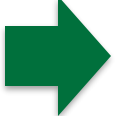


CS 10:

Problem solving via Object Oriented Programming

Streams

Agenda

- 
1. Streaming data
 2. Java streams

Streams allow us to process things “as they come”

Stream movie vs. file

	Stream (Netflix)	File (Movie on DVD)
Data production	Arrives as produced	Pre-produced

Streams allow us to process things “as they come”

Stream movie vs. file

	Stream (Netflix)	File (Movie on DVD)
Data production	Arrives as produced	Pre-produced
Data processing	As it arrives	All available, read as desired

Streams allow us to process things “as they come”

Stream movie vs. file

	Stream (Netflix)	File (Movie on DVD)
Data production	Arrives as produced	Pre-produced
Data processing	As it arrives	All available, read as desired
Synchronization	Keep producers and consumers in sync	No need for synchronization

Streams allow us to process things “as they come”

Stream movie vs. file

	Stream (Netflix)	File (Movie on DVD)
Data production	Arrives as produced	Pre-produced
Data processing	As it arrives	All available, read as desired
Synchronization	Keep producers and consumers in sync	No need for synchronization
Memory use	Not all in memory	All in memory (or disk)

Streams allow us to process things “as they come”

Stream movie vs. file

	Stream (Netflix)	File (Movie on DVD)
Data production	Arrives as produced	Pre-produced
Data processing	As it arrives	All available, read as desired
Synchronization	Keep producers and consumers in sync	No need for synchronization
Memory use	Not all in memory	All in memory (or disk)
Length	Can be infinite	Limited

Streams allow us to process things “as they come”

Stream movie vs. file

	Stream (Netflix)	File (Movie on DVD)
Data production	Arrives as produced	Pre-produced
Data processing	As it arrives	All available, read as desired
Synchronization	Keep producers and consumers in sync	No need for synchronization
Memory use	Not all in memory	All in memory (or disk)
Length	Can be infinite	Limited
Fast forward/reverse	Hard	Easy

Stream operations can be chained together to form a pipeline

Unix pipeline example

```
cat USConstitution.txt | tr 'A-Z' 'a-z' | tr -cs 'a-z' '\n' | sort | uniq | comm -23 - dictionary.txt
```

Pipeline



1. `cat` outputs contents of file
2. Pipe (`|`) passes output to next command
3. `tr` translates to lower case
4. `tr -cs` translates non-characters to new lines
5. `sort` puts words in alphabetical order
6. `uniq` removes duplicates
7. `comm` compares pipeline with another file, outputs only lines not in `dictionary.txt` (probably means word is misspelled)

Key points:

- One stage produces output the next stage consumes
- Operations form a “pipeline”

Agenda

1. Streaming data

 2. Java streams

Streams are a sequence of elements from a source that supports aggregate operations

Sequence of elements

- A stream provides an interface to a sequenced set of values of a specific element type
- Streams don't actually store elements; they are computed on demand; they don't change Source Object

Source

- Streams consume from a data-providing source such as collections, arrays, or I/O resources such as a web service streaming stock quotes

Aggregate operations

- Streams support SQL-like operations and common operations from functional programming languages, such as filter, map, reduce, find, match, sorted, and others

Two characteristics of Streams make them different from iterating over collections

Streams vs. iterating collections

1. Pipelining

- Many stream operations return a stream themselves
- Allows operations to be chained to form a larger pipeline
- Enables optimizations:
 - Short-circuiting – stop evaluation once you know the result
 - Laziness – wait to evaluate expressions until needed (sometimes can skip evaluation of items not needed)
 - We will see examples shortly

2. Internal iteration

- In contrast to collections, which you explicitly iterate yourself, stream operations do the iteration behind the scenes for you

There are two types of operations, intermediate and terminal

Types of operations

Terminal

Description

- Close a stream pipeline
- Produce a result such as a List or Integer (any non-stream type)

Examples:

- `collect(toList())`
- `count`
- `sum`

There are two types of operations, intermediate and terminal

Types of operations

Terminal

Description

- Close a stream pipeline
- Produce a result such as a List or Integer (any non-stream type)

Examples:

- `collect(toList())`
- `count`
- `sum`

Intermediate

- Output is a stream object
- Can be chained together into a pipeline
- “Lazy”, do not perform any processing until necessary

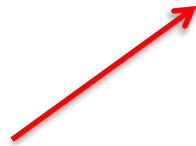
- `filter`
- `sorted`
- `map`
- `limit`
- `distinct`

Common Stream operations

.forEach

Iterate over each element of the Stream

```
//output hi and there (one per line)  
Stream.of("hi", "there") //stream of two strings  
    .forEach(System.out::println); //call println for each string
```



Double colon (::) means call method on right, using Class on left

Each item in Stream passes down pipeline of operations one at a time

First "hi" passed to second line, then "there" is passed to second line

Here both strings in stream are printed

Common Stream operations

.forEach

Iterate over each element of the Stream

```
//output hi and there (one per line)
Stream.of("hi", "there") //stream of two strings
    .forEach(System.out::println); //call println for each string
```

.map

Map each element to a corresponding result

```
//output 1 to 9 squared (1,4,9,16,25,36,49,64,81) one per line
IntStream.range(1,10) //integers in range 1...9
    .map(n -> n*n) //map n to n2
    .forEach(System.out::println); //call println for each integer
```

IntStream produces a Stream of Integers

Range is inclusive of start, exclusive of end

First 1 passed down to second line

1 squared in *map* command on second line and then passed to third line

1 printed as parameter to System.out::println on third line

Next 2 passed down, squared and printed

...

Notice there is no explicit iteration over Stream items

Common Stream operations

.forEach

Iterate over each element of the Stream

```
//output hi and there (one per line)
Stream.of("hi", "there") //stream of two strings
    .forEach(System.out::println); //call println for each string
```

.map

Map each element to a corresponding result

```
//output 1 to 9 squared (1,4,9,16,25,36,49,64,81) one per line
IntStream.range(1,10) //integers in range 1...9
    .map(n -> n*n) //map n to n2
    .forEach(System.out::println); //call println for each integer
```

.filter

Eliminate elements based on a criteria

```
//output even numbers 1 to 9 tripled (6,12,18,24) one per line
IntStream.range(1, 10) //integers in range 1...9
    .filter(i -> i%2 == 0)
    .map(i -> i*3)
    .forEach(System.out::println);
```


- Only even numbers pass *filter* on second line
- Odd numbers do not make it to *map* on third line, Java doesn't waste time tripling odd numbers ("lazy")

Common Stream operations

.limit

Reduce the size of the Stream

```
//output first three items  
IntStream.range(1, 10) //exclusive of 10, so 1..9 here  
            .limit(3) //stop after three items  
            .forEach(System.out::println);
```



Limit on second line stops pipeline once limit reached

Items 4...9 never evaluated because pipeline stop early (short circuits)

Common Stream operations

.limit

Reduce the size of the Stream

```
//output first three items  
IntStream.range(1, 10) //exclusive of 10, so 1..9 here  
    .limit(3) //stop after three items  
    .forEach(System.out::println);
```

.sorted

Sort the Stream

```
//words sorted alphabetically  
List<String> words = Arrays.asList("the", "quick", "brown", "fox");  
words.stream() //Stream of words  
    .sorted() //sort words  
    .forEach(System.out::println); //brown, fox, quick, the
```

List can be a stream source

Can provide own Comparator

If Object has `compareTo()`, can use that (can also reverse with `Comparator.reverseOrder()`)

Must wait for all input before proceeding

Common Stream operations

.limit

Reduce the size of the Stream

```
//output first three items
IntStream.range(1, 10) //exclusive of 10, so 1..9 here
    .limit(3) //stop after three items
    .forEach(System.out::println);
```

