I’m considering to write a few blogs in my journey deep-diving in the Java ecosystem. I’ve recently exposed myself to a lot of newer opportunities and learning across domains and technologies -AWS, Systems Design, DSA, Java, Distributed Computing, Competitive programming. And I said to myself, why not share it with others. So here’s the first, of many more to come, blogs. Criticism is warmly welcomed!

When I first heard about streams, it really got me frustrated with all the method chaining and use of Lambda expressions. It wasn’t until something something happened, that changed my view about streams *Once and for All*

* I failed to answer some very basic things about streams in quite a few interviews
* One of those gave me a review — Length of knowledge is good, depth is missing.

That’s it! mind-cracking! That’s when I said — enough is enough! Let’s get this done. **Once and for All!**

So here’s my take on giving it my all. I think you’re going to be here for a while, so grab your snacks, sit tight, and let’s get an in-depth introduction to stream. *All at once, all at one place*.

This will be a 2 part series. Here’s the part 1.

***Boilerplate code***

*We’ll use the following code later to perform some of our streams*operations

class Employee {  
 private int ID;  
 private String name;  
 private double salary;  
 Employee (int ID, String name, double salary) {  
 this.ID = ID;  
 this.name = name;  
 this.salary = salary;  
 }  
   
 public double getSalary() {  
 return salary;  
 }  
 public void incrementSalary(double percentage) {  
 this.salary += (this.salary \* percentage) / 100;  
 }  
}  
  
class Main {  
 public static void main(String[] args) {  
 Employee[] employees = {  
 new Employee(1, "Mickey Mouse", 100000.0),  
 new Employee(2, "Donald Duck", 200000.0),  
 new Employee(3, "Goofy Goo", 300000.0)  
 };  
 Employee brotherBear = new Employee(4, "Brother Bear", 5000.0);  
 Employee mufasa = new Employee(5, "Mufasa - The Lion King", 500000.0);  
 }  
}

**Java Streams Creation**

**Following are some snippets to help create streams.**

**Stream of Employees from an existing array**

Stream<Employee> employeeStream = Stream.of(employees);

**Stream of employees from an existing list**

List<Employee> empList = Arrays.asList(employees);  
Stream<Employee> employeeStream = empList.stream();

**Stream from a list of individual Employee objects.**

Stream<Employee> employeeStream = Stream.of(brotherBear, mufasa);

**Using Stream.Builder to build a stream of Employee objects**

Stream.Builder<Employee> builder = Stream.builder();  
builder.accept(brotherBear);  
builder.accept(mufasa);  
builder.accept(new Employee(6, "SherKhan", 450000.0));  
Stream<Employee> employeeStream = builder.build();

**Java Streams Operations**

There are two types of stream operations — **intermediate** & **terminal**.

**Terminal**

An operation that marks the stream as consumed. And ends the stream operation

***Intermediate***

An operation that returns a new stream after performing the supplied operation on input stream

**Following are some snippets of available operations on Java Streams and their usages.**

**forEach**

*Operation Type: Terminal*

Loop over stream element and call the supplied function over each. forEach() is a terminal operation, i.e. once forEach() is called, the stream is considered to be consumed.

List<Employee> empList = Arrays.asList(employees);  
empList.stream().forEach(e -> e.incrementSalary(10.0));

***NOTE:*** Stream can be consumed only once. If attempted to consume after being consumed, the following exception is thrown

**IllegalStateException: stream has already been operated upon or closed.**

**map**

*Operation Type: Intermediate*

map() applies the supplied function to each element of the current stream and returns a new stream. The resultant stream can be of the same or different type.

List<Employee> empList = Arrays.asList(employees);  
List<Double> salaries = empList.stream().  
 .map(Employee::getSalary).collect(Collectors.toList());

// stream of integers returned as stream of squared versions of themselves.  
Stream<Integer> squared = Stream.of(1, 2, 3, 4, 5).map(x -> x \* x);

**collect**

*Operation Type: Terminal*

Once all the stream processing is done, we can use the collect() with a suitable collector option. We saw in the previous example how we used collect with toList collector to collect stream of double salary into the List<Double> salaries. The strategy for collection is provided via the Collector interface.

