COPILOT

JAVA STREAM API: FROM BASICS TO ADVANCED PROGRAMMING

BASICS OF JAVA STREAM API

1. STREAM CREATION:

- **EMPTY STREAM:** Stream<String> streamEmpty = Stream.empty();
- **STREAM OF COLLECTION:** Collection<String> collection = Arrays.asList("a", "b", "c"); Stream<String> streamOfCollection = collection.stream();
- **STREAM OF ARRAY:** String[] arr = {"a", "b", "c"}; Stream<String> streamOfArray = Arrays.stream(arr);
- **STREAM.BUILDER():** Stream<String> streamBuilder = Stream.<String>builder().add("a").add("b").add("c").build();
- **STREAM.GENERATE():** Stream<String> streamGenerated = Stream.generate(() -> "element").limit(10);
- **STREAM.ITERATE():** Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);

2. INTERMEDIATE OPERATIONS:

- **FILTER:** stream.filter(element -> element.contains("a"));
- **MAP:** stream.map(String::toUpperCase);
- **SORTED:** stream.sorted();

3. TERMINAL OPERATIONS:

- **COLLECT:** List<String> result = stream.collect(Collectors.toList());
- **FOREACH:** stream.forEach(System.out::println);
- **REDUCE:** Optional<String> concatenated = stream.reduce((s1, s2) -> s1 + s2);

ADVANCED PROGRAMMING WITH STREAM API

1. PARALLEL STREAMS:

- List<String> list = Arrays.asList("a", "b", "c");
- list.parallelStream().forEach(System.out::println);

2. CUSTOM COLLECTORS:

- Collector<String, ?, TreeSet<String>> toTreeSet = Collectors.toCollection(TreeSet::new);
- TreeSet<String> result = stream.collect(toTreeSet);

3. FLATMAP:

- Stream<List<String>> listOfLists = Stream.of(Arrays.asList("a"), Arrays.asList("b"));
- Stream<String> flatStream = listOfLists.flatMap(Collection::stream);

USE CASES AND SOLUTIONS

1. FILTERING AND COLLECTING DATA:

- USE CASE: FILTER A LIST OF STRINGS TO FIND THOSE CONTAINING A SPECIFIC SUBSTRING AND COLLECT THEM INTO A NEW LIST.
- SOLUTION:

```
List<String> filteredList = list.stream()
                            .filter(s -> s.contains("example"))
                            .collect(Collectors.toList());
```

2. TRANSFORMING DATA:

- USE CASE: CONVERT A LIST OF STRINGS TO UPPERCASE.
- SOLUTION:

```
List<String> upperCaseList = list.stream()
                             .map(String::toUpperCase)
                             .collect(Collectors.toList());
```

3. AGGREGATING DATA:

- USE CASE: SUM A LIST OF INTEGERS.
- SOLUTION:

```
int sum = list.stream()
          .mapToInt(Integer::intValue)
          .sum();
```

4. PARALLEL PROCESSING:

- USE CASE: PROCESS A LARGE LIST OF DATA IN PARALLEL TO IMPROVE PERFORMANCE.
- SOLUTION:

```
list.parallelStream()
   .forEach(System.out::println);
```

THESE EXAMPLES ILLUSTRATE THE VERSATILITY AND POWER OF THE JAVA STREAM API, ENABLING EFFICIENT AND EXPRESSIVE DATA PROCESSING.