.sorted

Sort the Stream

```
//words sorted alphabetically
List<String> words = Arrays.asList("the", "quick", "brown", "fox");
words.stream() //Stream of words
    .sorted() //sort words
    .forEach(System.out::println); //brown, fox, quick, the
```

Also available:

- toSet()
- toMap()

Collectors

Combine results into a collection such as a List or String

```
List<String> strings = Arrays.asList("abc", "defg", "");
List<String> filtered = strings.stream() //Stream of words
    .filter(string -> !string.isEmpty()) //filter empty
    .collect(Collectors.toList()); //return List
```

Lazy computation and short circuiting save time by not evaluating all data

Short circuiting

```
5 public class StreamTest {
6
7     public static void main(String[] args) {
8         List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
9         List<Integer> twoEvenSquares = numbers.stream()
10            .filter(n -> {
11                System.out.println("filtering " + n);
12                return n % 2 == 0; //returns true if n is even
13            })
14            .map(n -> {
15                System.out.println("mapping " + n);
16                return n * n; //square n
17            })
18            .limit(2) //only return the first two squared evens
19            .collect(Collectors.toList()); //save in List
20        System.out.println(twoEvenSquares);
21    }
22
23 }
```

Square the even numbers in the Stream

**First 1 starts down pipeline
Its not even, so filtered out
Then 2 starts down pipeline
It passes all the way through
Map computes only on those items that reach its level
“Lazy” evaluation – only compute value when needed, don’t compute if not needed**

Stop once two items have been through the pipeline – “Short circuit”

Problems | Javadoc | Declaration | Console | Debug | Expressions | Error Log | Console | Call Hierarchy
<terminated> StreamTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Feb 24, 2018, 5:25:09 PM)

```
filtering 1
filtering 2
mapping 2
filtering 3
filtering 4
mapping 4
[4, 16]
```

- **3 filtered out, 4 goes through**
- **Numbers > 4 not evaluated because pipeline stops when limit reached**
- **“Short circuit” saves execution time by stopping early**
- **NOTE: can’t short circuit sorting, need all elements in place in order to sort**

Example: Get IDs of credit card Grocery transactions sorted by amount spent

- Given list of transactions on a credit card
- Extract purchases of Groceries
- Sort Grocery purchases by amount spent
- Return ID of Grocery transactions

Create a Transaction Object will hold details about credit card purchases

Transaction.java

```
6 public class Transaction implements Comparable<Transaction>{
7     protected Integer id;
8     protected String type;
9     protected Double amount;
10
11     public Transaction(Integer id, String type, double amount) {
12         this.id = id;
13         this.type = type;
14         this.amount = amount;
15     }
16
17     public int getId() {
18         return id;
19     }
20
21     public String getType() {
22         return type;
23     }
24
25     public Double getAmount() {
26         return amount;
27     }
28
29     public int compareTo(Transaction t1) {
30         return (int)Math.signum(amount-t1.getAmount());
31     }
32
33     public String toString() {
34         return getId() + "," + getType() + "," + getAmount();
35     }
36 }
```

Start with a Class that tracks a single purchase made on a Credit Card:

- **Has transaction ID**
- **Type (e.g., groceries, fuel, beer)**
- **Amount (monetary amount spent on this transaction)**
- **Getters for instance variables**

Also has *compareTo()* for sorting and *toString()* for printing

The traditional approach involves several iterations over transaction data

Transactions on Credit Card

ID	Type	Amount
123	Fuel	33.33
124	Groceries	120.12
125	Beer	175.75
126	Groceries	152.52
127	Groceries	12.12
...



