Stream API

What is Stream API in Java 8?

A stream in Java 8 is a sequence of data. This data can be obtained from several sources such as Collections, Arrays or I/O channels. There are two types of Stream: Sequential and Parallel. We can perform sequential operations when we obtain a stream using the stream() method, and parallel operations with parallelStream() method.

All elements of a stream are not populated at a time. They are lazily populated as per demand because intermediate operations are not evaluated until terminal operation is invoked.

What are the ways of creating a Stream in Java?

1) Using Stream.of() method

The Stream.of() method takes a variable argument list of elements:
static <T> Stream<T> of(T... values)

```
Stream<Integer> streamOfIntegers = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9);
```

2) using stream() & parallelStream() methods

Java.util.Collection interface has stream() and parallelStream() methods to create sequential and parallel streams respectively.

3) Stream from an Array using Arrays.stream()

We can also create a stream from an Array using stream() method of java.util.Arrays class which accepts an array as argument as shown below.

String[] arr= new String[] { "a", "b", "c" };

Stream<String> streamOfStrings = Arrays.stream(arr);

4) Stream using Stream.builder()

```
Builder<String> builder = Stream.<String>builder(); // creating a
builder
builder.add("a").add("b").add("c"); // adding elements
Stream<String> s = builder.build(); // creating stream
```

5) Creating an Empty Stream using Stream.empty()

Stream<String> emptyStream = Stream.empty();

6) Creating an infinite Stream using Stream.generate() method

```
Random random = new Random();
Stream<Integer> stream =
                  Stream.generate(
                    () -> {return random.nextInt(100);} //
generating random numbers between 0 and 99
                  ).limit(5);
```

7) Creating an infinite Stream using Stream.iterate() method

There is another way of creating an infinite stream using iterate() method. For example, if we want to create a stream of odd numbers, we would do it as below:

Stream<Integer> streamOfOddNumbers = Stream.iterate(1, n -> n + 2);

iterate() takes a seed or starting value as the first parameter. This is the first element that will be part of the stream. The other parameter is a lambda expression that gets passed the previous value and generates the next value. In this example, the next value after 1 will be 3. As with the

random numbers example, it will keep on producing odd numbers as long as you need them.

What is Stream of Primitives and Why?

As we know that the Streams primarily work with collections of objects. Moreover, Stream<T> is a generic interface, and there is no way to use primitives as a type parameter with generics. Therefore, three new special interfaces were created: IntStream, LongStream, DoubleStream for three primitive types: integer, long and double respectively. Furthermore, using the new interfaces reduces needless auto-boxing, which allows for increased productivity. IntStream and LongStream each have two additional factory methods for creating streams, range and rangeClosed. Their method signatures are similar:

```
static IntStream range(int startInclusive, int endExclusive)

static IntStream rangeClosed(int startInclusive, int endInclusive)

static LongStream range(long startInclusive, long endExclusive)

static LongStream rangeClosed(long startInclusive, long endInclusive)
```

The arguments show the difference between the two: rangeClosed includes the end value, and range doesn't. Each returns a sequential, ordered stream that starts at the first argument and increments by one after that.

1) Using the boxed method : boxed() method converts int to Integer. We can use the boxed method on Stream to convert the IntStream to a Stream<Integer> as shown below:

```
List<Integer> ints = IntStream.of(1, 2, 3, 4, 5, 6)

        .boxed()

        .collect(Collectors.toList());
```

2) Using the mapToObj method : The mapToObj() method converts each element from a primitive to an instance of the wrapper class as below:

```
List<Integer> ints = IntStream.of(1, 2, 3, 4, 5, 6)

        .mapToObj(Integer::valueOf)

        .collect(Collectors.toList());
```

Just as mapToInt, mapToLong, and mapToDouble parse streams of objects into the associated primitives, the mapToObj method from IntStream, LongStream, and Double Stream converts primitives to instances of the associated wrapper classes. The argument to mapToObj in this example uses the Integer constructor.

Converting an IntStream to an int array

```
int[] intArray = IntStream.of(1, 2, 3, 4, 5, 6).toArray();

// OR

int[] intArray = IntStream.of(1, 2, 3, 4, 5, 6).toArray(int[]::new);
```

Here, the first demo uses the default form of toArray, which returns Object[]. The second uses an IntFunction<int[]> as a generator, which creates an int[] of the proper size and populates it.

Intermediate Operations

For example: *map(), filter(), distinct(), sorted(), limit(), skip()*

Terminal Operations

forEach(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny()

Note: Stream processing consists of a series of zero or more intermediate operations followed by a terminal operation. Each intermediate operation returns a new stream. The terminal operation returns something other than a stream.

Below are some approaches to print streams:

1) s.forEach(System.*out*::println);

2) System.*out*.println(s.collect(Collectors.toList()));

3) s.limit(4).forEach(System.*out*::println);

4) s.peek(System.*out*::println).count();

How to use Stream Intermediate Operations?

filter( )

Method signature: Stream<T> filter(Predicate<? super T> predicate)

distinct( )

Method signature: Stream<T> distinct()

When to use distinct( )?

