Technical Article

Processing Data with Java SE 8 Streams, Part 1

by Raoul-Gabriel Urma

Use stream operations to express sophisticated data processing queries.

What would you do without collections? Nearly every Java application *makes* and *processes* collections. They are fundamental to many programming tasks: they let you group and process data. For example, you might want to create a collection of banking transactions to represent a customer's statement. Then, you might want to process the whole collection to find out how much money the customer spent. Despite their importance, processing collections is far from perfect in Java.

Originally published in the March/April 2014 issue of *Java Magazine*. Subscribe today.

First, typical processing patterns on collections are similar to SQL-like operations such as "finding" (for example, find the transaction with highest value) or "grouping" (for example, group all transactions related to grocery shopping). Most databases let you specify such operations declaratively. For example, the following SQL query lets you find the transaction ID with the highest value: `"SELECT id, MAX(value) from transactions"`.

As you can see, we don't need to implement *how* to calculate the maximum value (for example, using loops and a variable to track the highest value). We only express *what* we expect. This basic idea means that you need to worry less about how to explicitly implement such queries—it is handled for you. Why can't we do something similar with collections? How many times do you find yourself reimplementing these operations using loops over and over again?

Second, how can we process really large collections efficiently? Ideally, to speed up the processing, you want to leverage multicore architectures. However, writing parallel code is hard and error-prone.

Java SE 8 to the rescue! The Java API designers are updating the API with a new abstraction called *Stream* that lets you process data in a declarative way. Furthermore, streams can leverage multi-core architectures without you having to write a single line of multithread code. Sounds good, doesn't it? That's what this series of articles will explore.

Here's a mind-blowing idea: these two operations can produce elements "forever."

Before we explore in detail what you can do with streams, let's take a look at an example so you have a sense of the new programming style with Java SE 8 streams. Let's say we need to find all transactions of type `grocery` and return a list of transaction IDs sorted in decreasing order of transaction value. In Java SE 7, we'd do that as shown in **Listing 1**. In Java SE 8, we'd do it as shown in **Listing 2**.

```
List<Transaction> groceryTransactions = new Arraylist<>();
for(Transaction t: transactions){
  if(t.getType() == Transaction.GROCERY){
    groceryTransactions.add(t);
  }
}
Collections.sort(groceryTransactions, new Comparator(){
  public int compare(Transaction t1, Transaction t2){
    return t2.getValue().compareTo(t1.getValue());
  }
});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions){
  transactionsIds.add(t.getId());
}
```

Copy

Listing 1

```
List<Integer> transactionsIds =
    transactions.stream()
                .filter(t -> t.getType() == Transaction.GROCERY)
                .sorted(comparing(Transaction::getValue).reversed())
                .map(Transaction::getId)
                .collect(toList());
```

Copy

Listing 2

Figure 1 illustrates the Java SE 8 code. First, we obtain a stream from the list of transactions (the data) using the `stream()` method available on `List`. Next, several operations (`filter`, `sorted`, `map`, `collect`) are chained together to form a pipeline, which can be seen as forming a query on the data.

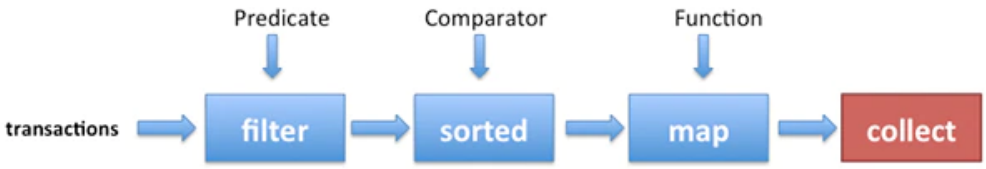

Figure 1

So how about parallelizing the code? In Java SE 8 it's easy: just replace `stream()` with `parallel Stream()`, as shown in **Listing 3**, and the Streams API will internally decompose your query to leverage the multiple cores on your computer.

```
List<Integer> transactionsIds =
    transactions.parallelStream()
                .filter(t -> t.getType() == Transaction.GROCERY)
                .sorted(comparing(Transaction::getValue).reversed())
                .map(Transaction::getId)
                .collect(toList());
```

Copy

Listing 3

Don't worry if this code is slightly overwhelming. We will explore how it works in the next sections. However, notice the use of lambda expressions (for example, `t-> t.getCategory() == Transaction.GROCERY`) and method references (for example, `Transaction::getId`), which you should be familiar with by now. (To brush up on lambda expressions, refer to previous *Java Magazine* articles and other resources listed at the end of this article.)

