Streams

What is a Stream?

First of all, streams are not collections.
A simple definition is that streams are wrappers for collections and arrays. They wrap an existing collection (or another data source) to support operations expressed with lambdas, so you specify what you want to do, not how to do it.

Characteristics of streams

- Streams work perfectly with lambdas.
- Streams don't store their elements.
- Streams are immutable.
- Streams are not reusable.
- Streams don't support indexed access to their elements.
- Streams are easily parallelizable.
- *Stream operations are lazy when possible.*

One thing that allows this laziness is the way their operations are designed. Most of them return a new stream, allowing operations to be chained and form a pipeline that enables this kind of optimizations.
To set up this pipeline you:

1. Create the stream.
2. Apply zero or more intermediate operations to transform the initial stream into new streams.
3. Apply a terminal operation to generate a result or a *side-effect*.

Stream Life Cycle

Creating Streams

A stream is represented by the java.util.stream.Stream<T> interface. This works with objects only.

There are also specializations to work with primitive types, such as IntStream, LongStream, and DoubleStream. Also, there are many ways to create a stream. Let's see the three most popular.

The first one is creating a stream from a java.util.Collection implementation using the stream() method:

```
List<String> words = Arrays.asList(new String[]{"hello", "hola", "hallo", "ciao"});
Stream<String> stream = words.stream();
```

The second one is creating a stream from individual values:

```
Stream<String> stream = Stream.of("hello","hola", "hallo", "ciao");
```

The third one is creating a stream from an array:

```
String[] words = {"hello", "hola", "hallo", "ciao"};
Stream<String> stream = Stream.of(words);
```

Intermediate Operations

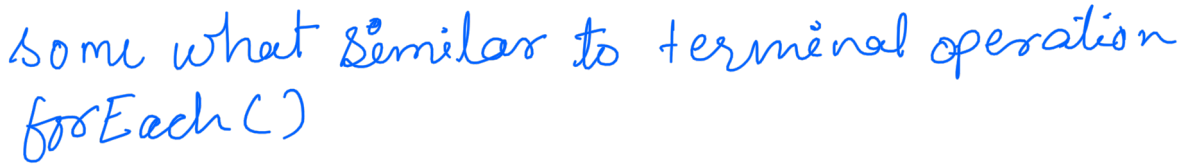
You can easily identify intermediate operations; they always return a new stream. This allows the operations to be connected.

```
Stream<String> s = Stream.of("m", "k", "c", "t")
    .sorted()
    .limit(3)
```

An important feature of intermediate operations is that they don't process the elements until a terminal operation is invoked; in other words, they're lazy.
Intermediate operations are further divided into stateless and stateful operations.

Stateless operations retain no state from previous elements when processing a new element so each can be processed independently of operations on other elements.
Some examples are:

- `Stream<T> filter(Predicate<? super T> predicate)`
  - Returns a stream of elements that match the given predicate.
- `<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`
  - Returns a stream with the content produced by applying the provided mapping function to each element. There are versions for `int`, `long` and `double` also.
- `<R> Stream<R> map(Function<? super T,? extends R> mapper)`
  - Returns a stream consisting of the results of applying the given function to the elements of this stream. There are versions for `int`, `long` and `double` also.
- `Stream<T> peek(Consumer<? super T> action)`



    - Returns a stream with the elements of this stream, performing the provided action on each element.

Stateful operations, such as distinct and sorted, may incorporate state from previously seen elements when processing new elements.
Some examples are:

- `Stream<T> distinct()`. Returns a stream consisting of the distinct elements.
- `Stream<T> limit(long maxSize)`. Returns a stream truncated to be no longer than `maxSize` in length.
- `Stream<T> skip(long n)`. Returns a stream with the remaining elements of this stream after discarding the first `n` elements.
- `Stream<T> sorted()`. Returns a stream sorted according to the natural order of its elements.
- `Stream<T> sorted(Comparator<? super T> comparator)`. Returns a stream with the sorted according to the provided `Comparator`.

Terminal Operations

You can also easily identify terminal operations because they always return something other than a stream.
After the terminal operation is performed, the stream pipeline is consumed and can't be used anymore.
For example:

```
int[] digits = {0, 1, 2, 3, 4 , 5, 6, 7, 8, 9};
IntStream s = IntStream.of(digits);
long n = s.count();
System.out.println(s.findFirst()); // An exception is thrown
```

If you need to traverse the same stream again, you must return to the data source to get a new one. For example:

```
int[] digits = {0, 1, 2, 3, 4 , 5, 6, 7, 8, 9};
long n = IntStream.of(digits).count();
System.out.println(IntStream.of(digits).findFirst()); // OK
```

The following methods represent terminal operations:

- boolean allMatch(Predicate<? super T> predicate)
  - Returns whether all elements of this stream match the provided predicate.
- boolean anyMatch(Predicate<? super T> predicate)
  - Returns whether any elements of this stream match the provided predicate.
- boolean noneMatch(Predicate<? super T> predicate)
  - Returns whether no elements of this stream match the provided predicate.

- Optional<T> findAny() recommended for parallel streams
  - Returns an Optional describing some element of the stream.
- Optional<T> findFirst()
  - Returns an Optional describing the first element of this stream.

