Adding Intermediate Operations on a Stream

Mapping a Stream to Another Stream

Mapping a stream consists of transforming its elements using a function. This transformation may change the types of the elements processed by that stream, but you can also transform them without changing their type.

You can map a stream to another stream with the `map()` method, which takes this `Function` as an argument. Mapping a stream means that all the elements processed by that stream will be transformed using that function.

The pattern of code is the following:

```
List<String> strings = List.of("one", "two", "three", "four");
Function<String, Integer> toLength = String::length;
Stream<Integer> ints = strings.stream()
                              .map(toLength);
```

You can copy this code, and paste it in your IDE to run it. You will not see anything and you may be wondering why.

The answer is in fact simple: there is no terminal operation defined on that stream. Your reflex should be to notice that and realize that this code does not do anything. It does not process any data. To answer the question: "What is this code doing?", there is only one valid answer: "Nothing".

Let us add a very useful terminal operation, which puts the processed elements in a list: `collect(Collectors.toList())`. If you are not sure about what this code really does, do not worry; we will cover that later in this tutorial. The code becomes the following.

```
List<String> strings = List.of("one", "two", "three", "four");
List<Integer> lengths = strings.stream()
                               .map(String::length)
```

```
                                .collect(Collectors.toList());
System.out.println("lengths = " + lengths);
```

Running this code prints the following:

```
lengths = [3, 3, 5, 4]
```

You can see that this pattern creates a `Stream<Integer>`, returned by the `map(String::length)`. You can also make it a specialized `IntStream` by calling `mapToInt()` instead of the regular `map()` call. This `mapToInt()` method takes a `ToIntFuction<T>` as an argument. Changing `.map(String::length)` to `.mapToInt(String::length)` in the previous example does not create a compiler error. The method reference `String::length` can be of both types: `Function<String, Integer>` and `ToIntFunction<String>`.

There is no `collect()` method that takes a `Collector` as an argument on specialized streams. So if you use `mapToInt()`, you cannot collect the result in a list anymore, at least not with this pattern. Let us get some statistics on that stream instead. This `summaryStatistics()` method is very handy and is only available on these specialized streams of primitive types.

```
List<String> strings = List.of("one", "two", "three", "four");
IntSummaryStatistics stats = strings.stream()
                                    .mapToInt(String::length)
                                    .summaryStatistics();
System.out.println("stats = " + stats);
```

The result is the following:

```
stats = IntSummaryStatistics{count=4, sum=15, min=3, average=3,750000, max=5}
```

There are three methods to go from `Stream` to a stream of primitive type: `mapToInt()`, `mapToLong()` and `mapToDouble()`.

Filtering a Stream

Filtering is about discarding some elements processed by a stream with a predicate. This method is available on streams of objects and stream of primitive types.

Suppose you need to count the strings of characters of length 3. You can write this code to do that:

```
List<String> strings = List.of("one", "two", "three", "four");
long count = strings.stream()
```

```
                    .map(String::length)
                    .filter(length -> length == 3)
                    .count();
System.out.println("count = " + count);
```

Running this code produces the following:

```
count = 2
```

Notice that you just used another terminal operation of the Stream API, `count()`, which just counts the number of processed elements. This method returns a `long`, so you can count a lot of elements with it. More than you can put in an `ArrayList`.

Flatmapping a Stream to Handle 1:p Relations

Let us see the `flatMap` operation in an example. Suppose you have two entities: `State` and `City`. A `state` instance holds several `city` instances, stored in a list.

Here is the code of the `City` class.

```
public class City {

    private String name;
    private int population;

    // constructors, getters
    // toString, equals and hashCode
}
```

Here is the code of the `State` class, with the relation to the `City` class.

```
public class State {

    private String name;
    private List<City> cities;

    // constructors, getters
    // toString, equals and hashCode
}
```

Suppose your code is processing a list of states, and at some point you need to count the population of all the cities.

