Java Boot Camp

Containers, Streams, Functional and Reactive Programming

3. Java Streams

Some Notation

- Java has a forEach(f) method that applies to collections
  - It applies the the function f to each element of the list in turn
  - Sort of a functional programming version of a for loop.
- There are three version of the for loop
  - The old style loop .. **for, while**, etc
  - The semi-functional style with an external iterator **for(int x : myIntList){}**
- The functional style using an internal iterator
  - **collection.forEach(function-to-apply)**
  - The internal iterator is maintained by the collection
  - We just tell the collection to iterate over itself
  - And then for each item, apply the function
  - The function can be a Lambda, a function variable or a built-in function

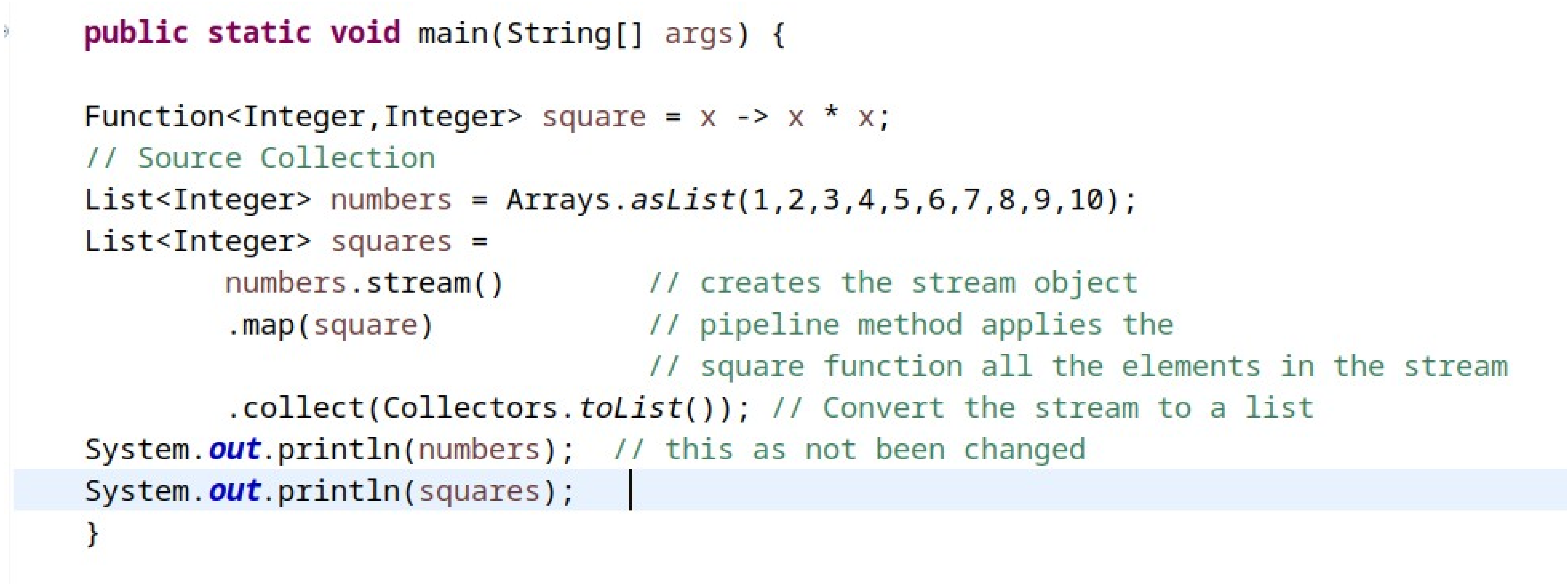
Java Streams Notation

Demo Zero

Java Streams

- Functional programming support was implemented in Java to support stream processing
- A stream processes a collection of data objects
  - It takes input from a source of some kind without altering the source
  - The data items move through a pipeline of transformations
  - A terminal operation ends the stream
- A stream in Java is not like a message queue
  - It can be helpful to think of it as a sequence of data objects in writing the code
  - Conceptually, all the elements in a stream are processed at the same time at each step of the pipeline
  - A terminal operation either returns some collection or some single result or performs an operation (like saving to persistent storage) for each element in the stream
  - The actual pipeline is optimized before any processing takes place

