Collectors: Collection Operations.

This lesson discusses the immutable reduction operations tusing the collect() method.

We'll cover the following

- Mutable reductions
- Collectors
  - 1. Collectors.toList()
  - 2. Collectors.toSet()
  - 3. Collectors.toCollection(Supplier collectionFactory)
  - 4. Collectors.toMap()
  - 5. collectingAndThen(Collector downstream, Function finisher)

In the previous lesson, we discussed some immutable reduction methods. In this lesson, we will discuss the mutable reduction methods.

Mutable reductions

The mutable reductions collect the desired results into a mutable container object, such as a `java.util.Collection` or an array.

The mutable reduction is achieved through the `collect()` method. It is one of the Java 8 Stream API's terminal methods.

There are two overloaded versions of the `collect()` method:

1. `collect(Collector<? super T,A,R> collector)`
2. `<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)`

This lesson focuses on the `collect()` method which takes an instance of `Collector` as input.

We have two options:

1. We can create our own `Collector` implementation.

2. We can use the predefined implementations provided by the `Collectors` class.

Before discussing the `collect()` method further, we will first discuss the `Collectors` class in detail and look at how its methods are used with the `collect()` method to reduce streams.

`Collectors`

`Collectors` is a final class that extends the `Object` class. It provides the most common mutable reduction operations that could be required by application developers as individual static methods.

Some of the important reduction operations already implemented in the `Collectors` class are listed below:

Method	Purpose
`toList()`	Collects stream elements in a List.
`toSet()`	Collects stream elements in a Set.
`toMap()`	Returns a `Collector` that accumulates elements into a `Map` whose keys and values are the result of applying the provided mapping functions to the input elements.
`collectingAndThen()`	Collects stream elements and then transforms them using a `Function`
`summingDouble()`, `summingLong()`, `summingInt()`	Sums-up stream elements after mapping them to a `Double`/`Long`/`Integer` value using specific type `Function`
`reducing()`	Reduces elements of stream based on

`reducing()`	the `BinaryOperator` function provided
`partitioningBy()`	Partitions stream elements into a `Map` based on the `Predicate` provided
`counting()`	Counts the number of stream elements
`groupingBy()`	Produces `Map` of elements grouped by grouping criteria provided
`mapping()`	Applyies a mapping operation to all stream elements being collected
`joining()`	For concatenation of stream elements into a single `String`
`minBy()/maxBy()`	Finds the minimum/maximum of all stream elements based on the `Comparator` provided

Let's look at these methods and discuss how they work.

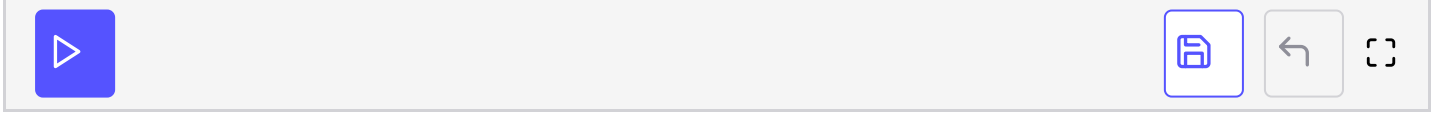
1. `Collectors.toList()`

It returns a `Collector` that collects all of the input elements into a new `List`.

Suppose we need to get a list of employee names. We can use the `toList()` method.

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

public class CollectorsDemo {

    public static void main(String args[]){
        List<Employee> employeeList = new ArrayList<>();
        employeeList.add(new Employee("Alex" , 23, 23000, "USA"));
        employeeList.add(new Employee("Ben" , 63, 25000, "India"));
```

```
        employeeList.add(new Employee("Dave" , 34, 56000, "Bhutan"));
        employeeList.add(new Employee("Jodi" , 43, 67000, "China"));
        employeeList.add(new Employee("Ryan" , 53, 54000, "Libya"));


        List<String> empName = employeeList.stream()
                .map(emp -> emp.getName())
                .collect(Collectors.toList());

        System.out.println(empName);
    }
}

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

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

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public Integer getSalary() {
        return salary;
    }

    public String getCountry() {
        return country;
    }

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



2. `Collectors.toSet()`

It returns a `Collector` that collects all input elements into a new `Set`.

Suppose we have a list of employees, and we need to get a set of countries to which

our employees belong then we can use `toSet()` method.

```
import java.util.ArrayList;
import java.util.Set;
import java.util.List;
import java.util.stream.Collectors;

public class CollectorsDemo {

    public static void main(String args[]){
        List<Employee> employeeList = new ArrayList<>();
        employeeList.add(new Employee("Alex" , 23, 23000, "USA"));
        employeeList.add(new Employee("Ben" , 63, 25000, "India"));
        employeeList.add(new Employee("Dave" , 34, 56000, "Bhutan"));
        employeeList.add(new Employee("Jodi" , 43, 67000, "China"));
        employeeList.add(new Employee("Ryan" , 53, 54000, "Libya"));

        Set<String> empName = employeeList.stream()
                .map(emp -> emp.getCountry())
                .collect(Collectors.toSet());

        System.out.println(empName);
    }
}

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

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

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public Integer getSalary() {
        return salary;
    }

    public String getCountry() {
        return country;
    }

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

