Streams

Shristi Technology Labs

Contents

- Streams
  - Intermediate Operations
  - Terminal Operations

Streams

- A Stream represents a sequence of elements and supports parallel and aggregate operations.
- Streams are abstraction for processing collections of values.
- Streams can be created from collections, arrays, or iterators.
- Streams allows to transform and retrieve data
- A stream does not store its elements
- Stream operations don’t change their source
- Returns new stream holding the result.
- Have Specialized Streams for primitive data types

Creation of Streams

- To convert a collection to stream, use ***stream()*** from Collection
- To convert an Array to a stream, use the static ***Stream.of***
- To make a stream from a part of the array, use ***Arrays.stream(array, from, to)***
- To make a stream with no elements, use static ***Stream.empty()***
- To make a stream with infinite elements, use static ***Stream.generate()***

Types of Streams

- **stream()**
  - Returns a sequential stream with collection as its source.
- **parallelStream()**
  - Returns a parallel Stream with collection as its source.
  - Convert streams into parallel stream to do filtering and counting in parallel

Streams can hold different data types

- Stream<String[]>
- Stream<Set<String>>
- Stream<List<String>>
- Stream<List<Object>>
- Stream<Integer>

Example

```
//converting a List to a stream
List<String> wordList = Arrays.asList("Hi","Welcome","to","Streams");
Stream<String> listStreams = wordList.stream();

//converting an array to a Stream
Stream<String> arrayStream = Stream.of("Have","a","good","day",".");
Stream<Integer> intStream = Arrays.stream(new Integer[]{10,20,40});

//Creating an empty stream
Stream<String> emptyStream = Stream.empty();

//Creating an infinite stream
Stream<String> infiniteStream = Stream.generate(()->"Hello");
Stream<Double> infiniteStream1 = Stream.generate(Math::random);
```

Operations on Streams

- Stream operations are divided into
  - Intermediate operations – returns a new stream
  - Terminal operation – returns a result

- Operations are combined to form stream pipelines.

- A stream pipeline consists of a source (such as a Collection, an array, a generator function) followed by zero or more intermediate operations and ending with a terminal operation
- Stream operations do the iterations internally over the source elements provided.

Intermediate operations

- Intermediate operations always return a new stream.
  - **eg.** `filter(), map(), limit(), skip(), distinct(), sorted()`
- Also called as lazy operations
- Multiple intermediate operations can be chained together
- The intermediate operation will NOT begin until the terminal operation of the pipeline is called.

Terminal operation

- The terminal operation traverses through the stream to produce a result.
- Also called as early operations
- After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used
- **eg.** `iterator(), reduce(), forEach(), findFirst(), match(), collect(), count()`

Pipeline of operations

- Create a stream.
- Specify **intermediate operations** *for transforming the initial stream into* another stream
- Apply a ***terminal operation*** *to produce a result.*
- Next, the stream can no longer be used.

Intermediate Operation

Stream Operations - filter

filter

- reads data from a stream and yields a new stream with all elements that match a certain condition.
- Is an intermediate operation ***Stream<T> filter(Predicate<T> predicate)***

```
//converting a List to a stream
List<String> wordList = Arrays.asList("Hi","Welcome","to","streams");
Stream<String> listStreams = wordList.stream();
Stream<String> longwords =listStreams.filter((w)->w.length()>3);
Iterator<String> it1 = longwords.iterator();
while (it1.hasNext()) {
    String word =  it1.next();
    System.out.print(word+" ");
}
```

Output: **Welcome streams**

Stream Operations - filter

Using method chaining

```
Stream<String> words =Arrays.asList("Hi","Welcome","Hi","Streams")
                    .stream()
                    .filter((x)->x.length()>2);
```

Output: **Welcome streams**

Stream Operations - map

map

- to transform the values in a stream as lowercase, uppercase, getting characters. etc
- Is an ***intermediate*** operation ***Stream<T > map(Function <T> val)***

```
List<String> wordList = Arrays.asList("Hi","Welcome","to","Streams");
Stream<String> listStreams = wordList.stream();
Stream<String> longwords =listStreams.map(String::toLowerCase);
System.out.print("Long words - ");
Iterator<String> it = longwords.iterator();
while (it.hasNext()) {
    String word =  it.next();
    System.out.print(word+" ");
}
```

