CT326 Programming III

STREAM PROCESSING I

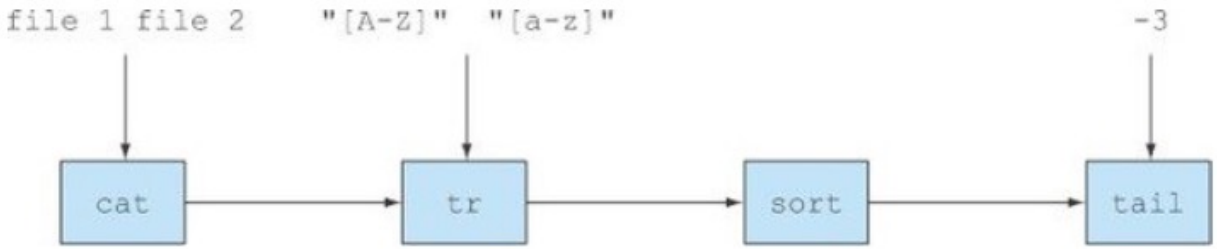
DR ADRIAN CLEAR
SCHOOL OF COMPUTER SCIENCE

Objectives for today

- Understand what streams are and their relationship to collections
- Illustrate examples of using some stream operations from the Stream API
- Distinguish between intermediate and terminal stream operations

Stream Processing

- Introduced in Java 8
- A *stream* is a sequence of data items that are **conceptually** produced one at a time
  - Items from a stream can be read or written one by one
  - Can be combined e.g., an output stream of one program can become the input stream of another
- Unix analogy: `cat file1 file2 | tr "[A-Z]" "[a-z]" | sort | tail -3`



Urma et al. Java 8 in action

Streams API

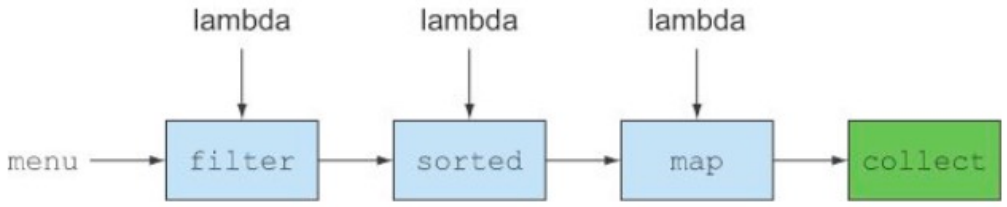
- java.util.stream
- Stream<T> is a sequence of items of type T
- Allows for programming at a higher level of abstraction (streams rather than items)
- Contains many methods that can be chained to make a pipeline
- Includes parallelism *almost for free*
  - Processing streams on multiple CPUs concurrently
  - Don’t have to deal with Threads

Limitations of Collections

- Operations like grouping by category, or searching involve reimplementing lots of iterators
- Take an SQL query as a counter example:
  - SELECT name FROM dishes WHERE calorie < 400
  - No need to worry about implementing the filtering based on attribute
    - Using iterator and accumulator
- Streams provide similar functionality for Collections.
  - Can manipulate collections of data in a **declarative** way (i.e., expressing query rather than implementing approach)

Declarative, composable, and parallelizable

- Declarative code – specify what you want to achieve
- Chain together building-block operations to create a processing pipeline
- Can be tailored using different lambda expressions
- Don’t have to worry about threads and locks for multi-threaded processing



Streams are…

- **sequences of elements…**
  - an interface to a sequenced set of values of a specific type (like collections)
- …from a **source…**
  - Consume from a data-providing source (e.g. collections) preserving the ordering
- …that supports **data processing operations**
  - E.g., filter, map, reduce, find, match, sort
  - Sequential or in parallel

