

A large, two-story, light blue building with a red-tiled roof and a central tower, surrounded by green lawns and trees under a clear blue sky.

MAHARISHI INTERNATIONAL UNIVERSITY

Engaging the Managing Intelligence of Nature

Computer Science Department

**CS401 Modern Programming
Practices (MPP)
Bright Gee Varghese**

Lecture 9: The Stream API

*Solving Problems by Engaging Deeper
Values of Intelligence*

Wholeness Statement

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 lead to struggle and only 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?

1. *What They Are.* 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. Why They Are Used. To understand why they are used, consider the following problem:

Problem. Given a list of words (say from a book), count how many of the words have length > 12.

Imperative-style solution

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

Issues:

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

Functional-style solution

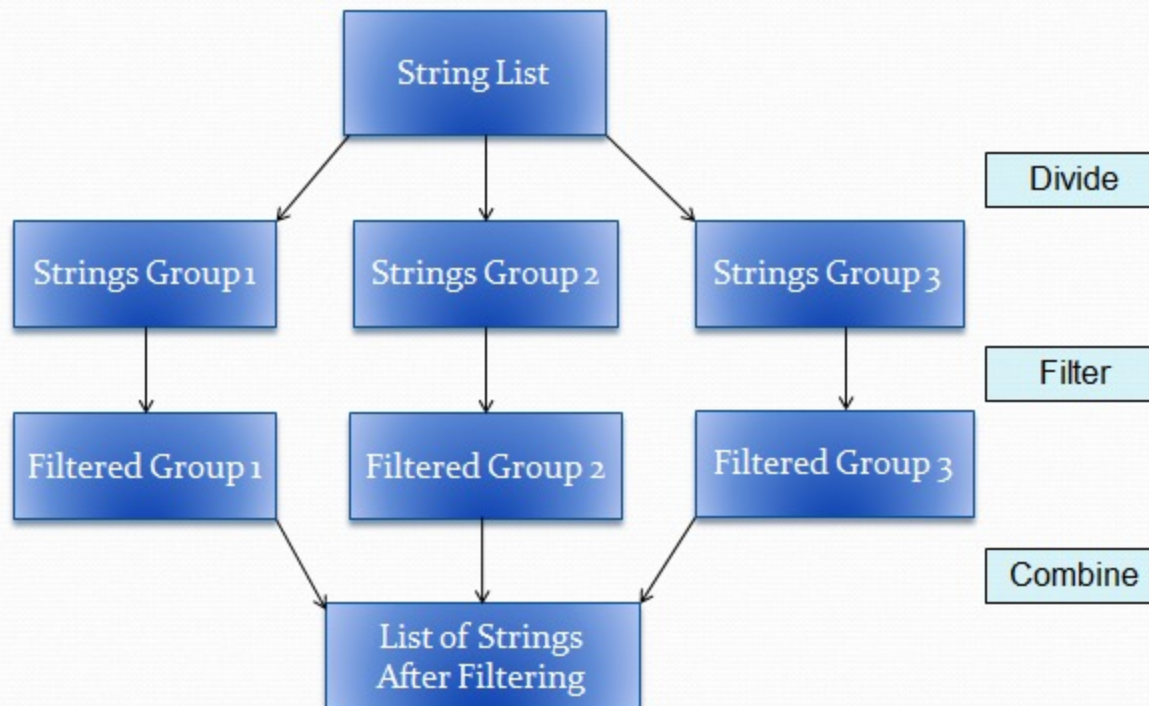
```
final long count
    = words.stream()
        .filter(w -> w.length() > 12)
        .count();
```

Advantages:

- i. Purely functional, so threadsafe
- ii. Makes no commitment to an iteration path, so more parallelizable
- iii. Declarative style – “what, not how”
- iv. With Java 8 or later it is easy to transform into a parallel processing solution

Parallel-processing solution

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




```
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.ForkJoinPool;

public class ThreadPoolDemo {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("one", "two");

        System.out.println(ForkJoinPool.commonPool().getParallelism());
        //Alternatively, you can use your own pool
        ForkJoinPool myPool = new ForkJoinPool(8);
        myPool.submit(
            () -> list.parallelStream()
                .forEach(
                    System.out::println
                )
        );
        System.out.println(myPool.getParallelism());
    }
}
```

Ref: <https://stackoverflow.com/questions/52287717/how-to-specify-forkjoinpool-for-java-8-parallel-stream>

