LAMBDA EXPRESSIONS AND STREAM API

Büşra Nur AVCI

Software QA Automation Test Engineer

https://www.linkedin.com/in/b%C3%BC%C5%9Fra-nur-avci-b2592a157/

What is a Lambda Expressions?

A Lambda Expressions are a "Functional Programming". The new “java.util.stream” package was added to JDK8 to enable Java developers to perform operations such as search, filter, map, reduce, or manipulate on collections like Lists. Lambda is similar to methods, but they do not need a name and they can be implemented right in the body of a method. In other words, ready-made methods are used because the focus is on "what to do". This makes our codes run faster.

Let’s take a look at the purposes of the stream codes that we can use in Lambda Expressions !

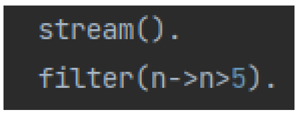
stream( )

It’s our important method that starts the flow for this process,so the first thing we should use. The “stream()” methods returns as a Stream object to process the elements of a collections. So, other ready-methods can be called thanks to “stream()” method.

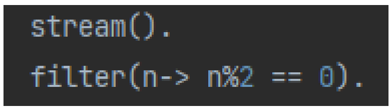
filter( )

filter() is used to filter elements according to a specific ‘condition’. In order to create our in-method condition, there is a spelling to be considered.

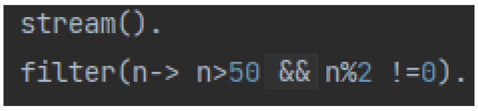
Some examples ;



```
stream().
filter(n->n>5).
```



```
stream().
filter(n-> n%2 == 0).
```



```
stream().
filter(n-> n>50 && n%2 !=0).
```

** *"n" is the general use value. It is not mandatory to use, it can be varied.*

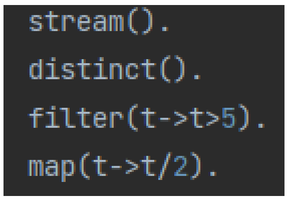
distinct( )

It is used to remove repetitive elements and make unique. It’s written the same as stream() and no value is written.

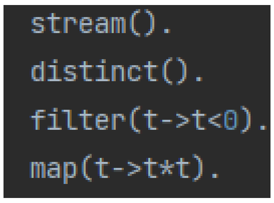
map( )

It’s processes the elements and “changes” the existing element. We can do this by manipulating a value or by calling methods created in Util class. For String values; the methods we call from the String class are used for the same purpose.

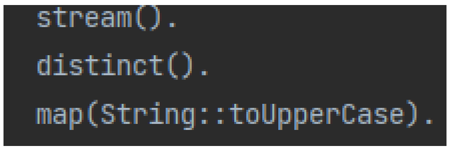
Some examples ;



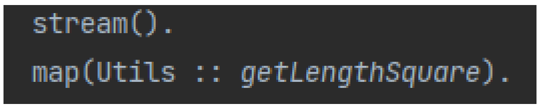
```
stream().
distinct().
filter(t->t>5).
map(t->t/2).
```



```
stream().
distinct().
filter(t->t<0).
map(t->t*t).
```



```
stream().
distinct().
map(String::toUpperCase).
```



```
stream().
map(Utils :: getLengthSquare).
```

** *The "t" value can be changed, as I mention above ^*

reduce( )

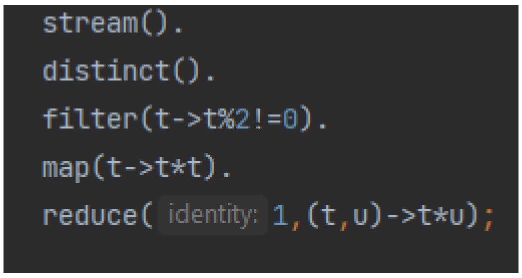
The result of the reduce() method is a single value produced by combining all elements in a Stream according to a specific operation. Therefore, reduce() is the last link in the chain of Stream operations and produces the result. That means it’s a terminal code.
→ Terminal code is code that produces the result of a Stream operation and terminates the flow.

reduce() has an initial value,specified as “identity”.
→(for sum: “0” , for multiplication: “1”) ←

Last important rule is; reduce() works with a “data type” and the inside of the method must be the same as the data type !

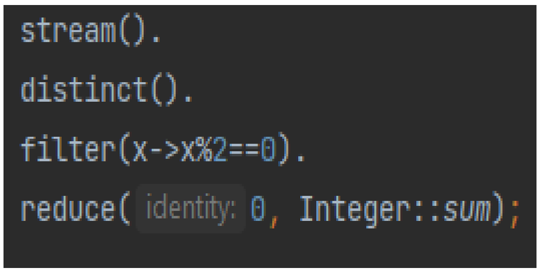
Some examples ;

int result =



```
stream().
distinct().
filter(t->t%2!=0).
map(t->t*t).
reduce( identity: 1,(t,u)->t*u);
```

int resultSum=



```
stream().
distinct().
filter(x->x%2==0).
reduce( identity: 0, Integer::sum);
```

** *The "t" and “x” values can be changed, as I mention above ^*

sorted( )

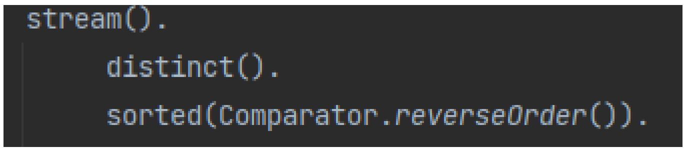
It is used to sort those elements in a specific order. sorted() works with the “natural order” rule ;

For Integer values it sorts →from smallest to largest,

For String values it sorts → alphabetical.