Suitable for Sequential stream

```
static void findAnyVSfindFirst(int times) {
   List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);

   System.out.println("findAny: ");
   for (int i = 0; i < times; i++) {
      Optional<Integer> result = list.stream().parallel().filter(num -> num <
4).findAny();
      System.out.print(result.get() + ", ");
   }
   System.out.println();

   System.out.println("findFirst: ");
   for (int i = 0; i < times; i++) {
```

```
      Optional<Integer> result = list.stream().parallel().filter(num -> num <
4).findFirst();
      System.out.print(result.get() + ", ");
   }
   System.out.println();
}

output:
findAny:
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 3, 3, 3, 3,
findFirst:
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

always return same value

- <R,A> R collect(Collector<? super T,A,R> collector)
  - Performs a mutable reduction operation on the elements of this stream using a Collector.
- long count()
  - Returns the count of elements in this stream.
- void forEach(Consumer<? super T> action)
  - Performs an action for each element of this stream.
- void forEachOrdered(Consumer<? super T> action)

```
static void forEachVSforEachOrdered(int times) {
   List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);

   System.out.println("forEach: ");
   for (int i = 0; i < times; i++) {
```

```
      list.stream().parallel().forEach(e -> System.out.print(e + ", "));
      System.out.println();
   }

   System.out.println("forEachOrdered: ");
   for (int i = 0; i < times; i++) {
      list.stream().parallel().forEachOrdered(e -> System.out.print(e + ", "));
      System.out.println();
   }
}
output:
forEach:
3, 1, 4, 5, 2,
3, 2, 1, 4, 5,
forEachOrdered:
1, 2, 3, 4, 5,
1, 2, 3, 4, 5,
```

order changed for parallel streams

    - Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
- `Optional<T> max(Comparator<? super T> comparator)`
    - Returns the maximum element of this stream according to the provided `Comparator`.
- `Optional<T> min(Comparator<? super T> comparator)`
    - Returns the maximum element of this stream according to the provided `Comparator`.
- `T reduce(T identity, BinaryOperator<T> accumulator)`

    - Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
- `Object[] toArray()`
    - Returns an array containing the elements of this stream.
- `<A> A[] toArray(IntFunction<A[]> generator)`
    - Returns an array containing the elements of this stream, using the provided generator function to allocate the returned array.
- `Iterator<T> iterator()` for sequential processing
    - Returns an iterator for the elements of the stream.
- `Spliterator<T> spliterator()` for parallel processing
    - Returns a `Spliterator` for the elements of the stream.

Operations on Collections

Usually, when you have a list, you'd want to iterate over its elements. A common way is to use a for block.
Either with an index:

```
List<String> words = ...
for(int i = 0; i < words.size(); i++) {
    System.out.println(words.get(i));
}
```

Or with an iterator:

```
List<String> words = ...
for(Iterator<String> it = words.iterator(); it.hasNext();) {
    System.out.println(it.next());
}
```

Or with the so-called for-each loop:

```
List<String> words = ...
for(String w : words) {
    System.out.println(w);
}
```

The Stream interface provides a corresponding forEach method:

```
void forEach(Consumer<? super T> action)
```

Since this method doesn't return a stream, it is a terminal operation.

Using it is not different from using the other methods:

```
Stream<String> stream = words.stream();
// As an anonymous class
stream.forEach((new Consumer<String>() {
    public void accept(String t) {
        System.out.println(t);
    }
});
// As a lamba expression
stream.forEach(t -> System.out.println(t));
```

Of course, the advantage of using streams is that you can chain operations.

```
words.sorted()
    .limit(2)
    .forEach(System.out::println);
```

Remember that because this is a terminal operation, you cannot do things like this:

```
words.forEach(t -> System.out.println(t.length()));
words.forEach(System.out::println);
```

If you want to do something like that, either create a new stream each time:

```
Stream.of(wordList).forEach(t -> System.out.println(t.length()));
Stream.of(wordList).forEach(System.out::println);
```

Or wrap the code inside one lambda:

```
Consumer<String> print = t -> {
    System.out.println(t.length());
    System.out.println(t);
};
words.forEach(print);
```

Also, you can't use `return`, `break` or `continue` to terminate an iteration either. `break` and `continue` will generate a compilation error since they cannot be used outside of a loop and `return` doesn't make sense when we see that the `foreach` method is implemented basically as:

```
for (T t : this) {
   // Inside accept, return has no effect
   action.accept(t);
}
```

Another common requirement is to filter (or remove) elements from a collection that don't match a particular condition.

You normally do this either by copying the matching elements to another collection:

```
List<String> words = ...
List<String> nonEmptyWords = new ArrayList<String>();
for(String w : words) {
    if(w != null && !w.isEmpty()) {
        nonEmptyWords.add(w);
    }
}
```

Or by removing the non-matching elements in the collection itself with an iterator (only if the collection supports removal):

```
List<String> words = new ArrayList<String>();
// ... (add some strings)
for (Iterator<String> it = words.iterator(); it.hasNext();) {
    String w = it.next();
    if (w == null || w.isEmpty()) {
        it.remove();
```

```
    }
}
```

For these cases, you can use the filter method of the Stream interface:

```
Stream<T> filter(Predicate<? super T> predicate)
```

That returns a new stream consisting of the elements that satisfy the given predicate. Since this method returns a stream, it represents an intermediate operation, which basically means that you can chain any number of filters or other intermediate operations:

```
List<String> words = Arrays.asList("hello", null, "");
words.stream()
    .filter(t -> t != null) // ["hello", ""]
    .filter(t -> !t.isEmpty()) // ["hello"]
    .forEach(System.out::println);
```

Of course, the result of executing this code is:

```
hello
```

Data Search

Searching is a common operation for when you have a set of data.
The Stream API has two types of operation for searching.
Methods starting with *Find*:

```
Optional<T> findAny()
Optional<T> findFirst()
```

`find` methods search for an element in a stream. Since there's a possibility that an element can't be found (if the stream is empty, for example), `find` methods return an `Optional`.
The other way to search is through methods ending with *Match*:

```
boolean allMatch(Predicate<? super T> predicate)
boolean anyMatch(Predicate<? super T> predicate)
boolean noneMatch(Predicate<? super T> predicate)
```

`match` methods indicate whether a certain element matches the given predicate. They return a boolean.
Since all these methods return a type different than a stream, they are considered terminal operations.
`findAny()` and `findFirst()` practically do the same, they return the first element they find in a stream:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
stream.findFirst()
    .ifPresent(System.out::println); // 1

IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
stream2.findAny()
    .ifPresent(System.out::println); // 1
```

Again, if the stream is empty, these return an empty `Optional`:

```
Stream<String> stream = Stream.empty();
System.out.println(
    stream.findAny().isPresent()
); // false
```

`java.util.Optional<T>` is a new class also introduced in Java 8.
When to use `findAny()` and when to use `findFirst()`?
When working with parallel streams, it's harder to find the first element. In this case, it's better to use `findAny()` if you don't really mind which element is returned.

