### The Stream API: Solving Problems by Engaging Deeper Values of Intelligence

Wholeness of the Lesson: The stream API is an abstraction of collections that supports aggregrate operations like *filter* and *map*. These operations make it possible to process collections in a declarative style that supports parallelization, compact and readable code, and processing without side effects. Deeper laws of nature are ultimately responsible for how things appear in the world. Efforts to modify the world from the surface level only lead to struggle and partial success. Affecting the world by accessing the deep underlying laws that structure everything can produce enormous impact with little effort. The key to accessing and winning support from deeper laws is going beyond the surface of awareness to the depths within.

Some material in the beginning may be a repeat (JL)

What Are Streams and Why Are They Used?

1. A stream is a way of representing data in a collection (and in a few other data structures) which supports functional-style operations to manipulate the data. From the API docs: A stream is “a sequence of elements supporting sequential and parallel aggregate operations.” Streams provide new ways of accessing and extracting data from Collections.
2. To understand why they are used, consider the following task as an example: Given a list of words (say from a book), count how many of the words have length > 12.
3. Imperative-style solution:

Issues:   
i. Relies on shared variable *count*, so may not be thread safe  
ii. Commits to a particular sequence of steps for iteration  
iii. Emphasis is on *how* to obtain the result, not *what*

*int count = 0;*

*for(String word : list) {  
 if(word.length() > 12)   
count++;*

*}*

1. Functional-style solution (using ideas introduced in previous Lesson)

*final long count = words.stream().filter(w ->w.length() > 12).count();*

Advantages:  
i. Purely functional, so thread safe  
ii. Makes no commitment to an iteration path, so more parallelizable  
iii. Declarative style – “what, not how”

Example of parallelizing stream processing: (on a multi-core processor, this is a real speedup)

*final long count = words.parallelStream().filter(w ->w.length() > 12).count();*

Facts About Streams

1. *Streams do not store the elements they operate on*. Typically they are stored in an underlying collection, or they may be generated on demand.
2. *Stream operations do not mutate their source*. Instead, they return new streams that hold the result.
3. *Stream operations are lazy whenever possible.* So they are not executed until their result is needed. Example: In previous example, if you request only the first 5 words of length > 12, the filter method will stop filtering after the fifth match. This makes it possible to have (potentially) *infinite streams.*

Template for Using Streams

1. *Create a stream*. Typically, the stream is obtained from some kind of *Collection*, but streams can also be generated from scratch.
2. *Create a pipeline of operations*. Each of the operations transforms the stream in some way, and returns a new stream.
3. *End with a terminal operation.* The terminal operation produces a result. It also forces lazy execution of the operations that precede it.

NOTE: After a terminal operation on a pipeline of operations on a stream, the stream can no longer be used. You have to be careful not to attempt to re-use a stream after a terminal operation has been called on it.

Example from Lesson 8:

*List<String> startsWithLetter =*

*list.stream() //create the stream*

*.filter(name ->name.startsWith(letter)) //build pipeline*

*.collect(Collectors.toList()); //invoke terminal operation*

Ways of Creating Streams

1. Obtain a *Stream* from any *Collection* object with a call to *stream()* (this *default* method was added to the *Collection* interface in Java 8)
2. Get a *Stream* from an array like this:  
   *int[] arrOfInt = {1, 3, 5, 7};*

*Stream<Integer> strOfInt = Stream.of(arrOfInt);*

1. Get a *Stream* from any sequence of arguments: (the *of* method accepts a *varargs* argument – for a review of varargs see <https://docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html>)

*Stream<String> song = Stream.of(“gently”, “down”, “the”, “stream”);*

1. Two ways to obtain *infinite* streams: *generate* and *iterate* (remember stream operations are lazy)
   1. The *generate* function accepts a *Supplier<T>* argument; in practice, this means that it accepts functions (lambda expressions) with zero parameters.