If we want or need to sort in the "opposite" of this rule, we must call the **reverseOrder**() method.

*Its usage should be as follows;



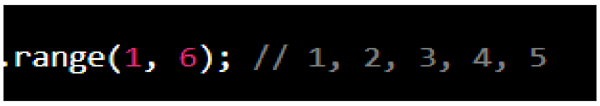
```
stream().
      distinct().
      sorted(Comparator.reverseOrder()).
```

Reminder Note

The “Comparator” class is an interface class in Java that allows two object to compared. This class is especially used in sorting operations .

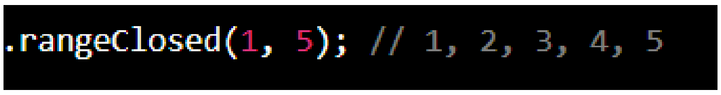
range( )
rangeClosed( )

The **range()** method returns a range containing the values between the given start and end values, but not including the last value (end) !!



```
.range(1, 6); // 1, 2, 3, 4, 5
```

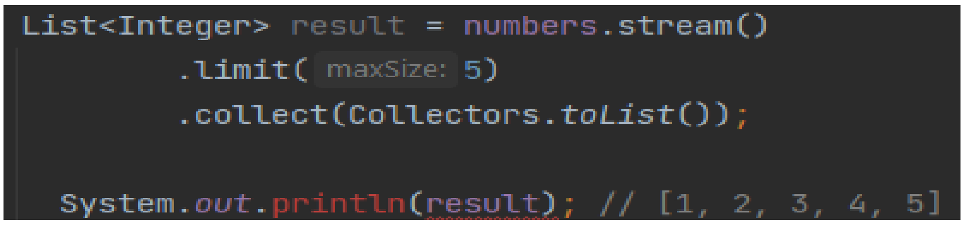
The **rangeClosed()** method creates a range in the same way, BUT the last value (end) is included. That is, BOTH the start and end values are part of the range !!



```
.rangeClosed(1, 5); // 1, 2, 3, 4, 5
```

limit( )

The limit() method returns a new Stream containing the "***first N elements***" of a Stream. (This method is useful when working with large amounts of data and only a few elements are needed.)

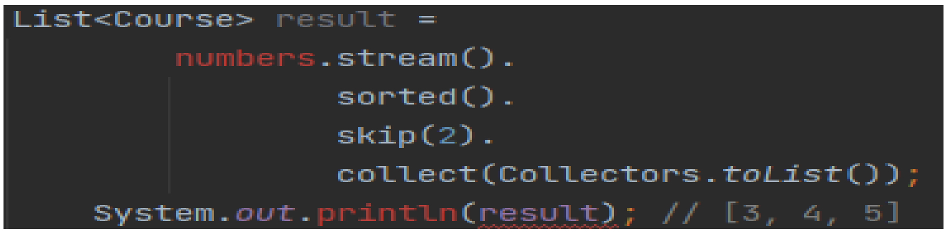


```
List<Integer> result = numbers.stream()
        .limit( maxSize: 5)
        .collect(Collectors.toList());

  System.out.println(result); // [1, 2, 3, 4, 5]
```

skip( )

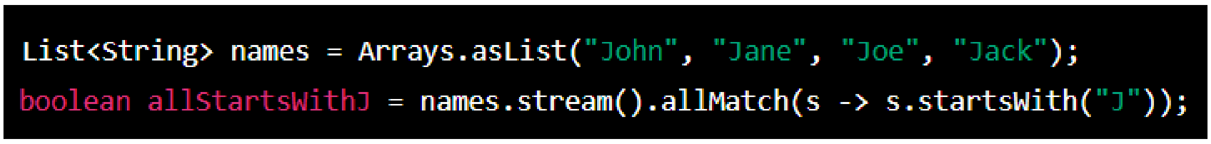
The skip() method creates a new Stream by "skipping a certain number of elements" from a Stream. (The use of this method can be very useful in situations where data needs to be sorted and a certain number of records need to be skipped.)



```
List<Course> result =
        numbers.stream().
                sorted().
                skip(2).
                collect(Collectors.toList());
    System.out.println(result); // [3, 4, 5]
```

allMatch( )
anyMatch( )
noneMatch( )

allMatch(): is used to check if **"all"** of the elements satisfy a given condition.



```
List<String> names = Arrays.asList("John", "Jane", "Joe", "Jack");
boolean allStartsWithJ = names.stream().allMatch(s -> s.startsWith("J"));
```

*** Output is "TRUE"*

anyMatch(): is used to check if "**at least one**" of the elements satisfies a certain condition.

```
List<String> fruits = Arrays.asList("banana", "orange", "apple", "grape");

boolean hasApple = fruits.stream().anyMatch(fruit -> fruit.equals("apple"));
```

*** Output is "TRUE"*

noneMatch(): is used to check if "**none**" of the elements in the collection satisfy a given condition.

```
List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);
boolean noneGreaterThan100 = numbers.stream().noneMatch(n -> n > 100);
```

*** Output is "TRUE"*

→ → They contain a return type which is "***boolean***" ← ←

There are more methods that can be added to all these ready methods, of course BUT this worksheet is entirely my own. In addition to this is the most general and summarized version of the ready- methods that JAVA has created for Lambda Expression (with Stream API).

I hope you like it

and

I hope our paths will cross one day! 😊

Büşra Nur AVCI