

# Lesson 9

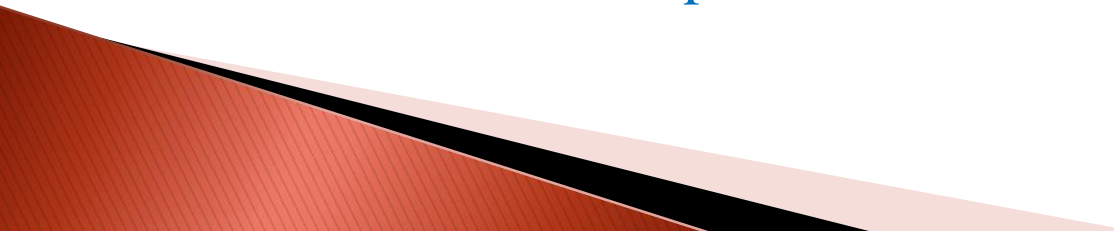
## The Stream API

*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.

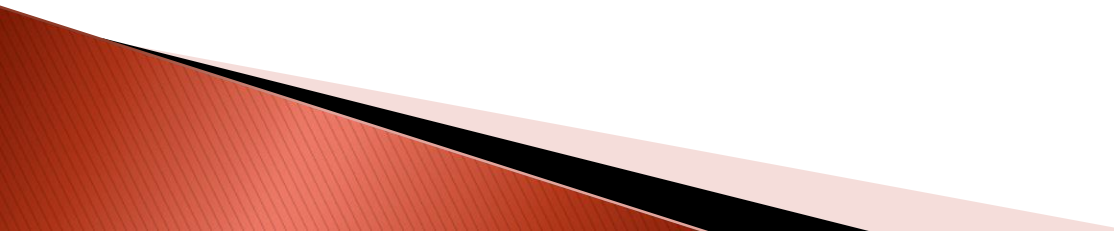
Deeper laws of nature are ultimately responsible for how things appear in the world. Efforts to modify the world from the surface level only lead to struggle and partial success. Affecting the world by accessing the deep underlying laws that structure everything can produce enormous impact with little effort. The key to accessing and winning support from deeper laws is going beyond the surface of awareness to the depths within.



# Outline

1. **Introduction to Java 8 Streams**
2. Streams: Basic Facts and a 3-Step Template
3. Different Ways to Create Streams
4. Intermediate Operations on Streams
5. Stateful Intermediate Operations
6. Terminal Operations
7. Working with the Optional class
8. Reduce
9. Collecting Results
10. Primitive Type Streams
11. Stream Reuse and Lambda Libraries

# What Are Streams and Why Are They Used?

- ▶ A stream is a way of representing data in a collection 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.
  - ▶ A stream represents a pipeline through which the data will flow and the functions to operate on the data.
- 

# Cont...

- ▶ 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  $> 12$ .
- ▶ 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*
- ▶ Imperative-style solution:

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

# Cont...

- ▶ Functional-style solution (using ideas introduced in Lesson 8)