*interface Supplier<T> {*

*T get();*

*}*

Example: *Stream* of constant values (“Echo”):   
*Stream<String> echoes = Stream.generate(() -> “Echo”);*

Example: *Stream* of random numbers:   
*Stream<Double>randoms = Stream.generate(Math::random);*

* 1. The *iterate* function accepts a seed value (of type *T*) and a *UnaryOperator<T>* argument.

*Interface UnaryOperator <T> {*

*T apply(T t);*

*}*

*iterate* - *Returns an infinite sequential ordered*Stream*produced by iterative application of a function*f*to an initial element*seed*, producing a*Stream*consisting of*seed*,*f(seed)*,*f(f(seed))*, etc.*

Example: *Stream* of natural numbers: (Here, *T* is *BigInteger*)   
*Stream<BigInteger> naturalNums =*

*Stream.iterate(BigInteger.ONE, n ->n.add(BigInteger.ONE))*

Can do the same thing with *Integers* instead of *BigIntegers*, but not with ints (we discuss Streams based on primitives later in the lesson)

Stream<Integer> stream2

= Stream.iterate(1, n ->n + 1));

Extracting Substreams and Combining Streams

1. *stream.limit(n).*The call *stream.limit(n)*returns a new stream that ends after n elements (or when the original stream ends if it is shorter). This method is useful for cutting infinite streams down to size.

Example:

*Stream<Double> randoms =*

*Stream.generate(Math::random).limit(100);*

yields a stream with 100 random numbers.

1. *stream.skip(n)*The call *stream.skip(n)discards* the first n elements.
2. *stream.concat(Stream)*You can concatenate two streams with the static *concat* method of the *Stream* class: This one IS IMPORTANT!   
     
   Example:

*Stream<Character> combined =*

*Stream.concat(characterStream("Hello"),*

*characterStream("World"));*

*// Yields the stream ['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd']*

Note: For concatenation, the first stream should not be infinite—otherwise the second wouldn’t Ever be accessed.

Here is the *characterStream* method – transforms a String into a Stream of Characters:

*public static Stream<Character> characterStream(String s) {*

*List<Character> result = new ArrayList<>();*

*for (char c : s.toCharArray()) result.add(c);*

*return result.stream(); // returning a Stream here!*

*}*

Stream Operations:   
Use *filter* to Extract a Substream that Satisfies Specified Criteria

1. *filter* accepts as its argument a *Predicate<T>* interface.

*interface Predicate<T> {*

*boolean test(T t);*

*}*

Recall the earlier example: *final long count = words.stream().filter(w ->w.length() > 12).count();*

(It looks like ‘w’ is the argument for the ‘test’ method, and, the predicate is w.length() > 12 (JL))

1. The return value of *filter* is another *Stream*, so filters can be chained:

*words.stream()   
.filter(name ->name.contains(""+c))*

*.filter(name -> !name.contains(""+d))*

*.filter(name ->name.length()==len)*

*.count();*

Stream Operations:Use *map* to Transform Each Element of a Substream

1. *map* accepts a *Function* interface. Typical special case of the *Function* interface is

*interface Function<T,R> {*

*R apply(T t);*

*}*

1. A map accepts this type of *Function*  interface and returns a *Stream<R>* -- a stream of values each having type *R*, which is the return type of the *Function* interface. *map*s can therefore be chained.  
     
   Example: Given a list

*List<Integer>list*

of *Integer*s, obtain a list of *String*s representing those *Integer*s (T is *Integer*, R is *String*)

*List<String> strings = list.stream()  
 .map(x -> x.toString()) // x is an*

*//Integer here  
.collect(Collectors.toList())*

Application: Using *map* with Constructor References – DO THIS!!

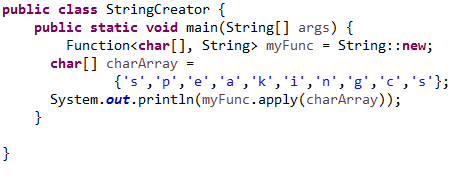
1. *Class::new* is a fourth type of method reference, where the method is the *new* operator. Examples:
2. *Button::new* - compiler must select which *Button* constructor to use; determined by context. When used with *map*, the *Button(String}* constructor would be used, and the constructor reference *Button::new* resolves to the following lambda:  *str -> new Button(str)*(which realizes a *Function* interface, as required by *map*).

