IBM **Developer**

Java

ARTICLE

Aggregating with Streams

Slice, dice, and chop data with ease

By Brian Goetz | Published May 9, 2016 - Updated July 6, 2016

Java

Part 1 in the *Java Streams* series introduced you to the `java.util.stream` library added in Java SE 8. This second installment focuses on one of the most important and flexible aspects of the Streams library — the ability to aggregate and summarize data.

The "accumulator antipattern"

The first example in Part 1 performed a simple summation with Streams, as shown in Listing 1.

Listing 1. Computing an aggregate value declaratively with Streams

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

Listing 2 shows how this example might be written the "old way."

Listing 2. Computing the same aggregate value imperatively

```
int sum = 0;
for (Txn t : txns) {
    if (t.getSeller().getAddr().getState().equals("NY"))
        sum += t.getAmount();
}
```

Part 1 offers several reasons why the new way is preferable, despite being longer than the old way:

- The code is clearer because it is cleanly factored into the composition of simple operations.
- The code is expressed declaratively (describing the desired result) rather than imperatively (a step-by-step procedure for how to compute the result).
- This approach scales more cleanly as the query being expressed gets more complicated.

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.

Some additional reasons apply to the particular case of aggregation. Listing 2 is an illustration of the *accumulator antipattern*, where the code starts out declaring and initializing a mutable accumulator variable (`sum`) and then proceeds to update the accumulator in a loop. Why is this bad? First, this style of code is difficult to parallelize. Without coordination (such as synchronization), every access to the accumulator would be a data race (and with coordination, the contention resulting from coordination would more than undermine the efficiency gain available from parallelism).

Another reason why the accumulator approach is less desirable is that it models the computation at too low a level — at the level of individual elements, rather than on the data set as a whole. “The sum of all the transaction amounts” is a more abstract and direct statement of the goal than “Iterate through the transaction amounts one by one in order, adding each amount to an accumulator that has been previously initialized to zero.”

So, if imperative accumulation is the wrong tool, what’s the right one? In this specific problem, you’ve seen a hint of the answer — the `sum()` method — but this is merely a special case of a powerful and general technique, *reduction*. Reduction is simple, flexible, and parallelizable, and operates at a higher level of abstraction than imperative accumulation.

Reduction

> *Reduction is simple, flexible, and parallelizable, and operates at a higher level of abstraction than imperative accumulation.“*

Reduction (also known as *folding*) is a technique from functional programming that abstracts over many different accumulation operations. Given a nonempty sequence X of elements x_1, x_2, ..., x_n of type T and a binary operator on T (represented here by *), the reduction of X under* is defined as:

> *< $_n$bsp/> < $_n$bsp/> < $_n$bsp/>(x $_1$ x $_2$ ... * x $_n$)*

When applied to a sequence of numbers using ordinary addition as the binary operator, reduction is simply summation. But many other operations can be described by reduction. If the binary operator is “take the larger of the two elements” (which could be represented in Java by the lambda expression `(x,y) -> Math.max(x,y)`, or more simply as the method reference `Math::max`), reduction corresponds to finding the maximal value.

By describing an accumulation as a reduction instead of with the accumulator antipattern, you describe the computation in a more **abstract** and **compact** way, as well as a more **parallel-friendly** way — provided your binary operator satisfies a simple condition: *associativity*. Recall that a binary operator * is *associative* if, for any elements a, b, and c:

> *((a b) c) = (a (b c))*

Associativity means that **grouping doesn’t matter**. If the binary operator is associative, the reduction can be safely performed in any order. In a sequential execution, the natural order of execution is from left to right; in a parallel execution, the data is partitioned into segments, reduce each segment separately, and combine the results. Associativity ensures that these two approaches yield the same answer. This is easier to see if the definition of associativity is expanded to four terms:

> *(((a b) c) d) = ((a b) (c d))*

The left side corresponds to a typical sequential computation; the right side corresponds to a partitioned execution that would be typical of a parallel execution where the input sequence is broken into parts, the parts reduced in parallel, and the partial results combined with *. (Perhaps surprisingly,* need not be commutative, though many operators commonly used for reduction, such as as plus and max, are. An example of a binary operator that's associative but not commutative is string concatenation.)

