COMP6700/2140 Streams

Alexei B Khorev

Research School of Computer Science, ANU

April 2017

Topics

1. What are Streams?
2. From containers to streams
   - collections → streams
   - arrays → streams
3. Generated Streams
4. Stream of Streams and Flat Map
5. From Iterations to Stream Operations
6. Parallel Streams
7. How streaming data can be utilised?

Streams

What is a stream?

[Quoting Maurice Naftalin's Lambda FAQ]

> *A stream is a sequence of values. The package `java.util.stream` defines types for streams of reference values (Stream) and some primitives (IntStream, LongStream, and DoubleStream). Streams are like iterators in that they yield their elements as required for processing, but unlike them in that they are not associated with any particular storage mechanism. A stream is either partially evaluated — some of its elements remain to be generated — or exhausted, when its elements are all used up. A stream can have as its source an array, a collection, a generator function, or an IO channel; alternatively, it may be the result of an operation on another stream. A partially evaluated stream may have infinitely many elements still to be generated, for example by a generator function.*

Why Streams?

- Iterators are good; they represent a data processing paradigm, yet
- They are rather rigid and they do not allow concurrent execution
- If data is large, it may be very memory demanding and require more complex algorithms (out-of-core algorithms, cache-oblivious algorithms *etc*)
- Streams provide an alternative to iterator-based data processing which allow to address some of those problems
- They provide an effective way to include/exclude data elements and transform them via the meta-operations `filter` and `map`
- They also allow to `limit` the number of elements, retain only `distinct` ones and get them `sorted`
- They are *lazy*: they generate values (data elements) upon request instead of storing them all in memory
- Therefore, streams can be only processed once, and do not allow recursive treatment (this limitation can be circumvented with clever tricks)
- Streams can be *parallelised* (there're constrains on type of operations)
- Data processed in streams can be placed into standard container objects, including sophisticated selection and grouping

Stream Creation

If you have a container (form Java's API), its elements can be readily streamed — the default method `java.util.Collection.stream` (and `parallelStream`) can be called on any list or other collection object:

```
// most economical (for memory) reading of a file contents
Path filepath = Paths.get("stephenson_comm_line.txt");
Stream<String> lines = Files.lines(filepath, StandardCharsets.ISO_8859_1));

// can read entire file content into a single string and then break it
String contents = new String(Files.readAllBytes(...);
Stream<String> words = Stream.of(contents.split("[\\P{L}]+"));//split against
                                                              //non-letters
// array can be replaced by varargs
Stream<String> song = Stream.of("gently", "down", "the", "stream");
```

Thus are created streams which give an alternative representation of data residing either inside (already existing collection objects), or outside (lines of text file) of the program.

More Stream Creation

One can convert a collection into an array and back using *JFC* helper class, `java.util.Arrays` and the method `java.util.Collection.toArray`

```
T[] array = ...;     // T must be chosen
List<T> list = ...;
list = Arrays.asList(array);
array = list.toArray(T[]); // the arg T[] is the array to which elements
                           // of list will be copied if there is room
                           // otherwise a new array will be created and
                           // returned (quirky trick meant to save memory)
```

Yet, if one has data in an array already, they can be steamed directly:

```
Arrays.stream(new Integer[] {2, 3, 5, 7, 11, 13}).allMatch(x -> isPrime(x));
```

Note Java still isn't very good with literal arrays:

```
Stream.of({2, 3, 5, 7, 11, 13}).reduce(1, (x,y) -> x*y);
         error: illegal start of expression
```

The situation with literal strings is better, though $\xrightarrow{\text{next slide}}$

(Almost) Iterable Strings

Strings cannot be iterated (*String* does not implement *Iterable*), like, *eg* in *Python*:

```
def count(hist,c):
   hist[c] = hist.get(c,0) + 1
   return hist

freqs = reduce(count,"Ministry of Silly Walks",{})
```

However, through the “default” extension of the interface `java.lang.CharSequence`, a method `chars()` can now be called on a string object directly, generating `IntStream` stream:

```
Map<String,Integer> freqs =
  "Ministry of Silly Walks".chars() // issuing stream of ints
     .mapToObj(c -> Character.valueOf((char)c)) // we need objects!
     .reduce(new HashMap<Character,Integer>(),  // 1st argument: empty Map
       (m,c) -> {m.put(c, m.getOrDefault(c, 0) + 1); return m;},// 2nd: add to it
       (m1,m2) -> {m1.putAll(m2); return m1;}); // 3d: merging [parallelised] maps
```

The `reduce` second and third arguments deal with an element-by-element accumulation into the map and (if the stream were parallelised and merged at the end) combining sub-maps. Two more `reduce` methods are available in *Stream*.