On the other hand, we have the *Match methods.
anyMatch() returns true if any of the elements in a stream matches the given predicate:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream.anyMatch(i -> i%3 == 0)
); // true
```

If the stream is empty or if there's no matching element, this method returns false:

```
IntStream stream = IntStream.empty();
System.out.println(
    stream.anyMatch(i -> i%3 == 0)
); // false

IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream2.anyMatch(i -> i%10 == 0)
); // false
```

allMatch() returns true only if all elements in the stream match the given predicate:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream.allMatch(i -> i > 0)
); // true

IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream2.allMatch(i -> i%3 == 0)
); // false
```

If the stream is empty, this method returns true without evaluating the predicate:

```
IntStream stream = IntStream.empty();
System.out.println(
    stream.allMatch(i -> i%3 == 0)
); // true
```

noneMatch() is the opposite of allMatch(), it returns true if none of the elements in the stream match the given predicate:

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream.noneMatch(i -> i > 0)
); // false

IntStream stream2 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream2.noneMatch(i -> i%3 == 0)
); // false

IntStream stream3 = IntStream.of(1, 2, 3, 4, 5, 6, 7);
System.out.println(
    stream3.noneMatch(i -> i > 10)
); // true
```

If the stream is empty, this method returns also true without evaluating the predicate:

```
IntStream stream = IntStream.empty();
System.out.println(
    stream.noneMatch(i -> i%3 == 0)
); // true
```

An important thing to consider is that all of these operations use something similar to the short-circuiting of `&&` and `||` operators.

Short-circuiting means that the evaluation stops once a result is found.
Thus `find*` operations stop at the first found element.

With `*Match` operations, however, why would you evaluate all the elements of a stream when, by evaluating the third element (for example), you can know if all or none (again for example) of the elements will match?

Sorting a Stream

Sorting a stream is simple.

```
Stream<T> sorted()
```

The method above returns a stream with the elements sorted according to their natural order. For example:

```
List<Integer> list = Arrays.asList(57, 38, 37, 54, 2);
list.stream()
    .sorted()
    .forEach(System.out::println);
```

Will print:

```
2
37
38
54
57
```

The only requirement is that the elements of the stream implement java.lang.Comparable (that way, they are sorted in natural order). Otherwise, a ClassCastException may be thrown.
If we want to sort using a different order, there's another version of this method that takes a java.util.Comparator (this version is not available for primitive stream like IntStream):

```
Stream<T> sorted(Comparator<? super T> comparator)
```

For example:

```
List<String> strings =
    Arrays.asList("Stream","Operations","on","Collections");
strings.stream()
    .sorted( (s1, s2) -> s2.length() - s1.length() )
    .forEach(System.out::println);
```

This method will print the following on execution:

```
Collections
Operations
Stream
on
```

Data and Calculation Methods

The Stream interface provides the following data and calculation methods:

```
long count()
Optional<T> max(Comparator<? super T> comparator)
Optional<T> min(Comparator<? super T> comparator)
```

The primitive versions of the Stream interface have the following methods:

IntStream

```
OptionalDouble average()
long count()
OptionalInt max()
OptionalInt min()
int sum()
```

LongStream

```
OptionalDouble average()
long count()
OptionalLong max()
OptionalLong min()
long sum()
```

DoubleStream

```
OptionalDouble average()
long count()
OptionalDobule max()
OptionalDouble min()
double sum()
```

count() returns the number of elements in the stream or zero if the stream is empty:

```
List<Integer> list = Arrays.asList(57, 38, 37, 54, 2);
System.out.println(list.stream().count()); // 5
```

min() returns the minimum value in the stream wrapped in an Optional or an empty one if the stream is empty.
max() returns the maximum value in the stream wrapped in an Optional or an empty one if the stream is empty.
When we talk about primitives, it is easy to know which the minimum or maximum value is. But when we are talking about objects (of any kind), Java needs to know how to compare them to know which one is the maximum and the minimum. That's why the Stream interface needs a Comparator for max() and min():

```
List<String> strings =
    Arrays.asList("Stream","Operations","on","Collections");
strings.stream()
    .min( Comparator.comparing(
                (String s) -> s.length())
    ).ifPresent(System.out::println); // on
```

v returns the sum of the elements in the stream or zero if the stream is empty:

```
System.out.println(
    IntStream.of(28,4,91,30).sum()
); // 153
```

average() returns the average of the elements in the stream wrapped in an OptionalDouble or an empty one if the stream is empty:

```
System.out.println(
    IntStream.of(28,4,91,30).average()
); // 38.25
```

Map

`map()` is used to transform the value or the type of the elements of a stream:

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
IntStream mapToInt(ToIntFunction<? super T> mapper)
LongStream mapToLong(ToLongFunction<? super T> mapper)
DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)
```