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

Facts About Streams

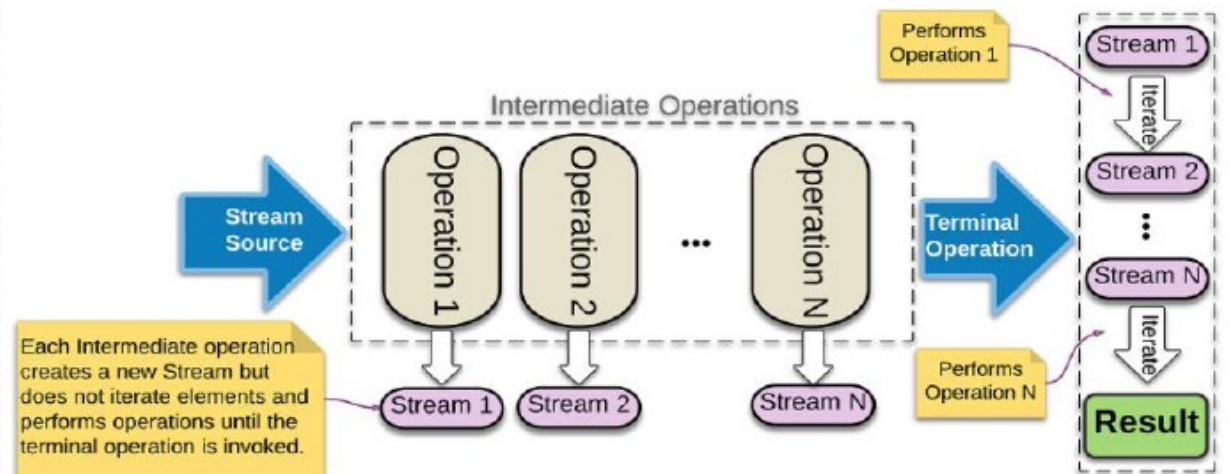
1. *Streams do not store the elements they operate on.*
Typically they are stored in an underlying collection, or they may be generated on demand.
2. *Stream operations do not mutate their source.* Instead, they return new streams that hold the result.
3. *Java Implementation.* The methods on the `Stream`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

4. *Stream operations are lazy whenever possible.* Elements in the stream are accessed in a way analogous to streaming video – only the elements that are absolutely necessary are accessed, and the intermediate operations act on those. When more elements are needed, they are accessed. Diagram below shows **User View**. See `lesson9.lecture.lazystream.UserView.java`

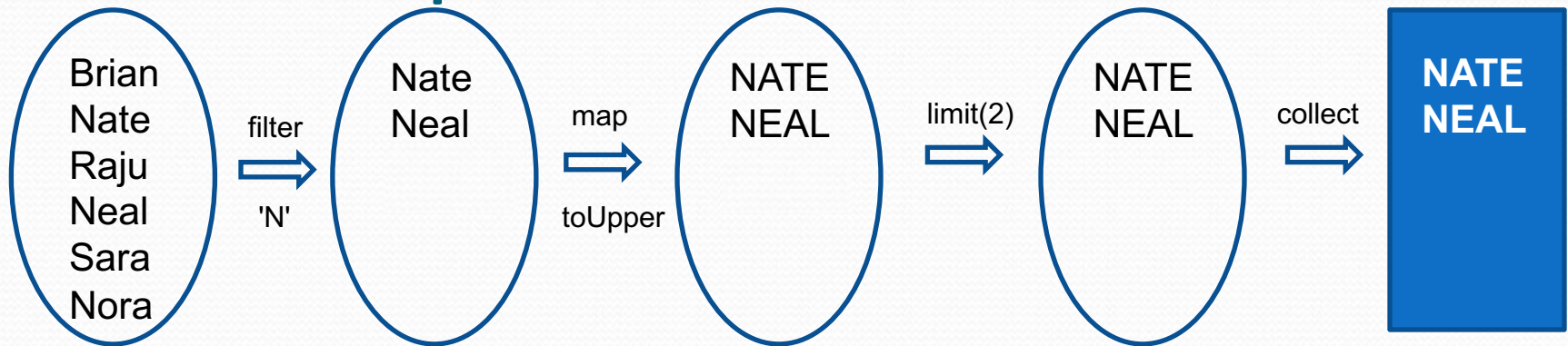
```
public List<String> find(List<String> list,
                        String letter) {
    return list.stream()
        .filter(name -> starts(name, letter))
        .map(name -> toUpper(name))
        .limit(2)
        .collect(Collectors.toList());
}
```

