Java Streams, Part 1: An introduction to the java.util.stream library

Run functional-style queries on collections and other data sets

Brian Goetz

July 06, 2016
(First published May 09, 2016)

Explore the Java™Streams library, introduced in Java SE 8, in this series by Java Language Architect Brian Goetz. By taking advantage of the power of lambda expressions, the`java.util.stream`package makes it easy to run functional-style queries on collections, arrays, and other data sets.

View more content in this series

The major new language feature in Java SE 8 was *lambda expressions*. You can think of a lambda expression as an anonymous method; like methods, lambdas have typed parameters, a body, and a return type. But the real news wasn't lambda expressions themselves, but what they enable. Lambdas make it easy to **express behavior as data**, which in turn makes it possible to develop more-expressive and more-powerful libraries.

One such library, also introduced in Java SE 8, is the `java.util.stream` package (Streams), which enables the concise and declarative expression of possibly-parallel bulk operations on various data sources. Libraries like Streams could have been written in earlier versions of Java, but without a compact behavior-as-data idiom they would have been so cumbersome to use that no one would have wanted to use them. You can consider Streams to be the first library to take advantage of the power of lambda expressions in Java, but there's nothing magical about it (though it is tightly integrated into the core JDK libraries). Streams isn't part of the language — it's a carefully designed library that takes advantage of some newer language features.

About this series
With the `java.util.stream` package, you can concisely and declaratively express possibly-parallel bulk operations on collections, arrays, and other data sources. In this series by Java Language Architect Brian Goetz, get a comprehensive understanding of the Streams library and learn how to use it to best advantage.

This article is the first in a series that explores the `java.util.stream` library in depth. This installment introduces you to the library and gives you an overview of its advantages and design

principles. In subsequent installments, you learn how to use streams to aggregate and summarize data and get a look at the library's internals and performance optimizations.

Querying with streams

One of the most common uses of streams is to represent *queries* over data in collections. Listing 1 shows an example of a simple stream pipeline. The pipeline takes a collection of transactions modeling a purchase between a buyer and a seller, and computes the total dollar value of transactions by sellers living in New York.

Listing 1. A simple stream pipeline

```
int totalSalesFromNY
    = txns.stream()
          .filter(t -> t.getSeller().getAddr().getState().equals("NY"))
          .mapToInt(t -> t.getAmount())
          .sum();
```

> *“ Streams exploit that most powerful of computing principles: composition. ”*

The `filter()` operation selects only transactions with sellers from New York. The `mapToInt()` operation selects the transaction amount for the desired transactions. And the terminal `sum()` operation adds up these amounts.

Although this example is pretty and easy to read, detractors might point out that the imperative (`for`-loop) version of this query is also simple and takes fewer lines of code to express. But the problem doesn't have to get much more complicated for the benefits of the stream approach to become evident. Streams exploit that most powerful of computing principles: composition. By composing complex operations out of simple building blocks (filtering, mapping, sorting, aggregation), streams queries are more likely to remain straightforward to write and read as the problem gets complicated than are more ad-hoc computations on the same data sources.

As a more complex query from the same domain as Listing 1, consider "Print the names of sellers in transactions with buyers over age 65, sorted by name." Writing this query the old-fashioned (imperative) way might yield something like Listing 2.

Listing 2. Ad-hoc query over a collection

```
Set<Seller> sellers = new HashSet<>();
for (Txn t : txns) {
    if (t.getBuyer().getAge() >= 65)
        sellers.add(t.getSeller());
}
List<Seller> sorted = new ArrayList<>(sellers);
Collections.sort(sorted, new Comparator<Seller>() {
    public int compare(Seller a, Seller b) {
        return a.getName().compareTo(b.getName());
    }
});
for (Seller s : sorted)
    System.out.println(s.getName());
```

Although this query is only slightly more complex than the first one, it's clear that the organization and readability of the resulting code under the imperative approach have already started to fall apart. The first thing the reader sees is neither the starting nor ending point of the computation; it's the declaration of a throwaway intermediate result. To read this code, you need to mentally buffer a lot of context before figuring out what the code actually does. Listing 3 shows how you might rewrite this query using Streams.

Listing 3. Query from Listing 2, expressed using Streams

```
txns.stream()
    .filter(t -> t.getBuyer().getAge() >= 65)
    .map(Txn::getSeller)
    .distinct()
    .sorted(comparing(Seller::getName))
    .map(Seller::getName)
    .forEach(System.out::println);
```

The code in Listing 3 is far easier to read, because the user is neither distracted with "garbage" variables — like `sellers` and `sorted`— and doesn't have to keep track of a lot of context while reading the code; the code reads almost exactly like the problem statement. Code that's more readable is also less error prone, because maintainers are more likely to be able to correctly discern at first glance what the code does.

