotateList function

INITIALIZE rotatedList with rotateList(list, k)

// PRINT the rotated list

PRINT rotatedList

END

**Java Code:**

import java.util.ArrayList;

import java.util.List;

public class RotateList {

// Function to rotate the list to the right by k places

public static List<Integer> rotateList(List<Integer> list, int k) {

int n = list.size();

// If the list is empty or k is 0, return the original list

if (n == 0 || k == 0) {

return list;

}

// Adjust k if it is greater than the size of the list

k = k % n;

// Split the list into two parts

List<Integer> part1 = new ArrayList<>(list.subList(n - k, n)); // Last k elements

List<Integer> part2 = new ArrayList<>(list.subList(0, n - k)); // First n-k elements

// Concatenate part1 and part2 to form the rotated list

part1.addAll(part2);

return part1;

}

public static void main(String[] args) {

// Define an example list

List<Integer> list = new ArrayList<>();

for (int i = 1; i <= 7; i++) {

list.add(i);

}

// Define rotation steps

int k = 3;

// Call the rotateList function

List<Integer> rotatedList = rotateList(list, k);

// Print the rotated list

System.out.println("Rotated list by " + k + " places: " + rotatedList);

}

}

**3. Write a function to group anagrams from a list of strings. Hint: Use a HashMap where the key is a sorted string and the value is a list of anagrams.**

**Pseudocode:**

BEGIN

DEFINE function groupAnagrams(wordsList):

// Initialize an empty HashMap

INITIALIZE map as empty HashMap

FOR each word in wordsList:

// Convert word to character array, sort it, and convert it back to string

INITIALIZE sortedWord as sorted characters of word

// If the sorted word is not already a key in map, add a new list for it

IF sortedWord not in map:

CREATE a new empty list for map[sortedWord]

// Add the current word to the list of anagrams

ADD word to map[sortedWord]

// Return the values (list of anagrams) from the map

RETURN all values of map as list of lists

FUNCTION main:

// Example input list

INITIALIZE wordsList with ["eat", "tea", "tan", "ate", "nat", "bat"]

// Call groupAnagrams function

INITIALIZE groupedAnagrams as groupAnagrams(wordsList)

// PRINT the result

PRINT groupedAnagrams

END

**Java Code:**

import java.util.\*;

public class GroupAnagrams {

// Function to group anagrams

public static List<List<String>> groupAnagrams(String[] wordsList) {

// HashMap to store the sorted string as key and list of anagrams as value

Map<String, List<String>> map = new HashMap<>();

// Loop through each word in the input list

for (String word : wordsList) {

// Convert the word into a char array and sort it

char[] chars = word.toCharArray();

Arrays.sort(chars);

// Convert the sorted char array back to a string

String sortedWord = new String(chars);

// If the sorted word is not in the map, create a new list for it

if (!map.containsKey(sortedWord)) {

map.put(sortedWord, new ArrayList<>());

}

// Add the original word to the corresponding anagram list

map.get(sortedWord).add(word);

}

// Return the list of all anagram groups

return new ArrayList<>(map.values());

}

public static void main(String[] args) {

// Example input list

String[] wordsList = {"eat", "tea", "tan", "ate", "nat", "bat"};

// Call groupAnagrams function

List<List<String>> groupedAnagrams = groupAnagrams(wordsList);

// Print the result

System.out.println("Grouped Anagrams: " + groupedAnagrams);

}

}

**4. Given a list containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the list. Hint: Use the sum formula for the first n natural numbers to find the missing number.**

**Pseudocode:**

BEGIN

DEFINE function findMissingNumber(numbersList, n):

// Calculate the expected sum of numbers from 0 to n using the sum formula

INITIALIZE expectedSum as (n \* (n + 1)) / 2

// Calculate the actual sum of numbers in numbersList

INITIALIZE actualSum as 0

FOR each number in numbersList:

ADD number to actualSum

// The missing number is the difference between expectedSum and actualSum

RETURN expectedSum - actualSum

FUNCTION main:

INITIALIZE numbersList with distinct numbers from 0 to n but missing one number

INITIALIZE n as the length of the numbersList plus 1

// Call findMissingNumber function

INITIALIZE missingNumber as findMissingNumber(numbersList, n)

// PRINT the missing number

PRINT "The missing number is: " + missingNumber

END

**Java code:**

public class MissingNumber {

// Function to find the missing number

public static int findMissingNumber(int[] numbersList, int n) {

// Calculate the expected sum of numbers from 0 to n using the formula

int expectedSum = (n \* (n + 1)) / 2;

// Calculate the actual sum of the numbers in the list

int actualSum = 0;

for (int number : numbersList) {

actualSum += number;

}

// The missing number is the difference between expectedSum and actualSum

return expectedSum - actualSum;

}

public static void main(String[] args) {

// Example input: array of distinct numbers from 0 to n, missing one number

int[] numbersList = {0, 1, 2, 4, 5}; // Missing number is 3

int n = numbersList.length; // n should be the length of the array + 1

// Call the function to find the missing number

int missingNumber = findMissingNumber(numbersList, n);

// Print the result

System.out.println("The missing number is: " + missingNumber);

}

}

**5. Create an abstract class with 2 abstract methods(total() and average()) and 3 concrete methods(mean(), mode(), median()). Now extend the abstract class from a concrete class and use all the methods in that abstract class**

