STREAMS

Interview questions on streams?

Explain predicate consumer producer ?

In Java, **Predicate**, **Consumer**, and **Producer** (Supplier) are functional interfaces from the java.util.function package. They are key components of **functional programming** and are mainly used in **lambda expressions** and **stream processing**.

**1️⃣ Predicate (Conditional Checking)**

A **Predicate** represents a boolean-valued function (true or false). It is mainly used for **filtering** and **conditional checks**.

**Functional Interface Definition**

@FunctionalInterface

public interface Predicate<T> {

boolean test(T t);

}

**Use Case**

* Used for **filtering collections**.
* Commonly used in **Streams** and **if conditions**.

**Example: Filtering Even Numbers**

import java.util.function.Predicate;

public class PredicateExample {

public static void main(String[] args) {

Predicate<Integer> isEven = num -> num % 2 == 0;

System.out.println(isEven.test(10)); // true

System.out.println(isEven.test(15)); // false

}

}

**Example: Filtering a List**

import java.util.List;

import java.util.function.Predicate;

import java.util.stream.Collectors;

public class PredicateListExample {

public static void main(String[] args) {

List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);

Predicate<Integer> isEven = num -> num % 2 == 0;

List<Integer> evenNumbers = numbers.stream()

.filter(isEven)

.collect(Collectors.toList());

System.out.println(evenNumbers); // [2, 4, 6]

}

}

**2️⃣ Consumer (Consumes Data)**

A **Consumer** represents an operation that takes an input and **performs an action** but does not return a value.

**Functional Interface Definition**

@FunctionalInterface

public interface Consumer<T> {

void accept(T t);

}

**Use Case**

* Used in **logging, printing, or modifying** objects.
* Commonly used in **forEach()** operations.

**Example: Printing Elements**

import java.util.function.Consumer;

public class ConsumerExample {

public static void main(String[] args) {

Consumer<String> print = s -> System.out.println("Hello, " + s);

print.accept("Alice"); // Output: Hello, Alice

print.accept("Bob"); // Output: Hello, Bob

}

}

**Example: Using Consumer with forEach**

import java.util.List;

import java.util.function.Consumer;

public class ConsumerForEachExample {

public static void main(String[] args) {

List<String> names = List.of("Alice", "Bob", "Charlie");

Consumer<String> printName = name -> System.out.println("Name: " + name);

names.forEach(printName);

}

}

**3️⃣ Supplier (Produces Data)**

A **Supplier** (also called a **Producer**) represents a function that **does not take input** but **returns a value**.

**Functional Interface Definition**

@FunctionalInterface

public interface Supplier<T> {

T get();

}

**Use Case**

* Used for **lazy initialization** (e.g., getting default values).
* Useful in **factory methods**.
* Often used in **Streams** to generate values dynamically.

**Example: Generating Random Numbers**

import java.util.function.Supplier;

import java.util.Random;

public class SupplierExample {

public static void main(String[] args) {

Supplier<Integer> randomNumber = () -> new Random().nextInt(100);

System.out.println(randomNumber.get()); // e.g., 42

System.out.println(randomNumber.get()); // e.g., 87

}

}

**Example: Supplying a Default Value**

import java.util.function.Supplier;

public class DefaultValueExample {

public static void main(String[] args) {

Supplier<String> defaultMessage = () -> "Default Message";

System.out.println(defaultMessage.get()); // Output: Default Message

}

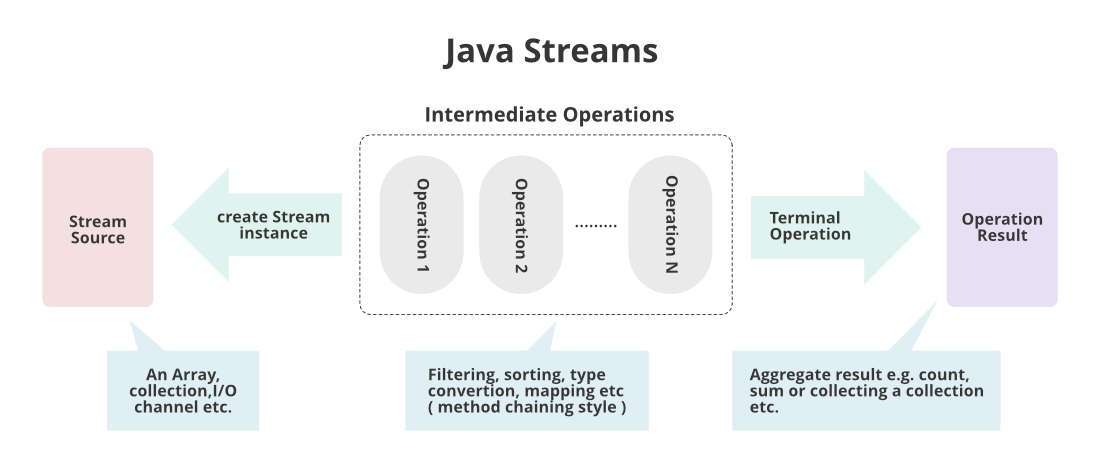
}

**🔥 Comparison Table**

| **Interface** | **Input** | **Output** | **Usage** |
| --- | --- | --- | --- |
| **Predicate** | T | boolean | Used for **filtering** and **conditional checks** |
| **Consumer** | T | void | Performs an **action** like **printing, modifying** objects |
| **Supplier** | None | T | Produces **data** (e.g., generating values, factory methods) |
| **Difference Between Stream API and Collection API in Java**  **🔹 Overview**  **Both Stream API and Collection API are used for handling data in Java, but they serve different purposes.**   * **Collection API: Stores and manages data in-memory (Lists, Sets, Maps, etc.).** * **Stream API: Processes data in a functional & lazy manner, often used for transformations and filtering.**   **🔹 Key Differences**   | **Feature** | **Collection API** | **Stream API** | | --- | --- | --- | | **Purpose** | **Stores and manipulates data** | **Performs data processing operations** | | **Type** | **Eager (computes results immediately)** | **Lazy (processed only when needed)** | | **Modification** | **Supports adding/removing elements** | **Cannot modify data source** | | **Iteration** | **Uses Iterator or forEach** | **Uses internal iteration** | | **Parallel Processing** | **Not designed for parallelism** | **Supports parallel processing (parallelStream())** | | **Intermediate Operations** | **Not applicable** | **Supports filter, map, sorted (lazy execution)** | | **Terminal Operations** | **Not applicable** | **Collect, forEach, reduce (triggers execution)** | | **Memory Usage** | **Holds elements in memory** | **Does not store data, processes on demand** | |  |  |  |

