Adding Intermediate Operations on a Stream ← Creating Streams → Reducing a Stream

Creating Streams

Creating a Stream

You have already created many streams in this tutorial, all by calling the `stream()` method of the `Collection` interface. Tl method is very convenient: creating streams in this way requires only two simple lines of code, and you can use this strea experiment with almost any feature of the Stream API.

As you are going to see, there are many other ways to create streams on many objects. Knowing these ways enables you leverage the Stream API in many places in your application and to write more readable and maintainable code.

Let us quickly browse through the ones that you are going to see in this tutorial before deep diving into each of them.

The first set of patterns uses factory methods from the `Stream` interface. With them, you can create streams from the foll elements:

- a vararg argument;
- a supplier;
- a unary operator, that generates the next element from the previous one;
- a builder.

You can even create an empty stream, which may be convenient in some circumstances.

You already saw that you can create a stream on a collection. If what you have is only an iterator and not a fully fledged collection, then there is a pattern for you: you can create a stream on an iterator. If you have an array, then there is also a p to create a stream on the elements of an array.

It does not stop here. Many patterns have also been added to well-known objects from the JDK. You can then create strea from the following elements:

- the characters of a string;
- the lines of a text file;
- the elements created by splitting a string of characters with a regular expressions;
- a random variable, that can create a stream of random numbers.

You can also create a stream with a builder pattern.

Creating a Stream from a Collection or an Iterator

You already know that there is a `stream()` method available in the `Collection` interface. This is probably the most classi of creating streams.

In some cases, you may need to create a stream on the content of a map. There is no `stream()` method in the `Map` interfa you cannot create such a stream directly. But you can access the content of a map through three collections:

- the set of the keys, with `keySet()`
- the set of the key-value pairs, with `entrySet()`
- the collection of the values, with `values()`.

The right pattern to use is to get one of these collections and create a stream on it.

The Stream API gives you a pattern to create a stream from a simple iterator. An iterator is a very simple object to create, may be a very convenient way to create a stream on a nonstandard source of data. The pattern is the following.

```
Iterator<String> iterator = ...;

long estimateSize = 10L;
int characteristics = 0;
Spliterator<String> spliterator = Spliterators.spliterator(strings.iterator(), estimateSize, characterist

boolean parallel = false;
Stream<String> stream = StreamSupport.stream(spliterator, parallel);
```

This pattern contains several magical elements that will be covered later in this tutorial. Let quickly browse through them.

The `estimateSize` is the number of elements you think this stream will be consuming. There are cases where this inform is simple to get: for example if you are creating a stream on an array or a collection. But there are also cases where this information is unknown.

The parameter `characteristics` will be covered later in this tutorial. It is used to optimize the processing of your data.

The `parallel` argument tells the API if the stream you want to create is a parallel stream or not. Parallel streams will be a covered later in this tutorial.

Creating an Empty Stream

Let us begin with the simplest of these patterns: the creation of an empty stream. There is a factory method for that in th `Stream` interface. You can use it in the following way.

```
Stream<String> empty = Stream.empty();
List<String> strings = empty.collect(Collectors.toList());

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

Running this code displays the following on your console.

```
strings = []
```

There are cases where creating an empty stream may be very handy. In fact, you saw one in the previous part of this tutor The pattern you saw uses empty streams and flatmap to remove invalid elements from a stream. Starting with Java SE 1 pattern has been replaced with the `mapMulti()` pattern.

Creating a Stream from a Vararg or an Array

These first two patterns are very similar. The first one uses the `of()` factory method in the `Stream` interface. The second c uses the `stream()` factory method of the `Arrays` factory class. And in fact, if you check the source of the `Stream.of()` m you will see that it calls `Arrays.stream()`.

Here is this first pattern in action.

```
Stream<Integer> intStream = Stream.of(1, 2, 3);
List<Integer> ints = intStream.collect(Collectors.toList());

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

Running this first example gives you the following:

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

Here is the second one.

```
String[] stringArray = {"one", "two", "three"};
Stream<String> stringStream = Arrays.stream(stringArray);
List<String> strings = stringStream.collect(Collectors.toList());

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

Running this second example gives you the following:

```
strings = [one, two, three]
```

Creating a Stream from a Supplier

There are two factory methods on the `Stream` interface for that.

The first one is `generate()` that takes a supplier as an argument. Everytime a new element is needed, this supplier is call

You can create such a stream with the following code, but don't do it!

