Mutable Reduction Through reduce()

In this lesson, yo will learn about mutable reduction using the reduce() method.

We'll cover the following

Introduction to reduction operations

Reduction stream operations are those operations which reduce the stream into a single value. The operations that we are going to discuss in this lesson are immutable operations because they reduce the result into a single-valued immutable variable. Given a collection of objects, we may need to get the sum of all the elements, the max element, or any other operation which gives us a single value as a result. This can be achieved through **reduction operations**.

Before we discuss all the reduction operations in detail, let's first look at some key concepts of reduction:

1. **Identity** – an element that is the initial value of the reduction operation and the default result if the stream is empty.
2. **Accumulator** – a function that takes two parameters: a partial result of the reduction operation and the next element of the stream.
3. **Combiner** – a function used to combine the partial result of the reduction operation when
    - the reduction is parallelized.
    - or there's a mismatch between the types of the accumulator arguments and the types of the accumulator implementation.

and the types of the accumulator implementation.

Now, let's look at some of the reduction methods.

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

As we can see, this method takes a binary operator as an input and returns an `Optional` that describes the reduced value.

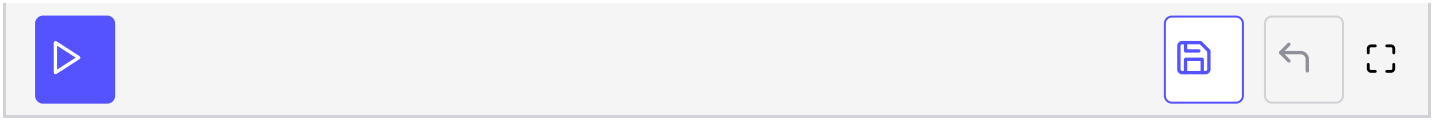
The `reduce()` method iteratively applies the accumulator function on the current input element.

In the below example, we need to find the total salaries of all the employees in an organization.

For this, we are going to use the `reduce(BinaryOperator<T> accumulator)` operation.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

public class StreamDemo {

    public static void main(String[] args) {
        List<Employee> list = new ArrayList<>();
        list.add(new Employee("Dave", 23,20000));
        list.add(new Employee("Joe", 18,40000));
        list.add(new Employee("Ryan", 54,100000));
        list.add(new Employee("Iyan", 5,34000));
        list.add(new Employee("Ray", 63,54000));

         Optional<Integer> totalSalary = list.stream()
                 .map(p -> p.getSalary())  //We are converting the Stream of Employees to Stream o
                 .reduce((p,q) -> p + q);

         if(totalSalary.isPresent()){
             System.out.println("The total salary is " + totalSalary.get());
         }
    }
}

class Employee {
    String name;
    int age;
    int salary;

    Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }
```

```
    public int getAge() {
        return age;
    }

    public int getSalary() {
        return salary;
    }

    @Override
    public String toString() {
        return "Employee{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", salary=" + salary +
                '}';
    }
}
```



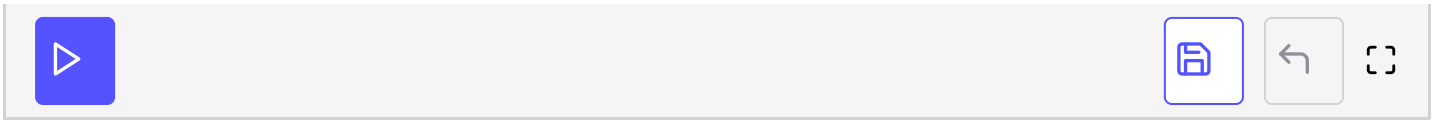
In the above example, we could have used a `sum()` operation instead of `reduce()`, but the `sum()` operation is available in `IntStream`.

So, if we need to get the sum of all the elements in our stream, we should convert it into `IntStream` and then directly use `sum()`.

Below is an example.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

public class StreamDemo {

    public static void main(String[] args) {
        List<Employee> list = new ArrayList<>();
        list.add(new Employee("Dave", 23, 20000));
        list.add(new Employee("Joe", 18, 40000));
        list.add(new Employee("Ryan", 54, 100000));
        list.add(new Employee("Iyan", 5, 34000));
        list.add(new Employee("Ray", 63, 54000));

        int totalSalary = list.stream()
                .mapToInt(p -> p.getSalary())
                .sum();

        System.out.println("The total salary is " + totalSalary);
    }

}

class Employee {
    String name;
    int age;
```

```
    int salary;

    Employee(String name, int age, int salary) {

        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public int getSalary() {
        return salary;
    }

    @Override
    public String toString() {
        return "Employee{" +
                "name='" + name + '\'' +
                ", age=" + age +
                ", salary=" + salary +
                '}';
    }
}
```



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

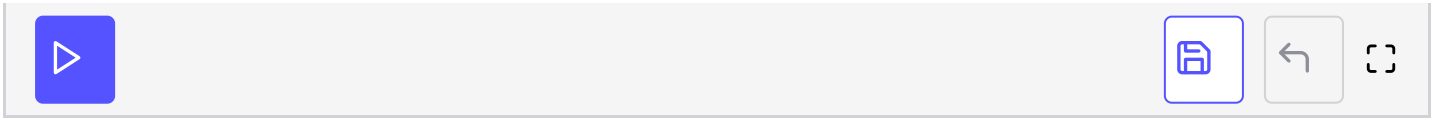
As per Java docs, this method *“performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.”*

This method has an extra ‘identity’ parameter. It is the initial value of reduction. It is the default result of reduction if there are no elements in the stream. That’s the reason, this version of the reduce method doesn’t return Optional because it would at least return the identity element.

In the below example, we provide **five** as an identity. If the stream is empty, five will be returned. If the stream is not empty, five will be added to the sum.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
```

```
public class StreamDemo {

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        list.add(6);

        int totalSum = list.stream()
                .reduce(5, (partialSum, num) -> partialSum + num);

        System.out.println("Total Sum is " + totalSum);
    }

}
```



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

As per Java Docs, this method *performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions.* If we are using a parallel stream, then 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 done by a combiner.

We will use a parallel stream in the example shown above to see how a combiner works.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

public class StreamDemo {

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        list.add(6);

        int totalSum = list.parallelStream()
```

```
        int totalSum = list.parallelStream()
                .reduce(0, (partialSum, num) -> partialSum + num, Integer::sum);

        System.out.println("Total Sum is " + totalSum);
    }
}
```




4. max() and min()

`max()` and `min()` operations are very helpful if we need to get the largest or smallest element from a stream.

Here is the syntax of `max()` operation

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

It takes a `Comparator` as a parameter and returns an `Optional`. Let's see an example.

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Optional;

public class StreamDemo {

    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        list.add(6);

        Optional<Integer> max = list.stream()
                .max(Comparator.naturalOrder());

        System.out.println("Max value is " + max.get());

        Optional<Integer> min = list.stream()
                .min(Comparator.naturalOrder());

        System.out.println("Min value is " + min.get());
    }
}
```




In the above example, we have a stream of integers. Therefore, we used a `Comparator` which sorts the integers according to the natural order.

If the stream is of a custom object, you can provide a custom comparator as well.

In the next lesson, you will learn about slicing operations in streams.