Output: **welcome streams**

Stream Operations - flatMap

flatMap

- to transform the values in a stream as lowercase, uppercase, getting characters etc.
- used when the stream is of type *List*, *Set* or *Array* like String<List<String>>
- Is an ***intermediate*** operation ***Stream<T> flatMap(Function <T>)***

```
Stream<String[]>      -> flatMap ->  Stream<String>
Stream<Set<String>>   -> flatMap ->  Stream<String>
Stream<List<String>>  -> flatMap ->  Stream<String>
Stream<List<Object>>  -> flatMap ->   Stream<Object>
```

Example – map & flatMap

```
String[] arrOne = new String[]{"Java","Spring","Angular"};
//String[] to String
Stream<String> ostreams = Arrays.stream(arrOne);
ostreams.map((x)->x.toLowerCase()).sorted().forEach(System.out::println);

String[][] arrTwo = new String[][]{{"Ram","Tom"},{"Zeena","Meena"}};
//String[][] to String[]
Stream<String[]> tstreams = Arrays.stream(arrTwo);
//String[] to String
Stream<String> newstream = tstreams.flatMap((one)->Arrays.stream(one));
newstream.map((x)->x.toUpperCase()).forEach(System.out::println);
```

Output:
java
spring
angular

Output:
RAM
TOM
ZEENA
MEENA

Stream Operations – limit, skip

limit

- to limit the number of elements to the size given **Stream<T> limit(long size)**

```
System.out.println("Infinite Numbers -10");
Stream<Double> infiniteStream1 = Stream.generate(Math::random)
                                 .limit(10);
```

skip

- to discard the first n elements **Stream<T> skip(long n)**

```
System.out.println("Infinite Numbers -10");
Stream<Double> infiniteStream1 = Stream.generate(Math::random)
                                .limit(10).skip(2);
```

Stream Operations - distinct

distinct

- Returns a stream consisting of the distinct elements. For ordered streams, the selection is stable (element appearing first will be preserved, in case of duplicate elements.)
- For unordered streams, no stability guarantees are made

Stream<T> distinct()

```
List<String> newWords = Arrays.asList("Hi","Welcome","Hi","Streams");
Stream<String> distinctWords =newWords.stream().distinct();
System.out.println("printing unique words");
Iterator<String> dist = distinctWords.iterator();
while (dist.hasNext()) {
    String word =  dist.next();
    System.out.print(word+" ");
}
```

Output: **Hi Welcome Streams**

Stream Operations - sorted

sorted

- Returns a stream consisting of the elements, sorted according to natural order.
- For ordered streams, the sorting is stable
- For unordered streams, *ClassCastException* will be thrown when a terminal operation is called

Stream<T> sorted()

```
Stream.of("Java","Spring","Angular","Node","Javascript","Ember")
.map((x)->x.toUpperCase())
.sorted().forEach(System.out::println);
```

Output:
ANGULAR
EMBER
JAVA
JAVASCRIPT
NODE
SPRING

Stream Operations - concat

concat

- Concatenate two streams using concat method.
- For ordered streams, the selection is stable (element appearing first will be preserved, in case of duplicate elements.)
- For unordered streams, no stability guarantees are made

Stream<T> concat(Stream a, Stream b)

```
Stream.concat(Stream.of("Hello","World","Hi"), Stream.of("World"))
    .distinct()
    .filter(x->x.length()>2)
    .sorted()
    .forEach(System.out::println);
```

Output: **Hello World**

Terminal Operation

Stream Operations - iterator

iterator

- Is a ***terminal*** operation ***Iterator<T> iterator()***
- To iterate the elements in the stream

```
List<String> wordList = Arrays.asList("Hi","Welcome","to","Streams");
Stream<String> listStreams = wordList.stream();
Stream<String> longwords =listStreams.map(String::toLowerCase);
System.out.print("Long words - ");
Iterator<String> it = longwords.iterator();
while (it.hasNext()) {
    String word =  it.next();
    System.out.print(word+" ");
}
```

Output: **welcome streams**

Stream Operations - forEach

forEach

- to iterate each element of the stream.
- Is a ***terminal*** operation

void forEach(Consumer<>)

```
Arrays.asList("Hi","Hello","welcome")
.stream().forEach(System.out::println);
```

