# Streams API

The **Streams API** introduced in **Java 8** allows for functional-style operations on collections of objects. A **Stream** is a sequence of elements that can be processed in parallel or sequentially. It provides a clean, concise, and efficient way to handle data processing tasks such as filtering, transforming, and aggregating data.

Streams can be created from collections, arrays, I/O channels, or even manually, and support operations like map, filter, reduce, etc.

### ****1. Stream Basics****

A **Stream** does not modify the underlying data structure. Instead, it creates a pipeline of operations that can be executed on the data in a functional style.

#### ****Key Characteristics of a Stream:****

* **No storage**: A Stream does not store elements. It simply conveys elements from a data source such as a collection or array.
* **Functional in nature**: You can pass functions as arguments to Stream methods, making it easier to chain operations.
* **Laziness-seeking**: Stream operations are lazy. The elements are processed only when needed, which can help improve performance.
* **Possibly unbounded**: Streams can represent infinite sequences of data.
* **Consumable**: A Stream can only be used once. After processing, a Stream is considered consumed.

### ****2. Creating Streams****

You can create a Stream from various sources:

* **From a Collection**:

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

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

* **From an Array**:

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

IntStream stream = Arrays.stream(numbers);

* **From Individual Values**:

Stream<String> stream = Stream.of("one", "two", "three");

* **From Files**:

Stream<String> lines = Files.lines(Paths.get("file.txt"));

### ****3. Stream Operations****

Stream operations are categorized into two types:

* **Intermediate Operations**: These operations return a new Stream and are lazily evaluated. They allow you to chain multiple operations together.
* **Terminal Operations**: These operations are the end of the Stream pipeline and trigger the processing of the stream.

#### ****Intermediate Operations****

Intermediate operations **do not** modify the original stream but return a new stream. They are lazy and only execute when a terminal operation is invoked.

##### **1.** filter()

Filters elements based on a condition.

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

Stream<String> filtered = list.stream().filter(s -> s.startsWith("a"));

filtered.forEach(System.out::println);

**Output:**

apple

##### **2.** map()

Transforms elements in the stream.

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

List<Integer> squared = numbers.stream().map(x -> x \* x).collect(Collectors.toList());

System.out.println(squared);

**Output:**

[1, 4, 9, 16]

##### **3.** flatMap()

Flattens a stream of collections into a single stream.

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

Arrays.asList("apple", "banana"),

Arrays.asList("cherry", "date")

);

List<String> merged = listOfLists.stream()

.flatMap(Collection::stream)

.collect(Collectors.toList());

System.out.println(merged);

**Output:**

[apple, banana, cherry, date]

##### **4.** distinct()

Returns a stream with duplicate elements removed.

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

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

**Output:**

1

2

3

##### **5.** sorted()

Sorts elements in the stream.

List<Integer> numbers = Arrays.asList(5, 3, 8, 1);

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

**Output:**

1

3

5

8

##### **6.** peek()

Performs an action on each element of the stream without modifying the stream. Useful for debugging.

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

list.stream().peek(System.out::println).map(String::toUpperCase).forEach(System.out::println);

**Output:**

a

b

c

d

A

B

C

D

##### **7.** limit()

Limits the number of elements in the stream.

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

numbers.stream().limit(3).forEach(System.out::println);

**Output:**

1

2

3

##### **8.** skip()

Skips the first n elements.

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

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

**Output:**

3

4

5

6

#### ****Terminal Operations****

Terminal operations trigger the processing of the stream and typically produce a result, such as a collection, a number, or a boolean value. After a terminal operation, the stream is consumed.

##### **1.** forEach()

Performs an action on each element of the stream.

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

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

**Output:**

apple

banana

cherry

##### **2.** collect()

Transforms the elements into a different form, such as a list, set, or map.

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

List<Integer> doubled = numbers.stream()

.map(x -> x \* 2)

.collect(Collectors.toList());

System.out.println(doubled);

**Output:**

[2, 4, 6, 8]

##### **3.** reduce()

Reduces the elements of the stream to a single value (e.g., sum, product, concatenation).

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

int sum = numbers.stream().reduce(0, (a, b) -> a + b);

System.out.println(sum);

**Output:**

10

##### **4.** anyMatch()

Checks if any element matches a given condition.

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

boolean result = list.stream().anyMatch(s -> s.startsWith("a"));

System.out.println(result);

**Output:**

true

##### **5.** allMatch()

Checks if all elements match a given condition.

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

boolean result = list.stream().allMatch(s -> s.length() > 3);

System.out.println(result);

**Output:**

true

##### **6.** noneMatch()

Checks if no elements match a given condition.

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

boolean result = list.stream().noneMatch(s -> s.startsWith("z"));

System.out.println(result);

**Output:**

true

##### **7.** findFirst()

Returns the first element of the stream (if it exists).

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

String result = list.stream().findFirst().orElse("No element");

System.out.println(result);

**Output:**

apple

##### **8.** count()

Counts the number of elements in the stream.

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

long count = list.stream().count();

System.out.println(count);

**Output:**

3

##### **9.** max() **and** min()

Returns the maximum or minimum element of the stream according to a comparator.

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

int max = numbers.stream().max(Integer::compare).orElseThrow();

int min = numbers.stream().min(Integer::compare).orElseThrow();

System.out.println("Max: " + max);

System.out.println("Min: " + min);

**Output:**

Max: 5

Min: 1

### ****4. Stream Pipeline Example:****

Let's combine multiple operations in a Stream pipeline:

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

long result = list.stream()

.filter(s -> s.length() > 5)

.map(String::toUpperCase)

.sorted()

.distinct()

.count();

System.out.println(result);

**Output:**

3

This code filters words longer than 5 characters, transforms them to uppercase, sorts them, removes duplicates, and counts the final result.

### ****5. Parallel Streams****

Streams can be processed in parallel, utilizing multiple threads to improve performance for large datasets.

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

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

System.out.println(sum);

**Output:**

55

Using parallelStream() splits the workload across multiple CPU cores for better performance on large collections.