**Important Theoritical Points**

1. Sorted() function : It performs natural sorting. Example: Sorting a list of integers.

But if we want to sort an list of objects based on a particular value in object the we should provide a comparator. Example: Sorting list of employees based on salary or based on age.

Accepts Stream of values and return a sorted stream of values. For doing custom sorted it can take Comparator as argument as well as normal function for comparing values.

1. Comparator class has following methods:

* Comparator.comparing(Employee::getLastName)
* Comparator.comparingInt(Employee::getAge)
* Comparator.comparingDouble(Employee::getSalary)
* Comparator.comparingLong(Employee::getId)
* Comparator.reverseOrder()

Above mentioned are List Comparators. We also have Map Comparators

* sorted(Map.Entry.comparingByKey()) : Creates a comparator that compares map entries based on their keys. Entries are sorted in ascending order of keys.
* sorted(Map.Entry.comparingByValue()) : Creates a comparator that compares map entries based on their values. Entries are sorted in ascending order of values.
* sorted(Map.Entry.comparingByKeyAndValue()): Creates a comparator that compares map entries first by key and then by value.

In Map Comparator we can have specialized comparing functions

* Comparator<String> customKeyComparator = (key1, key2) -> key2.compareTo(key1); // Custom key comparator

sorted(Map.Entry.comparingByKey(customKeyComparator))

* Comparator<Integer> customValueComparator = (value1, value2) -> value2.compareTo(value1); // Custom value comparator

sorted(Map.Entry.comparingByValue(customValueComparator))

* Comparator<String> customKeyComparator = (key1, key2) -> key2.compareTo(key1); // Custom key comparator

Comparator<Integer> customValueComparator = (value1, value2) -> value2.compareTo(value1); // Custom value comparator

sorted(Map.Entry.comparingByKeyAndValue(customKeyComparator, customValueComparator))

1. Collectors.groupingBy : return a map and always take a value that we can group.
2. Terminal Operations:

* forEach(Consumer<? super T> action): Performs an action for each element in the stream.
* toArray(): Collects the elements of the stream into an array.
* reduce(T identity, BinaryOperator<T> accumulator): Reduces the elements of the stream to a single value using an associative accumulation function.
* collect(Collector<? super T, A, R> collector): Collects the elements of the stream using a Collector to produce a result of a specified type.
* min(Comparator<? super T> comparator): Returns the Optional with minimum element of the stream according to the provided comparator.
* max(Comparator<? super T> comparator): Returns the Optional with maximum element of the stream according to the provided comparator.
* count(): Returns the count of elements in the stream as a long.

1. Collectors:

* **toList()**: Collects the elements of the stream into a List.
* **toSet()**: Collects the elements of the stream into a Set.
* **toMap(keyMapper, valueMapper)**: Collects the elements of the stream into a Map, where keyMapper and valueMapper are functions to extract keys and values.
* joining(): Collects the elements of the stream into a single String by concatenating them with a specified delimiter.
* **groupingBy(classifier)**: Groups elements of the stream into a Map based on a classifier function.
* **partitioningBy(predicate)**: Partitions the elements of the stream into two groups (true and false) based on a predicate, resulting in a Map<Boolean, List<T>>.
* **summingInt()/summingLong()/summingDouble()**: Calculates the sum of integer, long, or double values in the stream.
* **averagingInt()/averagingLong()/averagingDouble()**: Calculates the average of integer, long, or double values in the stream.
* **maxBy()/minBy()**: Collects the maximum or minimum element of a stream based on a provided comparator. It returns “Optional”.

1. stream().anyMatch(predicate) and stream().allMatch(predicate) : will always return boolean value.
2. Collectors.counting():Mostly used inside collectors.groupingBy(). For Counting we also have stream.count().
3. Map.Entry::getKey and Map.entry::getValue
4. **Suppose an Employee Array List contains 300 records. Each record is having name and salary attribute. How can we sort based on salary?**

List<Employee> sortedList = employees.stream().sorted((e1,e2) -> e1.getSalary()-e2.getSalary()) .collect(Collectors.toList());

Or

List <Employee> sortedList = employees.stream().sorted(new Comparator<Employee>(){  
 public int compare(Employee e1,Employee e2){ return e1.getSalary() – e2.getSalary();}})  
 .collect(Collectors.toList());

