### 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.

**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**

**Extracting Substreams and Combining Streams**

1. stream.concat(Stream) You can concatenate two streams with the static concat method of the Stream class:  
     
   Example:

Stream<Character> combined =

Stream.concat(characterStream("Hello"), **//EMPHASIZE!**

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<>();

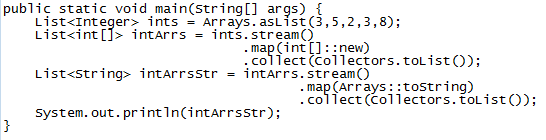
for (char c : s.toCharArray()) result.add(c);

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

}

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)

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



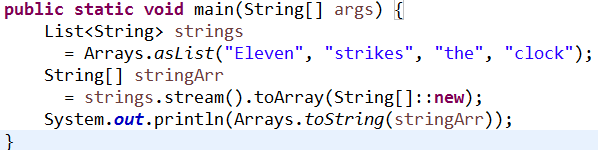
See Demo: lesson9.lecture.constructorref.IntArrayExample

**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]

See Demo: lesson9.lecture.constructorref.GenericArray.

**Collecting Results - DO THIS!!**

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);

Example: Collecting into a List

List<String> result = stream.collect(Collectors.toList());

Example: Collecting into a Set

Set<String> result = stream.collect(Collectors.toSet());

**Example: Collecting into a particular kind of Set (same idea for particular kinds of lists, maps)  
 TreeSet<String> result =   
 stream.collect(Collectors.toCollection(TreeSet::new));**

**Example: Collect all strings in a stream by concatenating them:  
 String result = stream.collect(Collectors.joining()); // DO THIS!!**

**//Look at interface Collector documentation.**

//separates strings by commas

String result = stream.collect(Collectors.joining(", "));

//prepares objects as strings before joining

String result = stream.map(Object::toString).collect(Collectors.joining(","));

**Note: Here instead of Object::toString you can use your own object type, like**

**Employee::toString**. **By polymorphism, either way works**!!

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

Note that there is also a ‘join’ method in the String class :

public static [String](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html) join([CharSequence](https://docs.oracle.com/javase/8/docs/api/java/lang/CharSequence.html" \o "interface in java.lang) delimiter,

[CharSequence](https://docs.oracle.com/javase/8/docs/api/java/lang/CharSequence.html)... elements)

Returns a new String composed of copies of the  CharSequence elements  joined together with a copy of the specified delimiter.

For example,

String message = String.join("-", "Java", "is", "cool");

// message returned is: "Java-is-cool"

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

Example: Collecting into a map – two typical examples. Here, **people is a Stream of Person** objects.

//key = id, value = name

Map<Integer, String> idToName

= people.collect(Collectors.toMap(Person::getId, Person::getName));

//key = id, value = the person object

Map<Integer, Person> idToPerson

= people.collect(Collectors.toMap(Person::getId, Function.identity()));

NOTE: identity is a static method on Function that returns a function that always returns its input argument. In the example, it is the function (Person p) -> p

**Example**: Collecting “summary statistics” for int-valued streams, providing sum,average, maximum, and minimum

IntSummaryStatistics summary

= words.collect(Collectors.summarizingInt(String::length));

double averageWordLength = summary.getAverage();

double maxWordLength = summary.getMax();

Similar SummaryStatistics classes are available for Double and Long types too: DoubleSummaryStatistics uses Collectors.summarizingDouble; LongSummaryStatistics uses Collectors.summarizingLong.

Note: IntSummaryStatistics **extracts int information** from an input Stream. The elements of the Stream must therefore be converted to (primitive) ints in order for summarizingInt to perform its tasks.

SummarizingInt expects an implementation of the ToIntFunction<T> interface:

interface ToIntFunction<T> {  
 int applyAsInt(T value);  
}

In the example, notice String::length is a realization of this interface:   
 str -> str.length()

**Can Streams Be Re-Used?**

* Once a terminal operation has been called on a stream, the stream becomes unusable, and if you do try to use it, you will get an IllegalStateException.
* But sometimes it would make sense to have a Stream ready to be used for multiple purposes.
* Example: We have a Stream<String> that we might want to use for different purposes:

Folks.friends.stream().filter(name -> name.startsWith(“N”))

* We may want to count the number of names obtained for one purpose, and output the names in upper case to a List, for another purpose. But once the stream has been used once, we can’t use it again.
* Solution #1 One solution is to place the stream-creation code in a method and call it for different purposes. S
* Solution #2 Another solution is to use a higher-order lambda to capture all the free variables in the first approach as parameters of some kind of a Function (might be a BiFunction, TriFunction, etc, depending on the number of parameters). See Reuse solution in package ***lesson9.lecture.streamreuse (Joseph Lerman - check this out)***

**Primitive Type Streams**

Streams cannot be used directly with primitive types, but there are variations of Stream that are specifically **designed for primitives:** int, double, and long. They are, respectively, IntStream, DoubleStream, and LongStream. To store primitive types short, char, byte, and boolean, use IntStream; to store floats, use DoubleStream.

