Collectors

Collectors is one of the utility class in JDK which contains a lot of utility functions. It is mostly used with Stream API as a final step. In this article, we will study different methods in the collector class.

When it comes to the functional style of programming in Java, we typically have few functions which we use widely and those functions are filter(), map(), reduce(), and collect() which belongs to the Streams API. collect() and reduce() methods are called as the terminal methods because here, the operation gets terminated with some outcome. Functions associated with Collectors usually get used inside collect() methods.

Methods inside the Collectors Class

Let’s consider a City class, which has attributes like name and temperature which are being initialized with the parameterized constructor. We will observe the different methods available in the collectors class using this example.

Below is the implementation of the City class:

```
// Java program to implement a
// City class

// Defining a city class
public class City {

    // Initializing the properties
    // of the city class
    private String name;
    private double temperature;

    // Parameterized constructor to
    // initialize the city class
    public City(String name, double temperature)
    {
        this.name = name;
        this.temperature = temperature;
    }

    // Getters to get the name and
    // temperature
    public String getName()
    {
        return name;
    }

    public Double getTemperature()
    {
        return temperature;
    }

    // Overriding the toString method
    // to return the name and temperature
    @Override
    public String toString()
    {
        return name + " --> " + temperature;
    }
}
```

Before getting into the different methods, let’s create a list of cities with name and temperature. Below is the implementation of a method to create a list of cities with name and temperature:

```
// Java program to create a list
// of cities with name and
// temperature

// Function to create a list of
// cities with name and temperature
private static List<City> prepareTemperature()
{
    List<City> cities = new ArrayList<>();
    cities.add(new City("New Delhi", 33.5));
    cities.add(new City("Mexico", 14));
    cities.add(new City("New York", 13));
    cities.add(new City("Dubai", 43));
    cities.add(new City("London", 15));
    cities.add(new City("Alaska", 1));
    cities.add(new City("Kolkata", 30));
    cities.add(new City("Sydney", 11));
    cities.add(new City("Mexico", 14));
    cities.add(new City("Dubai", 43));
    return cities;
}
```

The following are the various methods to perform operations on the above cities:

1. Collector<T, ?, List<T>> toList(): Transforms the input elements into a new List and returns a Collector. Here, T is the type of the input elements. In the following example, we are trying to process the list of cities whose temperature is more than 10, and get only the city names.

To do so, we use filter() to apply the filter check of temperature, we use map() to transform the city name and use collect() to collect these city names. Now this collect() method is basically used for collecting the elements passed though stream and its various functions and return a List instance.

```
// Java program to implement the
// toList() method
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class GFG {

    public static void main(String[] args)
    {

        // The following statement filters
        // cities having temp > 10
        // The map function transforms only
        // the names of the cities
        // The collect function collects the
        // output as a List
        System.out.println(prepareTemperature().stream()
                  .filter(f -> f.getTemperature() > 10)
                  .map(f -> f.getName())
                  .collect(Collectors.toList()));
    }
}
```

Output:

```
[New Delhi, Mexico, New York, Dubai, London, Kolkata, Sydney, Mexico, Dubai]
```

2. Collector<T, ?, Set<T>> toSet(): Transforms the input elements into a new Set and returns a Collector. This method will return Set instance and it doesn’t contain any duplicates.

```
// Java program to implement the
// toSet() method
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class GFG {

    public static void main(String[] args)
    {

        // Here, we have applied the filter
        // to get the set of the names
        // of the cities whose temperature
        // is greater than 10
        System.out.println(prepareTemperature()
                .stream()
                .filter(f -> f.getTemperature() > 10)
                .map(f -> f.getName())
                .collect(Collectors.toSet()));
        }
}
```

Output:

```
[New York, New Delhi, London, Mexico, Kolkata, Dubai, Sydney]
```

3. Collector<T, ?, C> toCollection(Supplier <C> collectionFactory): Transforms the input elements into a new Collection, and returns a Collector. If we observe toList() and toSet() methods discussed above, We don’t have control over their implementations. So with toCollection() we can achieve custom implementation where C is the type of the resulting collection and T is the type of the input elements.

```
// Java program to implement the
// toCollection() method
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.stream.Collectors;

public class GFG {

    public static void main(String[] args)
    {

        // Here, a list of all the names
        // have been returned
        System.out.println(prepareTemperature()
                .stream()
                .map(f -> f.getName())
                .collect(Collectors.toCollection(List::new)));
    }
}
```
Output:

```
[New Delhi, Mexico, New York, Dubai, London, Alaska, Kolkata, Sydney, Mexico,
Dubai]
```

Similarly, we can use all other implementation classes such as ArrayList, HashSet, TreeSet, etc.

