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.

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

```
List<String> names = Arrays.asList("Alice", "Bob");
names.forEach(System.out::println); // Traditional: explicit iteration
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.

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

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

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

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

```
List<Integer> numbers = Arrays.asList(3, 1, 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).

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

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

---

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

Removes duplicate elements.

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

```
Stream<Integer> infinite = Stream.iterate(0, n -> n + 1);
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.

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

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

```
List<String> names = Arrays.asList("Alice", "Bob");
Map<String, Integer> map = names.stream().collect(Collectors.toMap(n -> n, String::length));
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.

```
List<String> names = Arrays.asList("Alice", "Bob", "Anna");
Map<Character, List<String>> grouped = names.stream()
    .collect(Collectors.groupingBy(n -> n.charAt(0)));
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.

```
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`).

```
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:

```
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:

```
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:

```
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:

```
List<String> names = List.of("John", "Jane", "Alice");
List<String> sortedNames = names.stream()
                                .sorted(Comparator.naturalOrder())
                                .collect(Collectors.toList());
// 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:

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

```

24. What is aggregate ?

Aggregate Operations

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

- Example:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
int sum = numbers.stream()
                 .mapToInt(Integer::intValue)
                 .sum();
double average = numbers.stream()
                        .mapToInt(Integer::intValue)

```

```
                        .average()
                        .orElse(0.0);
long count = numbers.stream().count();
```

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

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

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

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

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

⭐ Coding Questions

---

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

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

---

2. Count Frequency of Elements in a List

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

---

3. Find Top-N Highest Numbers in a List

```
List<Integer> numbers = List.of(10, 20, 5, 30, 25);
```

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

---

4. Partition a List into Odd and Even Numbers

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

---

5. Flatten a List of Lists

```
List<List<Integer>> nestedList = List.of(List.of(1, 2), List.of(3, 4), List.of(5));
List<Integer> flatList = nestedList.stream()
    .flatMap(List::stream)
```

```
    .collect(Collectors.toList());
System.out.println(flatList); // Output: [1, 2, 3, 4, 5]
```

---

6. Find Duplicate Elements in a List

```
First Way
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 2, 3);
Set<Integer> duplicates = numbers.stream()
    .filter(n -> Collections.frequency(numbers, n) > 1)
    .collect(Collectors.toSet());
System.out.println(duplicates); // Output: [2, 3]

//Second Way
Set<Integer> set = new LinkedHashSet<>();
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

```
class Person {
```

```
    String name;
    int age;
    // Constructor, Getters, Setters
}
List<Person> people = List.of(new Person("John", 25), new Person("Alice", 30), new Person("John", 20));
List<Person> sorted = people.stream()
    .sorted(Comparator.comparing(Person::getName).thenComparing(Person::getAge))
    .collect(Collectors.toList());
sorted.forEach(p -> System.out.println(p.name + " - " + p.age));
// Output: John - 20, John - 25, Alice - 30
```

---

8. Calculate the Total Salary of Employees in Each Department

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

---

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

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

---

10. Find the Longest String in a List

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

---

11. Find Common Elements Between Two Lists

```
List<Integer> list1 = List.of(1, 2, 3, 4);
List<Integer> list2 = List.of(3, 4, 5, 6);
```

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

---

12. Find the nth Smallest or Largest Number

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

---

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

```
String s1= "sssssHiiiii";
Character c1 = s1.chars()

```

```
        .mapToObj(c -> (char) c)
        .collect(Collectors
        .groupingBy(c -> c, LinkedHashMap::new, Collectors.counting()))
        .entrySet()
        .stream().filter(entry -> entry.getValue() == 1)
        .map(Map.Entry::getKey).findFirst().orElse(null);
System.out.println(c1);
```

---

14. Count the Occurrences of Each Character in a String

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

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

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

16. Find the distinct numbers

```
List distinctNumbers = numbers.stream().distinct()
                                        .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

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

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

i.e

```
Predicate<Integer> isEven = x->x%2==0;
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 🚀✨