**Pseudocode:**

BEGIN

DEFINE Abstract Class `Statistics`

// Abstract method to calculate total

FUNCTION total() RETURN int

// Abstract method to calculate average

FUNCTION average() RETURN double

// Concrete method to calculate mean

FUNCTION mean(data[]) RETURN double

// Implementation for mean

// Concrete method to calculate mode

FUNCTION mode(data[]) RETURN int

// Implementation for mode

// Concrete method to calculate median

FUNCTION median(data[]) RETURN double

// Implementation for median

DEFINE Concrete Class `DataStatistics` EXTENDS Statistics

// Implement abstract method total

FUNCTION total() RETURN int

// Implementation of total calculation

// Implement abstract method average

FUNCTION average() RETURN double

// Implementation of average calculation

// Main program execution

CREATE an instance of DataStatistics

CALL total() method

CALL average() method

CALL mean() method with sample data

CALL mode() method with sample data

CALL median() method with sample data

END

Java Code:

// Define the abstract class Statistics

abstract class Statistics {

// Abstract method to calculate total

public abstract int total();

// Abstract method to calculate average

public abstract double average();

// Concrete method to calculate mean

public double mean(int[] data) {

if (data.length == 0) return 0;

double sum = 0;

for (int num : data) {

sum += num;

}

return sum / data.length;

}

// Concrete method to calculate mode

public int mode(int[] data) {

if (data.length == 0) return -1;

java.util.HashMap<Integer, Integer> freqMap = new java.util.HashMap<>();

int mode = data[0];

int maxCount = 0;

for (int num : data) {

int count = freqMap.getOrDefault(num, 0) + 1;

freqMap.put(num, count);

if (count > maxCount) {

maxCount = count;

mode = num;

}

}

return mode;

}

// Concrete method to calculate median

public double median(int[] data) {

if (data.length == 0) return 0;

java.util.Arrays.sort(data);

int middle = data.length / 2;

if (data.length % 2 == 0) {

return (data[middle - 1] + data[middle]) / 2.0;

} else {

return data[middle];

}

}

}

// Define the concrete class DataStatistics extending Statistics

class DataStatistics extends Statistics {

// Implement the abstract method total

@Override

public int total() {

// Example implementation

int[] data = {1, 2, 3, 4, 5}; // Sample data

int sum = 0;

for (int num : data) {

sum += num;

}

return sum;

}

// Implement the abstract method average

@Override

public double average() {

// Example implementation

int[] data = {1, 2, 3, 4, 5}; // Sample data

return mean(data);

}

// Additional methods can be overridden or used directly

}

// Main class to test the methods

public class Main {

public static void main(String[] args) {

// Create an instance of DataStatistics

DataStatistics stats = new DataStatistics();

// Call the abstract methods implemented in DataStatistics

System.out.println("Total: " + stats.total());

System.out.println("Average: " + stats.average());

// Sample data for mean, mode, and median

int[] data = {1, 2, 2, 3, 4, 5};

// Call the concrete methods

System.out.println("Mean: " + stats.mean(data));

System.out.println("Mode: " + stats.mode(data));

System.out.println("Median: " + stats.median(data));

}

}

**6. Create an interface with 4 methods called add(), sub(), mul() and div(). Then give implementation for all in the implementing class**

Pseudocode:

BEGIN

DEFINE Interface `Calculator`

// Method to add two numbers

FUNCTION add(int a, int b) RETURN int

// Method to subtract two numbers

FUNCTION sub(int a, int b) RETURN int

// Method to multiply two numbers

FUNCTION mul(int a, int b) RETURN int

// Method to divide two numbers

FUNCTION div(int a, int b) RETURN double

DEFINE Class `BasicCalculator` IMPLEMENTS Calculator

// Implement add method

FUNCTION add(int a, int b) RETURN int

RETURN a + b

// Implement sub method

FUNCTION sub(int a, int b) RETURN int

RETURN a - b

// Implement mul method

FUNCTION mul(int a, int b) RETURN int

RETURN a \* b

// Implement div method

FUNCTION div(int a, int b) RETURN double

IF b == 0

THROW ArithmeticException("Division by zero")

RETURN (double) a / b

// Main program execution

CREATE an instance of BasicCalculator

CALL add() method with sample inputs

CALL sub() method with sample inputs

CALL mul() method with sample inputs

CALL div() method with sample inputs

END

Java Code:

// Define the Calculator interface

interface Calculator {

// Method to add two numbers

int add(int a, int b);

// Method to subtract two numbers

int sub(int a, int b);

// Method to multiply two numbers

int mul(int a, int b);

// Method to divide two numbers

double div(int a, int b);

}

// Implement the Calculator interface in BasicCalculator class

class BasicCalculator implements Calculator {

**7. Create 3 interfaces with 1 method each sum(), avg(), percentage() respectively. Now implement all the 3 interfaces in your class**

**Pseudocode:**

BEGIN

BEGIN

DEFINE Interface `SumInterface`

// Method to calculate the sum

FUNCTION sum(int a, int b) RETURN int

DEFINE Interface `AvgInterface`

// Method to calculate the average

FUNCTION avg(int a, int b) RETURN double

DEFINE Interface `PercentageInterface`

// Method to calculate the percentage

FUNCTION percentage(int total, int part) RETURN double

DEFINE Class `Calculator` IMPLEMENTS SumInterface, AvgInterface, PercentageInterface

// Implement sum method

FUNCTION sum(int a, int b) RETURN int

RETURN a + b

// Implement avg method

FUNCTION avg(int a, int b) RETURN double

RETURN (a + b) / 2.0

// Implement percentage method

FUNCTION percentage(int total, int part) RETURN double

IF total == 0

RETURN 0

RETURN (part / (double) total) \* 100

// Main program execution

CREATE an instance of Calculator

CALL sum() method with sample inputs

CALL avg() method with sample inputs

CALL percentage() method with sample inputs

END

Java Code:

