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 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.

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?

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."

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 mutable variable `count`, so not 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, so 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 *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 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);
```

3. Get a `Stream` from any sequence of arguments: (the `of` method accepts a varargs argument)

```
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);
```

   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))
```

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.

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. `map`s can therefore be chained.

   Example: Given a list `List<Integer>` 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())
                            .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 label 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.

```
@FunctionalInterface
public interface FuncIf {
    String strFunc(char[] chArray);
}
public class StringCreator {
    public static void main(String[] args) {
        FuncIf myFunc = String::new;
        char[] charArray =
              {'s','p','e','a','k','i','n','g','c','s'};
        System.out.println(myFunc.strFunc(charArray));
    }
}
//output:  speakingcs
```

   Note: In this case, `String::new` is short for the lambda expression `charArray -> new String(charArray)`, which is a realization of both the `Function` interface and the `FuncIf` interface. The ambiguity is resolved because, in the code, `myFunc` is typed as `FuncIf`.

   Best Practice: Don't create your own functional interface if there is already a Java-defined functional interface that could be used instead.

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.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`?

2. *Solving the "no generic arrays" limitation for streams*. For any generic type `T`, the compiler does not allow you to form arrays of the form `T[]`, or arrays like `List<T>[]`. In particular, you cannot create a `List<String>[]` array. (More on this point in Lesson 11.)

   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]
```

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:

Consider the following function, which takes a `String` like "ball" and turns it into a `Stream ['b', 'a', 'l', 'l']`.

```
public static Stream<Character> characterStream(String s) {
   List<Character> result = new ArrayList<>();
   for (char c : s.toCharArray()) result.add(c);
   return result.stream();
}
```

Suppose we apply this `characterStream` method 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 `result` Stream 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`.
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.

```
Function<String, Integer> byLength = x -> x.length();
Stream<String> longestFirst
          = words.stream().sorted(Comparator.comparing(byLength).reversed()
```

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);
```

3. Support for `Comparators` that are consistent with equals.

- Recall when we wanted to sort `Employees` (where an `Employee` has a `name` and a `salary`) by `name`, we need to consider also the `salary`, or else the `Comparator` is not consistent with `equals`.

```
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 much more declarative 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));
    }
}
```

- We can get rid of the `if/else` branching using a `HashMap`, together with a `Pair` class, in a clever way. See Lab 9, Problem 3.

Getting Outputs from Streams:

Reduction Methods

1. The last step in a pipeline of `Stream`s 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.
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 return a `NullPointerException` if not handled properly. An easy way to handle (but not the best):

   Example: `max`

```
Optional<String> largest = words.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 (if stream was empty, in this case).

   Example: `findFirst`

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

Working with `Optional` – A Better Way to Handle `Nulls`

1. The previous slide introduced `Optional` class. `Optional` was added to Java to make handling of `null`s 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.

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")));
  }
}
```

3. Use `ifPresent(Function)` 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));
}
```

4. Can use `orElse` and `Optional.ofNullable` as an alternative to this pattern (commonly used to get a JDBC Connection)

"If X is null, populate X. Return X."

```
//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()  {
    Optional<Connection> connWrapper = Optional.ofNullable(conn);
        return connWrapper.orElse(myGetConn());
}
```

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. 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:

$$(((0 + 2) + 1) + 4) + 3$$

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. Here is the line of code that does the job:

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

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

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

Here, the computation proceeds like this:

```
2 - (3 - (4 - (1 - 0)))
```

The problem here is that performing this computation in parallel gives a different result; subtractions are grouped differently for a parallel computation. For this reason, a best practice 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 produces the same output *when the stream is nonempty.* When the stream is empty, though, the reduce operation returns a null, which is then embedded in an Optional.

Main Point 1

When a Collection is converted to 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 regular 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);
```

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());

//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(","));
```

Example: Collecting into a `map` – two typical examples

```
//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()));
```

Example: Collecting "summary statistics" for number-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.

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 function 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`

Primitive Type Streams

For efficiency, certain `Stream`s are dedicated to primitive types: `int`, `double`, and `long` – they are, respectively, `IntStream`, `DoubleStream`, and `LongStream`. To store primitive types `short`, `char`, `byte`, and `boolean`, use `IntStream`; to store `float`s, 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 = Integer.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.

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 Lambda/Stream Pipeline into a Library Element

To turn this Java solution 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 – this blueprint is called the *Veda.*

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 ones own consciousness.