

Lesson 9

The Stream API:

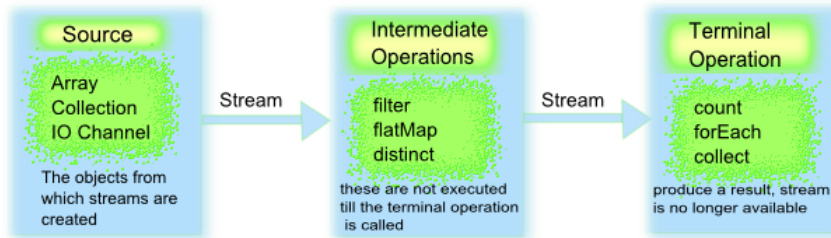
Solving Problems by Engaging Deeper Values of Intelligence

Wholeness of the Lesson: The stream API is an abstraction of collections that supports aggregate 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.

Maharishi's Science of Consciousness: 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.

What Are Streams and Why Are They Used?

Conceptually, a stream is a sequence of data items. Streams provide a view of data that lets you specify computations at a higher conceptual level than with collections. A stream is a concept, not a data structure. With a stream, you specify what you want to have done, not how to do it. See examples below.



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.

A. Imperative-style solution:

```
int count = 0;
for(String word : list) {
    if(word.length() > 12)
        count++;
}
```

Issues:

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

B. Functional-style solution (using ideas introduced in Lesson 2)

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

Advantages:

- i. Purely functional and threadsafe
- 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*.
4. *Java Implementation.* The methods on the `Stream` interface are implemented by the class `ReferencePipeline`. The method implementations involve a combination of technical operations internal to the `stream` package.

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 (by using `Stream.of(T...values)` method)
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. What happens if you do?

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:

```
Stream<Integer> strOfInt = Stream.of(1,3,5,7);
```

3. 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");
```

4. Two ways to obtain *infinite* streams: *generate* and *iterate* (remember stream operations are lazy)
 - a. 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);
```

why can `Math.random()` be used here? (See `StreamDemo2`)

- b. The `iterate` function accepts a seed value (of type `T`) and a `UnaryOperator<T>` argument.

```
interface UnaryOperator<T> {  
    T apply(T t);  
}
```

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.

2. `stream.skip(n)` The call `stream.skip(n)` *discards* the first `n` elements.
3. `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"),  
                  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();  
}
```

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();
```

2. The return value of `filter` is another `Stream`, so filters can be chained: Recall Lab 8:

```
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);
}
```
2. 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. `maps` can therefore be chained.

Example: Given a list

```
List<Integer> list
of Integers, obtain a list of Strings representing those Integers (T is Integer, R is
String)
```

```
List<String> strings = list.stream()
    .map(x -> x.toString())
    .collect(Collectors.toList())
```


Application: Using map with Constructor References

1. `Class::new` is a fourth type of method reference, where the method is the *new* operator.

Examples:

- A. `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.

- B. `String::new`

```
public class StringCreator {
    public static void main(String[] args) {
        Function<char[], String> myFunc = String::new;
        char[] charArray =
            {'s','p','e','a','k','i','n','g','c','s'};
        System.out.println(myFunc.apply(charArray));
    }
}
```

```
//output: speakingcs
```

See Demo: `lesson9.lecture.newstring`

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

c. `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?

```
public static void main(String[] args) {
    List<Integer> ints = Arrays.asList(3,5,2,3,8);
    List<int[]> intArrs = ints.stream()
        .map(int[]::new)
        .collect(Collectors.toList());
    List<String> intArrsStr = intArrs.stream()
        .map(Arrays::toString)
        .collect(Collectors.toList());
    System.out.println(intArrsStr);
}
```

See Demo: `lesson9.demos.StreamDemo1`

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`?

2. Turning a `Stream<String>` to `String[]` using 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);

public static void main(String[] args) {
    List<String> strings
        = Arrays.asList("Eleven", "strikes", "the", "clock");
    String[] stringArr
        = strings.stream().toArray(String[]::new);
    System.out.println(Arrays.toString(stringArr));
}
```

Output:
[Eleven, strikes, the, clock]

For more information, see

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>

See Demo: `lesson9.lecture.constructorref.GenericArray`.

Stream Operations, continued:

Use flatMap to Transform Each Element of a Substream and Flatten the Result

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 Stream result looks like a list of lists:

```
[[\J', \o', \e'], [\T', \o', \m'], [\A', \b', \e']].
```

“Flattening” this Stream means putting all elements together in a single list. 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`. What is the difference here? Think of Visitor Pattern compared with a stateless Polymorphic loop.
3. Example of `distinct`:

```
Stream<String> uniqueWords
    = Stream.of("merrily", "merrily", "merrily", "gently").distinct();
//output: ["merrily", "gently"]
```

4. 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()))));
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 Comparators with More Functional Style

[see package `lesson9.lecture.comparators1`]

1. 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());
```

