Grouping operations

Grouping operations are one of the most important features of streams because they can help you complete a task, which otherwise would have taken a lot of coding, in just 2-3 lines of code.

Let's say, for example, we have a list of `Employee` objects. We need to group all our employees based on their countries of residence. Or, say we need to find the average age/salary of all employees in a particular country. These kinds of operations can be done very easily with grouping APIs provided in the `Collectors` class.

Let's explore these APIs in detail.

1) `Collectors.groupingBy()`

This method groups the input elements according to the supplied classifier and returns the results in a `Map`.

This method is similar to the group by clause of SQL, which can group data on some parameters.

There are three overloaded versions of this method. We will discuss each one of them.

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

This method takes only an instance of a `Function` interface as a parameter.

In the below example, we use `groupingBy()` to group the `Employee` objects based on countries of residence.

```
import java.util.*;
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, 25000, "USA"));
        employeeList.add(new Employee("Ben", 63, 25000, "China"));
        employeeList.add(new Employee("Dave", 36, 50000, "India"));
        employeeList.add(new Employee("Jodi", 43, 67000, "USA"));
        employeeList.add(new Employee("Ryan", 53, 54000, "China"));

        // The employees are grouped by country using the groupingBy() method.
        Map<String,List<Employee>> employeeMap = employeeList.stream()
                .collect(Collectors.groupingBy(Employee::getCountry));

        System.out.println(employeeMap);
    }
}

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;
    }
}
```

Run | Save | Reset

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

This method takes an additional second collector, which is applied to the results of the first collector.

In the previous example, the value of `Map` was a `List` of employees. However, what if we need a `Set` of employees?

In that case, we can use this method to provide a *downstream* `Collector` as shown below:

```
import java.util.*;
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, 25000, "USA"));
        employeeList.add(new Employee("Ben", 63, 25000, "China"));
        employeeList.add(new Employee("Dave", 36, 50000, "India"));
        employeeList.add(new Employee("Jodi", 43, 67000, "USA"));
        employeeList.add(new Employee("Ryan", 53, 54000, "China"));

        Map<String, Set<Employee>> employeeMap = employeeList.stream()
                .collect(Collectors.groupingBy(Employee::getCountry, Collectors.toSet()));

        System.out.println(employeeMap);
    }
}

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;
    }
}
```

Run | Save | Reset

There are lots of interesting use cases that we can solve using this method.

Suppose we need to group on multiple conditions. Then we can provide another `groupingBy()` as *downstream*.

In the below example we will group by country and age of the employees.

```
import java.util.*;
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, 25000, "USA"));
        employeeList.add(new Employee("Ben", 63, 25000, "China"));
        employeeList.add(new Employee("Dave", 36, 50000, "India"));
        employeeList.add(new Employee("Jodi", 43, 67000, "USA"));
        employeeList.add(new Employee("Ryan", 53, 54000, "China"));

        // Grouping, once are grouped by country and later by using the groupingBy() method twice.
        Map<String, Map<Integer,List<Employee>>> employeeMap = employeeList.stream()
                .collect(Collectors.groupingBy(Employee::getCountry, Collectors.groupingBy(Employee::getAge)));

        System.out.println(employeeMap);
    }
}

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;
    }
}
```

Run | Save | Reset

Suppose we need to get a `Map` where the key is the name of the country and the value is the sum of salaries of all of the employees of that country.

This can be easily done by providing a `summingInt()` as the *downstream* `Collector`.

```
import java.util.*;
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, 25000, "USA"));
        employeeList.add(new Employee("Ben", 63, 25000, "China"));
        employeeList.add(new Employee("Dave", 36, 50000, "India"));
        employeeList.add(new Employee("Jodi", 43, 67000, "USA"));
        employeeList.add(new Employee("Ryan", 53, 54000, "China"));

        Map<String, Integer> employeeMap = employeeList.stream()
                .collect(Collectors.groupingBy(Employee::getCountry, Collectors.summingInt(Employee::getSalary)));

        System.out.println(employeeMap);
    }
}

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;
    }
}
```

Run | Save | Reset

Next, suppose we need to get a `Map` where the key is the name of the country and the value is the `Employee` object that has the max salary in that country.

This can be easily done by providing a `maxBy()` as the *downstream* `Collector`.

```
import java.util.*;
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, 25000, "USA"));
        employeeList.add(new Employee("Ben", 63, 25000, "China"));
        employeeList.add(new Employee("Dave", 36, 50000, "India"));
        employeeList.add(new Employee("Jodi", 43, 67000, "USA"));
        employeeList.add(new Employee("Ryan", 53, 54000, "China"));

        Map<String, Optional<Employee>> employeeMap = employeeList.stream()
                .collect(Collectors.groupingBy(Employee::getCountry, Collectors.maxBy(Comparator.comparing(Employee::getSalary))));

        System.out.println(employeeMap);
    }
}

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;
    }
}
```

Run | Save | Reset

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

The third variant of `groupingBy()` takes a supplier as an additional parameter.

This method is used, if we need to provide the implementation of the `Map` we need.

```
import java.util.*;
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, 25000, "USA"));
        employeeList.add(new Employee("Ben", 63, 25000, "China"));
        employeeList.add(new Employee("Dave", 36, 50000, "India"));
        employeeList.add(new Employee("Jodi", 43, 67000, "USA"));
        employeeList.add(new Employee("Ryan", 53, 54000, "China"));

        Map<String, Set<Employee>> employeeMap = employeeList.stream()
                .collect(Collectors.groupingBy(Employee::getCountry, HashMap::new, Collectors.toSet()));

        System.out.println(employeeMap);
    }
}

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;
    }
}
```

Run | Save | Reset

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

The `groupingByConcurrent()` collector is similar to the `groupingBy()` collector; the only difference is that this method returns an instance of `ConcurrentMap`.

This collector also has three overloaded methods that take the exact same arguments as the respective overloaded methods of the groupingBy collector.

```
import java.util.*;
import java.util.concurrent.ConcurrentMap;
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, 25000, "USA"));
        employeeList.add(new Employee("Ben", 63, 25000, "China"));
        employeeList.add(new Employee("Dave", 36, 50000, "India"));
        employeeList.add(new Employee("Jodi", 43, 67000, "USA"));
        employeeList.add(new Employee("Ryan", 53, 54000, "China"));

        ConcurrentMap<String,List<Employee>> employeeMap = employeeList.parallelStream()
                .collect(Collectors.groupingByConcurrent(Employee::getCountry));

        System.out.println(employeeMap);
    }
}

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;
    }
}
```

Run | Save | Reset

2) `Collectors.partitioningBy()`

This method partitions the input elements according to the supplied Predicate and returns a `Map<Boolean, List<T>>`.

Since the key is a boolean it only takes two values. Under the true key, we will find elements that match the given `Predicate`. Under the false key, we will find the elements which don't match the given `Predicate`.

In the given example, we will partition the employees that have an age greater than 30 and less than 30.

```
import java.util.*;
import java.util.concurrent.ConcurrentMap;
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, 25000, "USA"));
        employeeList.add(new Employee("Ben", 63, 25000, "China"));
        employeeList.add(new Employee("Dave", 36, 50000, "India"));
        employeeList.add(new Employee("Jodi", 43, 67000, "USA"));
        employeeList.add(new Employee("Ryan", 53, 54000, "China"));

        // Partitioning the list based on age
        Map<Boolean, List<Employee>> employeeMap = employeeList.stream()
                .collect(Collectors.partitioningBy(emp -> emp.getAge() > 30));

        System.out.println(employeeMap);
    }
}

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;
    }
}
```

Run | Save | Reset