**Java 8 Streams: Level-Wise Interview Questions**

The Java 8 Streams API is a powerful feature for processing collections and sequences of elements. Here are **level-wise interview questions** categorized as beginner, intermediate, and advanced, with answers to help you understand the key concepts.



**Beginner Level Questions**

**1. What is a Stream in Java 8?**

**Answer:**

* A Stream in Java 8 is a sequence of elements that supports functional-style operations such as filtering, mapping, and reducing.
* It is not a data structure but a pipeline of computations on data.

**2. How is a Stream different from a Collection?**

**Answer:**

* **Stream**:
  + Provides functional programming methods.
  + Does not store data; processes data on demand.
  + Supports lazy evaluation.
* **Collection**:
  + Stores data in memory.
  + Used to hold and manipulate data.
  + Eagerly evaluates data.

**3. How do you create a Stream in Java 8?**

**Answer:** Streams can be created in multiple ways:

* From a collection: List<String> list = Arrays.asList("a", "b"); Stream<String> stream = list.stream();
* From arrays: Stream<int[]> stream = Arrays.stream(array);
* Using Stream.of(): Stream<Integer> stream = Stream.of(1, 2, 3);

**4. What are intermediate and terminal operations in Streams?**

**Answer:**

* **Intermediate Operations:** Transform the stream (e.g., filter(), map()), producing another stream.
* **Terminal Operations:** Consume the stream, producing a result or a side effect (e.g., collect(), forEach()).

**5. What is lazy evaluation in Streams?**

**Answer:**

* Intermediate operations are not executed until a terminal operation is invoked, enabling optimization by processing elements as needed.

**Intermediate Level Questions**

**6. What is the difference between map() and flatMap() in Streams?**

**Answer:**

* map(): Transforms each element of the stream into another element. It produces a one-to-one mapping.
* flatMap(): Flattens the stream of streams into a single stream. It is used when each element maps to multiple elements.

**Example:**

List<String> list = Arrays.asList("Hello", "World");

list.stream().map(s -> s.split("")).forEach(System.out::print); // Outputs: [Ljava.lang.String;@...

list.stream().flatMap(s -> Arrays.stream(s.split(""))).forEach(System.out::print); // Outputs: HelloWorld

**7. What is the difference between findFirst() and findAny()?**

**Answer:**

* findFirst(): Returns the first element of the stream.
* findAny(): Returns any element of the stream, which is useful in parallel streams for faster processing.

**8. How does filter() work in Streams?**

**Answer:** The filter() method takes a **Predicate** and returns a new stream containing elements that match the given condition.

**Example:**

Stream<Integer> stream = Stream.of(1, 2, 3, 4);

stream.filter(n -> n % 2 == 0).forEach(System.out::println); // Outputs: 2, 4

**9. What is the use of collect() in Streams?**

**Answer:**

* collect() is a terminal operation used to accumulate elements from a stream into a collection, map, or other forms.

**Example:**

List<Integer> list = Stream.of(1, 2, 3).collect(Collectors.toList());

**10. What are some common Collectors used in Streams?**

**Answer:**

* toList(): Collects elements into a List.
* toSet(): Collects elements into a Set.
* joining(): Joins elements into a single String.
* groupingBy(): Groups elements by a classifier function.
* partitioningBy(): Partitions elements into two groups based on a predicate.

**Advanced Level Questions**

**11. What is the difference between reduce() and collect()?**

**Answer:**

* reduce(): Performs a reduction on the stream elements, producing a single result (e.g., sum, min, max).
* collect(): Gathers elements into a mutable container (e.g., List, Set, Map).

**Example:**

int sum = Stream.of(1, 2, 3).reduce(0, Integer::sum); // Outputs: 6

**12. How do Streams handle parallel processing?**

**Answer:** Streams can be converted to parallel streams using parallelStream(). Parallel streams divide the data into multiple chunks and process them concurrently, utilizing multiple CPU cores.

**Example:**

List<Integer> list = Arrays.asList(1, 2, 3, 4);

list.parallelStream().forEach(System.out::println); // May print in any order

**13. What is the purpose of peek() in Streams?**

**Answer:**

* peek() is used to perform a side effect for debugging or logging during intermediate operations. It is primarily used to inspect elements.

**Example:**

Stream.of(1, 2, 3)

.peek(n -> System.out.println("Processing: " + n))

.map(n -> n \* n)

.forEach(System.out::println);

**14. Explain short-circuit operations in Streams.**

**Answer:** Short-circuit operations stop processing the stream as soon as the required result is found. Examples include:

* findFirst()
* findAny()
* anyMatch()
* allMatch()
* noneMatch()

**15. Can Streams be reused? Why or why not?**

**Answer:** No, a Stream cannot be reused. Once a terminal operation is invoked, the Stream is considered consumed. Attempting to reuse it will throw an IllegalStateException.

**Expert Level Questions**

**16. What are Infinite Streams, and how are they created?**

**Answer:**

* Infinite streams are streams that generate an unbounded sequence of elements.
* They can be created using Stream.iterate() or Stream.generate().

**Example:**

Stream<Integer> infiniteStream = Stream.iterate(1, n -> n + 1);

infiniteStream.limit(5).forEach(System.out::println); // Outputs: 1, 2, 3, 4, 5

**17. What is the difference between limit() and skip()?**

**Answer:**

* limit(n): Truncates the stream to contain only the first n elements.
* skip(n): Discards the first n elements from the stream.

