5-stream-api-essentials

Contents

- Streams
  - Optional
  - Operations
    - Searching
    - Calculation
    - Sorting
    - Collecting
    - Grouping
    - Mapping
- Summary

Streams

Possibly the most important addition to the Java library in the Java 8 release, the stream API. The stream API is part of the `java.util.stream` package.

Optional

The optional class type was introduced to allow users of the language to express a value that might be empty or in the case of java, null, the optional class provides certain methods to validate that the value which it holds is not nil, before it can be used, it also provides convenient interface to avoid the dreaded `if not nil do-this else do-that` blocks which can often make code less readable, instead the optional interface often provides way to express these expressions in one line of code, chaining certain operations.

One can certainly think of the Optional class / interface as a Stream with one element only, since some of the methods do match the ones in the Stream interface, Optional values can be extracted, mapped and filtered

To create a Optional object there are several options by default provided by default from the language

- Creating from no value - `Optional.empty()` - this is usually required when for example a method that returns an optional value does not find any value to return, so instead of returning plain `null` the empty optional is preferred
- Creating from concrete value - `Optional.of(<value>)` - wrapping a value in an optional, can be done by using the Optional.of(variable), note that this method is nil checking, if the variable is nil, calling this method will throw an exception.
- Creating from nullable value - `Optional.ofNullable(<value>)` - allows one to create both an empty and non-empty optional with a single statement, meaning that if the value is nil, the optional that will be created is an empty one, and exception will NOT be thrown

The optional class type similarly to the streams has also primitive versions - `OptionalDouble`, `OptionalLong`, `OptionalInt`, the interface of these optional types is pretty much the same as the regular Optional type, however with some very slight differences, they also do not inherit from the Optional type, just as like the primitive streams does not inherit from Stream, only from `BaseStream`

```
DoubleStream temperatures = DoubleStream.of(24.5, 23.6, 27.9, 21.1, 23.5,
   25.5, 28.3);
OptionalDouble max = temperatures.max();
max.ifPresent(System.out::println);
```

```
The primitive versions of the Streams use the primitive versions of the Optional types,
which should be obvious and expected, the primitive streams do not interact with the base
 Optional non-primitive type, as it is only a wrapper around object / class type and can
not be used on primitive types
```

Note that of course there is a way to convert between the primitive version of the optional and the object type by simply for example using the auto-boxed numeric types, which are classes

Operations

There are certain important operations which are exposed by the stream API, to allow one to search, filter and convert data in the stream. They are used to build up the stream transformation pipeline

Searching

Methods ending with the word "Match and methods starting with the word"find" in the Stream interface are useful for searching data from the stream. These methods return a `boolean` value. For searching operations `findFirst()` and `findAny()` matching elements may not be present in the Stream, so they return Optional

Method	Description
boolean anyMatch(Predicate<? super T> check)	Returns true if there is any elements in the stream that matches the given predicate. Returns false if the stream is empty or if there are no matching elements.
boolean allMatch(Predicate<? super T> check)	Returns true only if all elements in the stream matches the given predicate. Returns true if the stream is empty without evaluating the predicate!
boolean noneMatch(Predicate<? super T> check)	Returns true only if none of the elements in the stream matches the given predicate. Returns true if the stream is empty without evaluating the predicate!
Optional findFirst()	Returns the first element from the stream; if there is no element present in the stream, it returns an empty Optional object.

Method	Description
Optional findAny()	Returns one of the elements from the stream; if there is no element present in the stream, it returns an empty Optional object.

Unlike the anyMatch()method that returns false when the stream is empty the allMatch()and noneMatch()return true if the stream is empty. Which should be rather obvious, since no elements in the stream will always imply that match allMatch()or noneMatch()return true for any condition for those methods

```
boolean anyMatch = IntStream.of(-56, -57, -55, -52, -48, -51, -49).anyMatch(
   temp -> temp > 0);
System.out.println("anyMatch(temp -> temp > 0): " + anyMatch);

boolean allMatch = IntStream.of(-56, -57, -55, -52, -48, -51, -49).allMatch(
   temp -> temp > 0);
System.out.println("allMatch(temp -> temp > 0): " + allMatch);

boolean noneMatch = IntStream.of(-56, -57, -55, -52, -48, -51, -49).noneMatch
   (temp -> temp > 0);
System.out.println("noneMatch(temp -> temp > 0): " + noneMatch);
```

The example above demonstrates a simple usage of the matching functions, those are useful to avoid filtering first and then checking the size count/size of the final collection, instead, this operation is condensed into a single searching operation instead.