You may write the following code:

```
List<State> states = ...;

int totalPopulation = 0;
for (State state: states) {
    for (City city: state.getCities()) {
        totalPopulation += city.getPopulation();
    }
}

System.out.println("Total population = " + totalPopulation);
```

The inner loop of this code is a form of map-reduce that you can write with the following stream:

```
totalPopulation += state.getCities().stream().mapToInt(City::getPopulation).sum();
```

The connection between the loop on the states and this stream does not fit well in the map/reduce pattern, and putting a stream in a loop is not a very nice pattern of code.

This is precisely the role of the flatmap operator. This operator opens one-to-many relations between objects and create streams on these relations. The `flatMap()` method takes a special function as an argument that returns a `Stream` object. The relationship between a given class and another class is defined by this function.

In the case of our example, this function is simple because there is a `List<City>` in the `State` class. So you can write it in the following way.

```
Function<State, Stream<City>> stateToCity = state -> state.getCities().stream();
```

This list is not mandatory. Suppose you have a `Continent` class that holds a `Map<String, Country>`, where the keys are the country codes of the countries (CAN for Canada, MEX for Mexico, FRA for France, and so on). Suppose that the `Continent` class has a method `getCountries()` that returns this map.

In that case, this function could be written in this way.

```
Function<Continent, Stream<Country>> continentToCountry =
    continent -> continent.getCountries().values().stream();
```

The `flatMap()` method processed a stream in two streps.

- The first step consists of the mapping of all the elements of the stream with this function. From a `Stream<State>` it creates a `Stream<Stream<City>>`, because every state is mapped to a stream of cities.
- The second step consists of flattening the stream of streams that is produced. Instead of having a stream of streams of cities (one stream for each state), you end up with a single

stream, with all the cities of all the states in it.

So the code written with a nested for loop pattern can become the following, thanks to the flatmap operator.

```
List<State> states = ...;

int totalPopulation =
        states.stream()
              .flatMap(state -> state.getCities().stream())
              .mapToInt(City::getPopulation)
              .sum();

System.out.println("Total population = " + totalPopulation);
```

Using Flatmap and MapMulti to Validate Elements Transformation

The `flatMap` operation can be used to validate the transformation of the elements of your stream.

Suppose you have a stream of strings of characters, that represent integers. You need to convert them to integers using `Integer.parseInt()`. Unfortunately some of these strings are corrupted: maybe some are empty, null, or have extra blank characters at the end of them. All these will make the parsing fail with a `NumberFormatException`. Of course, you can try to filter this stream to remove the buggy strings with predicates, but the safest way to do that is to use a try-catch pattern.

Trying to use a filter is not the right way to go. The predicate you are going to write will look like the following.

```
Predicate<String> isANumber = s -> {
    try {
        int i = Integer.parseInt(s);
        return true;
    } catch (NumberFormatException e) {
        return false;
    }
};
```

This first flaw is that you need to actually do the conversion to see if it is working or not. Then you will have to do it again in your mapping function, which will be executed next: don't do that! The second flaw is that it is never a good idea to return from a catch block.

What you really need to do is to return an integer when you have a proper integer in this string and nothing if it is a corrupted string. This is a job for a flatmapper. If you can parse an integer, you can return a stream with the result. In the other case, you can return an empty stream.

You can then write the following function.

```
Function<String, Stream<Integer>> flatParser = s -> {
    try {
        return Stream.of(Integer.parseInt(s));
    } catch (NumberFormatException e) {
    }
    return Stream.empty();
};

List<String> strings = List.of("1", " ", "2", "3 ", "", "3");
List<Integer> ints =
    strings.stream()
           .flatMap(flatParser)
           .collect(Collectors.toList());
System.out.println("ints = " + ints);
```

Running this code produces the following result. All the faulty strings have been silently removed.

```
ints = [1, 2, 3]
```

This use of the flatmap code works well, but it has an overhead: there is one stream created for each element of the stream you need to process. Starting with Java SE 16, a method has been added to the Stream API. It has been added exactly for this case: when you create many streams of zero or one object. This method is called `mapMulti()` and takes a `BiConsumer` as an argument.

This `BiConsumer` consumes two arguments:

- The element of the stream that needs to be mapped
- A `Consumer` that this `BiConsumer` needs to call with the result of the mapping

Calling the consumer with an element adds that element to the resulting stream. In case the mapping cannot be done, the biconsumer does not call this consumer, and no element will be added.