If you want to return a stream from another stream with duplicate values removed.

limit( ) and skip( )

Method signature: Stream<T> limit(int maxSize)
Stream<T> skip(int n)

When to use limit( ) and skip( )?

If you want to make your stream smaller. Also, if you want to make a finite stream out of an infinite stream.

The following code creates an infinite stream of numbers counting from 1. The skip() operation returns an infinite stream starting with the numbers counting from 10, as it skips the first nine elements because of skip(9). The limit() call takes the first six of those. Now we have a finite stream with six elements:

```
Stream<Integer> s = Stream.iterate(1, n -> n + 1);
s.skip(9).limit(6).forEach(System.out::print);
```

map( ) vs flatMap( )

Method signatures:

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)
```

When to use map( ) and flatMap( )?

If you want to transform the elements of a stream in some way. Use map() if you want to transform each element into a single value.

Use flatMap() if you want to transform each element to multiple values and also compress/flatten the resulting stream.

What is the difference between map( ) and flatMap( )?

The additional word 'flat' in flatMap() method indicates the flattening, which is the additional task done by flatMap(). However below is the list of common differences between them.

map( )

1) It works on stream of values.
2) It performs the only transformation.
3) It produces a single value for each input value.

flatMap( )

1) It works on a stream of stream of values.
2) It performs transformation as well as flattening.
3) It produces multiple values for each input value.

```
List<Programmer> listOfProgrammers = List.of(

    new Programmer("Programmer1", List.of("Java", "Python", "Angular")),

    new Programmer("Programmer2", List.of("Ruby", "Angular", "Java")),

    new Programmer("Programmer3", List.of("React", "Spring", "Angular"))

);
```

/ /Extracting the name of all Programmers using stream API in Java 8

```
listOfProgrammers.stream().map(Programmer::getName)

.collect(Collectors.toList()) .forEach(System.out::println);
```

//Now, let's use flatMap() to extract distinct skills out of all programmers as below.

```
listOfProgrammers.stream()

    .flatMap(p -> p.getSkills().stream())

    .collect(Collectors.toSet())

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

sorted( )

Method signature:

Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comparator)

When to use sorted( )?

When we need to return a stream with the elements sorted. Just like sorting arrays, Java uses natural ordering unless we specify a comparator.

peek()

Method signature: Stream<T> peek(Consumer<? super T> action)

When to use peek( )?

Sometimes we need to perform some operations on each element of the stream before any terminal operation is applied. In fact, peek() performs the specified operation on each element of the stream and returns a new stream that we can further use. It is useful for debugging because it allows us to perform a stream operation without actually changing the stream. The most common use for peek() is to output the contents of the stream before any terminal operation is applied.

Example

```
public int triplesDivisibleBy2Sum(int start, int end) {

   return IntStream.rangeClosed(start, end)

    .peek(n -> System.out.println("original element : " +n))        //prints value before multiplying by 3

    .map(n -> n * 3)

    .peek(n -> System.out.println("Tripled element : " +n))        //prints value before filtering

    .filter(n -> n % 2 == 0)

    .peek(n -> System.out.println("Divisible By 2 element : " +n))   //prints value after filtering but before summing

    .sum();

}
```

Stream Terminal Operations

count( )

Method signature: long count()

When to use count( )?

When you want to determine the number of elements in a finite stream.

min( ) and max( )

Method signature: Optional<T> min(<? super T> comparator)
Optional<T> max(<? super T> comparator)

When to use min( ) and max( )?

when you want to find the smallest or largest value in a finite stream. As the method signature represents, the min() and max() methods allow us to pass a custom comparator and find the smallest or largest value in a finite stream according to that sort order. Like

count(), min() and max() works on a finite stream. Also like count(), both methods are reductions because they return a single value after looking at the entire stream.

Example

This example finds the programming language with the fewest letters in its name using stream API in Java 8. Notice that the code returns an Optional rather than the value. We use the Optional method and a method reference to print out the minimum only if one is found.

```
Stream s = Stream.of("Java", "Python", "Scala");
Optional min = s.min((s1, s2) -> s1.length() — s2.length());
min.ifPresent(System.out::println);
```

findAny( ) and findFirst( )

Method signature: Optional<T> findAny()
Optional<T> findFirst()

When to use findAny( ) and findFirst( )?

You wish to find the first element in a stream that satisfies a particular condition then use findFirst(). The findFirst() and findAny() methods in Stream return an Optional describing the first element of a stream. Neither takes an argument, implying that any mapping or filtering operations have already been done.

The findAny() method returns an Optional describing some element of the stream, or an empty Optional if the stream is empty. In a non-parallel operation, findAny() will most likely return the first element in the Stream, but there is no guarantee for this. For maximum performance when processing the parallel operation, the result cannot be reliably determined.

