Streams

Week 5 Presentation 1

Streams

~~Input Stream~~

~~Output Stream~~

They are not input streams or output streams

Beware that in spite of the name, it's unrelated to files.
It's about chaining processing.

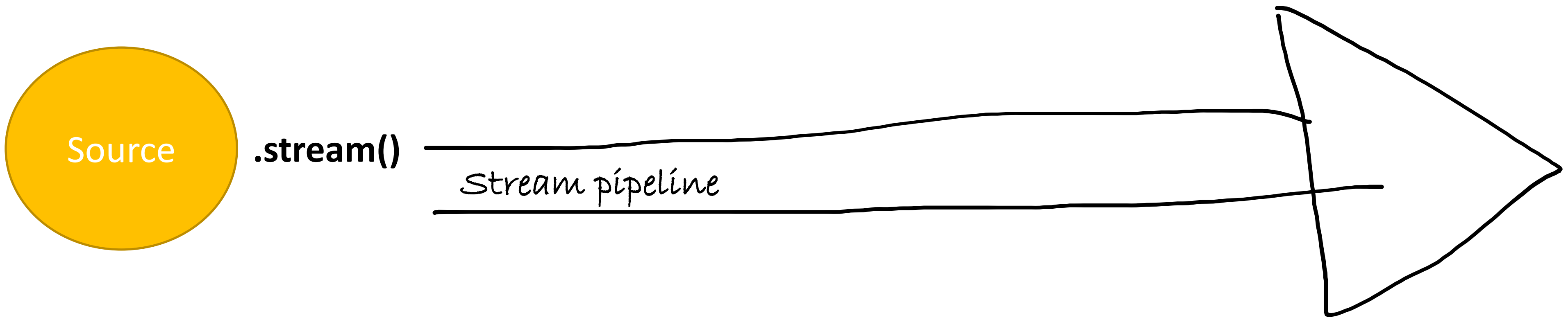
Streams

- Can demonstrate the concept with String operations
- When you apply a method to a string you can apply a new method to the result… and so on.

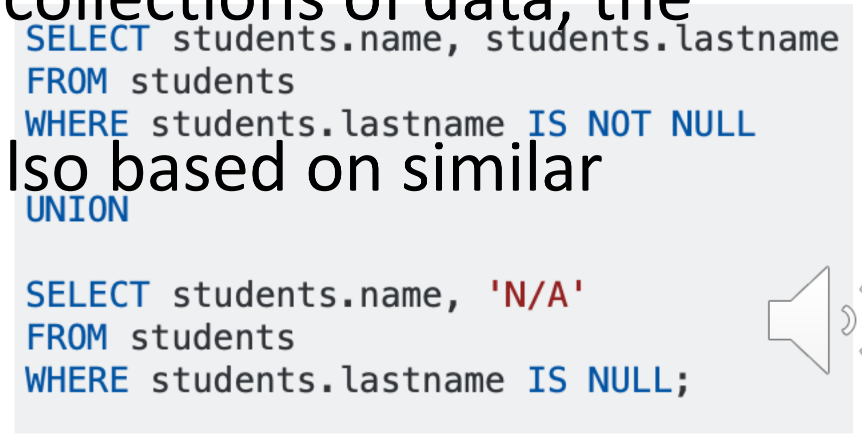
```
String str = “now let’s have some fun”;

Str.toUpperCase().replace(’S’, ‘D’).substring(15, 19).replace(‘M’, ‘N’);

Str                                                  "now let's have some fun"
        .toUpperCase()                               "NOW LET'S HAVE SOME FUN"
                .replace(’S’, ‘D’)                   "NOW LET'D HAVE DOME FUN"
                        .substring(15, 19)           "DOME"
                                .replace(‘M’, ‘N’);  "DONE"
```

Streams

- The same idea may be applied to collections, arrays, functions, IO and others:

