TechPro Education

Java Interview Code Questions
Answers & Explanations

1) Write a program that takes a string as input and outputs the reverse of that string in Java.

First way:

```java
package MyPackage;

import java.util.Scanner;

no usages
public class StringReversal {
    no usages
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a string: ");
        String input = scanner.nextLine();

        String reversed = reverseString(input);
        System.out.println("Reversed string: " + reversed);
    }
    1 usage
    public static String reverseString(String input) {
        String reversed = "";
        for (int i = input.length() - 1; i >= 0; i--) {
            reversed = reversed + input.charAt(i);
        }
        return reversed;
    }
}
```

In this program, we first prompt the user to enter a string using the `Scanner` class. The `reverseString()` method takes the input string and uses a `StringBuilder` to build the reversed string character by character, starting from the last character of the input string. Finally, the reversed string is printed to the console.

When you run this program and provide a string as input, it will output the reverse of that string.

Second way:

```java
package MyPackage;

import java.util.Scanner;

no usages
public class StringReversal {
        no usages
        public static void main(String[] args) {
            Scanner scanner = new Scanner(System.in);

            System.out.print("Enter a string: ");
            String input = scanner.nextLine();

            String reversed = reverseString(input);
            System.out.println("Reversed string: " + reversed);
        }
        1 usage
        public static String reverseString(String input) {
            StringBuilder sb = new StringBuilder(input);
            sb.reverse();
            return sb.toString();
        }
}
```

We create a StringBuilder object named sb and pass the original string to its constructor.
Then, we call the reverse() method on the sb object, which modifies the string in place and reverses it.
Finally, we print both the original string and the reversed string using System.out.println()

2) Create a program that checks if a given string is a palindrome, meaning it reads the same forwards and backwards.

```java
package MyPackage;
import java.util.Scanner;
no usages
public class PalindromeChecker {
        no usages
        public static void main(String[] args) {
            Scanner scanner = new Scanner(System.in);
            System.out.print("Enter a string: ");
            String input = scanner.nextLine();

            boolean isPalindrome = checkPalindrome(input);
            if (isPalindrome) {
                System.out.println(input + " is a palindrome.");
            } else {
                System.out.println(input + " is not a palindrome.");
            }
        }
        1 usage
        public static boolean checkPalindrome(String input) {
            String reversed = reverseString(input);
            return input.equalsIgnoreCase(reversed);
        }
        1 usage
        public static String reverseString(String input) {
            StringBuilder reversed = new StringBuilder();
            for (int i = input.length() - 1; i >= 0; i--) {
                reversed.append(input.charAt(i));
            }
            return reversed.toString();
        }
}
```

In this program, we prompt the user to enter a string using the `Scanner` class. The `checkPalindrome()` method first calls the `reverseString()` method to obtain the reversed version of the input string. Then, it compares the original string (`input`) with the reversed string (`reversed`) using the `equalsIgnoreCase()` method to check if they are equal regardless of case.

If the input string is a palindrome, it will output "The string is a palindrome." Otherwise, it will output "The string is not a palindrome."

When you run this program and provide a string as input, it will check if the string is a palindrome and provide the corresponding output.

3) Write a program to generate the Fibonacci series up to a given number or a specified number of terms.

```java
package MyPackage;

import java.util.Scanner;
no usages
public class FibonacciSeries {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of terms to generate: ");
        int numTerms = scanner.nextInt();

        System.out.println("Fibonacci series:");
        generateFibonacciSeries(numTerms);
    }

    1 usage
    public static void generateFibonacciSeries(int numTerms) {
        int firstTerm = 0;
        int secondTerm = 1;

        for (int i = 1; i <= numTerms; i++) {
            System.out.print(firstTerm + " ");

            int nextTerm = firstTerm + secondTerm;
            firstTerm = secondTerm;
            secondTerm = nextTerm;
        }
    }
}
```

In this program, we prompt the user to enter the number of terms to generate in the Fibonacci series using the **`Scanner`** class. The **`generateFibonacciSeries()`** method takes the number of terms as input and calculates and prints the Fibonacci series.

