Practical Concurrent and Parallel Programming VI

Streams & Parallel Streams

IT University of Copenhagen

Friday 2020-10-02

Kasper Østerbye

Starts at 8:00

Plan for today

Follow up on last week

A bit of history

Streams

Parallel streams

In other languages

Some history

Mapping over a list

- Lisp (1958) had lambda expressions and higher order functions. The notion of `map` - applying a lambda expression to each element in a list to get a new list has been around since the beginning of programming.
- Smalltalk (1980) has a large collection library, and also has `map` and `filter` (called collect and select).

Lazy eval

Lazy evaluation assigns an *expression* to a variable. When the value of the variable is read, the expression is evaluated.

```
x = 10+y; // corresponds to x = ()-> 10+y.eval()
println( x ); // corresponds to print( x.eval() ) - printing the value of 10+y
```

The earliest usage is in ALGOL 60. Here it is not used for general variables, but for only parameters. It was named *Jensens Device* after *Jørn Jensen* who worked at Regnecentralen in the early 1960 & 70s.

More history - lazy streams

The language Icon (1977-2018) was build around lazy streams. All expressions produced stream results.

```
readline = ("bingo" | "banko")
```

This is a boolean expression which is true if any line in input is either "bingo" or "banko".

Once a match is found, no further elements are read, and the streams do not produce any more elements.

Functional programming

Many functional programming languages use lazy streams. Haskell seems to be one of the earliest to go all in, but most functional programming languages support lazy streams

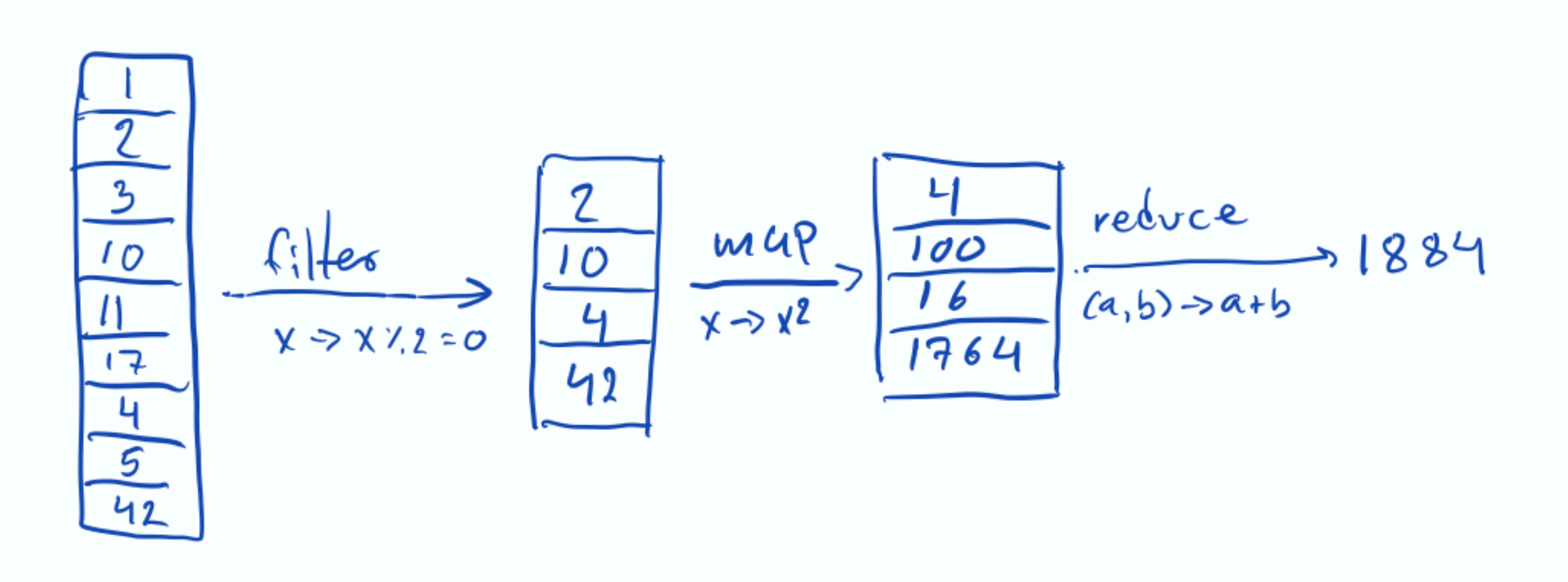
Mainsteam streams

C# 3.5 (2007)