As you can see, `map()` takes a `Function` to convert the elements of a stream of type `T` to type `R`, returning a stream of that type `R`:

```
Stream.of('a', 'b', 'c', 'd', 'e')
    .map(c -> (int)c)
    .forEach(i -> System.out.format("%d ", i));
```

The output:

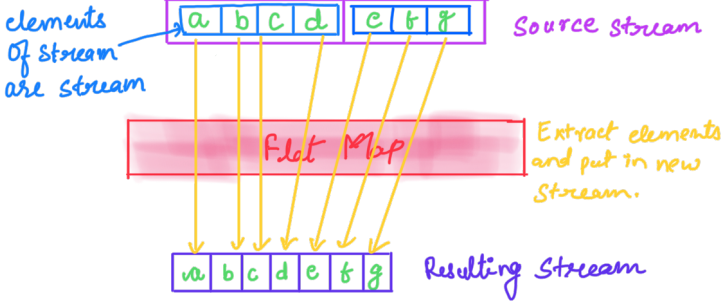
```
97 98 99 100 101
```

There are versions for transforming to primitive types. For example:

```
IntStream.of(100, 110, 120, 130 ,140)
    .mapToDouble(i -> i/3.0)
    .forEach(i -> System.out.format("%.2f ", i));
```

Will output:

```
33.33 36.67 40.00 43.33 46.67
```



FlatMap

flatMap() is used to flatten (or combine) the elements of a stream into one (new) stream:

```
<R> Stream<R> flatMap(Function<? super T,
                      ? extends Stream<? extends R>> mapper)
DoubleStream flatMapToDouble(Function<? super T,
                      ? extends DoubleStream> mapper)
IntStream flatMapToInt(Function<? super T,
                      ? extends IntStream> mapper)
LongStream flatMapToLong(Function<? super T,
                      ? extends LongStream> mapper)
```

From its signature (and the signature of the primitive versions) we can see that, in contrast to map() which returns a single value, flatMap() must return a Stream.
If flatMap() maps to null, the return value will be an empty stream, not null itself.
Let's see how this works. Suppose we have a stream comprising lists of characters:

```
List<Character> aToD = Arrays.asList('a', 'b', 'c', 'd');
List<Character> eToG = Arrays.asList('e', 'f', 'g');
Stream<List<Character>> stream = Stream.of(aToD, eToG);
```

We want to convert all the characters to their int representation. Notice through the code below that we can't use map() anymore; c represents an object of type List<Character>, not Character:

```
stream .map(c -> (int)c)
```

Instead, we need to get the elements of the lists into one stream and then convert each character to an int. Fortunately, we have flatMap() to combine the list elements into a single Stream object:

```
stream
    .flatMap(l -> l.stream())
    .map(c -> (int)c)
    .forEach(i -> System.out.format("%d ", i));
```

This outputs the following:

```
97 98 99 100 101 102 103
```

Function parameter for flatMap() should return a stream whereas *Function parameter for map()* should returns an element.
Using peek() (which just executes the provided expression and returns a new stream with the same elements of the original one) after flatMap() may clarify how the elements are processed:

```
stream
    .flatMap(l -> l.stream())
    .peek(System.out::print)
    .map(c -> (int)c)
```

```
    .forEach(i -> System.out.format("%d ", i));
```

As you can see from the output, the stream returned from `flatMap()` is passed through the pipeline, as if we were working with a stream of single elements rather than a stream of lists of elements:

```
a97 b98 c99 d100 e101 f102 g103
```

This way, with `flatMap()` you can convert a `Stream<List<Object>>` to `Stream<Object>`.

Reduction

A reduction is an operation that takes many elements and combines them to reduce them into a single value or object. Reduction is done by applying an operation multiple times. Some examples of reductions include summing N elements, finding the maximum element of N numbers, or counting elements.
In the following example, we use a for loop to reduce an array of numbers to their sum:

```
int[] numbers = {1, 2, 3, 4, 5, 6};
int sum = 0;
for(int n : numbers) {
    sum += n;
}
```

Of course, making reductions with streams instead of loops has benefits, such as easier parallelization and improved readability.
The Stream interface has two methods for reduction:

```
collect()
reduce()
```

We can implement reductions with both of these methods, but collect() helps us implement a type of reduction called mutable reduction, where a container (like a Collection) is used to accumulate the result of the operation.
The other reduction operation, reduce(), has three versions:

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

```
T reduce(T identity,
         BinaryOperator<T> accumulator)

<U> U reduce(U identity,
             BiFunction<U,? super T,U> accumulator,
             BinaryOperator<U> combiner)
```

Remember that a `BinaryOperator<T>` is equivalent to a `BiFunction<T, T, T>`, where the two arguments and the return type are all of the same types.
Let's start with the version that takes one argument. This is equivalent to:

```
boolean elementsFound = false;
T result = null;
for (T element : stream) {
    if (!elementsFound) {
        elementsFound = true;
        result = element;
    } else {
        result = accumulator.apply(result, element);
    }
return elementsFound ? Optional.of(result)
                     : Optional.empty();
```

This code just applies a function for each element, accumulating the result and returning an Optional wrapping that result, or an empty Optional if there were no elements.
Let's see a concrete example. We just see how a sum is a reduce operation:

```
int[] numbers = {1, 2, 3, 4, 5, 6};
int sum = 0;
for(int n : numbers) {
   sum += n;
}
```

Here, the accumulator operation is:

```
sum += n;   //or sum = sum + n
```

This translates to:

```
OptionalInt total = IntStream.of(1, 2, 3, 4, 5, 6)
                        .reduce( (sum, n) -> sum + n );
```

Notice how the primitive version of Stream uses the primitive version of Optional.
This is what happens step by step:

1. An internal variable that accumulates the result is set to the first element of a stream (1).
2. This accumulator and the second element of the stream (2) are passed as arguments to the BinaryOperator represented by the lambda expression (sum, n) -> sum + x.
3. The result (3) is assigned to the accumulator.
4. The accumulator (3) and the third element of the stream (3) are passed as arguments to the BinaryOperator.