User view of a Stream pipeline

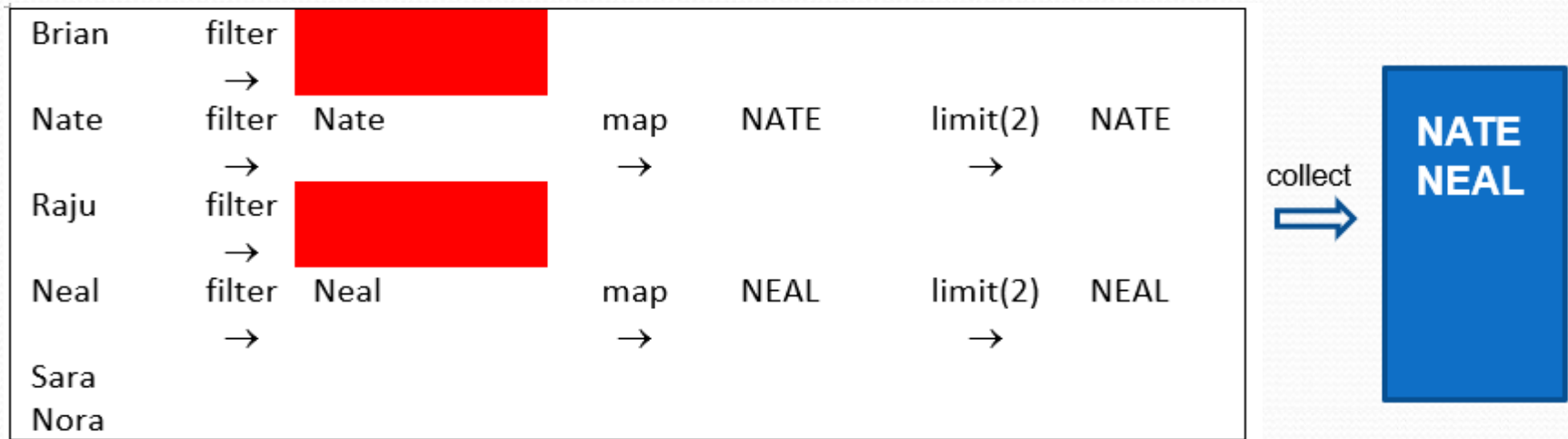
Stream Lazy Evaluation



Stream Pipelines: Under the Hood



User View: New Stream after Each Operation



Under the Hood: *Stream Fusion*.

Loops are fused to produce single stream. Demo:

`lesson9.lecture.lazystream.LoopFusion.java`

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 performs some stream transformation and returns a new stream. Called *intermediate operations*.
3. *End with a terminal operation.* The terminal operation produces a result. It also forces lazy execution of the operations that precede 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
```


Terminal Operations

- anyMatch()
- allMatch()
- noneMatch()
- collect()
- count()
- findAny()
- findFirst()
- forEach()
- min()
- max()
- reduce()
- toArray()
- sum()

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

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 of objects using the static `of` method:


```
Integer[] arrOfInt = {1, 3, 5, 7};  
Stream<Integer> strOfInt = Stream.of(arrOfInt);
```

Cannot do the same for `int[]`

```
int[] arrOfInt = {1, 3, 5, 7};  
    //one-element Stream  
Stream<int[]> strOfInt = Stream.of(arrOfInt);
```


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");
```



```
Stream<Integer> integerStream = Arrays.asList(1, 2, 3).stream();  
System.out.println(integerStream.count()); //3
```

```
Stream<int[]> stream = Stream.of(new int[]{10, 20});  
System.out.println(stream.count()); //1
```

```
int marks [] = {1, 2, 3};  
IntStream intStream = Arrays.stream(marks);  
System.out.println(Arrays.stream(marks).count()); //3  
System.out.println(IntStream.of(new int[]{10, 20}).count()); //2
```

```
public class Main {  
    public static void main(String[] args) {  
        int marks [] = {1, 2, 3};  
        IntStream intStream = Arrays.stream(marks);  
        System.out.println(  
            Arrays.stream(marks) //IntStream  
                .sum()  
        );  
        System.out.println(  
            IntStream.of(marks) //IntStream  
                .sum()  
        );  
    }  
}
```


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);
```

— ~~See lesson9.lecture.generate~~

- 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 `Integer`)

```
Stream<Integer> stream2 = Stream.iterate(1, n -> n + 1);
```

Here, 1 is the first element of the generated `Stream`. Then 1 is substituted for `n` in the lambda to obtain the next element: $1 + 1 = 2$. Then 2 is substituted for `n` to obtain the next. And so on.

Note: You cannot use `int` in place of `Integer`. We discuss `Streams` based on primitives later in the lesson.

Demo: `lesson9.lecture.iterate`


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);  
// Produces a stream with 100 random numbers.
```

2. `stream.skip(n)` The call `stream.skip(n)` *discards* the first `n` elements.



```
Stream.of(1, 2, 3, 4, 5)
    .skip(2)
    .forEach(System.out::println);
```

Output

3

4

5


3. Stream.concat(Stream, 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"));  
  
// Produces the Stream  
//    ['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd']
```

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<Character> characterStream1 =  
    "Welcome".chars()  
        .mapToObj(letter -> (char)letter);  
Stream<Character> characterStream2 =  
    "to MIU".chars()  
        .mapToObj(letter -> (char)letter);  
Stream.concat(characterStream1, characterStream2)  
    .forEach(System.out::println);
```


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)

Solution

```
10      List<Integer> integers =
11          Stream.iterate( seed: 1, n->n+2)
12              .limit( maxSize: 10)
13              .collect(Collectors.toList());
14      System.out.println(integers);
15      System.out.println(
16          Stream.iterate(
17              seed: 9,
18                  n -> n <= 15,
19                  n -> n + 2
20              )
21          .collect(Collectors.toList())
22      );
```


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.