4. Collector<T, ?, Map< K, U>> toMap(Function keyMapper, Function valueMapper): Transforms the elements into a Map whose keys and values are the results of applying the passed mapper functions to the input elements and returns a Collector. toMap() is used to collect input of elements and convert it into Map instance. toMap() methods ask for following arguments:

```
K - Key function
U - Value Function
BinaryOperator(optional)
Supplier(Optional)
```

Let’s try to understand this with an example. For above discussed list of cities and temperatures, we want to fetch the city name with temperature in the Map.

```
prepareTemperature().stream().filter(city -> city.getTemperature() > 10)
.collect(Collectors.toMap(City::getName, City::getTemperature));
```

The above statements work perfectly if the list doesn’t have duplicates. Since our list contains duplicates then it will not filter it out silently as toSet(). Instead, it throws an IllegalStateException. We can avoid and fix this issue by avoiding the key collision(in case of duplicate keys) with the third argument that is BinaryOperator. For example:

```
// Java program to implement the
// Map() method

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

public class GFG {

    public static void main(String[] args)
    {
        // Here, the name and the temperature
        // are defined as the type City
        System.out.println(prepareTemperature()
                .stream()
                .filter(city -> city.getTemperature() > 10)
                .collect(Collectors.toMap(
                            City::getName,
                            City::getTemperature,
                            (key, identicalKey) -> key)));
    }
}
```

Output:

```
{New York=13.0, New Delhi=33.5, London=15.0, Mexico=14.0, Kolkata=30.0,
Dubai=43.0, Sydney=11.0}
```

Binary operator specifies, how can we handle the collision. Above statements pick either of the colliding elements.

5. Collector <T, ?, Map<K, List>> groupingBy(Function classifier): Performs group by operation on input elements of type T. The grouping of elements is done as per the passed classifier function and returns the Collector with result in a Map.

```
// Java program to implement the
// groupingBy() method
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class GFG {

    public static void main(String[] args)
    {
        System.out.println(prepareTemperature()
                .stream()
                .collect(Collectors.groupingBy(City::getName)));
    }
}
```

Output:

```
{New York=[New York -> 13.0], New Delhi=[New Delhi -> 33.5], London=[London ->
15.0], Alaska=[Alaska -> 1.0], Mexico=[Mexico -> 14.0, Mexico -> 14.0],
Kolkata=[Kolkata -> 30.0], Dubai=[Dubai -> 43.0, Dubai -> 43.0], Sydney=[Sydney
-> 11.0]}
```

In the above example, cities like Mexico and Dubai have been grouped, the rest of the groups contains only one city because they are all alone. The return type of above groupingBy() is Map<String, List>.

6. Collector <T, ?, Map> groupingBy(Function classifier, Collector downstream): Performs group by operation on input elements of type T. The grouping of the elements is done as per the passed classifier function and then performs a reduction operation on the values associated with a given key as per the specified downstream Collector and returns the Collector with result in a Map.

7. Collector groupingBy(Function classifier, Supplier mapFactory, Collector downstream): Performs group by operation on input elements of type T, the grouping of elements is done as per the passed classifier function and then performs a reduction operation on the values associated with a given key as per the specified downstream Collector and returns the Collector.

8. Collector counting(): It counts the number of input elements of type T and returns a Collector. This method is used in a case where we want to group and count the number of times each city is present in the collection of elements.

```
// Java program to implement the
// counting() method
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class GFG {

    public static void main(String[] args)
    {
        System.out.println(prepareTemperature()
                .stream()
                .collect(Collectors.groupingBy(
                            City::getName,
                            Collectors.counting())));
    }
}
```

Output:

```
{New York=1, New Delhi=1, London=1, Alaska=1, Mexico=2, Kolkata=1, Dubai=2,
Sydney=1}
```

We can see that the cities Mexico and Dubai count is 2, and the rest are available once. And, the return type of groupingBy is Map.

9. Collector collectingAndThen(Collector downstream, Function finisher): This method allows us to perform another operation on the result after collecting the input element of collection.

```
// Java program to implement the
// collectingAndThen() method
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class GFG {

    public static void main(String[] args)
    {
        // Collects the elements and
        // counts the occurrences
        System.out.println(prepareTemperature()
                .stream()
                .collect(Collectors.groupingBy(
                            City::getName,
                            Collectors.collectingAndThen(
                                    Collectors.counting(),
                                    f -> f.intValue()))));
    }
}
```

Output:

```
{New York=1, New Delhi=1, London=1, Alaska=1, Mexico=2, Kolkata=1, Dubai=2,
Sydney=1}
```

10. Collector joining(): Concatenates the input elements into a String and returns a Collector.

11. Collector joining(CharSequence delimiter): Concatenates the input elements, separated by the specified delimiter, and returns a Collector.

```
// Java program to implement the
// joining() method
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class GFG {

    public static void main(String[] args)
    {
        // Concatenate the collection with
        // comma
        System.out.println(prepareTemperature()
                .stream()
                .filter(city -> city.getTemperature() > 10)
                .map(f -> f.getName())
                .collect(Collectors.joining(", ")));
    }
}
```

Output:

```
New Delhi, Mexico, New York, Dubai, London, Kolkata, Sydney, Mexico, Dubai
```

12. Collector joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix): Concatenates the input elements, separated by the specified delimiter, as per the specified prefix and suffix, and returns a Collector.

```
// Java program to implement the
// joining() method
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class GFG {

    public static void main(String[] args)
    {
        System.out.println(prepareTemperature()
                .stream()
                .filter(city -> city.getTemperature() > 10)
                .map(f -> f.getName())
                .collect(Collectors.joining(" ",
                "Prefix:", ":Suffix")));
    }
}
```