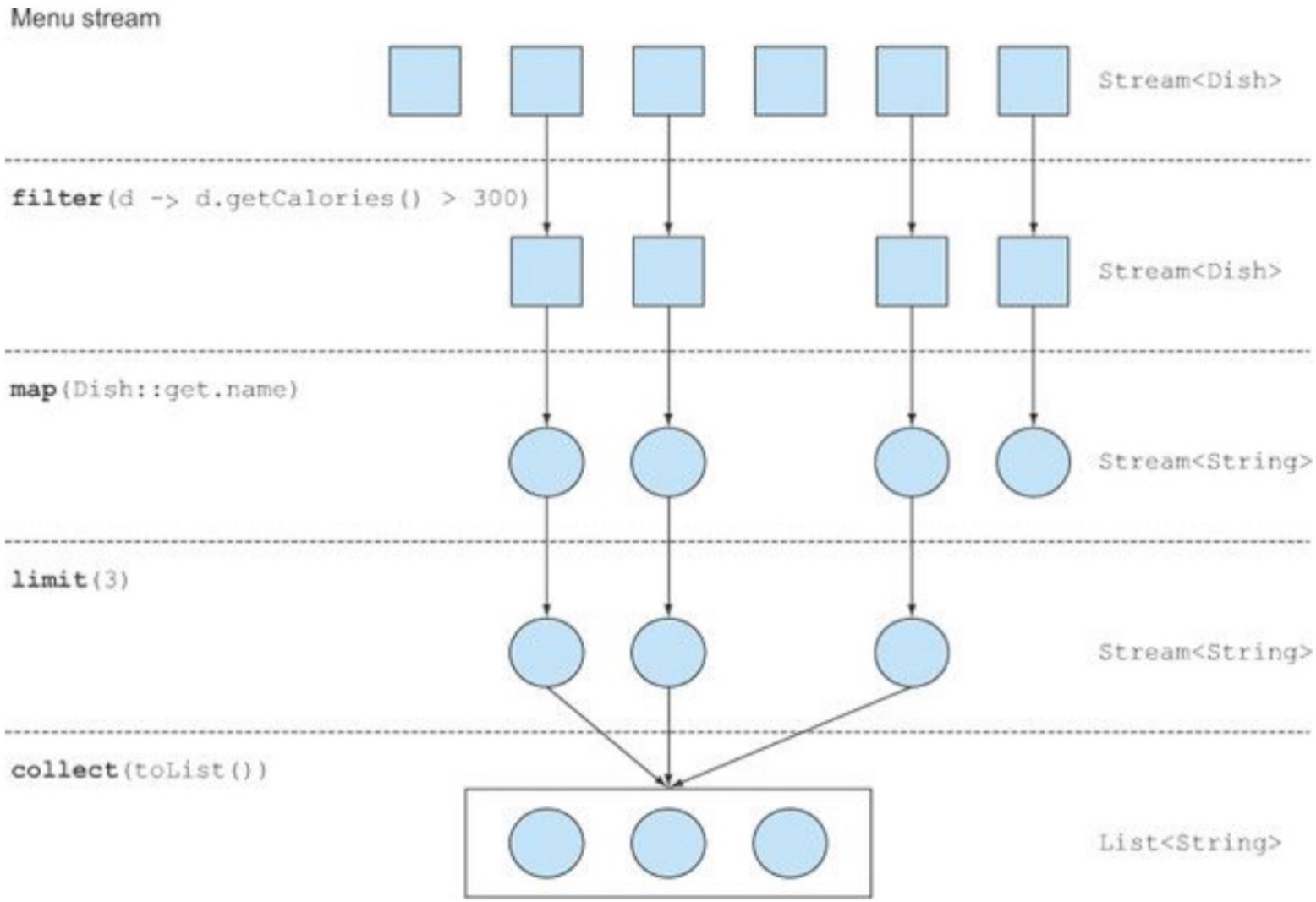
Stream characteristics

- **Pipelining**
  - Many stream operations return streams themselves
  - Operations can be chained to form a larger pipeline to form a database-like query
- **Internal iteration**
  - Iteration occurs behind the scenes

```
List<String> threeHighCalorieDishNames = menu.stream()
                                             .filter((d) -> d.getCalories() > 300)
                                             .map(Dish::getName)
                                             .limit(3)
                                             .collect(toList());
System.out.println(threeHighCalorieDishNames);
```

Method reference operator

- <Class name>::<method name>
- Same as `d -> d.getName()`


Menu stream
Stream<Dish>
filter(d -> d.getCalories() > 300)
Stream<Dish>
map(Dish::get.name)
Stream<String>
limit(3)
Stream<String>
collect(toList())
List<String>

Streams vs. Collections

- Interfaces to data structures representing sequenced set of values of a particular type
- An analogy: Consider watching a pre-recorded football match (Collection) compared to streaming it live over the web (Stream)
  - The latter needs only download and buffer a few frames in advance
- Collections represented completely in memory
  - Can be manipulated (add, remove)
- Streams are conceptually fixed data structures (can't add or remove from them)
  - Elements are computed *on demand*
  - User only selects the values they require; values computed as and when required (recall Producer-Consumer)
- Streams are lazily (just in time) constructed; Collections are eagerly constructed

Stream iteration

- Like iterators, streams are traversable **only once**
  - A stream that has been completely traversed is said to be **consumed**
  - Attempting to traverse a consumed stream will result in an `IllegalStateException` being thrown
- Unlike collections which require **external iteration**...

```
List<String> names = new ArrayList<>();
for(Dish d: menu){
    names.add(d.getName());
}
```

Explicitly iterate the list of menu sequentially.

Extract the name and add it to an accumulator.