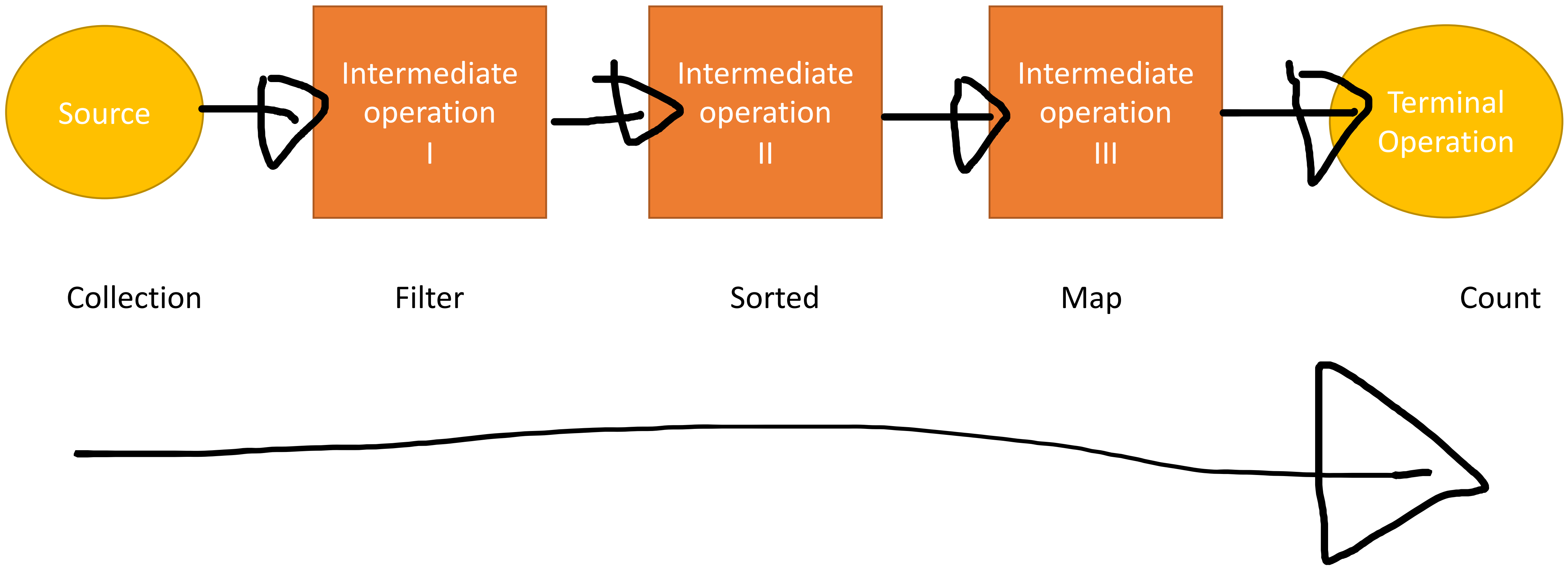


- Because serial processing is often applied to collections of data, the idea is to do something similar in Java
- If you know SQL this will not be new as it is also based on similar ideas



```
SELECT students.name, students.lastname
FROM students
WHERE students.lastname IS NOT NULL
UNION

SELECT students.name, 'N/A'
FROM students
WHERE students.lastname IS NULL;
```

Stream Pipeline



Streams

- Intermediate operations
  - Filter
  - Distinct
  - Sorted
  - Map
- Terminal Operations
  - roreach
  - toArray
  - reduce
  - count
  - min/max

$\}$ With stream operations you have methods that return a stream (which can be fed into something else), and those that don't and terminate the streaming process.

Another Example

Suppose we have an array of Film objects read from a file:

```
class Film {
            private String          title;
            private String          countries;
            private int             year
            private double          billionRMB

public Film(String title, String countries, int year, double billionRMB){
            …
}
…
public toString() …
}
…
ArrayList<Film> films = new ArrayList<Film>();
```

C.SV

We can build a collection read from a file, and then the problem is how to search this collection. We can search on many different criteria – film title, year of release, country, how much it made so far.