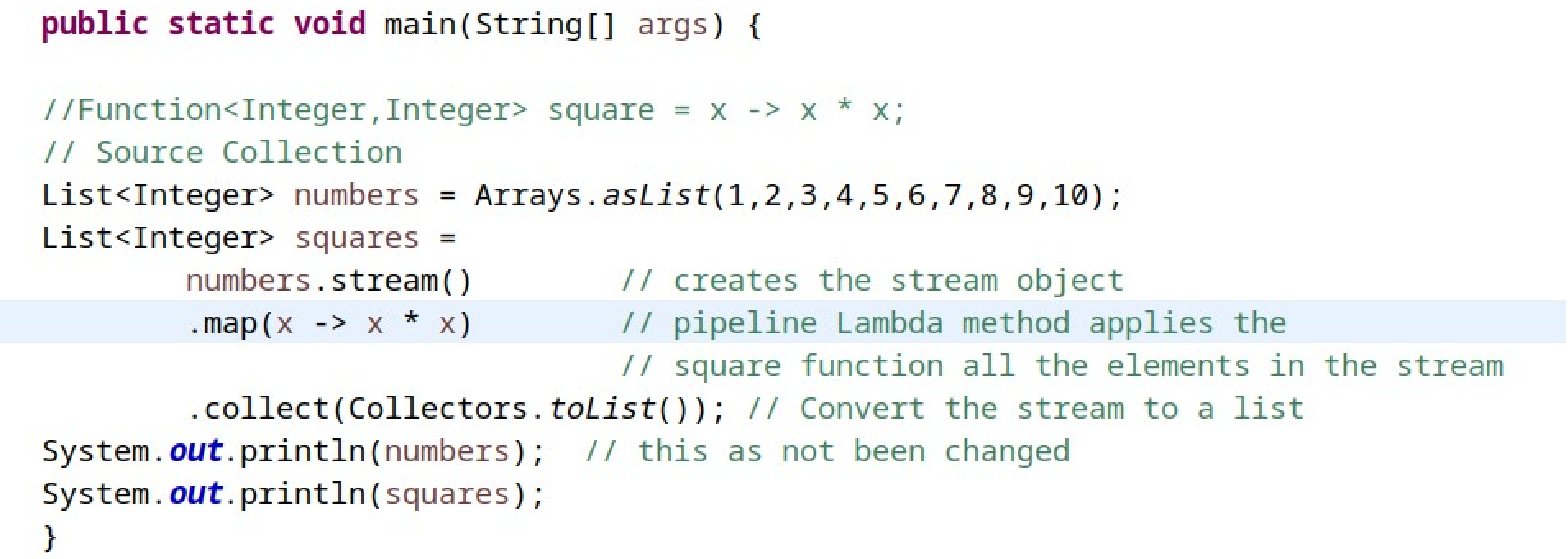
Java Streams Simple Example



```
public static void main(String[] args) {

Function<Integer,Integer> square = x -> x * x;
// Source Collection
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
List<Integer> squares =
        numbers.stream()        // creates the stream object
        .map(square)            // pipeline method applies the
                                // square function all the elements in the stream
        .collect(Collectors.toList()); // Convert the stream to a list
System.out.println(numbers);  // this as not been changed
System.out.println(squares);
}
```

Java Streams Simple Lambda Example



```java
public static void main(String[] args) {

//Function<Integer,Integer> square = x -> x * x;
// Source Collection
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
List<Integer> squares =
        numbers.stream()        // creates the stream object
        .map(x -> x * x)        // pipeline Lambda method applies the
                                // square function all the elements in the stream
        .collect(Collectors.toList()); // Convert the stream to a list
System.out.println(numbers);  // this as not been changed
System.out.println(squares);
}
```

Initial Methods

- These take a stream as an input
- Return a stream as an output
- Large library of methods - some of these are:
  - map(function) – applies the function to each element of the stream
  - filter(predicate) – keeps the elements that match the predicate, discards the others
  - sorted() – sorts the stream
- Other pipeline methods are in the java.util.streams library
  - We won’t be going into these in class but they will be demoed

Pipeline Methods

- These take a stream as input
- Return a stream as output
- Large library of methods - some of these are:
  - **map**(function) – applies the function to each element of the stream
  - **filter**(predicate) – keeps the elements that match the predicate, discards the others
  - **sorted**() – sorts the stream
- Other pipeline methods are in the java.util.streams library

