Stream API

1. What is the Stream API in Java, and why was it introduced?

The Stream API processes collections in a functional and declarative way, introduced in Java 8 to simplify bulk operations on data like filtering, mapping, and reducing.

```
1   List<Integer> numbers = Arrays.asList(1, 2, 3);
2   numbers.stream().filter(n -> n > 1).forEach(System.out::println);
```

2. How does the Stream API differ from traditional iteration?

- **Declarative**: Focus on "what to do" rather than "how to do".
- **Lazy Evaluation**: Processes elements only when needed.
- **Parallel Processing**: Easily supports concurrent execution.

```
1   List<String> names = Arrays.asList("Alice", "Bob");
2   names.forEach(System.out::println); // Traditional: explicit iteration
3   names.stream().forEach(System.out::println); // Stream: functional style
```

3. What are the key characteristics of a Stream in Java?

- **Non-Storage**: Does not store data.
- **Lazy Execution**: Operations are executed only when required.
- **Immutability**: Original data source is not modified.
- **Can be Sequential or Parallel**.

4. Can you explain the difference between intermediate and terminal operations in Streams?

Intermediate and Terminal Operations

- **Intermediate Operations**:
    - Transform a stream into another stream.
    - They are **lazy**, meaning they do not execute until a terminal operation is invoked.
    - Examples: `filter()`, `map()`, `sorted()`, `distinct()`, `limit()`, `skip()`.
- **Terminal Operations**:
    - Trigger the execution of the intermediate operations and produce a result or a side effect.
    - They are **eager**, meaning they process the entire stream when invoked.
    - Examples: `collect()`, `forEach()`, `reduce()`, `count()`, `min()`, `max()`, `anyMatch()`, `allMatch()`, `noneMatch()`.

5. How does the `filter()` method work in the Stream API?

Filters elements based on a predicate.

```
1   List<String> names = Arrays.asList("Alice", "Bob");
2   names.stream().filter(n -> n.startsWith("A")).forEach(System.out::println); // Alice
```

6. What is the purpose of the `map()` method in Streams?

Transforms elements in a Stream.

```
1   List<Integer> numbers = Arrays.asList(1, 2, 3);
2   numbers.stream().map(n -> n * 2).forEach(System.out::println); // 2, 4, 6
```

7. Can you explain the use of the `flatMap()` method with an example?

Flattens nested structures.

```
1   List<List<Integer>> lists = Arrays.asList(Arrays.asList(1, 2), Arrays.asList(3, 4));
2   lists.stream().flatMap(List::stream).forEach(System.out::println); // 1, 2, 3, 4
```

8. What are collectors, and how are they used with Streams?

Collectors collect and reduce Stream elements into data structures or results.

```
1   List<Integer> numbers = Arrays.asList(1, 2, 3);
2   List<Integer> squared = numbers.stream().map(n -> n * n).collect(Collectors.toList());
```

9. How can you perform sorting in Streams using `sorted()`?

Sorts elements in natural or custom order.

```
1   List<Integer> numbers = Arrays.asList(3, 1, 2);
2   numbers.stream().sorted().forEach(System.out::println); // 1, 2, 3
```

10. What is the difference between `findFirst()` and `findAny()` in Streams?

- `findFirst()` : Returns the first element.
- `findAny()` : Returns any element (useful in parallel streams).

```
1   List<Integer> numbers = Arrays.asList(1, 2, 3);
2   System.out.println(numbers.stream().findFirst().orElse(-1)); // 1
```

11. How does the `reduce()` method work in the Stream API?

Aggregates elements into a single result.

```
1   List<Integer> numbers = Arrays.asList(1, 2, 3);
2   int sum = numbers.stream().reduce(0, Integer::sum); // 6
```

12. What is the purpose of the `distinct()` method in Streams?

Removes duplicate elements.

```
1  List<Integer> numbers = Arrays.asList(1, 2, 2, 3);
2  numbers.stream().distinct().forEach(System.out::println); // 1, 2, 3
```

13. How can you create an infinite Stream in Java?

Use `Stream.generate()` or `Stream.iterate()`.

```
1  Stream<Integer> infinite = Stream.iterate(0, n -> n + 1);
2  infinite.limit(5).forEach(System.out::println); // 0, 1, 2, 3, 4
```

14. What is the role of `peek()` in debugging Stream operations?

Allows inspection without modifying the Stream.

```
1  List<Integer> numbers = Arrays.asList(1, 2, 3);
2  numbers.stream().peek(System.out::println).map(n -> n * 2).forEach(System.out::println);
```

15. How do parallel Streams differ from sequential Streams?

Parallel Streams execute operations concurrently.

```
1  List<Integer> numbers = Arrays.asList(1, 2, 3);
2  numbers.parallelStream().forEach(System.out::println);
```

16. What are some common pitfalls of using parallel Streams?

- Overhead of thread management.
- Thread-safety issues with shared resources.
- Inefficient for small datasets.