**18. What is the significance of the unordered() method in Streams?**

**Answer:** The unordered() method removes the encounter order of elements in a stream, which can improve performance in parallel processing.

**19. How does groupingBy() differ from partitioningBy() in Collectors?**

**Answer:**

* groupingBy(): Groups elements into multiple categories based on a classifier function.
* partitioningBy(): Divides elements into two groups based on a predicate.

**20. How do Streams handle exceptions?**

**Answer:** Streams do not have built-in support for checked exceptions. Developers can handle exceptions by:

* Wrapping the operation in a try-catch block.
* Writing custom utility methods for exception handling.

**Conclusion**

These questions, organized by difficulty, cover the most common concepts of Java 8 Streams. By understanding 80% of these, you’ll be well-prepared for most interviews! Let me know if you'd like to dive deeper into any specific topic.

PRACTICAL EXAMPLES:

1.What are streams?

Java Streams are part of the Java 8 Stream API, which provides a functional programming approach to process collections of data (e.g., lists, sets, arrays) in a declarative and pipeline-like manner.

Streams will be used with foreach.

If once you iterate through streams it cannot be used for next iteration.

If you want to use two times create a object and use it two times.

Ex:parllelstream,findfirst etc.

For each is for iteration.

Stream is a pillar which holds many sub methods. Like filter etc.

2.\*\*Print all elements:

l2.stream().forEach(i->System.out.println(i));

3.\*\*print 5 element.(use ifpresent for this).

L2.stream().skip(4).ifpresent(i->sop(i)).

4.\*\*print all elements after 2nd:

l2.stream().skip(2).forEach(i->System.out.println(i));

5.write if condition in streams:(filter)://output <input

l1.stream().filter(name->name.charAt(0)=='s').forEach(i->System.out.println(i));

use map: (input==output):

6.\*\*\* using map and filter:

l2.stream().filter(i->i%2==0).map(i->i\*i).forEach(i->System.*out*.println(i));

7.Print max elements (single element):

l2.stream().sorted().skip(1).findFirst().ifPresent(i->System.*out*.println(i));

8.what is flatmap ?

flat map:

9.what are collectors?

tocollectors:

store in list and map:

find duplicates using java 8 streams:

explain some intermediate operators:

10.what is filter?

FILTER:

**Purpose**: Used to filter elements in a stream based on a given condition (predicate). It only keeps elements that satisfy the condition.

\*\*if you want to write condition write it in filter using streams.

 intList.stream().filter(

          element -> (element%2==0)

        )

        .forEach(

          element -> System.out.print(element+ " ")

        );

11.what is map?

MAP:

Transforms each element of a stream into another form (mapping). You provide a function that defines how each element is transformed.

It transforms the values.

If you have 10 values it will transform all the 10 values ex:i\*i

It multiplies every value by itself and provides us.a

import java.util.Arrays;

import java.util.List;

public class MapExample {

public static void main(String[] args) {

List<Integer> l2 = Arrays.asList(1, 2, 3, 4);

// Map each integer to its square

l2.stream()

.map(i -> i \* i) // Transform each number to its square

.forEach(i->System.out::println(i)); // Print the result

}

}

What is FLAT MAP?

The **flatMap** operation is an **intermediate operation** in Java Streams. It transforms each element of a stream into a new stream, then combines all those streams into a single flat stream.

Ex:

List<List<Integer>> a3=Arrays.*asList*(Arrays.*asList*(1,2),  
Arrays.*asList*(2,3),  
Arrays.*asList*(3,4));

a3.stream().flatMap(Collection::stream).map(i->i).forEach(i-> System.*out*.println(i));

.What is distinct?

A,

It removes the duplicates.

List<Integer> numbers = List.of(1, 2, 2, 3, 4, 4, 5);

List<Integer> uniqueNumbers = numbers.stream()

.distinct() // Removes duplicates

.collect(Collectors.toList());

System.out.println(uniqueNumbers); // Output: [1, 2, 3, 4, 5]

what is sort in streams?

A,

It sorts the elements.

List<Integer> numbers = List.of(5, 2, 9, 1, 7);

// Using sorted

numbers.stream()

.sorted()

.forEach(System.out::print); // Output: 12579

// Using TreeSet

Set<Integer> sortedSet = numbers.stream()

.collect(Collectors.toCollection(TreeSet::new));

System.out.println(sortedSet); // Output: [1, 2, 5, 7, 9]

Explain comperator function in sorting streams based on length?

List<String> ll1=l1.stream().sorted(Comparator.*comparing*(String::length)).collect(Collectors.*toList*());  
System.*out*.println(ll1);

For reversed order:

List<String> ll1=l1.stream().sorted(Comparator.*comparing*(String::length).reversed()).collect(Collectors.*toList*());  
System.*out*.println(ll1);

What is skip?

Ans,

The **skip** method in Java Streams is an **intermediate operation** that skips a specified number of elements from the stream and returns a new stream with the remaining elements.

Example:

List<String> result = words.stream() .skip(2) // Skips the first 2 elements .collect(Collectors.toList());

System.out.println(result);

What is limit?

It limits elements to print.

List<String> result = words.stream() .limit(3) // Limiting the stream to the first 3 elements .collect(Collectors.toList());

13.explain some methods used in streams?