For now, you can see a stream as an abstraction for expressing efficient, SQL-like operations on a collection of data. In addition, these operations can be succinctly parameterized with lambda expressions.

At the end of this series of articles about Java SE 8 streams, you will be able to use the Streams API to write code similar to **Listing 3** to express powerful queries.

Getting Started with Streams

Let's start with a bit of theory. What's the definition of a stream? A short definition is "a sequence of elements from a source that supports aggregate operations." Let's break it down:

- **Sequence of elements:** A stream provides an interface to a sequenced set of values of a specific element type. However, streams don't actually store elements; they are computed on demand.
- **Source:** Streams consume from a data-providing source such as collections, arrays, or I/O resources.
- **Aggregate operations:** Streams support SQL-like operations and common operations from functional programing languages, such as `filter`, `map`, `reduce`, `find`, `match`, `sorted`, and so on.

Furthermore, stream operations have two fundamental characteristics that make them very different from collection operations:

- **Pipelining:** Many stream operations return a stream themselves. This allows operations to be chained to form a larger pipeline. This enables certain optimizations, such as *laziness* and *short-circuiting*, which we explore later.
- **Internal iteration:** In contrast to collections, which are iterated explicitly (*external iteration*), stream operations do the iteration behind the scenes for you.

Let's revisit our earlier code example to explain these ideas. **Figure 2** illustrates **Listing 2** in more detail.

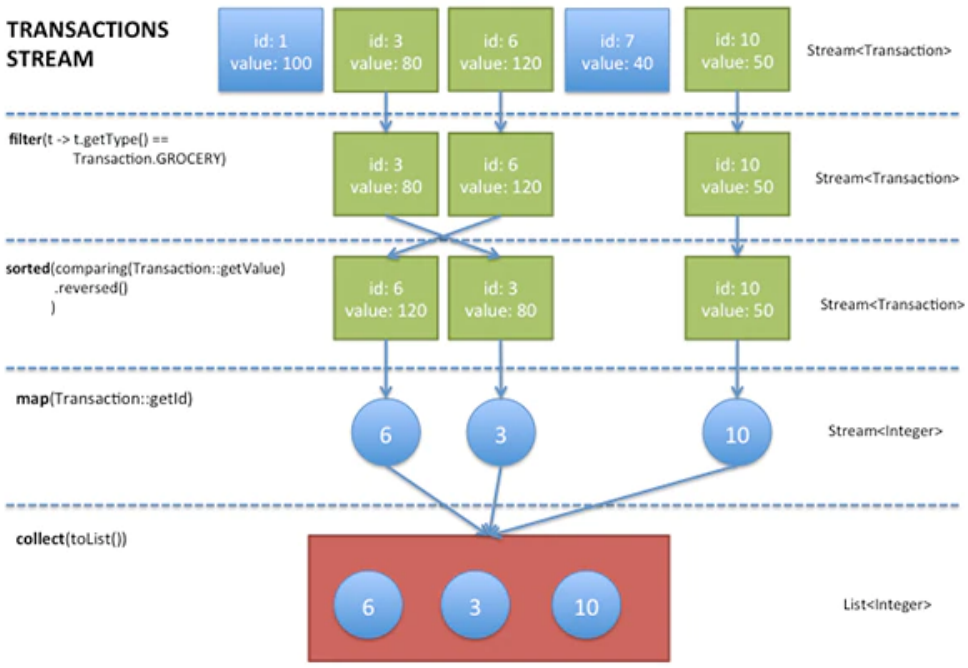

Figure 2

We first get a stream from the list of transactions by calling the `stream()` method. The datasource is the list of transactions and will be providing a sequence of elements to the stream. Next, we apply a series of aggregate operations on the stream: `filter` (to filter elements given a predicate), `sorted` (to sort the elements given a comparator), and `map` (to extract information). All these operations except `collect` return a `Stream` so they can be chained to form a pipeline, which can be viewed as a query on the source.

No work is actually done until `collect` is invoked. The `collect` operation will start processing the pipeline to return a result (something that is not a `Stream`; here, a `List`). Don't worry about `collect` for now; we will explore it in detail in a future article. At the moment, you can see `collect` as an operation that takes as an argument various recipes for accumulating the elements of a stream into a summary result. Here, `toList()` describes a recipe for converting a `Stream` into a `List`.

