Feb 2017

STREAMS java.util.stream.Stream

classmate Date 20/01/17 Page

Streams allow to write collection processing code at a higher level of abstraction. It allows programmers to write codes that are -
(i) declarative - more concise and readable
(ii) composable - greater flexibility
(iii) parallelizable - better performance (maximize performance for multicore arch. transparently; dont need to specify how many threads to use)

Streams can be defined as a sequence of elements from a source that supports data processing operations.

Collections are data structures focusing on storing and accessing of elements. Streams are about expressing computations. But like collection, stream provides an interface to a sequence of specific type of elements.

Source - Streams consume data from a data providing source such as collections, arrays or I/O resources. Stream from an ordered collection preserves the ordering.

Data processing operations - • Supports both database like operations and functional programming operations to manipulate data. e.g filter, map, find, sort etc.
• Operations can be executed in sequence or in parallel.

Two characteristics of stream operations -

- PIPELINING
  - LAZINESS
  - SHORT-CIRCUITING
- INTERNAL ITERATIONS

```
List<String> threeHighCalorieDishes = menu.stream().
                                      .filter(d -> d.getCalories > 300)
                                      .map(Dish::getName)
                                      .limit(3)
                                      .collect(toList());
```

(menu = data source)

forms the pipeline: filter, map, limit
processes the pipeline to return a result: collect
Data processing operations: filter, map, limit, collect

Dish (java bean)

```
String name; boolean vegetarian;
int calories; Type type;
```
final, private

```
String getName();
boolean isVegetarian();
int getCalories();
Type getType();
String toString();
```
public

```
public enum Type {MEAT, FISH, OTHER}
```

STREAMS Vs COLLECTIONS

Similarity - Both provide interfaces to data structures representing a sequenced set of values of the element type.

Difference - when things are computed -

(i) every element is computed before it is added to a collection
conceptually stream is a fixed data structure whose elements are computed on demand. Thus stream can be viewed as a lazily constructed collection where values are computed when solicited by the user (consumer). e.g. constructing a stream of all prime numbers. — Consumer driven

Collection follows supplier-driven model as the elements are computed eagerly. Thus not possible to construct list of prime nos

Visual Metaphor - DVD vs ONLINE STREAMING

(iii) Streams are traversable only once. After that the stream is said to be consumed. e.g.

```
List<String> title = Arrays.asList("Java8", "In", "Action");
Stream<String> s = title.stream();
s.forEach(System.out::println);
s.forEach(System.out::println); → java.lang.IllegalStateException
```

prints title words

⇒ Stream is a set of values spread out in time.
Collection is a set of values spread out in space.

(iv) Internal vs External Iteration :-

- Using internal iteration, processing of items can be done in ||^{l} or in a different order that may be more optimized.
- In internal operation, stream library can automatically choose data representation and implementation of ||^{l}ism to match the machine hardware.

In collec^{n}s programmer needs to implement parallelism and define the order in which elements of a collec^{n} can be processed.

STREAM OPERATIONS

INTERMEDIATE
filter, map, limit

TERMINAL
collect, ~~toList~~ count, forEach

```
List<String> Names = menu.stream()
                .filter(d -> { S.o.p("filtering" + d.getName());
                    return d.getCalories() > 300; })
                .map(d -> { S.o.p("mapping" + d.getName());
                    return d.getName(); })
                .limit(3)
                .collect(toList());
S.o.p(names);
```

(i) Despite the fact that there are many high calorie dishes only first 3 are selected for filtering. — because of limit(3) ⇒ Short Circuiting

(ii) filter and map two separate operations are merged into one pass. ⇒ Loop Fusion

O/p → filtering soup
mapping soup
filtering boiled veg
mapping " "
filtering chicken
mapping chicken
[soup, boiledVeg, Chicken]

collect() is a terminal operation as it returns a non-Stream value.

Intermediate operations are not computed unless a terminal operation is specified.