**1. sorted():**

* **Functionality**: Sorts the elements of the stream in natural order (ascending for numbers, lexicographical for strings).
* **Theory**: The sorted method returns a new stream with elements arranged in natural or custom order specified by a comparator.
* **Example**:

java

Copy code

l2.stream().sorted().forEach(System.out::println);

**Output (for l2 = [3, 1, 2])**:

Copy code

1

2

3

**2. findFirst():**

* **Functionality**: Retrieves the first element of the stream.
* **Theory**: The findFirst method returns an Optional containing the first element of the stream or an empty Optional if the stream is empty.
* **Example**:

java

Copy code

l2.stream().sorted().findFirst().ifPresent(System.out::println);

**Output (for l2 = [3, 1, 2])**:

Copy code

1

**3. skip(n):**

* **Functionality**: Skips the first n elements of the stream and processes the remaining elements.
* **Theory**: The skip method creates a stream that discards the first n elements while retaining all subsequent elements.
* **Example**:

java

Copy code

l2.stream().skip(2).forEach(System.out::println);

**Output (for l2 = [3, 1, 2])**:

Copy code

2

**4. forEach():**

* **Functionality**: Performs an action (e.g., printing) on each element of the stream.
* **Theory**: The forEach method iterates over all elements in the stream and executes the provided consumer function.
* **Example**:

java

Copy code

l2.stream().forEach(System.out::println);

**Output (for l2 = [3, 1, 2])**:

Copy code

3

1

2

**5. filter(Predicate):**

* **Functionality**: Selects elements from the stream that match the given predicate.
* **Theory**: The filter method creates a stream containing only elements that satisfy the predicate condition.
* **Example**:

java

Copy code

l1.stream().filter(name -> name.startsWith("s")).forEach(System.out::println);

**Output (for l1 = ["sun", "moon", "star"])**:

Copy code

sun

star

**6. max(Comparator):**

* **Functionality**: Finds the maximum element in the stream based on the comparator.
* **Theory**: The max method returns an Optional containing the largest element or an empty Optional if the stream is empty.
* **Example**:

java

Copy code

l2.stream().max(Integer::compareTo).ifPresent(System.out::println);

**Output (for l2 = [3, 1, 2])**:

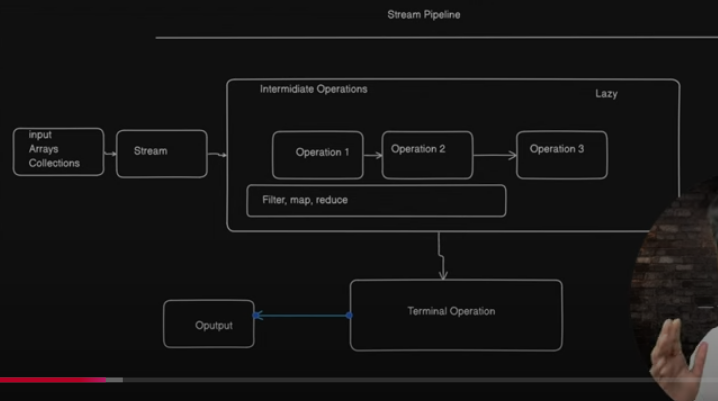
Copy code

3

4

14.what are intermediate and terminal operators ?

Ans,



**Intermediate Operators**

* **Definition**: Intermediate operators transform a stream into another stream. These operations are lazy; they don’t perform processing until a terminal operation is invoked.
*  Do not execute immediately; instead, they prepare a pipeline of operations.
*  Always return a new stream

**Examples**:

* **filter()**: Filters elements based on a predicate.

java

Copy code

List<Integer> numbers = List.of(1, 2, 3, 4, 5);

numbers.stream().filter(n -> n % 2 == 0).forEach(System.out::println); // Prints: 2, 4

* **map()**: Transforms each element.

java

Copy code

List<String> names = List.of("Alice", "Bob", "Charlie");

names.stream().map(String::toUpperCase).forEach(System.out::println); // Prints: ALICE, BOB, CHARLIE

* **sorted()**: Sorts elements.
* **distinct()**: Removes duplicates.
* **limit()**: Limits the size of the stream.
* **skip()**: Skips a given number of elements.

**Terminal Operators**

**Definition**: Terminal operators trigger the execution of the stream pipeline. They produce a result or a side-effect.

 Consume the stream, so it cannot be reused.

 Return a result (e.g., a value, a collection, etc.) or have a side-effect (e.g., printing to console).

**Examples**:

* **forEach()**: Applies an action to each element.

java

Copy code

List<Integer> numbers = List.of(1, 2, 3);

numbers.stream().forEach(System.out::println); // Prints: 1, 2, 3

* **collect()**: Collects elements into a collection.

java

Copy code

List<Integer> evenNumbers = numbers.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList());

* **reduce()**: Reduces the elements to a single value.

java

Copy code

int sum = numbers.stream().reduce(0, Integer::sum); // Sum = 6

* **count()**: Counts the number of elements.
* **anyMatch(), allMatch(), noneMatch()**: Tests elements against a predicate.

15.convert arrays into streams ?

int[] a={1,3,11,2};  
IntStream a1=Arrays.*stream*(a);

16.convert list into streams?

List<Integer> l2= Arrays.*asList*(8,4,77,2,1);

17.explain some terminal operators?

Reduce:

The **reduce** method in Java Streams is a **terminal operation** used to combine elements of a stream into a single result by repeatedly applying a specified operation.

Ex:

Integer sum=l2.stream().reduce(0,(a2,b1)->a2+b1);  
System.*out*.println(sum);

Group BY:

the groupBy() operation is typically used to group elements of a stream based on a specified classifier function. This is not a direct method of the Stream API but is part of the Collectors utility class in the java.util.stream package, specifically using Collectors.groupingBy().

Ex-1:

List<Employee> employees = Arrays.*asList*(  
 new Employee("Alice", "HR", 5000),  
 new Employee("Bob", "IT", 4000),  
 new Employee("Charlie", "IT", 3500),  
 new Employee("Dave", "IT", 4500),  
 new Employee("Eve", "Finance", 3000)  
);

Map<String,List<Employee>> hm=employees.stream().collect(Collectors.*groupingBy*(i->i.getDepartment()));  
System.*out*.println(hm);

Ex-2:

Counting in groupby class:

Map<String, Long> studentCountByGrade = students.stream()

.collect(Collectors.groupingBy(Student::getGrade, Collectors.counting()));//can use I arrow operator.