Or

List<Employee> sortedList = employees.stream().sorted(Comparator.comparingDouble(Employee::getSalary()));

1. **Suppose there is list of employees (employee class has name, age and salary as properties).  
   Write a Program in which we have to increase salary of employees by 10% if there age is greater than 25**

List <Employee> newEmpList = empList.stream().map(e->{  
 If(e.getAge()>25){ e.setSalary(e.getSalary()\*1.10);}  
 return e;  
 }).collect(Collectors.toList());

1. **Sort HashMap By Value**

Map<String, Integer> sortedMap = unsortedMap.entrySet()  
 .stream()  
 .sorted(Map.Entry.comparingByValue())  
 .collect(Collectors.toMap(  
 Map.Entry::getKey,  
 Map.Entry::getValue,  
 (e1, e2) -> e1,  
 LinkedHashMap::new  
 )  
 );

Map<String, Integer> sortedMap = unsortedMap.entrySet().  
 .stream()  
 .sorted((e1,e2)-> e1.getValues()- e2.getVaues())  
 .collect(Collectors.toMap(  
 Map.Entry:: getKey  
 Map.Entry::getValue,  
 (e1,e2)->e1,  
 LinkedHashMap::new  
 ))

Note: *in collectors.toMap() we have passed 4 parameters*

1. Map.Entry::getKey 🡪 Indicate how to get key for map
2. Map.Entry:: getValue 🡪 Indicate how to get value for map
3. (e1,e2) -> e1 -> function that indicates that, in the case of a collision, we keep the existing(e1) entry.
4. By default, a toMap() method will return a HashMap. for returning LinkedHashMap we use LinkedHashMap::new
5. **Employee class has 2 attributes (name and city) and we want to groupBy employees based on city.**

Input :   
Name : Amar, City: Pune  
Name: Raj, City: Pune  
Name: Neha, City: Mumbai  
Name: Sam, City: Mumbai

Output:   
Pune : Amar, Raj  
Mumbai: Neha, Sam  
Solution:  
*Map<String, List<employees>> groupedBy = employees.stream().collect(Collectors.groupingBy(Employee::getCity));*Note : Collectors.groupingBy by default returns a Map and always take a value via which we can group a list of collection.

1. **Given a list of integers, write a program to find list of integers that starts with 1.**

*numbers.stream().map(num-> “”+num).filter(num->num.startsWith(1)).forEach(System.out::println);*

1. **Find Highest Paid employee from Department | Parallel & Sequential Stream**

Input:  
List<Employee> employees = Arrays.asList(  
new Employee(“Emp1”, “CS”, 10000),  
new Employee(“Emp2”, “CS”, 15000),  
new Employee(“Emp3”, “IT”, 20000),  
new Employee(“Emp4”, “IT”, 25000)  
);

Output: To find highest employee in each department.  
CS: Employee(“Emp2”, “CS”, 15000)  
IT: Employee(“Emp4”, “CS”, 25000)

*Map<String, Employee> topSalaryByDepartment = employeeList.stream()  
 .collect(Collectors.groupingBy(  
 Employee::getDepartment,  
 Collectors.collectingAndThen(  
 Collectors.maxBy(Comparator.comparingDouble(Employee::getSalary)),  
 Optional::get  
 )*

*));*

* *We use the Collectors.groupingBy collector to group employees by their department. The first argument Employee::getDepartment specifies how to extract the department name as the key.*
* *Within groupingBy, we use Collectors.collectingAndThen to further process the grouped elements. Here, we use Collectors.maxBy to find the employee with the maximum salary within each group (department).* ***Collectors.collectingAndThen is used when we use maxBy and minBy inside it. minBy and maxBy both return optional and to get value out of this optional we use collectingAndThen.***
* *Collectors.maxBy uses Comparator.comparingDouble(Employee::getSalary) to compare employees based on their salaries and select the one with the maximum salary.*
* *Finally, Collectors.collectingAndThen converts the Optional<Employee> result of Collectors.maxBy to an Employee using Optional::get.*