```
                '}';
    }
}
```



3. `Collectors.toCollection(Supplier<C> collectionFactory)`

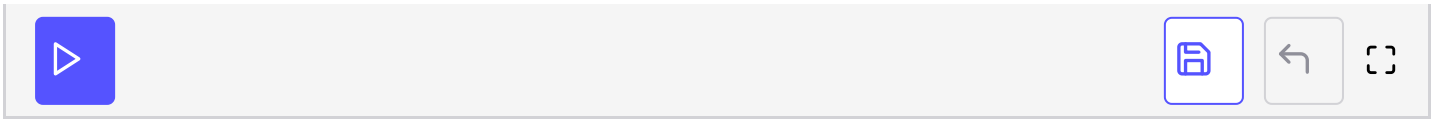
This method returns a `Collector` that collects all of the input elements into a new `Collection`. This method takes a `Supplier` as a parameter. The `Supplier` supplies the collection of our choice.

Below is an example of collecting the first three employees in a `LinkedList`.

> **Note:** In the below example, at **line 18** we provid the supplier to `toCollection()` method as `LinkedList::new`. We caan also write it as `() -> new LinkedList<>();` but we should always prefer method references as they are shorter and more readable.

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.stream.Collectors;

public class CollectorsDemo {

    public static void main(String args[]) {
        List<Employee> employeeList = new ArrayList<>();
        employeeList.add(new Employee("Alex", 23, 23000));
        employeeList.add(new Employee("Ben", 63, 25000));
        employeeList.add(new Employee("Dave", 34, 56000));
        employeeList.add(new Employee("Jodi", 43, 67000));
        employeeList.add(new Employee("Ryan", 53, 54000));

        LinkedList<String> empName = employeeList.stream()
                .map(emp -> emp.getName())
                .collect(Collectors.toCollection(LinkedList::new));

        System.out.println(empName);
    }
}

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

    Employee(String name) {
        this.name = name;
    }

    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 +
                '}';
    }
}
```



4. `Collectors.toMap()`

`toMap()` is used to collect stream elements into a `Map` instance. This method takes two parameters

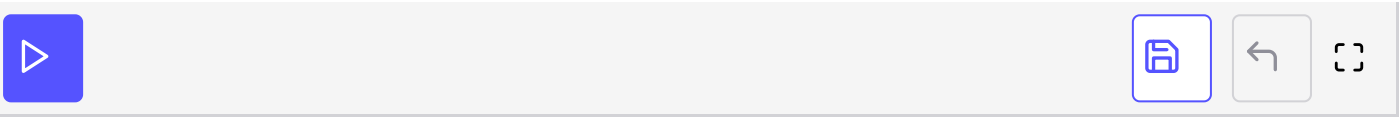
keyMapper - used for extracting a `Map` key from a stream element

valueMapper - used for extracting a value associated with a given key

Suppose we have a list of strings, and we need to create a map where the key is the string and the value is the length of the string. In this case, we can use the `toMap()` method.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectorsDemo {

    public static void main(String args[]) {
        List<String> list = new ArrayList<>();
        list.add("done");
        list.add("far");
        list.add("away");
```

```
        list.add("again");

        Map<String,Integer> nameMap = list.stream()
                .collect(Collectors.toMap(s -> s , s -> s.length()));

        System.out.println(nameMap);
    }
}
```

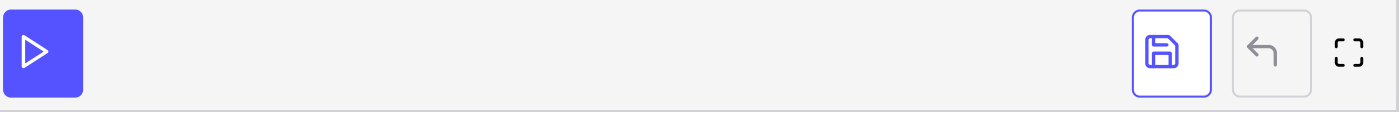


The problem with the above example is that, if the list has duplicate elements, `toMap()` will throw an exception.

To solve this problem, there is an overloaded version of `toMap()` that takes an additional `BinaryOperator` as a parameter. This is used to decide which element should be considered in case of duplicates.

In the below example, we have provided a `BinaryOperator` that will take the first element in case a duplicate element is found. Since the length of both strings will be the same it doesn't matter which element we take.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectorsDemo {

    public static void main(String args[]) {
        List<String> list = new ArrayList<>();
        list.add("done");
        list.add("far");
        list.add("away");
        list.add("done");

        Map<String,Integer> nameMap = list.stream()
                .collect(Collectors.toMap(s -> s , s -> s.length(), (s1,s2) -> s1));

        System.out.println(nameMap);
    }
}
```



There is one more overloaded version of `toMap()` method, which allows us to provide the implementation of `Map` that you want to use.

In the below example, we will convert our stream to a HashMap.

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class CollectorsDemo {

    public static void main(String args[]) {
        List<String> list = new ArrayList<>();
        list.add("done");
        list.add("far");
        list.add("away");
        list.add("done");

        Map<String,Integer> nameMap = list.stream()
                .collect(Collectors.toMap(s -> s , s -> s.length(), (s1,s2) -> s1, HashMap::new));

        System.out.println(nameMap);
    }
}
```

5. `collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)`

This method returns a `Collector` that accumulates the input elements into the given `Collector` and then performs an additional finishing function.

In the below example, we are collecting the elements in a list and then converting the list into an unmodifiable list.

```
import java.util.*;
import java.util.stream.Collectors;

public class CollectorsDemo {

    public static void main(String args[]) {
        List<String> list = new ArrayList<>();
        list.add("done");
        list.add("far");
        list.add("away");
        list.add("done");

        List<String> unmodifiableList = list.stream()
                .collect(Collectors.collectingAndThen(Collectors.toList(), Collections::unmodifiab

        System.out.println(unmodifiableList);
    }
}
```

In the next lesson, we will discuss a few other methods, which are used to calculate some data, available in the `Collectors` class.