Ex-3:

**Example: Collect Student Names in Each Grade**

Group students by grade but collect only their names in a List:

java

Copy code

Map<String, List<String>> studentNamesByGrade = students.stream()

.collect(Collectors.groupingBy(

Student::getGrade,

Collectors.mapping(Student::getName, Collectors.toList())

));

Ex-4:

**Example: Grouping Students by Grade and Counting Them**

**Input:**

Imagine a list of students with their names and grades:

java

Copy code

class Student {

private String name;

private String grade;

// Constructor, Getters, and Setters

public Student(String name, String grade) {

this.name = name;

this.grade = grade;

}

public String getName() {

return name;

}

public String getGrade() {

return grade;

}

}

List<Student> students = List.of(

new Student("Alice", "A"),

new Student("Bob", "B"),

new Student("Charlie", "A"),

new Student("Dave", "C"),

new Student("Eve", "B")

);

**Code: Group Students by Grade**

Group the students into a map where the key is the grade, and the value is a list of students in that grade.

java

Copy code

Map<String, List<Student>> studentsByGrade = students.stream()

.collect(Collectors.groupingBy(Student::getGrade));

**Output:**

The map studentsByGrade will look like this:

java

Copy code

{

"A" -> [Alice, Charlie],

"B" -> [Bob, Eve],

"C" -> [Dave]

}

**Example: Count Students in Each Grade**

Modify the grouping to count the number of students in each grade.

java

Copy code

Map<String, Long> studentCountByGrade = students.stream()

.collect(Collectors.groupingBy(Student::getGrade, Collectors.counting()));

**Output:**

The map studentCountByGrade will look like this:

java

Copy code

{

"A" -> 2,

"B" -> 2,

"C" -> 1

}

**Example: Collect Student Names in Each Grade**

Group students by grade but collect only their names in a List:

java

Copy code

Map<String, List<String>> studentNamesByGrade = students.stream()

.collect(Collectors.groupingBy(

Student::getGrade,

Collectors.mapping(Student::getName, Collectors.toList())

));

**Output:**

The map studentNamesByGrade will look like this:

java

Copy code

{

"A" -> ["Alice", "Charlie"],

"B" -> ["Bob", "Eve"],

"C" -> ["Dave"]

}

**Example: Use a TreeMap for Sorted Keys**

If you want the keys (grades) in sorted order, use a TreeMap as the map supplier:

java

Copy code

Map<String, List<Student>> sortedStudentsByGrade = students.stream()

.collect(Collectors.groupingBy(

Student::getGrade,

TreeMap::new,

Collectors.toList()

));

**Output:**

The map sortedStudentsByGrade will look like this (keys are sorted alphabetically):

java

Copy code

{

"A" -> [Alice, Charlie],

"B" -> [Bob, Eve],

"C" -> [Dave]

}

**Practical Example**

**With groupingBy**

If the classifier returned values other than true or false, such as salary ranges, groupingBy would handle it:

java

Copy code

Map<String, List<Employee>> groupedBySalaryRange = employees.stream()

.collect(Collectors.groupingBy(e -> {

if (e.getSalary() > 5000) return "High";

else if (e.getSalary() > 2000) return "Medium";

else return "Low";

}));

**\*\*With partitioningBy**

This would not work because partitioningBy is limited to Boolean classifiers. It works when there are only two categories, such as:

java

Copy code

Map<Boolean, List<Employee>> partitioned = employees.stream()

.collect(Collectors.partitioningBy(e -> e.getSalary() > 2000));

**\*\*Summary**

* Use partitioningBy when your grouping condition naturally results in a true/false partition.
* Use groupingBy for more general grouping logic where multiple keys or non-Boolean keys are required.

\*\*Example for partition by:

Map<Boolean, List<Employee>> hm1 = employees.stream()

.collect(Collectors.partitioningBy(i -> i.getSalary() > 2000));

\*\*MIN ELEMENT:

Optional<Integer> integer=l2.stream().min(Comparator.*naturalOrder*());  
System.*out*.println(integer);

\*\*MAX ELEMENT:

Optional<Integer> integer=l2.stream().max(Comparator.*naturalOrder*());  
System.*out*.println(integer);

\*\*ANY MATCH:

boolean anyMatch = l2.stream().anyMatch(i -> i > 3);

\*\*FIND FIRST:

Optional<Integer> firstMatch = l2.stream().filter(i -> i > 3).findFirst();

\*\*COUNT:

Long aaa=l1.stream().count();  
System.*out*.println(aaa);

SUM: using map to int.

int a2=l2.stream().mapToInt(i->i).sum();  
System.*out*.println(a2);

18.explain about peek ?

Ans,

to log elements during stream processing.

List<Integer> numbers = List.of(1, 2, 3, 4, 5);

numbers.stream()

.filter(n -> n % 2 == 0) // Keep even numbers

.peek(n -> System.out.println("Filtered: " + n)) // Log filtered numbers

.map(n -> n \* n) // Square each number

.peek(n -> System.out.println("Squared: " + n)) // Log squared numbers

.forEach(System.out::println); // Print final results

// Output:

// Filtered: 2

// Squared: 4

// 4

// Filtered: 4

// Squared: 16

// 16

19.what are parallel streams?

A,

Example:

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Main {

public static void main(String[] args) {

// List of integers

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Example with parallel stream for squaring and sequential stream for filtering and summing

int result = numbers.parallelStream() // Start with parallel stream

.map(n -> n \* n) // Square the numbers (using parallelism)

.sequential() // Switch to sequential stream for filtering and summing

.filter(n -> n % 2 == 0) // Keep only even squares

.mapToInt(Integer::intValue) // Convert to int for summing

.sum(); // Sum the even squares

System.out.println("Sum of even squares: " + result); // Output: Sum of even squares: 220

// Separate example with sequential stream

List<Integer> resultSequential = numbers.stream() // Use sequential stream

.map(n -> n \* n) // Square the numbers sequentially

.filter(n -> n % 2 == 0) // Keep only even squares

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

System.out.println("Even squares (sequential): " + resultSequential); // Output: [4, 16, 36, 64, 100]

}

}

20.when to use parallel streams?