The Streams library has several methods for reduction, including:

```
Optional<T> reduce(BinaryOperator<T> op)
T reduce(T identity, BinaryOperator<T> op)
```

The simpler of these methods takes only an associative binary operator and computes the reduction of the stream elements under that operator. The result is described as an `Optional`; if the input stream is empty, the result of reduction is also empty. (If the input has only a single element, the result of the reduction is that element.) If you had a collection of strings, you could compute the concatenation of the elements as:

```
String concatenated = strings.stream().reduce("", String::concat);
```

With the second of the two method forms, you provide an identity value, which is also used as the result if the stream is empty. The identity value must satisfy the constraints that for all *x*:

> *identity x = x x identity = x*

Not all binary operators have identity values, and when they do, they might not yield the results that you're looking for. For example, when computing maxima, you might be tempted to use the value `Integer.MIN_VALUE` as your identity (it does satisfy the requirements). But the result when that identity is used on an empty stream might not be what you want; you wouldn't be able to tell the difference between an empty input and a nonempty input containing only `Integer.MIN_VALUE`. (Sometimes that's not a problem, and sometimes it is — which is why the Streams library leaves it to the client to specify, or not specify, an identity.)

For string concatenation, the identity is the empty string, so you could recast the previous example as:

```
String concatenated = strings.stream().reduce("", String::concat);
```

Similarly, you could describe integer summation over an array as:

```
int sum = Stream.of(ints).reduce(0, (x,y) -> x+y);
```

(In practice, though, you'd use the `IntStream.sum()` convenience method.)

Reduction needn't apply only to integers and strings; it can be applied in any situation where you want to reduce a sequence of elements down to a single element of that type. For example, you can compute the tallest person via reduction:

```
Comparator<Person> byHeight = Comparators.comparingInt(Person::getHeight);
BinaryOperator<Person> tallerOf = BinaryOperator.maxBy(byHeight);
Optional<Person> tallest = people.stream().reduce(tallerOf);
```

If the provided binary operator isn't associative, or the provided identity value isn't actually an identity for the binary operator, then when the operation is executed in parallel, the result might be incorrect, and different executions on the same data set might produce different results.

Mutable reduction

Reduction takes a sequence of values and reduces it to a single value, such as the sequence's sum or its maximal value. But sometimes you don't want a single summary value; instead, you want to organize the results into a data structure like a `List` or `Map`, or reduce it to more than one summary value. In that case, you should use the mutable analogue of `reduce`, called `collect`.

Consider the simple case of accumulating elements into a `List`. Using the accumulator antipattern, you might write it this way:

```
ArrayList<String> list = new ArrayList<>();
for (Person p : people)
    list.add(p.toString());
```

Just as reduction is a better alternative to accumulation when the accumulator variable is a simple value, there's also a better alternative when the accumulator result is a more complex data structure. The building blocks of reduction are an identity value and a means of combining two values into a new value; the analogues for mutable reduction are:

- A means of producing an empty result container
- A means of incorporating a new element into a result container
- A means of merging two result containers

These building blocks can be easily expressed as functions. The third of these functions enables mutable reduction to occur in parallel: You can partition the data set, produce an intermediate accumulation for each section, and then merge the intermediate results. The Streams library has a `collect()` method that takes these three functions:

```
<R> collect(Supplier<R> resultSupplier,
            BiConsumer<R, T> accumulator,
            BiConsumer<R, R> combiner)
```

In the preceding section, you saw an example of using reduction to compute string concatenation. That idiom produces the correct result, but — because strings are immutable in Java and concatenation entails copying the whole string — it will have $O(n^2)$ runtime (some strings will be copied many times). You can express string concatenation more efficiently by collecting into a `StringBuilder`:

```
StringBuilder concat = strings.stream()
                              .collect(() -> new StringBuilder(),
                                       (sb, s) -> sb.append(s),
                                       (sb, sb2) -> sb.append(sb2));
```