The Fibonacci series starts with 0 and 1, and each subsequent term is the sum of the previous two terms. The **`generateFibonacciSeries()`** method uses a loop to iterate and calculate the Fibonacci series up to the specified number of terms. It prints each term in the series using **`System.out.print()`**.

When you run this program and provide the number of terms, it will generate and print the Fibonacci series up to that number of terms.

4) Create a program that calculates the factorial of a given number.

```java
package MyPackage;

import java.util.Scanner;
no usages
public class FactorialCalculator {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a number: ");
        int number = scanner.nextInt();

        long factorial = calculateFactorial(number);
        System.out.println("Factorial of " + number + " is: " + factorial);
    }

    1 usage
    public static long calculateFactorial(int number) {
        if (number < 0) {
            throw new IllegalArgumentException("Number cannot be negative.");
        }

        long factorial = 1;
        for (int i = 1; i <= number; i++) {
            factorial *= i;
        }
        return factorial;
    }
}
```

In this program, we prompt the user to enter a number using the `Scanner` class. The `calculateFactorial()` method takes the number as input and calculates its factorial using a loop.

The factorial of a non-negative integer is the product of all positive integers less than or equal to that number. The `calculateFactorial()` method initializes a `factorial` variable to 1 and then multiplies it with the numbers from 1 to the given number using a loop.

If the given number is negative, the program throws an `IllegalArgumentException` since factorial is not defined for negative numbers.

When you run this program and provide a number, it will calculate and print the factorial of that number.

5) Create a program that swaps the values of two integers.

First way:

```java
int a = 5;
int b = 10;

int temp = a;
a = b;
b = temp;

System.out.println("After swapping: a = " + a + ", b = " + b);
```

In this approach, we use a temporary variable `temp` to hold the value of `a` before swapping. We assign the value of `b` to `a` and then assign the value of `temp` (which was originally the value of `a`) to `b`. As a result, the values of `a` and `b` are swapped.

Second way:

```java
int a = 5;
int b = 10;

a = a ^ b;
b = a ^ b;
a = a ^ b;

System.out.println("After swapping: a = " + a + ", b = " + b);
```

In this approach, we perform bitwise XOR operations to swap the values without using an extra variable. The XOR operator (`^`) performs the exclusive OR operation on the bits. The advantage of this method is that it avoids the need for a temporary variable.

After swapping the values using either approach, you can print the new values of `a` and `b` to verify the swap.

6) Create a program to find unique characters different from space in a String

```java
package MyPackage;

import java.util.HashSet;
import java.util.Set;
no usages
public class UniqueCharactersFinder {

    public static void main(String[] args) {
        String inputString = "Hello World!";
        Set<Character> uniqueCharacters = findUniqueCharacters(inputString);

        System.out.println("Unique characters in the string: " + uniqueCharacters);
    }
    1 usage
    public static Set<Character> findUniqueCharacters(String str) {

        Set<Character> uniqueCharacters = new HashSet<>();

        for (char c : str.replace( target: " ",  replacement: "").toCharArray()) {
                uniqueCharacters.add(c);
        }

        return uniqueCharacters;
    }
}
```

In this program, we have a string `inputString` which contains the text "Hello, World!". The `findUniqueCharacters()` method takes the input string and returns a `Set` of unique characters found in the string.

Within the `findUniqueCharacters()` method, we iterate through each character of the input string using a `for-each` loop. We add each character to the `uniqueCharacters` set if it is a letter (you can add additional checks based on your requirements). The `Set` data structure ensures that only unique characters are stored.

Finally, in the `main()` method, we call `findUniqueCharacters()` with the input string and print the unique characters using `System.out.println()`.