17. Can you explain the `toMap()` collector in the Stream API?

Converts elements into a `Map`.

```
1  List<String> names = Arrays.asList("Alice", "Bob");
2  Map<String, Integer> map = names.stream().collect(Collectors.toMap(n -> n, String::length));
3  System.out.println(map); // {Alice=5, Bob=3}
```

18. How do you group elements in a Stream using the `groupingBy()` collector?

Groups elements by a classifier function.

```
1  List<String> names = Arrays.asList("Alice", "Bob", "Anna");
2  Map<Character, List<String>> grouped = names.stream()
3      .collect(Collectors.groupingBy(n -> n.charAt(0)));
4  System.out.println(grouped); // {A=[Alice, Anna], B=[Bob]}
```

19. What is the use of the `partitioningBy()` collector in Streams?

Partitions elements into two groups based on a predicate.

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4);
Map<Boolean, List<Integer>> partitioned = numbers.stream()
    .collect(Collectors.partitioningBy(n -> n % 2 == 0));
System.out.println(partitioned); // {false=[1, 3], true=[2, 4]}
```

20. How does short-circuiting work in the Stream API?

Stops processing when the result is determined (e.g., `limit`, `findFirst`).

```java
Stream<Integer> numbers = Stream.of(1, 2, 3, 4);
numbers.filter(n -> n > 2).findFirst().ifPresent(System.out::println); // 3
```

21. Filter vs Map vs Reduce

`filter()`:

- Used to filter elements based on a predicate (condition).
- Example:

```java
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
List<Integer> evenNumbers = numbers.stream()
                                    .filter(n -> n % 2 == 0)
                                    .collect(Collectors.toList());
```

`map()`:

- Transforms each element in a stream into another value.
- Example:

```java
List<String> names = List.of("John", "Jane");
List<Integer> nameLengths = names.stream()
                                    .map(String::length)
                                    .collect(Collectors.toList());
```

`reduce()`:

- Performs aggregation or combines elements of the stream into a single value.
- Example:

```java
List<Integer> numbers = List.of(1, 2, 3, 4);
int sum = numbers.stream()
                    .reduce(0, Integer::sum);
```

22. What is Comparator in java?

A `Comparator` is used to compare two objects for custom sorting.

- Example:

```java
List<String> names = List.of("John", "Jane", "Alice");
List<String> sortedNames = names.stream()
                                    .sorted(Comparator.naturalOrder())
```

```
4                                    .collect(Collectors.toList());
5   // or Comparator.reverseOrder() for reverse order
```

23. What is min() and max()?

`min()` & `max()`

- Find the minimum or maximum element based on a comparator.
- Example:

```
1   List<Integer> numbers = List.of(10, 20, 5, 30);
2   int min = numbers.stream().min(Integer::compare).orElseThrow();
3   int max = numbers.stream().max(Integer::compare).orElseThrow();
4
```

24. What is aggregate ?

Aggregate Operations

Aggregate operations perform calculations such as summation, averaging, or counting.

- Example:

```
1   List<Integer> numbers = List.of(1, 2, 3, 4, 5);
2   int sum = numbers.stream()
3                   .mapToInt(Integer::intValue)
4                   .sum();
5   double average = numbers.stream()
6                       .mapToInt(Integer::intValue)
7                       .average()
8                       .orElse(0.0);
9   long count = numbers.stream().count();
```

25. Difference between `.toList()` and `collect(Collectors.toList())`

- `.toList()` :
    - Introduced in Java 16.
    - Returns an **unmodifiable** list.
    - Example:

```
1   List<Integer> result = numbers.stream().toList();
```

- `collect(Collectors.toList())` :
    - Available in earlier Java versions.
    - Returns a **modifiable** list.
    - Example:

```
1   List<Integer> result = numbers.stream().collect(Collectors.toList());
2   result.add(6); // Modifiable
```

⭐ Coding Questions

---

1. Find the First Non-Repeating Character in a String

```
1  String input = "swiss";
2  Character firstNonRepeating = input.chars()
3      .mapToObj(c -> (char) c)
4      .filter(c -> input.indexOf(c) == input.lastIndexOf(c))
5      .findFirst()
6      .orElse(null);
7  System.out.println(firstNonRepeating); // Output: 'w'
```

2. Count Frequency of Elements in a List

```
1  List<String> items = List.of("apple", "banana", "apple", "orange", "banana");
2  Map<String, Long> frequencyMap = items.stream()
3      .collect(Collectors.groupingBy(item -> item, Collectors.counting()));
4  System.out.println(frequencyMap); // Output: {orange=1, banana=2, apple=2}
```

3. Find Top-N Highest Numbers in a List

```
1  List<Integer> numbers = List.of(10, 20, 5, 30, 25);
2  int n = 3;
3  List<Integer> topN = numbers.stream()
4      .sorted(Comparator.reverseOrder())
5      .limit(n)
6      .collect(Collectors.toList());
7  System.out.println(topN); // Output: [30, 25, 20]
```

4. Partition a List into Odd and Even Numbers

```
1  List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
2  Map<Boolean, List<Integer>> partitioned = numbers.stream()
3      .collect(Collectors.partitioningBy(n -> n % 2 == 0));
4  System.out.println(partitioned);
5  // Output: {false=[1, 3, 5], true=[2, 4, 6]}
```

5. Flatten a List of Lists