Terminal Methods

- Terminal methods are methods that take an input from a stream and produce a final result
- Terminal methods mark the end of a stream – each stream can have only one terminal method
- Some terminal methods are:
  - **collect**(collection) – returns the result of the intermediate operations as a collection (e.g. list, array etc)
  - **forEach**(function) – applies the function to each element of the stream – does not produce an output stream
  - **reduce**(function) – uses function to collapse a stream into a single value

Java Streams Basics

Demo One and Two

Lazy Invocation

- Streams are not executed until a terminal method is encountered
- The stream is represented as a directed acyclic graph (DAG)
- This DAG can be optimized at compile time with a number of standard rewrite rules
  - The stream can only be optimized when the whole DAG is complete
  - And that happens when a terminal operation is encountered

Lazy Invocation

Demo Three

Streams Basics

Lab Streams 1

Intermediate Methods

- Any stream method that returns a stream is an intermediate or pipeline method
- Some can be though of as working on individual elements
  - Specifically, they can operate on an element without reference to other elements
  - Examples - filter(), map()
  - These operations can be parallelized
- Others need to examine the relationships between stream elements
  - Examples - sorted(), distinct()
- Intermediate operations should not have side effects
  - We violated this in some of the demos to see what was happening in a stream
  - The terminal methods are where any side effects should occur

Intermediate Methods

- map(f)
  - Applies a monadic function f to each element in the stream and returns a transformed element
- filter(p)
  - Applies the predicate p to each element in the stream, if the result is false, the element is removed from the stream
- peek(f)
  - Used for debugging, it executes f on each element, when you want to see the elements as they flow past a certain point in a pipeline
- distinct()
  - removes duplicates based on the defined equality operator for the stream elements
- sorted()
  - sorts the elements based on the defined comparison operator for the stream elements

Intermediate Methods

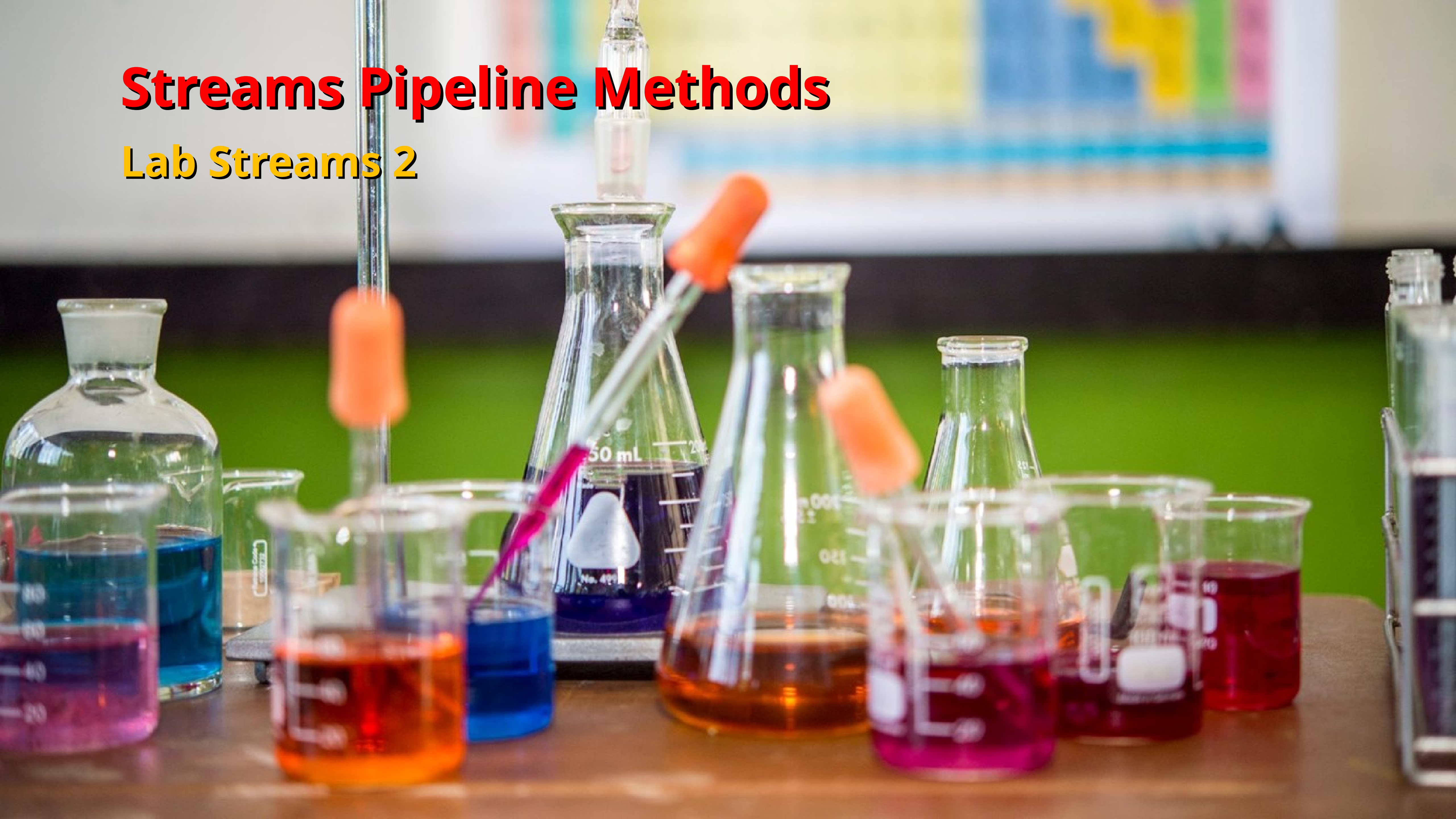
- skip(n)
  - Omits the first n elements of a stream
- limit(n)
  - Truncates the stream after the n elements
- flatMap(stream)
  - removes levels of structure to flatten a stream
  - For example, a list of lists of integers has two levels of structure
    - [[2, 3, 5], [7, 11, 13], [17, 19, 23]]
  - Flattening the list removes the nested structure
    - [2, 3, 5, 7, 11, 13, 17, 19, 23]