// Define the SumInterface interface

interface SumInterface {

int sum(int a, int b);

}

// Define the AvgInterface interface

interface AvgInterface {

double avg(int a, int b);

}

// Define the PercentageInterface interface

interface PercentageInterface {

double percentage(int total, int part);

}

// Implement all three interfaces in the Calculator class

class Calculator implements SumInterface, AvgInterface, PercentageInterface {

// Implement sum method

@Override

public int sum(int a, int b) {

return a + b;

}

// Implement avg method

@Override

public double avg(int a, int b) {

return (a + b) / 2.0;

}

// Implement percentage method

@Override

public double percentage(int total, int part) {

if (total == 0) {

return 0;

}

return (part / (double) total) \* 100;

}

}

// Main class to test the implemented methods

public class Main {

public static void main(String[] args) {

// Create an instance of Calculator

Calculator calc = new Calculator();

// Test sum method

System.out.println("Sum: 10 + 5 = " + calc.sum(10, 5));

// Test avg method

System.out.println("Average: (10 + 5) / 2 = " + calc.avg(10, 5));

// Test percentage method

System.out.println("Percentage: 5 out of 20 = " + calc.percentage(20, 5) + "%");

}

}

**8. Create an interface called Tree and extend 2 classes from it called Branch1 and Branch2. Tree should contain methods fruits(), leaves() and flowers(), these methods contain 2,3,4 parameters respectively.**

Pseudocode:

BEGIN

DEFINE Interface `Tree`

// Method to handle fruits with 2 parameters

FUNCTION fruits(String fruitType, int quantity)

// Method to handle leaves with 3 parameters

FUNCTION leaves(String leafType, int length, int width)

// Method to handle flowers with 4 parameters

FUNCTION flowers(String flowerType, int petalCount, String color, boolean hasFragrance)

DEFINE Class `Branch1` IMPLEMENTS Tree

// Implement fruits method

FUNCTION fruits(String fruitType, int quantity)

PRINT "Branch1: Fruits - Type: " + fruitType + ", Quantity: " + quantity

// Implement leaves method

FUNCTION leaves(String leafType, int length, int width)

PRINT "Branch1: Leaves - Type: " + leafType + ", Length: " + length + ", Width: " + width

// Implement flowers method

FUNCTION flowers(String flowerType, int petalCount, String color, boolean hasFragrance)

PRINT "Branch1: Flowers - Type: " + flowerType + ", Petal Count: " + petalCount + ", Color: " + color + ", Fragrance: " + hasFragrance

DEFINE Class `Branch2` IMPLEMENTS Tree

// Implement fruits method

FUNCTION fruits(String fruitType, int quantity)

PRINT "Branch2: Fruits - Type: " + fruitType + ", Quantity: " + quantity

// Implement leaves method

FUNCTION leaves(String leafType, int length, int width)

PRINT "Branch2: Leaves - Type: " + leafType + ", Length: " + length + ", Width: " + width

// Implement flowers method

FUNCTION flowers(String flowerType, int petalCount, String color, boolean hasFragrance)

PRINT "Branch2: Flowers - Type: " + flowerType + ", Petal Count: " + petalCount + ", Color: " + color + ", Fragrance: " + hasFragrance

// Main program execution

CREATE instances of Branch1 and Branch2

CALL fruits(), leaves(), and flowers() methods on both instances

END

Java Code:

// Define the Tree interface

interface Tree {

// Method to handle fruits with 2 parameters

void fruits(String fruitType, int quantity);

// Method to handle leaves with 3 parameters

void leaves(String leafType, int length, int width);

// Method to handle flowers with 4 parameters

void flowers(String flowerType, int petalCount, String color, boolean hasFragrance);

}

// Implement the Tree interface in Branch1

class Branch1 implements Tree {

// Implement fruits method

@Override

public void fruits(String fruitType, int quantity) {

System.out.println("Branch1: Fruits - Type: " + fruitType + ", Quantity: " + quantity);

}

// Implement leaves method

@Override

public void leaves(String leafType, int length, int width) {

System.out.println("Branch1: Leaves - Type: " + leafType + ", Length: " + length + ", Width: " + width);

}

// Implement flowers method

@Override

public void flowers(String flowerType, int petalCount, String color, boolean hasFragrance) {

System.out.println("Branch1: Flowers - Type: " + flowerType + ", Petal Count: " + petalCount + ", Color: " + color + ", Fragrance: " + hasFragrance);

}

}

// Implement the Tree interface in Branch2

class Branch2 implements Tree {

// Implement fruits method

@Override

public void fruits(String fruitType, int quantity) {

System.out.println("Branch2: Fruits - Type: " + fruitType + ", Quantity: " + quantity);

}

// Implement leaves method

@Override

public void leaves(String leafType, int length, int width) {

System.out.println("Branch2: Leaves - Type: " + leafType + ", Length: " + length + ", Width: " + width);

}

// Implement flowers method

@Override

public void flowers(String flowerType, int petalCount, String color, boolean hasFragrance) {

System.out.println("Branch2: Flowers - Type: " + flowerType + ", Petal Count: " + petalCount + ", Color: " + color + ", Fragrance: " + hasFragrance);

}

}

// Main class to test the implementations

public class Main {

public static void main(String[] args) {

// Create instances of Branch1 and Branch2

Tree branch1 = new Branch1();

Tree branch2 = new Branch2();

// Test methods for Branch1

branch1.fruits("Apple", 10);

branch1.leaves("Maple", 5, 3);

branch1.flowers("Rose", 5, "Red", true);