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

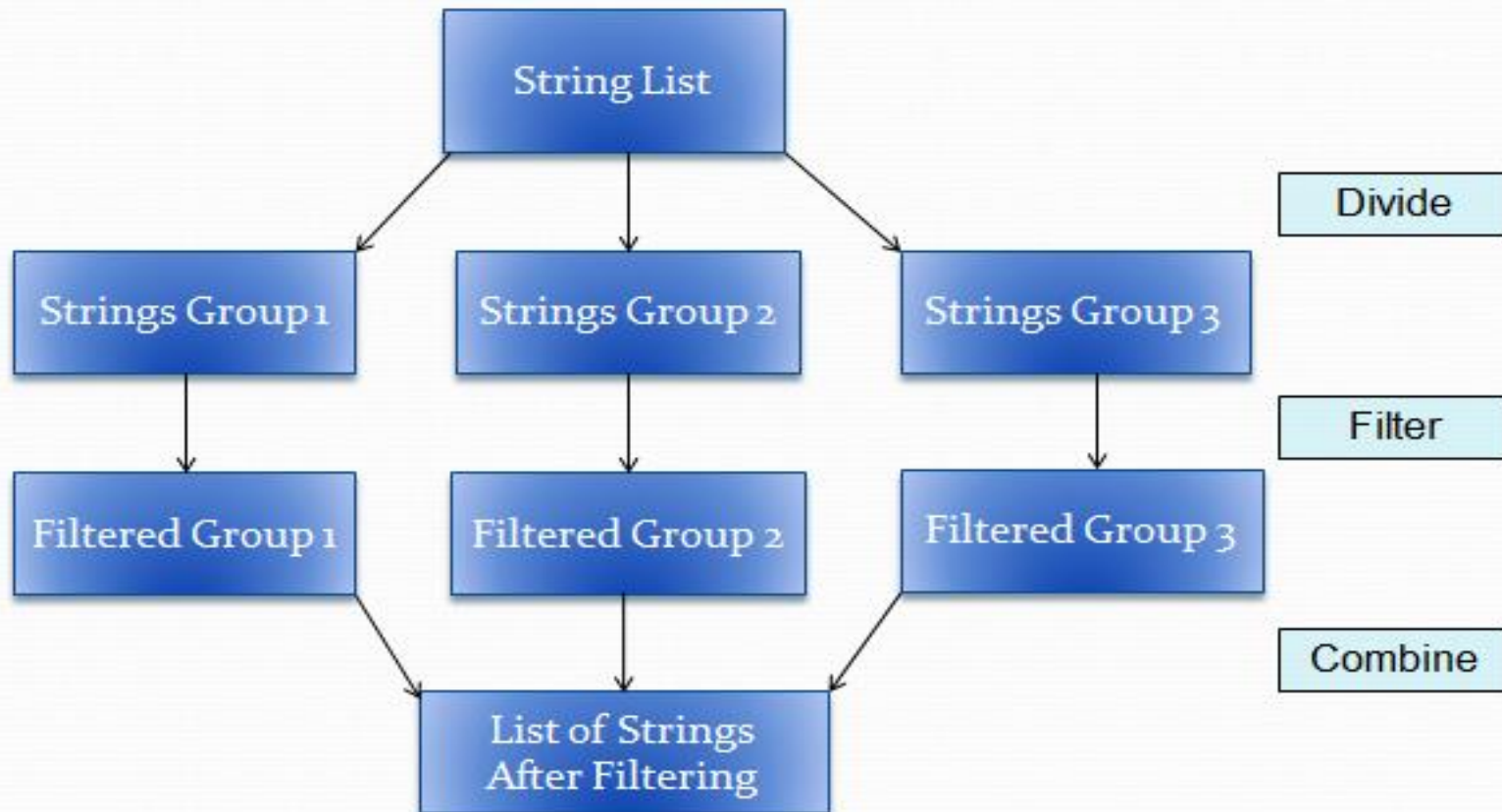
**// filter accepts a type of Predicate Functional Interface**

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

## Parallel-processing solution (on a multi-core processor, this is a real speedup)

```
final long count
```

```
= words.parallelStream()  
    .filter(w -> w.length() > 12)  
    .count();
```



## 2. Basic Facts About Streams

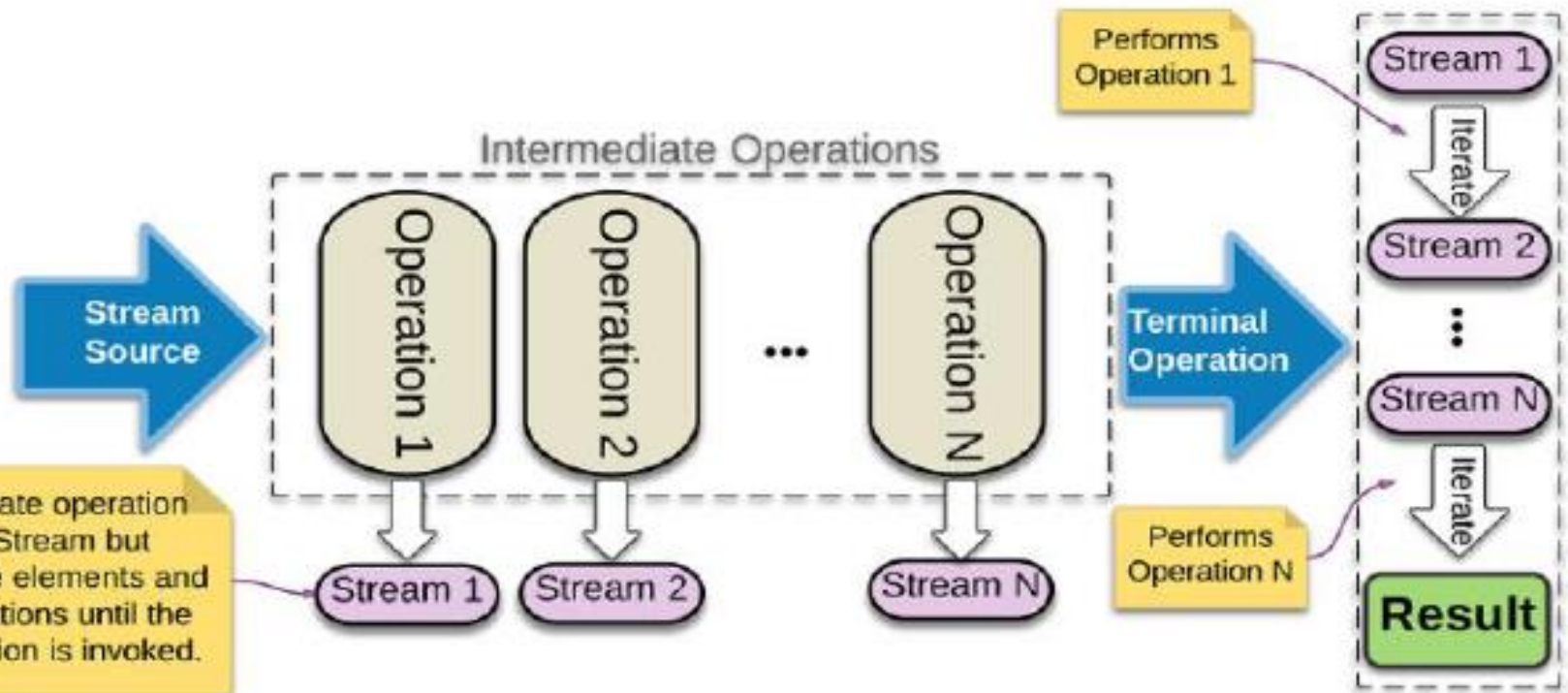
1. *Streams do not store the elements they operate on.* Typically they are stored in an underlying collection, or they may be generated on demand.
2. *Stream operations do not mutate their source.* Instead, they return new streams that hold the result.
3. *Java Implementation.* The methods on the Stream interface are implemented by the abstract class ReferencePipeline. The method implementations involve a combination of technical operations internal to the stream package.



