

Java 8 Streams

“Code with Passion!”



Topics

- What is and Why Streams?
- Intermediate and terminal operations
- Laziness
- Parallelism

What is and Why Streams?

What is a Stream?

- Wrappers around data sources
 - > Stream itself does not store data
 - > Stream carry values from a data source through a pipeline of operations
- Supports many convenient and high-performance operations expressed succinctly with Lambda expressions
 - > All Stream operations take Lambda expressions as arguments
- Operations can be executed in sequence or in parallel
 - > Parallel operation results in better performance when there are large number of items to process
- Support laziness
 - > Many Stream operations are postponed until how much data is eventually needed – this result in more efficient operations

Issues of using Collections for processing

- Issue #1: You must use “for” loop or “while” loop (external iteration) in order to get the answer you want - basically you must specify “how it needs to be done” instead of “what needs to be done” letting the system to handle the iteration (internal iteration)
 - > Example #1: You have a list of customers. You want to find out who spent the most during the month of May among these customers
 - > Example #2: You also want to compute the average age of these customers
- Issue #2: Writing parallelizable code with collection is very hard
 - > Example: The number of customers could be in the range of millions. And processing each of these customers in sequential manner could be very slow

Stream to the Rescue!

- Stream lets you process data **in a declarative way - simpler code**
 - > You specify “what needs to be done” not “how it needs to be done”
 - > More English-like description of a problem to solve (expressive)
- Stream support parallel processing (concurrency)
 - > Streams can leverage multi-core architectures without you having to write a single line of multi-threaded code
 - > Streams is easier to optimize due to “functional purity” (there is no global state to be maintained)

Example #1: Using Collection in Java 7

- Suppose you want to compute sum of all Integer's whose value is greater than 10, you would write code like following

```
private static int sumUsingForLoop(List<Integer> list) {  
    int sum = 0;  
    int currentIndex = 0;  
    for (Integer integer : list) {  
        if (list.get(currentIndex) > 10) {  
            sum += list.get(currentIndex);  
        }  
        currentIndex++;  
    }  
    return sum;  
}
```

You must specify
how it needs to be done
using either for loop or
while loop

Example #1: Using Stream in Java 8

- Same logic can be rewritten using Stream - simpler, fluent, parallel

// Non-parallel stream code

```
private static int sumStream(List<String> list) {  
    return list.stream()           // convert the List to sequential stream  
        .mapToInt(stringNumber -> Integer.parseInt(stringNumber) )  
        .filter(i -> i > 10)      // filter only the number > 10 (Predicate)  
        .sum();                  // perform sum operation  
}
```

// Parallel stream code

```
private static int sumStream(List<String> list) {  
    return list.parallelStream()   // convert the List to parallel stream  
        .mapToInt(stringNumber -> Integer.parseInt(stringNumber) )  
        .filter(i -> i > 10)      // filter only the number > 10 (Predicate)  
        .sum();                  // perform sum operation  
}
```


Example #2: Using Collection in Java 7

- You want to (1) find all transactions of type Grocery and (2) return a list of transaction IDs (3) sorted in decreasing order of transaction value

```
List<Transaction> groceryTransactions = new ArrayList<>();
```

```
for (Transaction t : allTransactions) {  
    if (t.getType() == TransactionType.GROCERY) {  
        groceryTransactions.add(t);  
    }  
}
```

Select Grocery transactions

Sort based on
transaction value

```
Collections.sort(groceryTransactions, new Comparator<Transaction>() {  
    public int compare(Transaction t1, Transaction t2) {  
        return t2.getValue()  
            .compareTo(t1.getValue());  
    }  
});
```

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

Return list of transaction ids

Example #2: Using Stream in Java 8

- Same logic can be rewritten using Stream - simpler, fluent, parallel

// Non-parallel

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

// Parallel

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

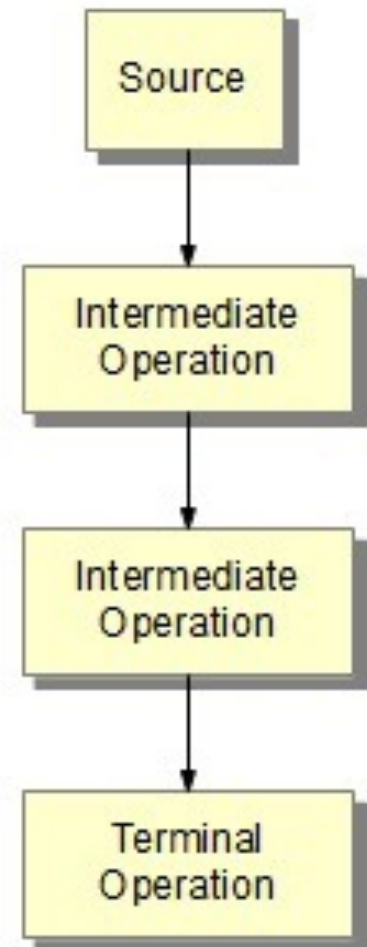
Collections vs Streams

- Collections are about data while streams are about computations
- Using the Collection interface requires iteration to be done by the developer (for example, using “for” or “while” loop)
 - > This is called **external iteration**
- In contrast, the Streams library 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
 - > This is called **internal iteration**

Intermediate and Terminal Operations

Stream Operations

- Stream-based processing is a **pipe of operations**
 - > Pipe is made of multiple intermediate operations
 - > And a single terminal operation at the end
- Intermediate operations
 - > *filter, map, sorted*
- Terminal operations
 - > *count, sum, reduce, forEach, findFirst, etc.*