```
Stream<String> generated = Stream.generate(() -> "+");
List<String> strings = generated.collect(Collectors.toList());
```

If you run this code (once again, don't), you will see that it will never stop. If you did and were patient enough, you may see `OutOfMemoryError`. If not, you are good to terminate your application through your IDE. This stream produces elements a never stops. It really produces an infinite stream.

We have not covered this point yet, but it is perfectly legal to have such streams! You may be wondering what could be th of them? In fact there are many. To use them, you need to cut this stream at some point, and the Stream API gives you se way to do it. You already saw one, and there are more to come.

The one you saw is to call `limit()` on that stream. Let us rewrite the previous example, and fix it.

```
Stream<String> generated = Stream.generate(() -> "+");
List<String> strings =
        generated

```

```
            .limit(10L)
            .collect(Collectors.toList());

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

Running this code prints the following.

```
strings = [+, +, +, +, +, +, +, +, +, +]
```

The `limit()` method is called a *short-circuiting* method: it can stop the consumption of the elements of a stream. You ma remember that the data is processed on element at a time in a stream: each element traverses all the operations defined stream, from the first one to the last one. This is the reason why this limit operation can stop the generation of more elem

Creating a Stream from a UnaryOperator and a Seed

Using a supplier is great if you need to generate constant streams. If you need an infinite stream with varying values, then can use the `iterate()` pattern.

This pattern works with a seed, which is the first generated element. Then it uses a `UnaryOperator` to generate the next e of the stream by transforming the previous element.

```
Stream<String> iterated = Stream.iterate("+", s -> s + "+");
iterated.limit(5L).forEach(System.out::println);
```

You should see the following result.

```
+
++
+++
++++
+++++
```

Do not forget to limit the number of elements processed by your stream when using this pattern.

Starting with Java SE 9, this pattern has an overload, which takes a predicate as an argument. The `iterate()` method sto generating elements when this predicate becomes false. The previous code can use this pattern in the following way.

```
Stream<String> iterated = Stream.iterate("+", s -> s.length() <= 5, s -> s + "+");
iterated.forEach(System.out::println);
```

Running this code gives you the same result as the previous one.

Creating a Stream from a Range of Numbers

It is easy to create a range of numbers with the previous pattern. But it is even easier with the specialized streams of num and their `range()` factory methods.

The `range()` method takes the initial value and the upper bound of the range, excluded. You can also include the upper bo with the `rangeClosed()` method. Calling `LongStream.range(0L, 10L)` will simply generate a stream with all the longs be 0 and 9.

This range() method can also be used to iterate through the elements of an array. Here is how you can do that.

```
String[] letters = {"A", "B", "C", "D"};
List<String> listLetters =
    IntStream.range(0, 10)
             .mapToObj(index -> letters[index % letters.length])
             .collect(Collectors.toList());
System.out.println("listLetters = " + listLetters);
```

The result is the following.

```
listLetters = [A, B, C, D, A, B, C, D, A, B]
```

There are plenty of things you can do, based on this pattern. Note that because IntStream.range() creates an IntStrea need to use the mapToObj() method to map it to a stream of objects.

Creating a Stream of Random Numbers

The Random class is used to create random series of numbers. Starting with Java SE 8, several methods have been added class to create stream of random numbers of different types: int, long, and double.

You can create an instance of Random providing a seed. This seed is a long parameter. The random numbers depends on seed. For a given seed you will always get the same sequence of numbers. This may be handy in many circumstances, in the writing of tests. In that case, you can rely on a sequence of numbers that is known in advance.

There are three methods to generate such a stream, all defined in the Random class: ints(), longs(), and doubles().

Several overloads are available for all these methods, which accept the following arguments:

- the number of elements this stream will generate;
- the upper and lower bounds of the random numbers generated.

Here is a first pattern of code that generates 10 random integers between 0 and 5.

```
Random random = new Random(314L);
List<Integer> randomInts =
    random.ints(10, 1, 5)
          .boxed()
          .collect(Collectors.toList());
System.out.println("randomInts = " + randomInts);
```

If you used the same seed as the one used in this example, you will have the following in your console.

```
randomInts = [4, 4, 3, 1, 1, 1, 2, 2, 4, 2]
```

Note that we used the boxed() method available on the specialized stream of numbers, which simply maps this stream t equivalent stream of wrapper types. So an IntStream is mapped to a Stream<Integer> by this method.