5. The result (6) is assigned to the accumulator.
6. Steps 4 and 5 are repeated for the next elements of the stream until there are no more elements.

However, what if you need to have an initial value? For cases like that, we have the version that takes two arguments:

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

The first argument is the initial value, and it is called the identity because, strictly speaking, this value must be an identity for the accumulator function. In other words, for each value v, accumulator.apply(identity, v) must be equal to v.
This version of reduce() is equivalent to:

```
T result = identity;
for (T element : stream)  {
    result = accumulator.apply(result, element);
}
return result;
```

Notice that this version does not return an Optional object because if the stream empty, the identity value is returned.
For example, the sum example can be rewritten as:

```
int total = IntStream.of(1, 2, 3, 4, 5, 6)
                .reduce( 0,
                        (sum, n) -> sum + n ); // 21
```

Or using a different initial value:

```
int total = IntStream.of(1, 2, 3, 4, 5, 6)
                .reduce( 4,
                        (sum, n) -> sum + n ); // 25
```

However, notice that in the example above, the first value cannot be considered an identity because, for instance, 4 + 1 is not equal to 4.
This can bring some problems when working with parallel streams, which we'll review in a few moments.
Now, notice that with these versions, you take elements of type T and return a reduced value of type T as well.
However, if you want to return a reduced value of a different type, you have to use the three arguments version of reduce():

```
<U> U reduce(U identity,
             BiFunction<U,? super T, U> accumulator,
             BinaryOperator<U> combiner)
```

This is equivalent to using:

```
U result = identity;
for (T element : stream) {
    result = accumulator.apply(result, element)
}
return result;
```

Consider for example that we want to get the sum of the length of all strings of a stream, so we take strings (type T), and we want an integer result (type U).

In that case, we use reduce() like this:

```
int length =
    Stream.of("Parallel", "streams", "are", "great")
        .reduce(0,
                (accumInt, str) ->
                   accumInt + str.length(), //accumulator
                (accumInt1, accumInt2) ->
                   accumInt1 + accumInt2);//combiner
```

We can make it clearer by adding the argument types:

```
int length =
  Stream.of("Parallel", "streams", "are", "great")
      .reduce(0,
            (Integer accumInt, String str) ->
               accumInt + str.length(), //accumulator
            (Integer accumInt1, Integer accumInt2) ->
               accumInt1 + accumInt2);//combiner
```

As the accumulator function adds a mapping (transformation) step to the accumulator function, this version of the reduce() can be written as a combination of map() and the other versions of the reduce() method (you may know this as the MapReduce pattern):

```
int length =
```

```
    Stream.of("Parallel", "streams", "are", "great")
        .mapToInt(s -> s.length())
        .reduce(0,
                (sum, strLength) ->
                    sum + strLength);
```

Or simply:

```
int length = Stream.of("Parallel", "streams", "are", "great")
                .mapToInt(s -> s.length())
                .sum();
```

In fact, the calculation operations that we learned about earlier are implemented as reduce operations under the hood:

```
average
count
max
min
sum
```

Also, notice that if we return a value of the same type, the combiner function is no longer necessary (it turns out that this function is the same as the accumulator function). So, in this case, it's better to use the two argument version.
It's recommended to use the 3rd version `reduce()` method when:

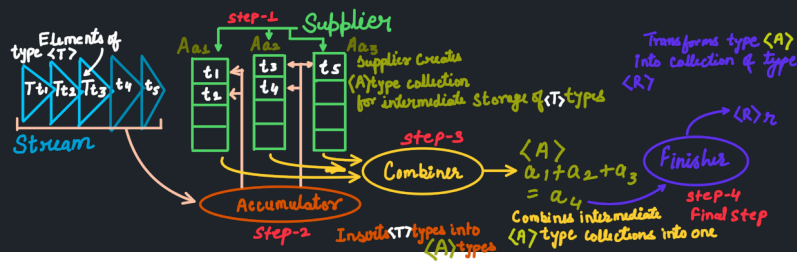
- Working with parallel streams

- Having one function as a mapper and accumulator is more efficient than having separate mapping and reduction functions.

On the other hand, collect() has two versions:

```
<R,A> R collect(Collector<? super T,A,R> collector)

<R> R collect(Supplier<R> supplier,
              BiConsumer<R,? super T> accumulator,
              BiConsumer<R,R> combiner)
```



The first version uses predefined collectors from the Collectors class while the second one allows you to create your own collectors. Primitive streams (like IntStream) only have this last version of collect().
Remember that collect() performs a mutable reduction on the elements of a stream, which means that it uses a mutable object for accumulating, like a Collection or a StringBuilder. In contrast, reduce() combines two elements to produce a new one and represents an immutable reduction.
However, let's start with the version that takes three arguments, as it's similar to the reduce() version that also takes three arguments.
As you can see from its signature, first, it takes a Supplier that returns the object that will be used to returned a container.
The second parameter is an accumulator function, which takes the container and the element to be added to it.
The third parameter is the combiner function, which merges the intermediate results into the final one (useful when working with parallel streams).
This version of collect() is equivalent to:

```
R result = supplier.get();
for (T element : stream) {
    accumulator.accept(result, element);
}
return result;
```

For example, if we want to "reduce" or "collect" all the elements of a stream into a `List`, use the following algorithm:

```
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
          () -> new ArrayList<>(),// Creating the container
           (l, i) -> l.add(i), // Adding an element
           (l1, l2) -> l1.addAll(l2) // Combining elements
        );
```