*List<String> labels = ...;*

*Stream<Button> stream = labels.stream().map(Button::new);*

*List<Button> buttons = stream.collect(Collectors.toList());*In this example, *map* passed each String in *labels* into the *Button* constructor and creates in this way a stream of labeled buttons, which are then collected together into a list at the end.

1. String::new GO OVER BRIEFLY!! (JL)

 *//This works because there is a String constructor that takes in a //’char[ ]’ as an argument (*String(char[] value)

*//Look at the Function class in Oracle Java API!!*

*//output: speakingcs*

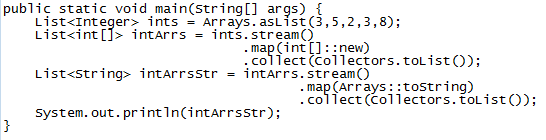
Note: In this case*, String::new* is short for the lambda expression   
*charArray -> new String(charArray),* which is a realization of the *Function* interface*.*

*See Demo: lesson9.lecture.newstring*

Look at this one again!

1. int[]::new is another constructor reference, short for the lambda expression   
   *len -> new int[len]* (where *len* is an integer that is used as the new array length)

*OPTIONAL -* Exercise: What is the following code doing? What is the output when it is run?



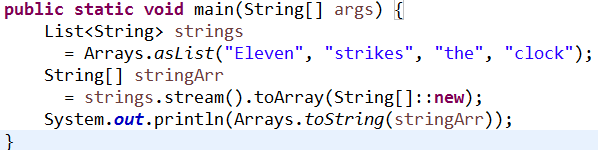
Question: In this example, the list *ints* is turned into a stream – could we change the code so that we start with a stream of integers*,*using one of the stream operations *iterate* or *generate*?

1. *Array constructor reference and the toArray method*If you have created a *Stream<String>,* we have seen how to output a *List<String>* from this stream, using *collect* (more on this later), but how to obtain an array *String[]*? A first try would be to provide a *toArray* method:  
    *Stream<String> stringStream = //…  
    String[] vals = stringStream.toArray(); //compiler error*

The *toArray* method exists, but produces an *Object[],* not a *String[]*. Can solve with a constructor reference:

*String[] vals = stringStream.toArray(String[]::new); //see*

*//Oracle API*



Output:  
[Eleven, strikes, the, clock]

Stream Operations, continued: Use *flatMap* to Transform Each Element of a Substream and Flatten the Result (from a Stream of Streams (JL))

We illustrate *flatMap* with an example:

Suppose we apply the *characterStream* method (see earlier slide) to each element of a list, using map:

*List<String> list = Arrays.asList(“Joe”, “Tom”, “Abe”);*

*Stream<Stream<Character>> result = list.stream().map(s ->*

*characterStream(s))*

The *Streamresult* looks like a list of lists (really a stream of streams): *[[‘J’, ‘o’, ‘e’],[‘T’, ‘o’, ‘m’],[‘A’, ‘b’, ‘e’]].*

“Flattening” this *Stream* means putting all elements together in a single stream. This is accomplished using *flatMap* in place of *map*:

*Stream <Character> flatResult* = *list.stream().flatMap(s ->*

*characterStream(s))*

Output in this case has been *flattened*: *[‘J’, ‘o’, ‘e’, ‘T’, ‘o’, ‘m’, ‘A’, ‘b’, ‘e’].*

Stateful Transformations

1. The transformations discussed so far – *map, filter, limit, skip, concat* – have been *stateless:* each element of the stream is processed and forgotten.
2. Two *stateful* transformations available from a *Stream* are *distinct* and *sorted.*
3. Example of distinct:

*Stream<String> uniqueWords =*

*Stream.of("merrily", "merrily", "merrily", “gently").distinct();*

*//output: ["merrily", "gently"] // ‘distinct’ remembers!*

1. Example of sorted*:* (sorted accepts a *Comparator* parameter)

*//sort by decreasing lengths of words*

*List<String> words = Arrays.asList("Tom", "Joseph", "Richard");*

*Stream <String> longestFirst*

*= words.stream().sorted((String x, String y) ->*

*(new Integer(y.length()).compareTo(new Integer(x.length()))));*

*//show how this works with a concrete example like “hat” and “runner”. A trick is used here.*

*System.out.println(longestFirst.collect(Collectors.toList()));*

*//output: Richard, Joseph, Tom*

Note: This code uses some functional techniques, but notice that the *Comparator* still has the flavor of “how” rather than “what”.

Implementing *Comparator*s with More Functional Style

In previous example, we are seeking to sort “by *String* length”, in reverse order. Rather than specifying *how* to do that, we can use the new static *comparing* method in *Comparator*: *Stream<String> longestFirst = words.stream().sorted(Comparator.comparing(String::length).reversed());*

1. *Comparator.comparing* takes a *Function<T,U>* argument. The type *T* is the type of the object being compared – in the example, *T* is *String*. The type *U*  is the type of object that will actually be compared - since we are comparing lengths of words, the type *U* is *Integer* in this case.   
     
   Knowing these points makes it possible to write the call to *sort* even more intuitively.  
     
   *Function<String, Integer> byLength = x ->x.length(); //same*

*// as String::length*

*Stream<String> longestFirst = words.stream().sorted(Comparator.comparing(byLength).reversed())*Note: *reversed()* is a default method in *Comparator* that reverses the order defined by the instance of *Comparator* that it is being applied to.

1. Another example of *comparing* function: Create a *Comparator<Employee>* that compares *Employees* by *name*, and another that compares by *salary*

*Comparator<Employee> NameComparator*

*= Comparator.comparing(Employee::getName);*

*Comparator<Employee> SalaryComparator*

*= Comparator.comparing(Employee::getSalary);*

1. Support for *Comparators* that are *consistent with equals.*(Review consistency with equals issue in *lesson8.lecture.exercise.employeecode* and also in Lab 8)

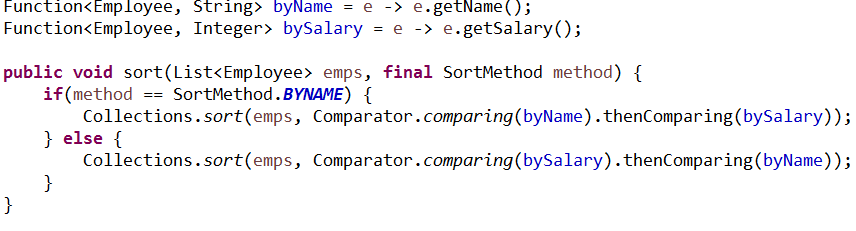
- Recall when we wanted to sort *Employees* (where an *Employee* has a *name* and a *salary*) by

*name*, we needed to consider also the *salary*, or else the *Comparator* is not consistent with

*equals (Two people can have the same name here).*

- We use the *Comparing* and *thenComparing* methods of *Comparator*

*(‘thenComparing’ breaks ties when the comparing method has values that are not unique. (JL).)*



*-* Notes:

* *comparing* is a static method of *Comparator*, and therefore cannot be chained
* *thenComparing* is a default method so can be chained; it modifies current *Comparator* by introducing its *compare* method just when the current *compare* method returns 0. (i.e. a tie, the same name in the first line above - byName (JL))

Getting Outputs from Streams: Reduction Methods

1. The last step in a pipeline of *Streams* is an operation that produces a final output – such operations are called *terminal operations* because, once they are called, the stream can no longer be used. They are also called *reduction methods* because they reduce the stream to some final value. We have already seen one example: *collect(Collectors.toList())*
2. *count:* Counts the number of elements in a *Stream.*