- First mainstream programming language to support lambda expressions and lazy streams
- It was a revolution, and earned Microsoft a lot of honor in academic circles as it truely is well designed

Java 8 (2014)

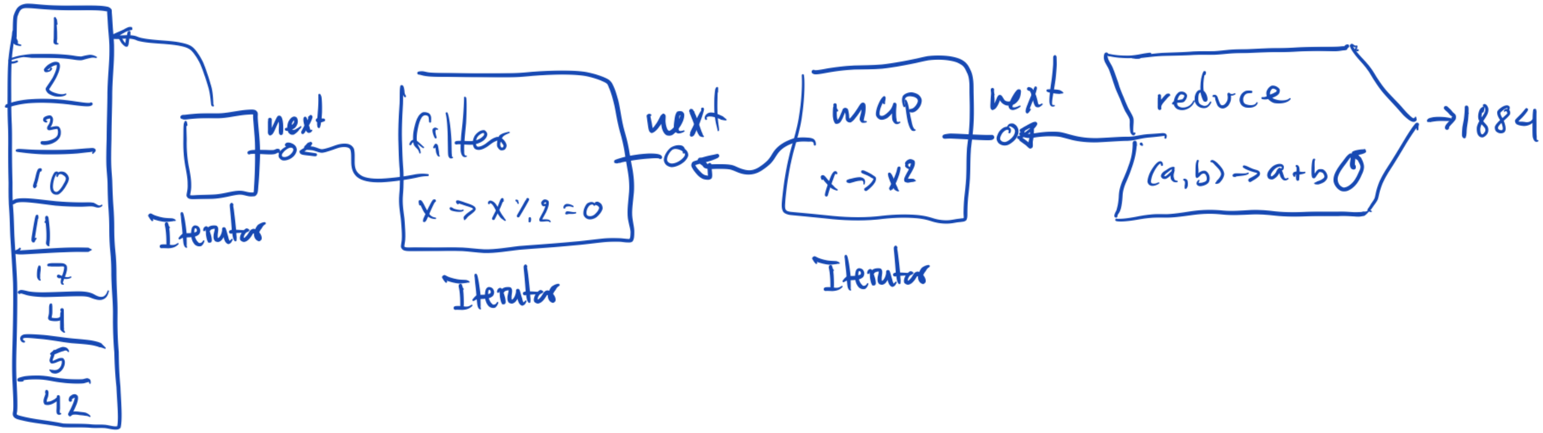
Filter, map, reduce



Eager model

- Not supported in Java (but lisp and smalltalk for example)
- New collections are generated at each link in the chain.
- Not used in modern frameworks as it potentially can give many copies of the same data
- Does not support *infinite streams*.