Before we explore the different methods available on a stream, it is good to pause and reflect on the conceptual difference between a stream and a collection.

Streams Versus Collections

Both the existing Java notion of collections and the new notion of streams provide interfaces to a sequence of elements. So what's the difference? In a nutshell, collections are about data and streams are about computations.

Consider a movie stored on a DVD. This is a collection (perhaps of bytes or perhaps of frames—we don't care which here) because it contains the whole data structure. Now consider watching the same video when it is being streamed over the internet. It is now a stream (of bytes or frames). The streaming video player needs to have downloaded only a few frames in advance of where the user is watching, so you can start displaying values from

the beginning of the stream before most of the values in the stream have even been computed (consider streaming a live football game).

In the coarsest terms, the difference between collections and streams has to do with *when* things are computed. A collection is an in-memory data structure, which holds all the values that the data structure currently has—every element in the collection has to be computed before it can be added to the collection. In contrast, a stream is a conceptually fixed data structure in which elements are computed on demand.

Using the `Collection` interface requires iteration to be done by the user (for example, using the enhanced `for` loop called `foreach`); this is called external iteration.

In contrast, the Streams library uses internal iteration—it does the iteration for you and takes care of storing the resulting stream value somewhere; you merely provide a function saying what's to be done. The code in **Listing 4** (external iteration with a collection) and **Listing 5** (internal iteration with a stream) illustrates this difference.

```
List<String> transactionIds = new ArrayList<>();
for(Transaction t: transactions){
    transactionIds.add(t.getId());
}
```

Copy

Listing 4

```
List<Integer> transactionIds =
    transactions.stream()
                .map(Transaction::getId)
                .collect(toList());
```

Copy

Listing 5

In **Listing 4**, we explicitly iterate the list of transactions sequentially to extract each transaction ID and add it to an accumulator. In contrast, when using a stream, there's no explicit iteration. The code in **Listing 5** builds a query, where the `map` operation is parameterized to extract the transaction IDs and the `collect` operation converts the resulting `Stream` into a `List`.

You should now have a good idea of what a stream is and what you can do with it. Let's now look at the different operations supported by streams so you can express your own data processing queries.

Stream Operations: Exploiting Streams to Process Data

The `Stream` interface in `java.util .stream.Stream` defines many operations, which can be grouped in two categories. In the example illustrated in **Figure 1**, you can see the following operations:

- `filter`, `sorted`, and `map`, which can be connected together to form a pipeline
- `collect`, which closed the pipeline and returned a result

Stream operations that can be connected are called *intermediate operations*. They can be connected together because their return type is a `Stream`. Operations that close a stream pipeline are called *terminal operations*. They produce a result from a pipeline such as a `List`, an `Integer`, or even `void` (any non-`Stream` type).

You might be wondering why the distinction is important. Well, intermediate operations do not perform any processing until a terminal operation is invoked on the stream pipeline; they are "lazy." This is because intermediate operations can usually be "merged" and processed into a single pass by the terminal operation.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
List<Integer> twoEvenSquares =
    numbers.stream()
           .filter(n -> {
                    System.out.println("filtering " + n);
                    return n % 2 == 0;
                })
           .map(n -> {
                    System.out.println("mapping " + n);
                    return n * n;
                })
           .limit(2)
           .collect(toList());
```

Copy

Listing 6

For example, consider the code in **Listing 6**, which computes two even square numbers from a given list of numbers. You might be surprised that it prints the following:

```
filtering 1
filtering 2
mapping 2
filtering 3
filtering 4
mapping 4
```

Copy

This is because `limit(2)` uses *short-circuiting*; we need to process only part of the stream, not all of it, to return a result. This is similar to evaluating a large Boolean expression chained with the `and` operator: as soon as one expression returns `false`, we can deduce that the whole expression is `false` without evaluating all of it. Here, the operation `limit` returns a stream of size `2`.

In addition, the operations `filter` and `map` have been merged in the same pass.

To summarize what we've learned so far, working with streams, in general, involves three things:

- A datasource (such as a collection) on which to perform a query
- A chain of intermediate operations, which form a stream pipeline

- One terminal operation, which executes the stream pipeline and produces a result

> *The Streams API will internally decompose* your query to leverage the multiple cores on your computer.

Let's now take a tour of some of the operations available on streams. Refer to the `java.util .stream.Stream` interface for the complete list, as well as to the resources at the end of this article for more examples.

Filtering. There are several operations that can be used to filter elements from a stream:

- `filter(Predicate)`: Takes a predicate (`java.util.function.Predicate`) as an argument and returns a stream including all elements that match the given predicate
- `distinct`: Returns a stream with unique elements (according to the implementation of `equals` for a stream element)
- `limit(n)`: Returns a stream that is no longer than the given size `n`
- `skip(n)`: Returns a stream with the first n number of elements discarded

Finding and matching. A common data processing pattern is determining whether some elements match a given property. You can use the `anyMatch`, `allMatch`, and `noneMatch` operations to help you do this. They all take a predicate as an argument and return a `boolean` as the result (they are, therefore, terminal operations). For example, you can use `allMatch` to check that all elements in a stream of transactions have a value higher than 100, as shown in **Listing 7**.

