# Guide to Stream.reduce()



Last updated: January 8, 2024

Written by: Alejandro Ugarte (https://www.baeldung.com/author/alejandro-ugarte)

Reviewed by: Kevin Gilmore (https://www.baeldung.com/editor/kevin-author)

**Java Streams (https://www.baeldung.com/category/java/java-streams)**

**JMH (https://www.baeldung.com/tag/jmh)**

## 1. Overview

The Stream API (/java-8-streams-introduction) provides a rich repertoire of intermediate, reduction and terminal functions, which also support parallelization.

More specifically, **reduction stream operations allow us to produce one single result from a sequence of elements**, by repeatedly applying a combining operation to the elements in the sequence.

In this tutorial, **we'll look at the general-purpose *Stream.reduce()* (https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html) operation** and see it in some concrete use cases.

## Further reading:

### Summing Numbers with Java Streams (/java-stream-sum)

A quick and practical guide to summing numbers with Java Stream API.

**Read more (/java-stream-sum) →**

### Introduction to Java Streams (/java-8-streams-introduction)

A quick and practical introduction to Java 8 Streams.

**Read more (/java-8-streams-introduction) →**

### Guide to Java BiFunction Interface (/java-bifunction-interface)

Learn some common patterns for Java functional interfaces that take two parameters.

**Read more (/java-bifunction-interface) →**

# 2. The Key Concepts: Identity, Accumulator and Combiner

Before we look deeper into using the *Stream.reduce()* operation, let's break down the operation's participant elements into separate blocks. That way, we'll understand more easily the role that each one plays.

- *Identity* – an element that is the initial value of the reduction operation and the default result if the stream is empty
- *Accumulator* – a function that takes two parameters: a partial result of the reduction operation and the next element of the stream

- *Combiner* – a function used to combine the partial result of the reduction operation when the reduction is parallelized or when there's a mismatch between the types of the accumulator arguments and the types of the accumulator implementation

## 3. Using *Stream.reduce()*

To better understand the functionality of the identity, accumulator and combiner elements, let's look at some basic examples:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
int result = numbers
  .stream()
  .reduce(0, (subtotal, element) -> subtotal + element);
assertThat(result).isEqualTo(21);
```

In this case, **the *Integer* value 0 is the identity.** It stores the initial value of the reduction operation and also the default result when the stream of *Integer* values is empty.

Likewise, **the lambda expression**:

```
subtotal, element -> subtotal + element
```

**is the accumulator** since it takes the partial sum of *Integer* values and the next element in the stream.

To make the code even more concise, we can use a method reference instead of a lambda expression:

```
int result = numbers.stream().reduce(0, Integer::sum);
assertThat(result).isEqualTo(21);
```

Of course, we can use a *reduce()* operation on streams holding other types of elements.

For instance, we can use *reduce()* on an array of *String* elements and join them into a single result:

```
List<String> letters = Arrays.asList("a", "b", "c", "d", "e");
String result = letters
  .stream()
  .reduce("", (partialString, element) -> partialString + element);
assertThat(result).isEqualTo("abcde");
```

Similarly, we can switch to the version that uses a method reference:

```
String result = letters.stream().reduce("", String::concat);
assertThat(result).isEqualTo("abcde");
```

Let's use the *reduce()* operation for joining the uppercase elements of the *letters* array:

```
String result = letters
  .stream()
  .reduce(
    "", (partialString, element) -> partialString.toUpperCase() +
element.toUpperCase());
assertThat(result).isEqualTo("ABCDE");
```

In addition, we can use *reduce()* in a parallelized stream (more on this later):

```
List<Integer> ages = Arrays.asList(25, 30, 45, 28, 32);
int computedAges = ages.parallelStream().reduce(0, (a, b) -> a + b,
Integer::sum);
```

When a stream executes in parallel, the Java runtime splits the stream into multiple substreams. In such cases, **we need to use a function to combine the results of the substreams into a single one. This is the role of the combiner** — in the above snippet, it's the *Integer::sum* method reference.

Funnily enough, this code won't compile:

```
List<User> users = Arrays.asList(new User("John", 30), new User("Julie", 35));
int computedAges =
  users.stream().reduce(0, (partialAgeResult, user) -> partialAgeResult +
user.getAge());
```

In this case, we have a stream of *User* objects, and the types of the accumulator arguments are *Integer* and *User*. However, the accumulator implementation is a sum of *Integers*, so the compiler just can't infer the type of the *user* parameter.

We can fix this issue by using a combiner:

