CS 10: Problem solving via Object Oriented Programming

Winter 2017

Tim Pierson
260 (255) Sudikoff

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”

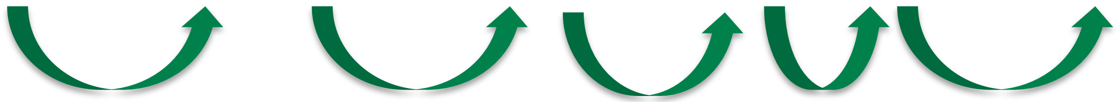
Stream movie vs. file

	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

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 - biggerSorted.txt
```



1. `cat` outputs contents of file
2. Pipe (‘|’) passes output to next command
3. `tr` translates to lower case
4. `tr` translate non-characters to new lines
5. `sort` puts words in order
6. `uniq` removes duplicates
7. `com` compares pipeline with another file, outputs only lines not in biggerSorted.txt

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

Source

- Streams consume from a data-providing source such as collections, arrays, or I/O resources

Aggregate operations

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

http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

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 certain optimizations, such as laziness and short-circuiting

2. Internal iteration

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

Example: Make sorted list of transaction IDs from collection of bank transactions

Usual approach

```
List<Transaction> groceryTransactions = new Arraylist<>();
for(Transaction t: transactions){
  if(t.getType() == Transaction.GROCERY){
    groceryTransactions.add(t);
  }
}
Collections.sort(groceryTransactions, new Comparator(){
  public int compare(Transaction t1, Transaction t2){
    return t2.getValue().compareTo(t1.getValue());
  }
});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions){
  transactionsIds.add(t.getId());
}
```

Get Grocery items from all transactions

Sort by value

Get transaction IDs

Example: Make sorted list of transaction IDs from collection of bank transactions

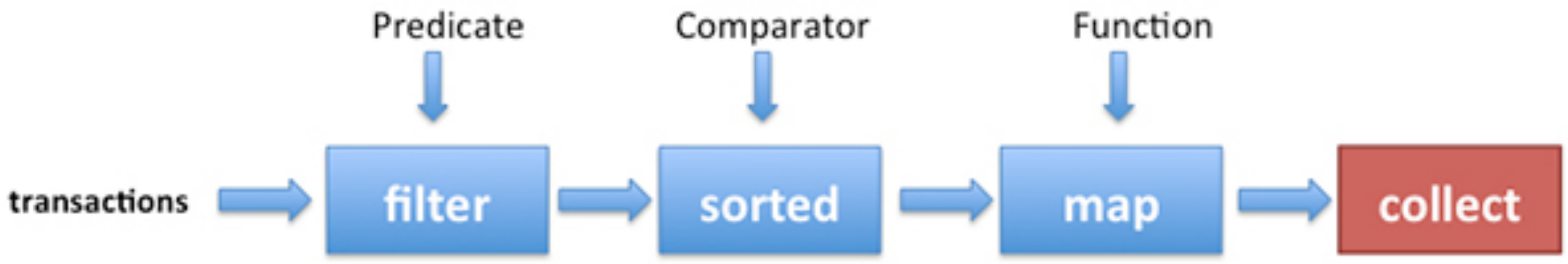
Using Java streams