4. 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 once you invoke the terminal operation.

## Stream Lazy Evaluation

LogicBIG.COM



# 3 – Step Template for Using Streams

1. *Create a stream*
2. *Create a pipeline of operations(Intermediate operations)*
3. *End with a terminal operation*

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
```

# Cont...

- ▶ Stream Operations : A stream supports two types of operations:
  - Intermediate operations
  - Terminal operations
- ▶ Intermediate operations are also known as *lazy* operations.
- ▶ Terminal operations are also known as *eager* operations. Operations are known as lazy and eager based on the way they pull the data elements from the data source.
- ▶ A lazy operation on a stream does not process the elements of the stream until another eager operation is called on the stream.

# Cont...

## Example 2: Sum the squared values of Odd numbers.

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
int sum = numbers.stream() //1. create the stream
```

```
.filter(n -> n % 2 == 1) // 2. intermediate operation – Filter odd numbers
```

```
.map(n -> n * n) // 2. intermediate operation - Multiples of odd numbers
```

```
.reduce(0, Integer::sum); // 3. terminal operation – Sum of multiplied odds
```



```
numbers.stream().filter(n -> n % 2 == 1).map(n -> n * n).reduce(0, Integer::sum)
```

# Ways of Creating Streams - Streams from Values

- ▶ The Stream interface contains the following two static of() methods to create a sequential Stream from a single value and multiple values:

- `<T> Stream<T> of(T t)`
- `<T> Stream<T> of(T...values)`

- ▶ The following snippet of code creates two streams:

// Creates a stream with one string elements

```
Stream<String> stream = Stream.of("Hello");
```

// Creates a stream with four strings

```
Stream<String> stream = Stream.of("Ken", "Jeff", "Chris",  
"Ellen");
```

// Compute the sum of the squares of all odd integers in the list

```
int sum = Stream.of(1, 2, 3, 4, 5)  
    .filter(n -> n % 2 == 1)  
    .map(n -> n * n)  
    .reduce(0, Integer::sum);  
System.out.println("Sum = " + sum);
```

# Ways of Creating Streams –Infinite Streams

- ▶ An infinite stream is a stream with a data source *capable* of generating infinite number of elements.
- ▶ The Stream interface contains the following two static methods to generate an infinite stream:

`<T> Stream<T> iterate(T seed, UnaryOperator<T> f)`

`<T> Stream<T> generate(Supplier<T> s)`

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.generate(Math::random)  
    .limit(5)  
    .forEach(System.out::println);
```

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

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

Example :

```
// Creates a stream of natural numbers
```

```
Stream<Long> naturalNumbers = Stream.iterate(1L, n -> n + 1);
```

```
// Creates a stream of odd natural numbers
```

```
Stream<Long> oddNaturalNumbers = Stream.iterate(1L, n -> n + 2);
```

- ▶ What do you do with an infinite stream? You understand that it is not possible to consume all elements of an infinite stream. This is simply because the stream processing will take forever to complete. Typically, you convert the infinite stream into a fixed-size stream by applying a limit operation that truncates the input stream to be no longer than a specified size.



# Extracting Sub streams 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.

// yields a stream with 100 random numbers.

```
Stream<Double> randoms =  
Stream.generate(Math::random).limit(100);
```

2. [stream.skip\(n\)](#) : The call `stream.skip(n)` *discards* the first `n` elements.