*Map<String, Employee> topSalaryByDepartment = employeeList.stream()  
 .collect(Collectors.toMap(  
 Employee::getDepartment,  
 e -> e,  
 (e1, e2) -> e1.getSalary() > e2.getSalary() ? e1 : e2*

*));*

*The toMap collector in Java's Stream API is used to transform elements of a stream into a Map. In the code provided, we are using the toMap collector to group employees by their department and select the employee with the highest salary for each department. Let's break down the parameters of the toMap function:*

*Key Mapper Function (Employee::getDepartment): This parameter specifies how to extract the keys (i.e., the department names) from the stream elements (employees). In our case, we use the Employee::getDepartment method reference to get the department name of each employee as the key.*

*Value Mapper Function (e -> e): This parameter specifies how to extract the values (i.e., the employees themselves) from the stream elements. In this code, we simply use a lambda expression (e -> e) to map each employee to itself.*

*Merge Function ((e1, e2) -> e1.getSalary() > e2.getSalary() ? e1 : e2): This parameter is crucial when two elements in the stream have the same key (i.e., they belong to the same department). It specifies how to resolve conflicts in case of key collisions. In our case, if two employees have the same department, we compare their salaries using the lambda expression (e1, e2) -> e1.getSalary() > e2.getSalary() ? e1 : e2. This lambda function checks which of the two employees has a higher salary and selects that one as the value for the key (department).*

*Here's a step-by-step explanation of how the toMap collector works in the provided code:*

*The stream of employees (employeeList.stream()) is processed.*

*For each employee in the stream, the key mapper function (Employee::getDepartment) extracts the department name, and the value mapper function (e -> e) keeps the employee itself.*

*As the stream is processed, the toMap collector collects these key-value pairs into a Map.*

*When it encounters employees with the same department (key collision), it uses the merge function ((e1, e2) -> e1.getSalary() > e2.getSalary() ? e1 : e2) to determine which employee to keep based on their salaries. The one with the higher salary is selected.*

*The result is a Map<String, Employee> where each department name is associated with the employee who has the highest salary in that department.*

1. **Write A Java Program to group the words in sentence by length using java 8 features**

Map<Integer, List<String>>lengthMap = words.stream().collect(collectors.groupingBy(String::length))

1. **Practical Implementation of How Parallel Stream is different from Sequential Stream.**  
   List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9);  
   //Sequential Stream  
   *numbers.stream().forEach(num -> {  
    System.out.println(e+ “ ” + Thread.currentThread().getName());  
   });*

//Parallel Stream

*numbers.parallelStream().forEach(num -> {  
 System.out.println(e+ “ ” + Thread.currentThread().getName());  
});*

1. . List <Employee> employees = new ArrayList<Employee>();  
    employees.add(new Employee(“ABC”,30 ,“Female”, “HR”));  
    employees.add(new Employee(“PQR”,25 ,“Male”, “IT”));  
    employees.add(new Employee(“LMN”, 30, “Male”, “HR”));  
    employees.add(new Employee(“XYZ”, 28, “Female”, “IT”));
2. **Find List of distinct department.**

*List<String> distinctDepartments = employeeList.stream()*

*.map(Employee::getDepartment)*

*.distinct()*

*.collect(Collectors.toList());*

1. **Find Count Of Employees working in each department**  
   *Map<String, Long> departmentCounts = employeeList.stream()*

*.collect(Collectors.groupingBy(*

*Employee::getDepartment,*

*Collectors.counting()*

*));*

1. **Find Average age of Male Employee and female employee.**  
   *Map<String, Double> avgAge = employees.stream().collect(Collectors.groupingBy(Employee::getGender, Collectors.averagingDouble(Employee::getAge)));*
2. **Find Average Age Of all Employees.**  
   *employees.stream().collect(Collectors.averagingDouble(Employee::getAge));*
3. **From a list of numbers find the list of non duplicate integers.**

List<Integer> numbers = Arrays.asList(10,20,27,25,20,25,30);  
Set <Integer> hs = new HashSet<Integer>();  
numbers.stream().filter(num ->hs.add (num) ).collect(Collectors.toList());

1. **Find the list of duplicate integers from list.**

in above example if we add this particular condition  
numbers.stream().filter(num->!hs.add(num)).collect(Collectors.toList());

