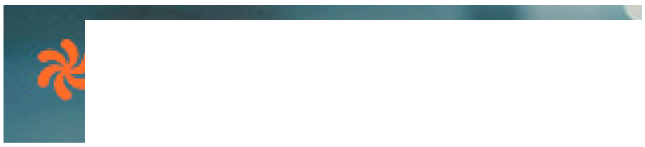
Java 8 Stream – Java Stream

PANKAJ — 54 COMMENTS



Welcome to Java 8 Stream API tutorial. In the last few java 8 posts, we looked into Java 8 Interface Changes and Functional Interfaces and Lambda Expressions. Today we will look into one of the major API introduced in Java 8 – **Java Stream**.

Java 8 Stream

1. Java 8 Stream
2. Collections and Java Stream

Java Stream

Before we look into Java Stream API Examples, let's see why it was required. Suppose we want to iterate over a list of integers and find out sum of all the integers greater than 10.

Prior to Java 8, the approach to do it would be:

```
private static int sumIterator(List<Integer>
list) {
        Iterator<Integer> it = list.iterator();
        int sum = 0;
        while (it.hasNext()) {
                int num = it.next();
                if (num > 10) {
                        sum += num;
                }
        }
        return sum;
}
```

There are three major problems with the above approach:

1. We just want to know the sum of integers but we would also have to provide how the iteration will take place, this is also called **external iteration** because client program is handling the algorithm to iterate over the list.
2. The program is sequential in nature, there is no way we can do this in parallel easily.
3. There is a lot of code to do even a simple task.

To overcome all the above shortcomings, Java 8 Stream API was introduced. We can use Java Stream API to implement **internal iteration**, that is better because java framework is in control of the iteration.

Internal iteration provides several features such as sequential and parallel execution, filtering based on the given criteria, mapping etc.

Most of the Java 8 Stream API method arguments are functional interfaces, so lambda expressions work very well with them. Let's see how can we write above logic in a single line statement using Java Streams.

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

Notice that above program utilizes java framework iteration strategy, filtering and mapping methods and would increase efficiency.

First of all we will look into the core concepts of Java 8 Stream API and then we will go through some examples for understanding most commonly used methods.

Collections and Java Stream

A collection is an in-memory data structure to hold values and before we start using collection, all the values should have been populated. Whereas a java Stream is a data structure that is computed on-demand.

Java Stream doesn't store data, it operates on the source data structure (collection and array) and produce pipelined data that we can use and perform specific operations. Such as we can create a stream from the list and filter it based on a condition.

Java Stream operations use functional interfaces, that makes it a very good fit for functional programming using lambda expression. As you can see in the above example that using lambda expressions make our code readable and short.

Java 8 Stream internal iteration principle helps in achieving lazy-seeking in some of the stream operations. For example filtering, mapping, or duplicate removal can be implemented lazily, allowing higher performance and scope for optimization.

Java Streams are consumable, so there is no way to create a reference to stream for future usage. Since the data is on-demand, it's not possible to reuse the same stream multiple times.

Java 8 Stream support sequential as well as parallel processing, parallel processing can be very helpful in achieving high performance for large collections.

All the Java Stream API interfaces and classes are in the `java.util.stream` package. Since we can use primitive data types such as int, long in the collections using auto-boxing and these operations could take a lot of time, there are specific classes for primitive types – `IntStream`, `LongStream` and `DoubleStream`.

Functional Interfaces in Java 8 Stream

Some of the commonly used functional interfaces in the Java 8 Stream API methods are:

1. **Function and BiFunction**: Function represents a function that takes one type of argument and returns another type of argument. `Function<T, R>` is the generic form where T is the type of the input to the function and R is the type of the result of the function. For handling primitive types, there are specific Function interfaces – `ToIntFunction`, `ToLongFunction`, `ToDoubleFunction`, `ToIntBiFunction`, `ToLongBiFunction`, `ToDoubleBiFunction`, `LongToIntFunction`, `LongToDoubleFunction`, `IntToLongFunction`, `IntToDoubleFunction` etc.