**filter**

*Operation Type: Intermediate*

filter() as the name suggests, helps filter a given stream. It does so, by passing the elements of input stream via a Predicate, that evaluates to either true / false based on a condition on the element. This also means, that flter() generates a stream of the same type of elements.

List<Employee> employeesWithSalariesUnder200K = empList.stream()  
 .filter(e -> e.getSalary() < 200000)  
 .collect(Collectors.toList());

**findFirst**

*Operation Type: Terminal*

findFirst() returns an Optional for the first entry in the stream; the Optional can, of course, be empty.

Optional<Employee> employee = empList.stream().findFirst();

**toArray**

*Operation Type: Terminal*

collect() is used to collect the stream into a Collection. If we need to get an array out of the stream, we can simply use toArray().

Employee[] employees = empList.stream().toArray(Employee[]::new);

**flatMap**

*Operation Type: Intermediate*

flatMap() helps us flatten a complex stream such as below. We have a List<List<Stream>>>; flatMap would help us flatten it to a Stream<String>which can be collected into a List<String>. map() takes one input, generates one output. flatMap() takes one input and generates zero or more outputs.

List<List<String>> couples = Arrays.asList(  
 Arrays.asList("Donald Duck", "Daisy Duck"),  
 Arrays.asList("Mickey Mouse", "Minnie Mouse"),  
 Arrays.asList("Nobita", "Shizuka"));  
  
List<String> employees = couples.stream()  
 .flatMap(Collection::stream)  
 .collect(Collectors.toList());

**peek**

*Operation Type: Intermediate*

peek() is an intermediate operation that helps perform a function over each element of a stream.

**Method Types and Pipelines**

In the last section we saw different stream operations and available methods to use them. Some methods are terminal(*returns non-stream types*) which marks the stream as consumed some are intermediate(*returns streams*), output of which can be further processed.

A stream pipeline consists of a stream source, followed by zero or more intermediate operations, and a terminal operation.

**Here’s a short example** Here we are using a stream() that is passed to filter(), that intern returns a filtered-stream, which is forwarded to collect(). Here, stream(), filter(), are intermediate operations, and collect() is a terminal operation.

List<Employee> employeesWithSalariesUnder200K = empList.stream()  
 .filter(e -> e.getSalary() < 200000)  
 .collect(Collectors.toList());

Some operations are called **short-circuiting operations**. **Short-circuiting** operations allow computations on infinite streams to complete in finite time. Here, Stream<T> iterate(final T seed, final UnaryOperator<T> f) will generate an infinite stream starting with 0, with next elements incremented by 1 more than current. the skip() short-circuit operation is used to skip the first n elements in the input stream. if stream().count() < n, then an empty stream is generated. here we skip the first 5 elements. That gives us a stream of numbers starting from 5, incremented by 1. limit() helps to limit the stream up to 10. i.e. a stream from 5 to 14.

Stream.iterate(0, n -> n + 1)  
 .skip(5)  
 .limit(10)  
 .forEach(System.out::println);

**Lazy Evaluation**

One thing that significantly improves Java streams is the ability to evaluate  
operations lazily. It is not until the terminal operation is triggered, the source data is computed. Consumption happens only as needed. All intermediate operations are lazy, i.e. not executed until a result of a processing is actually needed.

Employee employee = Stream.of(employees)  
 .filter(e -> e != null)  
 .filter(e -> e.getSalary() > 100000)  
 .findFirst()  
 .orElse(null);

Here, all the operations are performed on first employee from the array. Since the first employee has salary less than 200000, it proceeds to run the operations on the second object. this object satisfies the first and second filter() calls. Hence, the third operation findFirst() is evaluated, and Employee object is returned. No operations are performed on the third object.