```
int result = users.stream()
  .reduce(0, (partialAgeResult, user) -> partialAgeResult + user.getAge(),
Integer::sum);
assertThat(result).isEqualTo(65);
```

**To put it simply, if we use sequential streams and the types of the accumulator arguments and the types of its implementation match, we don't need to use a combiner.**

## 4. Reducing in Parallel

As we learned before, we can use *reduce()* on parallelized streams.

When we use parallelized streams, we should make sure that *reduce()* or any other aggregate operations executed on the streams are:

- associative: the result is not affected by the order of the operands
- non-interfering: the operation doesn't affect the data source
- stateless and deterministic: the operation doesn't have state and produces the same output for a given input

We should fulfill all these conditions to prevent unpredictable results.

As expected, operations performed on parallelized streams, including *reduce()*, are executed in parallel, hence taking advantage of multi-core hardware architectures.

For obvious reasons, **parallelized streams are much more performant than the sequential counterparts.** Even so, they can be overkill if the operations applied to the stream aren't expensive, or the number of elements in the stream is small.

Of course, parallelized streams are the right way to go when we need to work with large streams and perform expensive aggregate operations.

Let's create a simple JMH (/java-microbenchmark-harness) (the Java Microbenchmark Harness) benchmark test and compare the respective execution times when using the *reduce()* operation on a sequential and a parallelized stream:

```
@State(Scope.Thread)
private final List<User> userList = createUsers();

@Benchmark
public Integer executeReduceOnParallelizedStream() {
    return this.userList
      .parallelStream()
      .reduce(
        0, (partialAgeResult, user) -> partialAgeResult + user.getAge(),
Integer::sum);
}

@Benchmark
public Integer executeReduceOnSequentialStream() {
    return this.userList
      .stream()
      .reduce(
        0, (partialAgeResult, user) -> partialAgeResult + user.getAge(),
Integer::sum);
}
```

**In the above JMH benchmark, we compare execution average times.** We simply create a *List* containing a large number of *User* objects. Next, we call *reduce()* on a sequential and a parallelized stream and check that the latter performs faster than

the former (in seconds per operation).

These are our benchmark results:

```
Benchmark                                                    Mode  Cnt  Score
Error  Units
JMHStreamReduceBenchMark.executeReduceOnParallelizedStream  avgt    5  0,007 ±
0,001   s/op
JMHStreamReduceBenchMark.executeReduceOnSequentialStream    avgt    5  0,010 ±
0,001   s/op
```

## 5. Throwing and Handling Exceptions While Reducing

In the above examples, the *reduce()* operation doesn't throw any exceptions. But it might, of course.

For instance, say that we need to divide all the elements of a stream by a supplied factor and then sum them:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
int divider = 2;
int result = numbers.stream().reduce(0, a / divider + b / divider);
```

This will work, as long as the *divider* variable is not zero. But if it is zero, *reduce()* will throw an *ArithmeticException* (https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/ArithmeticException.html) exception: divide by zero.

We can easily catch the exception and do something useful with it, such as logging it, recovering from it and so forth, depending on the use case, by using a try/catch (/java-exceptions) block:

```
public static int divideListElements(List<Integer> values, int divider) {
    return values.stream()
      .reduce(0, (a, b) -> {
          try {
              return a / divider + b / divider;
          } catch (ArithmeticException e) {
              LOGGER.log(Level.INFO, "Arithmetic Exception: Division by Zero");
          }
          return 0;
      });
}
```

While this approach will work, **we polluted the lambda expression with the *try/catch* block**. We no longer have the clean one-liner that we had before.

To fix this issue, we can use the extract function refactoring technique (https://refactoring.com/catalog/extractFunction.html) **and extract the *try/catch* block into a separate method**:

```
private static int divide(int value, int factor) {
    int result = 0;
    try {
        result = value / factor;
    } catch (ArithmeticException e) {
        LOGGER.log(Level.INFO, "Arithmetic Exception: Division by Zero");
    }
    return result
}
```

Now the implementation of the *divideListElements()* method is again clean and streamlined:

```
public static int divideListElements(List<Integer> values, int divider) {
    return values.stream().reduce(0, (a, b) -> divide(a, divider) + divide(b,
divider));
}
```

Assuming that *divideListElements()* is a utility method implemented by an abstract *NumberUtils* class, we can create a unit test to check the behavior of the *divideListElements()* method:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
assertThat(NumberUtils.divideListElements(numbers, 1)).isEqualTo(21);
```

Let's also test the *divideListElements()* method when the supplied *List* of *Integer* values contains a 0:

```
List<Integer> numbers = Arrays.asList(0, 1, 2, 3, 4, 5, 6);
assertThat(NumberUtils.divideListElements(numbers, 1)).isEqualTo(21);
```

Finally, let's test the method implementation when the divider is 0 too:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
assertThat(NumberUtils.divideListElements(numbers, 0)).isEqualTo(0);
```

## 6. Complex Custom Objects

We **can also use *Stream.reduce()* with custom objects that contain non-primitive fields.** To do so, we need to provide a relevant i*dentity*, *accumulator* and *combiner* for the data type.