7) Create a program to find non-repeated characters different from space in a String

```java
package MyPackage;

import java.util.ArrayList;
import java.util.List;

no usages
public class nonRepeatedCharactersFinder {

        public static void main(String[] args) {
            String inputString = "Hello World!";
            List<Character> uniqueCharacters = findUniqueCharacters(inputString);

            System.out.println("Unique characters in the string: " + uniqueCharacters);
        }

        1 usage
        public static List<Character> findUniqueCharacters(String str) {
            List<Character> uniqueCharacters = new ArrayList<>();
            str = str.replace( target: " ", replacement: "");

            for (int i = 0; i < str.length(); i++) {
                char c = str.charAt(i);
                if (str.indexOf(c) == str.lastIndexOf(c)) {
                    uniqueCharacters.add(c);
                }
            }

            return uniqueCharacters;
        }
}
```

In this program, we have a string inputString which contains the text "Hello, World!". The findUniqueCharacters() method takes the input string and returns a List of unique characters found in the string.

Within the findUniqueCharacters() method, we iterate through each character of the input string using a for loop and an index variable i.
For each character at index i, we use indexOf(c) to find the first occurrence of that character in the string and lastIndexOf(c) to find the last occurrence of that character in the string.
If the indices are the same, it means that the character is unique and we add it to the uniqueCharacters list.

Finally, in the main() method, we call findUniqueCharacters() with the input string and print the unique characters using System.out.println().

8) Given an array of non-duplicating numbers from 1 to n where one number is missing, write an efficient java program to find that missing number.

```java
package MyPackage;

import java.util.stream.IntStream;
no usages
public class MissingNumberFinder {

    public static void main(String[] args) {
        int[] array = {1, 3, 4, 2, 6, 7, 5, 9, 8, 11, 12};
        int n = array.length + 1;

        int missingNumber = findMissingNumber(array, n);

        System.out.println("Missing number: " + missingNumber);
    }

    1 usage
    public static int findMissingNumber(int[] array, int n) {

        int expectedSum = IntStream.rangeClosed(1,n).sum();
        int actualSum = 0;

        for (int num : array) {
            actualSum += num;
        }

        return expectedSum - actualSum;
    }
}
```

In this program, we have an array array containing non-duplicating numbers from 1 to n with one number missing.
The findMissingNumber() method takes the array and the value of n as input and returns the missing number.

To find the missing number, we first calculate the expected sum of all numbers from 1 to n . This gives us the sum of a series of consecutive numbers.
Then, we iterate through the given array and calculate the actual sum of its elements.

The missing number can be determined by subtracting the actual sum from the expected sum, as the missing number is the difference between the expected sum and the actual sum.

In the main() method, we provide a sample array and calculate the missing number using findMissingNumber(). Finally, we print the missing number.

9) Create a program to find repeated characters different from space in a String.

```java
package MyPackage;

import java.util.HashSet;
import java.util.Set;

no usages
public class repeatedCharactersFinder {

    public static void main(String[] args) {
        String inputString = "Hello World!";
        Set<Character> repeatedCharacters = findRepeatedCharacters(inputString);

        System.out.println("Non-repeated characters in the string: " + repeatedCharacters);
    }

    1 usage
    public static Set<Character> findRepeatedCharacters(String str) {
        Set<Character> repeatedCharacters = new HashSet<>();
        str = str.replace( target: " ", replacement: "");

        for (int i = 0; i < str.length(); i++) {
            char c = str.charAt(i);
            if (str.indexOf(c) != str.lastIndexOf(c)) {
                repeatedCharacters.add(c);
            }
        }
        return repeatedCharacters;
    }
}
```

In this program, we have a string inputString which contains the text "Hello, World!". The findRepeatedCharacters() method takes the input string and returns a Set of repeated characters found in the string.

Within the findRepeatedCharacters() method, we iterate through each character of the input string using a for loop and an index variable i.
For each character at index i, we use indexOf(c) to find the first occurrence of that character in the string and lastIndexOf(c) to find the last occurrence of that character in the string.
If the indices are different, it means that the character is repeated and we add it to the repeatedCharacters set.