1. **Sort the list of Integers in descending order.**

numbers.stream().sorted(Collections.reverseOrder()).collect(Collectors.toList());  
or  
List<Integer> sortedNumbers = numbers.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());

1. **For given 2 arrays, find the common number between 2 arrays without using collection, filter and using only stream. Show single instance of duplicate elements.**

int firstArray[] = {1,2,2,1};  
int secondArray[] = {2,2}; Output: [2] **OR**  
intfirstArray[] = {4,9,5};  
int secondArray[] = {9,4,9,8,4} Output : [9,4]Arrays.stream(firstArray).filter(num1->Arrays.stream(secondArray).**anyMatch**(num2->num1==num2))  
.distinct()  
.forEach(System.out::println)  
adding distinct will remove duplicacy

1. **Check if all list elements are divisible by 2. If yes, print “all the numbers are divisible by 2” else print “all the numbers are not divisible by 2”, using java 8 features and without using filter.**

List<Integer> intList = Arrays.asList(2,4,5,6,8);

boolean areDivisibleByTwo = intList.stream().**allMatch**(n->n%2==0);

if(areDivisibleByTwo){

SYSO(“All the numbers are divisible by 2”);

}else {

SYSO(“All the numbers are not divisible by 2”)

}

1. **Check if atleast one list elements is divisible by 2. If yes, print “atleast one number is divisible by 2” else print “all the numbers are not divisible by 2”, using java 8 features and without using filter.**

List<Integer> intList = Arrays.asList(2,4,5,6,8);

boolean isDivisibleByTwo = intList.stream().**anyMatch**(n->n%2==0);

if(isDivisibleByTwo){

SYSO(“Atleast one number is divisible by 2”);

}else {

SYSO(“None of the numbers are divisible by 2”)

}

1. **Find the frequency of each word in given list and print only those words which are having count >=2 using java 8 features.**

Input = [“AA”, “BB”, “B” , “D”,“AA”, “DD”, “CC”, “DD”];

Output = AA, DD  
List<String> words = Arrays.asList(“AA” ”, “BB”, “B” , “D”,“AA”, “DD”, “CC”, “DD”);  
Map<String, Long> wordFrequency = words.stream()  
 .collect(Collectors.groupingBy(  
 word -> word,  
 Collectors.counting()));

List<String> result = wordFrequency.entrySet()  
 .stream()  
 .filter(entry -> entry.getValue() >= 2)  
 .map(Map.Entry::getKey).collect(Collectors.toList());

1. **In the given list of words find the count of words whose length is greater than 5**

Int count = wordlist.stream().filter(word -> word.length() > 5).count();

1. **How to get Even and Odd Number from the list using Java 8.**

List<integer> nums = Arrays.asList(1,2,3,4,5,6);

Map <Boolean, List<Integer>> resultMap =

nums.stream().collect(Collectors.partioningBy(n -> n%2 ==0));

List<Integer> evenList = resultMap.get(true);

List<Integer> oddList = resultMap.get(false);

Or

Map<String, List<Integer>> evenOddGroups = numbers.stream()

.collect(Collectors.groupingBy(

num -> num % 2 == 0 ? "Even" : "Odd",

Collectors.toList()

));

1. **From a string find the list of duplicate characters using Java 8 Features.**

List<Character> duplicates = inputString.chars() // Step 1: Convert to character stream

.mapToObj(c -> (char) c) // Convert to Character objects

.collect(Collectors.groupingBy(

c -> c, // Step 2: Group by character

Collectors.counting() // Count occurrences

))

.entrySet()

.stream()

.filter(entry -> entry.getValue() > 1) // Step 3: Filter duplicates

.map(Map.Entry::getKey) // Step 4: Map to characters

.collect(Collectors.toList()); // Collect duplicates into a list

1. **Get List of all products that belongs to Books Category and product price is greater than 200.**

Products.stream().filter(product -> product.getBook().equalsIgnoreCase(“Books”))

.filter(book -> book.getPrice() > 200).forEach(System.out :: println)

1. **Get Cheapest Product in “Phone” Category**

Product cheapestPhone = Products.stream()

.filter(product -> products.getBook().equalsIgnoreCase(“Phone”))

.min(Comparator.comparing(Product::getPrice)).get();

