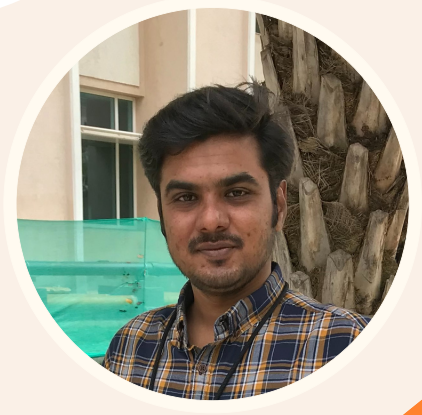


@iamgnanadeep

Java Streams

SWIPE

CloudnLoud Community

1

Stream

Stream is conceptually a pipeline in which elements are computed on demand.

java.util.Stream interface represents a stream on which one or more operations can be performed

@iamgnanadeep

2

Filtering Data

Streams can be used to filter data based on specific criteria

@iamgnanadeep

3

Use Case

We can filter a list of integers to get only the even numbers

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
List<Integer> evenNumbers = numbers.stream()
                                   .filter(n -> n % 2 == 0)
                                   .collect(Collectors.toList());
```

4

Mapping Data

Streams can transform data by applying a function to each element.

@iamgnanadeep

5

Use Case

We can convert a list of strings to uppercase:

```
List<String> words = Arrays.asList("hello", "world");
List<String> uppercaseWords = words.stream()
                                   .map(String::toUpperCase)
                                   .collect(Collectors.toList());
```

6

Reducing Data

Streams can be used to perform operations that reduce the stream to a single value

@iamgnanadeep

7

Use Case

Finding the sum of a list of numbers

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream()
                 .reduce(0, Integer::sum);
```

8

Grouping and Partitioning

Streams can be used to group data by a certain criterion or partition it into different sets

@iamgnanadeep

9

Use Case

We can group a list of people by their age

```
List<Person> people = /* List of Person objects */;
Map<Integer, List<Person>> peopleByAge = people.stream()
                                              .collect(Collectors.groupingBy(Person::getAge));
```

10

Sorting Data

Streams can be used to sort data according to specific criteria

@iamgnanadeep

11

Use Case

Sorting Names

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
List<String> sortedNames = names.stream()
                                .sorted()
                                .collect(Collectors.toList());
```

12

Performing Parallel Processing

Streams can be easily parallelized to take advantage of multi-core processors for enhanced performance.

@iamgnanadeep

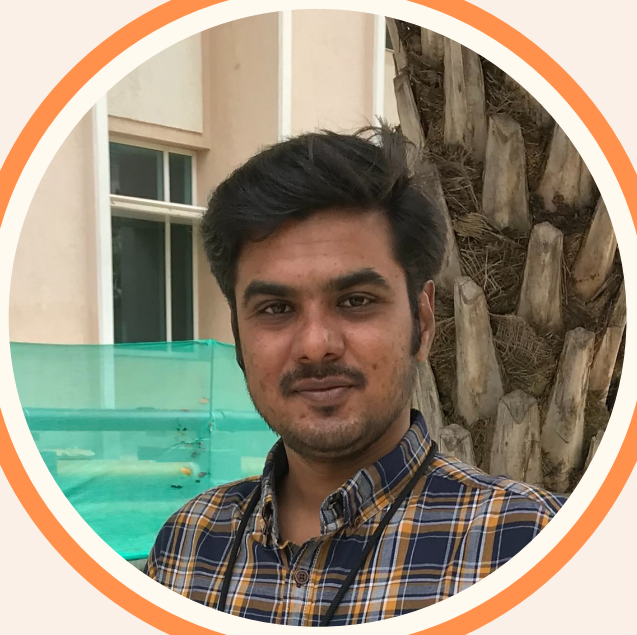
13

Use Case

You can convert a sequential stream into a parallel stream using parallel()

```
List<Integer> numbers = /* List of numbers */;
int sum = numbers.parallelStream()
                 .reduce(0, Integer::sum);
```

REPOST IF USEFUL



CloudnLoud Community

FOLLOW FOR MORE

@iamgnanadeep