The other companion function to the `boolean` any/all/none match, are the find first and any, the example below demonstrates how one can use a transformation pipeline to first filter on all methods in the Stream API / interface, that end with "Match", sort and then extract the first one that is found, in this case the sorting will be in ascending order, in that case the `allMatch`, since all is sorted before `anyMatch`, and `noneMatch`. Note that the find similarly to the boolean match methods, are also terminating operations

```
Method[] methods = Stream.class.getMethods();
Optional<String> methodName = Arrays.stream(methods)
                    .map(method -> method.getName())
                    .filter(name -> name.endsWith("Match"))
                    .sorted()
                    .findFirst();
System.out.println("Result: " + methodName.orElse("No suitable method found")
   );
```

The reason there are two find methods in the API, is that the method called findAny is tailored for usecases which involve parallel streams, however for any streams, the findFirst is always defined to return the first element in the stream, however the findAny 's behavior is not defined for normal and parallel streams, it may return any element in the stream, especially true for parallel streams.

It is prudent to note that the match and find operations are short-circuiting in nature, meaning that the moment they find the matching element they will terminate, and the iteration over the rest of the entries in the stream is not done.

Calculation

There are certain calculation methods on the Stream and primitive streams API. Methods like min(), max() and average(). These operations are also called implicit `reducers`, since internally they are implemented using the reduce method of the stream, which is made to accumulate a result of some computation and then returns that result. The reduce() method is however much more general and has applications outside the simple operations such as the ones listed above.

```
String[] string = "you never know what you have until you clean your room".
   split(" ");
System.out.println(Arrays.stream(string).min(String::compareTo).get());
```

The min and max methods in the Stream version (non primitive one) require a comparator to be passed in in order for the min/max calculation to be performed, otherwise there is no way to truly find a meaningful result to the min/max expression. For the primitive versions of the streams, that is not necessary, since the integral types have well defined comparison rules.

Method	Description
long count()	Returns the number of elements in the stream; 0 if the stream is empty.
Optional min(Comparator<? super T> comparator)	Returns the minimum value in the stream; an empty Optional value in case the stream is empty.
Optional max(Comparator<? super T> comparator)	Returns the maximum value in the stream; an empty Optional value in case the stream is empty.
int sum()	Returns the sum of elements in the stream; 0 in case the stream is empty.
long count()	Returns the number of elements in the stream; 0 if the stream is empty.
Optional{Primitive} min()	Returns the minimum integer value in the stream; an empty Optional{Primitive} value in case the stream is empty.
Optional{Primitive} max()	Returns the maximum integer value in the stream; an empty Optional{Primitive} value in case the stream is empty.
OptionalDouble average()	Returns the average value of the elements in the stream; an empty OptionalDouble value in case the stream is empty.
{Primitive}SummaryStatistics summaryStatistics()	Returns an {Primitive}SummaryStatistics object that has sum, count, average, min, and max values.

The table above shows the interfaces for both the Stream and primitive Stream versions, note that not all methods from the primitive versions are present in the base Stream API. That is because not all of them make sense on non primitive types such as `sum` or `average` for example