*List<String> words = //…*

*int numLongWords = words.stream().filter(w ->w.length() > 12).count();*

1. *max, min, findFirst, findAny* search a stream for particular values and will throw an exception if not handled properly. An easy way to handle:  
   Example: *max*

*Optional<String> largest = words.stream()*

*.max(String::compareToIgnoreCase);*

*if (largest.isPresent())*

*System.out.println("largest: " + largest.get());*

*An Optional* is a wrapper for the answer – either the found *String* can be read via *get(),* or a *Boolean* flag (isPresent) can be read that says no value was found (for example, if the stream was empty).You can call *get()* on an *Optional* to retrieve the stored value, but if the value was not found, so that the *Optional* flag *isPresent* is false, calling *get()*produces a *NoSuchElementException.*

Example*: findFirst*

*Optional<String> startsWithQ*

*= words.stream().filter(s ->s.startsWith("Q")).findFirst();*

Example*: findAny* This operation returns *an element* if any match is found*, empty* otherwise; this one works well with parallel streams: LOOK AT API, a bit different!

*Optional<String> startsWithQ = words.parallelStream()  
 .filter(s ->s.startsWith("Q"))  
 .findAny();*

Working with *Optional* – A Better Way to Handle *Null*s

1. The previous slide introduced the *Optional*class. *Optional* was added to Java to make handling of *nulls*  less error prone. However notice

*if (optionalValue.isPresent()) optionalValue.get().someMethod();*

is no easier than

*if (value != null) value.someMethod();*

The *Optional* class, however, supports other techniques that are superior to checking *null*s*.*

1. The *orElse* method – if result is *null*, give alternative output using *orElse*

*//*NEW WAY

*public static void pickName(List<String> names, String startingLetter) {*

*final Optional<String> foundName =*

*names.stream().filter(name -> name*

*.startsWith(startingLetter))*

*.findFirst();*

*System.out.println(String.format("A name   
 starting with %s: %s",startingLetter,*

*foundName.orElse("No name found")));*

*}  
}*

*//The orElse method on an Optional returns the //value stored in the Optional if present, or //else returns the alternative value (having same //type as the value stored in original Optional) //that is supplied as the argument to orElse*

//OLD WAY

*Public static void pickName(List<String> names, String startingLetter) {  
 String foundName = null;  
for(String name : names){  
if(name.startsWith(startingLetter)) { foundName = name;  
 break;  
 }  
 }  
System.out.print(String.format("A name  
 starting with %s: ", startingLetter));  
if(foundName != null) {   
System.out.println(foundName);  
 } else {  
System.out.println("No name found");  
 }  
}*

Use *ifPresent(Consumer)* to invoke an action and skip the *null* case completely.

*Public static void pickName(List<String> names, String startingLetter) {*

*final Optional<String> foundName =*

*names.stream()*

*.filter(name ->name.startsWith(startingLetter))*

*.findFirst();*

*foundName.ifPresent(name ->System.out.println("Hello " + name));*

*}*

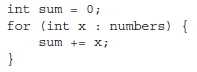
Question*.* Why did the Java creators insert a lambda as the argument for *ifPresent* here?

Answer. See the Oracle Java API.

The *reduce* Operation

The *reduce* operation lets you combine the terms of a stream into a single value by repeatedly applying an operation.

Example We wish to sum the values in a list of numbers. Procedural code:



Using the *reduce* operation, the code looks like this:



First argument is an initial value; it is the value that is returned if the stream is empty (it is also the *identity element* for the combining operation). The second argument is a lambda for *BinaryOperator<T>*

*Interface BinaryOperator<T> {  
 T apply(T a, T b);  
}*

Applied to a list of numbers, this *reduce* operation returns the sum of all the numbers. The initial value makes sense here because the “sum of an empty set of numbers is 0”.

The initial value is also used to produce the final computation. For example, if *numbers* is [2,1,4,3], then the *reduce* method performs the following computation:

(((0 + 2) + 1) + 4) + 3 = 10

A parallel computation can improve performance. Say [2,1,4,3] is broken up into [2,3],[4,1]. Then in parallel we arrive at the same answer in the following way:

*sum1 = (0 + 2) + 3 sum2 = (0 + 4) + 1 combined = sum1 + sum2 = 10*

How could we form the *product* of a list of numbers?