Traditional steps:

1. Extract Grocery purchases from other purchases in *transactions* List
2. Sort Grocery purchases by amount spent
3. Get IDs of top purchases

Assume several Transaction objects have been loaded into List called *transactions*

The traditional approach involves several iterations over transaction data

TransactionList.java

```
public static ArrayList<Integer> getTransactionIDTraditional(ArrayList<Transaction> transactions, String type) {  
    //extract transactions of type matching parameter type  
    ArrayList<Transaction> list = new ArrayList<>(); //List of transactions matching type parameter  
    for (Transaction t : transactions) {  
        if (t.getType().equals(type)) {  
            list.add(t);  
        }  
    }  
  
    //sort by amount descending  
    list.sort((t1,t2) -> t2.getAmount().compareTo(t1.getAmount()));  
  
    //get transaction IDs  
    ArrayList<Integer> ids = new ArrayList<>();  
    for (Transaction t: list) {  
        ids.add(t.getId());  
    }  
    return ids;  
}
```

Extract Grocery items from all transactions

Explicitly iterate over collection two times (plus a sort)

Sort by descending value

Create two different Lists during process

Extract transaction IDs

```
public static void main(String[] args) {  
    //create transaction list and add some transactions of vaying types and amounts  
    ArrayList<Transaction> transactions = new ArrayList<>();  
    transactions.add(new Transaction( id: 123, type: "Fuel", amount: 33.33));  
    transactions.add(new Transaction( id: 124, type: "Groceries", amount: 120.12));  
    transactions.add(new Transaction( id: 125, type: "Beer", amount: 175.75));  
    transactions.add(new Transaction( id: 126, type: "Groceries", amount: 152.52));  
    transactions.add(new Transaction( id: 127, type: "Groceries", amount: 12.12));  
    //extract transactions of Groceries, sort by amount spent, and get transaction IDs  
    System.out.println("Traditional method: " + getTransactionIDTraditional(transactions, type: "Groceries"));  
}
```

Add a number of Transactions to transactions ArrayList

Java's Streams do the iteration for us

TransactionList.java

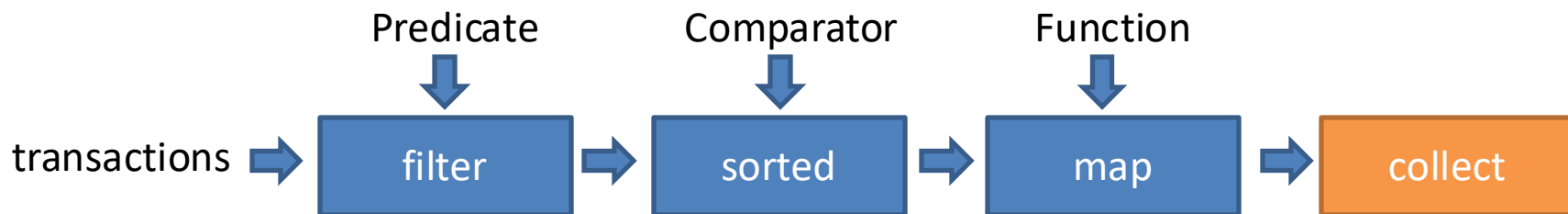
```
public static List<Integer> getTransactionIDStream(ArrayList<Transaction> transactions, String type) {  
    List<Integer> list = transactions.stream() //user transactions ArrayList as source  
        .filter(t -> t.getType().equals(type)) //filter on type  
        .sorted(Comparator.reverseOrder()) //sort using compareTo, but reversed  
        .map(Transaction::getId) //extract IDs  
        .collect(Collectors.toList()); //save in List  
    return list;  
}
```

