More Streams

New Austrian Coding School

Programming - memo

The purpose of a Stream is to **conduct** an **information flow**. The end of this flow can be a rearrangement of its elements, a side operation, finding one element, counting them, accumulating them or check a condition.

Collect

We can **group** elements **counting** how many times they appear.

```
// count how many times each word appears
Map<String, Long> groupped = words.stream()
                                        .collect(Collectors.groupingBy(e -> e, Collectors.counting()));
```

We can **group** them by some common **factor** that they share.

```
// group words by word length
Map<Integer, List<String>> lengths = words.stream()
                                                .collect(Collectors.groupingBy(e -> e.length(), Collectors.toList()));
```

Count

The count **terminal operation** returns a **long** representing the number of elements.

```
// count how many words with at least two e's
long numberOfWords = words.stream()
                                .filter(e -> 2 <= Stream.of(e.split("")).filter(l -> l.equalsIgnoreCase("e")).count())
                                .count();
```

Reduce

The purpose of the reduce **terminal operation** is to have **one** single element at the end.

```
// accumulate the total sum of all ascii codes
Integer totalAscii = letters.stream()
    .map(e -> (int) e.charAt(0))
    .reduce(0, (e1, e2) -> e1 + e2);
```

Perform checks

Checks are **terminal operations** and return a **boolean** that answers a question.

noneMatch returns true if no element fulfills a condition, otherwise it returns false.

```
boolean noneStartsWithLi = words.stream()
    .noneMatch(e -> e.startsWith("Li"));

anyMatch returns true if any element fulfills a condition

boolean atLeastOneContainsLetterU = words.stream()
```

```
    .anyMatch(e -> e.contains("u"));
```

allMatch returns true if every element fulfills a condition

```
boolean areAllLongerThan1Letter = words.stream()
    .allMatch(e -> e.length() > 1);
```

Producing more elements

Some operations **produce** more elements that the ones **originally** in the Stream. If we want to stream them **one** by **one** we use the *flatMap* **intermediate operation**. We do so by explaining **how** we can **create** another Stream from those elements.

```
// count the number of letter a's
long numberOfLetterAs = words.stream()
    .map(e -> e.split("")) // words as String into letters as String[] Array
    .flatMap(e -> Stream.of(e)) // letters as String[] Array into letters as Stream of String one by one
    .filter(e -> e.equalsIgnoreCase("a")) // keep only letter a's
    .count();
```