The `summaryStatistics` is one of the more interesting methods, it provides a detailed information about the contents of the stream, things like average, min, max and sum in one shot, basically collates the calls to all methods into one object, which can be used to obtain information about the stream, since after calling the sum or count the stream will be otherwise closed, and unable to call any more termination operations on it

```
String limerick = "There was a young lady named Bright " +
                  "who traveled much faster than light " +
                  "She set out one day " +
                  "in a relative way " +
                  "and came back the previous night ";
IntSummaryStatistics wordStatistics = Pattern.compile(" ")
                  .splitAsStream(limerick)
```

```
                    .mapToInt(word -> word.length())
                    .summaryStatistics();

System.out.printf(" Number of words = %d \n Sum of the length of the words =
   %d \n" +
                    " Minimum word size = %d \n Maximum word size %d \n " +
                    " Average word size = %f \n", wordStatistics.getCount(),
                    wordStatistics.getSum(), wordStatistics.getMin(),
                    wordStatistics.getMax(), wordStatistics.getAverage());
```

Sorting

Sorting is another of the capabilities of the stream, allows the items within the stream to be ordered based on some sort of criteria. The method sorted(Compoarator<? super T> compartor) is used to do that

```
List words = Arrays.asList("follow your heart but take your brain with you".
   split(" "));
Comparator<String> lengthCompare = (str1, str2) -> str1.length() - str2.
   length();
words.stream().distinct().sorted(lengthCompare).forEach(System.out::println);
```

The example splits the string into words, and they are sorted based on the length of the words, in this case in ascending order (since the comparison is doing left.length - right.length).

Now what if one wanted to sort them based on their length and then alphabetically, the `Compartor` method has additional ways of chaining comparison operations.

```
// compare and sort based on the length of the words first, then do an
   alphabetically, the way thenComparing works, is
// if the first comparator considers the two entries to be equal, then the
   second comparator, in this case
// String::compareTo, is used to resolve the comparison operations, these can
    be chained as many times, since thenComparing
// variants return a comparator instance
words.stream()
    .distinct()
    .sorted(lengthCompare.thenComparing(String::compareTo))
    .forEach(System.out::println);
```

Another option is to reverse the order of the sorting, by default the compator functions, by convention should sort in ascending order, comparing the first to second argument, the way they are passed in to the compareTo method by the sorting algorithm internally, however to obtain a reverse order there are two ways - modify the compartor to compare the second to the first argument, which is not always possible, if re-using existing comparison method (i.e. String::compareTo) or to use reversed(), which basically wraps the previous comparator instance, previously chained, internally, calling the compare to with swapped arguments instead

```
// the compartor interface also provides means of reversing the natural order
   , by calling reversed, what it does is
// return an instance of comparator which wraps the previous one chained, and
    simply inverts the passed arguments to the
// compare to method, so instead of comparing (a, b) it executes the
   comparator with (b, a), which reversed the order in
// essence
```

```
words.stream()
    .distinct()
    .sorted(lengthCompare.thenComparing(String::compareTo).reversed())
    .forEach(System.out::println);
```

The natural order term which is often used in the Comparable and Comparator interface is the default order which is implied when comparing, as mentioned above, that is comparing the elements in the order they are passed to the compare method, this is done to avoid confusion and introduce a consistency across the API.

Collecting

The stream API has several methods which allows the transformation to be captured in a re-usable data structure, after the transformation pipeline has finished and all the operations chained on it have finished.

There is one generic method - collect, which accepts a Collector interface on its input, allowing the elements of the stream to be collected in an arbitrary way, there are also helper functions which provide a pre-defined collectors for list, set and map.

```
// this simple example splits words : into a stream and then collects them
   into a list, note the usage of the
// pre-defined toList method in the Collectors interface, which provides a
   dirty and fast way to simply provide a
// container, by default ArrayList, where to collect the entries of the
   stream
String frenchCounting = "un:deux:trois:quatre";
List gmailList = Pattern.compile(":")
                .splitAsStream(frenchCounting)
                .collect(Collectors.toList());
gmailList.forEach(System.out::println);

// similar example as the one above, using the pre-defined method toSet from
   the Collectors, in this case by default the
// set that is created is HashSet, where each element is hashed using the
   hashCode of the object and put into the set,
// remember that sets do not allow duplicate values
String []roseQuote = "a rose is a rose is a rose".split(" ");
Set words = Arrays.stream(roseQuote).collect(Collectors.toSet());
words.forEach(System.out::println);

// in this example the more complex mapping is used, the Collectors also
   provide a toMap method, which however accepts
// at least two arguments (there are overloads which make it take up to four)
   . The first one is how to map the key of the
// HashMap from the stream entry, the second is the value for this key
   mapping, in the example below, the key entry itself,
// the value is the string entry length
Map<String, Integer> nameLength = Stream.of("Arnold", "Alois", "
   Schwarzenegger")
.collect(Collectors.toMap(name -> name, name -> name.length()));
nameLength.forEach((name, len) -> System.out.printf("%s - %d \n", name, len))
   ;
```

```
// this example shows how to provide a custom lambda reference for the
   creation of the collection, since all collections
// have a way to add elements, in the most simplest collection, one needs
   provide only a way to create the desired
// collection, the rest is handled internally, adding the elements of the
   Stream to that collection instance
String []roseQuote = "a rose is a rose is a rose".split(" ");
Set words = Arrays.stream(roseQuote).collect(Collectors.toCollection(TreeSet
   ::new));
words.forEach(System.out::println);
```