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

Intermediate Operations on Streams:

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

- `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();
```

- The return value of `filter` is another `Stream`, so filters can be chained:

```
words.stream()  
    .filter(name -> name.contains(" " + c))  
    .filter(name -> !name.contains(" " + d))  
    .filter(name -> name.length() == len)  
    .count();
```

Intermediate Operations on Streams :

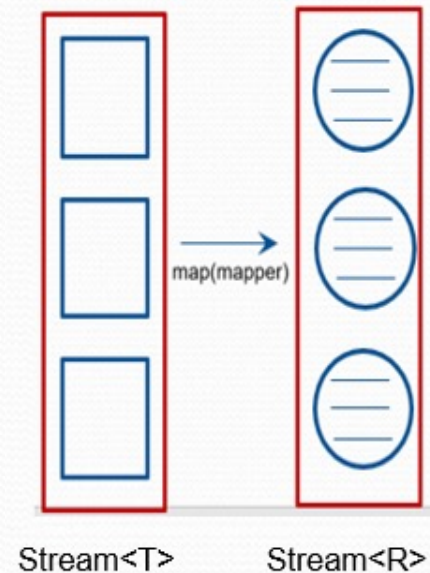
Use *map* to Transform Each Element of a Stream

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

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



- A map accepts a Function as input (sometimes called a "mapper") 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 Method References

1. *Class::new* is a fourth type of method reference, where the method is the *new* operator. Typical translation: `z -> new Class(z)`

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());
```

//Output: a Stream of labeled Buttons, converted to a list

B. String::new. Again, the choice of `String` constructor depends on context.

```
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));  
        System.out.println(new String(charArray));  
    }  
}
```

//output: speakingcs

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

See Demo: `lesson9.lecture.newstring`

- 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)

```
Function<Integer, int[]> function = n -> new int[n];  
//      Function<Integer, int[]> function = int[]::new;  
int scores[] = function.apply(3);  
scores[0] = 10;  
scores[1] = 11;  
scores[2] = 20;  
System.out.println(scores.length);  
System.out.println(Arrays.toString(scores));
```

2. *Array constructor reference and the toArray method*

Problem

```
Stream<String> stringStream = // create Stream

//Can obtain a list of Strings like this
List<String> stringList = stringStream.collect(Collectors.toList());

//How to obtain an array of Strings? (Not like this!)
String[] vals = stringStream.toArray(); //compiler error
```

Solution: Use a constructor reference to specify the correct type:

```
String[] vals = stringStream.toArray(String[]::new);
public static void main(String[] args) {
    List<String> strings
    = Arrays.asList("Eleven", "strikes", "the", "clock");
    String[] stringArr2 = strings.stream().toArray(String[]::new);
    System.out.println(Arrays.toString(stringArr2));
}
//Output: [Eleven, strikes, the, clock]
```

See Demo: [lesson9.lecture.constructorref.GenericArray](#).


```
List<String> fruits = Arrays.asList("Apple", "Orange", "Mango");  
Stream<String> stream = fruits.stream();
```

```
System.out.println(  
    Arrays.toString(  
        stream.toArray(new IntFunction<String[]>() {  
            @Override  
            public String[] apply(int value) {  
                return new String[value];  
            }  
        })  
    )  
);
```

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.

Intermediate Operations on Streams :

Use *flatMap* to Transform Each Element to a Stream and Flatten the Result

Example Apply `characterStream` (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 result Stream is a Stream of Streams:

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

Typically, we would like to *flatten* the output, to obtain a single Stream of characters.

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

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

Output:

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

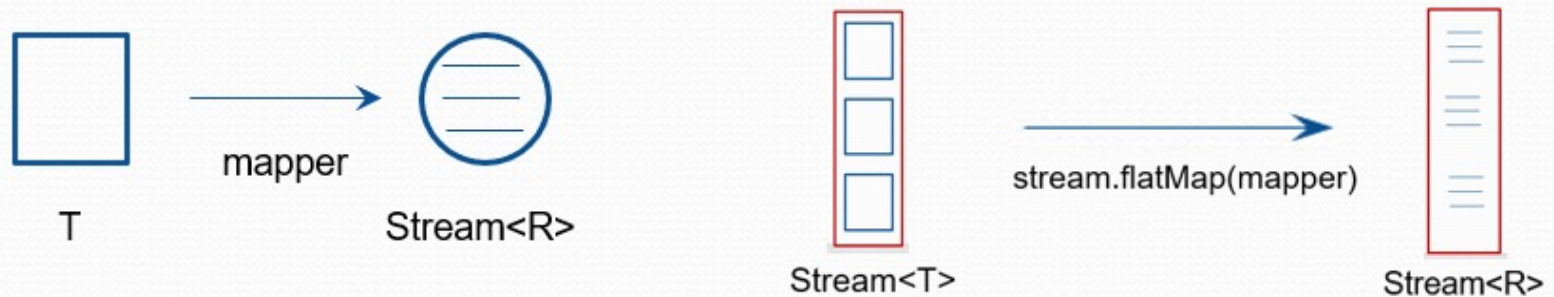
See `lesson9.lecture.flatmapstream.Test.java`.

See also `lesson9.lecture.flatmapstream.Test2.java` for a second example.