We can make it clearer by adding the argument types:

```
List<Integer> list =
   Stream.of(1, 2, 3, 4, 5)
        .collect(
          () -> new ArrayList<>(),
           (l, i) -> l.add(i),
```

```
            (l1, l2) -> l1.addAll(l2)
        );
```

We can also use method references:

```
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
            ArrayList::new,
            ArrayList::add,
            ArrayList::addAll
        );
```

Collectors

The previous version of collect() is useful to learn how collectors work, but in practice, it's better to use the other version.
Some common collectors of the Collectors class are:

- toList Accumulates elements into a List.
- toSet Accumulates elements into a Set.
- toCollection Accumulates elements into a Collection implementation.
- toMap Accumulates elements into a Map.
- joining Concatenates elements into a String.
- groupingBy Groups elements of type T in lists according to a classification function, into a map with keys of type K.
- partitioningBy Partitions elements of type T in lists according to a predicate, into a map.

Since calculation methods can be implemented as reductions, the Collectors class also provides them as collectors:

- averagingInt/averagingLong/averagingDouble Methods return the average of the input elements.
- counting Counts the elements of input elements.
- maxBy Returns the maximum element according to a given Comparator.
- minBy Returns the minimum element according to a given Comparator.
- summingInt/summingLong/summingDouble Returns the sum of the input elements.

This way, we can rewrite our previous example:

```
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(
           ArrayList::new,
           ArrayList::add,
           ArrayList::addAll
    );
```

As:

```
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(Collectors.toList()); // [1, 2, 3, 4, 5]
```

Since all these methods are `static`, we can use `static` imports.

```
import static java.util.stream.Collectors.*;
List<Integer> list =
    Stream.of(1, 2, 3, 4, 5)
        .collect(toList()); // [1, 2, 3, 4, 5]
```

If we are working with streams of strings, we can join all the elements into one `String` with:

```
String s = Stream.of("a", "simple", "string")
                .collect(joining()); // "asimplestring"
```

We can also pass a separator:

```
String s = Stream.of("a", "simple", "string")
              .collect(joining(" ")); // " a simple string"
```

And a prefix and a suffix:

```
String s = Stream.of("a", "simple", "string")
             .collect(
                 joining(" ", "This is ", ".")
             ); // "This is a simple string."
```

The calculation methods are easy to use. Except for `counting()`, they either take a `Function` to produce a value to apply the operation or (in the case of `maxBy` and `minBy`) they take a `Comparator` to produce the result:

```
double avg = Stream.of(1, 2, 3)
                .collect(averagingInt(i -> i * 2)); // 4.0

long count = Stream.of(1, 2, 3)
                .collect(counting()); // 3

Stream.of(1, 2, 3)
   .collect(maxBy(Comparator.naturalOrder()))
    .ifPresent(System.out::println); // 3
```

```
Integer sum = Stream.of(1, 2, 3)
    .collect(summingInt(i -> i)); // 6
```

The Collectors class also provides two functions to group the elements of a stream into a list, in a kind of an SQL GROUP BY style.
The first method is groupingBy() and it has three versions. This is the first one:

```
groupingBy(Function<? super T,? extends K> classifier)
```

It takes a Function that classifies elements of type T, groups them into a list and returns the result in a Map where the keys (of type K) are the Function returned values. For example, if we want to group a stream of numbers by the range they belong (tens, twenties, etc.), we can do it with something like this:

```
Map<Integer, List<Integer>> map =
   Stream.of(2, 34, 54, 23, 33, 20, 59, 11, 19, 37)
      .collect( groupingBy (i -> i/10 * 10 ) );
```

The moment you compare this code with the iterative method (with a for loop), you realize the power of streams and collect(). Just look at how many lines of code are used in the traditional implementation.

```
List<Integer> stream =
    Arrays.asList(2,34,54,23,33,20,59,11,19,37);
Map<Integer, List<Integer>> map = new HashMap<>();


for(Integer i : stream) {
```

```
    int key = i/10 * 10;
    List<Integer> list = map.get(key);

    if(list == null) {
        list = new ArrayList<>();
        map.put(key, list);
    }
    list.add(i);
}
```

In the end, both strategies return the same map.

```
{0=[2], 50=[54,59], 20=[23,20], 10=[11,19], 30=[34,33,37]}
```

Downstream Collectors

As you may have noticed, the second version takes a *downstream collector* as an additional argument:

```
groupingBy(Function<? super T,? extends K> classifier,
           Collector<? super T,A,D> downstream)
```

A downstream collector is a collector that is applied to the results of another collector. We can use any collector here, for instance, to count the elements in each group of the previous example:

```
Map<Integer, Long> map =
   Stream.of(2, 34, 54, 23, 33, 20, 59, 11, 19, 37)
         .collect(
             groupingBy(i -> i/10 * 10,
                        counting()
             )
         );
```

Notice how the type of the values of the `Map` change to reflect the type returned by the downstream collector, `counting()`.
This will return the following map:

```
{0=1, 50=2, 20=2, 10=2, 30=3}
```

We can even use another `groupingBy()` to classify the elements in a second level. For instance, instead of counting, we can further classify the elements as even or odd:

```
Map<Integer, Map<String, List<Integer>>> map =
   Stream.of(2,34,54,23,33,20,59,11,19,37)
       .collect(groupingBy(i -> i/10 * 10,
                  groupingBy(i ->
                              i%2 == 0 ? "EVEN" : "ODD")
                )
       );
```

This will return the following map (with a little formatting):

```
{
    0 = {EVEN=[2]},
   50 = {EVEN=[54], ODD=[59]},
   20 = {EVEN=[20], ODD=[23]},
   10 = {ODD=[11, 19]},
   30 = {EVEN=[34], ODD=[33, 37]}
}
```

The key to the high-level map is an `Integer` because the first `groupingBy()` returns a one. The type of the values of the high-level map changed (again) to reflect the type returned by the downstream collector, `groupingBy()`.
In this case, a `String` is returned; this will be the type of the keys of the second-level map, and since we are working with an `Integer` Stream, the values have a type of `List<Integer>`.