```
List<Integer> transactionsIds =
   transactions.stream()
          .filter(t -> t.getType() == Transaction.GROCERY)
          .sorted(comparing(Transaction::getValue).reversed())
          .map(Transaction::getId)
          .collect(toList());
```

Double colon means call this function

Data passes to next stage in pipeline

Pipeline



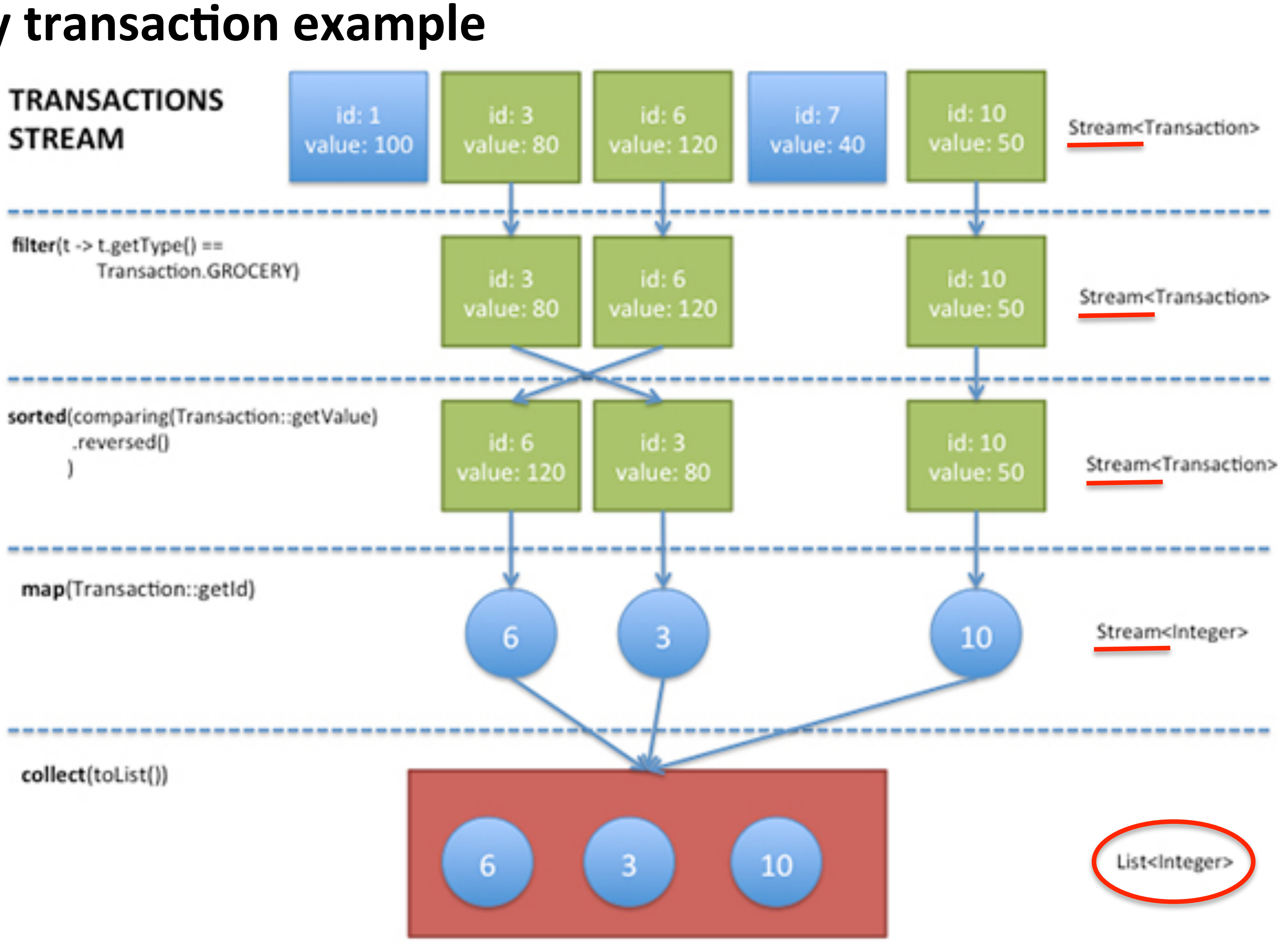
Graphical depiction of grocery transaction example

Filter data

Sort data

Extract data

Return not a stream



http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

There are two types of operations, terminal and intermediate

Types of operations

Terminal

- Close a stream pipeline
- Produce a result such as a List or Integer

Example:

- collect(toList())
- count
- sum

There are two types of operations, intermediate and terminal

Types of operations

Terminal	• Close a stream pipeline • Produce a result such as a List or Integer (any non-stream type)	Example: • collect(toList()) • count • sum
Intermediate	• Output is a stream object • Can be chained together into a pipeline • “Lazy”, do not perform any processing until terminal operation invoked • Pipeline can often be merged into a single pass	Examples: • filter • sorted • map • limit • distinct

Lazy computation allows short circuiting where not all data is evaluated

Short circuiting

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
List<Integer> twoEvenSquares =
    numbers.stream()
        .filter(n -> {
            System.out.println("filtering " + n);
            return n % 2 == 0;
          })
        .map(n -> {
            System.out.println("mapping " + n);
            return n * n;
          })
        .limit(2)
        .collect(toList());
```

Output

```
filtering 1
filtering 2
mapping 2
filtering 3
filtering 4
mapping 4
```

Inputs greater than 4 not evaluated because limit(2) short circuits evaluation; no need to check more items after two have been found

Can't short circuit things like sorting

Examples

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 reimplementation 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 specalized 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).

Examples

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.

Filtering operations

`filter(Predicate)`
Takes a predicate (`java.util.function.Predicate`) as an argument and returns a stream including all elements that match the given predicate

`distinct`
Returns a stream with unique elements (according to the implementation of equals for a stream element)

`limit(n)`
Returns a stream that is no longer than the given size n

`skip(n)`
Returns a stream with the first n number of elements discarded

Streams allow us to process things “as they come”

Stream vs. file

Stream (Streaming movie)	**File (Movie on DVD)**
• Data arrives as it is produced	• Data pre-produced
• Consumers process data as it arrives	• Consumer reads data as desired
• Synchronization keeps Producers and Consumers in sync	• No need for synchronization
• Not all data in memory at once (might cache some)	• All data in memory (or disk) at once
• Can be content can be infinite length	• Content is finite in length
• No way to fast forward or rewind (ignoring caching) without resending data	• Can easily fast forward to rewind

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 - biggerSorted.txt
```

1. cat outputs contents to file

Output

```
We the People of the United States, in
Order to form a more perfect Union,
establish Justice, insure domestic
Tranquility, provide for the common
```

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 - biggerSorted.txt
```

1. cat outputs contents to file
2. Pipe (‘|’) passes output to next command
3. tr translates to lower case

Output

```
we the people of the united states, in
order to form a more perfect union,
Establish justice, insure domestic
tranquility, provide for the common
```

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 - biggerSorted.txt
```

1. cat outputs contents to file
2. Pipe (‘|’) passes output to next command
3. tr translates to lower case

Output

we the people of the united states, in order to form a more perfect union, Establish justice, insure domestic tranquility, provide for the common