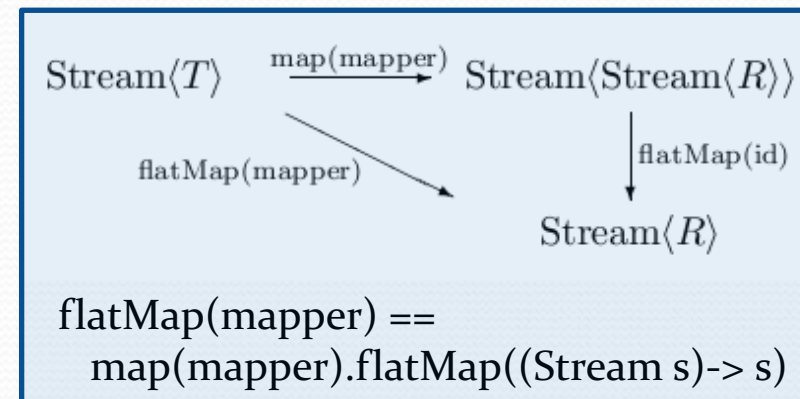
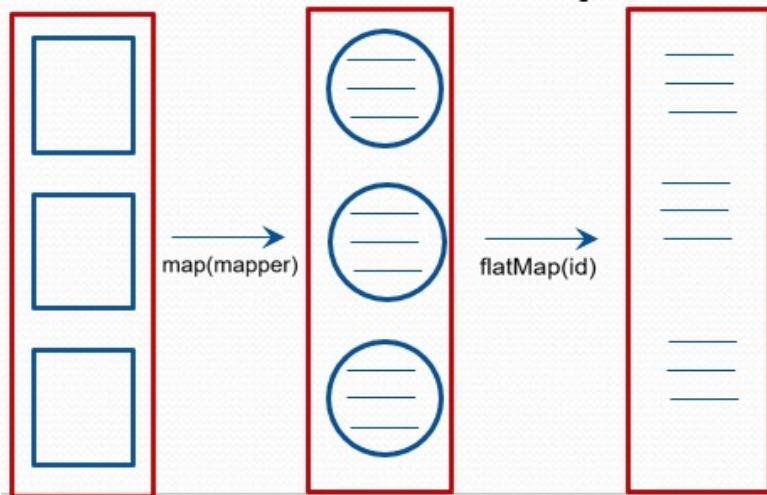
flatMap

Demo: lesson9.lecture.flatmapstream

- Start with `mapper: T → Stream<R>` (like `String → Stream<Character>`)



- `flatMap` transforms a `Stream` of `T`'s first to a `Stream` of `Stream<R>`'s, and then finally to a `Stream` of `R`'s



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

Stateful Intermediate Operations

1. Most of the operations discussed so far (`map`, `filter`, `concat`) have been *stateless*: each element of the stream is processed and forgotten. (Recall the "under the hood" view of `Streams`.)
2. Two *stateful* transformations we have seen so far are `limit` and `skip`. Two others that are 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) -> y.length()-x.length());
System.out.println(longestFirst.collect(Collectors.toList()));
//output: Richard, Joseph, Tom
Demo: lesson9.lecture.comparators1
```

Note: Sorting logic: x comes before y if $y.length() - x.length()$ is negative, which happens when x is longer. So "Richard" comes before "Tom"

Note: `distinct()` and `sorted()` prevent stream fusion because they are stateful, so pipelines that use these are slightly less efficient

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

Demo: `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 and returns another `Comparator`.

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.

Can now 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())
```

Note: `reversed()` is a default method in `Comparator` that reverses the order defined by the instance of `Comparator` that it is being applied to.

Note: The lambda `x -> x.length()` is the same as `String::length`

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. Support for Comparators that are *consistent with equals*.

Demo: `lesson9.lecture.comparators2`

(Recall consistency with equals issue 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 – the following Comparator, passed to `Collections.sort`, 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`. At the same time, we make it consistent with `equals`

```
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:

- `thenComparing` is a default method that also accepts a function of type `Function<T,U>` and it can be chained.
- `thenComparing` modifies current `Comparator` by applying its `compare` method just when the current `compare` method returns 0.

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());
```


Main Point 3

The `comparing` and `thenComparing` methods are examples of the new declarative style that can be used in working with `Streams` in Java 8. This style of programming makes it possible to obtain results simply by *declaring* what is needed, rather than specifying in detail *how* to obtain the results. In a similar way, as awareness becomes more and more familiar with its unbounded transcendental value, we are able to achieve goals with less effort, because transcending engages the deeper levels of nature's functioning, winning support for our individual intentions.

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

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* operations 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 (in that case 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();
```

Another Example: `findAny`. This operation returns an `Optional` that contains some value in an input `Stream`. (How this value is chosen from the `Stream` is not guaranteed.) If the `Stream` is empty, the returned `Optional` is empty. This operation works much better with parallel `Streams` than `findFirst`.

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

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

Working with Optional

– *A Better Way to Handle Nulls*

1. The previous slides introduced the `Optional` class. One reason `Optional` was added to Java was to make handling of `nulls` less error prone. However notice

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

has no real advantage over the following usual code:

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

The `Optional` class, however, supports other techniques that are superior to checking `nulls`.

Working with Optional

orElse

2. The `orElse` method – if result is `null`, provides alternative output using `orElse`

```
//OLD WAY
public static void pickName(final List<String> names, final 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(final List<String> names, final 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. If not present, returns the alternative value supplied as an argument to `orElse`.

The alternative value supplied must have the same type as the value stored in the original `Optional`.

