Streams

CPSC 1181 – O.O.

Jeremy Hilliker

Spring 2017

Overview

- Streams
  - Types
  - Sources
  - Intermediate operations
  - Terminal operations
  - Examples

Motivation

- We do things like this a lot:

```
List<String> wordList = . . .;
long count = 0;
for (String w : wordList) {
    if (w.length() > 10) { count++; }
}
```

- Can we do it better?

Inspiration

- Pipes & Filters!
  - A pipeline
  - “aggregate operations” if you are pretentious
- Declarative languages
  - SQL
  - Say what you want to achieve
    - Not how to achieve it
    - “Count the strings longer than 10 characters”
- Advantages:
  - Because we’ve said “what,” not “how”
    - Operations can occur in parallel
    - Operations can be reordered
    - Data can be from anywhere (memory, file, database, etc)

Streams

```
Stream<String> words = wordlist.stream();
long count = words                      // not "how"
      .filter(w -> w.length > 10) // what I want
      .count();                         // what I want
```

Streams

- Are immutable
  - They cannot be changed
  - Operations on them create new stream
  - Advantage: no side effects -> easy to make parallel
- Are lazy
  - Does not store elements
  - Only do the work that is necessary, when it is necessary
  - Can stop work early if we find enough of an answer
    - “do any match?” (stops on the “first” match)
    - “do none match?” (stops on the “first” match)
    - “do all match?” (stops on the “first” non-match)
    - “are there at least 3?” (stops counting at 3)
    - “get me any” (returns the “first” one it finds)

4 Types of Streams

- Stream<T>
  - A template type to stream anything
- IntStream
  - Streams ints
- LongStream
  - Streams longs
- DoubleStream
  - Streams doubles
- The number type streams give us extra terminals: average, sum, & summary stats.

3 types of operations

- A **source**
  - [the pump]
  - Supplier of the data
    - Collection, array, generator function, I/O channel, etc
- Zero or more **intermediate operations**
  - [the filters]
  - Transforms the stream and produces a new one
- A **terminal** operation
  - [the sink]
  - The stream is “closed” on a terminal operation
  - Can no longer be used

Sources

- Collection.stream()
- Collection.parallelStream()
- Stream.generate(Supplier<T> s)
- Stream.iterate(T seed, UnaryOperator<T> f)
- Stream.of(T... values)
- Stream.concat(Stream, Stream)
- IntStream.range(int startInclusive, int endExclusive) [also Long & Double]
- Random.doubles() [also ints() & longs())
- Files.lines()
- Files.list()
- Etc…

Intermediate Operations

- **filter**
- **map**

- distinct
- flatMap
- limit
- peek
- skip
- sorted

- parallel
- sequential
- unordered

Terminal Operations

- allMatch
- anyMatch
- **collect**
- count
- noneMatch
- **forEach**
- findAny
- findFirst
- forEachOrdered
- max
- min
- reduce
- toArray
- average*
- sum*
- summaryStatistics*

Example: filter & count

- How many words are longer than 10 letters?

```
List<String> wordList = . . .;
long count = 0;
for (String w : wordList) {
    if (w.length() > 10) { count++; }
}
// or:
long count = wordlist.stream()
     .filter(w -> w.length > 10)
     .count();
```

- Q: Identify: Source, Intermediate operations, Terminal operation

Example: filter & allMatch

- Are all words longer than 10 characters?

```
for(String w : words) {
        if(w.length() <= 10) {
                return false;
        }
}
return true;
```

```
return words.stream()
        . allMatch(w -> w.length() > 10);
```

Example: map, distinct, & count

- How many unique words, case insensitive?

```
Set<String> w = new HashSet<>();
for(String s : words) {
        w.add(s.toLowerCase());
}
return s.size();
```

```
return words.stream()
        .map(String::toLowerCase)
        .distinct()
        .count();
```

Example: max

- Find the longest word

```
String longest = null;
for(String w : words) {
    if(longest == null ||
            longest.length() < w.length()) {
        longest = w;
    }
}
return longest;
```

```
return words.stream()
            .max(Comparator.comparintInt(String::length))
            .orElse(null);
```

Optional

java.util

Class Optional<T>

java.lang.Object
    java.util.Optional<T>

```
public final class Optional<T>
extends Object
```

A container object which may or may not contain a non-null value. If a value is present, `isPresent()` will return `true` and `get()` will return the value.