A,When we have heavy operations use this else go with sequential streams.

21,terminology in java 8 features ?

A,

Java 8 introduced Streams as a powerful abstraction for working with sequences of data. A Stream represents a sequence of elements supporting parallel and sequential aggregate operations. Here are the main terminologies related to Streams in Java 8:

1. Stream

• Definition: A sequence of elements supporting parallel and sequential aggregate operations. It is not a data structure; it is an abstraction for operations on data.

• Example: List<String> list = Arrays.asList("a", "b", "c");

• Stream<String> stream = list.stream();

2. Intermediate Operations

• Definition: Operations on streams that transform the data and return a new stream. These operations are lazy and are only executed when a terminal operation is invoked.

• Common Intermediate Operations:

• filter(): Filters elements based on a condition.

Stream<Integer> evenNumbers = numbers.stream().filter(n -> n % 2 == 0);

• map(): Transforms each element in the stream.

Stream<String> upperCase = strings.stream().map(String::toUpperCase);

• flatMap(): Flattens a stream of streams into a single stream.

Stream<String> words = sentences.stream().flatMap(sentence -> Arrays.stream(sentence.split(" ")));

• distinct(): Removes duplicate elements from the stream.

• sorted(): Sorts the elements of the stream.

• peek(): Performs an action on each element, primarily used for debugging.

• limit(): Limits the number of elements in the stream.

• skip(): Skips the first n elements in the stream.

3. Terminal Operations

• Definition: Operations that produce a result or a side-effect, and terminate the stream pipeline. These operations are eager and trigger the processing of the stream.

• Common Terminal Operations:

• collect(): Collects the elements of the stream into a collection like a List, Set, Map, etc.

List<Integer> list = numbers.stream().collect(Collectors.toList());

• forEach(): Performs an action for each element in the stream.

numbers.stream().forEach(System.out::println);

• reduce(): Reduces the stream to a single value using an associative accumulation function.

int sum = numbers.stream().reduce(0, Integer::sum);

• count(): Returns the number of elements in the stream.

• min() and max(): Return the minimum or maximum element based on a comparator.

• anyMatch(), allMatch(), noneMatch(): Check if any/all/no elements match a given predicate.

• findFirst() and findAny(): Returns the first or any element from the stream.

• toArray(): Collects the elements into an array.

4. Lazy Evaluation

• Definition: Stream operations are lazy, meaning they are not executed until a terminal operation is invoked. Intermediate operations are just set up and are only executed when required.

• Example: In the chain of operations, no computation is done until a terminal operation is triggered (e.g., collect() or forEach()).

5. Short-Circuiting Operations

• Definition: Operations that terminate early when a condition is satisfied, without processing the entire stream.

• Examples:

• anyMatch(), allMatch(), noneMatch(): These operations can short-circuit by stopping early once a matching element is found.

• findFirst(), findAny(): These operations return as soon as the first element is found, making them short-circuiting.

6. Parallel Streams

• Definition: Streams that can be processed in parallel using multiple threads. This can improve performance on large datasets by leveraging multi-core processors.

• Usage: stream.parallel() allows you to perform parallel operations.

• Example:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.parallelStream().reduce(0, Integer::sum);

7. Collector

• Definition: A Collector is a special type of reduction operation that is used with the collect() method to gather elements into a collection, summarize data, or group data.

• Common Collectors:

• toList(), toSet(), toMap(): Collect stream elements into a List, Set, or Map.

• groupingBy(): Groups elements of a stream by a classifier function.

• partitioningBy(): Partitions the elements of the stream into two groups based on a predicate.

• joining(): Concatenates elements into a string.

• summarizingInt(), summarizingDouble(), summarizingLong(): Collects statistics like sum, average, count, min, and max.

8. Optional

• Definition: A container object which may or may not contain a value. It is used to represent the potential absence of a value, reducing the need for null checks.

• Common methods:

• isPresent(): Checks if a value is present.

• ifPresent(): Executes a lambda if the value is present.

• get(): Retrieves the value if present.

• orElse(), orElseGet(), orElseThrow(): Return the value or a default if not present.

9. Stream Pipelines

• Definition: A sequence of stream operations where data is passed from one operation to the next. Streams support a fluent interface, allowing multiple operations to be chained together in a readable way.

• Example:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> result = numbers.stream()

.filter(n -> n % 2 == 0)

.map(n -> n \* n)

.collect(Collectors.toList());

10. Stream vs Collection

• Stream: A sequence of elements that allows processing on the fly (without modifying the underlying data source).

• Collection: A container (e.g., List, Set, Map) that holds elements in a particular order or structure.

11. Reduction

• Definition: A stream operation that combines all elements of a stream into a single result, using an associative accumulation function. Examples are reduce(), collect(), and various summarization operations.

• Example:

int sum = numbers.stream().reduce(0, Integer::sum); // Reduction

Conclusion

These are the core terminologies and concepts in Java 8 Streams. The Stream API allows for a functional and declarative approach to data processing, enabling powerful operations like filtering, mapping, reducing, and collecting. Understanding these terminologies will help in efficiently using streams for data processing tasks in Java.