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(div(1,x));  
}  
//returns a/b if possible  
private static Double div(Double a, Double b) {  
    if(b == 0) return null;  
    return a/b;  
}
```

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

Creating Your Own Optionals - Continued

- **Using *ofNullable*.** The static method *ofNullable* lets you read in a possibly null value. In particular, *ofNullable* returns an *Optional* that embeds the specified value if non-null, otherwise returns an empty *Optional*.
- Can use *orElse/orElseGet* together with *Optional.ofNullable*.
- **Example.** Here, *readInput()* returns either null or a person's name.

```
void createOutput() {  
    String outputMessage = "Hello ";  
    outputMessage  
        += Optional.ofNullable(readInput()).orElse("World!");  
    return outputMessage;  
}
```

ofNullable used with orElse/orElseGet

When you wish to obtain a value from a method call using `orElse`, 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 accessed lazily within a `ConnectionManager` class.

See `lesson5.lecture.factorymethods3.dataaccess.DataAccessSystem`

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


orElseGet Syntax

- `orElseGet` expects an instance of `Supplier` as input

```
interface Supplier<T> {  
    T get();  
}
```

Difference Between `orElse` and `orElseGet`

- *Warning.* If you use `orElse` to call a method and use the return value, even if `ofNullable` encounters a non-null input, the method call in `orElse` will still execute, but the return value is ignored. (This is usually undesirable)

In the code on previous slide, if we write instead

```
public Connection getConnection() {  
    return Optional.ofNullable(conn)  
        .orElse(ConnectManager::createConnection);  
}
```

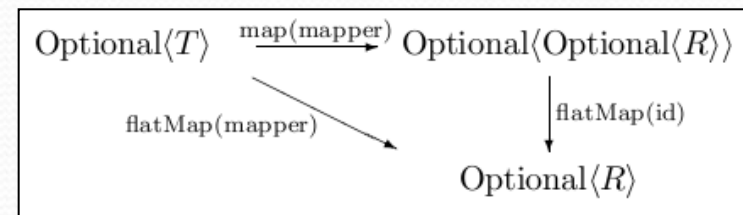
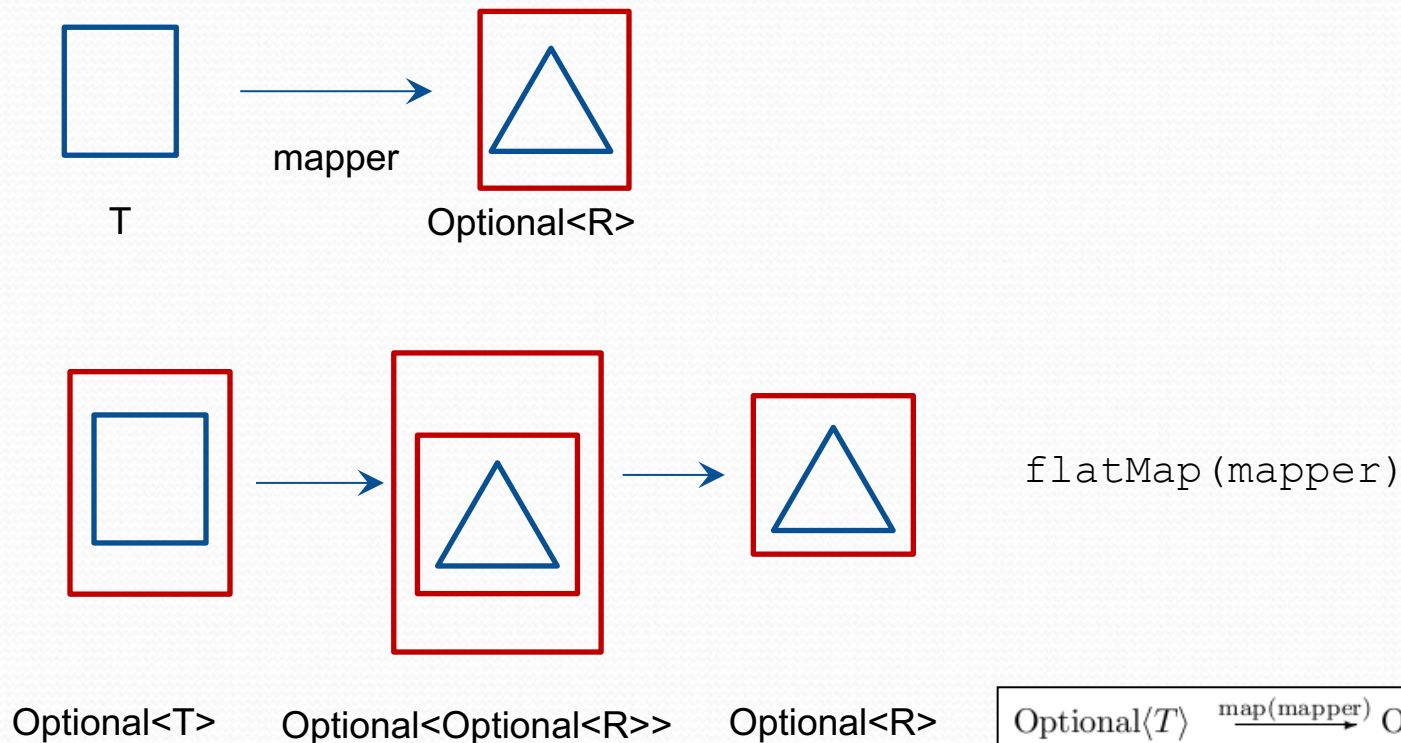
then, when `conn` is not null, `orElse` will cause `createConnection()` to be invoked (not good), but will ignore its return value and return `conn` instead.

In the same scenario, the following code will *not* cause `createConnection()` to be invoked when `conn` is not null.