Let us rewrite your pattern with this `mapMulti()` method.

```
List<Integer> ints =
        strings.stream()
               .<Integer>mapMulti((string, consumer) -> {
                    try {
                        consumer.accept(Integer.parseInt(string));
                    } catch (NumberFormatException ignored) {
```

```
                    }
                })
                .collect(Collectors.toList());
System.out.println("ints = " + ints);
```

Running this code produces the same result as before. All the faulty strings have been silently removed, but this time, no other stream has been created.

```
ints = [1, 2, 3]
```

To use this method, you need to tell the compiler the type of `Consumer` used to add elements to the resulting stream. This is done with this special syntax where you put this type before calling `mapMulti()`. It is not a syntax that you see very often in Java code. You can use it in a static and nonstatic context.

Removing Duplicates and Sorting a Stream

The Stream API has two methods, `distinct()` and `sorted()`, that will simply detect and remove duplicates and sort the elements of your stream. The `distinct()` method uses the `hashCode()` and `equals()` methods to spot the duplicates. The `sorted()` method has an overload that takes a comparator, which will be used to compare and sort the elements of your stream. If you do not provide a comparator, then the Stream API assumes that the elements of your stream are comparable. If they are not, then a `ClassCastException` is raised.

You may remember from the previous part of this tutorial that a stream is supposed to be an empty object that does not store any data. There are several exceptions to this rule, and these two methods belong to them.

Indeed, to spot duplicates, the `distinct()` method needs to store the elements of your stream. When it processes an element, it first checks if that element has already been seen or not.

The same goes for the `sorted()` method. This method needs to store all of your elements and then to sort them in an internal buffer before sending them to the next step of your processing pipeline.

The `distinct()` method can be used on unbound (infinite) streams, the `sorted()` method cannot.

Limiting and Skipping the Elements of a Stream

The Stream API gives you two ways of selecting the elements of a stream: based on their index, or with a predicate.

The first way is to use the `skip()` and `limit()` methods, both take a `long` as an argument. There is a little trap to avoid when you use these methods. You need to keep in mind that every time an intermediate method is called on stream, a new stream is created. So if you call `limit()` after `skip()`, do not forget to count your elements starting on that new stream.

Suppose you have a stream of all the integers, starting at 1. You need to select the integers between 3 and 8 on a stream of integers. You may be tempted to call `skip(2).limit(8)`, passing the bound computed on the first stream. Unfortunately this is not how streams work. The second call `limit(8)` operates on a stream that starts at 3, so it will select the integers until 11, which is not what you need. The correct code is the following.

```
List<Integer> ints = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);

List<Integer> result =
    ints.stream()
        .skip(2)
        .limit(5)
        .collect(Collectors.toList());

System.out.println("result = " + result);
```

This code prints the following.

```
result = [3, 4, 5, 6, 7]
```

It is important to understand that `skip(2)` has been called on a stream that processes the elements `1, 2, 3, ...`, and produces another stream that processes the elements `3, 4, 5, 6, ...`.

So `limit(3)` selects the first 5 elements of that stream, thus `3, 4, 5, 6, 7`.

Java SE 9 saw the introduction of two more methods in this field. Instead of skipping and limiting the elements based on their index in the stream, it does so based on the value of a predicate.

- `dropWhile(predicate)` drops the elements processed by the stream until the application of the predicate on these elements becomes true. At that point, all the elements processed by that stream are transmitted to the following stream.
- `takeWhile(predicate)` does the contrary: it transmits the elements to the next stream until the application of this predicate on these elements become false.

Note that these methods work like doors. Once `dropWhile()` has opened the door to let the processed elements flow it will not close it. Once `takeWhile()` has closed the door it cannot not

reopen it, no more elements will be sent to the next operation.

Concatenating Streams

The Stream API offers several patterns to concatenate several streams into one. The most obvious way is to use a factory method defined in the `Stream` interface: `concat()`.

This method takes two streams and produces a stream with the elements produced by the first stream, followed by the elements of the second stream.

You may be wondering why this method does not take a vararg to allow for the concatenation of any number of streams.