Lazy Evaluation avoids examination of the data that’s not necessary. This behaviour becomes even more important when the input stream is infinite and not just very large.

**Comparison Based Stream Operations**

**sorted**

sorted() sorts the input stream based on the comparator passed inside it.

empList.stream()  
 .sorted((e1, e2) -> e1.getSalary() > e2.getSalary())  
 .collect(Collectors.toList());

Here we sorted the empList by the employee salaries.

***NOTE:*** short-circuit operations will not be applied for sorted(). Even if we had used findFirst(), it would have first sorted the array. Because without that, one would not know which would be the first sorted element.

**min and max**

As the name suggests, these are used to get the maximum or minimum element from a stream based on a comparator. The return is wrapped in an Optional since either of these could be absent in the stream.

Employee highestSalariedEmployee = empList.stream()  
 .max(Comparator.comparing(Employee::getSalary))  
 .orElseThrow(NoSuchElementException::new);  
  
Employee lowestSalariedEmployee = empList.stream()  
 .min(Comparator.comparing(Employee::getSalary))  
 .orElseThrow(NoSuchElementException::new);

Removes duplicates from the input stream by using the equals() method of the stream elements to conclude whether two elements are same or not.

Stream.of(1, 3, 5, 2, 2, 8, 3).distinct().collect(Collectors.toList());

**allMatch, anyMatch, and noneMatch**

All of these operations take a Predicate and return a boolean. Short-circuiting is applied and processing is stopped as soon as the answer is determined

boolean allEven = intList.stream().allMatch(i -> i % 2 == 0);  
boolean oneEven = intList.stream().anyMatch(i -> i % 2 == 0);  
boolean noneMultipleOfThree = intList.stream().noneMatch(i -> i % 3 == 0);

allMatch()checks if the Predicate is true for all the elements in the stream. Here, it returnsfalseas soon as it encounters 5, which is not divisible by 2.

anyMatch()checks if the Predicate is true for any one element in the stream. Here, again short-circuiting is applied and trueis returned immediately after the first element.

noneMatch()checks if there are no elements matching the Predicate. Here, it simply returns falseas soon as it encounters 6, which is divisible by 3.

**Java Stream Specialisations**

So far we have dealt with object streams. But, there exist streams to work with the primitive data types — IntStream, DoubleStream, LongStream. These are super helpful while we are dealing with a numerical operations. These do not extend Stream interface, but rather extend BaseStream interface, which is being extended by Stream. So many of the stream operations available via Stream interface are not present in these streams.

**Creation**

The most common way of creating an IntStream is to call mapToInt() on an existing stream

Double latestEmpId = empList.stream()  
 .mapToDouble(Employee::getSalary)  
 .max()  
 .orElseThrow(NoSuchElementException::new);

Here, we start with a Stream<Employee> and get an IntStream by supplying the Employee::getId to *mapToDouble*. Finally, we call *max()* which returns the highest integer.

We can also use IntStream.of() for creating the IntStream:

IntStream stream = IntStream.of(1, 2, 3, 4, 5);.

or IntStream.range():

IntStream.range(1, 10);

which creates an IntStream from 1 to 9.

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

Above statement returns us a Stream<Integer> and not an IntStream. One has to be heedful as to which operation returns which type of stream. Similarly, Stream.map(Employee::id) returns a Stream<Integer>, but if we do this - Stream.mapToInt(Employee:id), that yields us an IntStream.

**Specialised Operations**

Specialised streams provide some additional operations that make dealing with numbers quite effortless.

For example — average(), range(), sum()

Double averageSalary = empList.stream()  
 .mapToDouble(Employee::getSalary)  
 .average()  
 .orElseThrow(NoSuchElementException::new);

That was a lot! At least for me! I hope it helped all of you to understand The Java 8 Streams API in some form or the other. We learnt a lot here and that has prepared us well for what’s next in Part 2. So stay tuned, and see you on the 2nd Part!