Stream Pipeline Methods

Demo Four

Streams Pipeline Methods

Lab Streams 2



Terminal Operations

- The three basic types of terminal operations are
  - **Reducers** - returns a single value representing a computation on the stream - the stream is reduced to a single value
  - **Collectors** - returns some sort of collection
  - **Operators** - performs an operation on each element of the stream and returns void
- Terminal methods are terminal because they do not return a stream
  - They represent the end of the stream

Reducers

- There are a number of standard reducers
- count()
  - Returns the number of elements in the stream
- min(comparator), max(comparator)
  - These returns Optionals which are like futures to account for the cases where the value may or may not be returned
  - If isPresent() is true meaning that the value exists, it can be retrieved with the get() method
  - The comparator is the predicate used to determine how to order the elements
- anyMatch(p), allMatch(p), noneMatch(p)
  - Returns a Boolean if the predicate p is
    - true for any one of the elements in the stream
    - true for all the elements in the stream
    - true for none of the elements in the stream

Reducers

- reduce(accumulator, operator)
  - The accumlator is the last value computed (ie. from the previous element)
  - The operator is a function applied to combine the accumulator with the current element
  - Like summing an array in a loop
    - The accumulator is the running total
    - The operator is adding the current element to the running total

Collector

- The the Collectors class has a number of methods that return collections of various types
  - Eg. toList(), toMap(), toSet()
- There are other sorts of collectors that, for example:
  - Combine the stream into a single String
  - Do reduction type operations as well
  - In fact, reducers can be thought of as special cases of collectors

Java Streams Reducers

Demo Five - Seven

Streams Terminal Methods

Lab Streams 3

Questions?

CMA CGM
CMAU 115887 0
22G1
CMAU 159369 3
22G1
GATU 082138 7
22G1
TRLU US 308341 2
2210
GESU 276340 7
22G1
DELMAS
DVRU 147046 5
22G1
DVRU 143916 1
22G1
XINU 108806 5
CLHU 309435 0
FCIU 268180 8
TTNU 168218 2
DVRU 151838 4
TRIU US 396740 3
2210
TRLU US 275847 1
2210
凯 航
CNC LINE
tex

End Module