5. Working with streams

1. FILTERING

1.1. Filtering with a predicate

- ***filter***(Predicate) - filter operation takes a predicate as argument

1.2. Filtering unique elements

- ***distinct***() - returns a stream with unique elements

2. SLICING A STREAM

2.1. Slicing using a predicate

- ***takeWhile***(Predicate) - stops once it has found an element that fails to match (e.g. sorted stream, where we need elements *below* some value)
- ***dropWhile*** - Once the predicate evaluates to true it stops and returns all the remaining elements, and it even works if there are an infinite number of remaining elements (e.g. sorted stream, where we need elements *above* some value)

2.2. Truncating a stream

- ***limit***(n) - returns another stream of size n

2.3. Skipping elements

- ***skip***(n) - return a stream that discards the first n elements

3. MAPPING

In SQL you can select a particular column from a table - the Streams API provides similar facilities through the map and flatMap methods

3.1. Applying a function to each element of a stream

- ***map***(Function) - function is applied to each element, mapping it into a new element

3.2. Flattening streams

- ***flatMap***(Function) - lets you replace each value of a stream with another stream and then concatenates all the generated streams into a single stream

4. FINDING AND MATCHING

Short-circuiting evaluation

4.1. Checking to see if a predicate matches at least one element

- ***anyMatch***(Predicate) - returns a boolean

4.2. Checking to see if a predicate matches all elements

- ***allMatch***(Predicate) - returns a boolean
- ***noneMatch***(Predicate) - returns a boolean

4.3. Finding an element

Return Optional incase of no match, instead of null

- ***findAny***() - returns an arbitrary element

4.4. Finding the first element

- ***findFirst***() - return the first element

findFirst vs findAny: findFirst is more constraining in parallel, so prefer findAny

5. REDUCING

reduction operations (a stream is reduced to a value)

5.1. Summing the elements

- reduce taking two arguments:
    - An initial value, here 0.
    - A BinaryOperator to combine two elements and produce a new value; e.g. (a, b) -> a + b
- reduce taking one argument:
    - No initial value
    - Returns an *Optional*

5.2. Maximum and minimum

- reduce with argument - ***(x, y) -> x < y ? x : y*** or ***Integer::min***

Benefit of the reduce method and parallelism

- the lambda passed to reduce can't change state and the operation needs to be associative and commutative so it can be executed in any order
- there's almost no modification to your code: stream() becomes parallelStream()

Stream operations: stateless vs. stateful

- map / filter: *stateless*: they don't have an internal state (assuming the user-supplied lambda or method reference has no internal mutable state)
- reduce / sum / max: need to have internal state to accumulate the result, but the internal state is small
- sorted / distinct: *stateful*: Both sorting and removing duplicates from a stream require knowing the previous history to do their job; the storage requirement of the operation is *unbounded*

Table 5.1. Intermediate and terminal operations

Operation	Type	Return type	Type/functional interface used	Function descriptor
filter	Intermediate	Stream	Predicate	T -> boolean
distinct	Intermediate (stateful-unbounded)	Stream		
takeWhile	Intermediate	Stream	Predicate	T -> boolean
dropWhile	Intermediate	Stream	Predicate	T -> boolean
skip	Intermediate (stateful-bounded)	Stream	long	
limit	Intermediate (stateful-bounded)	Stream	long	
map	Intermediate	Stream	Function<T, R>	T -> R
flatMap	Intermediate	Stream	Function<T, Stream>	T -> Stream
sorted	Intermediate (stateful-unbounded)	Stream	Comparator	(T, T) -> int
anyMatch	Terminal	boolean	Predicate	T -> boolean
noneMatch	Terminal	boolean	Predicate	T -> boolean
allMatch	Terminal	boolean	Predicate	T -> boolean
findAny	Terminal	Optional		
findFirst	Terminal	Optional		
forEach	Terminal	void	Consumer	T -> void
collect	Terminal	R	Collector<T, A, R>	
reduce	Terminal (stateful-bounded)	Optional	BinaryOperator	(T, T) -> T
count	Terminal	long		

7. NUMERIC STREAMS

7.1. Primitive stream specializations

- IntStream, DoubleStream, and LongStream, which respectively specialize the elements of a stream to be int, long, and double—and thereby avoid hidden boxing costs
- ***mapToInt, mapToDouble, and mapToLong***
- supports other convenience methods such as ***max, min, and average***
- Converting back to stream objects: ***boxed***
- primitive specialized version of Optional: ***OptionalInt, OptionalDouble, and OptionalLong***

7.2. Numeric ranges

- ***range and rangeClosed*** - in classes *IntStream and LongStream* . Both methods take the starting value of the range as the first parameter and the end value of the range as the second parameter. But range is exclusive, whereas rangeClosed is inclusive

8. BUILDING STREAMS

8.1. Streams from values

- ***Stream.of***()
- ***Stream.empty()***

8.2. Stream from nullable

- ***Stream.ofNullable***()
- This pattern can be particularly handy in conjunction with flatMap and a stream of values that may include nullable objects

8.3. Streams from arrays

- ***Arrays.stream***(int[])

8.4. Streams from files

- ***java.nio.file.Files.lines returns a Stream***

8.5. Streams from functions: creating infinite streams!

- Streams API provides two static methods to generate a *infinite stream* from a function: ***Stream.iterate and Stream.generate***
- generally sensible to use limit(n) on such streams to avoid printing an infinite number of values
- use takeWhile instead of filter, to terminate after a condition is met
- ***generate***: takes a lambda of type Supplier to provide new values. Let's look at an example of how to use it

SUMMARY

- The Streams API lets you express complex data processing queries. Common stream operations are summarized in Table 5.1.
- You can filter and slice a stream using the filter, distinct, takeWhile (Java 9), dropWhile (Java 9), skip, and limit methods.
- The methods takeWhile and dropWhile are more efficient than a filter when you know that the source is sorted.

- You can extract or transform elements of a stream using the map and flatMap methods.
- You can find elements in a stream using the findFirst and findAny methods. You can match a given predicate in a stream using the allMatch, noneMatch, and anyMatch methods.
- These methods make use of short-circuiting: a computation stops as soon as a result is found; there's no need to process the whole stream.
- You can combine all elements of a stream iteratively to produce a result using the reduce method, for example, to calculate the sum or find the maximum of a stream.
- Some operations such as filter and map are stateless: they don't store any state. Some operations such as reduce store state to calculate a value. Some operations such as sorted and distinct also store state because they need to buffer all the elements of a stream before returning a new stream. Such operations are called *stateful operations*.
- There are three primitive specializations of streams: IntStream, DoubleStream, and LongStream. Their operations are also specialized accordingly.
- Streams can be created not only from a collection but also from values, arrays, files, and specific methods such as iterate and generate.
- An infinite stream has an infinite number of elements (for example all possible strings). This is possible because the elements of a stream are only produced *on demand*. You can get a finite stream from an infinite stream using methods such as limit.