The reason is that using this method is OK as long as you have two streams to join. If you have more than two, then the JavaDoc API documentation advises you to use another pattern, based on the use of flatmap.

Let us see how this works on an example.

```
List<Integer> list0 = List.of(1, 2, 3);
List<Integer> list1 = List.of(4, 5, 6);
List<Integer> list2 = List.of(7, 8, 9);

// 1st pattern: concat
List<Integer> concat =
    Stream.concat(list0.stream(), list1.stream())
          .collect(Collectors.toList());

// 2nd pattern: flatMap
List<Integer> flatMap =
    Stream.of(list0.stream(), list1.stream(), list2.stream())
          .flatMap(Function.identity())
          .collect(Collectors.toList());

System.out.println("concat  = " + concat);
System.out.println("flatMap = " + flatMap);
```

Running this code produces the following result:

```
concat  = [1, 2, 3, 4, 5, 6]
flatMap = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The reason why it is better to use the `flatMap()` way is that `concat()` creates intermediates streams during the concatenation. When you use `Stream.concat()`, a new stream is created to

concatenate your two streams. If you need to concatenate three streams, you will end up creating a first stream to handle the first concatenation, and a second one for the second concatenation. So each concatenation requires a stream that will be thrown away very quickly.

With the flatmap pattern, you just create a single stream to hold all your streams and do the flatmap. The overhead is much lower.

You may be wondering why these two patterns have been added. It looks like `concat()` is not really useful. In fact there is a subtle difference between the stream produced by the concat and the flatmap patterns.

If the size of the source of the two streams you are concatenating is known, then the size of the resulting stream is known too. In fact, it is simply the sum of the two concatenated streams.

Using flatmap on a stream may create an unknown number of elements to be processed in the resulting stream. The Stream API loses track of the number of elements that will be processed in the resulting stream.

In other words: concat produces a `SIZED` stream, whereas flatmap does not. This `SIZED` property is a property a stream may have, which will be covered later in this tutorial.

Debugging Streams

It may sometimes be convenient to examine the elements processed by a stream at run time. The Stream API has a method for that: the `peek()` method. This method is meant to be used to debug your data processing pipeline. You should not use this method in your production code.

> *You should absolutely refrain from using this method to perform some side effects in your application.*

This method takes a consumer as an argument that will be invoked by the API on each element of the stream. Let us see this method in action.

```
List<String> strings = List.of("one", "two", "three", "four");
List<String> result =
        strings.stream()
                .peek(s -> System.out.println("Starting with = " + s))
                .filter(s -> s.startsWith("t"))
                .peek(s -> System.out.println("Filtered = " + s))
                .map(String::toUpperCase)
                .peek(s -> System.out.println("Mapped = " + s))
                .collect(Collectors.toList());
System.out.println("result = " + result);
```

If you run this code, you will see the following on your console.

```
Starting with = one
Starting with = two
Filtered = two
Mapped = TWO
Starting with = three
Filtered = three
Mapped = THREE
Starting with = four
result = [TWO, THREE]
```

Let us analyze this output.

1. The first element to be processed is *one*. You can see that it was filtered out.
2. The second is *two*. This element passed the filter and then is mapped to upper case. It is then added to the result list.
3. The third one is *three*, that also passes the filter and is also mapped to upper case before being added to the result list.
4. The fourth and last one is *four* that is rejected by the filtering step.

There is one point that you saw earlier in this tutorial and that appears clearly now: a stream does process all the elements it has to process one by one, from the beginning to the end of the stream. This was mentioned before, and now you can see it in action.

You can see that this `peek(System.out::println)` pattern is very useful to follow the elements processed by your stream one by one, without having to debug your code. Debugging a stream is hard because you need to be careful where you put your breakpoints. Most of the time, putting breakpoints on a stream processing will send you to the implementation of the `Stream` interface. This is not what you need. Most of the time you need to put these breakpoints in the code of your lambda expressions.

Last update: *September 14, 2021*

Processing Data in Memory Using the Stream API ← **Adding Intermediate Operations on a Stream** → Creating Streams

Home > Tutorials > The Stream API > Adding Intermediate Operations on a Stream