// Test methods for Branch2

branch2.fruits("Orange", 20);

branch2.leaves("Oak", 6, 4);

branch2.flowers("Tulip", 6, "Yellow", false);

}

}

**9. Create a Set and find the following**

1. **Find Union of 2 sets.**

**Hint: Use the addAll method to combine elements from both sets**

1. **Find Intersection of 2 sets.**

**Hint: Use the retainAll method to combine elements from both sets**

1. **Find Difference of 2 sets.**

**Hint: Use the removeAll method to combine elements from both sets**

1. Pseudocode:

BEGIN

FUNCTION union(set1, set2):

CREATE resultSet as a new Set

ADD all elements of set1 to resultSet

ADD all elements of set2 to resultSet using addAll method

RETURN resultSet

FUNCTION main:

INITIALIZE set1 with some elements

INITIALIZE set2 with some elements

INITIALIZE unionSet by calling union(set1, set2)

PRINT unionSet

END

Java Code:

import java.util.HashSet;

import java.util.Set;

public class SetUnion {

// Union of two sets

public static Set<Integer> union(Set<Integer> set1, Set<Integer> set2) {

Set<Integer> resultSet = new HashSet<>(set1);

resultSet.addAll(set2); // Use addAll to combine the sets

return resultSet;

}

public static void main(String[] args) {

// Example Sets

Set<Integer> set1 = new HashSet<>();

set1.add(1);

set1.add(2);

set1.add(3);

set1.add(4);

Set<Integer> set2 = new HashSet<>();

set2.add(3);

set2.add(4);

set2.add(5);

set2.add(6);

// Union

Set<Integer> unionSet = union(set1, set2);

System.out.println("Union of set1 and set2: " + unionSet);

}

}

1. Pseudo code:

BEGIN

FUNCTION intersection(set1, set2):

CREATE resultSet as a new Set initialized with set1

RETAIN only the elements that are also in set2 using retainAll method

RETURN resultSet

FUNCTION main:

INITIALIZE set1 with some elements

INITIALIZE set2 with some elements

INITIALIZE intersectionSet by calling intersection(set1, set2)

PRINT intersectionSet

END

Java code:

import java.util.HashSet;

import java.util.Set;

public class SetIntersection {

// Intersection of two sets

public static Set<Integer> intersection(Set<Integer> set1, Set<Integer> set2) {

Set<Integer> resultSet = new HashSet<>(set1);

resultSet.retainAll(set2); // Use retainAll to keep only common elements

return resultSet;

}

public static void main(String[] args) {

// Example Sets

Set<Integer> set1 = new HashSet<>();

set1.add(1);

set1.add(2);

set1.add(3);

set1.add(4);

Set<Integer> set2 = new HashSet<>();

set2.add(3);

set2.add(4);

set2.add(5);

set2.add(6);

// Intersection

Set<Integer> intersectionSet = intersection(set1, set2);

System.out.println("Intersection of set1 and set2: " + intersectionSet);

}

}

C)Pseudo code:

BEGIN

FUNCTION difference(set1, set2):

CREATE resultSet as a new Set initialized with set1

REMOVE all elements that are in set2 from resultSet using removeAll method

RETURN resultSet

FUNCTION main:

INITIALIZE set1 with some elements

INITIALIZE set2 with some elements

INITIALIZE differenceSet by calling difference(set1, set2)

PRINT differenceSet

END

Java code:

import java.util.HashSet;

import java.util.Set;

public class SetDifference {

// Difference of two sets

public static Set<Integer> difference(Set<Integer> set1, Set<Integer> set2) {

Set<Integer> resultSet = new HashSet<>(set1);

resultSet.removeAll(set2); // Use removeAll to remove common elements

return resultSet;

}

public static void main(String[] args) {

// Example Sets

Set<Integer> set1 = new HashSet<>();

set1.add(1);

set1.add(2);

set1.add(3);

set1.add(4);

Set<Integer> set2 = new HashSet<>();

set2.add(3);

set2.add(4);

set2.add(5);

set2.add(6);

// Difference

Set<Integer> differenceSet = difference(set1, set2);

System.out.println("Difference of set1 and set2: " + differenceSet);

}

}

**10. Write a function to find the k-th smallest element in a set.**

1. **Convert the set to a list and sort it, then find the k-th element.**

**Hint: Use a HashSet to convert the list to a set.**

1. **Use an ArrayList to convert the set to a list.**

**Write a function to check if two sets are equal**

**Pseudocode:**

BEGIN

FUNCTION findKthSmallestElement(set, k):

CONVERT set to list

SORT the list

RETURN the element at index k-1 (since list indices start at 0)

FUNCTION main():

DECLARE a set of integers

ADD elements to the set

DECLARE k as the position of the smallest element

CALL findKthSmallestElement with the set and k

PRINT the result

END

\*\*Java Code:\*\*

import java.util.\*;

public class KthSmallestElement {

// Function to find the k-th smallest element in a set

public static int findKthSmallest(Set<Integer> set, int k) {

// Convert set to list

List<Integer> list = new ArrayList<>(set);

// Sort the list

Collections.sort(list);

// Return the k-th smallest element

return list.get(k - 1); // k-1 because index starts at 0

}

public static void main(String[] args) {

// Example set

Set<Integer> numbers = new HashSet<>(Arrays.asList(5, 2, 9, 1, 7, 6));

// Specify the k-th position

int k = 3; // Find the 3rd smallest element

// Call the function and display result

int result = findKthSmallest(numbers, k);

System.out.println("The " + k + "-th smallest element is: " + result);