Here is a second pattern that generates a stream of random booleans. Any element of that stream is true with a probabili 80%.

```
Random random = new Random(314L);
List<Boolean> booleans =
    random.doubles(1_000, 0d, 1d)
```

```
            .mapToObj(rand -> rand <= 0.8) // you can tune the probability here
            .collect(Collectors.toList());

// Let us count the number of true in this list
long numberOfTrue =
    booleans.stream()
             .filter(b -> b)
             .count();
System.out.println("numberOfTrue = " + numberOfTrue);
```

If you used the same seed as the one we used in this example, you will see the following result.

```
numberOfTrue = 773
```

You can adapt this pattern to generate any kind of object with the probability you need. Here is another example that gene a stream with the letters A, B, C, and D. The probability for each letter is the following:

- 50% of A;
- 30% of B;
- 10% of C;
- 10% of D.

```
Random random = new Random(314L);
List<String> letters =
    random.doubles(1_000, 0d, 1d)
          .mapToObj(rand ->
                    rand < 0.5 ? "A" : // 50% of A
                    rand < 0.8 ? "B" : // 30% of B
                    rand < 0.9 ? "C" : // 10% of C
                                 "D")  // 10% of D
          .collect(Collectors.toList());

Map<String, Long> map =
    letters.stream()
             .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));

map.forEach((letter, number) -> System.out.println(letter + " :: " + number));
```

With the same seed, you will get the following result.

```
A :: 470
B :: 303
C :: 117
D :: 110
```

The building of the map with this `groupingBy()` may look unclear to you at this point. Do not worry; this pattern will be co later in this tutorial.

Creating a Stream from the Characters of a String

The `String` class saw the addition of a `chars()` method in Java SE 8. This method returns an `IntStream` that gives you th characters of this string.

Each character is given as a code point, an integer that may remind you of the ASCII codes. In some cases, you may need convert this integer to a string, just holding this character.

You have two patterns to do that, depending on the version of the JDK you are using.

Up to Java SE 10, you can use the following code.

```
String sentence = "Hello Duke";
List<String> letters =
    sentence.chars()
            .mapToObj(codePoint -> (char)codePoint)
            .map(Object::toString)
            .collect(Collectors.toList());
System.out.println("letters = " + letters);
```

A `toString()` factory method has been added on the `Character` class in Java SE 11, that you can use to make this code simpler.

```
String sentence = "Hello Duke";
List<String> letters =
    sentence.chars()
            .mapToObj(Character::toString)
            .collect(Collectors.toList());
System.out.println("letters = " + letters);
```

Both codes print out the following.

```
letters = [H, e, l, l, o,  , D, u, k, e]
```

Creating a Stream from the Lines of a Text File

Being able to open a stream on a text file is a very powerful pattern.

The Java I/O API has a pattern to read a single line from a text file: `BufferedReader.readLine()`. You can call this metho a loop and read your entire text file line by line to process it.

Being able to process these lines with the Stream API gives you a more readable and more maintainable code.

There are several patterns to create such a stream.

If you need to refactor existing code based on the use of a buffered reader, then you can use the `lines()` method defined this object. If you are writing new code to process the content of your text file, then you can use the factory method `Files.lines()`. This last method takes a `Path` as an argument and has an overloaded method that takes a `CharSet` in ca file you are reading is not encoded in UTF-8.

You may be aware that a file resource, as any I/O resource, should be closed when you do not need it anymore. Since you using this file resource through the Stream API, you may be wondering how you are going to handle that.

Well the good news is that the `Stream` interface implements the `AutoCloseable` interface. A stream is itself a resource th can close in case you need. This was not really needed in all the in-memory examples that you saw, but it definitely is in th case.

Here is an example that counts the number of warnings in a log file.

```
Path log = Path.of("/tmp/debug.log"); // adjust to fit your installation
try (Stream<String> lines = Files.lines(log)) {

    long warnings =
        lines.filter(line -> line.contains("WARNING"))
             .count();
    System.out.println("Number of warnings = " + warnings);

} catch (IOException e) {
    // do something with the exception
}
```

The try-with-resources pattern will call the `close()` method of your stream, which will in turn properly close the text file yc parsed.

Creating a Stream from a Regular Expression

The last example of this series of patterns is a method added to the `Pattern` class to create a stream on the elements generated by the application of a regular expression to a string of characters.