**1. Given two arrays. Find common numbers from these two arrays/ intersection b/w 2 arrays.**Set <Integer> s = new HashSet<>();  
for (int i = 0; i<arr1.length;i++){s.add(arr[i]);}  
for(int i = 0;i<arr2.length;i++ ){ if(s.contains(arr2[i])){System.out.println(“ ”+arr2[i]);} }

**2. Given two arrays. Find uncommon numbers from these two arrays/ union b/w 2 arrays.**  
Set <Integer> s = new HashSet<>();  
for (int i = 0; i<arr1.length; i++){ s.add(arr[i]); }  
for(int i = 0; i<arr2.length; i++){ s.add(arr[i]); }  
System.out.println(s);

**3. From a given string we need to count the number of special character and the string without special character.**

String s = “Shivam @123!”;  
String stringWithSpecialCharacterRemoved = “”;  
int count = 0;  
for (int i = 0; i < s.length(); i++) {

if(!Character.isDigit(s.charAt(i)) && !Character.isLetter(s.charAt(i)) && !Character.isWhiteSpace(s.charAt(i))) {count++}

}

else {

stringWithSpecialCharacterRemoved = stringWithSpecialCharacterRemoved+s.charAt(i);

}  
System.out.print(“Count is ”+count);

**4. 2 Strings are given to us. How to find whether these both strings are rotations of each other.**use this following condition 🡪 (str1+str2).subString(str2)

**5. First n natural numbers are present in an array except one natural number, which is missing. Write a program to find that natural number.**  
int sumOfNaturalNumber = (n\*(n+1))/2;  
int actualSum = 0;  
for (int i = 0;i<arr1.length;i++){ actualSum = actualSum+arr1[i];}  
int missingNumber = sumOfNaturalNumber – actualNumber;

**6. We are given a String(str) and we have to remove all the occurrence of a particular character(c).**

String newString = “”;  
for (int i =0; i < str.length;i++){  
 if(str.charAt(i)!=c) {newString = newString+str.charAt(i);}  
}  
System.out.println(“”+newString);

**7. We are given two arrays. Find whether these 2 arrays are same or not?**

if (arr1.length()! = arr2.length() ) return “Arrays are not same”;  
Set <Integer> hs = new HashSet<integer>();  
for(int i = 0;i< arr1.length;i++) { hs.add(arr[i]);}  
bool same = true;  
for (int i = 0; i< arr2.length; i++) { if(!hs.contains(arr2[i])){same = false;break;} }  
If(same == false){return “Array not same”;}

**8. Find Palindrome words from sentence.**

String str = “My name is nitin and I can speak malayalam”;  
String Words = str.split(“ ”);  
public boolean isPallindrome(String s) {

int i1 = 0;

int i2 = s.length()-1;

while(i1<i2) {

if(s.charAt(i1)!=s.charAt(i2)) return false;

i1++;

i2--;

}

return true;

}

Words.stream.forEach (word -> {  
 If(isPallindrome(word)) { System.out.println(word); }  
});

**9. Write a code to generate random numbers.**

Here we are going to use Java’s own Random class.   
Random random = new Random();  
random.ints().forEach(System.out::println);

**Write a program to write only 10 numbers.**  
random.ints().limit(10).forEach(System.out::println);

**Write a program to print 10 number between 1 & 100.**  
random.ints(1,100).limit(10).forEach(System.out::println);

**Write a program to sort the above randomly generated numbers.**  
random.ints(1,100).limit(10).sorted().forEach(System.out::println);

**DS Algo Codes**

public void **recursivePreOrder**(TreeNode root) {  
 if(root == null) return;  
 System.out.print(root.data+" ");  
 recursivePreOrder(root.left);  
 recursivePreOrder(root.right);  
}

public void **recursiveInOrder**(TreeNode root) {  
 if(root == null) return;  
 recursiveInOrder(root.left);  
 System.out.print(root.data+" ");  
 recursiveInOrder(root.right);  
}

public void **recursivePostOrder**(TreeNode root) {  
 if(root == null) return;  
 recursivePostOrder(root.left);  
 recursivePostOrder(root.right);  
 System.out.print(root.data+" ");  
 }

public void **iterativePreOrder**(TreeNode root) {  
 if(root == null) return;  
 Stack<TreeNode> stack = new Stack<>();  
 stack.push(root);  
 while(!stack.isEmpty()) {  
 TreeNode current = stack.pop();  
 System.out.print(current.data+ " ");  
 if(current.right!=null)stack.push(current.right);  
 if(current.left!=null)stack.push(current.left);  
 // here we are pushing right tree firstly into stack because we want to traverse left tree first.