}

}

**11.Use ‘this’ keyword in the following purposes**

1. **Referencing instance variable**
2. **Invoking another constructor**
3. **Passing current object as a parameter**
4. **Returning current object**

**a)\*\*Pseudocode:\*\***

BEGIN

// Define a class demonstrating various uses of 'this' keyword

DEFINE CLASS `ThisKeywordExample`

// Instance variables

DECLARE int instanceVariable1

DECLARE int instanceVariable2

// Constructor with parameters

DEFINE CONSTRUCTOR `ThisKeywordExample(int a, int b)`

SET this.instanceVariable1 = a

SET this.instanceVariable2 = b

// Constructor invoking another constructor

DEFINE CONSTRUCTOR `ThisKeywordExample(int a)`

CALL this(a, 0) // Invoking another constructor with default value

// Method referencing instance variable

DEFINE FUNCTION `showInstanceVariables()`

PRINT "Instance Variable 1: " + this.instanceVariable1

PRINT "Instance Variable 2: " + this.instanceVariable2

// Method passing current object as a parameter

DEFINE FUNCTION `passObject(ThisKeywordExample obj)`

CALL obj.showInstanceVariables()

// Method returning current object

DEFINE FUNCTION `returnCurrentObject() RETURNS ThisKeywordExample`

RETURN this

// Main program execution

DEFINE CLASS `Main`

DEFINE FUNCTION `main(String[] args)`

// Create an instance of ThisKeywordExample

CREATE INSTANCE of `ThisKeywordExample` with parameters 10 and 20

CALL showInstanceVariables() from instance

// Create an instance using another constructor

CREATE INSTANCE of `ThisKeywordExample` with parameter 30

CALL showInstanceVariables() from instance

// Pass current object as a parameter

CALL passObject(instance) from instance

// Return current object and use it

CREATE INSTANCE of `ThisKeywordExample` with parameters 40 and 50

CREATE INSTANCE of `ThisKeywordExample` from returnCurrentObject() method

CALL showInstanceVariables() from newInstance

END

\*\*Java Code:\*\*

public class ThisKeywordExample {

// Instance variables

int instanceVariable1;

int instanceVariable2;

// Constructor with parameters

public ThisKeywordExample(int a, int b) {

this.instanceVariable1 = a; // Referencing instance variable

this.instanceVariable2 = b; // Referencing instance variable

}

// Constructor invoking another constructor

public ThisKeywordExample(int a) {

this(a, 0); // Invoking another constructor with default value

}

// Method referencing instance variables

public void showInstanceVariables() {

System.out.println("Instance Variable 1: " + this.instanceVariable1);

System.out.println("Instance Variable 2: " + this.instanceVariable2);

}

// Method passing current object as a parameter

public void passObject(ThisKeywordExample obj) {

obj.showInstanceVariables(); // Passing current object

}

// Method returning current object

public ThisKeywordExample returnCurrentObject() {

return this; // Returning current object

}

// Main method to demonstrate the use of 'this' keyword

public static void main(String[] args) {

// Create an instance of ThisKeywordExample using the constructor with parameters

ThisKeywordExample instance = new ThisKeywordExample(10, 20);

instance.showInstanceVariables();

// Create an instance using the constructor that invokes another constructor

ThisKeywordExample instance2 = new ThisKeywordExample(30);

instance2.showInstanceVariables();

// Pass current object as a parameter

instance.passObject(instance);

// Return current object and use it

ThisKeywordExample instance3 = new ThisKeywordExample(40, 50);

ThisKeywordExample returnedInstance = instance3.returnCurrentObject();

returnedInstance.showInstanceVariables();

}

}

b)pseudo code:

BEGIN

FUNCTION areSetsEqual(set1, set2):

IF set1 size is not equal to set2 size:

RETURN False

FOR each element in set1:

IF element is not present in set2:

RETURN False

RETURN True

FUNCTION main():

DECLARE two sets

ADD elements to both sets

CALL areSetsEqual with set1 and set2

PRINT whether the sets are equal or not

END

Java code:

import java.util.\*;

public class SetEqualityCheck {

// Function to check if two sets are equal

public static boolean areSetsEqual(Set<Integer> set1, Set<Integer> set2) {

return set1.equals(set2); // Check if two sets are equal

}

public static void main(String[] args) {

// Example sets

Set<Integer> set1 = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5));

Set<Integer> set2 = new HashSet<>(Arrays.asList(1, 2, 3, 4, 5));

// Call the function and display result

if (areSetsEqual(set1, set2)) {

System.out.println("The sets are equal.");

} else {

System.out.println("The sets are not equal.");

}

}

}

**12. Write a program for following to implement TreeSet and Sorted**

**Set**

1. **Sort even numbers in ascending order and odd numbers in descending order using SortedSet**
2. **Use TreeSet to perform binary search**

**Use TreeSet to perform Red-Black search**

\*\*Pseudocode:\*\*

BEGIN

FUNCTION customSort(set):

CREATE a SortedSet

FOR each number in set:

IF the number is even:

ADD to ascending order set

ELSE:

ADD to descending order set

MERGE both sets in the order (even asc, odd desc)

RETURN the sorted set

FUNCTION main():

DECLARE a set of integers

ADD elements to the set

CALL customSort with the set

PRINT the sorted result

END

Java Code:

import java.util.\*;

public class CustomSortedSet {

public static SortedSet<Integer> customSort(Set<Integer> numbers) {

SortedSet<Integer> evenAsc = new TreeSet<>(); // Even numbers in ascending order

SortedSet<Integer> oddDesc = new TreeSet<>(Collections.reverseOrder()); // Odd numbers in descending order