Finally, in the main() method, we call findRepeatedCharacters() with the input string and print the unique characters using System.out.println().

10) Create a program to remove duplicates in an ArrayList.

```java
package MyPackage;

import java.util.ArrayList;
import java.util.HashSet;
no usages
public class RemoveDuplicates {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(2);
        numbers.add(1);
        numbers.add(5);
        System.out.println("Original ArrayList: " + numbers);
        removeDuplicates(numbers);
        System.out.println("ArrayList after removing duplicates: " + numbers);
    }
    1 usage
    public static void removeDuplicates(ArrayList<Integer> list) {
        HashSet<Integer> uniqueSet = new HashSet<>();
        ArrayList<Integer> newList = new ArrayList<>();
        for (Integer number : list) {
            if (uniqueSet.add(number)) {
                newList.add(number);
            }
        }
        list.clear();
        list.addAll(newList);
    }
}
```

In this program, we have an ArrayList called `**numbers**` that contains integers with duplicates. The `**removeDuplicates()**` method takes the ArrayList as input and removes the duplicate elements.

Within the `**removeDuplicates()**` method, we create a HashSet called `**uniqueSet**` to store unique elements. We also create a new ArrayList called `**newList**` to hold the elements without duplicates.

We iterate through each element in the original ArrayList. If the element is not already present in the HashSet (`**uniqueSet.add(number)**` returns true), we add it to the `**newList**`. This ensures that only unique elements are added to the `**newList**`.

After iterating through all the elements, we clear the original ArrayList and add all the elements from the `**newList**` back to the original ArrayList using `**list.addAll(newList)**`.

In the `**main()**` method, we provide a sample ArrayList, call the `**removeDuplicates()**` method, and print the ArrayList before and after removing duplicates.

11) Create a program to count the words used in a sentence.

```java
package MyPackage;

import java.util.HashMap;
no usages
public class WordCountFinder {
        public static void main(String[] args) {
            String sentence = "This is a sample sentence to demonstrate word count in Java";

            HashMap<String, Integer> wordCountMap = countWords(sentence);

            for (String word : wordCountMap.keySet()) {
                int count = wordCountMap.get(word);
                System.out.println(word + ": " + count);
            }
        }
        1 usage
        public static HashMap<String, Integer> countWords(String sentence) {
            HashMap<String, Integer> wordCountMap = new HashMap<>();
            String[] words = sentence.split( regex: "\\s+");
            for (String word : words) {
                if (wordCountMap.containsKey(word)) {
                    int count = wordCountMap.get(word);
                    wordCountMap.put(word, count + 1);
                } else {
                    wordCountMap.put(word, 1);
                }
            }
            return wordCountMap;
        }
}
```

In this program, we have a string `sentence` that contains the sentence for which we want to count the words. The `countWords()` method takes the sentence as input and returns a HashMap containing each unique word as the key and its count as the value.

Within the `countWords()` method, we create a HashMap called `wordCountMap` to store the word-count pairs. We split the sentence into an array of words using the `split("\\s+")` method, which splits the string at one or more whitespace characters.

We then iterate through each word in the `words` array. If the word is already present in the `wordCountMap`, we retrieve its current count and increment it by 1. Otherwise, we add the word to the `wordCountMap` with a count of 1.

Finally, we return the `wordCountMap` from the `countWords()` method.

In the `main()` method, we provide a sample sentence, call the `countWords()` method, and print the word count for each word in the HashMap.