This approach uses a `StringBuilder` as the result container. The three functions passed to `collect()` use the default constructor to create an empty container, the `append(String)` method to add an element to the container, and the `append(StringBuilder)` method to merge one container into another. This code is probably better expressed using method references rather than lambdas:

```
StringBuilder concat = strings.stream()
                              .collect(StringBuilder::new,
                                       StringBuilder::append,
                                       StringBuilder::append);
```

Similarly, to collect a stream into a `HashSet`, you could do this:

```
Set<String> uniqueStrings = strings.stream()
                                   .collect(HashSet::new,
                                            HashSet::add,
                                            HashSet::addAll);
```

In this version, the result container is a `HashSet` rather than a `StringBuilder`, but the approach is the same: Use the default constructor to create a new result container, the `add()` method to incorporate a new element to the set, and the `addAll()` method to merge two sets. It's easy to see how to adapt this code to any other sort of collection.

You might think, because a mutable result container (a `StringBuilder` or `HashSet`) is being used, that this is also an example of the accumulator antipattern. However, that's not the case. The analogue of the accumulator antipattern in this case would be:

```
Set<String> set = new HashSet<>();
strings.stream().forEach(s -> set.add(s));
```

> *Collectors can be composed together to perform more complex aggregations."*

Just as reduction can parallelize safely provided the combining function is associative and free of interfering side effects, mutable reduction with `Stream.collect()` can parallelize safely if it meets certain simple consistency requirements (outlined in the specification for `collect()`). The key difference is that, with the `forEach()` version, multiple threads are trying to access a single result container simultaneously, whereas with parallel `collect()`, each thread has its own local result container, the results of which are merged afterward.

Collectors

The relationship among the three functions passed to `collect()`— creating, populating, and merging result containers — is important enough to be given its own abstraction, `Collector`, along with a corresponding simplified version of `collect()`. The string-concatenation example can be rewritten as:

```
String concat = strings.stream().collect(Collectors.joining());
```

And the collect-to-set example can be rewritten as:

```
Set<String> uniqueStrings = strings.stream().collect(Collectors.toSet());
```

The `Collectors` class contains factories for many common aggregation operations, such as accumulation to collections, string concatenation, reduction and other summary computation, and the creation of summary tables (via `groupingBy()`). Table 1 contains a partial list of built-in collectors, and if these aren't sufficient, it's easy to write your own (see the " Custom collectors" section).

Table 1. Built-in collectors

Collector	Behavior
`toList()`	Collect the elements to a `List`.

Collector	Behavior
`toSet()`	Collect the elements to a `Set`.
toCollection(Supplier(less-thanCollection>)	Collect the elements to a specific kind of `Collection`.
toMap(Function(less-thanT, K>, Function(less-thanT, V>)	Collect the elements to a `Map`, transforming the elements into keys and values according to the provided mapping functions.
summingInt(ToIntFunction(less-thanT>)	Compute the sum of applying the provided `int`-valued mapping function to each element (also versions for `long` and `double`).
summarizingInt(ToIntFunction(less-thanT>)	Compute the `sum`, `min`, `max`, `count`, and `average` of the results of applying the provided `int`-valued mapping function to each element (also versions for `long` and `double`).
`reducing()`	Apply a reduction to the elements (usually used as a downstream collector, such as with `groupingBy`) (various versions).
partitioningBy(Predicate(less-thanT>)	Divide the elements into two groups: those for which the supplied predicate holds and those for which it doesn't.
partitioningBy(Predicate(less-thanT>, Collector)	Partition the elements, and process each partition with the specified downstream collector.
groupingBy(Function(less-thanT,U>)	Group elements into a `Map` whose keys are the provided function applied to the elements of the stream, and whose values are lists of elements that share that key.
groupingBy(Function(less-thanT,U>, Collector)	Group the elements, and process the values associated with each group with the specified downstream collector.
minBy(BinaryOperator(less-thanT>)	Compute the minimal value of the elements (also `maxBy()`).
mapping(Function(less-thanT,U>, Collector)	Apply the provided mapping function to each element, and process with the specified downstream collector (usually used as a downstream collector itself, such as with `groupingBy`).
`joining()`	Assuming elements of type `String`, join the elements into a string, possibly with a delimiter, prefix, and suffix.
`counting()`	Compute the count of elements. (Usually used as a downstream collector.)