2. `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 and more functionally.

```
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.

3. 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);
```

4. Comparator's comparing and thenComparing methods.

(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 or by salary.

```
Collections.sort(emps, (e1,e2) ->
{
    if(method == SortMethod.BYNAME) {
        return e1.name.compareTo(e2.name);
    } else {
        if(e1.salary == e2.salary) return 0;
        else if(e1.salary < e2.salary) return -1;
        else return 1;
    }
});
```

- This approach is “how”-oriented, and can be made more declarative (and nicer) by using the comparing and thenComparing methods of Comparator

```
Function<Employee, String> byName = e -> e.getName();
Function<Employee, Integer> bySalary = e -> e.getSalary();

public void sort(List<Employee> emps, final SortMethod method) {
    if(method == SortMethod.BYNAME) {
        Collections.sort(emps, Comparator.comparing(byName).thenComparing(bySalary));
    } else {
        Collections.sort(emps, Comparator.comparing(bySalary).thenComparing(byName));
    }
}
```

- **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.
- we can get rid of the if/else branching using a HashMap – see Lab 9, Problem 3.

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();
```

3. *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 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 true if any match is found, false otherwise; this one works well with parallel streams:

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


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:

```
int sum = 0;
for (int x : numbers) {
    sum += x;
}
```

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

```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

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:

```
int product = numbers.stream().reduce(1, (a, b) -> a * b);
```

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:

```
Optional<Integer> sum = numbers.stream().reduce((a, b) -> (a + b));
```

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 `Streams`. In this sense, `Streams` 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 (words being a `Stream<String>`)

```
String[] result = words.toArray(String[]::new);
```

Example: Collecting into a `List` (stream being a `Stream<String>`)

```
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());
```

```
//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. See demo `lesson9.lecture.collect`

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));
```

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. See Good solution in package `lesson9.lecture.streamreuse`
- 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`

Working with Optional – A Better Way to Handle Nulls

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 nulls.

2. The `orElse` method – if result is null, give alternative output using `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");
    }
}
```

//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
```

3. 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));  
}
```

Note: `ifPresent` take an instance of a `Consumer<>` [here](#).

Creating Your Own Optionals

Using *of* and *empty*. You can create an `Optional` instance in your own code using the static method *of*. However, if *of* is used on a null value, a `NullPointerException` is thrown, so the best practice is to use *of* together with *empty*, as in the following:

```
public static Optional<Double> inverse(Double x) {  
    return x == 0 ? Optional.empty() : Optional.of(1 / x);  
}
```

`Optional.empty()` simply creates an `Optional` with no wrapped value; in that case, the `isPresent` flag is set to `false`.

Using *ofNullable* The static method *ofNullable* lets you read in a possibly null value (unlike *of*) (read “*ofNullable*” as “do *of* but also allow nulls”).

Can use `orElse` and `Optional.ofNullable` together, as an alternative to the following pattern (typical example of this pattern is getting a JDBC Connection)

“If X is null, populate X, then return X.”

NOTE: `ofNullable` returns an `Optional` that embeds the specified value if non-null, otherwise returns an empty `Optional`

//OLD WAY

```
private static Connection conn = null;  
public Connection getConnection() throws SQLException {  
    if(conn == null) {  
        conn = DriverManager.getConnection(DB_URL, USERNAME, PASSWORD);  
        System.out.println("Got connection...");  
    }  
    System.out.println("Is conn null? " + (conn==null));  
    return conn;  
}
```

//NEW WAY

```
private Connection conn = null;  
private Connection myGetConn() {  
    try {  
        conn = DriverManager.getConnection(DB_URL, USERNAME, PASSWORD);  
        return conn;  
    } catch(SQLException e) {  
        throw new RuntimeException(e);  
    }  
}  
public Connection getConnection() {  
    return Optional.ofNullable(conn).orElse(myGetConn());  
}
```

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`:
 - a. `IntStream ints = IntStream.of(1, 2, 4, 8);`
 - b. `IntStream ones = IntStream.generate(() -> 1);`
 - c. `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);
```

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

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

4. 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);
```

5. The methods on primitive type streams are analogous to those on object streams. Here are the main differences:
 - a. The `toArray` methods return primitive type arrays.
 - b. 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.
 - c. 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`.)

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)
        -> list.stream()
            .filter(cust -> cust.getCity().startsWith(searchStr))
            .map(cust -> cust.getName())
            .sorted()
            .collect(Collectors.toList());
```

The Java solution can now be rewritten like this:

```
List<String> listStr = LambdaLibrary.NAMES_IN_CITY.apply(list, "Ma");
```

See the code in `lesson9.lecture.lambdalibrary`.

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.