   Some of the Stream methods where `Function` or it's primitive specialization is used are:

   - <R> Stream<R> map(Function<? super T, ? extends R> mapper)
   - IntStream mapToInt(ToIntFunction<? super T> mapper) – similarly for long and double returning primitive specific stream.
   - IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper) – similarly for long and double
   - <A> A[] toArray(IntFunction<A[]> generator)
   - <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)

2. **Predicate and BiPredicate**: It represents a predicate against which elements of the stream are tested. This is used to filter elements from the java stream. Just like `Function`, there are primitive specific interfaces for int, long and double.
   Some of the Stream methods where `Predicate` or `BiPredicate` specializations are used are:

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

3. **Consumer and BiConsumer**: It represents an operation that accepts a single input argument and returns no result. It can be used to perform some action on all the elements of the java stream. Some of the Java 8 Stream methods where `Consumer`, `BiConsumer` or it's primitive specialization interfaces are used are:

    - Stream<T> peek(Consumer<? super T> action)
    - void forEach(Consumer<? super T> action)
    - void forEachOrdered(Consumer<? super T> action)

4. **Supplier**: Supplier represent an operation through which we can generate new values in the stream. Some of the methods in Stream that takes `Supplier` argument are:
    - public static<T> Stream<T> generate(Supplier<T> s)
    - <R> R collect(Supplier<R> supplier,BiConsumer<R, ? super T> accumulator,BiConsumer<R, R> combiner)

java.util.Optional

Java Optional is a container object which may or may not contain a non-null value. If a value is present, `isPresent()` will return true and `get()` will return the value. Stream terminal operations return Optional object. Some of these methods are:

- Optional<T> reduce(BinaryOperator<T> accumulator)
- Optional<T> min(Comparator<? super T> comparator)
- Optional<T> max(Comparator<? super T> comparator)

- Optional<T> findFirst()
- Optional<T> findAny()

java.util.Spliterator

For supporting parallel execution in Java 8 Stream API, `Spliterator` interface is used. Spliterator `trySplit` method returns a new Spliterator that manages a subset of the elements of the original Spliterator.

Java Stream Intermediate and Terminal Operations

Java Stream API operations that returns a new Stream are called intermediate operations. Most of the times, these operations are lazy in nature, so they start producing new stream elements and send it to the next operation. Intermediate operations are never the final result producing operations. Commonly used intermediate operations are `filter` and `map`.

Java 8 Stream API operations that returns a result or produce a side effect. Once the terminal method is called on a stream, it consumes the stream and after that we can't use stream. Terminal operations are eager in nature i.e they process all the elements in the stream before returning the result. Commonly used terminal methods are `forEach`, `toArray`, `min`, `max`, `findFirst`, `anyMatch`, `allMatch` etc. You can identify terminal methods from the return type, they will never return a Stream.

Java Stream Short Circuiting Operations

An intermediate operation is called short circuiting, if it may produce finite stream for an infinite stream. For example `limit()` and `skip()` are two short circuiting intermediate operations.

A terminal operation is called short circuiting, if it may terminate in finite time for infinite stream. For example `anyMatch`, `allMatch`, `noneMatch`, `findFirst` and `findAny` are short circuiting terminal operations.

Java Stream Examples

I have covered almost all the important parts of the Java 8 Stream API. It's exciting to use this new API features and let's see it in action with some java stream examples.

Creating Java Streams

There are several ways through which we can create a java stream from array and collections. Let's look into these with simple examples.