Seeing the output of these examples, you may be wondering, is there a way to order the results?
Well, TreeMap is the only implementation of Map that is ordered. Fortunately, the third version of groupingBy() has a Supplier argument that lets us choose the type of the resulting Map:

```
groupingBy(Function<? super T,? extends K> classifier,
           Supplier<M> mapFactory,
           Collector<? super T,A,D> downstream)
```

This way, if we pass an instance of TreeMap:

```
Map<Integer, Map<String, List<Integer>>> map =
    Stream.of(2,34,54,23,33,20,59,11,19,37)
       .collect( groupingBy(i -> i/10 * 10,
                 TreeMap::new,
                  groupingBy(i -> i%2 == 0 ? "EVEN" : "ODD")
                )
       );
```

This will return the following map:

```
{
    0 = {EVEN=[2]},
   10 = {ODD=[11, 19]},
   20 = {EVEN=[20], ODD=[23]},
```

```
    30 = {EVEN=[34], ODD=[33, 37]},
    50 = {EVEN=[54], ODD=[59]}
}
```

partitioningBy()

The second method for grouping is partitioningBy().
The difference with groupingBy() is that partitioningBy() will return a Map with a Boolean as the key type, which means there are only two groups, one for true and one for false.
There are two versions of this method. The first one is:

```
partitioningBy(Predicate<? super T> predicate)
```

It partitions the elements according to a Predicate and organizes them into a Map<Boolean, List<T>>.
For example, if we want to partition a stream of numbers by the ones that are less than 50 and the ones that don't, we can do it this way:

```
Map<Boolean, List<Integer>> map =
    Stream.of(45, 9, 65, 77, 12, 89, 31)
        .collect(partitioningBy(i -> i < 50));
```

This will return the following map:

```
{false=[65, 77, 89], true=[45, 9, 12, 31, 12]}
```

As you can see, because of the Predicate, the map will always have two elements.
And like groupingBy(), this method has a second version that takes a downstream collector.
For example, if we want to remove duplicates, we just have to collect the elements into a Set like this:

```
Map<Boolean, Set<Integer>> map =
    Stream.of(45, 9, 65, 77, 12, 89, 31, 12)
```

```
        .collect(
            partitioningBy(i -> i < 50,
                           toSet()
            )
        );
```

This will produce the following `Map`:

```
{false=[65, 89, 77], true=[9, 12, 45, 31]}
```

However, unlike `groupingBy()`, there's no version that allows us to change the type of the `Map` returned. However, we only need two keys for our groups.

```
Set<Integer> lessThan50 = map.get(true);
Set<Integer> moreThan50 = map.get(false);
```

Parallel Streams

Until now, all the examples have used sequential streams, where each element are processed one by one.
Parallel streams split the stream into multiple parts. Each part is processed by a different thread at the same time (in parallel).
Under the hood, parallel streams use the Fork Join framework.
This means that, by default, the number of threads available to process parallel streams equals the number of available cores in your machine's processor (CPU).

Parallel stream operations

To create a parallel stream just use the `parallel()` method:

```
Stream<String> parallelStream =
                    Stream.of("a","b","c").parallel();
```

To create a parallel stream from a `Collection` use the `parallelStream()`method:

```
List<String> list = Arrays.asList("a","b","c");
Stream<String> parStream = list.parallelStream();
```

You can turn a parallel stream into a sequential one with the `sequential()`method:

```
stream
   .parallel()
      .filter(..)
         .sequential()
            .forEach(...);
```

Check if a stream is parallel with `isParallel()`:

```
stream.parallel().isParallel(); // true
```

And turn an ordered stream into an unordered one (or ensure that the stream is unordered) with `unordered()`;

```
stream
  .parallel()
     .unordered()
         .collect(...);
```

Under the hood

But how do parallel streams work? Let's start with the simplest example:

```
Stream.of("a","b","c","d","e")
    .forEach(System.out::print);
```

Printing a list of elements with a sequential stream will output the expected result:

```
abcde
```

However, when using a parallel stream:

```
Stream.of("a","b","c","d","e")
    .parallel()
    .forEach(System.out::print);
```

The output can be different for each execution:

```
cbade // One execution
cebad // Another execution
cbdea // Yet another execution
```

Going back to the definition of parallel streams, this output starts making sense. The differences in output can be attributed to thread processing; it is possible that a different core is involved with a particular command each time the code is executed. Thus parallel streams are more appropriate for operations where the order of processing doesn't matter and the operations don't need to keep a state (stateless and independent operations).

An example to see this difference is the use of `findFirst()` versus `findAny()`.

Earlier, we mentioned that `findFirst()` method returns the first element of a stream. But since we're using parallel streams, this method has to "know" which element is the first one:

```
long start = System.nanoTime();
String first = Stream.of("a","b","c","d","e")
    .parallel().findFirst().get();
long duration = (System.nanoTime() - start) / 1000000;
System.out.println(
    first + " found in " + duration + " milliseconds");
```

The output:

```
a found in 2.436155 milliseconds
```

Because of that, if the order doesn't matter, it's better to use `findAny()` with parallel streams:

```
long start = System.nanoTime();
String any = Stream.of("a","b","c","d","e")
    .parallel().findAny().get();
long duration = (System.nanoTime() - start) / 1000000;
System.out.println(
    any + " found in " + duration + " milliseconds");
```

The output:

```
c found in 0.063169 milliseconds
```

Since a parallel stream is processed by multiple cores, it's reasonable to believe that it will be processed faster than a sequential stream. But as you can see with `findFirst()`, this is not always the case.