There is *still* considerable price to pay for static type safety!

Computationally generated streams

Streams can be produced by a computation within a program:

```
Stream<String> echos = Stream.generate(() -> "Echo");

Random rand = new Random();

Stream<Integer> ints1 = Stream.generate(() -> rand.nextInt(200) - 100)

IntStream ints2 = new Random().ints(-100, 100); // primitive type stream
ints2.limit(100).forEach(i -> System.out.printf("%d ", i));

IntStream ints = IntStream.range(1,200); // finally, Java's API has range!

Stream<Integer> peano = Stream.iterate(0, i -> i + 1); // who is Peano?

Stream<BigInteger> integers
    = Stream.iterate(BigInteger.ZERO, n -> n.add(BigInteger.ONE));
```

From Streams to Iterables

The stream *raison d'être* is to offer an alternative ("streamy") data processing paradigm to iterations — the availability of creating streams from collections is natural.

If we want to go the opposite way — create an iterable object using an existing stream? Owing to a method `java.util.stream.BaseStream.iterator` (*BaseStream* is a parent interface to `java.util.stream.Stream`), this is possible:

```java
Stream<String> stream = ...;
for (String s : (Iterable<String>)stream::iterator) {
    ...
}
```

(the cast `(Iterable<String>)` is needed because the method reference `stream::iterator` requires a target type)

Merging Substreams

If we have a text which can be broken into words.

```
List<String> wordList = ...;
Stream<String> words = wordList.stream();
```

Define a method which creates a stream of characters extracted from a string (*String* did not get a stream extension to generate a stream yet, will it?):

```
public static Stream<Character> characterStream(String s) {
   List<Character> result = new ArrayList<>();
   for (char c : s.toCharArray()) result.add(c);
   return result.stream();
}
```

Now, if we map the `wordList` with `characterStream`, we shall get a nested stream:

```
Stream<Stream<Character>> wfts = words.map(w -> characterStream(w));
```

This may not be what we need (a uniform stream of all characters in the original order). Instead of using `map`, we should use `flatMap`:

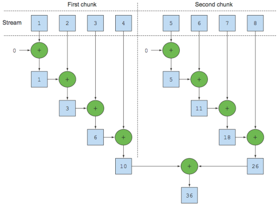
```
Stream<Character> letters = words.flatMap(w -> characterStream(w));
```

Parallel Streams

Some stream operations can be parallelised — a source stream can be split into several "parallel" streams, each processed independently, and at the end merged to produce the final result when a reduce-like or a collect-like operation is present as terminal (more on *terminal* and other type of ops in F5; terminal ops produce a non-stream value like `int`, when something is counted/reduced, or `List`, or `String`, when the data are collected or joined).

For a simple problem of counting a sum of streamed `int`'s (the image on the right is courtesy of Urma *etal* Java 8 in Action):

```java
public static long parallelSum(long n) {
  return Stream.iterate(1L, i -> i + 1)
    .limit(n)
    //turns the stream into parallel
    .parallel()
    .reduce(0L, Long::sum);
}
```



Correct use of parallel streams which does not result in a worser performance requires some care.

Uses of Streams

1. Stream object(s) created (from data existing in memory or from persistent storage)
2. Streams can have structure, *eg* stream elements can be streams, too; Streams can be *flattened* and *merged* (concatenated).
3. Streams can be:
   - processed to remove or retain only part of their elements (`filter`)
   - transformed element-by-element (`map`)
   - sorted and "uniqued" (`sorted`, `distinct`)
   - collected into standard data containers (list, set, map)
   - used (element-by-element) to compute a value or values (`reduce` and `collect`)
4. Once processed, they are exhausted and cannot be re-used (if you intend to use same stream more than once, you need create a *stream supplier*, see F5).

Where to look for this topic in the textbook?

- Hortsmann's Core Java for the Impatient, Ch. 8.1, 8.2
- Oracle's Java Tutorial chapter on Aggregate Operations
- The `java.util.stream` package API documentation is succinct and precise (clearly, more care has been taken of writing Java docs lately ☺)
- Maurice Naftalin's Lambda FAQ has a section "Idioms and Techniques", many entries in which deal with issues of stream creation, conversion and operations (very useful)
- Benjamin Winterberg's Java 8 Stream Tutorial