Grouping the collector functions together into the `Collector` abstraction is syntactically simpler, but the real benefit comes from when you start to compose collectors together — such as when you want to create complex summaries such as those created by the `groupingBy()` collector, which collects elements into a `Map` according to a key derived from the element. For example, to create a `Map` of transactions over $1,000, keyed by seller:

```
Map<Seller, List<Txn>> bigTxnsBySeller =
    txns.stream()
        .filter(t -> t.getAmount() > 1000)
        .collect(groupingBy(Txn::getSeller));
```

Suppose, though, that you don't want a `List` of transactions for each seller, but instead the largest transaction from each seller. You still want to key the result by seller, but you want to do further processing of the transactions associated with that seller, to reduce it down to the largest transaction. You can use an alternative version of `groupingBy()` that, rather than collecting the elements for each key into a list, feeds them to another collector (the *downstream* collector). For your downstream collector, you can choose a reduction such as `maxBy()`:

```
Map<Seller, Txn> biggestTxnBySeller =
    txns.stream()
        .collect(groupingBy(Txn::getSeller,
                            maxBy(comparing(Txn::getAmount))));
```

Here, you group the transactions into a map keyed by seller, but the value of that map is the result of using the `maxBy()` collector to collect all the sales by that seller. If you want not the largest transaction by seller, but the sum, you could use the `summingInt()` collector:

```
Map<Seller, Integer> salesBySeller =
    txns.stream()
         .collect(groupingBy(Txn::getSeller,
                            summingInt(Txn::getAmount)));
```

To get a multilevel summary, such as sales by region and seller, you can simply use another `groupingBy` collector as the downstream collector:

```
Map<Region, Map<Seller, Integer>> salesByRegionAndSeller =
    txns.stream()
        .collect(groupingBy(Txn::getRegion,
                            groupingBy(Txn::getSeller,
                                       summingInt(Txn::getAmount))));
```

To pick an example from a different domain: To compute a histogram of word frequencies in a document, you can split the document into lines with `BufferedReader.lines()`, break it up into a stream of words by using `Pattern.splitAsStream()`, and then use `collect()` and `groupingBy()` to create a `Map` whose keys are words and whose values are counts of those words, as shown in Listing 3.

Listing 3. Computing a word-count histogram with Streams

```
    Pattern whitespace = Pattern.compile("\s+");
    Map<String, Integer> wordFrequencies =
        reader.lines()
              .flatMap(s -> whitespace.splitAsStream())
              .collect(groupingBy(String::toLowerCase),
                                  Collectors.counting());
```

Custom collectors

While the standard set of collectors provided by the JDK is fairly rich, it's also easy to write your own collector. The `Collector` interface, shown in Listing 4, is fairly simple. The interface is parameterized by three types — the input type `T`, the accumulator type `A`, and the final return type `R` (`A` and `R` are often the same) — and the methods return functions that look similar to the functions accepted by the three-argument version of `collect()` illustrated earlier.

Listing 4. The Collector interface

```
public interface Collector<T, A, R> {
    / Return a function that creates a new empty result container */
    Supplier<A> supplier();
    / Return a function that incorporates an element into a container /
    BiConsumer<A, T> accumulator();

    /** Return a function that merges two result containers /
    BinaryOperator<A> combiner();

    / Return a function that converts the intermediate result container
        into the final representation */
```

Show more

The implementation of most of the collector factories in `Collectors` is trivial. For example, the implementation of `toList()` is:

```
return new CollectorImpl<>(ArrayList::new,
                           List::add,
                           (left, right) -> { left.addAll(right); return left; },
                           CH_ID);
```

This implementation uses `ArrayList` as the result container, `add()` to incorporate an element, and `addAll()` to merge one list into another, indicating through the characteristics that its finish function is the identity function (which enables the stream framework to optimize the execution).