Example

For example, given a list of integers, to find the first even number, apply an even number filter and then use findFirst() using stream API in Java 8, as in example below.

```
Optional firstEvenNumber = Stream.of(9, 5, 8, 7, 4, 9, 2, 11, 3)

 .filter(n -> n % 2 == 0).findFirst();

System.out.println(firstEvenNumber);
```

anyMatch( ), allMatch( ), and noneMatch( )

Method signature:

boolean anyMatch(Predicate <? super T> predicate)
boolean allMatch(Predicate <? super T> predicate)
boolean noneMatch(Predicate <? super T> predicate)

When to use anyMatch( ), allMatch( ), and noneMatch( )?

When you wish to determine if any elements in a stream match a Predicate, or if all match, or if none match, then use the methods anyMatch(), allMatch(), and noneMatch() respectively.

```
List<String> listOfSkills = Arrays.asList("Core Java", "Spring Boot", "Hibernate", "Angular");

Predicate<String> pred = x -> x.startsWith("S");

System.out.println(listOfSkills.stream().anyMatch(pred));        // true

System.out.println(listOfSkills.stream().allMatch(pred));        // false

System.out.println(listOfSkills.stream().noneMatch(pred));     //false
```

forEach( )

Method signature: void forEach(Consumer<? super T> action)

When to use forEach( )?

Needless to mention, when we want to iterate the elements of a stream. Notice that this is the only terminal operation with a return type of void. Moreover, note that you can call forEach() directly on a Collection or on a Stream. Stream API in Java 8 cannot use a traditional for loop to run because they don't implement the Iterable interface.

Example

Below code example demonstrates the concept using Stream API in Java 8.

```
Stream<String> streamofSkills = Stream.of("Java", "Python", "Angular");

streamofSkills.forEach(System.out::println);
```

collect( )

Please note that the collect() method doesn't belong to the Collectors class. It is defined in Stream class and that's the reason you can call it on Stream after doing any filtering or mapping operations. However, it accepts a Collector to accumulate elements of Stream into a specified Collection.

Method signature:

<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)

<R, A> R collect(Collector<? super T, A, R> collector)

```
Stream<String> streamOfSkills = Stream.of("Java", "Scala", "Python", "Spring");

 streamOfSkills

    .filter(x -> x.startsWith("S"))

    .collect(Collectors.toList())     //Collecting the result of a stream into a List

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

Let's take an example of stream to convert it into a collection of your choice like ArrayList, HashSet, LinkedList etc. For example, let's assume the same stream from previous example.

```
streamOfSkills

 .filter(x -> x.length() > 3)

 .collect(Collectors.toCollection(ArrayList::new))    //Collecting the result of a stream into a List of our choice

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

reduce()

The reduce() method combines a stream into a single object. As we can tell from the name, it is a reduction.

Method signature:

```
T reduce(T identity, BinaryOperator<T> accumulator)
Optional<T> reduce(BinaryOperator<T> accumulator)
<U> U reduce(U identity,  BiFunction<U, ? super T, U> accumulator,
BinaryOperator<U> combiner)
```

Identity: An element that is the initial value of the reduction operation and the default result if the stream is empty

Accumulator: A function that takes two parameters: a partial result of the reduction operation and the next element of the stream

Combiner: A function that we use to combine the partial result of the reduction operation when the reduction is parallelized or when there's a mismatch between the types of the accumulator arguments and the types of the accumulator implementation

When to use reduce( )?

When you wish to produce one single result from a sequence of elements, by repeatedly applying a combining operation to the elements in the sequence.

or example, let's observe the below code snippet and understand the reduce() in a better way under stream API in Java 8.

```
List<String> letters = Arrays.asList("j", "a", "v", "a", "t", "e", "c", "h", "o", "n", "l", "i", "n", "e");
String result = letters .stream()
.reduce(" ", (partialString, element) -> partialString + element);
System.out.println(result);
```

In the example above, " " is the identity. It indicates the initial value of the reduction operation and also the default result when the stream of String values is empty. Likewise, the lambda expression "(partialString, element) -> partialString + element" is the accumulator as it takes the partial concatenation of String values and the next element in the stream.

Of course, we can also change it to the expression that uses a method reference like as below using Stream API in Java 8:

```
String result = letters.stream().reduce(" ", String::concat);
```

Output

javatechonline

Likewise, let's observe an example of Stream of Integers as below using Stream API in Java 8:

```
BinaryOperator op = (a, b) -> a * b;

Stream empty = Stream.empty();

Stream oneElement = Stream.of(3);

Stream threeElements = Stream.of(3, 4, 5);

empty.reduce(op).ifPresent(System.out::print); // no output

oneElement.reduce(op).ifPresent(System.out::print); // 3

threeElements.reduce(op).ifPresent(System.out::print); // 60
```