Points about IntStream:

1. Creation methods are similar to those for Stream:
   1. IntStream ints = IntStream.of(1, 2, 4, 8);
   2. IntStream ones = IntStream.generate(() -> 1);
   3. IntStream naturalNums = IntStream.iterate(1, n -> n+1);
2. IntStream (and also LongStream) have static methods range and rangeClosed that

generate integer ranges with step size one:

// Upper bound is excluded

IntStream zeroToNinetyNine = IntStream.range(0, 100);  
  
// Upper bound is included

IntStream zeroToHundred = IntStream.rangeClosed(0, 100);

1. To convert a primitive type stream to an object stream, use the boxed() method:

Stream<Integer> integers = IntStream.range(0, 100).boxed();

1. To convert an object stream to a primitive type stream, there are methods mapToInt, mapToLong, and mapToDouble. In the examples, a Stream of strings is converted to an IntStream (of lengths).

Stream<String> words = ...;

IntStream lengths = words.mapToInt(String::length);

1. The methods on primitive type streams are analogous to those on object streams. Here are the main differences:
2. The toArray methods return primitive type arrays.
3. Methods that yield an optional result return an OptionalInt, OptionalLong, or

OptionalDouble. These classes are analogous to the Optional class, but they have

methods getAsInt, getAsLong, and getAsDouble instead of the get method.

1. There are methods sum, average, max, and min that return the sum, average,

maximum, and minimum. These methods are not defined for object streams. (Note that the functions max and min defined on an ordinary Stream, require a Comparator argument, and return an Optional.)

**Do the file ‘Lesson\_Do\_Fourth\_Streams\_From\_MPP\_Committee\_flatMap\_and\_Optional\_B’ Here!!**

**Creating a Lambda Library**

One of the biggest innovations in Java 8 is the ability to perform *queries* to extract or manipulate data in a Collection of some kind. Combining the use of lambdas and streams, one can almost always obtain the same efficient query statements one could expect to formulate using SQL (to obtain similar results).

Database Problem. You have a database table named Customer. Return a collection of the names of those Customers whose city of residence begins with the string “Ma”, arranged in sorted order.  
  
Solution. SELECT name FROM Customer WHERE city LIKE 'Ma%' ORDER BY name

Java Problem: You have a List of Customers. Output to a list, in sorted order, the names of those Customers whose city of residence begins with the string “Ma.”

Solution.

List<String> listStr = list.stream()

.filter(cust -> cust.getCity().startsWith("Ma"))

.map(cust -> cust.getName())

.sorted()

.collect(Collectors.toList());

**Turning Your Stream Pipeline into a Library Element**

To turn the Java solution in the previous slide into a reusable element in a Lambda Library, **identify the parameters that are combined together in your pipeline**, and consider those to be arguments for some kind of Java function-type interface (Function, BiFunction, TriFunction, etc).

Parameters:

* An input list of type List<Customer>
* A target string used to compare with name of city, of type String
* Return type: a list of strings: List<String>

These suggest using a BiFunction as follows:

public static final BiFunction<List<Customer>, String, List<String>> *NAMES\_IN\_CITY*

= (list, searchStr) //first two arguments up above.

-> list.stream()

.filter(cust -> cust.getCity().startsWith(searchStr))

.map(cust -> cust.getName())

.sorted()

.collect(Collectors.*toList*()); //the third argument (return value)

The Java solution can now be rewritten like this:

List<String> listStr = LambdaLibrary.NAMES\_IN\_CITY.apply(list, "Ma");

**// LambdaLibrary for example can be a Java class.**

# Connecting the Parts of Knowledge

# With the Wholeness of Knowledge

Lambda Libraries

|  |  |
| --- | --- |
| 1. Prior to the release of Java 8, extracting or manipulating data in one or more lists or other Collection classes involved multiple loops and code that is often difficult to understand.   2. With the introduction of lambdas and streams, Java 8 makes it possible to create compact, readable, reusable expressions that accomplish list-processing tasks in a very efficient way. These can be accumulated in a Lambda Library.  3*. Transcendental Consciousness* is the field that underlies all thinking and creativity, and, ultimately, all manifest existence.  4*. Impulses Within the Transcendental Field*. The hidden self-referral dynamics within the field of pure intelligence provides the blueprint for emergence of all diversity. This blueprint is formed from compact expressions of intelligence coherently arranged*.*  5*. Wholeness Moving Within Itself.* In Unity Consciousness, the fundamental forms out of which manifest existence is structured are seen to be vibratory modes of one’s own consciousness. | arrow |