For example:

```
Stream<T> distinct()
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comparator)
Stream<T> limit(long maxSize)
Stream<T> skip(long n)
```

The stateful operations above incorporate state from previously processed elements and usually need to go through the entire stream to produce a result. Thus they work better with sequential streams since they end up looking through the stream anyway.
But don't believe that by first executing the stateful operations in a sequential format and then turning the stream into a parallel one, the performance will be better in all cases. It would be worse to assume that the entire operation may run in parallel, like the following example:

```java
double start = System.nanoTime();
Stream.of("b","d","a","c","e")
    .sorted()
    .filter(s -> {
        System.out.println("Filter:" + s);
        return !"d".equals(s);
    })
    .parallel()
    .map(s -> {
        System.out.println("Map:" + s);
        return s += s;
    })
    .forEach(System.out::println);
double duration = (System.nanoTime() - start) / 1_000_000;
System.out.println(duration + " milliseconds");
```

One might think that the stream is sorted and filtered sequentially, but the output shows something else:

```
Filter:c
Map:c
cc
Filter:a
Map:a
aa
Filter:b
Map:b
bb
Filter:d
Filter:e
Map:e
ee
79.470779 milliseconds
```

Compare this with the output of the sequential version (just comment out `.parallel()`):

```
Filter:a
Map:a
aa
Filter:b
Map:b
bb
Filter:c
Map:c
cc
Filter:d
Filter:e
Map:e
ee
1.554562 milliseconds
```

Clearly, the sequential version performed better; it took 78 milliseconds less. But if we have an independent or stateless operation, and order doesn't matter, such as with counting the number of odd numbers in a large range, the parallel version will perform better:

```
double start = System.nanoTime();
long c = IntStream.rangeClosed(0, 1_000_000_000)
            .parallel()
            .filter(i -> i % 2 == 0)
            .count();
double duration = (System.nanoTime() - start) / 1_000_000;
System.out.println("Got " + c + " in " + duration + " milliseconds");
```

The parallel version output:

```
Got 500000001 in 738.678448 milliseconds
```

The sequential version output:

```
Got 500000001 in 1275.271882 milliseconds
```

In summary, parallel streams don't always perform better than sequential streams when it comes to stateful operations, but they usually perform better when ordering is not an issue and operations are independent and stateless.
This, the fact that parallel streams process results independently, and the idea that the order cannot be guaranteed are the most important things you need to know.

Tips for deciding between sequential and parallel streams

In practice, how do you know when to use sequential or parallel streams for better performance?
Here are some rules:

- For a small set of data, sequential streams are almost always the best choice due to the overhead of the parallelism. Using parallel streams is simply unnecessary.
- Typically avoid using parallel streams with stateful (like `sorted()`) and order-based (like `findFirst()`) operations. Sequential streams do just fine (if not better) in these cases.
- Use parallel streams with operations that are computationally expensive (considering all the operation in the pipeline).
- When in doubt, check the performance with an appropriate benchmark. To demonstrate, I used an execution time comparison, but this is just one benchmark. You may need your program to compile faster or use less memory.

Reducing Parallel Streams

In concurrent environments, assignments are bad.
This is because if you mutate the state of variables (especially if they are shared by more than one thread), you may run into invalid states.
Consider this example, which implements the factorial of 10 in a very particular way:

```
class Total {
    public long total = 1;
    public void multiply(long n) { total *= n; }
}
...
Total t = new Total();
LongStream.rangeClosed(1, 10)
    .forEach(t::multiply);
System.out.println(t.total);
```

Here, we are using a variable to gather the result of the factorial. The output of executing this snippet of code is:

```
3628800
```

However, when we turn the stream into a parallel one:

```
LongStream.rangeClosed(1, 10)
    .parallel()
    .forEach(t::multiply);
```

Sometimes we get the correct result and other times we don't.
The problem is caused by the multiple threads accessing the variable total concurrently.
Yes, we can synchronize the access to this variable, but that defeats the purpose of parallelism.
Here's where reduce() comes in handy.
Remember that reduce() combines the elements of a stream into a single one.
With parallel streams, this method creates intermediate values and then combines them, avoiding the ordering problem while still allowing streams to be processed in parallel by eliminating the shared state and keeping it inside the reduction process.
The only requirement is that the applied reducing operation must be associative.
This means that the operation op must follow this equality:

```
(a op b) op c == a op (b op c)
```

Or:

```
a op b op c op d == (a op b) op (c op d)
```

So we can evaluate (a op b) and (c op d) in parallel.
We can implement our example using parallel() and reduce() in this way:

```
long tot = LongStream.rangeClosed(1, 10)
                .parallel()
                .reduce(1, (a,b) -> a*b);
System.out.println(tot);
```

When we execute this snippet of code, it produces the correct result every time (3628800).
Reduce guaranteed that the threads would not access the same stream entries simultaneously and throw off the results.

Plus, if we time the execution of the first snippet and this last one, we can see a drastic improvement in performance.
We can safely use collect() with parallel streams if we follow the same requirements of associativity and identity. (For example, combining any partially accumulated result with an empty result container must produce an equivalent result.)
Or, if we are grouping with the Collectors class and ordering is not important, we can use the method groupingByConcurrent(), the concurrent version of groupingBy().
If you understand when to use parallel streams and the issues associated with concurrent execution, you should be ready to use parallel streams in practice.

Q&A

for further queries reachable at
ksingh33@bofa.com