```
public Connection getConnection() {  
    return Optional.ofNullable(conn)  
        .orElseGet (ConnectManager::createConnection);  
}
```

Using Optional : flatMap

The `flatMap` method works the same way as it does on `Stream`. It allows you to chain method calls and skip null checks. Given a mapper from type `T` to an `Optional<R>`, `flatMap` transforms an `Optional<T>` first to an `Optional<Optional<R>>`, then to an `Optional<R>`





```
String courses[][][] =  
    {  
        {  
            {"C", "Fundamentals of Computing"}, //1st semester  
            {"C++", "FPP"}, //2nd semester  
        },  
        {  
            {"MPP", "WAP"}, //3rd semester  
            {"Android", "iOS"} //4th semester  
        }  
    };
```

```
System.out.println(  
    Arrays.stream(courses) Stream<String[][]>  
        .flatMap(year -> Arrays.stream(year)) Stream<String[]>  
        .flatMap(semester -> Arrays.stream(semester)) Stream<String>  
        .map(course -> course.toUpperCase())  
        .collect(Collectors.toList())  
);
```

```
System.out.println(  
    Arrays.stream(courses) Stream<String[][]>  
        .flatMap(year -> Arrays.stream(year)) Stream<String[]>  
        .flatMap(semester -> Arrays.stream(semester)) Stream<String>  
        .map(course -> course.toUpperCase())  
        .collect(Collectors.toList())  
);
```

//step-by-step process

```
Stream<String [][]> allCourses = Arrays.stream(courses);  
Stream<String[]> perYear = allCourses.flatMap(year -> Arrays.stream(year));  
Stream<String> perSemester = perYear.flatMap(course -> Arrays.stream(course));  
System.out.println(  
    perSemester.map(item -> item.toUpperCase())  
        .collect(Collectors.toList())  
);
```



```
System.out.println(  
    Arrays.stream(courses)  
        .flatMap(  
            year -> Arrays.stream(year)  
                .flatMap(  
                    semester -> Arrays.stream(semester)  
                        .map(course -> course.toUpperCase())  
                )  
        )  
    ).collect(Collectors.toList())  
);
```



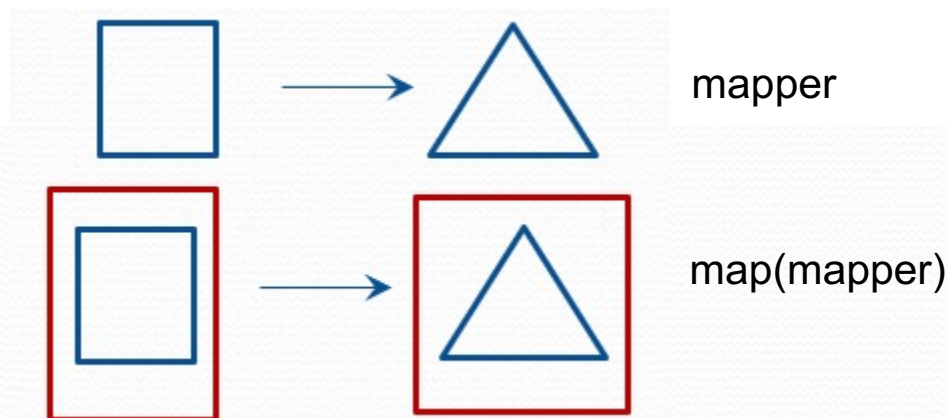
Example: Replacing null checks with Optional

Consider imperative 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;  
}
```


- Using Optionals with map completely eliminates these (unnecessary) null checks

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



- Notes.**

1. If an `Optional opt` is empty, `opt.map(mapper)` returns an empty `Optional`
2. See `lesson9.lecture.optional_map.usingoptionals`

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

Terminal Operations continued: 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 **`digits`** of numbers.
Procedural code:

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

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

```
Integer sum2 = digits.stream().reduce(0, (x, y) -> x + y);
```

The reduce Operation (cont.)

```
digits.stream().reduce(0, (x, y) -> x + y)
```

- *First Argument: Initial Value.* First argument is the initial value; it is the value that is returned if the stream is empty and is the starting point for the computation. It is also (or *should be*) the *identity element* for the combining operation.

- *Second Argument:* A lambda of type `BinaryOperator<T>`.

```
interface BinaryOperator<T> {  
    T apply(T a, T b);  
}
```

- *The Computation.* The computation starts by applying the binary operator to the pair (e, f), where e is the initial value (0 in this example) and f is the first element of the `Stream`, producing `accum`. Then the binary operator is applied to (accum, g) where g is the next element in the `Stream`, and the result is stored in `accum` again. This continues until all `Stream` values have been used.