Output:

```
Prefix:New Delhi Mexico New York Dubai London Kolkata Sydney Mexico Dubai:Suffix
```

13. Collector mapping(Function mapper, Collector downstream): Transforms a Collector of the input elements of type U to one the input elements of type T by applying a mapping function to every input element before the transformation.

```
// Java program to implement the
// mapping() method
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class GFG {

    public static void main(String[] args)
    {
        System.out.println(prepareTemperature()
                .stream()
                .collect(Collectors.groupingBy(
                            City::getName,
                            Collectors.mapping(
                                City::getTemperature,
                                Collectors.toList()))));
    }
}
```

Output:

```
{New York=[13.0], New Delhi=[33.5], London=[15.0], Alaska=[1.0], Mexico=[14.0,
14.0], Kolkata=[30.0], Dubai=[43.0, 43.0], Sydney=[11.0]}
```

In the above output, each city group contains only the temperature, and this is done with the help of mapping() method, which takes two function argument of type function and collector. Above mapping method return a list and finally the return type of the above groupingBy() method becomes Map<String, List>. We can also use toSet() method to get the set of elements instead of the list as follows:

```
// Java program to implement the
// joining() method

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

public class GFG {

    public static void main(String[] args)
    {
        System.out.println(prepareTemperature()
                .stream()
                .collect(Collectors.groupingBy(
                            City::getName,
                            Collectors.mapping(
                                    City::getTemperature,
                                    Collectors.toSet()))));
    }
}
```

Output:

```
{New York=[13.0], New Delhi=[33.5], London=[15.0], Alaska=[1.0], Mexico=[14.0],
Kolkata=[30.0], Dubai=[43.0], Sydney=[11.0]}
```

If we observe the output and compare it from the previous one, the duplicates have been removed because it is a set now. The return type of above groupingBy() now becomes Map<String, Set>.

14. Collector<T, ?, Map<Boolean, List>> partitioningBy(Predicate predicate): Partitions the input elements as per the passed Predicate, and transforms them into a Map and returns the Collector.

```
// Java program to implement the
// partitioningBy() method
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class GFG {

    public static void main(String[] args)
    {
        // Here, we are partitioning the list
        // in two groups i.e., Cities having
        // temperatures more than 15
        // and other than that.
        System.out.println(prepareTemperature()
                .stream()
                .collect(Collectors.partitioningBy(
                            city -> city.getTemperature() > 15)));
    }
}
```

Output:

```
{false=[Mexico-> 14.0, New York-> 13.0, London-> 15.0, Alaska-> 1.0, Sydney->
11.0, Mexico-> 14.0], true=[New Delhi-> 33.5, Dubai-> 43.0, Kolkata-> 30.0,
Dubai-> 43.0]}
```

15. Collector partitioningBy(Predicate predicate, Collector downstream): Partitions the input elements as per the passed Predicate, and collects the values of each partition as per another Collector, and transforms them into a Map whose values are the results of the downstream reduction and then return Collector.

16. Collector averagingDouble(ToDoubleFunction mapper): Performs the average of the input elements of type Double, and returns the Collector as a result.

17. Collector averagingInt(ToIntFunction mapper): Performs the average of the input elements of type Int, and returns the Collector as a result.

18. Collector averagingLong(ToLongFunction mapper): Performs the average of the input elements of type Long, and returns the Collector as a result.

19. Collector<T, ?, ConcurrentMap<K, List>> groupingByConcurrent(Function classifier): Performs group by operation on the input elements of type T, the grouping of elements is done as per the passed classifier function and returns concurrent Collector.

20. Collector<T, ?, ConcurrentMap> groupingByConcurrent(Function classifier, Collector downstream): Performs group by operation on the input elements of type T, the grouping of elements is done as per the passed classifier function, and then performs a reduction operation on the values associated with a given key as per the specified downstream Collector and returns a concurrent Collector.

21. Collector groupingByConcurrent(Function classifier, Supplier mapFactory, Collector downstream): Performs group by operation on input elements of type T, the grouping of elements is done as per the passed classifier function, and then performs a reduction operation on the values associated with a given key as per the specified downstream Collector and returns a concurrent Collector.

22. **Collector reducing(BinaryOperator op)**: Performs a reduction of its input elements as per passed BinaryOperator and returns a Collector.

23. Collector reducing(T identity, BinaryOperator op): Performs a reduction of its input elements as per passed BinaryOperator and as per the passed identity and returns Collector.

24. Collector<T, ?, Optional> maxBy(Comparator comparator): Produces the maximal element as per given Comparator, returns an Optional Collector.

25. Collector<T, ?, Optional> minBy(Comparator comparator): Produces the minimal element as per given Comparator, returns an Optional Collector.

26. Collector<T, ?, ConcurrentMap> toConcurrentMap(Function keyMapper, Function valueMapper): Transforms elements into a ConcurrentMap whose keys and values are the results of the passed mapper functions to the input elements and returns a concurrent Collector.