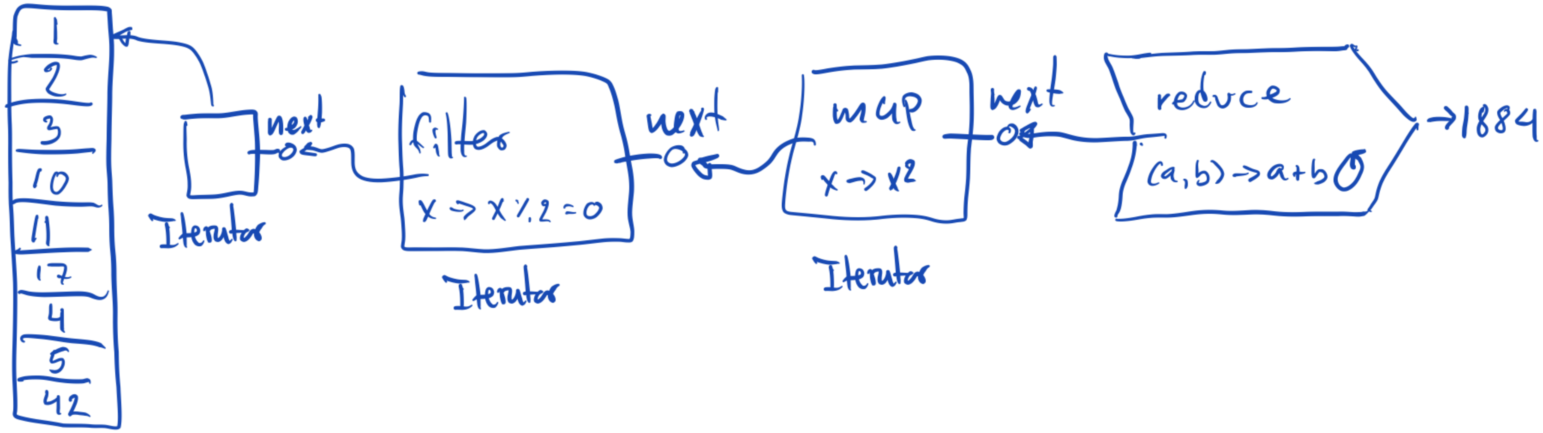
Lazy or Iterator model



Lazy/Iterator model

- Used in Java and C# - Python?
- No intermediate collections generated
- Supports *infinite streams*
- Can be difficult to re-start (you cannot (easily) store the chain in a variable and use it several times)

Stream pipeline



There are three different types of elements:

- sources (arrays, collections, IO, generators)
- intermediate operations (transforming one stream into an other - filter, map,…)
- terminal operations (count, sum, forEach, …)

The streams are lazy, driven by the terminal

Sources

- Input (files or network) - Here from `BufferedReader`

Stream<String>	**lines**() Returns a `Stream`, the elements of which are lines read from this `BufferedReader`.

- The `Arrays` class has a number of utilities - for example: `Arrays.stream(array);`

- All java collections has a `stream()` method which is a stream over its elements

- Stream.of("Huey", "Dewey" "Louie") - returns a stream of the three strings
  .

Example - CSV reader

CSV - comma separated values - used to store tables in text In Denmark we use `;` to separate because `,` is used as decimal character in numbers

Assume we have a file of addresses:

```
Jomfru Ane gade 17; 9000; Ålborg
Kannikegade 10; 8000; Aarhus
```

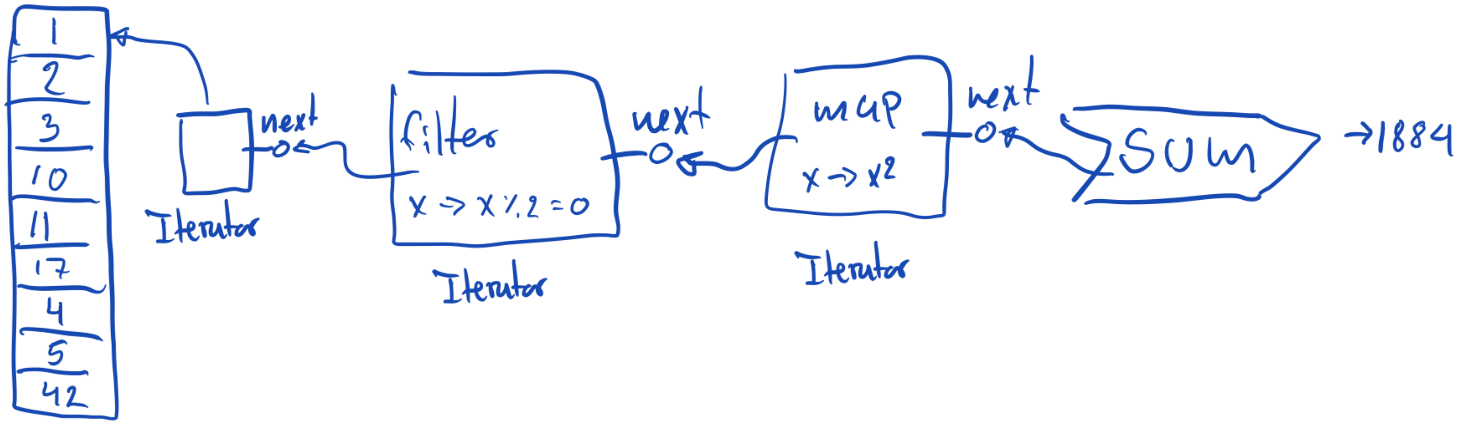
```
bufferedReader.lines()
.map(line -> line.split(";"))
.map(elements -> new Address(elements[0], elements[1], elements[2]))
.filter( addr -> addr.getZip() >= 9000 )
.forEach( Address::toString )
```