Use *transactions* ArrayList as stream Source

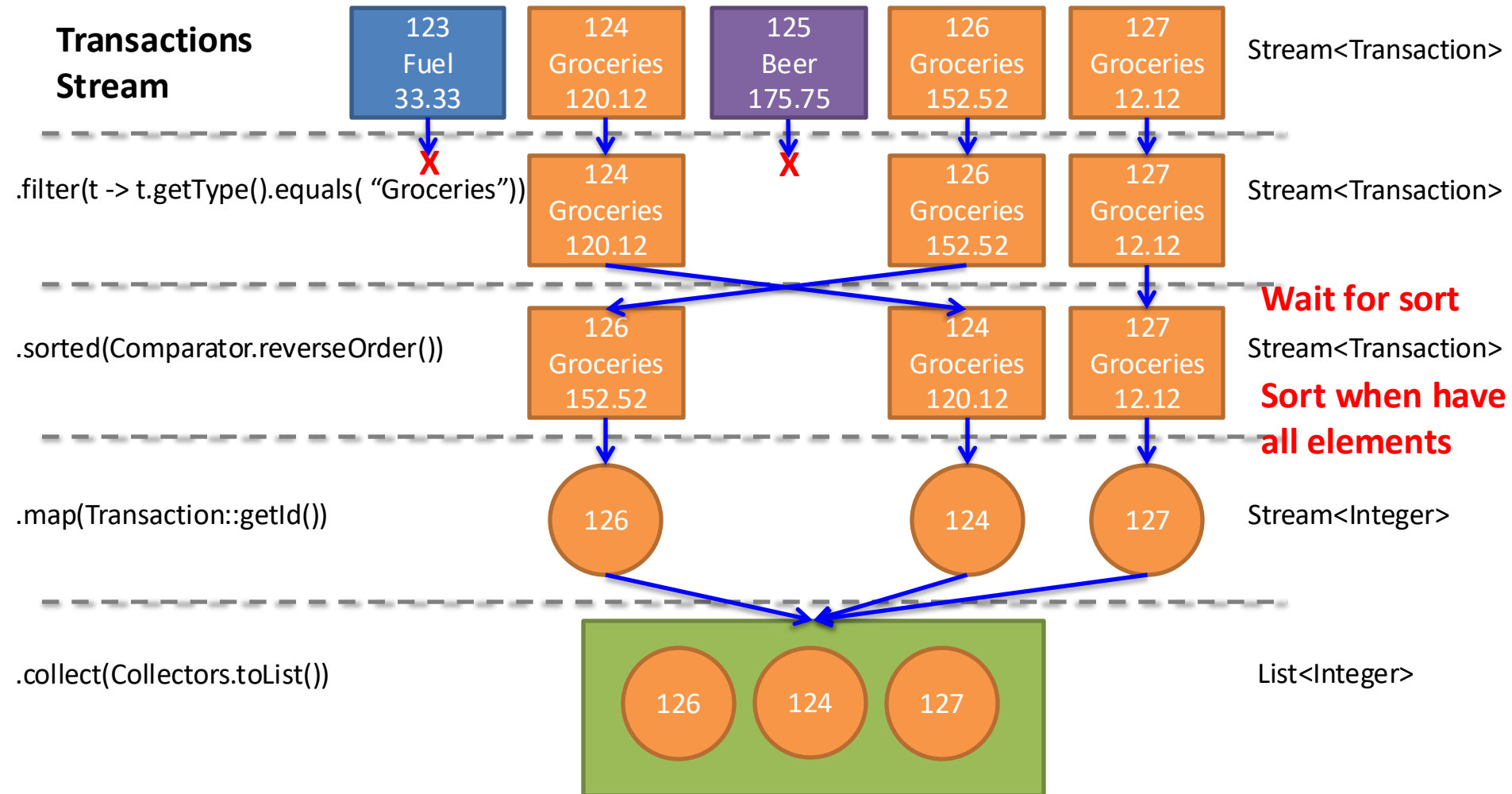
- **Filter on type (groceries)**
- **Sort by amount in reverse order**
- **Extract IDs with map**
- **Return List**