Common Stream Operations

1. **Filtering** – where clause of select statement ⇒ takes as argument a predicate and returns a stream including all elements that match the predicate.

```
List<String> beginWithNos = Stream.of("a", "1ab", "2abc", "i2")
        .filter(value -> isDigit(value.charAt(0)))
        .collect(toList());
```

ordered collection

T → [Predicate] → boolean

```
List<Integer> numbers = Arrays.asList(1,2,1,3,3,4);
numbers.stream().filter(i -> i%2==0)
        .distinct().forEach(d -> {s.o.p("..." + d);});
```

Find out vegetarian

```
menu.stream().filter(d -> d.isVegetarian()).collect(toList());
```

Can filter unique elements, ignoring first few elements or truncating a stream to a given size.

2. **Skipping elements / limiting elements** – limit(n) takes first n members of a stream while skip(n) skips first n members and returns the rest. If a stream has fewer than n elements then empty stream is returned by skip(n).

```
List<Integer> numbers = Arrays.asList(1,2,1,3,3,5,7,9);
List<Integer> n = numbers.stream().filter(i -> i%2==0)
        .skip(2)
        .collect(toList());
```

Limit also works on unordered streams like Set as well.

3. **Mapping** – Project op^n of relational algebra. – Replaces value in a stream with a new value.

The lamda expression passed to the map() must be an element of Function.

T → [Function] → R

```
List<String> collected = Stream.of("a", "abc", "At")
        .map(s -> s.length()) / .map(s -> s.toUpperCase())
        .collect(toList());      .collect(toList());
```

FlatMap replaces a value with a stream and concatenates all streams together.

```
List<Integer> together = Stream.of(asList(1,2), asList(2,3))
        .flatMap(numbers -> numbers.stream())
        .collect(toList());
```

T → [Function] → Stream<R>

1. Given a list {1,2,3,4} square each no. `numbers.stream().map(n -> n*n).collect(toList())`
where `List<Integer> numbers = Arrays.asList(1,2,3...`

2. Given 2 lists, form a pair such that sum of the nos in a pair is even.
i.e. { (x, y, z) }

```
List<int[]> pairs = numbers1.stream()
    .flatMap(i -> numbers2.stream().filter(j -> (i+j)%2 == 0)
    .map(j -> new int[]{i, j})).collect(to
```

```
String[] arrayOfWords = {"Hi", "High", "Height"};
List<String> uniqueCharacters = arrayOfWords.stream()
    .map(w -> w.split(""))
    .flatMap(x -> Arrays.stream(x))
    .distinct()
    .collect(Collectors.toList());
```

Converts each word into an array of individual letters → It returns Stream<Stri
flattens each gen. stream into a single stream → returns Streams<Stri

4. Max and Min - These fns takes as input an element of the Comparator. A static method comparing() is added in Java 8 that takes

4. Finding and matching - is of type Dish. Returns a boolean ⇒ terminal op

(a) anyMatch(Predicate) if (menu.stream()
    .anyMatch(d -> d.isVegetarian())) {
        S.o.p("This menu is somewhat veg friendly
    if any element matches P

(b) allMatch(Predicate) if (menu.stream().allMatch(d -> {d.getCalories < 800)
    if all elements match P

(c) noneMatch(Predicate) ⇒ Opposite of allMatch().

allMatch, noneMatch, anyMatch are short-circuiting operations. These do not need all elements to be processed to produce a result. These can turn an infinite stre into constant size. Other short-circuiting operations include limit, findFirst, findAny. If a match is found, the object is returned otherwise an Optional object is returned. e.g. menu.stream(),

```
    .filter(d -> d.isVegetarian)
    .findAny()
    .ifPresent(d -> S.o.p(d.getName()));
```

void ifPresent(Consumer<T> block) — pass lambda expression
returns an Optional dish
if the value is conta... it is printed

findFirst() is more constraining in II^L so should only be used when ordering is important.