```
boolean expensive =
    transactions.stream()
                .allMatch(t -> t.getValue() > 100);
```

Copy

Listing 7

In addition, the `Stream` interface provides the operations `findFirst` and `findAny` for retrieving arbitrary elements from a stream. They can be used in conjunction with other stream operations such as `filter`. Both `findFirst` and `findAny` return an `Optional` object, as shown in **Listing 8**.

```
Optional<Transaction> =
    transactions.stream()
                .filter(t -> t.getType() == Transaction.GROCERY)
                .findAny();
```

Copy

Listing 8

The `Optional<T>` class (`java.util .Optional`) is a container class to represent the existence or absence of a value. In **Listing 8**, it is possible that `findAny` doesn't find any transaction of type `grocery`. The `Optional` class contains several methods to test the existence of an element. For example, if a transaction is present, we can choose to apply an operation on the optional object by using the `ifPresent` method, as shown in **Listing 9** (where we just print the transaction).

```
transactions.stream()
              .filter(t -> t.getType() == Transaction.GROCERY)
              .findAny()
              .ifPresent(System.out::println);
```

Copy

Listing 9

Mapping. Streams support the method `map`, which takes a function (`java.util.function.Function`) as an argument to project the elements of a stream into another form. The function is applied to each element, "mapping" it into a new element.

For example, you might want to use it to extract information from each element of a stream. In the example in **Listing 10**, we return a list of the length of each word from a list. **Reducing.** So far, the terminal operations we've seen return a `boolean` (`allMatch` and so on), `void` (`forEach`), or an `Optional` object (`findAny` and so on). We have also been using `collect` to combine all elements in a `Stream` into a `List`.

```
List<String> words = Arrays.asList("Oracle", "Java", "Magazine");
 List<Integer> wordLengths =
    words.stream()
         .map(String::length)
         .collect(toList());
```

Copy

Listing 10

However, you can also combine all elements in a stream to formulate more-complicated process queries, such as "what is the transaction with the highest ID?" or "calculate the sum of all transactions' values." This is possible using the `reduce` operation on streams, which repeatedly applies an operation (for example, adding two numbers) on each element until a result is produced. It's often called a *fold operation* in functional programming because you can view this operation as "folding" repeatedly a long piece of paper (your stream) until it forms one little square, which is the result of the fold operation.

It helps to first look at how we could calculate the sum of a list using a `for` loop:

```
int sum = 0;
for (int x : numbers) {
    sum += x;
}
```

Copy

Each element of the list of numbers is combined iteratively using the addition operator to produce a result. We essentially "reduced" the list of numbers into one number. There are two parameters in this code: the initial value of the `sum` variable, in this case `0`, and the operation for combining all the elements of the list, in this case `+`.

Using the `reduce` method on streams, we can sum all the elements of a stream as shown in **Listing 11**. The `reduce` method takes two arguments:

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

Copy

Listing 11

- An initial value, here `0`
- A `BinaryOperator<T>` to combine two elements and produce a new value

The `reduce` method essentially abstracts the pattern of repeated application. Other queries such as "calculate the product" or "calculate the maximum" (see **Listing 12**) become special use cases of the `reduce` method.

```
int product = numbers.stream().reduce(1, (a, b) -> a * b);
int product = numbers.stream().reduce(1, Integer::max);
```

Copy

Listing 12

Numeric Streams

You have just seen that you can use the `reduce` method to calculate the sum of a stream of integers. However, there's a cost: we perform many boxing operations to repeatedly add `Integer` objects together. Wouldn't it be nicer if we could call a `sum` method, as shown in **Listing 13**, to be more explicit about the intent of our code?