Stream handles implicit iteration for us

Pipeline



Graphical depiction of grocery transaction example



More examples in code for today

StringStreams.java

1. Initiate a stream with a fixed list of strings, terminate it by printing each out. Note the Java 8 syntax for passing a defined method, here the `println` method of `System.out`, which takes a string and returns nothing, as appropriate for termination here.
2. Now we have an intermediate operation, consuming a string and produces a number (its length), passing the `String` member function `length` to do that.
3. A different intermediate, here a static method in this class, which consumes a string and produces a transformed string.
4. The intermediate passes forward only some of the things it gets, discarding those that don't meet the predicate. It uses an anonymous function as we discussed in comparators and events.
5. Other predefined intermediates process the stream to sort it, eliminate duplicates, etc. Some of these can take arguments (e.g., how to sort).
6. A reimplement of the frequency counting stuff from info retrieval, now letting streams do all the work. "Collector" terminal operations collect whatever is emerging from the stream, into a list, set, map, etc. Here we collect into a map, from word to count. The first argument is a method to specify for each object a value on which to group (things with the same value are grouped). Here we group by the word itself, so all copies of the word get bundled up. The second argument then says how to produce a value from the group; here, by counting.
7. Similar, but now grouping by the first letter in the word.
8. Assuming we already have a list of words, now we want to count the letter frequencies. (For illustration, this doesn't count whitespace frequencies, as the words are pre-extracted.) Split each word into characters. But now we've got a stream of arrays of characters, and we want just a single stream of characters. So we make a stream of streams (characters within words), and "flatten" it into a single stream (characters) by essentially appending the streams together.
9. Same thing could come directly from a file, producing a stream of lines that we flatten into a stream of words. Note another intermediate operation keeps only the first 25 it gets.
10. A new final operation counts how many things ultimately emerged from the stream.
11. A comparator for sorting.
12. Partway through, we convert from a generic `Stream` to a specialized `DoubleStream` that deals with double values (not boxed `Double` objects) and lets us do math. Interestingly, the `average` operation recognizes that it could be faced with an empty stream to average. Rather than throwing an exception, it uses the `Optional` class to return something that may be a double or may be null. We could test, but here, just force it to be a double (an exception will be thrown if it isn't).

More examples in code for today

NumberStreams.java

1. Rather than enumerating explicit objects to initiate a stream, we can implicitly enumerate numbers with a range. (Might be familiar from other languages...). Note that this is the specialized `IntStream`, working on raw int values.
2. And we can do appropriate intermediate processing of the numbers.
3. Illustrates the very important general stream processing pattern `reduce` (the other keyword in the map-reduce architecture; we've already done plenty of mapping). The idea is to "wrap up" all the elements in a stream, pair-by-pair. `Reduce` takes an initial value and a function to combine two values to get a result. So `sum` essentially starts at 0, adds that to the first number, adds that result to the second number, etc. Importantly, though, if the operation is associative (doesn't matter where things are parenthesized), it need not be done sequentially from beginning to end, but intermediate results can be computed and combined. That's key in parallel settings.
4. See how general `reduce` is? Could also combine strings with appending, etc.
5. As mentioned, streams only evaluate something when there's a need to. It's like the demand comes from the end of the stream, and that demand propagates one step up asking to produce something to be consumed, and so forth. Since there's a limit of 3 things being produced, the demand for the rest of the range never comes, and the range isn't fully produced.
6. An infinite stream, with the `iterate` method starting with some number and repeatedly applying the transform to get from current to next. So produce 0, from 0 `iterate` to 1 and produce it, from 1 to 2, from 2 to 3, etc. Since limited to 10, the whole iteration isn't realized (fortunately!).
7. Exponentially increasing steps.
8. Filling the stream by generating random numbers "independently" each time.
9. Requesting parallel processing of a stream is as simple as inserting the method. Whether or not that's a good idea, and how it will play out, depends very much on the processing. Here we do have a bunch of independent maps and filters, and as discussed above, reducing with an associative operation (`sum`) can be done in parallel. Sorting would be a bottleneck, for example. Note from print statements that the stuff is going on in non-sequential order.
10. Parallel beats sequential on my machine in this non-scientific test.

