**Java Stream Examples**

#### Creating Java Streams

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);**

1. 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>

1. 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();**

1. 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);

1. 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.

**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}**

**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

**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]

**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()).collect(Collectors.toList());

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]

**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

**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

**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

**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,

**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

**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

}

**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(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);

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

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

stream.map(s -> {

synchronized (result) {

if (result.size() < 10) {

result.add(s);

}

}

return s;

}).forEach( e -> {});

System.out.println(result);

}

}

1. 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.