```
Stream.iterate(1L, n -> n + 2)
```

```
.skip(5)
```

```
.limit(5)
```

```
.forEach(System.out::println); // Output : 11 13 15 17 19
```



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

### Example:

```
List<String> list = Arrays.asList("Red", "Pink", "Black", "Blue", "Brown");  
List<String> list1 = Arrays.asList("Java", "Design Pattern", "Data Structures");  
Stream<String> res= Stream.concat(list.stream(), list1.stream());  
res.forEach(System.out::println);
```

#### ▶ **Output :**

- ▶ Red
- ▶ Pink
- ▶ Black
- ▶ Blue
- ▶ Brown
- ▶ Java
- ▶ Design Pattern
- ▶ Data Structures

```
Stream<String> res1= Stream.concat(list.stream()  
.filter(ele->ele.startsWith("B")), list1.stream());  
res1.forEach(System.out::println);
```

# stream.concat(Stream)

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();  
}
```

# Ways of Creating Streams -Streams from Arrays

- ▶ The Arrays class in the java.util package contains an overloaded stream() static method to create sequential streams from arrays.

```
// Creates a stream from an int array with elements 1, 2, and 3  
IntStream numbers = Arrays.stream(new int[]{ 1, 2, 3});
```

```
// Creates a stream from a String array with elements "Ken", and "Jeff"  
Stream<String> names = Arrays.stream(new String[] { "Ken",  
"Jeff" });
```

# Ways of Creating Streams -Streams from Collections

- ▶ The Collection interface contains the `stream()` and `parallelStream()` methods that create sequential and parallel streams from a Collection, respectively. The following snippet of code creates streams from a set of strings:

```
// Create and populate a set of strings
```

```
Set<String> names = new HashSet<>();
```

```
names.add("Ken");
```

```
names.add("jeff");
```

```
// Create a sequential stream from the set
```

```
Stream<String> sequentialStream = names.stream();
```

```
// Create a parallel stream from the set
```

```
Stream<String> parallelStream = names.parallelStream();
```

# Exercise 9.1

1. Use `Stream's iterate` method to produce a `Stream` consisting of all the odd natural numbers `1, 3, 5, ...`
2. Modify your solution so that your `Stream` consists of exactly these numbers: `9, 11, 13, 15`. Print your `Stream` to the console (somehow)

# Main Point 1

The iterate operation on a Stream makes it possible to generate an infinite sequence of values based on two pieces of data: The initial value, and a rule or principle that tells how one value should be transformed to the next.

The iterate method is an expression of the Principle of Diving. The "correct angle" is the initial value. "Letting go" has the right effect because there is an underlying rule that directs flow from the initial value to subsequent values. During meditation, what guides awareness from one level to the next – the underlying principle or rule – is the gentle pull to move toward what is most charming at each moment while awareness remains lively and awake.

# Intermediate and Terminal Operations on Streams : Refer `streamandoperations` package

Operation	Type	Description
<code>Distinct</code>	Intermediate	Returns a stream consisting of the distinct elements of this stream. Elements <code>e1</code> and <code>e2</code> are considered equal if <code>e1.equals(e2)</code> returns true.
<code>filter</code>	Intermediate	Returns a stream consisting of the elements of this stream that match the specified predicate.
<code>flatMap</code>	Intermediate	Returns a stream consisting of the results of applying the specified function to the elements of this stream. The function produces a stream for each input element and the output streams are flattened. Performs one-to-many mapping.
<code>limit</code>	Intermediate	Returns a stream consisting of the elements of this stream, truncated to be no longer than the specified size.
<code>map</code>	Intermediate	Returns a stream consisting of the results of applying the specified function to the elements of this stream. Performs one-to-one mapping.
<code>peek</code>	Intermediate	Returns a stream whose elements consist of this stream. It applies the specified action as it consumes elements of this stream. It is mainly used for debugging purposes.
<code>skip</code>	Intermediate	Discards the first <code>n</code> elements of the stream and returns the remaining stream. If this stream contains fewer than <code>n</code> elements, an empty stream is returned.
<code>sorted</code>	Intermediate	Returns a stream consisting of the elements of this stream, sorted according to natural order or the specified <code>Comparator</code> . For an ordered stream, the sort is stable.
<code>allMatch</code>	Terminal	Returns true if all elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
<code>anyMatch</code>	Terminal	Returns true if any element in the stream matches the specified predicate, false otherwise. Returns false if the stream is empty.
<code>findAny</code>	Terminal	Returns any element from the stream. An empty <code>Optional</code> object is for an empty stream.
<code>findFirst</code>	Terminal	Returns the first element of the stream. For an ordered stream, it returns the first element in the encounter order; for an unordered stream, it returns any element.
<code>noneMatch</code>	Terminal	Returns true if no elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
<code>forEach</code>	Terminal	Applies an action for each element in the stream.
<code>reduce</code>	Terminal	Applies a reduction operation to computes a single value from the stream.