Suppose our *User* is part of a review website. Each of our *User*s can possess one *Rating*, which is averaged over many *Review*s.

First, let's start with our *Review* object.

Each *Review* should contain a simple comment and score:

```
public class Review {

    private int points;
    private String review;

    // constructor, getters and setters
}
```

Next, we need to define our *Rating,* which will hold our reviews alongside a *points* field. As we add more reviews, this field will increase or decrease accordingly:

```
public class Rating {

    double points;
    List<Review> reviews = new ArrayList<>();

    public void add(Review review) {
        reviews.add(review);
        computeRating();
    }

    private double computeRating() {
        double totalPoints =
          reviews.stream().map(Review::getPoints).reduce(0, Integer::sum);
        this.points = totalPoints / reviews.size();
        return this.points;
    }

    public static Rating average(Rating r1, Rating r2) {
        Rating combined = new Rating();
        combined.reviews = new ArrayList<>(r1.reviews);
        combined.reviews.addAll(r2.reviews);
        combined.computeRating();
        return combined;
    }

}
```

We have also added an *average* function to compute an average based on the two input *Ratings*. This will work nicely for our *combiner* and *accumulator* components.

Next, let's define a list of *Users*, each with their own sets of reviews:

```
User john = new User("John", 30);
john.getRating().add(new Review(5, ""));
john.getRating().add(new Review(3, "not bad"));
User julie = new User("Julie", 35);
john.getRating().add(new Review(4, "great!"));
john.getRating().add(new Review(2, "terrible experience"));
john.getRating().add(new Review(4, ""));
List<User> users = Arrays.asList(john, julie);
```

Now that John and Julie are accounted for, let's use *Stream.reduce()* to compute an average rating across both users.

**As an *identity*, let's return a new *Rating* if our input list is empty**:

```
Rating averageRating = users.stream()
  .reduce(new Rating(),
    (rating, user) -> Rating.average(rating, user.getRating()),
    Rating::average);
```

If we do the math, we should find that the average score is 3.6:

```
assertThat(averageRating.getPoints()).isEqualTo(3.6);
```

# 7. Conclusion

In this article, **we learned how to use the *Stream.reduce()* operation.**

**In addition, we learned how to perform reductions on sequential and parallelized streams and how to handle exceptions while reducing.**

As usual, all the code samples shown in this tutorial are available over on GitHub (https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-streams-simple).

Since its introduction in Java 8, the Stream API has become a staple of Java development. The basic operations like iterating, filtering, mapping sequences of elements are deceptively simple to use.

But these can also be overused and fall into some common pitfalls.

To **get a better understanding on how Streams work** and how to combine them with other language features, check out our guide to Java Streams:

**>> Download the E-book (/eBook-Java-Streams-NPI-XXX8e)**

2 COMMENTS

Oldest

View Comments

## COURSES

ALL COURSES (/COURSES/ALL-COURSES)
ALL BULK COURSES (/COURSES/ALL-BULK-COURSES)
ALL BULK TEAM COURSES (/COURSES/ALL-BULK-TEAM-COURSES)
THE COURSES PLATFORM (HTTPS://COURSES.BAELDUNG.COM)

## SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
APACHE HTTPCLIENT TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)
SPRING REACTIVE TUTORIALS (/SPRING-REACTIVE-GUIDE)

## ABOUT

ABOUT BAELDUNG (/ABOUT)
THE FULL ARCHIVE (/FULL_ARCHIVE)

EDITORS (/EDITORS)

(/)

OUR PARTNERS (/PARTNERS/)

PARTNER WITH BAELDUNG (/PARTNERS/WORK-WITH-US)

EBOOKS (/LIBRARY/)

FAQ (HTTPS://WWW.BAELDUNG.COM/LIBRARY/FAQ)

BAELDUNG PRO (/MEMBERS/)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)