The design approach taken by libraries like Streams leads to a practical separation of concerns. The client is in charge of specifying the "what" of a computation, but the library has control over the "how." This separation tends to parallel the distribution of expertise; the client writer generally has better understanding of the problem domain, whereas the library writer generally has more expertise in the algorithmic properties of the execution. The key enabler for writing libraries that allow this sort of separation of concerns is the ability to pass behavior as easily as passing data, which in turn enables APIs where callers can describe the structure of a complex calculation, and then get out of the way while the library chooses the execution strategy.

Anatomy of a stream pipeline

All stream computations share a common structure: They have a *stream source*, zero or more *intermediate operations*, and a single *terminal operation*. The elements of a stream can be object references (`Stream<String>`) or they can be primitive integers (`IntStream`), longs (`LongStream`), or doubles (`DoubleStream`).

Because most of the data that Java programs consume is already stored in collections, many stream computations use collections as their source. The `Collection` implementations in the JDK have all been enhanced to act as efficient stream sources. But other possible stream sources also exist — such as arrays, generator functions, or built-in factories such as numeric ranges — and (as shown in the third installment in this series) it's possible to write custom stream adapters so that any data source can act as a stream source. Table 1 shows some of the stream-producing methods in the JDK.

Table 1. Stream sources in the JDK

Method	Description
`Collection.stream()`	Create a stream from the elements of a collection.
`Stream.of(T...)`	Create a stream from the arguments passed to the factory method.
`Stream.of(T[])`	Create a stream from the elements of an array.
`Stream.empty()`	Create an empty stream.
`Stream.iterate(T first, BinaryOperator<T> f)`	Create an infinite stream consisting of the sequence `first, f(first), f(f(first)), ...`
`Stream.iterate(T first, Predicate<T> test, BinaryOperator<T> f)`	(Java 9 only) Similar to `Stream.iterate(T first, BinaryOperator<T> f)`, except the stream terminates on the first elements for which the test predicate returns `false`.
`Stream.generate(Supplier<T> f)`	Create an infinite stream from a generator function.
`IntStream.range(lower, upper)`	Create an `IntStream` consisting of the elements from lower to upper, exclusive.
`IntStream.rangeClosed(lower, upper)`	Create an `IntStream` consisting of the elements from lower to upper, inclusive.
`BufferedReader.lines()`	Create a stream consisting of the lines from a `BufferedReader.`
`BitSet.stream()`	Create an `IntStream` consisting of the indexes of the set bits in a `BitSet`.
`CharSequence.chars()`	Create an `IntStream` corresponding to the chars in a `String`.

Intermediate operations — such as `filter()` (selecting elements matching a criterion), `map()` (transforming elements according to a function), `distinct()` (removing duplicates), `limit()` (truncating a stream at a specific size), and `sorted()`— transform a stream into another stream. Some operations, such as `mapToInt()`, take a stream of one type and return a stream of a different type; the example of Listing 1 starts as a `Stream<Transaction>` and later switches to an `IntStream`. Table 2 shows some of the intermediate stream operations.

Table 2. Intermediate stream operations

Operation	Contents
`filter(Predicate<T>)`	The elements of the stream matching the predicate
`map(Function<T, U>)`	The result of applying the provided function to the elements of the stream
`flatMap(Function<T, Stream<U>>`	The elements of the streams resulting from applying the provided stream-bearing function to the elements of the stream
`distinct()`	The elements of the stream, with duplicates removed

`sorted()`	The elements of the stream, sorted in natural order
`Sorted(Comparator<T>)`	The elements of the stream, sorted by the provided comparator
`limit(long)`	The elements of the stream, truncated to the provided length
`skip(long)`	The elements of the stream, discarding the first N elements
`takeWhile(Predicate<T>)`	(Java 9 only) The elements of the stream, truncated at the first element for which the provided predicate is not `true`
`dropWhile(Predicate<T>)`	(Java 9 only) The elements of the stream, discarding the initial segment of elements for which the provided predicate is `true`

Intermediate operations are always *lazy*: Invoking an intermediate operation merely sets up the next stage in the stream pipeline but doesn't initiate any work. Intermediate operations are further divided into *stateless* and *stateful* operations. Stateless operations (such as `filter()` or `map()`) can operate on each element independently, whereas stateful operations (such as `sorted()` or `distinct()`) can incorporate state from previously seen elements that affects the processing of other elements.

The processing of the data set begins when a terminal operation is executed, such as a reduction (`sum()` or `max()`), application (`forEach()`), or search (`findFirst()`) operation. Terminal operations produce a result or a side effect. When a terminal operation is executed, the stream pipeline is terminated, and if you want to traverse the same data set again, you can set up a new stream pipeline. Table 3 shows some of the terminal stream operations.