1. We can use `Stream.of()` to create a stream from similar type of data. For example, we can create Java Stream of integers from a group of int or Integer objects.

```
Stream<Integer> stream =
Stream.of(1,2,3,4);
```

2. We can use `Stream.of()` with an array of Objects to return the stream. Note that it doesn't support autoboxing, so we can't pass primitive type array.

```
Stream<Integer> stream = Stream.of(new
Integer[]{1,2,3,4});
//works fine

Stream<Integer> stream1 = Stream.of(new
int[]{1,2,3,4});
//Compile time error, Type mismatch: cannot
convert from Stream<int[]> to
Stream<Integer>
```

3. We can use Collection `stream()` to create sequential stream and `parallelStream()` to create parallel stream.

```
List<Integer> myList = new ArrayList<>();
for(int i=0; i<100; i++) myList.add(i);

//sequential stream
Stream<Integer> sequentialStream =
myList.stream();
```

```
//parallel stream
Stream<Integer> parallelStream =
myList.parallelStream();
```

4. We can use `Stream.generate()` and `Stream.iterate()` methods to create Stream.

```
Stream<String> stream1 = Stream.generate(()
-> {return "abc";});
Stream<String> stream2 =
Stream.iterate("abc", (i) -> i);
```

5. Using `Arrays.stream()` and `String.chars()` methods.

```
LongStream is = Arrays.stream(new long[]
{1,2,3,4});
IntStream is2 = "abc".chars();
```

Converting Java Stream to Collection or Array

There are several ways through which we can get a Collection or Array from a java Stream.

1. We can use java Stream `collect()` method to get List, Map or Set from stream.

```
Stream<Integer> intStream =
Stream.of(1,2,3,4);
List<Integer> intList =
intStream.collect(Collectors.toList());
System.out.println(intList); //prints [1,
2, 3, 4]

intStream = Stream.of(1,2,3,4); //stream is
closed, so we need to create it again
Map<Integer,Integer> intMap =
intStream.collect(Collectors.toMap(i -> i,
i -> i+10));
```

```
System.out.println(intMap); //prints {1=11,
2=12, 3=13, 4=14}
```

2. We can use stream `toArray()` method to create an array from the stream.

```
Stream<Integer> intStream =
Stream.of(1,2,3,4);
Integer[] intArray =
intStream.toArray(Integer[]::new);
System.out.println(Arrays.toString(intArray)
 //prints [1, 2, 3, 4]
```

Java Stream Intermediate Operations

Let's look into commonly used java Stream intermediate operations example.

1. **Stream filter() example**: We can use filter() method to test stream elements for a condition and generate filtered list.

```
List<Integer> myList = new ArrayList<>();
for(int i=0; i<100; i++) myList.add(i);
Stream<Integer> sequentialStream =
myList.stream();

Stream<Integer> highNums =
sequentialStream.filter(p -> p > 90);
//filter numbers greater than 90
System.out.print("High Nums greater than
90=");
highNums.forEach(p -> System.out.print(p+"
"));
//prints "High Nums greater than 90=91 92
93 94 95 96 97 98 99 "
```

2. **Stream map() example**: We can use map() to apply functions to an stream. Let's see how we can use it to apply upper case function to a list of Strings.

```
Stream<String> names = Stream.of("aBc",
"d", "ef");
System.out.println(names.map(s -> {
                return s.toUpperCase();
        }).collect(Collectors.toList()));
//prints [ABC, D, EF]
```

3. **Stream sorted() example**: We can use sorted() to sort the stream elements by passing Comparator argument.

```
Stream<String> names2 = Stream.of("aBc",
"d", "ef", "123456");
List<String> reverseSorted =
names2.sorted(Comparator.reverseOrder()).col

System.out.println(reverseSorted); // [ef,
d, aBc, 123456]

Stream<String> names3 = Stream.of("aBc",
"d", "ef", "123456");
List<String> naturalSorted =
names3.sorted().collect(Collectors.toList())

System.out.println(naturalSorted);
//[123456, aBc, d, ef]
```

4. **Stream flatMap() example**: We can use flatMap() to create a stream from the stream of list. Let's see a simple example to clear this doubt.

```
Stream<List<String>> namesOriginalList =
Stream.of(
        Arrays.asList("Pankaj"),
        Arrays.asList("David", "Lisa"),
        Arrays.asList("Amit"));
//flat the stream from List<String> to
String stream
Stream<String> flatStream =
namesOriginalList
```

```
        .flatMap(strList ->
strList.stream());

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

Java Stream Terminal Operations

Let's look at some of the java stream terminal operations example.

1. **Stream reduce() example**: We can use reduce() to perform a reduction on the elements of the stream, using an associative accumulation function, and return an Optional. Let's see how we can use it multiply the integers in a stream.