```
1  List<List<Integer>> nestedList = List.of(List.of(1, 2), List.of(3, 4), List.of(5));
2  List<Integer> flatList = nestedList.stream()
3      .flatMap(List::stream)
4      .collect(Collectors.toList());
5  System.out.println(flatList); // Output: [1, 2, 3, 4, 5]
```

6. Find Duplicate Elements in a List

```
1  First Way
2  List<Integer> numbers = List.of(1, 2, 3, 4, 5, 2, 3);
3  Set<Integer> duplicates = numbers.stream()
4      .filter(n -> Collections.frequency(numbers, n) > 1)
5      .collect(Collectors.toSet());
6  System.out.println(duplicates); // Output: [2, 3]
7
   //Second Way
8  Set<Integer> set = new LinkedHashSet<>();
9  List<Integer> dup = numbers.stream().filter(x -> !set.add(x)).toList();
   System.out.println(duplicates);// Output: [2, 3]
```

7. Sort a List of Custom Objects by Multiple Fields

```
1  class Person {
2      String name;
3      int age;
4      // Constructor, Getters, Setters
5  }
6  List<Person> people = List.of(new Person("John", 25), new Person("Alice", 30), new Person("John", 20));
7  List<Person> sorted = people.stream()
8      .sorted(Comparator.comparing(Person::getName).thenComparing(Person::getAge))
9      .collect(Collectors.toList()).
10 sorted.forEach(p -> System.out.println(p.name + " - " + p.age));
11 // Output: John - 20, John - 25, Alice - 30
```

8. Calculate the Total Salary of Employees in Each Department

```
1  Map<String, Double> totalSalaryByDept = employees.stream()
2      .collect(Collectors.groupingBy(
3          Employee::getDepartment,
4          Collectors.summingDouble(Employee::getSalary)
5      ));
6  System.out.println(totalSalaryByDept);
```

9. Filter Strings Starting with a Specific Letter and Collect Them

```
1  List<String> names = List.of("Alice", "Bob", "Charlie", "Alex", "Brian");
2  List<String> filtered = names.stream()
3      .filter(name -> name.startsWith("A"))
4      .collect(Collectors.toList());
5  System.out.println(filtered); // Output: [Alice, Alex]
```

10. Find the Longest String in a List

```
1  List<String> strings = List.of("short", "medium", "longest", "tiny");
2  String longest = strings.stream()
3      .max(Comparator.comparingInt(String::length))
4      .orElse("");
5  System.out.println(longest); // Output: "longest"
```

11. Find Common Elements Between Two Lists

```
1  List<Integer> list1 = List.of(1, 2, 3, 4);
2  List<Integer> list2 = List.of(3, 4, 5, 6);
3  List<Integer> common = list1.stream()
4      .filter(list2::contains)
5      .collect(Collectors.toList());
6  System.out.println(common); // Output: [3, 4]
```

12. Find the nth Smallest or Largest Number

```
1  List<Integer> numbers = List.of(10, 20, 30, 40, 50);
2  int n = 2;
3  int nthLargest = numbers.stream()
4      .sorted(Comparator.reverseOrder())
5      .skip(n - 1)
6      .findFirst()
7      .orElseThrow();
8  System.out.println(nthLargest); // Output: 40
```

13. Find the First Non-Repeating Character in a String

```
1  String s1= "ssssssHiiiii";
2  Character c1 = s1.chars()
3          .mapToObj(c -> (char) c)
4          .collect(Collectors
5          .groupingBy(c -> c, LinkedHashMap::new, Collectors.counting()))
6          .entrySet()
7          .stream().filter(entry -> entry.getValue() == 1)
8          .map(Map.Entry::getKey).findFirst().orElse(null);
9  System.out.println(c1);
```

14. Count the Occurrences of Each Character in a String

```
1  String s1= "ssssssHiiiii";
2  Map<Character, Long> collect = s1.chars().mapToObj(c -> (char) c)
3          .collect(Collectors.groupingBy(c -> c, Collectors.counting()));
4  System.out.println(collect);
```

15. Find the Sum and Average of a List of Numbers

```
1  int sum = numbers.stream().mapToInt(Integer::intValue).sum();
2  OptionalDouble average = numbers.stream()
3                                  .mapToInt(Integer::intValue).average();
```

16. Find the distinct numbers

```
1  List distinctNumbers = numbers.stream().distinct()
2                                          .collect(Collectors.toList());
```

Note:

Functional Interface ⟶ interface with single abstract method (i.e Runnable)

Lamba funtion⟶Anonymous function without return type,name,access modifier.

        used for implementing functional interface.

i.e

```
1  Thread t1 = new Thread(()->System.out.println("Hi"));
```

Predicate ⟶Functional interface (Boolean-values function).Used for apply condition

i.e

```
1  Predicate<Integer> isEven = x->x%2==0;
2  System.out.println(isEven.test(3));
```

Function ⟶Functional interface .Used for doing computation.

Comsumer ⟶ Only take a value;

Supplier⟶ Only provide a value (Do not take any value).

Method reference ⟶ use method without invoking & in place of lambda expression

Follow for more such content🚀✨