# Stream Operations:

## Use filter to Extract a Substream that Satisfies Specified Criteria

filter accepts as its argument as 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();
```

- ▶ The following snippet of code uses a stream of people and filters in only females. It maps the females to their names and prints them on the standard output.

```
Person.persons()  
    .stream()  
    .filter(Person::isFemale)  
    .map(Person::getName)  
    .forEach(System.out::println);
```

- ▶ The following snippet of code applies two filter operations to print the names of all males having income more than 5000.0:

```
Person.persons()  
    .stream()  
    .filter(Person::isMale)  
    .filter(p -> p.getIncome() > 5000.0)  
    .map(Person::getName)  
    .forEach(System.out::println);
```



# Stream Operations:

## Use map to Transform Each Element of a Substream

- ▶ map accepts a Function interface. Typical special case of the Function interface is

```
interface Function<T,R> {  
    R apply(T t);  
}
```

- ▶ 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 = Arrays.asList("New", "Update", "Save");
```

```
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));  
    }  
}
```

▶ //

- ▶ See Demo: lesson9.lecture.newstring,
- ▶ Refer : Streamandoperations.ConstructorReference.java
- ▶ **Note:** In this case, String::new is short for the lambda expression  
charArray -> new String(charArray), which is a realization  
of the Function interface.

## Exercise 9.2

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);  
}
```

# Solution

See `lesson9.lecture.constructorref.IntArrayExample`

## Main Point 2

Java 8 makes it possible to dynamically construct objects using lambdas or method references. An application of this feature is that a developer can construct an entire Stream of new objects with very little code, by combining `map` with a constructor method reference.

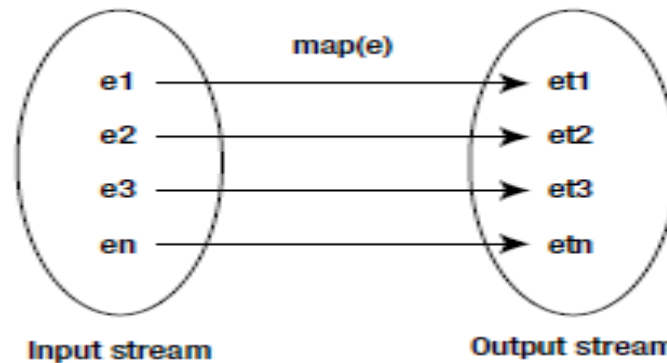
This combination of new Java features reflects a deeper quality that is found in the field of intelligence itself: Intelligence naturally tends toward creative expression.

Contact with our own deepest levels of intelligence through transcending results in enhanced ability for creative expression.

# Stream Operations, continued:

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

- ▶ map operation that facilitates a one-to-one mapping. Each element of the input stream is mapped to an element in the output stream.



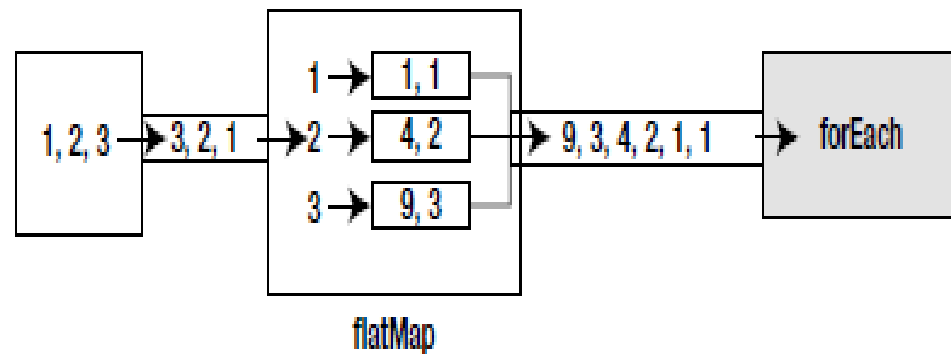
### key difference between map() vs flatMap() in Java 8:

- The function you pass to map() operation returns a single value.
- The function you pass to flatMap() operation returns a Stream of value.
- flatMap() is combination of map and flat operation.
- map() is used for transformation only, but flatMap() is used for both transformation and flattening.

# flatMap

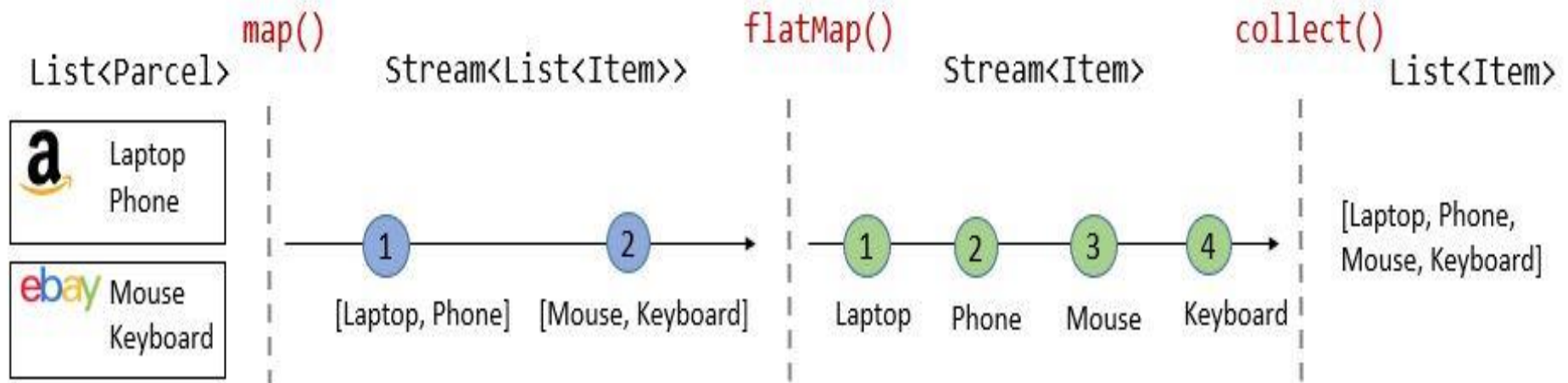
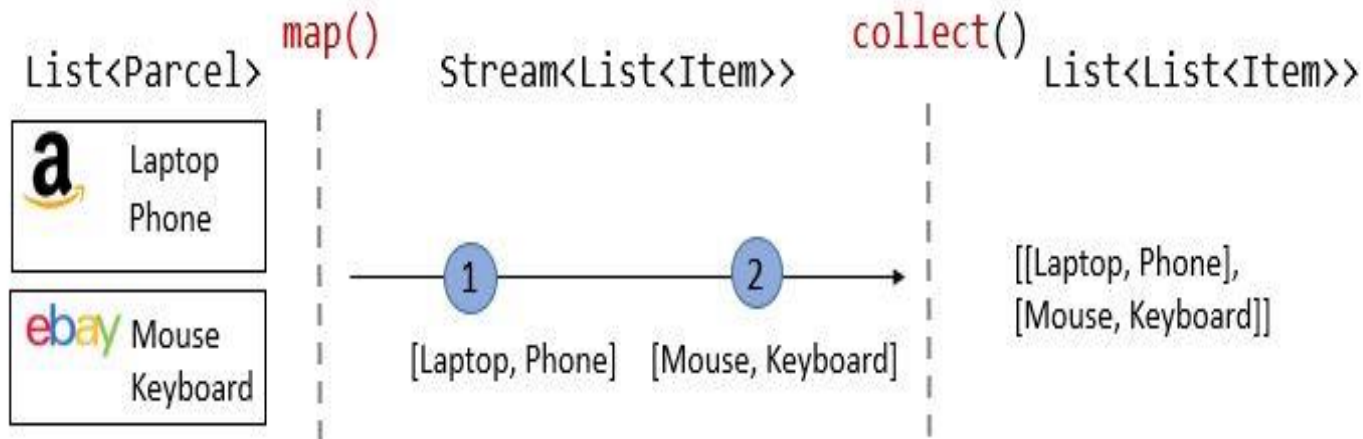
- ▶ The Streams API also supports one-to-many mapping through the flatMap operation
- ▶ We illustrate flatMap with an example:
- ▶ “Flattening” this Stream means putting all elements together in a single list. This is accomplished using flatMap in place of map:
- ▶ Refer : Streamandoperations. MapFlatMapDemo.java