```
Stream<Integer> numbers =
Stream.of(1,2,3,4,5);

Optional<Integer> intOptional =
numbers.reduce((i,j) -> {return i*j;});
if(intOptional.isPresent())
System.out.println("Multiplication =
"+intOptional.get()); //120
```

2. **Stream count() example**: We can use this terminal operation to count the number of items in the stream.

```
Stream<Integer> numbers1 =
Stream.of(1,2,3,4,5);

System.out.println("Number of elements in
stream="+numbers1.count()); //5
```

3. **Stream forEach() example**: This can be used for iterating over the stream. We can use this in place of iterator. Let's see how to use it for printing all the elements of the stream.

```
Stream<Integer> numbers2 =
Stream.of(1,2,3,4,5);
```

```
numbers2.forEach(i ->
System.out.print(i+",")); //1,2,3,4,5,
```

4. **Stream match() examples**: Let's see some of the examples for matching methods in Stream API.

```
Stream<Integer> numbers3 =
Stream.of(1,2,3,4,5);
System.out.println("Stream contains 4?
"+numbers3.anyMatch(i -> i==4));
//Stream contains 4? true

Stream<Integer> numbers4 =
Stream.of(1,2,3,4,5);
System.out.println("Stream contains all
elements less than 10?
"+numbers4.allMatch(i -> i<10));
//Stream contains all elements less than
10? true

Stream<Integer> numbers5 =
Stream.of(1,2,3,4,5);
System.out.println("Stream doesn't contain
10? "+numbers5.noneMatch(i -> i==10));
//Stream doesn't contain 10? true
```

5. **Stream findFirst() example**: This is a short circuiting terminal operation, let's see how we can use it to find the first string from a stream starting with D.

```
Stream<String> names4 =
Stream.of("Pankaj","Amit","David", "Lisa");
Optional<String> firstNameWithD =
names4.filter(i ->
i.startsWith("D")).findFirst();
if(firstNameWithD.isPresent()){
        System.out.println("First Name
starting with D="+firstNameWithD.get());
//David
}
```

Java 8 Stream API Limitations

Java 8 Stream API brings a lot of new stuffs to work with list and arrays, but it has some limitations too.

1. **Stateless lambda expressions**: If you are using parallel stream and lambda expressions are stateful, it can result in random responses. Let's see it with a simple program.
   StatefulParallelStream.java

```
package com.journaldev.java8.stream;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class StatefulParallelStream {

        public static void main(String[]
args) {

                List<Integer> ss =
Arrays.asList(1,2,3,4,5,6,7,8,9,10,11,12,

                List<Integer> result =
new ArrayList<Integer>();

                Stream<Integer> stream =
ss.parallelStream();
```

   If we run above program, you will get different results because it depends on the way stream is getting iterated and we don't have any order defined for parallel processing. If we use sequential stream, then this problem will not arise.

2. Once a Stream is consumed, it can't be used later on. As you can see in above examples that every time I am creating a stream.
3. There are a lot of methods in Stream API and the most confusing part is the overloaded methods. It

makes the learning curve time taking.

That's all for Java 8 Stream example tutorial. I am looking forward to use this feature and make the code readable with better performance through parallel processing.

Reference: Java Stream API Doc

« PREVIOUS
Java 8 Functional Interfaces

NEXT »
Java 8 Date – LocalDate, LocalDateTime, Instant

About Pankaj

I love Open Source technologies and writing about my experience about them is my passion. You can connect with me directly on **Facebook**, **Twitter**, and **YouTube**.

FILED UNDER: JAVA