Table 3. Terminal stream operations

Operation	Description
`forEach(Consumer<T> action)`	Apply the provided action to each element of the stream.
`toArray()`	Create an array from the elements of the stream.
`reduce(...)`	Aggregate the elements of the stream into a summary value.
`collect(...)`	Aggregate the elements of the stream into a summary result container.
`min(Comparator<T>)`	Return the minimal element of the stream according to the comparator.
`max(Comparator<T>)`	Return the maximal element of the stream according to the comparator.
`count()`	Return the size of the stream.
`{any,all,none}Match(Predicate<T>)`	Return whether any/all/none of the elements of the stream match the provided predicate.
`findFirst()`	Return the first element of the stream, if present.
`findAny()`	Return any element of the stream, if present.

Streams versus collections

While streams can resemble collections superficially — you might think of both as containing data — in reality they differ significantly. A collection is a data structure; its main concern is the organization of data in memory, and a collection persists over a period of time. A collection might often be used as the source or target for a stream pipeline, but a stream's focus is on **computation**, not data. The data comes from elsewhere (a collection, array, generator function, or I/O channel) and is processed through a pipeline of computational steps to produce a result

or side effect, at which point the stream is finished. Streams provide no storage for the elements that they process, and the lifecycle of a stream is more like a point in time — the invocation of the terminal operation. Unlike collections, streams can also be infinite; correspondingly, some operations (`limit()`, `findFirst()`) are *short-circuiting* and can operate on infinite streams with finite computation.

Collections and streams also differ in the way that their operations are executed. Operations on collections are eager and mutative; when the `remove()` method is called on a `List`, after the call returns, you know that the list state was modified to reflect the removal of the specified element. For streams, only the terminal operation is eager; the others are lazy. Stream operations represent a functional transformation on their input (also a stream), rather than a mutative operation on a data set (filtering a stream produces a new stream whose elements are a subset of the input stream but doesn't remove any elements from the source).

Expressing a stream pipeline as a sequence of functional transformations enables several useful execution strategies, such as *laziness*, *short circuiting*, and *operation fusion*. Short-circuiting enables a pipeline to terminate successfully without examining all the data; queries such as "find the first transaction over $1,000" needn't examine any more transactions after a match is found. Operation fusion means that multiple operations can be executed in a single pass on the data; in the example in Listing 1, the three operations are combined into a single pass on the data — rather than first selecting all the matching transactions, then selecting all the corresponding amounts, and then adding them up.

The imperative version of queries like the ones in Listing 1 and Listing 3 often resort to materializing collections for the results of intermediate calculations, such as the result of filtering or mapping. Not only can these results clutter the code, but they also clutter the execution. Materialization of intermediate collections serves only the implementation, not the result, and it consumes compute cycles organizing intermediate results into data structures that will only be discarded.

Stream pipelines, in contrast, fuse their operations into as few passes on the data as possible, often a single pass. (Stateful intermediate operations, such as sorting, can introduce barrier points that necessitate multipass execution.) Each stage of a stream pipeline produces its elements lazily, computing elements only as needed, and feeds them directly to the next stage. You don't need a collection to hold the intermediate result of filtering or mapping, so you save the effort of populating (and garbage-collecting) the intermediate collections. Also, following a "depth first" rather than "breadth first" execution strategy (tracing the path of a single data element through the entire pipeline) causes the data being operated upon to more often be "hot" in cache, so you can spend more time computing and less time waiting for data.

In addition to using streams for computation, you might want to consider using streams to return aggregates from API methods, where previously you might have returned an array or collection. Returning a stream is often more efficient, since you don't have to copy all the data into a new array or collection. Returning a stream is also often more flexible; the form of collection the library chooses to return might not be what the caller needs, and it's easy to convert a stream into any

collection type. (The main situation in which returning a stream is inappropriate, and falling back to returning a materialized collection is better, is when the caller would need to see a consistent snapshot of the state at a point in time.)

Parallelism

A beneficial consequence of structuring computations as functional transformations is that you can easily switch between sequential and parallel execution with minimal changes to the code. The sequential expression of a stream computation and the parallel expression of the same computation are almost identical. Listing 4 shows how to execute the query from Listing 1 in parallel.

Listing 4. Parallel version of Listing 1

```
int totalSalesFromNY
    = txns.parallelStream()
          .filter(t -> t.getSeller().getAddr().getState().equals("NY"))
          .mapToInt(t -> t.getAmount())
          .sum();
```

> *“ Expressing a stream pipeline as a series of functional transformations enables several useful execution strategies, such as laziness, parallelism, short-circuiting, and operation fusion. ”*

The first line's request for a parallel stream instead of a sequential one is the only difference from Listing 1, because the Streams library effectively factors the description and structure of a computation from the strategy for executing it. Previously, going parallel entailed a complete rewrite of the code, which was not only expensive but also often error prone, because the resulting parallel code looked little like the sequential version.