```
Stream.of(1, 2, 3)  
  .flatMap(n -> Stream.of(n, n * n))  
  .forEach(System.out::println);
```





# Map FlatMap Difference



# Stateful Transformations

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

```
Stream<String> uniqueWords
```

```
= Stream.of("merrily", "merrily", "merrily", "gently").distinct();
```

```
//output: ["merrily", "gently"]
```

- ▶ 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”.

# Day -2

Stream API

# Implementing Comparators with More Functional Style

[see package `lesson9.lecture.comparators1`]

- ▶ 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());
```

- ▶ `Comparator.comparing` takes a `Function<T,R>` argument. The type `T` is the type of the object being compared – in the example, `T` is `String`. The type `R` is the type of object that will actually be compared - since we are comparing lengths of words, the type `R` 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.

- ▶ 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);
```

- ▶ Support for Comparators that are *consistent with equals*.  
(Review consistency with equals issue in lesson8.lecture.exercise.employeeeocode 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.

```
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 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 about `comparing` and `thenComparing`:

- ▶ `comparing` is a static method of `Comparator`, and therefore cannot be chained
- ▶ The `thenComparing()` method is a default method. It is used to specify a secondary comparison if two objects are the same in sorting order based on the primary comparison. It can be chained.
- ▶ we can get rid of the if/else branching.