// Iterate through the set and separate even and odd numbers

for (Integer num : numbers) {

if (num % 2 == 0) {

evenAsc.add(num); // Add even numbers to ascending set

} else {

oddDesc.add(num); // Add odd numbers to descending set

}

}

// Merging both sorted sets

SortedSet<Integer> result = new TreeSet<>(evenAsc); // Add even numbers first

result.addAll(oddDesc); // Then add odd numbers

return result;

}

public static void main(String[] args) {

Set<Integer> numbers = new HashSet<>(Arrays.asList(5, 2, 9, 1, 7, 6, 3, 8, 10));

// Call the function

SortedSet<Integer> sortedNumbers = customSort(numbers);

// Print the sorted result

System.out.println("Sorted Set: " + sortedNumbers);

}

}

**13. Write a program to implement Dequeue using push() and pop() on both ends**

\*\*Pseudocode:\*\*

BEGIN

CLASS Deque:

DECLARE a list to store elements

METHOD pushFront(element):

Add element at the front of the deque

METHOD pushBack(element):

Add element at the back of the deque

METHOD popFront():

Remove and return element from the front of the deque

IF deque is empty:

RETURN error message

METHOD popBack():

Remove and return element from the back of the deque

IF deque is empty:

RETURN error message

METHOD display():

Print the current state of the deque

FUNCTION main():

CREATE instance of Deque class

CALL pushFront and pushBack methods

CALL popFront and popBack methods

CALL display to show the deque content

END

\*\*Java Code:\*\*

import java.util.LinkedList;

public class Deque {

private LinkedList<Integer> deque;

// Constructor

public Deque() {

deque = new LinkedList<>();

}

// Method to push an element at the front

public void pushFront(int element) {

deque.addFirst(element);

System.out.println("Pushed " + element + " to the front.");

}

// Method to push an element at the back

public void pushBack(int element) {

deque.addLast(element);

System.out.println("Pushed " + element + " to the back.");

}

// Method to pop an element from the front

public int popFront() {

if (!deque.isEmpty()) {

int removedElement = deque.removeFirst();

System.out.println("Popped " + removedElement + " from the front.");

return removedElement;

} else {

System.out.println("Deque is empty, cannot pop from front.");

return -1; // Indicating error

}

}

// Method to pop an element from the back

public int popBack() {

if (!deque.isEmpty()) {

int removedElement = deque.removeLast();

System.out.println("Popped " + removedElement + " from the back.");

return removedElement;

} else {

System.out.println("Deque is empty, cannot pop from back.");

return -1; // Indicating error

}

}

// Method to display the elements in the deque

public void display() {

System.out.println("Current Deque: " + deque);

}

public static void main(String[] args) {

Deque deque = new Deque();

// Push elements to the deque

deque.pushFront(10);

deque.pushBack(20);

deque.pushFront(5);

deque.pushBack(30);

// Display the current state of deque

deque.display();

// Pop elements from both ends

deque.popFront();

deque.popBack();

// Display the current state of deque after pop

deque.display();

}

}

14. Implement a Queue Using Two Stacks\*

\*\*Pseudocode:\*\*

1. Create two stacks, stack1 for enqueue and stack2 for dequeue.

2. For \*enqueue\*, push elements onto stack1.

3. For \*dequeue\*:

- If stack2 is empty, pop all elements from stack1 and push onto stack2.

- Pop the top element from stack2.

\*\*Java Code:\*\*

import java.util.Stack;

class QueueUsingTwoStacks {

Stack<Integer> stack1 = new Stack<>();

Stack<Integer> stack2 = new Stack<>();

public void enqueue(int value) {

stack1.push(value);

}

public int dequeue() {

if (stack2.isEmpty()) {

while (!stack1.isEmpty()) {

stack2.push(stack1.pop());

}

}

if (!stack2.isEmpty()) {

return stack2.pop();

}

throw new RuntimeException("Queue is empty");

}

public static void main(String[] args) {

QueueUsingTwoStacks queue = new QueueUsingTwoStacks();

queue.enqueue(1);

queue.enqueue(2);

queue.enqueue(3);

System.out.println(queue.dequeue()); // Output: 1

System.out.println(queue.dequeue()); // Output: 2

}

}

### \*15. Reverse the First K Elements of a Queue Using Stack

\* \*\*Pseudocode:\*\*

1. Dequeue the first K elements from the queue and push them onto a stack.

2. Pop the elements from the stack and enqueue them back to the queue.

3. Move the remaining elements in the queue to the back.

\*\*Java Code:\*\*

import java.util.LinkedList;

import java.util.Queue;

import java.util.Stack;

public class ReverseKElements {

// Function to reverse the first K elements of a queue

public static void reverseFirstK(Queue<Integer> queue, int k) {

if (queue.isEmpty() || k > queue.size() || k <= 0) {

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

return;

}

Stack<Integer> stack = new Stack<>();

// Step 1: Push the first K elements into the stack

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

stack.push(queue.poll()); // Dequeue element and push into stack

}

// Step 2: Pop elements from the stack and enqueue back to the queue

while (!stack.isEmpty()) {

queue.add(stack.pop());

}

// Step 3: Move the remaining elements to the back of the queue

int size = queue.size() - k;

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

queue.add(queue.poll()); // Dequeue and enqueue elements to maintain order

}

}

// Function to print the queue

public static void printQueue(Queue<Integer> queue) {

for (int element : queue) {

System.out.print(element + " ");

}

System.out.println();

}

public static void main(String[] args) {

Queue<Integer> queue = new LinkedList<>();

// Adding elements to the queue

queue.add(10);

queue.add(20);

queue.add(30);

queue.add(40);

queue.add(50);

queue.add(60);

queue.add(70);

System.out.println("Original Queue:");

printQueue(queue);

int k = 4; // Number of elements to reverse

reverseFirstK(queue, k);

System.out.println("Queue after reversing first " + k + " elements:");

printQueue(queue);

}

}

16. Restrict use of primitive types using generics

\*\*Pseudocode:\*\*

BEGIN

DEFINE a generic class called RestrictedType<T>

CHECK if T is a valid type (i.e., not a primitive)

IF not, throw an exception or show an error message

DEFINE a method to perform operations on the valid type

Perform operations like addition, multiplication, etc.