Retrieve information using different conditions…

Remember predicate – built in functional interface?

```
import java.util.function.Predicate;
static void filter (Predicate<Film> pred){
        Film f;
        ListIterator<Film> iter = films.listIterator();
        while(iter.hasNext()){
                f = iter.next();
                if (pred.test(f)) {
                        System.out.println(f);
                }
        }
}
```

The predicate functional interface has a single method called test

Remember predicate – built in functional interface?

- Using this method we can very concisely apply different filters for country, year, revenue, etc:

```
filter((film)->film.getYear() == 2014);
```

- To extract the films from a collection meeting the condition that the year is 2014

```
import java.util.function.Predicate;
static void filter (Predicate<Film> pred){
        Film f;
        ListIterator<Film> iter = films.listIterator();
        while(iter.hasNext()){
                f = iter.next();
                if (pred.test(f)) {
                        System.out.println(f);
                }
        }
}
```

Streams

- First we take the collection and turn it into a stream:

```
films.stream()
```

ArrayList<Film>

- Then apply serial operations ending in a termination.

```
films.stream()
        .filter((film)->film.getYear() == 2014)
        .forEach(System.out::println);
```

We can display any film that "gets through" with a forEach() call (a terminal operation) that applies println() to it.

Note the use of the double colon notation that is used here to specify a method applied to each element.

You can insert intermediate operations before the terminal one, for instance sort the output, if of course Java knows how to sort Film objects.

Note that it's usually FAR more efficient to sort AFTER filtering rather than BEFORE filtering because the set to sort is smaller, even if both are possible ...

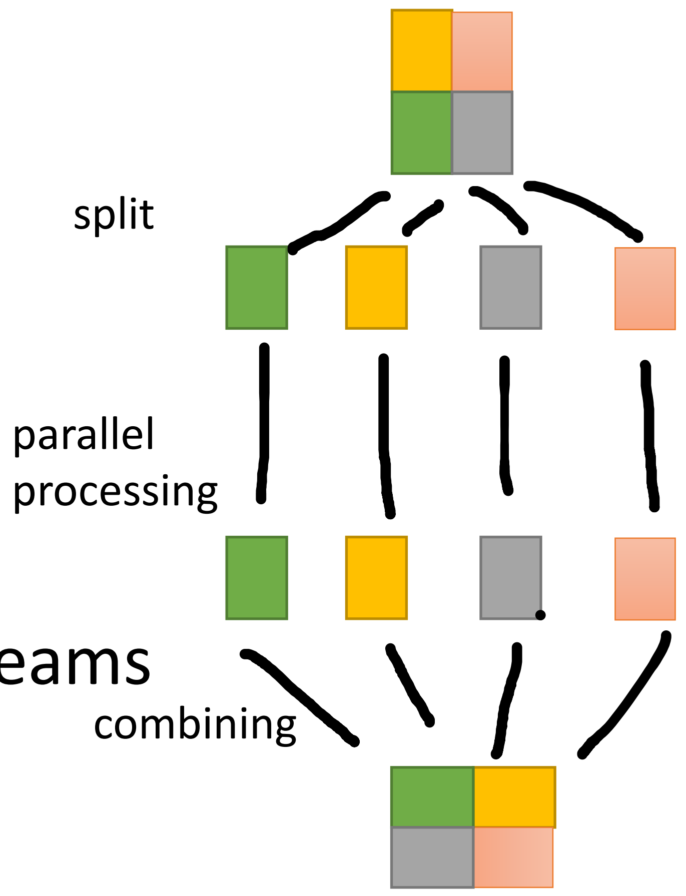
```
films.stream()
        .filter((film)->film.getYear() == 2014)
        .sorted()
        .forEach(System.out::println);
```

NOTE: needs to implement Comparable

Parallel Streams

.parallelStream()

Like a river that reaches the sea with a delta, streams can be split into multiple parallel streams for faster processing. This is important in applications of big data.



https://docs.oracle.com/javase/tutorial/collections/streams/index.html

further examples and discussions of streams can be found in the docs here