## Exercise 9.3

Use the `comparing` and `thenComparing` methods of `Comparator` to sort a list of `Accounts` first by `balance`, then by `ownerName`. See `lesson9.exercise_3` in the `InClassExercises` project.

```
public class Account {  
    private String ownerName;  
    private int balance;  
    private int acctId;  
    public Account(String owner, int bal, int id) {  
        ownerName = owner;  
        balance = bal;  
        acctId = id;  
    }  
}
```



# Solution

```
List<Account> sorted  
    = accounts.stream()  
        .sorted(Comparator.comparing((Account a) -> a.getBalance())  
            .thenComparing(a -> a.getOwnerName()))  
        .collect(Collectors.toList());
```

```
Collections.sort(accounts,  
    Comparator.comparing(Account::getBalance)  
        .thenComparing(Account::getOwnerName));  
System.out.println(accounts);
```



# Terminal Operations on Stream

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();
```

# Terminal Operations

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`.

# Terminal Operations

**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();
```

# Optional Class

- ▶ null references have been historically introduced in programming languages to generally signal the absence of a value.
- ▶ Java 8 introduces the class `java.util.Optional<T>` to model the presence or absence of a value.
- ▶ You can create `Optional` objects with the static factory methods `Optional.empty`, `Optional.of`, and `Optional.ofNullable`.
- ▶ The `Optional` class supports many methods such as `map`, `flatMap`, and `filter`, which are conceptually similar to the methods of a stream.
- ▶ Using `Optional` forces you to actively unwrap an optional to deal with the absence of a value; as a result, you protect your code against unintended null pointer exceptions.
- ▶ Using `Optional` can help you design better APIs.

## The methods of the Optional class

Method	Description
<code>empty</code>	Returns an empty Optional instance
<code>filter</code>	If the value is present and matches the given predicate, returns this Optional; otherwise returns the empty one
<code>flatMap</code>	If a value is present, returns the Optional resulting from the application of the provided mapping function to it; otherwise returns the empty Optional
<code>get</code>	Returns the value wrapped by this Optional if present; otherwise throws a <code>NoSuchElementException</code>
<code>ifPresent</code>	If a value is present, invokes the specified consumer with the value; otherwise does nothing
<code>isPresent</code>	Returns true if there is a value present; otherwise false
<code>map</code>	If a value is present, applies the provided mapping function to it
<code>of</code>	Returns an Optional wrapping the given value or throws a <code>NullPointerException</code> if this value is null
<code>ofNullable</code>	Returns an Optional wrapping the given value or the empty Optional if this value is null
<code>orElse</code>	Returns the value if present or the given default value otherwise
<code>orElseGet</code>	Returns the value if present or the one provided by the given Supplier otherwise
<code>orElseThrow</code>	Returns the value if present or throws the exception created by the given Supplier otherwise

# Working with Optional – A Better Way to Handle Nulls

- ▶ 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 easier than  
if (value != null) // Imperative way  
    value.someMethod();
- ▶ The Optional class, however, supports other techniques that are superior to checking nulls.

## ► 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(null)));
}

//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
```

# Working with Optional

- ▶ 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)); ( if no match -> nothing printed on console)  
}
```



# Summary about Optional result

- ▶ max, min, findFirst, findAny search a stream for particular values and return Optional.

- ▶ Different ways to make use of Optional result

**1. isPresent()** : Check the result has an element using isPresent() return boolean value, and retrieve the result by invoking get() method.

**2. orElse()** : You want to say something to the client, if the result is null use orElse("Message").

**3. ifPresent()** : Accepts of Consumer interface type, if the optional object has a result, perform actions using ifPresent(Consumer Type)

# 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.

# ofNullable used with orElse/orElseGet

When the `orElse` clause needs to obtain a value from a method call, use `orElseGet` instead.

- ▶ **Example.** Use `orElseGet` as a way to implement a lazy singleton. The following is an example in which a Connection object (JDBC) is implemented as a lazy singleton within a ConnectionManager class.

```
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).orElseGet(this::myGetConn);
}
```

# Difference Between `orElse` and `orElseGet`

- ▶ The description for the `orElse()` method is *"Return the value if present, otherwise return other."*
- ▶ While, the description for the `orElseGet()` method is *"Return the value if present, otherwise invoke given behavior of Supplier Functional Interface logic and return the result of that invocation."*

In the code on previous slide, if we write instead

```
public Connection getConnection() {  
    return Optional.ofNullable(conn).orElse(myGetConn());  
}
```

then, when `conn` is not null `conn` will be assigned to `Optional`.

If `conn` is null, `orElse` will cause `myGetConn()` to be invoked.