5. Reducing - Terminal op^n that combines all elements of the stream repeatedly produce a single value as result. In functional programming th... is called fold. I, T → BinaryOperator → Optional<T>

classmate
Date 30/01/2017
Page

sum());

a) Summing of elements – int sum = numbers.stream().reduce(0,(a,b)→a+b);

int product=numbers.stream()
.reduce(1,(a,b)→a*b);

initial value; Binary Operator <T>

a=1; b=first member of stream (say 4)

a=a*b ⇒ a = 4x1 = 4

b=second member of stream

then again lambda expression is evaluated

More concise notation : numbers.stream().reduce(0, Integer::sum);

b) Maximum and Minimum –

returns Optional object

overloaded version with no initial value

Optional <Integer> max=numbers.stream().reduce(Integer::max);

(x,y)→(x>y)?x:y;

c) Count :

Count the no. of items present in a menu :

menu.stream().map(d→1).reduce(0,(a,b)→a+b);

Alternatively, long count = menu.stream().count();

In reduce, ~~ad~~ internal iteration is advantageous. For parallelStream,

int sum = numbers.parallelStream().reduce(0,(a,b)→a+b);

The lambda passed to this reduce cant change states. Thus it can be performed in any order as the operation is associative.

Operations like map and filter take an input stream, process each element of the stream and produce zero or one result in the o/p stream. Thus these operations are stateless.

But sum, max, reduce, limit, skip need an internal state to accumulate results. The internal state is small and bounded, do not depend on the stream being processed. Operations like sort() and distinct() take an input stream, process each element to produce ~~zero~~ one result in the o/p stream but to compute, they need the previous history to do their job. Thus these are stateful operations requiring unbounded storage space.

The stream oper^{n} that do not pose an order are easier to ~~for~~ parallelization. Stream poses an encounter order in which each element is operated upon. This depends on both the source of the data and the operations performed on the stream.

NUMERIC STREAMS

There are numeric stream interfaces are provided to reduce the cost of boxing — IntStream, DoubleStream, LongStream through mapToInt(), mapToDouble and mapToLong respectively.

To get the sum of calories — menu.stream().mapToInt(Dish::getCalories).sum();

Other fns are max, min and average.

```
IntStream intStream = menu.stream().mapToInt(Dish::getCalories);
Stream<Integer> streamInt = intStream.boxed();
OptionalInt maxCalories = menu.stream().mapToInt(Dish::getCalories)
                                       .max();
int max = maxCalories.orElse(1);
```

Building Streams

1. Static methods like Stream.of(..., ..., ...); Stream.empty();
   - when there is a semantic reason for a method, it should not be hidden in the interface/class itself
   - utility class but should be put in
2. From arrays Arrays.stream(<array variable>)
   - eg `int[] numbers = {1,2,3,4,5};`
   - `int sum = Arrays.stream(numbers).sum();`
3. From files → using java's NIO (New Nonblocking IO) API
   - No shared states

```
long uniqueWords = 0;
try { Stream<String> lines =
      Files.lines(Paths.get("data.txt"), Charset.defaultCharset()))
  uniqueWords = lines.flatMap(lines -> Arrays.stream(lines.split(" ")))
                     .distinct()
                     .count();
} catch (IOException e) { ... }
```

- each line is a stream element
- generate a stream of words

Creating Infinite Streams

1. Stream.iterate
2. Stream.generate

} → generated streams do not have a fixed size, can be infinite, create values on demand, sensible to put a limit.

Stream.iterate

initial value

UnaryOperater<T>

```
Stream.iterate(0, n -> n+2)
    .limit(10)
    .forEach(x -> System.out.println(x));
```

successively applied to create new values

This code generates first 10 even numbers. iterate() is fundamentally sequential as o/p depends on previous result. – Stream gen. is unbounded.

```
Stream.iterate(new int[]{0,1}, t -> {t[1], t[0]+t[1]})
    .limit(20).map(t -> t[0]).forEach(System.out::println)
    .forEach(t -> System.out.println("("+t[0]+","+t[1]")");
```

prints Fibonacci series

Stream.generate

generate() takes as argument Supplier<T> and does not apply successively on each new value.

```
Stream.generate(Math::random())
    .limit(5).forEach(System.out::println);
```

stateless as the generated values are not recorded for later computations

otherwise the stream would be unbounded