EXPLORE NEW PERSPECTIVES

Parallel and concurrent programming in Java 8

Part VI - advanced stream operations

peek()

Stream<T> **peek**(Consumer<? super T> action)

produces a stream after applying the operation

only for debugging!

```
OptionalInt value = IntStream.of(1, 2, 3, 4)
        .peek(x -> System.out.println("processing: " + x))
        .filter(x -> x % 2 == 0)
        .peek(x -> System.out.println("accepted " + x))
        .findFirst();
```

Other map flavors

produces a stream of
primitive types

DoubleStream **mapToDouble**(ToDoubleFunction<? super T> mapper)
IntStream **mapToInt**(ToIntFunction<? super T> mapper)
LongStream **mapToLong**(ToLongFunction<? super T> mapper)

```
List<String> list6 = Arrays.asList("Mariapia", "Teresa");

int sum = list6.stream()
               .mapToInt(String::length)
               .sum()
```

Other map flavors

can change the type
of a stream of
primitive types

IntStream **map**(IntUnaryOperator mapper)
DoubleStream **mapToDouble**(IntToDoubleFunction mapper)
LongStream **mapToLong**(IntToLongFunction mapper)
Stream<T> **mapToObj**(IntFunction<? extends T> mapper)

```
List<Integer> list7 = IntStream.rangeClosed(1, 10)
        .mapToObj(x -> x * 2)
        .collect(Collectors.toList());
```

boxed()

converts a specialized stream into a Stream with boxed values

```
List<Integer> list8 = IntStream
        .rangeClosed(1, 10)
        .boxed()
        .collect(Collectors.toList());
```

forEachOrdered()

processes the elements in the order specified by the stream, independently if the stream is executed serial or parallel

```
IntStream.rangeClosed(1, 100)
        .parallel()
        .map(x -> x + 1)
        .forEachOrdered(System.out::println);
```

unordered(), parallel() and sequential()

unordered() transforms the stream from sequential to unordered

parallel() determines a parallel mode for execution of the stream

sequential() determines a sequential mode for execution of the stream

unordered(), parallel() and sequential()

parallel processing example

```
List<Integer> list8 = IntStream.rangeClosed(1, 10)
        .boxed()
        .collect(Collectors.toList());

List<Integer> list9 = list8.stream()
        .unordered()
        .parallel()
        .peek(x -> System.out.println(Thread.currentThread()
                                        .getName()))
        .map(x -> x + 1)
        .collect(Collectors.toList());
```

unordered(), parallel() and sequential()

what happens here?

```
List<Integer> list8 = IntStream.rangeClosed(1, 10)
        .boxed()
        .collect(Collectors.toList());

List<Integer> list9 = list8.stream()
        .unordered()
        .parallel()
        .peek(x -> System.out.println(Thread.currentThread()
                                                        .getName()))
        .sequential()
        .map(x -> x + 1)
        .collect(Collectors.toList());
```

unordered(), parallel() and sequential()

the stream has a single
execution mode!

A bit more about flatMap()

these two examples are equivalent

```
List<String> list13 = Arrays.asList("Mariapia", "Teresa");

list13.stream()
        .map(x -> x.length())
        .forEachOrdered(System.out::println);

list13.stream()
        .flatMap(x -> Stream.of(x.length()))
        .forEachOrdered(System.out::println);
```

A bit more about flatMap()

get, for each number *x* in the input stream,
the pair (*x*, *2*x*)

```
List<Integer> list8 = IntStream.rangeClosed(1, 10)
        .boxed()
        .collect(Collectors.toList());

list8.stream()
        .map(x -> new int[]{x, 2 * x})
        .forEach(x -> System.out.println(x[0] + ", " + x[1]));
```

A bit more about flatMap()

can be implemented as

```
list8.stream()
        .flatMap(x -> Stream.of(x, 2 * x))
        .forEach(System.out::println);
```

or even better

```
IntStream.rangeClosed(1, 10)
        .flatMap(x -> IntStream.of(x, 2 * x))
        .forEach(System.out::println);
```

A bit more about flatMap()

create a single stream from two lists

```
Stream.of(list11, list12)
	.flatMap(x -> x.stream())
	.forEachOrdered(System.out::println);
```

A bit more about flatMap()

combining values from two streams

```
list11.stream()
        .flatMap(x -> list12.stream()
                .flatMap(y -> Stream.of(x, y)))
        .forEachOrdered(x -> System.out.print(x + " "));
```

reduce()

combine the elements of a stream repeatedly to produce a single value

summation

```
int tot = list15.stream()
        .reduce(0, (x, y) -> x + y);
```

product

```
int tot = list15.stream()
        .reduce(1, (x, y) -> x * y);
```

reduce()

can be also written as

```
int tot3 = list15.stream()
        .reduce(0, Integer::sum);
```

the initial value can be omitted

```
Optional<Integer> tot4 = list15.stream()
                                                     .reduce((x,y) -> x + y);
```

reduce()

calculate the minimum

```
Optional<Integer> tot5 = list15.stream()
      .reduce((x, y) -> x < y ? x : y);
```

other possibility

```
Optional<Integer> tot6 = list15.stream()
                                        .reduce(Integer::min);
```

reduce()

what about concatenation of strings?

```
List<String> list16 = Arrays
                              .asList("Stefano", "Mariapia", "Enrico");
String str = list16.stream().reduce("", (x,y) -> x + y);
```

other possibility:

```
String str2 = books
        .stream()
        .collect(Collectors
                .reducing("titles: ", Book::getTitle, (x, y) -> x + y));
```

reduce

other examples

```
int count = books
        .stream()
        .map(x -> 1)
        .reduce(0, (x,y) -> x + y);
```

```
int totalPages = books
        .stream()
        .collect(Collectors
                .reducing(0, Book::getNumberOfPages,
                                (x,y) -> x + y));
```

ESTECO

EXPLORE NEW PERSPECTIVES

Thank you
for your attention!