Grouping

There is another sub-set of collection operations which is the grouping, the grouping operation provides a way to create a collation of the map entries based on some criteria, i.e. to group them based on certain rule, in this case for the grouping result a Map is always used, where the key of the map is the unique criteria, and the value of the Map, is a List, where the list represents all the values which match the criteria

```
// this simple example simply groups all words in the split string based on
   the length of the word, the final map will
// have keys which describe the length, and a list of words which match these
    lengths as the value
String []string= "you never know what you have until you clean your room".
   split(" ");
Stream<String> distinctWords = Arrays.stream(string).distinct();
Map<Integer, List<String>> wordGroups = distinctWords.collect(Collectors.
   groupingBy(String::length));

// the example below shows a special case of grouping which partition, the
   partition is pretty much like grouping but
// instead of resolving to one concrete value for each group, the values are
   loosely grouped, in the example, all words
// with length less than 4 are grouped in the map key {false} and all words
   with length greater than 4 are grouped under the
// map key of {true}. Note the map is not keyed on Integer, but rather on the
    result of the grouping which in this case
// is a boolean one, either the length is greater than 4 or not
String []string= "you never know what you have until you clean your room".
   split(" ");
Stream<String> distinctWords = Arrays.stream(string).distinct();
Map<Boolean, List<String>> wordBlocks =
distinctWords.collect(Collectors.partitioningBy(str -> str.length() > 4));
```

The groupingBy method always accepts at least one argument which is a of the type Function from the functional interface, however the partitioningBy always accepts an argument of type Predicate again from the functional interface package java.util.function. This is due to the nature of both methods - one groups the entries into neat buckets, while the other splits them into at most two groups, ones that match the predicate and the rest that do not

Mapping

Mapping is the process of converting the starting type of the stream entry, to another one, these operations can be chained, meaning that unlimited calls to .map can be made transforming each preceding type into a new one. Mapping is very handy when the starting type of the stream entries is not suitable to perform some sort of operations and one wishes to refine the type, so the calculation can be performed more efficiently or performed at all in the first place

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);
integers.stream()
        .map(i -> i * i)
        .forEach(System.out::println);
```

The example above simply finds and converts the map entries into their squares, the mapped type is not changed however, it still remains an Integer, only the value is. However this is the most simplest demonstration of what mapping means, it allows one to mutate each entry of the stream, the value and even the its type, however note that each map must return the same type. It is not possible to return heterogeneous types (unless they are all children of the same super type)

Method	Description
DoubleStream mapToDouble(ToDoubleFunction<? super T>)	Maps the current stream into a primitive double stream
LongStream mapToLong(ToLongFunction<? super T>)	Maps the current stream into a primitive long stream
IntStream mapToInt(ToIntFunction<? super T>)	Maps the current stream into a primitive integer stream
Stream map(Function<? super T, ? extends R>)	Maps the current stream into one containing elements of type T

Flat mapping is a special case of the regular mapping, it is usually used when the source collection is a nested 2 or N dimensional one, in the most simplest example a 2 dimensional array, flat mapping means that this structure will be reduced into a single dimension or level after the mapping. Flat map always accepts a Function which as an input argument takes the current type of entry in the stream and returns a Stream (or the primitive versions of the streams).

Method	Description
DoubleStream flatMapToDouble(Function<? super T, ? extends DoubleStream>)	
LongStream flatMapToLong(Function<? super T, ? extends LongStream>)	
IntStream flatMapToInt(Function<? super T, ? extends IntStream>)	
Stream flatMap(Function<? super T, ? extends Stream<? extends R»)	