}

}

public void **iterativeInOrder**(TreeNode root) {  
 if(root == null) return;  
 Stack<TreeNode> stack = new Stack<>();  
 TreeNode current = root;  
 while(!stack.isEmpty() || current!= null){  
 if(current != null){  
 stack.push(current);  
 current = current.left;  
 } else { // if temp == null but stack is not empty  
 current = stack.pop();  
 System.out.print(current.data+" ");  
 current = current.right;  
 } } }

public void **iterativePostOrder**(TreeNode root) {  
 if(root == null) return;  
 Stack<TreeNode> stack = new Stack<>();  
 TreeNode current = root;  
 while(!stack.isEmpty() || current!= null) {  
 if(current != null) { // traversing up to left most part  
 stack.push(current);  
 current = current.left;  
 } else {  
 TreeNode temp = stack.peek().right;  
 // checking whether the last pushed element into stack has right node or not  
 if(temp != null) {  
 // this will be executed when left subtree is null and right subtree is present  
 current = temp;  
 } else {  
 // this situation arises when last element pushed into stack is leaf node  
 temp = stack.pop();  
 System.out.print(temp.data + " ");  
 // Using Second while loop we are checking whether the last popped leaf node (temp) was left node or right node  
 // Condition in while loop will be true when we have visited both left and right node and want to print the parent.  
 // temp == stack.peek().right will be true when temp is right child of its parent.  
 while(!stack.isEmpty() && temp == stack.peek().right) {  
 temp = stack.pop();  
 System.out.print(temp.data+ " ");  
 } } } } }

public void **levelOrderTraversal**() {  
 if(root == null) return;  
 Queue<TreeNode> queue = new LinkedList<>();  
 queue.offer(root); // offer adds an element at the end of queue  
 while(!queue.isEmpty()) {  
 TreeNode current = queue.poll(); // poll method removes first element from queue  
 System.out.print(current.data+" ");  
 if(current.left!=null)queue.offer(current.left);  
 if(current.right!=null)queue.offer(current.right);  
 }  
}

public int **findMax**(TreeNode parent){  
 if(parent == null) return Integer.MIN\_VALUE;  
 int leftSubTreeMaxValue = findMax(parent.left);  
 int rightSubTreeMaxValue = findMax(parent.right);  
 int maxValue = parent.data;  
 if(maxValue<leftSubTreeMaxValue ) maxValue = leftSubTreeMaxValue;  
 if(maxValue<rightSubTreeMaxValue) maxValue = rightSubTreeMaxValue;  
 return maxValue;  
}

public void **BFS**(Character source) {  
 Set <Character> visitedNodes = new HashSet<>();  
 Queue <Character> queue = new LinkedList<>();  
 queue.add(source);  
 visitedNodes.add(source);  
 while(!queue.isEmpty()) {  
 Character current = queue.poll();  
 System.out.print(current+" ");  
 List<Character> neighbours = adjacencyList.get(current);  
 if(neighbours != null && !neighbours.isEmpty()) {  
 for (Character neighbour : neighbours) {  
 if(!visitedNodes.contains(neighbour)) {  
 queue.add(neighbour);  
 visitedNodes.add(neighbour);  
 } } } } }

private void **DFS**(Character source) {  
 Set<Character> visitedNodes = new HashSet<>();  
 Stack<Character> stack = new Stack<>();  
 stack.push(source);  
 visitedNodes.add(source);  
 while(!stack.isEmpty()) {  
 Character current = stack.pop();  
 System.out.print(current+" ");  
 List<Character> neighbours = adjacencyList.get(current);  
 if(neighbours!=null && !neighbours.isEmpty()) {  
 for (Character neighbour: neighbours) {  
 if(!visitedNodes.contains(neighbour)) {  
 stack.push(neighbour);  
 visitedNodes.add(neighbour);

} } } } }