# Theory: Reduction methods



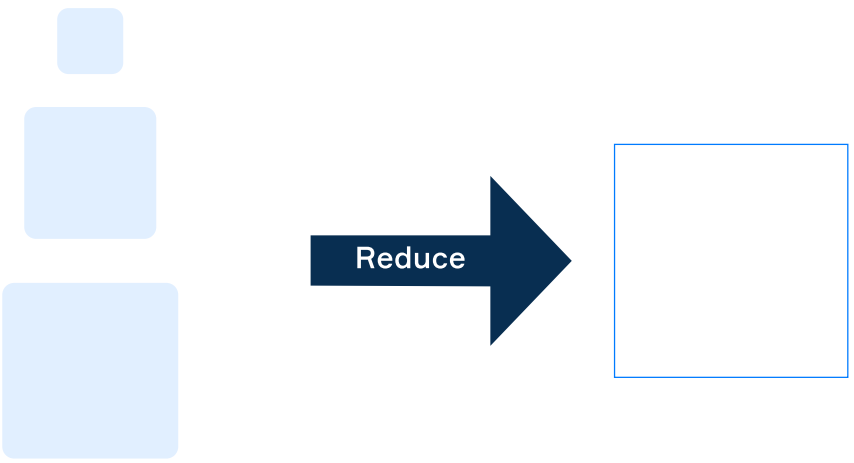
24 minutes  0 / 1 problems solved



While dealing with collections in Java, you can often face the challenge of reducing all collection elements to a single result. The example of such a problem is finding a bank account with the maximum amount of money among the collection of accounts or reducing account transaction values to the total amount of transferred money. Java Stream API provides several *terminal* operations that allow us to solve this problem in a functional way. These operations combine stream elements and return a single value.

## §1. The reduce operation

`reduce` is a method of a `Stream` class that combines elements of a stream into a *single value*. The result can be a value of a primitive type or a complex object.




Note that `reduce` is a terminal operation, which means that it begins all evaluations with the stream and produces a final result.

Let's consider two of the most common uses of the Java Stream API `reduce` operation:

1) In the simplest case, the `reduce` method accepts a two-argument **accumulator** function. The first argument of the accumulator is a partial result of the reduction, while the second one is the next element of a stream. The accumulator should return a reduction value that will be assigned to the partial result. Let's consider the following list of transactional values:

```
1  List<Integer> transactions = List.of(20, 40, -60, 5);
```

Now we sum up all transaction values to get the total amount of transferred money by using `reduce` operation:

```
1
transactions.stream().reduce((sum, transaction) -> sum + transaction)
;
```

At the first iteration of the reduction, the `sum` argument equals to the first element of the stream whose value is 20. The `transaction` argument represents the next element of the stream whose value is 40. After the first iteration, the `sum` accumulates the current value of the `transaction` argument and will be equal to 20 + 40 = 60. In the table below you can see the values of `sum` and `transaction` arguments on each iteration:

1 required topic

✓ Functional data processing with streams

2 dependent topics

Collectors

Stream pipelines

Table of contents:

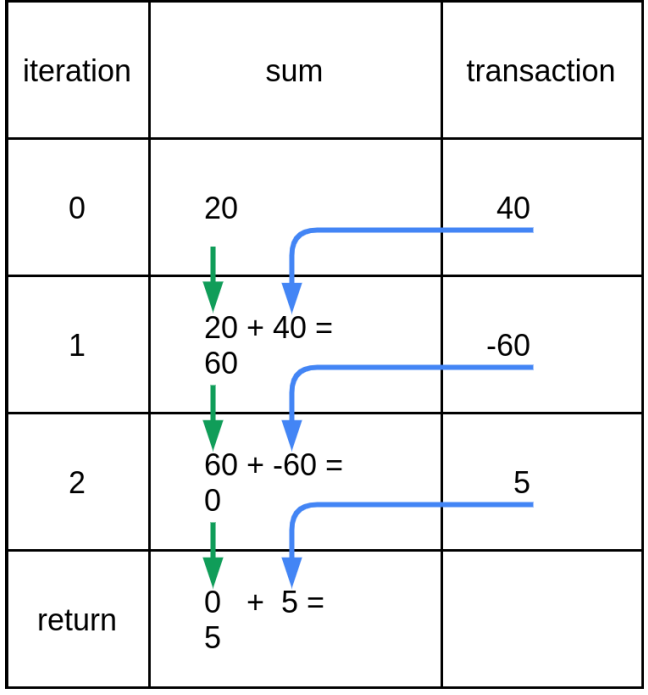
↑ Reduction methods

§1. The reduce operation

§2. Other reduction operations

§3. Conclusion

Discussion




The `reduce` operation that accepts the accumulator function returns `Optional` type. In the example above the `reduce` method returns a container `Optional<Integer>` that contains an `Integer` value of 5.

2) Another `reduce` implementation has one additional parameter: identity value or seed. The **identity** value represents the initial value for the reduction operation.

Let's rewrite our previous example using `identity` argument:

```
transactions.stream().reduce(0, (sum, transaction) -> sum + transacti
on);
```

Now, the initial value of the partial result `sum` is 0 and the initial value of the `transaction` element is 20.

Note, that if a `reduce` method accepts both identity value and accumulator function, it will return a primitive type or an object, but not an `Optional` container. If the stream is empty, the `reduce` operation will return identity value.

## §2. Other reduction operations

While the `reduce` is a generally purposed operation, Stream API provides many specific reduction operations such as `sum`, `min`, `max`, etc. Similar to the `reduce` operation they are terminal operations that produce a single value. Let's consider an example with a generic stream, assuming that we need to find the maximum value from the list of given balances. We can do it by using reduce operation:

```
transactions.stream().reduce((t1, t2) -> t2 > t1 ? t2 : t1)
```

The code above compares a partial result `t1` with the next element of the stream `t2` and assigns the value of `t2` to `t1` if `t2` is numerically greater than `t1`. We can get the same result in a more elegant way using the `max` method:

```
transactions.stream().max(Integer::compareTo);
```

As you may notice the method `max` of the `Stream<T>` class accepts a comparison function in order to find a maximum element among stream elements since we need to specify how exactly to compare generic type. Unlike the generic streams, primitive-specialized streams such as `IntStream` or `LongStream` already "know" the type of their elements. That is why their `max` and `min` functions do not require any comparison function:

```
IntStream.of(20, 40, -60, 5).max();
```

Because of type awareness, primitive-specialized streams have numerically specialized functions such as `average` and `sum`, which cannot be provided by generic streams.

## §3. Conclusion

The `reduce` is a terminal operation that produces a single value result by combining stream elements. It can accept the accumulator function only or in combination with identity value. The accumulator function takes two arguments: partial result and the next element of a stream. If the identity value is provided, the initial value of the partial result will be equal to that value. Otherwise, the partial result's initial value will be equal to the first element of a stream. Besides the `reduce`, primitive-specialized streams provide more reduction operations that depend on the numeric type.