Additional methods that depend on the presence or absence of a contained value are provided, such as `orElse()` (return a default value if value not present) and `ifPresent()` (execute a block of code if the value is present).

This is a value-based class; use of identity-sensitive operations (including reference equality (`==`), identity hash code, or synchronization) on instances of `Optional` may have unpredictable results and should be avoided.

Since:

`1.8`

Example: mapToInt & average

- What's the average word length?

```
int sum = 0;
int count = 0;
for(String w : words) {
        sum += w.length();
        count++;
}
double avg = sum / count;
```

```
double avg = words.stream()
        .mapToInt(String::length)
        .average()
        .orElse(Double.NaN);
```

Example: filter & forEach

- Move all the elements that need moving:

```
for(Elem e : elements) {
    if(e.needMove()) {
        e.move();
    }
}
```

```
elements.stream()
        .filter(Elem::needMove)
        .forEach(Elem::move);
```

Example: flatMap & max

- Find the longest word on any line

```
String longest = null;
Pattern d = Pattern.compile("\W+");
for(String l : lines) {
    String[] ws = d.split(l);
    for(String w : ws) {
        if(longest == null ||
            w.length() > longest.length()) {
            longest = w;
        }
    }
}
return w;
```

```
Pattern d = Pattern.compile("\W+");
return lines.stream()
    .flatMap(l -> d.splitAsStream(l))
    .max(Comparator.comparingInt(
            String::length))
    .orElse(null);
```

Example: averagingDouble

- Get average GPA

```
int count = 0;
double sum = 0;
for(Student s : students) {
    count++;
    sum += s.getGPA();
}
return sum / count;
```

```
return students.stream()
    .mapToDouble(Student::getGPA)
    .average()
    .orElse(Double.NaN);

return students.stream()
    .collect(Collectors.averagingDouble(
            Student::getGPA));
```

Example: filter, sorted, collect

- Get all CPSC students with GPA >= 4.0
  - ordered by GPA:

```
List<Student> out = new ArrayList<>();
for(Student s : students) {
    if(s.getGPA() >= 4.0) {
        out.add(s);
    }
}
out.sort((a,b) ->
    return Double.compare(a.getGPA(), b.getGPA());
```

```
return students.stream()
    .filter(s -> s.getGPA() >= 4.0)
    .sorted(Comparator.
        comparingDouble(Student::getGPA))
    .collect(Collectors.toList());
```

Example: filter, anyMatch, collect

- Get all students who have ever failed a class:

```
Set<Student> out = new HashSet<>();
for(Student s : students) {
    for(Course c : s.getCourses()) {
        if(c.failure()) {
            out.add(s);
        }
    }
}
return s;
```

```
return students.stream()
    .filter(s -> s.getCourses().stream()
                .anyMatch(Course::failure))
    .collect(Collectors.toSet());
```

Example: filter, anyMatch, collect

- Get all students, partitioned by GPA 3.0

```
Map<Boolean, List<Student>>
    out = new HashMap<>();
out.put(true, new LinkedList<>());
out.put(false, new LinkedList<>());
for(Student s : students) {
    List <Student> list = out.get(
        s.getGPA() >= 3)
    list.add(s);
}
return out;
```

```
return students.stream()
    .collect(Collectors.partitioningBy(
        s -> s.getGPA() >= 3 ));
```

Example: collect, groupingBy

- Get all Students, grouped by term:

```
Map<Integer, List<Student>> out = new HashMap<>();
for(Student s : students) {
    out.compute(s.getTerm(), (term,ls) -> {
        if(ls == null) {
            List<Student> list = new LinkedList<>();
            list.add(s);
            return list;
        } else {
            ls.add(s);
            return ls;
}
return out;
```

```
return students.stream()
        .collect(Collectors.groupingBy(
                Student::getTerm));
```

Example
collect & groupingBy

- Customers, grouped by Province & City

```
Map<String, Map<String, List<Customer>>> customersByProvCity =
        customers.stream().collect(
        Collectors.groupingBy(Customer::getCity,
                Collectors.groupingBy(Customer::getProv));
```

Recap

- Streams
  - Types
    - 4: Template, int, long, double
  - Sources
  - Intermediate operations
    - filter
    - map
  - Terminal operations
    - collect
    - forEach
  - Examples