IMPLEMENT the class in the main program

CREATE an object of the generic class RestrictedType using Integer

CALL the method and print results

CREATE an object of the generic class RestrictedType using Double

CALL the method and print results

TRY to create an object of the generic class RestrictedType using int

SHOW an error message (since primitive types are not allowed)

END

\*\*Java Code:\*\*

public class RestrictedType<T extends Number> {

private T value;

// Constructor

public RestrictedType(T value) {

this.value = value;

}

// Method to perform some operation (like addition)

public void displayValue() {

System.out.println("The value is: " + value);

}

// Method to add two values (as an example)

public double add(T otherValue) {

return value.doubleValue() + otherValue.doubleValue();

}

public static void main(String[] args) {

// Using Integer type

RestrictedType<Integer> intType = new RestrictedType<>(10);

intType.displayValue();

System.out.println("Addition result: " + intType.add(15));

// Using Double type

RestrictedType<Double> doubleType = new RestrictedType<>(20.5);

doubleType.displayValue();

System.out.println("Addition result: " + doubleType.add(10.3));

// The following code will not compile, as int is not allowed directly

// RestrictedType<int> intTypeInvalid = new RestrictedType<>(10); // Error

// Only wrapper classes like Integer, Double, etc., are allowed

}

}

### \*17. Implement Stack and Queue Operations Using Vector\*

\*\*Pseudocode:\*\*

BEGIN

DEFINE a class called StackUsingVector

INITIALIZE a Vector to hold stack elements

DEFINE method push(element)

Add element to the end of the vector

DEFINE method pop()

Remove and return the element from the end of the vector

DEFINE method displayStack()

Print all elements in the vector

DEFINE a class called QueueUsingVector

INITIALIZE a Vector to hold queue elements

DEFINE method enqueue(element)

Add element to the end of the vector

DEFINE method dequeue()

Remove and return the element from the beginning of the vector

DEFINE method displayQueue()

Print all elements in the vector

IN MAIN PROGRAM:

CREATE an object of StackUsingVector

PUSH elements into the stack

POP elements from the stack

DISPLAY the elements in the stack

CREATE an object of QueueUsingVector

ENQUEUE elements into the queue

DEQUEUE elements from the queue

DISPLAY the elements in the queue

END

\*\*Java Code:\*\*

import java.util.Vector;

// Stack implementation using Vector

class StackUsingVector {

private Vector<Integer> stack;

// Constructor

public StackUsingVector() {

stack = new Vector<>();

}

// Push method to add element at the end

public void push(int element) {

stack.add(element);

System.out.println("Pushed: " + element);

}

// Pop method to remove and return the last element

public int pop() {

if (!stack.isEmpty()) {

int removedElement = stack.remove(stack.size() - 1);

System.out.println("Popped: " + removedElement);

return removedElement;

} else {

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

return -1;

}

}

// Display stack elements

public void displayStack() {

System.out.println("Stack: " + stack);

}

}

// Queue implementation using Vector

class QueueUsingVector {

private Vector<Integer> queue;

// Constructor

public QueueUsingVector() {

queue = new Vector<>();

}

// Enqueue method to add element at the end

public void enqueue(int element) {

queue.add(element);

System.out.println("Enqueued: " + element);

}

// Dequeue method to remove and return the first element

public int dequeue() {

if (!queue.isEmpty()) {

int removedElement = queue.remove(0);

System.out.println("Dequeued: " + removedElement);

return removedElement;

} else {

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

return -1;

}

}

// Display queue elements

public void displayQueue() {

System.out.println("Queue: " + queue);

}

}

public class Main {

public static void main(String[] args) {

// Stack operations

StackUsingVector stack = new StackUsingVector();

stack.push(10);

stack.push(20);

stack.push(30);

stack.displayStack();

stack.pop();

stack.displayStack();

System.out.println();

// Queue operations

QueueUsingVector queue = new QueueUsingVector();

queue.enqueue(100);

queue.enqueue(200);

queue.enqueue(300);

queue.displayQueue();

queue.dequeue();

queue.displayQueue();

}

}

### \*18. Use Vector to Store Custom Objects\*

\*\*Pseudocode:\*\*

BEGIN

DEFINE a class called CustomObject

DECLARE variables id, name, age

CONSTRUCTOR to initialize the variables

DEFINE method toString() to return string representation of the object

DEFINE a class called VectorExample

INITIALIZE a Vector to hold CustomObject elements

DEFINE method addObject(id, name, age)

Create a new CustomObject and add it to the vector

DEFINE method removeObject(id)

Remove the CustomObject from the vector based on its id

DEFINE method displayObjects()

Print all elements in the vector using toString() method

IN MAIN PROGRAM:

CREATE an object of VectorExample

ADD multiple custom objects to the vector

DISPLAY all objects

REMOVE an object from the vector by id

DISPLAY remaining objects

END

\*\*Java Code:\*\*

import java.util.Vector;

// Custom class to store data

class CustomObject {

private int id;

private String name;

private int age;

