Here’s a complete and detailed explanation of the  **8 Stream API**, which is one of the most powerful features added to 8 for working with collections in a **declarative, functional style**.

**🧠 What is the Stream API?**

The **Stream API** in 8 provides a **high-level abstraction** for processing sequences of elements (like Lists, Sets, etc.) with operations like filtering, mapping, reducing, and collecting — **without modifying the original data structure**.

**✅ Key Characteristics**

| **Feature** | **Description** |
| --- | --- |
| Declarative | Focuses on **what** to do, not how |
| Functional | Uses lambdas and functional interfaces |
| Lazy evaluation | Intermediate operations are not executed until terminal |
| Parallel processing | Easy to parallelize with parallelStream() |
| Non-mutating | Does not modify original collections |

**🔰 Stream Creation**

// From Collection

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

// From Arrays

Stream<Integer> stream = Arrays.stream(new Integer[]{1, 2, 3});

// From values

Stream<String> stream = Stream.of("A", "B", "C");

// Infinite Stream (controlled with limit)

Stream<Double> infinite = Stream.generate(Math::random).limit(5);

**🔄 Types of Operations**

**1. 🔧 Intermediate Operations (Lazy)**

They return another Stream.

| **Operation** | **Description** |
| --- | --- |
| filter() | Filters elements based on condition |
| map() | Transforms each element |
| flatMap() | Flattens nested structures |
| sorted() | Sorts elements |
| distinct() | Removes duplicates |
| limit() | Limits the size |
| skip() | Skips first N elements |

**2. 🏁 Terminal Operations (Triggers execution)**

| **Operation** | **Description** |
| --- | --- |
| forEach() | Iterates over each element |
| collect() | Collects results into a List/Map/Set |
| reduce() | Aggregates values into one |
| count() | Returns number of elements |
| anyMatch() | Checks if any element matches condition |
| allMatch() | Checks if all match |
| noneMatch() | Checks if none match |
| findFirst() | Returns first element (Optional) |
| findAny() | Returns any element (Optional) |

**📌 Common Examples**

**🔹 filter() and collect()**

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

List<String> result = names.stream()

.filter(name -> name.startsWith("A"))

.collect(Collectors.toList());

**🔹 map()**

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

List<Integer> lengths = names.stream()

.map(String::length)

.collect(Collectors.toList()); // [5, 3]

**🔹 flatMap()**

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

Arrays.asList("A", "B"),

Arrays.asList("C", "D")

);

List<String> flat = nested.stream()

.flatMap(List::stream)

.collect(Collectors.toList()); // [A, B, C, D]

**💥 Reduce Example**

int sum = Arrays.asList(1, 2, 3, 4).stream()

.reduce(0, (a, b) -> a + b); // 10

**✅ Collectors Utility**

Collectors help gather results of streams into collections or strings.

List<String> names = Arrays.asList("A", "B", "C");

String joined = names.stream()

.collect(Collectors.joining(", ")); // "A, B, C"

Map<Integer, List<String>> grouped = names.stream()

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

**🧵 Parallel Streams**

list.parallelStream()

.filter(e -> e.startsWith("A"))

.forEach(System.out::println);

✅ Runs on multiple threads — great for large data, but use wisely.

**🧠 Stream Pipeline Structure**

A stream pipeline has:

1. **Source**: e.g., list.stream()
2. **Intermediate operations**: .filter(), .map(), etc.
3. **Terminal operation**: .collect(), .forEach(), etc.

List<String> result = list.stream()

.filter(name -> name.startsWith("A"))

.map(String::toUpperCase)

.collect(Collectors.toList());

**⚠️ Cautions**

* A stream can be used only **once** (reusing throws IllegalStateException)
* Not meant for modifying the source
* Use parallelStream() only for **CPU-heavy** tasks

**📊 Stream vs for-loop**

| **Aspect** | **Stream** | **for-loop** |
| --- | --- | --- |
| Style | Declarative | Imperative |
| Readability | Cleaner for transformations | Verbose |
| Threading | Easy with parallelStream() | Manual thread management |
| Performance | Better for large data (with care) | More control for small data |

**🧪 Real-World Use Case**

**Find Top 3 highest paid employees in a department:**

List<Employee> top3 = employees.stream()

.filter(e -> e.getDepartment().equals("IT"))

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

.limit(3)

.collect(Collectors.toList());

**📌 Summary**

| **Feature** | **Benefit** |
| --- | --- |
| Functional style | Easy to transform, filter, group data |
| Laziness | Efficient — operations are delayed |
| Composition | Chain multiple operations |
| Parallelism | Improves performance on multi-core CPUs |
| Clean syntax | Reduces boilerplate code |