java.util.stream.Stream.collect Method

Unlike the `reduce` method, which always creates a new value when it processes an element, the `collect` method modifies, or mutates, an existing value.

Consider how to find the average of values in a stream. You require two pieces of data: the total number of values and the sum of those values. However, like the `reduce` method and all other reduction methods, the `collect` method returns only one value. You can create a new data type that contains member variables that keep track of the total number of values and the sum of those values, such as the following class, `Averager`:

```
class Averager implements IntConsumer
{
    private int total = 0;
    private int count = 0;

    public double average() {
        return count > 0 ? ((double) total)/count : 0;
    }

    public void accept(int i) { total += i; count++; }
    public void combine(Averager other) {
        total += other.total;
        count += other.count;
    }
}
```

The following pipeline uses the `Averager` class and the `collect` method to calculate the average age of all male members:

```
Averager averageCollect = roster.stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(Person::getAge)
    .collect(Averager::new, Averager::accept, Averager::combine);

System.out.println("Average age of male members: " +
    averageCollect.average());
```

The `collect` operation in this example takes three arguments:

- `supplier`: The supplier is a factory function; it constructs new instances. For the `collect` operation, it creates instances of the result container. In this example, it is a new instance of the `Averager` class.
- `accumulator`: The accumulator function incorporates a stream element into a result container. In this example, it modifies the `Averager` result container by incrementing the `count`variable by one and adding to the `total` member variable the value of the stream element, which is an integer representing the age of a male member.
- `combiner`: The combiner function takes two result containers and merges their contents. In this example, it modifies an `Averager` result container by incrementing the `count`variable by the `count` member variable of the other `Averager` instance and adding to the `total` member variable the value of the other `Averager` instance's `total` member variable.

Note the following:

- The supplier is a lambda expression (or a method reference) as opposed to a value like the identity element in the `reduce` operation.
- The accumulator and combiner functions do not return a value.
- You can use the `collect` operations with parallel streams; see the section Parallelism for more information. (If you run the `collect` method with a parallel stream, then the JDK creates a new

thread whenever the combiner function creates a new object, such as an Averager object in this example. Consequently, you do not have to worry about synchronization.)

Although the JDK provides you with the average operation to calculate the average value of elements in a stream, you can use the collect operation and a custom class if you need to calculate several values from the elements of a stream.

The collect operation is best suited for collections. The following example puts the names of the male members in a collection with the collect operation:

```
List<String> namesOfMaleMembersCollect = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(p -> p.getName())
    .collect(Collectors.toList());
```

This version of the collect operation takes one parameter of type Collector. This class encapsulates the functions used as arguments in the collect operation that requires three arguments (supplier, accumulator, and combiner functions).

The Collectors class contains many useful reduction operations, such as accumulating elements into collections and summarizing elements according to various criteria. These reduction operations return instances of the class Collector, so you can use them as a parameter for the collect operation.

This example uses the Collectors.toList operation, which accumulates the stream elements into a new instance of List. As with most operations in the Collectors class, thetoList operator returns an instance of Collector, not a collection.

The following example groups members of the collection roster by gender:

```
Map<Person.Sex, List<Person>> byGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(Person::getGender));
```

The groupingBy operation returns a map whose keys are the values that result from applying the lambda expression specified as its parameter (which is called a *classification function*). In this example, the returned map contains two keys, Person.Sex.MALE and Person.Sex.FEMALE. The keys' corresponding values are instances of List that contain the stream elements that, when processed by the classification function, correspond to the key value. For example, the value that corresponds to key Person.Sex.MALE is an instance of List that contains all male members.

The following example retrieves the names of each member in the collection roster and groups them by gender:

```
Map<Person.Sex, List<String>> namesByGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(
                Person::getGender,
                Collectors.mapping(
                    Person::getName,
                    Collectors.toList())));
```

The groupingBy operation in this example takes two parameters, a classification function and an instance of Collector. The Collector parameter is called a *downstream collector*. This is a collector that the Java runtime applies to the results of another collector. Consequently, this groupingBy operation enables you to apply a collect method to the List values created by the groupingBy operator. This example applies the collector mapping, which applies the mapping function Person::getName to each element of the stream. Consequently, the resulting stream consists of only the names of members. A pipeline that contains one or more downstream collectors, like this example, is called a *multilevel reduction*.

The following example retrieves the total age of members of each gender:

```
Map<Person.Sex, Integer> totalAgeByGender =
    roster
        .stream()
        .collect(
            Collectors.groupingBy(
                Person::getGender,
                Collectors.reducing(
                    0,
                    Person::getAge,
                    Integer::sum)));
```

The reducing operation takes three parameters:

- identity: Like the Stream.reduce operation, the identity element is both the initial value of the reduction and the default result if there are no elements in the stream. In this example, the identity element is 0; this is the initial value of the sum of ages and the default value if no members exist.
- mapper: The reducing operation applies this mapper function to all stream elements. In this example, the mapper retrieves the age of each member.
- operation: The operation function is used to reduce the mapped values. In this example, the operation function adds Integer values.

The following example retrieves the average age of members of each gender:

```
Map<Person.Sex, Double> averageAgeByGender = roster
    .stream()
    .collect(
        Collectors.groupingBy(
            Person::getGender,
            Collectors.averagingInt(Person::getAge)));
```