All stream operations can be executed either sequentially or in parallel, but bear in mind that parallelism isn't magic performance dust. A parallel execution might be faster than, the same speed as, or slower than a sequential one. It's best to start out with sequential streams and apply parallelism when you know that you will get — and benefit from — a speedup. A later installment in this series returns to analyzing a stream pipeline for parallel performance.

The fine print

Because the Streams library is orchestrating the computation, but performing the computation involves callbacks to lambdas provided by the client, what those lambda expressions can do is subject to certain contraints. Violating these constraints could cause the stream pipeline to fail or compute an incorrect result. Additionally, for lambdas with side effects, the timing (or existence) of these side effects might be surprising in some cases.

Most stream operations require that the lambdas passed to them be *non-interfering* and *stateless*. Non-interfering means that they won't modify the stream source; stateless means that they won't

access (read or write) any state that might change during the lifetime of the stream operation. For reduction operations (for example, computing summary data such as `sum`, `min`, or `max`) the lambdas passed to these operations must be *associative* (or conform to similar requirements).

These requirements stem in part from the fact that the stream library might, if the pipeline executes in parallel, access the data source or invoke these lambdas concurrently from multiple threads. The restrictions are needed to ensure that the computation remains correct. (These restrictions also tend to result in more-straightforward, easier-to-understand code, regardless of parallelism.) You might be tempted to convince yourself that you can ignore these restrictions because you don't think a particular pipeline will ever run in parallel, but it's best to resist this temptation or else you'll be burying time bombs in your code. Make the effort to express your stream pipelines such that they'll be correct regardless of execution strategy.

The root of all concurrency risks is *shared mutable state*. One possible source of shared mutable state is the stream source. If the source is a traditional collection like `ArrayList`, the Streams library assumes that it remains unmodified during the course of a stream operation. (Collections explicitly designed for concurrent access, such as `ConcurrentHashMap`, are exempt from this assumption.) Not only does the noninterference requirement exclude the source being mutated by other threads during a stream operation, but the lambdas passed to stream operations themselves should also refrain from mutating the source.

In addition to not modifying the stream source, lambdas passed to stream operations should be stateless. For example, the code in Listing 5, which tries to eliminate any element that's twice some preceding element, violates this rule.

Listing 5. Stream pipeline using stateful lambdas (don't do this!)

```
HashSet<Integer> twiceSeen = new HashSet<>();
int[] result
    = elements.stream()
              .filter(e -> {
                  twiceSeen.add(e * 2);
                  return twiceSeen.contains(e);
              })
              .toArray();
```

If executed in parallel, this pipeline would produce incorrect results, for two reasons. First, access to the `twiceSeen` set is done from multiple threads without any coordination and therefore isn't thread safe. Second, because the data is partitioned, there's no guarantee that when a given element is processed, all elements preceding that element were already processed.

It's best if the lambdas passed to stream operations are entirely *side effect free*— that is, that they don't mutate any heap-based state or perform any I/O during their execution. If they do have side effects, it's their responsibility to provide any required coordination to ensure that such side effects are thread safe.

Further, it's not even guaranteed that all side effects will be executed. For example, in Listing 6, the library is free to avoid executing the lambda passed to `map()` entirely. Because the source has a known size, the `map()` operation is known to be size preserving, and the mapping doesn't affect the

result of the computation, the library can optimize the calculation by not performing the mapping at all! (This optimization can turn the computation from *O(n)* to *O(1)*, in addition to eliminating the work associated with invoking the mapping function).

Listing 6. Stream pipeline with side effects that might not get executed

```
int count =
    anArrayList.stream()
               .map(e -> { System.out.println("Saw " + e); e })
               .count();
```

The only case in which you would notice the effect of this optimization (other than the computation being dramatically faster) is if the lambda passed to `map()` had side effects — in which case you might be surprised if those side effects don't happen. Being able to make these optimizations rests on the assumption that stream operations are functional transformations. Most of the time, we like it when the library makes our code run faster with no effort on our part. The cost of being able to make optimizations like this is that we must accept some restrictions on what the lambdas we pass to stream operations can do, and on some of our reliance on side effects. (Overall, this is a pretty good trade.)

Conclusion to Part 1

The `java.util.stream` library provides a simple and flexible means to express possibly-parallel functional-style queries on various data sources, including collections, arrays, generator functions, ranges, or custom data structures. Once you start using it, you'll be hooked! The next installment looks at one of the most powerful features of the Streams library: aggregation.

Related topics

- Package documentation for java.util.stream
- Functional Programming in Java: Harnessing the Power of Lambda Expressions (Venkat Subramaniam, Pragmatic Bookshelf, 2014)
- Mastering Lambdas: Java Programming in a Multicore World (Maurice Naftalin, McGraw-Hill Education, 2014)
- Should I return a Collection or a Stream?
- RxJava library
- IBM Code: Java journeys