Intermediate operations

- filter - lambda returns a boolean, if the boolean is true the element is included in the output stream
- map - the lambda computes a value. The output stream consists of computed values.
- limit(n) - returns a stream of the first n elements
- skip(n) - returs a stream without the first n elements
- distinct - removes duplicates from stream
- peek - copies the elements to output, but executes lambda for each element (can be used for debugging)
- sorted - returns a stream with the elements sorted

Terminal operations

- Number streams has min, max, sum, average, count



- Number streams even has `summaryStatistics()`

- allMatch - returns true if the lambda is true for all elements - sort of ∀

- anyMatch - returns true if the lambda is true for a element () - sorf of ∃
- noneMatch - returns true if the lambda is false for all elements - sort of ∄

- findAny and findFirst returns an Optional (as the stream might be empty).

- forEach - executes the lambda for all elements in the stream

Reduce

Mostly you will not need this, but use one of the predefined ones (sum, average)

```
stream.reduce( (a,b) -> a+b*b)
```

will compute the sum of squares

the result is computed like this:

```
     boolean foundAny = false;
     T result = null;
     for (T element : this stream) {
         if (!foundAny) {
             foundAny = true;
             result = element;
         }
         else
             result = theLambda.apply(result, element);
     }
     return foundAny ? Optional.of(result) : Optional.empty();
```

Notice: It returns an `Optional` as the stream might be empty

Reduce with intial value

```
stream.reduce(0, (a,b) -> a+b*b)
```

which is computed as:

```
     T result = identity;
     for (T element : this stream)
         result = theLambda.apply(result, element)
     return result;
```

Notice: No optional - as we give an initial value.

Java type in reduce

That is, one **cannot** do this:

```
List<Integer> list = Arrays.asList(1,2,3,10,11,17,4,5,42);
System.out.println("Reduced to: " + list.stream()
  .filter( x -> x%2 == 0)
  .map( x -> x*x )
  .reduce(LocalDate.now(), (a,b) -> a.plusDays( b ) ));
```

Which to me looks quite normal….

Stream to array

Often one want to produce a collection of the elements in a stream.

- `toArray()` - returns an `Object[]` - that is, untyped elements
- `toArray(IntFunction<A[]> generator)` - returns an array of type A The parameter looks a bit odd, but what is ask for is a lambda which takes a size and returns an array of type A of that size.

  ```
  intStream.filter( i -> i.isPrime()).toArray( n -> new Integer[n]);
  ```

  which can be written as

  ```
  intStream.filter( i -> i.isPrime()).toArray(Integer[]::new);
  ```

Stream to Lists

- `toList()` - yes, finally something simple

Stream to ...

There is a general mechanism:

```
stream.collect(aCollector)
```

The java class `Collectors` has many different factory methods for aggregating the elements in a stream.