Output:
Hi
Hello
welcome

Stream Operations - findFirst

findFirst

- finds the first element in a *Stream.*
- returns an *Optional* instance which is empty if the *Stream* is empty:
- Is a ***terminal*** operation

Optional<T> findFirst()

```
Optional<String> result = Arrays.asList("Hi","Hello","welcome")
                          .stream().findFirst();
result.ifPresent(System.out::println);
```

```
Arrays.asList("Hi","Hello","welcome")
.stream()
.findFirst()
.ifPresent(System.out::println);
```

Output: Hi

Stream Operations - orElse

orElse

- Checks for value in Optional, if present returns that value, else returns the value passed in the argument.
- Is a ***terminal*** operation

T orElse(T other)

```
String name = Arrays.asList("Archie","Bella","Juno","Scooby")
              .stream()
              .filter((x)->x.equals("Bela"))
              .findAny()
              .orElse("Not Available");
System.out.println(name);

Integer val = Arrays.asList(5,7,9,11).stream()
         .filter((x)->x%2==0)
         .findFirst()
         .orElse(0);
System.out.println(val);
```

Output:

Not Available

0

Stream Operations - orElseGet

orElseGet

- Checks for value in Optional, if present returns that value, else returns the result of code inside the argument.
- Is a ***terminal*** operation

orElseGet(Supplier<T> other)

```
String name = Arrays.asList("Archie","Bella","Juno","Scooby")
              .stream()
              .filter((x)->x.equals("Bela"))
              .findAny()
              .orElseGet(()->{return "not available";});
System.out.println(name);
```

Output: not available

Stream Operations - collect

collect

- The object resulting from intermediate operations will be collected

List<T> collect(Collectors List<Collector> collector)

```
Arrays.asList("Archie","Arav","Juno","Scooby")
.stream()
.filter((x)->x.startsWith("A"))
.collect(Collectors.toList()).forEach(System.out::println);
```

Output:
Archie
Arav

Stream Operations - reduce

reduce with Accumulator

- Allows to calculate result using all the elements in the stream.
- Allows to perform arithmetic operations like *sum*, *average* and *count* on stream objects and get numbers as results.
- Is a Terminal Operation

Optional<T> reduce(BinaryOperator accumulator)

```
Arrays.asList(10,20,30).stream()
 .reduce((x,y)-> x+y).ifPresent(System.out::println);
```

Output: 60

Stream Operations - reduce

reduce with identity & Accumulator

- Allows to calculate result using all the elements in the stream.
- Result will be identity + sum of array. .
- Is a Terminal Operation

T reduce(T identity, BinaryOperator<T> accumulator)

```
int startVal = 100;
int sum = Arrays.asList(110,120,130).stream()
 .reduce(startVal,(x,y)-> x+y);
System.out.println("Sum "+sum);
```

Output: Sum 460

Stream Operations - mapToInt

mapToInt

- Returns an IntStream with the results of applying the given function to the elements of this stream

IntStream mapToInt(ToIntFunction<? super T> mapper)

```
int total = Arrays.asList("Archie", "Arav", "Juno", "Scooby")
.stream()
.map(s -> s.length())
.mapToInt(Integer::new )
.sum();
System.out.println(total);

Arrays.asList("900", "567", "876", "911")
.stream()
.mapToInt(s->Integer.parseInt(s))
.average().ifPresent(System.out::println);
```

Output:
20
813.5

Streams for primitives

- Special streams for working with the primitive data types int, long and double.
  - IntStream
  - LongStream
  - DoubleStream
- Uses specialized lambda expressions –

e.g. IntFunction , IntPredicate

- Supports the terminal aggregate operations sum() and average()

```
IntStream stream = Arrays.stream(new int[] { 40, 20, 30, 91, 16, 76 });
int sum = stream.filter(x -> x > 20).sum();
System.out.println(sum);
```

Example

```
IntStream.range(10,20).forEach(System.out::println);

LongStream.of(911,322,673,184,295)
    .filter(x->x>300)
    .forEach(System.out::println);

int sum = IntStream.range(10,20).sum();
System.out.println("Sum " +sum);

Arrays.stream(new int[]{10,20,30})
     .average()
     .ifPresent(System.out::println);
```

Summary

- Streams

Thank you