Some interview questions from backend developer?  
**Q1. How many male and female employees are there in the organization?** You can count the number of male and female employees using streams:

long maleCount = employees.stream().filter(e -> e.getGender().equals("Male")).count();

long femaleCount = employees.stream().filter(e -> e.getGender().equals("Female")).count();

System.out.println("Male employees: " + maleCount);

System.out.println("Female employees: " + femaleCount);

**Q2. Find the sum of even numbers and the sum of odd numbers in a given list.**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

int sumEven = numbers.stream().filter(n -> n % 2 == 0).mapToInt(Integer::intValue).sum();

int sumOdd = numbers.stream().filter(n -> n % 2 != 0).mapToInt(Integer::intValue).sum();

System.out.println("Sum of even numbers: " + sumEven);

System.out.println("Sum of odd numbers: " + sumOdd);

**Q3. Find duplicate elements in a given integers list.**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 3, 2, 1);

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

Set<Integer> duplicateNumbers = numbers.stream()

.filter(n -> !uniqueNumbers.add(n))

.collect(Collectors.toSet());

System.out.println("Duplicate elements: " + duplicateNumbers);

**Q4. Remove all the occurrences of a given character.**

String str = "Java programming";

char removeChar = 'a';

Using the provided code:

String result = str.chars()

.filter(ch -> ch != removeChar)

.mapToObj(ch -> String.valueOf((char) ch))

.collect(Collectors.joining());

**Q5. Find special characters in a string.**

String str = "hello@world#2021!";

String specialChars = str.replaceAll("[a-zA-Z0-9]", "");

System.out.println("Special characters: " + specialChars);

String str = "Hello, World! 123";

Using the provided code:

String specialChars = str.chars()

.filter(ch -> !Character.isLetterOrDigit(ch))

.mapToObj(ch -> String.valueOf((char) ch))

.collect(Collectors.joining());

**Q6. Find out all the numbers starting with 1 from an integer list.**

List<Integer> numbers = Arrays.asList(10, 15, 21, 31, 41, 51);

List<Integer> result = numbers.stream()

.filter(n -> String.valueOf(n).startsWith("1"))

.collect(Collectors.toList());

System.out.println("Numbers starting with 1: " + result);

**Q7. Find the employee with the highest salary from a list of employee objects.**

See in intellij java coding file.

System.out.println("Highest paid employee: " + highestPaidEmployee);

**Q8. Concatenate a list of strings into a single string, separated by commas.**

List<String> strings = Arrays.asList("apple", "banana", "cherry");

String result = String.join(", ", strings);

System.out.println("Concatenated string: " + result);

String result = strings.stream()

.collect(Collectors.joining(", "));

**Q9. Find the first non-repeating character in a string.**

String str = "swiss";

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

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

str.chars()

.mapToObj(c -> (char) c)

.forEach(c -> {

if (!seen.add(c)) {

repeating.add(c);

}

});

Character firstNonRepeating = seen.stream()

.filter(c -> !repeating.contains(c))

.findFirst()

.orElse(null);

System.out.println(firstNonRepeating); // Output: w

**Q10. Find the frequency of characters in a given string.**

String str = "hello world";

Map<Character, Long> frequency = str.chars()

.mapToObj(c -> (char) c)

.collect(Collectors.groupingBy(c -> c, Collectors.counting()));

System.out.println("Character frequency: " + frequency);

**Q11. Group a list of strings by their length.**

List<String> strings = Arrays.asList("apple", "banana", "cherry", "date");

Map<Integer, List<String>> groupedByLength = strings.stream()

.collect(Collectors.groupingBy(String::length));

System.out.println("Grouped by length: " + groupedByLength);

**Q12. Count the number of occurrences of each character in a string.**

String str = "hello world";

Map<Character, Long> occurrences = str.chars()

.mapToObj(c -> (char) c)

.collect(Collectors.groupingBy(c -> c, Collectors.counting()));

System.out.println("Character occurrences: " + occurrences);

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

Map<Character, Long> characterCounts = names.stream()

.flatMap(name -> name.chars().mapToObj(c -> (char) c))

.collect(Collectors.groupingBy(c -> c, Collectors.counting()));

System.out.println("Character counts: " + characterCounts);

**Q13. Filter a list of numbers to only include those greater than 10 and then find their average.**

List<Integer> numbers = Arrays.asList(5, 10, 15, 20, 25);

double average = numbers.stream()

.filter(n -> n > 10)

.mapToInt(Integer::intValue)

.average()

.orElse(0);

System.out.println("Average of numbers greater than 10: " + average);

**Q14. Convert a list of strings to a map where the key is the string and the value is the length.**

List<String> strings = Arrays.asList("apple", "banana", "cherry");

Map<String, Integer> stringLengthMap = strings.stream()

.collect(Collectors.toMap(Function.identity(), String::length));

System.out.println("String length map: " + stringLengthMap);

**Q15. Flatten a list of lists of integers into a single list of integers.**

List<List<Integer>> listOfLists = Arrays.asList(

Arrays.asList(1, 2, 3),

Arrays.asList(4, 5, 6),

Arrays.asList(7, 8, 9)

);

List<Integer> flattenedList = listOfLists.stream()

.flatMap(List::stream)

.collect(Collectors.toList());

System.out.println("Flattened list: " + flattenedList);

**Q16. Given a list of transactions, filter out transactions of a specific type and collect them into a set.**

Set<Transaction> filteredTransactions = transactions.stream()

.filter(t -> t.getType().equals("SPECIFIC\_TYPE"))

.collect(Collectors.toSet());

System.out.println("Filtered transactions: " + filteredTransactions);

**Q17. Find the first name of the oldest person in a list of Person objects.**

String oldestPersonName = persons.stream()

.max(Comparator.comparing(Person::getAge))

.map(Person::getFirstName)

.orElse("No persons available");

System.out.println("Oldest person's first name: " + oldestPersonName);

**Q18. Skip the first 5 elements in a list and then print the rest.**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

numbers.stream()

.skip(5)

.forEach(System.out::println);

**Q19. Collect all unique words from a list of sentences.**

List<String> sentences = Arrays.asList("hello world", "hello java", "java streams");

Set<String> uniqueWords = sentences.stream()

.flatMap(sentence -> Arrays.stream(sentence.split(" ")))

.collect(Collectors.toSet());

System.out.println("Unique words: " + uniqueWords);

**Q20. Filter out null values from a list of strings using streams.**

List<String> strings = Arrays.asList("apple", null, "banana", null, "cherry");

List<String> filteredStrings = strings.stream()

.filter(Objects::nonNull)

.collect(Collectors.toList());

System.out.println("Filtered strings: " + filteredStrings);

**Q21. Partition a list of integers into even and odd numbers.**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

Map<Boolean, List<Integer>> partitioned = numbers.stream()