// Constructor

public CustomObject(int id, String name, int age) {

this.id = id;

this.name = name;

this.age = age;

}

// Override toString() to display object data

@Override

public String toString() {

return "CustomObject [ID: " + id + ", Name: " + name + ", Age: " + age + "]";

}

// Getter for ID

public int getId() {

return id;

}

}

// Class to demonstrate storing custom objects in a Vector

class VectorExample {

private Vector<CustomObject> vector;

// Constructor

public VectorExample() {

vector = new Vector<>();

}

// Add a custom object to the vector

public void addObject(int id, String name, int age) {

CustomObject obj = new CustomObject(id, name, age);

vector.add(obj);

System.out.println("Added: " + obj);

}

// Remove a custom object from the vector by id

public void removeObject(int id) {

for (CustomObject obj : vector) {

if (obj.getId() == id) {

vector.remove(obj);

System.out.println("Removed: " + obj);

break;

}

}

}

// Display all objects in the vector

public void displayObjects() {

System.out.println("Custom Objects in Vector:");

for (CustomObject obj : vector) {

System.out.println(obj);

}

}

}

public class Main {

public static void main(String[] args) {

// Create an instance of VectorExample

VectorExample vectorExample = new VectorExample();

// Add custom objects to the vector

vectorExample.addObject(1, "Alice", 25);

vectorExample.addObject(2, "Bob", 30);

vectorExample.addObject(3, "Charlie", 22);

// Display all objects

vectorExample.displayObjects();

// Remove an object by ID

vectorExample.removeObject(2);

// Display remaining objects

vectorExample.displayObjects();

}

}

### \*19. Evaluate a Postfix Expression Using Stack\*

\*\*Pseudocode:\*\*

BEGIN

FUNCTION evaluatePostfix(expression)

INITIALIZE an empty stack

FOR each character 'ch' in expression

IF 'ch' is a number

PUSH 'ch' onto the stack

ELSE IF 'ch' is an operator

POP two operands from the stack (operand1 and operand2)

PERFORM the operation 'ch' on operand2 and operand1

PUSH the result back onto the stack

END FOR

RETURN the single remaining value from the stack, which is the final result

END

\*\*Java Code:\*\*

import java.util.Stack;

public class PostfixEvaluator {

// Method to evaluate a postfix expression

public static int evaluatePostfix(String expression) {

// Stack to store operands

Stack<Integer> stack = new Stack<>();

// Traverse the given expression

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

char ch = expression.charAt(i);

// If the character is a digit, push it to the stack

if (Character.isDigit(ch)) {

stack.push(ch - '0'); // Convert char to int by subtracting '0'

}

// If the character is an operator, pop two operands from stack

else {

int operand2 = stack.pop();

int operand1 = stack.pop();

int result = 0;

// Perform the appropriate operation

switch (ch) {

case '+':

result = operand1 + operand2;

break;

case '-':

result = operand1 - operand2;

break;

case '\*':

result = operand1 \* operand2;

break;

case '/':

result = operand1 / operand2;

break;

}

// Push the result back onto the stack

stack.push(result);

}

}

// The final result will be the only element left in the stack

return stack.pop();

}

public static void main(String[] args) {

// Example postfix expression

String expression = "231\*+9-"; // Equivalent to (2 + (3 \* 1)) - 9

// Evaluate the postfix expression

int result = evaluatePostfix(expression);

// Output the result

System.out.println("Postfix Expression: " + expression);

System.out.println("Evaluated Result: " + result);

}

}

### \*20. Implement Depth-First Search (DFS) Using a Stack\* \*\*Pseudocode:\*\*

BEGIN

FUNCTION depthFirstSearch(graph, startVertex)

INITIALIZE a stack and push startVertex onto it

INITIALIZE a set to keep track of visited vertices

WHILE the stack is not empty

CURRENT vertex = POP from stack

IF CURRENT vertex is not in visited set

ADD CURRENT vertex to visited set

PRINT CURRENT vertex

FOR each neighbor in graph[CURRENT vertex]

IF neighbor is not in visited set

PUSH neighbor onto the stack

END

**Java Code:**

import java.util.\*;

public class DepthFirstSearch {

// Method to perform DFS using a stack

public static void depthFirstSearch(Map<Integer, List<Integer>> graph, int startVertex) {

// Stack to keep track of vertices to explore

Stack<Integer> stack = new Stack<>();

// Set to keep track of visited vertices

Set<Integer> visited = new HashSet<>();

// Push the start vertex onto the stack

stack.push(startVertex);

while (!stack.isEmpty()) {

// Pop the top vertex from the stack

int currentVertex = stack.pop();

// If the vertex has not been visited, process it

if (!visited.contains(currentVertex)) {

visited.add(currentVertex);

System.out.print(currentVertex + " "); // Print the current vertex

// Get the neighbors of the current vertex

List<Integer> neighbors = graph.get(currentVertex);

// Push unvisited neighbors onto the stack

for (int neighbor : neighbors) {

if (!visited.contains(neighbor)) {

stack.push(neighbor);

}

}

}

}

}

public static void main(String[] args) {

// Create a sample graph using an adjacency list

Map<Integer, List<Integer>> graph = new HashMap<>();

graph.put(0, Arrays.asList(1, 2));

graph.put(1, Arrays.asList(0, 3, 4));

graph.put(2, Arrays.asList(0, 5));

graph.put(3, Arrays.asList(1));

graph.put(4, Arrays.asList(1));

graph.put(5, Arrays.asList(2));

// Perform DFS starting from vertex 0

System.out.println("DFS traversal starting from vertex 0:");

depthFirstSearch(graph, 0);

}

}