```
int statement =
    transactions.stream()
                .map(Transaction::getValue)
                .sum(); // error since Stream has no sum method
```

Copy

Listing 13

Java SE 8 introduces three primitive specialized stream interfaces to tackle this issue—`IntStream`, `DoubleStream`, and `LongStream`—that respectively specialize the elements of a stream to be `int`, `double`, and `long`.

The most-common methods you will use to convert a stream to a specialized version are `mapToInt`, `mapToDouble`, and `mapToLong`. These methods work exactly like the method `map` that we saw earlier, but they return a specialized stream instead of a `Stream<T>`. For example, we could improve the code in **Listing 13** as shown in **Listing 14**. You can also convert from a primitive stream to a stream of objects using the `boxed` operation.

```
int statementSum =
    transactions.stream()
                .mapToInt(Transaction::getValue)
                .sum(); // works!
```

Copy

Listing 14

Finally, another useful form of numeric streams is numeric ranges. For example, you might want to generate all numbers between 1 and 100. Java SE 8 introduces two static methods available on `IntStream`, `DoubleStream`, and `LongStream` to help generate such ranges: `range` and `rangeClosed`.

Both methods take the starting value of the range as the first parameter and the end value of the range as the second parameter. However, `range` is exclusive, whereas `rangeClosed` is inclusive. **Listing 15** is an example that uses `rangeClosed` to return a stream of all odd numbers between 10 and 30.

```
IntStream oddNumbers =
    IntStream.rangeClosed(10, 30)
             .filter(n -> n % 2 == 1);
```

Copy

Listing 15

Building Streams

There are several ways to build streams. You've seen how you can get a stream from a collection. Moreover, we played with streams of numbers. You can also create streams from values, an array, or a file. In addition, you can even generate a stream from a function to produce infinite streams!

Creating a stream from values or from an array is straightforward: just use the static methods `Stream .of` for values and `Arrays.stream` for an array, as shown in **Listing 16**.

```
Stream<Integer> numbersFromValues = Stream.of(1, 2, 3, 4);
int[] numbers = {1, 2, 3, 4};
IntStream numbersFromArray = Arrays.stream(numbers);
```

Listing 16

In contrast to collections, which are iterated explicitly (*external iteration*), *stream operations do the*

iteration behind the scenes for you.

You can also convert a file in a stream of lines using the `Files.lines` static method. For example, in **Listing 17** we count the number of lines in a file.

```
long numberOfLines =
    Files.lines(Paths.get("yourFile.txt"), Charset.defaultCharset())
         .count();
```

Copy

Listing 17

Infinite streams. Finally, here's a mind-blowing idea before we conclude this first article about streams. By now you should understand that elements of a stream are produced on demand. There are two static methods—`Stream.iterate` and `Stream .generate`—that let you create a stream from a function. However, because elements are calculated on demand, these two operations can produce elements "forever." This is what we call an *infinite stream*: a stream that doesn't have a fixed size, as a stream does when we create it from a fixed collection.

Learn More

Java 8 Lambdas in Action

GitHub repository with Java SE 8 code examples

"Java 8: Lambdas, Part 1" by Ted Neward

Listing 18 is an example that uses `iterate` to create a stream of all numbers that are multiples of 10. The `iterate` method takes an initial value (here, `0`) and a lambda (of type `UnaryOperator<T>`) to apply successively on each new value produced.

```
Stream<Integer> numbers = Stream.iterate(0, n -> n + 10);
```

Listing 18

We can turn an infinite stream into a fixed-size stream using the `limit` operation. For example, we can limit the size of the stream to 5, as shown in **Listing 19**.

```
numbers.limit(5).forEach(System.out::println); // 0, 10, 20, 30, 40
```

Listing 19

Conclusion

Java SE 8 introduces the Streams API, which lets you express sophisticated data processing queries. In this article, you've seen that a stream supports many operations such as `filter`, `map`, `reduce`, and `iterate` that can be combined to write concise and expressive data processing queries. This new way of writing code is very different from how you would process collections before Java SE 8. However, it has many benefits. First, the Streams API makes use of several techniques such as laziness and short-circuiting to optimize your data processing queries. Second, streams can be parallelized automatically to leverage multicore architectures. In the next article in this series, we will explore more-advanced operations, such as `flatMap` and `collect`. Stay tuned.

Raoul-Gabriel Urma is currently completing a PhD in computer science at the University of Cambridge, where he does research in programming languages. In addition, he is an author of *Java 8 in Action: Lambdas, Streams and Functional-style Programming* (Manning, 2014).