12) Create a program to find the Second Highest number in an ArrayList.

```java
package MyPackage;

import java.util.ArrayList;
no usages
public class SecondHighestNumberFinder {
        public static void main(String[] args) {
            ArrayList<Integer> numbers = new ArrayList<>();
            numbers.add(5);
            numbers.add(8);
            numbers.add(2);
            numbers.add(10);
            numbers.add(3);
            numbers.add(7);
            int secondHighest = findSecondHighest(numbers);
            System.out.println("Second highest number: " + secondHighest);
        }
        1 usage
        public static int findSecondHighest(ArrayList<Integer> list) {
            int highest = list.get(0);
            int secondHighest = list.get(0);
            for (int number : list) {
                if (number > highest) {
                    secondHighest = highest;
                    highest = number;
                } else if (number > secondHighest && number != highest) {
                    secondHighest = number;
                }
            }
            return secondHighest;
        }
}
```

In this program, we have an ArrayList called `**numbers**` that contains integers. The `**findSecondHighest()**` method takes the ArrayList as input and returns the second highest number.

Within the `**findSecondHighest()**` method, we initialize two variables `**highest**` and `**secondHighest**` to the minimum possible value of an integer using `**Integer.MIN_VALUE**`. We iterate through each number in the list and update the `**highest**` and `**secondHighest**` variables accordingly.

If a number is greater than the current `**highest**` value, we update both `**highest**` and `**secondHighest**`. If a number is greater than the current `**secondHighest**` value but not equal to the `**highest**` value, we update only the `**secondHighest**` value.

Finally, we return the `**secondHighest**` value from the `**findSecondHighest()**` method.

In the `**main()**` method, we provide a sample ArrayList, call the `**findSecondHighest()**` method, and print the second highest number.

13) Create a program to find the longest palindromic substring of a given substring

```java
public class LongestPalindromicSubstring {
    public static void main(String[] args) {
        String str = "babad";
        String longestPalindromicSubstring = findLongestPalindromicSubstring(str);
        System.out.println("Longest Palindromic Substring: " + longestPalindromicSubstring);
    }
    1 usage
    public static String findLongestPalindromicSubstring(String str) {
        if (str == null || str.length() < 2) { return str; }
        int start = 0;
        int end = 0;
        for (int i = 0; i < str.length(); i++) {
            int len1 = expandAroundCenter(str, i, i);
            int len2 = expandAroundCenter(str, i,  right: i + 1);
            int length = Math.max(len1, len2);
            if (length > end - start) {
                start = i - (length - 1) / 2;
                end = i + length / 2;
            }
        }
        return str.substring(start, end + 1);
    }
    2 usages
    public static int expandAroundCenter(String str, int left, int right) {
        while (left >= 0 && right < str.length() && str.charAt(left) == str.charAt(right)) {
            left--;
            right++;
        }
        return right - left - 1;
    }
}
```

In this program, we have a given string `str`. The `findLongestPalindromicSubstring()` method takes the string as input and returns the longest palindromic substring.

We use the "expand around center" approach to find palindromic substrings. For each character in the string, we expand around it to check if it forms a palindrome. We handle both odd and even length palindromes separately.

The `expandAroundCenter()` method takes the string, left index, and right index as input and expands the indices outward while the characters at those indices are the same. It returns the length of the palindromic substring.

In the `findLongestPalindromicSubstring()` method, we initialize `start` and `end` variables to keep track of the indices of the longest palindromic substring found so far. We iterate through each character in the string and expand around it. If the length of the palindrome is greater than the current longest substring, we update the `start` and `end` indices accordingly.

Finally, we return the substring from the `start` index to the `end` index.

In the `main()` method, we provide a sample string, call the `findLongestPalindromicSubstring()` method, and print the result.

14) Create a Java program to sort all letters in a String in alphabetical order

```java
package MyPackage;

import java.util.Arrays;
no usages
public class StringSorter {

    public static void main(String[] args) {
        String input = "TechPro Education!";

        String sortedString = sortString(input);
        System.out.println("Sorted String: " + sortedString);
    }
    1 usage
    public static String sortString(String input) {

        //remove all characters different from letters from the String
        input = input.replaceAll( regex: "[^A-Za-z]", replacement: "");

        // Convert the string to character array
        char[] chars = input.toCharArray();

        // Sort the character array
        Arrays.sort(chars);

        // Convert the sorted character array back to string
        String sortedString = new String(chars);

        return sortedString;
    }
}
```

In this program, we have a string `**input**` that contains the string we want to sort. The `**sortString()**` method takes the input string and returns the sorted string with all letters in alphabetical order.

