SOME OF THE MOST IMPORTANT FEATURES OF

STREAM API

EVERY DEVELOPER SHOULD KNOW THIS

Sanjay Yadav
Software Developer

Why we need Stream API?

1. lets understand from example,if we want to iterate any list or array of integer and if have to find out sum of all the integer less than 1000 then below will be our normal code

```
private static int sumIterator(List<Integer> list) {
    Iterator<Integer> it = list.iterator();
    int sum = 0;
    while (it.hasNext()) {
        int num = it.next();
        if (num <1000) {
            sum += num;
        }
    }
    return sum;
}
```

important points to note

1. Program is sequential in nature, no way to do this parallel easily
2. A lot of code to do even for very simple tasks
3. External iteration is required

Sanjay Yadav
Software Developer

Using Stream API...

- we can use Stream API to implement internal interation
- less no. line of code using lamda expression in stream API
- parallel execution is possible

After Using Stream API for above CODE

```
private static int sumStream(List<Integer> list)
{
return list.stream().filter(i -> i >1000).mapToInt(i -> i).sum();
}
```

- Stream data structure that is computed on demand
- Java Stream operations use functional interfaces, that makes it a very good fit for functional programming using lambda expression

lets talks about main features of stream API

Sanjay Yadav
Software Developer

1. Declarative Style

- Unlike traditional loops, streams focus on what needs to be done rather than how.

```
// Traditional loop
for (int num : numbers) {
    if (num % 2 == 0) {
        System.out.println(num);
    }
}
```

```
// Stream equivalent
numbers.stream().filter(n -> n % 2 == 0).forEach(System.out::println);
```

Sanjay Yadav
Software Developer

2. Lazy evaluation

- **Streams execute only when necessary, improving performance by avoiding unnecessary computations.**

- **it also improve the efficiency.**

```
// This will print "Aman" only, not iterate further
names.stream().filter(name ->
name.startsWith("A")).findFirst().orElse("No
names");
```

Sanjay Yadav
Software Developer

3. Parallel Processing

- **Streams can leverage multicore architectures by allowing operations to be executed in parallel easily.**

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
long count = numbers.parallelStream()
                    .filter(n -> n % 2 == 0)
                    .count();
```

Sanjay Yadav
Software Developer

4.Functional Programming

- Streams support functional-style operations such as map, filter, reduce, and forEach, which allow concise and expressive code.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.stream()
    .map(n -> n * 2)
    .forEach(System.out::println);
```

Sanjay Yadav
Software Developer

5.Non-Mutating Operations

- **Stream operations do not modify the original data structure. They produce a new stream or collection with the desired transformations, ensuring immutability and avoiding unintended side effects.**

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> doubledNumbers = numbers.stream()
                                      .map(n -> n * 2)
                                      .collect(Collectors.toList());
```

Sanjay Yadav
Software Developer

Follow Me for This Type of Content

Sanjay Yadav

Software Developer