```
// this example the stream starts as a stream of words, however the desired
   effect is to find the unique characters in
// the sentence, therefore each word has to be split into characters, then
   only the unique ones extracted, however each
// word is an array of characters, therefore this example will not work,
   since the map method here would map the 'word'
// entry to a String[], why an array, calling split on each word, will
   produce an array of String[].
```

```
String []string= "you never know what you have until you clean your room".
   split(" ");
Arrays.stream(string)
            .map(word -> word.split(""))
            .distinct()
            .forEach(System.out::print);

// this is the fixed version of the example above, using flatMap, flatMap
   will make sure that mapping a nested streams
// is flattened out into its composite elements instead.
String []string= "you never know what you have until you clean your room".
   split(" ");
Arrays.stream(string)
            .flatMap(word -> Arrays.stream(word.split("")))
            .distinct()
            .forEach(System.out::print);
```

Summary

Optional

- When there are no entries in a stream and operations such as max are called, then instead of returning null or throwing an exception, the (better) approach taken in Java 8 is to return Optional values.
- Primitive type versions of Optional for int, long, and double are `OptionalInteger`, `OptionalLong`, and `OptionalDouble` respectively.

Streams

- The Stream interface has data and calculation methods `count()`, `min()` and `max()`; you need to pass a `Comparator` object as the parameter when invoking these `min()` and `max()` methods.
- The primitive type versions of Stream interface have the following data and calculation methods: `count()`, `sum()`, `average()`, `min()`, and `max()`.
- The `summaryStatistics()` method in `IntStream`, `LongStream`, and `DoubleStream` have methods for calculating count, sum, average, minimum, and maximum values of elements in the stream.
- The `peek()` method is useful for debugging: it helps us understand how the elements are transformed in the pipeline.
- You can transform (or just extract) elements in a stream using `map()` method Search for data by using search methods of the Stream classes including `findFirst`, `findAny`, `anyMatch`, `allMatch`, `noneMatch`
- You can match for a given predicate in a stream using the `allMatch()`, `noneMatch()`, and `anyMatch()` methods. Unlike the `anyMatch()` method that returns false when the stream is empty, the `allMatch()` and `noneMatch()` methods return true if the stream is empty.
- You can look for elements in a stream using the `findFirst()` and `findAny()` methods. The `findAny()` method is faster to use than the `findFirst()` method in case of parallel streams.
- The "match" and "find" methods "short-circuit": the evaluation stops once the result is found and the rest of the stream is not evaluated.

Comparator & Comparable

- One way to sort a collection is to get a stream from the collection and call `sorted()` method on that stream. The `sorted()` method sorts the elements in the stream in natural order (it requires that the stream elements implements the Comparable interface).
- When you want to sort elements in the stream other than the natural order, you can pass a `Comparator` object to the `sorted()` method.
- The `Comparator` interface has been enhanced with many useful static or default methods in Java 8 such as `thenComparing()` and `reversed()` methods.

Collectors & Collections

- The `collect()` method of the Collectors class has methods that support the task of collecting elements to a collection.
- The Collectors class provides methods such as `toList()`, `toSet()`, `toMap()`, and `toCollection()` to create a collection from a stream.
- You can group the elements in a stream using the Collectors.`groupingBy()` method and pass the criteria for grouping (given as a Function) as the argument.
- You can separate the elements in a stream based on a condition (given as a Predicate) using the `partition()` method in the Collectors class. . Use `flatMap()` method of the Stream API
- The `flatMap()` method in Stream flattens the streams that result from mapping each element into one flat stream.