.collect(Collectors.partitioningBy(n -> n % 2 == 0));

System.out.println("Even numbers: " + partitioned.get(true));

System.out.println("Odd numbers: " + partitioned.get(false));

**Q22. Generate an infinite sequence of random numbers and print the first 10.**

new Random().ints()

.limit(10)

.forEach(System.out::println);

**Q23. Find the sum of the squares of a list of integers.**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sumOfSquares = numbers.stream()

.mapToInt(n -> n \* n)

.sum();

System.out.println("Sum of squares: " + sumOfSquares);

**Q24. Convert a list of strings to a list of their respective lengths using streams.**

List<String> strings = Arrays.asList("apple", "banana", "cherry");

List<Integer> lengths = strings.stream()

.map(String::length)

.collect(Collectors.toList());

System.out.println("Lengths of strings: " + lengths);

**Q25. Find the product of all elements in a list of integers.**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int product = numbers.stream()

.reduce(1, (a, b) -> a \* b);

System.out.println("Product of all elements: " + product);

**Q26. Merge two lists of integers and remove duplicates.**

List<Integer> list1 = Arrays.asList(1, 2, 3);

List<Integer> list2 = Arrays.asList(3, 4, 5);

List<Integer> mergedList = Stream.concat(list1.stream(), list2.stream())

.distinct()

.collect(Collectors.toList());

System.out.println("Merged list without duplicates: " + mergedList);

**Q27. Check if any string in a list starts with a specific prefix.**

List<String> strings = Arrays.asList("apple", "banana", "cherry");

boolean startsWithA = strings.stream()

.anyMatch(s -> s.startsWith("a"));

System.out.println("Any string starts with 'a': " + startsWithA);

**Q28. List of employees, having properties id, name, salary. List the top three highest-paid employees.**

List<Employee> topThreeEmployees = employees.stream()

.sorted(Comparator.comparing(Employee::getSalary).reversed())

.limit(3)

.collect(Collectors.toList());

System.out.println("Top three highest-paid employees: " + topThreeEmployees);

**Q29. Array of numbers, find the second highest number from the array.**

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

int secondHighest = Arrays.stream(numbers)

.boxed()

.sorted(Comparator.reverseOrder())

.skip(1)

.findFirst()

.orElseThrow(NoSuchElementException::new);

System.out.println("Second highest number: " + secondHighest);

**Q30. How to sort a map by its values using streams API?**

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

map.put("A", 10);

map.put("B", 30);

map.put("C", 20);

Map<String, Integer> sortedMap = map.entrySet().stream()

.sorted(Map.Entry.comparingByValue())

.collect(Collectors.toMap(

Map.Entry::getKey,

Map.Entry::getValue,

(e1, e2) -> e1,

LinkedHashMap::new

));

System.out.println("Sorted map by values: " + sortedMap);

**Q31. What are Streams in Java 8?** Streams in Java 8 are a new abstraction that allows processing sequences of elements in a functional style. They support operations like filtering, mapping, and reducing, and can be either sequential or parallel.

**Q32. Explain Intermediate operations in Streams.** Intermediate operations are operations that transform a stream into another stream. They are lazy and do not trigger any processing until a terminal operation is invoked. Examples include filter(), map(), flatMap(), sorted(), and distinct().

**33. Explain Terminal Operations in Streams.** Terminal operations are operations that produce a result or a side-effect and trigger the processing of the stream. Examples include collect(), forEach(), reduce(), count(), findFirst(), and anyMatch().

**Q34. List All Examples for Stream Intermediate Operations.**

* filter(): Filters elements based on a predicate.
* map(): Transforms each element using a function.
* flatMap(): Flattens nested structures.
* sorted(): Sorts elements.
* distinct(): Removes duplicate elements.
* limit(): Limits the number of elements.
* skip(): Skips the first N elements.

**Q35. List All Examples for Stream Terminal Operations.**

* collect(): Collects elements into a collection.
* forEach(): Performs an action for each element.
* reduce(): Reduces elements to a single value.
* count(): Counts the number of elements.
* findFirst(): Finds the first element.
* anyMatch(): Checks if any element matches a predicate.
* allMatch(): Checks if all elements match a predicate.
* noneMatch(): Checks if no elements match a predicate.

**Q36. Explain the reduce method with examples.** The reduce method combines elements of a stream into a single result using an associative accumulation function.

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()

.reduce(0, Integer::sum);

System.out.println("Sum: " + sum);

int product = numbers.stream()

.reduce(1, (a, b) -> a \* b);

System.out.println("Product: " + product);

**Q37. Explain distinct and sorted methods with examples.**

* distinct(): Removes duplicate elements.

List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 4, 4, 5);

List<Integer> distinctNumbers = numbers.stream()

.distinct()

.collect(Collectors.toList());

System.out.println("Distinct numbers: " + distinctNumbers);

* sorted(): Sorts elements.

List<Integer> numbers = Arrays.asList(5, 3, 1, 4, 2);

List<Integer> sortedNumbers = numbers.stream()

.sorted()

.collect(Collectors.toList());

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

**Q38. How to Sort Streams with sorted using a custom Comparator?**

List<Employee> sortedEmployees = employees.stream()

.sorted(Comparator.comparing(Employee::getSalary).reversed())

.collect(Collectors.toList());

System.out.println("Sorted employees by salary: " + sortedEmployees);

**Q39. How to use Streams collect method?** The collect method is used to accumulate elements of a stream into a collection or other data structures.

List<String> strings = Arrays.asList("apple", "banana", "cherry");

List<String> collectedList = strings.stream()

.collect(Collectors.toList());

System.out.println("Collected list: " + collectedList);

Set<String> collectedSet = strings.stream()

.collect(Collectors.toSet());

System.out.println("Collected set: " + collectedSet);

Map<String, Integer> collectedMap = strings.stream()

.collect(Collectors.toMap(Function.identity(), String::length));

System.out.println("Collected map: " + collectedMap);