- When the `stream` consists of numbers and the binary operator is ordinary addition, the output is the sum of all the numbers in the stream. If the stream is empty, the initial value is returned – note that the sum of an empty collection of numbers is by convention 0.
- **Example**. Start with the following `stream`: `[2, 1, 4, 3]`, and use ordinary addition for the `BinaryOperator` functor. 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:

$$\begin{aligned} \text{sum1} &= (0 + 2) + 3 & \text{sum2} &= (0 + 4) + 1 \\ \text{combined} &= \text{sum1} + \text{sum2} = 10 \end{aligned}$$

The reduce Operation (continued)

- **Question:** How could we form the *product* of a list of numbers?
- **Answer:** We form the product of a list `numbers` of Integers. For the initial value, we ask, “What is the identity element for the multiplication operation?” The identity is 1. (**Note** that, by convention, the product of an empty list of numbers is 1.)

Here is the line of code that does the job:

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


The reduce Operation (continued)

- 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, not only should the initial value be an identity element (it isn't an identity for subtraction) but also:

Use reduce only on operations that are commutative and associative.

(Note that + and * are commutative & associative, but subtraction is not.)

See the demo `lesson9.lecture.reduce`.

The reduce Operation (continued)

- The reduce method has an overloaded version with only one argument.
- Continuing with the sum example, here is a computation with the overridden version:

```
Optional<Integer> sum  
    = Stream.of(digits).reduce((x, y) -> x + y);
```

- 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 strings = Stream.of("A", "good", "day", "to", "write", "some", "Java");  
}
```

Main Point 4

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 inherent in 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.

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

Collecting Results

- One kind of terminal operation in a stream pipeline is a *reduction* that outputs a single value, like `max` or `count` or `reduce`. 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 = stream.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));
```


Collecting Results (cont.)

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`

Collecting Results (cont.)

Example Collecting into a map – two typical examples.

Here, `personStrm` is a `Stream` of `Person` objects.

```
//key = id, value = name
```

```
Map<Integer, String> idToName =  
    personStrm.collect(Collectors.toMap(Person::getId,  
                                         Person::getName));
```

```
//key = id, value = the person object
```

```
Map<Integer, Person> idToPerson =  
    personStrm.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
```


Collecting Results (cont.)

- Can collect “summary statistics” for `Streams` whose elements can be mapped to `ints`. `IntSummaryStatistics` provides `sum`, `average`, `maximum`, and `minimum`

```
IntSummaryStatistics summary =  
    words.collect(Collectors.summarizingInt(String::length));  
double averageWordLength = summary.getAverage();  
double maxWordLength = summary.getMax();
```

//Recall: `String::length` means `str -> str.length()`

- 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` will therefore be converted at runtime to (primitive) `ints` in order for `summarizingInt` to perform its tasks.
- The `summarizingInt` method expects an argument that is an implementation of the `ToIntFunction<T>` interface:

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


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));  
        }  
    };  
}
```

Main Point 5

To obtain some information about a collection, we can lift the collection to a Stream, perform transformations at the Stream level, and then, by way of some terminal operation, collapse the Stream to obtain a final output. It is by virtue of accessing the powerful dynamics available in Stream that the result is obtained.

In a similar way, experience of transcending makes it possible to fulfill desires and intentions with a minimum of effort: When the mind transcends, awareness become saturated with the level from which all the laws of nature begin to operate – the unified field. Having the support of this powerful level, desires can effortlessly meet with fulfillment.

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

Primitive Type Streams

There are variations of `Stream` specifically designed for primitives: `int`, `double`, and `long`:

- `IntStream`
- `DoubleStream`
- `LongStream`

For **primitive types** `short`, `char`, `byte`, and `boolean`, **use** `IntStream`; for floats, **use** `DoubleStream`.

1. Creation methods are similar to those for `Stream`:

- `IntStream ints = IntStream.of(1, 2, 4, 8);`
- `IntStream ones = IntStream.generate(() -> 1);`
- `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 stream of objects, 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 example, 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:
- 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`.)

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**

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"))
```

Typical Uses:

1. count the number of names obtained
2. output the names in upper case to a List

But once the `Stream` has been used once, we can't use it again.

Stream Re-use Techniques

- 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 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`
- The second solution leads to a useful way of storing stream pipelines for reuse similar to techniques from database management

Creating a Lambda Library

Java 8 lets you perform *queries* to work with data in a `Collection` of some kind. The query style is similar to SQL queries. And queries can be stored in a way that is similar to SQL queries.

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`

We can store the query by naming it and replacing parameters with wildcards:

```
SEARCH_CITY_IN_CUST = "SELECT name FROM Customer WHERE city LIKE '?'  
ORDER BY name"
```

Then use in various applications:

```
Connection conn = ConnectManager.getConnection();  
PreparedStatement stat = conn.prepareStatement(SEARCH_CITY_IN_CUST );  
stat.setString(1, 'Ma%');
```


Similar 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

How to turn your solution into a reusable Lambda Library element:

Identify the parameters and treat them as arguments for some kind of Java function-type interface (Function, BiFunction, TriFunction, etc).

Parameters in this problem:

- 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>`

Turning Your Stream Pipeline into a Library Element (cont.)

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 are coherently arranged in the first sprouting of existence.
 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.