Working with Stream involves

- Think of it as a pipeline that is made of
 - > A data source (such as a collection object)
 - > A chain of intermediate operations, which form a stream pipeline
 - > One terminal operation, which
 - > Represents end of pipeline and
 - > Triggers execution of the stream pipeline and produces a result

Types of Stream Operations

- Filtering operations
 - > *filter (Predicate)*
 - > *distinct*
 - > *limit(n)*
 - > *skip(n)*
- Finding and matching operations
 - > *anyMatch, allMatch, and noneMatch*
- Mapping operations
 - > *map*
- Reducing operations
 - > *reduce*

Intermediate Operations

- Stream producing
 - > Always returns another stream
- Always lazy
 - > Actual work is always triggered by a terminal operation
- Two types of intermediate operations
 - > Stateless operations - don't require information of other items
 - > Stateful operations - require information of other items
- Stateless operations
 - > *filter, map, ..*
- Stateful operations
 - > *distinct, sorted, limit, peek, ..*

Terminal Operations

- Represents end of pipeline
 - > Produces a result from a pipeline - the result could be a List, an Integer, or even void (any non-Stream type)
- Triggers intermediate operations (this is called Lazy)
 - > 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, thus results in efficient processing

Terminal Operations

- *forEach*
- *count and sum*
- *collect*
- *findFirst, findAny*
- *anyMatch, allMatch, noneMatch*
- *reduce*

Terminal Operation: forEach

- Performs an action for each element of a stream
- The behavior of this operation is non-deterministic
 - > For parallel stream pipelines, this operation does not guarantee to respect the encountering order of the stream, as doing so would sacrifice the benefit of parallelism

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
```

```
numbers.stream()  
    .filter(p -> p > 5)  
    .forEach(p -> System.out.println(p + " "));
```

Terminal Operation: count and sum

- *count*
 - > Returns the count of elements in the stream
- *sum*
 - > Returns the sum of values of elements in the stream

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
```

```
long count = numbers.stream().filter(n -> n > 3).count();  
int sum = numbers.stream().filter(n -> n > 3).mapToInt(i->i).sum();
```

Terminal Operation: collect

- *collect(..)* method performs a mutable reduction operation on the elements of this stream using a Collector
 - Examples of mutable reduction operations include: accumulating elements into a Collection
- *Collectors.toList()* collects *Stream* elements into a *List* instance
- *Collectors.toSet()* collects *Stream* elements into a *Set* instance

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

Terminal Operation: `findFirst`, `findAny`

- *`findFirst`*
 - > Returns an Optional describing the first element of the stream, or an empty Optional if the stream is empty

`Optional<T> findFirst()`
- *`findAny`*
 - > Returns an Optional describing some element of the stream, or an empty Optional if the stream is empty

`Optional<T> findAny()`

 - > The behavior of this operation is explicitly non-deterministic; it is free to select any element in the stream

```
Optional<String> firstNameWithD = names4.filter(i -> i.startsWith("D")).findFirst();
if(firstNameWithD.isPresent()){
    System.out.println("First Name starting with D = " + firstNameWithD.get());
}
```

Terminal Operation: `anyMatch`, `allMatch`, `noneMatch`

- `anyMatch(Predicate predicate)`
 - > Returns whether any element of this stream matches the provided predicate
- `allMatch(Predicate predicate)`
 - > Returns whether all elements of this stream match the provided predicate
- `noneMatch(Predicate predicate)`
 - > Returns whether no elements of this stream match the provided predicate

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 6, 7, 8);
```

```
boolean result = numbers.stream().anyMatch(i -> i==5); // false  
result = numbers.stream().allMatch(i -> i<10);         // true  
result = numbers.stream().noneMatch(i -> i==10);        // true
```

Terminal Operation: reduce

- Performs a reduction on the elements of the stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
Optional<Integer> intOptional = numbers.stream().reduce((i,j) -> {return i*j;});  
if (intOptional.isPresent()) {  
    System.out.println("Multiplication through reduce = " + intOptional.get()); // 120  
}
```




Laziness

Laziness

- Intermediate operator runs only when terminate operator asks

// An example stream without a terminal operator

```
Stream.of("d2", "a2", "b1", "b3", "c")
```

```
    .map(s -> s.toUpperCase())
```

```
    .filter(s -> {System.out.println(s); return s.startsWith("B");}); // no print
```

Laziness Example

- In the example below, `limit(2)` uses short-circuiting; we need to process only part of the stream, not all of it, to return a result - similar to evaluating a large Boolean expression

```
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(Collectors.toList());
```

filtering 1

filtering 2

mapping 2

filtering 3

filtering 4

mapping 4

Observe that
only two items are
processed

Parallelism

Why Parallelism in Java 8 Streams?

- The join/fork framework introduced in Java 7 still requires developers to specify how the problems are subdivided (partitioned)
 - > With aggregate operations in Java 8, it is the Java runtime that performs this partitioning and combining of solutions for you
- Collections are not thread-safe
 - > Aggregate operations and parallel streams enable you to implement parallelism with non-thread-safe collections provided that you do not modify the collection while you are operating on it

Why Parallelism in Java 8 Streams?

- You can execute streams in serial or in parallel
 - > When a stream executes in parallel, the Java runtime partitions the stream into multiple substreams
 - > Aggregate operations iterate over and process these substreams in parallel and then combine the results
- To create a parallel stream
 - > Invoke the operation `Collection.parallelStream`
 - > Invoke the operation `BaseStream.parallel`

double average = roster

.parallelStream()

.filter(p -> p.getGender() == Person.Sex.MALE)

.mapToInt(Person::getAge)

.average()

.getAsDouble();

Code with Passion!