As you've seen before, there are some consistency requirements that are analogous to the constraints between the identity and the accumulator function in reduction. These requirements are outlined in the specification for `Collector`.

As a more complex example, consider the problem of creating summary statistics on a data set. It's easy to use reduction to compute the sum, minimum, maximum, or count of a numeric data set (and you can compute average from sum and count). Its harder to compute them all at once, in a single pass on the data, using reduction. But you can easily write a `Collector` to do this computation efficiently (and in parallel, if you like).

The `Collectors` class contains a `collectingInt()` factory method that returns an `IntSummaryStatistics`, which does exactly what you want — compute `sum`, `min`, `max`, `count`, and `average` in one pass. The implementation of `IntSummaryStatistics` is trivial, and you can easily write your own similar collectors to compute arbitrary data summaries (or extend this one).

Listing 5 shows the `IntSummaryStatistics` class. The actual implementation has more detail (including getters to return the summary statistics), but the heart of it is the simple `accept()` and `combine()` methods.

Listing 5. The IntSummaryStatistics class used by the summarizingInt() collector

```
public class IntSummaryStatistics implements IntConsumer {
    private long count;
    private long sum;
    private int min = Integer.MAX_VALUE;
    private int max = Integer.MIN_VALUE;

    public void accept(int value) {
        ++count;
        sum += value;
        min = Math.min(min, value);
        max = Math.max(max, value);
    }
```

Show more

As you can see, this is a pretty simple class. As each new data element is observed, the various summaries are updated in the obvious way, and two `IntSummaryStatistics` holders are combined in the obvious way. The implementation of `Collectors.summarizingInt()`, shown in Listing 6, is similarly simple; it creates a `Collector` that incorporates an element by applying an integer-valued extractor function and passing the result to `IntSummaryStatistics.accept()`.

Listing 6. The summarizingInt() collector factory

```
public static <T>
Collector<T, ?, IntSummaryStatistics> summarizingInt(ToIntFunction<? super T> mapper) {
    return new CollectorImpl<T, IntSummaryStatistics, IntSummaryStatistics>(
            IntSummaryStatistics::new,
            (r, t) -> r.accept(mapper.applyAsInt(t)),
            (l, r) -> { l.combine(r); return l; },
            CH_ID);
}
```

The ease of composing collectors (which you saw with the `groupingBy()` examples) and the ease of creating new collectors combine to make it possible to compute almost arbitrary summaries of stream data while keeping your code compact and clear.

Conclusion to Part 2

The facilities for aggregation are one of the most useful and flexible parts of the Streams library. Simple values can be easily aggregated, sequentially or in parallel, using reduction; more complex summaries can created via `collect()`. The library ships with a simple set of basic collectors that can be composed together to perform more complex aggregations, and you can easily add your own collectors to the mix.

In Part 3, dig into the internals of Streams to understand how to use the library most efficiently when performance is critical.

SOCIAL

CONTENTS

RESOURCES

Functional Programming in Java: Harnessing the Power of Lambda Expressions (Venkat Subramaniam, Pragmatic Bookshelf, 2014)

Should I return a Collection or a Stream?

IBM Code: Java journeys

Related content

MEETUP

Serverless: A New Way to Build Modern Applications

December 6, 2018

Containers Java +

BLOG | NOV 28, 2018

Java licensing is changing, and you could be affected

Java Java Platform

MEETUP

Serverless: A New Way to Build Modern Applications

December 6, 2018

Containers Java +

IBM **Developer**

About

Site Feedback & FAQ

Submit content

Report abuse

Third-party notice

Follow us

Select a language

English

中文

日本語

Русский

Português

Español

한글

Code Patterns

Articles

Tutorials

Recipes

Open Source Projects

Videos

Newsletters

Events

Cities

Developer Answers

Contact Privacy Terms of use Accessibility Feedback Cookie Preferences