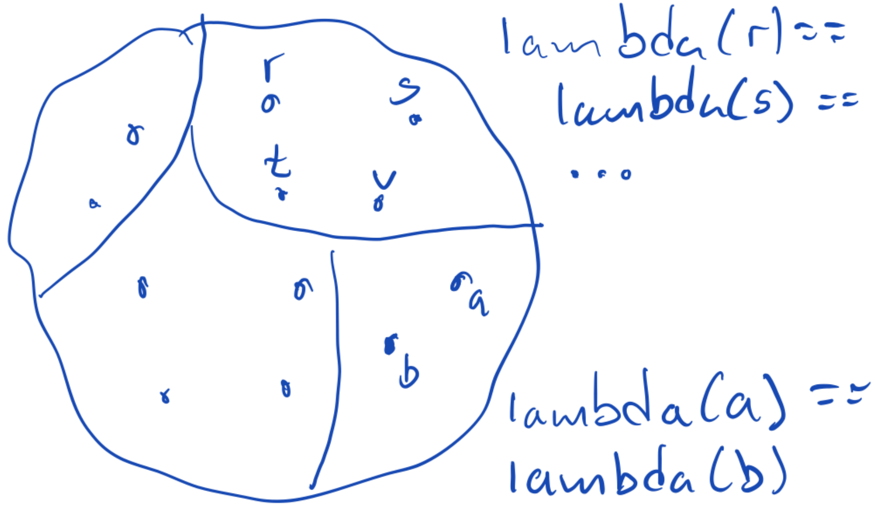
The class has several examples as well.

Reduce and collect

"The normal reduction is meant to **combine two immutable values** such as int, double, etc. and produce a new one; it’s an immutable reduction. In contrast, the **collect method is designed to mutate a container to accumulate the result** it’s supposed to produce."

It is possible to write your own collectors, but in nearly all cases you can depend on those declated in `Collectors`(https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html)

Grouping - principle



each group is represented by a value of the lambda, and all the elements which gave that value
(Sorry for the round stream, was just easier to draw)

The full grouping is a `Map`.

`Map<K, List<T>>`, where K is the type of the results of the lambda, and T is the type of the values in the stream.

Grouping example

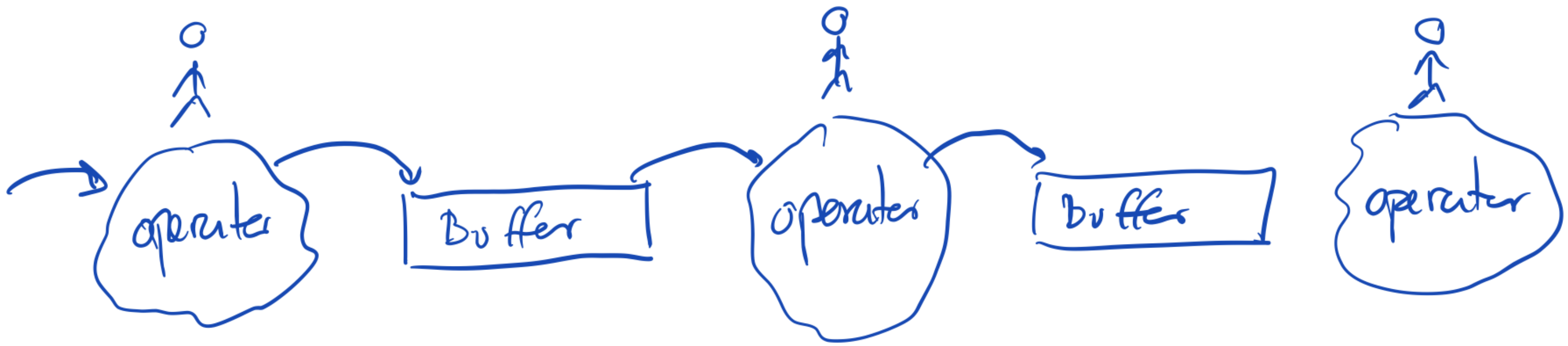
```
Map<Integer,List<Integer>> groupings = Stream.of(1,2,3,10,11,17,4,5,42)
  .collect(Collectors.groupingBy(i-> i / 5)); // integer division by 5

  for (int key: groupings.keySet()){
    System.out.format("Key: %d with list: %s\n",
      key, groupings.get(key).toString());
  }
```