Within the `**sortString()**` method, we convert the input string to a character array using the `**toCharArray()**` method. We then use the `**Arrays.sort()**` method to sort the character array in ascending order.

Finally, we convert the sorted character array back to a string using the `**String**` constructor, and return the sorted string.

In the `**main()**` method, we provide a sample input string, call the `**sortString()**` method, and print the sorted string.

15) Create a Java program to find the sum of the digits of an integer

```java
package MyPackage;
no usages
public class DigitSumCalculator {

    public static void main(String[] args) {
        int number = 12345;
        int digitSum = calculateDigitSum(number);
        System.out.println("Sum of the digits: " + digitSum);
    }
    1 usage
    public static int calculateDigitSum(int number) {
        int sum = 0;

        while (number != 0) {
            int digit = number % 10;
            sum += digit;
            number /= 10;
        }
        return sum;
    }
}
```

In this program, we have an integer `number` for which we want to find the sum of its digits. The `calculateDigitSum()` method takes the number as input and returns the sum of its digits.

Within the `calculateDigitSum()` method, we initialize a variable `sum` to 0. We use a while loop to iterate until the number becomes 0. In each iteration, we extract the last digit of the number using the modulus operator `%` and add it to the sum. We then update the number by dividing it by 10 to remove the last digit.

Finally, we return the sum of the digits.

In the `main()` method, we provide a sample number, call the `calculateDigitSum()` method, and print the result.

16) Create a Java program to find the two closest integers in an ArrayList

1

```java
package MyPackage;

import java.util.ArrayList;
import java.util.Collections;
no usages
public class ClosestIntegersFinder {

    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(5);
        numbers.add(8);
        numbers.add(2);
        numbers.add(10);
        numbers.add(3);
        numbers.add(7);

        Pair closestIntegers = findClosestIntegers(numbers);
        System.out.println("Closest Integers: " + closestIntegers.getFirst() + ", " + closestIntegers.getSecond());
    }
```

2

```java
    public static Pair findClosestIntegers(ArrayList<Integer> list) {
        if (list.size() < 2) {
            throw new IllegalArgumentException("At least two integers are required.");
        }
        Collections.sort(list); // Sort the list in ascending order

        int minDiff = list.get(0);
        Pair closestIntegers = null;

        for (int i = 0; i < list.size() - 1; i++) {
            int diff = list.get(i + 1) - list.get(i);

            if (diff < minDiff) {
                minDiff = diff;
                closestIntegers = new Pair(list.get(i), list.get(i + 1));
            }
        }
        return closestIntegers;
    }
```

3

```java
    static class Pair {
        2 usages
        private int first, second;
        1 usage
        public Pair(int first, int second) {
            this.first = first;
            this.second = second;
        }
        1 usage
        public int getFirst() {
            return first;
        }
        1 usage
        public int getSecond() {
            return second;
        }
    }
}
```

In this program, we have an ArrayList called `numbers` that contains integers. The `findClosestIntegers()` method takes the ArrayList as input and returns a Pair object representing the two closest integers.

Within the `findClosestIntegers()` method, we sort the ArrayList in ascending order using `Collections.sort()`. Then, we iterate through the sorted list and calculate the difference between each pair of adjacent numbers. We keep track of the minimum difference found so far and update it along with the closestIntegers Pair accordingly.

The `Pair` class is a simple class that holds two integers representing the closest pair found.

In the `main()` method, we provide a sample ArrayList, call the `findClosestIntegers()` method, and print the closest integers.