- …streams use **internal iteration**

```
List<String> names = menu.stream()
                      .map(Dish::getName)
                      .collect(toList());
```

Start executing the pipeline of operations; no iteration!

Parameterize `map` with the `getName` method to extract the name of a dish.

`java.util.Stream` operations

- **Intermediate operations** are those which can be connected together to form a pipeline
- **Terminal operations** cause the pipeline to be executed and closes it

java.util.Stream operations

```
List<String> names = menu.stream()
                      .filter(d -> d.getCalories() > 300)
                      .map(Dish::getName)
                      .limit(3)
                      .collect(toList());
```

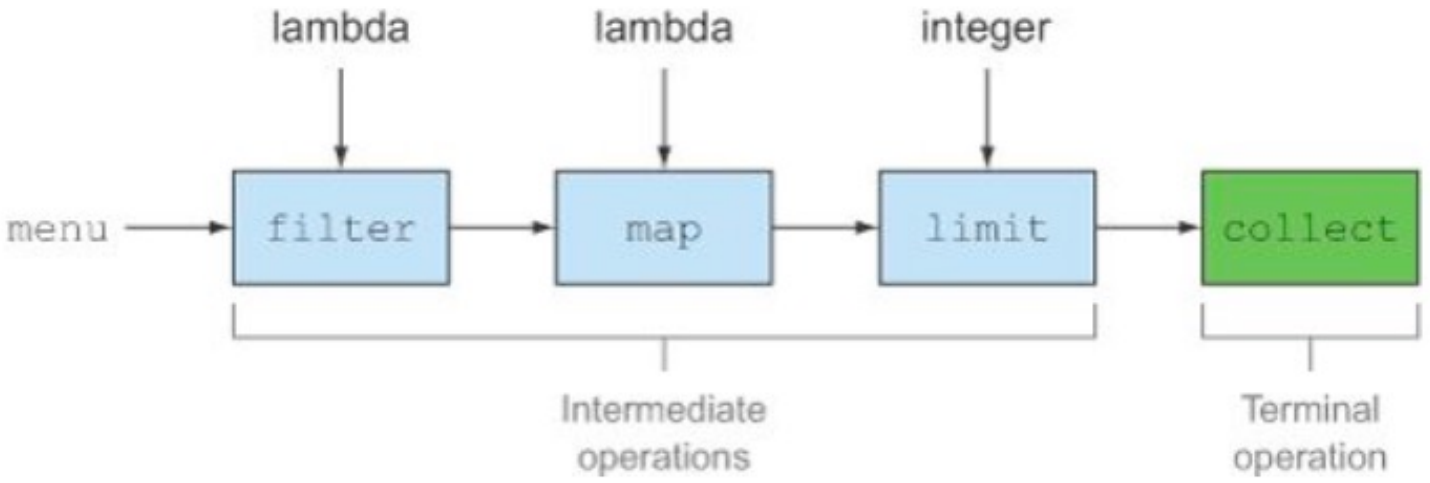
Get a stream from the list of dishes.

Intermediate operation.

Intermediate operation.

Intermediate operation.

Converts the Stream into a List.



Intermediate operations

- Return another stream as the return type
- Allows operations to be combined to form a query
- Don’t perform any processing until terminal operation invoked on pipeline (i.e, lazy)
  - Intermediate operations can usually be merged and processed into single pass by the terminal operation (*loop fusion)*
- Operations include `filter` and `sorted`

Terminal operations

- Produce a result from a pipeline
  - Result is a non-stream value like a `List`, `Integer`, or even `void`

What's this code doing?

- Which are the intermediate and the terminal operators?

```
long count = menu.stream()
                .filter(d -> d.getCalories() > 300)
                .distinct()
                .limit(3)
                .count();
```

Working with streams involves:

- A *data source* (e.g., a Collection) to perform a query on
- A chain of *intermediate operations* that form a stream pipeline
- A *terminal operation* that executes the stream pipeline and produces a result

Intermediate operations

Operation	Type	Return type	Argument of the operation	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
map	Intermediate	Stream<R>	Function<T,R>	T -> R
limit	Intermediate	Stream<T>		
sorted	Intermediate	Stream<T>	Comparator<T>	(T,T) -> int
distinct	Intermediate	Stream<T>		

Terminal operations

Operation	Type	Purpose
forEach	Terminal	Consumes each element from a stream and applies a lambda to each of them. The operation returns void.
count	Terminal	Returns the number of elements in a stream. The operation returns a long.
collect	Terminal	Reduces the stream to create a collection such as a List, a Map, or even an Integer.

Summary

- A stream is a sequence of elements from a sources that supports data processing
- Streams make use of internal iteration and are computed *on demand*
- Operations can be intermediate (return a stream; can be chained) or terminal (return a non-stream value; process the pipeline)