Prints:

```
Key: 0 with list: [1, 2, 3, 4]
Key: 1 with list: [5]
Key: 2 with list: [10, 11]
Key: 3 with list: [17]
Key: 8 with list: [42]
```

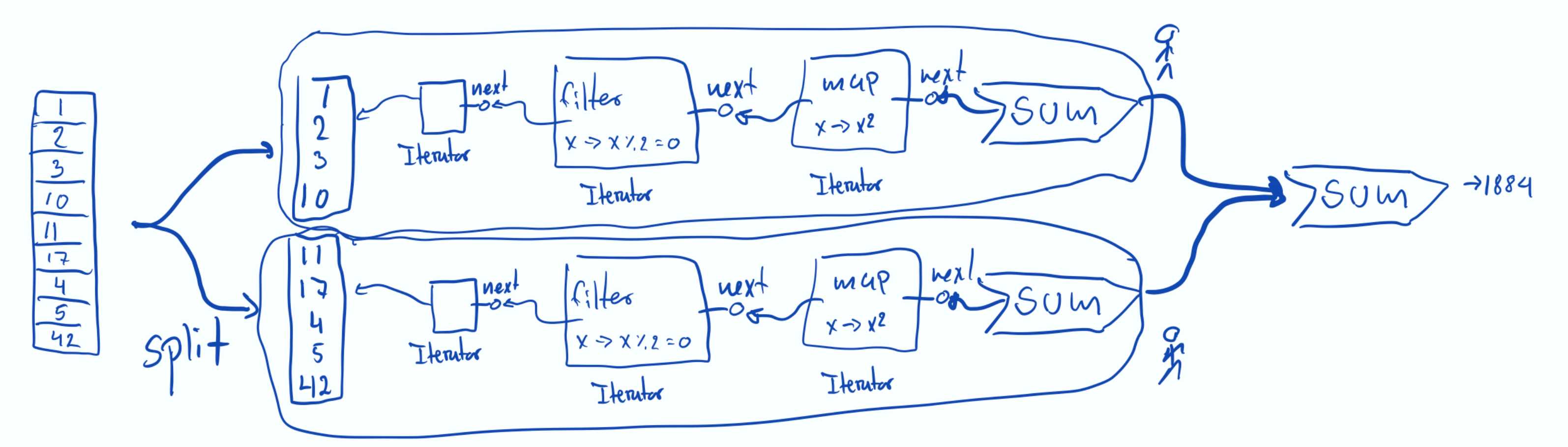
Parallelism and streams I

Strategy 1 - parallel pipelining - (not used):



Parallelism and streams II

Strategy 2 - splitting the iterator - (used):



In java

```
anyStream.parallelStream().allAsBeforeStillWorksButIsNowDoneInParallel
```

A special case is with grouping:

```
Map<Person.Sex, List<Person>> byGender =
    personStream
        .collect(
            Collectors.groupingBy(Person::getGender));
```

The following is the parallel equivalent:

```
ConcurrentMap<Person.Sex, List<Person>> byGender =
    personStream
        .parallelStream()
        .collect(
            Collectors.groupingByConcurrent(Person::getGender));
```

C

- Java streams are pretty much modeled after C#
- C# has a different kind of type parameters - allowing them to have streams of primitive types
- C# has *extension methods*, which in many cases gives a more concise syntax
- C# has *anonymous classes*, which work well with `map` (called *select* in C#)

Mongo aggregation

- the source is collection of json documents
- filter allows to look down into the json tree
- map allow to pick out subtrees and substitute elements
- sum, average,… all exists
- a special `unwind` operator can create copies of a json node
  for example, if a person has three emails
  `emails: ["email1", "email2", "email3"]`
  `unwind` will produce three persons, each with only one email.

It is a bit unclear how parallelism is used except on large scale (sharding).

Python, Ruby, Go, Rust, Kotlin, Scala, F#,…## Left as an exercise

- The devil is in detail