Example We form the product of a list *numbers* of numbers. For the initial value, we ask, “What is the product of an empty set of numbers?” By convention, the product is 1. (Note that 1 is the identity element for multiplication.) Here is the line of code that does the job:



Example. What about subtraction? What happens when the following line of code is executed? Try it when *numbers* is the list [2, 1, 4, 3].

*int difference = numbers.stream().reduce(0, (a, b) ->a - b);*

Here, the computation proceeds like this: *(((0 – 2) – 1) – 4) – 3) //output: -10*

The problem here is that performing this computation in parallel gives a different result; subtractions are grouped differently for a parallel computation. For instance, during parallel computation, if [2,1,4,3] is broken up into [2,3] and [4,1], the computation would look like this:

*diff1* = *(0 – 2) – 3 diff2 = (0 – 1) – 4 combined = diff1 - diff2 = 0*

For this reason, a requirement concerning *reduce* is:

*Only use reduce on associative operations.*

(Note that + and \* are associative, but subtraction is not.)See the demo *lesson9.lecture.reduce.*

The reduce method has an overridden version with only one argument*.*

Continuing with the sum example, here is a computation with the overridden version:



This version of *reduce* produces the same output as the earlier version *when the stream is nonempty,* but it is stored in an *Optional* in this case. When the stream is empty, the *reduce* operation returns a null, which is again embedded in an *Optional.*

Main Point 1

When a *Collection* is wrapped in a *Stream*, it becomes possible to rapidly make transformations and extract information in ways that would be much less efficient, maintainable, and understandable without the use of *Stream*s. In this sense, *Stream*s in Java represent a deeper level of intelligence of the concept of “collection” that has been implemented in the Java language. When intelligence expands, challenges and tasks that seemed difficult and time-consuming before*,* can become effortless and meet with consistent success. This is one of the documented benefits of TM practice.

Collecting Results

One kind of terminal operation in a stream pipeline is a *reduction* that outputs a single value, like *max* or *count*. Another kind of terminal operation collects the elements of the *Stream* into some type of collection, like an array, list, or map. We have seen examples already*.*

Example: Collecting into an array

*String[] result = words.toArray(String[]::new); //toArray is overloaded in a List.*

*For the teacher - Look at the original file, at the end, in*

*the Standard courses directory.*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*Teach the peek() method*

This method exists mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline: (from the Oracle Java API)

Stream.of("one", "two", "three", "four")

.filter(e -> e.length() > 3)

.peek(e -> System.out.println("Filtered value: " + e))

.map(String::toUpperCase)

.peek(e -> System.out.println("Mapped value: " + e))

.collect(Collectors.toList());

CONNECTING THE PARTS OF KNOWLEDGE  
WITH THE WHOLENESS OF KNOWLEDGE

## Declarative programming and command of all the laws of nature

1. In Java SE 7, the only first-class citizens are objects, created from classes. The valuable techniques of functional programming and a declarative style can be approximated using functional interfaces.
2. In Java SE 8, functions – in the form of lambda expressions – have become first-class citizens, and can be passed as arguments and occur as return values. In this new version, the advantage of functional programming with its declarative style is now supported in the language.

-------------------------------------------------------------------------------------------------

1. Transcendental Consciousness: TC, which can be experienced in the stillness of one’s awareness through transcending, is where the laws of nature begin to operate – it is the *home of all the laws of nature.*
2. Impulses Within the Transcendental Field: As TC becomes more familiar, more and more, intentions and desires reach fulfillment effortlessly, because of the hidden support of the laws of nature.
3. **Wholeness moving within Itself**: In Unity Consciousness, one finally recognizes the universe in oneself – that all of life is simply the impulse of one’s own consciousness. In that state, one effortlessly commands the laws of nature for all good in the universe.