Suppose you need to split a string on a given separator. You have two patterns to do that.

- You can call the `String.split()` method;
- Or, you can use the `Pattern.compile().split()` pattern.

Both patterns give you an array of strings, containing the resulting elements of the splitting.

You saw the pattern to create a stream from this array. Let us write this code.

```
String sentence = "For there is good news yet to hear and fine things to be seen";

String[] elements = sentence.split(" ");
Stream<String> stream = Arrays.stream(elements);
```

The `Pattern` class has also a method for you. What you can do is call `Pattern.compile().splitAsStream()`. Here is the you can write using this method.

```
String sentence = "For there is good news yet to hear and fine things to be seen";

Pattern pattern = Pattern.compile(" ");
Stream<String> stream = pattern.splitAsStream(sentence);
List<String> words = stream.collect(Collectors.toList());

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

Running this code produces the following result.

```
words = [For, there, is, good, news, yet, to, hear, and, fine, things, to, be, seen]
```

You may be wondering which of the two patterns is the best. To answer this question, you need to take a close look at the pattern. First, you create an array to store the result of the splitting, then you create a stream on this array.

There is no array creation in the second pattern, so less overhead.

You already saw that some streams may use *short-circuit* operations (more on this point later in this tutorial). If you have stream, splitting the whole string and creating the resulting array may be an important but useless overhead. It is not cert that your stream pipeline will consume all its elements to produce a result.

Even if your stream needs to consume all the elements to produce its result, storing all these elements in an array is still a overhead that you do not need to pay.

So in both cases, using the `splitAsStream()` pattern is better. It is better memory-wise, and in some cases, CPU-wise.

Creating a Stream with the Builder Pattern

Creating a stream using this pattern is a two-step process. First, you add the elements your stream will be consuming in t builder. Then you create the stream from this builder. Once your builder has been used to create your stream, you cannot more elements to it, nor you can use it again to build another stream. You will get an `IllegalStateException` if you do th

The pattern is the following.

```
Stream.Builder<String> builder = Stream.<String>builder();

builder.add("one")
       .add("two")
       .add("three")
       .add("four");

Stream<String> stream = builder.build();

List<String> list = stream.collect(Collectors.toList());
System.out.println("list = " + list);
```

Running this code prints the following.

```
list = [one, two, three, four]
```

Creating a Stream on an HTTP Source

The last pattern we cover in this tutorial is about analyzing the body of an HTTP response. You saw that you can create a on the lines of a text file, you can do the same on the body of an HTTP response. This pattern is given by the HTTP Client added to JDK 11.

Here is how it works. We are going to use it on a text available online: *The Tale of Two Cities*, by Charles Dickens, made av online by the Gutenberg Project here: https://www.gutenberg.org/files/98/98-0.txt

The beginning of the text file gives you information about the text itself. The book starts on the line that contains "A TALE TWO CITIES". The end of the file is the licence under which this file is distributed.

We only need the text of the book, and would like to remove the header and the footer of this distributed file.

```
// The URI of the file
URI uri = URI.create("https://www.gutenberg.org/files/98/98-0.txt");

// The code to open create an HTTP request

```

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder(uri).build();


// The sending of the request
HttpResponse<Stream<String>> response = client.send(request, HttpResponse.BodyHandlers.ofLines());
List<String> lines;
try (Stream<String> stream = response.body()) {
    lines = stream
        .dropWhile(line -> !line.equals("A TALE OF TWO CITIES"))
        .takeWhile(line -> !line.equals("*** END OF THE PROJECT GUTENBERG EBOOK A TALE OF TWO CITIES ***"
        .collect(Collectors.toList());
}
System.out.println("# lines = " + lines.size());
```

Running this code will print out the following.

```
# lines = 15904
```

The stream is created by the body handler you give as an argument to the `send()` method. The HTTP Client API gives you several body handlers. The one you need to consume the body as a stream is the one created by the factory method `HttpResponse.BodyHandlers.ofLines()`. This way of consuming the body of the response is very memory efficient. If yc your stream carefully, the body of the response will never be stored in memory.

We decided to put all the lines of the text in a list, but, depending on the processing you need to conduct on this data, you necessarily need to do that. In fact, in most cases it is probably a bad idea to store this data in memory.

Last update: *September 14, 2021*

Adding Intermediate Operations on a Stream ← **Creating Streams** → Reducing a Stream

Home > Tutorials > The Stream API > Creating Streams