17) Create a Java program to find the all subsets of a set

```java
package MyPackage;

import java.util.ArrayList;
import java.util.List;
no usages
public class SubsetGenerator {
        public static void main(String[] args) {
            int[] set = {1, 2, 3};
            List<List<Integer>> subsets = generateSubsets(set);

            System.out.println("Subsets of the set:");
            for (List<Integer> subset : subsets) {
                System.out.println(subset);
            }
        }
        1 usage
        public static List<List<Integer>> generateSubsets(int[] set) {
            List<List<Integer>> subsets = new ArrayList<>();
            backtrack(subsets, new ArrayList<>(), set,  start: 0);
            return subsets;
        }
        2 usages
        public static void backtrack(List<List<Integer>> subsets, List<Integer> currentSubset, int[] set, int start) {
            subsets.add(new ArrayList<>(currentSubset));
            for (int i = start; i < set.length; i++) {
                currentSubset.add(set[i]);
                backtrack(subsets, currentSubset, set,  start: i + 1);
                currentSubset.remove( index: currentSubset.size() - 1);
            }
        }
}
```

In this program, we have an array `set` that represents the original set for which we want to generate all subsets. The `generateSubsets()` method takes the set as input and returns a list of all subsets.

Within the `generateSubsets()` method, we create an empty `subsets` list to store the subsets. We then call the `backtrack()` method to generate the subsets recursively.

The `backtrack()` method follows a backtracking approach. It adds the currentSubset to the subsets list, then iterates through the remaining elements in the set, adding each element to the currentSubset, recursively calling the `backtrack()` method to generate subsets starting from the next index, and finally removing the added element from the currentSubset before moving to the next iteration.

The base case for the recursion is when we have processed all elements in the set, at which point we add the currentSubset to the subsets list.

In the `main()` method, we provide a sample set, call the `generateSubsets()` method, and print the generated subsets.

18) Create a Java program to find a pair whose sum is the closest to zero in an Array

1

```java
package MyPackage;

import java.util.Arrays;
no usages
public class ClosestSumPairFinder {

    public static void main(String[] args) {
        int[] array = {4, -2, 6, 8, -5, -1, 2, 7};
        Pair closestSumPair = findClosestSumPair(array);

        System.out.println("Pair with closest sum to zero: " + closestSumPair);
    }
```

1

```java
    public static Pair findClosestSumPair(int[] array) {
        if (array.length < 2) {
            throw new IllegalArgumentException("At least two elements are required.");
        }
        Arrays.sort(array); // Sort the array in ascending order
        int left = 0;
        int right = array.length - 1;
        int closestSum = Integer.MAX_VALUE;
        Pair closestPair = null;
        while (left < right) {
            int sum = array[left] + array[right];
            if (Math.abs(sum) < Math.abs(closestSum)) {
                closestSum = sum;
                closestPair = new Pair(array[left], array[right]);
            }
            if (sum < 0) {
                left++;
            } else {
                right--;
            }
        }
        return closestPair;
    }
```

3

```java
    static class Pair {
        3 usages
        private int first;
        3 usages
        private int second;
        1 usage
        public Pair(int first, int second) {
            this.first = first;
            this.second = second;
        }
        no usages
        public int getFirst() {
            return first;
        }
        no usages
        public int getSecond() {
            return second;
        }
        @Override
        public String toString() {
            return "(" + first + ", " + second + ")";
        }
    }
}
```

In this program, we have an array called `array` in which we want to find a pair whose sum is closest to zero. The `findClosestSumPair()` method takes the array as input and returns a Pair object representing the pair with the closest sum to zero.

Within the `findClosestSumPair()` method, we first sort the array in ascending order using `Arrays.sort()`. Then, we initialize two pointers `left` and `right` pointing to the start and end of the array, respectively.

We also initialize variables `closestSum` and `closestPair` to keep track of the closest sum found so far and the corresponding pair.