In the same scenario, the following code will *not* cause `myGetConn` to be invoked if `conn` is not null.

```
public Connection getConnection() {  
    return Optional.ofNullable(conn).orElseGet(() ->  
        myGetConn());  
}
```

# Optional's flatMap

Java 7 Way : Consider ordinary code for seeing if a Person from Fairfield can be found in a list of Persons:

```
private static boolean personFromFairfield(List<Person> persons) {  
    Person foundPerson = null;  
    for(Person p: persons) {  
        if(p != null) {  
            Address addr = p.getAddress();  
            if(addr != null) {  
                String city = addr.getCity();  
                if(city != null) {  
                    if(city.equals("Fairfield")) {  
                        foundPerson = p;  
                    }  
                }  
            }  
        }  
    }  
    return foundPerson != null;  
}
```

- ▶ **Java 8 Way** - Using `Optionals` with `flatMap` completely eliminates these (unnecessary) null checks

```
private static boolean personFromFairfield(List<Optional<Person>> persons) {  
    for(Optional<Person> p: persons) {  
        if(p.flatMap(x -> x.getAddress())  
            .flatMap(x -> x.getCity())  
            .orElse("").equals("Fairfield")) {  
            return true;  
        }  
    }  
    return false;  
}
```

1. If an `Optional opt` is empty, `opt.flatMap(mapper)` returns an empty `Optional`
2. To get this nice behavior, you have to set up your classes so that they are using `Optional`. See Demo code

[lesson9.lecture.optional\\_flatmap.usingoptionals.optionalway](#)  
[lesson9.lecture.optionaldemo](#), [lesson9.lecture.stremoperations](#),  
[lesson9.lecture.optional](#)

# Monads in Java 8

- ▶ A *monad* is a special data structure, available in some languages, that serves as a wrapper class, to support various operations.
- ▶ In Java, these operations include filter and map.
- ▶ Monads are designed to support chaining operations, so that the output of each monad operation is another monad.
- ▶ Monads are considered to be a key construct in functional languages
- ▶ Every monad has a *unit* operation *of* and a *flatMap* operation
- ▶ More formal definition: *Monads are chainable container types that trap values or computations and allow them to be transformed in confinement.*
- ▶ In Java 8, two monads were introduced:
  - **Optional**
  - **Stream**

# 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:

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

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:

$$\text{sum1} = (0 + 2) + 3 \quad \text{sum2} = (0 + 4) + 1$$

$$\text{combined} = \text{sum1} + \text{sum2} = 10$$

- ▶ Java SE provides the fork/join framework to implement parallel computing.
- ▶ 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:  
$$(((0 - 2) - 1) - 4) - 3 \quad //\text{output: } -10$$
- ▶ 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:  
$$\text{diff1} = (0 - 2) - 3 \quad \text{diff2} = (0 - 1) - 4 \quad \text{combined} = \text{diff1} - \text{diff2} = 0$$
- ▶ 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.

## Exercise 9.4


Use `reduce` to concatenate the `Strings` in the `Stream` below to form a single, space-separated `String`. Print the result to the console. See `lesson9.exercise_4` in the `InClassExercises` project.

```
public static void main(String[] args) {  
    Stream.of("A", "good", "day", "to", "write", "some", "Java");  
}
```

# 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.

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**

```
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));
```

**Refer API : <https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html>**

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](#), [lesson9.lecture.collect.streamoperations](#)**

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.stream.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) int 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()
```



# Exercise 9.5

Use `DoubleSummaryStatistics` to output to the console the top test score, lowest test score, and average among all test scores in a given list. See `lesson9.exercise_5` in `InClassExercises` project.

```
public class ExamData {  
    private String studentName;  
    private double testScore;  
    public ExamData(String name, double score) {  
        studentName = name;  
        testScore = score;  
    }  
}
```

```
public static void main(String[] args) {  
    List<ExamData> data = new ArrayList<ExamData>() {  
        {  
            add(new ExamData("George", 91.3));  
            add(new ExamData("Tom", 88.9));  
            add(new ExamData("Rick", 80));  
            add(new ExamData("Harold", 90.8));  
            add(new ExamData("Ignatius", 60.9));  
            add(new ExamData("Anna", 77));  
            add(new ExamData("Susan", 87.3));  
            add(new ExamData("Phil", 99.1));  
            add(new ExamData("Alex", 84));  
        }  
    };  
}
```

# 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`

# 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 `float`, use `DoubleStream`.

Points about `IntStream`:

- ▶ Creation methods are similar to those for `Stream`:

```
IntStream ints = IntStream.of(1, 2, 4, 8);
```

```
IntStream ones = IntStream.generate(() -> 1); // infinite stream
```

```
IntStream naturalNums = IntStream.iterate(1, n -> n+1); // infinite stream
```

- ▶ `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);
```

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

```
Stream<Integer> integers = IntStream.range(0, 100).boxed();  
integers.forEach(System.out::println);
```

- ▶ 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);
```

- ▶ The methods on primitive type streams are analogous to those on object streams. Here are the main differences:
  - The `toArray` methods return primitive type arrays.
  - 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.
  - 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.”

# 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.