We iterate through the array while the `left` pointer is less than the `right` pointer. In each iteration, we calculate the sum of the current pair and compare its absolute value with the absolute value of the `closestSum`. If the current sum is closer to zero, we update the `closestSum` and `closestPair`.

To move the pointers, if the current sum is less than zero, we increment the `left` pointer to consider a larger element. Otherwise, if the sum is greater than or equal to zero, we decrement the `right` pointer to consider a smaller element.

Finally, we return the `closestPair` containing the pair with the closest sum to zero.

In the `main()` method, we provide a sample array, call the `findClosestSumPair()` method, and print the closest sum pair.

19) Create a Java program to separate odd and even numbers in an integer ArrayList

```java
package MyPackage;

import java.util.ArrayList;
import java.util.List;
no usages
public class OddEvenSeparator {
        public static void main(String[] args) {
            ArrayList<Integer> numbers = new ArrayList<>();
            numbers.add(1);
            numbers.add(2);
            numbers.add(3);
            numbers.add(4);
            numbers.add(5);
            numbers.add(6);
            List<Integer> oddNumbers = new ArrayList<>();
            List<Integer> evenNumbers = new ArrayList<>();
            separateOddEven(numbers, oddNumbers, evenNumbers);
            System.out.println("Odd Numbers: " + oddNumbers);
            System.out.println("Even Numbers: " + evenNumbers);
        }
        1 usage
        public static void separateOddEven(List<Integer> numbers, List<Integer> oddNumbers, List<Integer> evenNumbers) {
            for (Integer number : numbers) {
                if (number % 2 == 0) {
                    evenNumbers.add(number);
                } else {
                    oddNumbers.add(number);
                }
            }
        }
}
```

In this program, we have an ArrayList called `**numbers**` that contains integers. The `**separateOddEven()**` method takes the `**numbers**` list as input, and two empty ArrayLists `**oddNumbers**` and `**evenNumbers**` to store the separated odd and even numbers.

Within the `**separateOddEven()**` method, we iterate through each number in the `**numbers**` list. If the number is divisible by 2 (i.e., an even number), we add it to the `**evenNumbers**` list. Otherwise, if it is not divisible by 2 (i.e., an odd number), we add it to the `**oddNumbers**` list.

In the `**main()**` method, we provide a sample ArrayList of numbers, create two empty ArrayLists for odd and even numbers, call the `**separateOddEven()**` method, and print the separated odd and even numbers.

20) Create a Java program to put all zeros to the end in an Array

```java
package MyPackage;

import java.util.Arrays;
no usages
public class ZeroToEnd {

    public static void main(String[] args) {

        int[] array = {0, 2, 0, 1, 0, 3, 0, 4};
        System.out.println("Original Array: " + Arrays.toString(array));

        zeroToEnd(array);
        System.out.println("Array with Zeros at the End: " + Arrays.toString(array));
    }

    1 usage
    public static void zeroToEnd(int[] array) {
        int index = 0;

        for (int i = 0; i < array.length; i++) {
            if (array[i] != 0) {
                array[index++] = array[i];
            }
        }

        while (index < array.length) {
            array[index++] = 0;
        }
    }
}
```

In this program, we have an array called `array` that contains elements, including zeros. The `zeroToEnd()` method takes the array as input and modifies it in-place to put all the zeros at the end.

Within the `zeroToEnd()` method, we use two pointers: `index` and `i`. The `index` pointer keeps track of the next position where a non-zero element should be placed, and the `i` pointer iterates through the array elements.

We iterate through the array with the `i` pointer. If the current element is non-zero, we assign it to the `index` position and increment the `index` pointer. This process effectively moves all non-zero elements towards the beginning of the array.

After processing all non-zero elements, we fill the remaining positions in the array from the `index` pointer onwards with zeros.

In the `main()` method, we provide a sample array, call the `zeroToEnd()` method, and print the modified array before and after moving zeros to the end.