

CS 213 – Software Methodology

Spring 2016

Lecture 27: Apr 26

Streams (Java 8)

Example: Movie Stats

```
public class Movie {  
  
    public static enum Genre {  
        ACTION, ADVENTURE, DRAMA, MYSTERY, ROMANCE, SCIFI, THRILLER  
    }  
  
    private String name;  
    private int year;  
    private int rating;  
    private Genre category;  
  
    public Movie(String name, int year, int rating, Genre genre) {  
        this.name=name; this.year=year; this.rating=rating; category=genre;  
    }  
  
    public String getName() { return name; }  
  
    public int getYear() { return year; }  
  
    public int getRating() { return rating; }  
  
    public Genre getCategory() { return category; }  
  
}
```

Example: Movie Stats

```
public static List<Movie> movies = Arrays.asList(  
    new Movie("Max Max: Fury Road",2015,  
        5,Genre.ACTION),  
    new Movie("Straight Outta Compton", 2015,  
        5,Genre.DRAMA),  
    new Movie("Fifty Shades of Grey", 2015,  
        1,Genre.DRAMA),  
    new Movie("American Sniper, 2014,  
        4,Genre.ACTION),  
    new Movie("Transcendence", 2014,  
        1,Genre.THRILLER),  
    new Movie("Conan The Barbarian", 2011,  
        2,Genre.ADVENTURE),  
    new Movie("The Last Airbender", 2010,  
        2,Genre.ADVENTURE),  
    new Movie("Harry Potter and the Deathly Hallows: Part 1", 2010,  
        4,Genre.ADVENTURE),  
    new Movie("Sicario", 2015,  
        4,Genre.MYSTERY),  
    new Movie("The Gift", 2000,  
        3,Genre.MYSTERY)  
);
```

Movies: Ratings < 3

Want to get list of movies with ratings < 3

Iterator Version:

Implement a filter+mapper that will filter movies for some predicate, and map to movie name

```
public static <T,R>
List<R> filterMap(List<T> list, Predicate<T> p, Function<T,R> f) {
    List<R> result = new ArrayList<R>();
    for (T t: list) {
        if (p.test(t)) {
            result.add(f.apply(t));
        }
    }
    return result;
}
```

Call the filter+mapper:

```
System.out.println(
    filterMap(movies, m -> m.getRating() < 3, Movie::getName)
);
```

Movies: Ratings < 3

Want to get list of movies with ratings < 3

Stream Version:

Source the movies list to a stream and apply a sequence of stream operations:

```
List<String> badMovies =  
    movies.stream()  
        .filter(m -> m.getRating() < 3)  
        .map(Movie::getName)  
        .collect(toList());  
System.out.println(badMovies);
```

[Fifty Shades of Grey, Transcendence, Conan The Barbarian, The Last Airbender]

Benefits of Streams

Declarative:

You specify what you want to get done, don't worry about how

Composable:

You can put together a chain of operations to express a complex processing pipeline while keeping the code and intention clear

Parallelizable:

Streams can be run in parallel with a trivial change:

```
List<String> badMovies =  
    movies.parallelStream()  
        .filter(...)  
        ...
```

The mechanics of scheduling to multiple cores is handled by VM/OS

Stream Operation Types

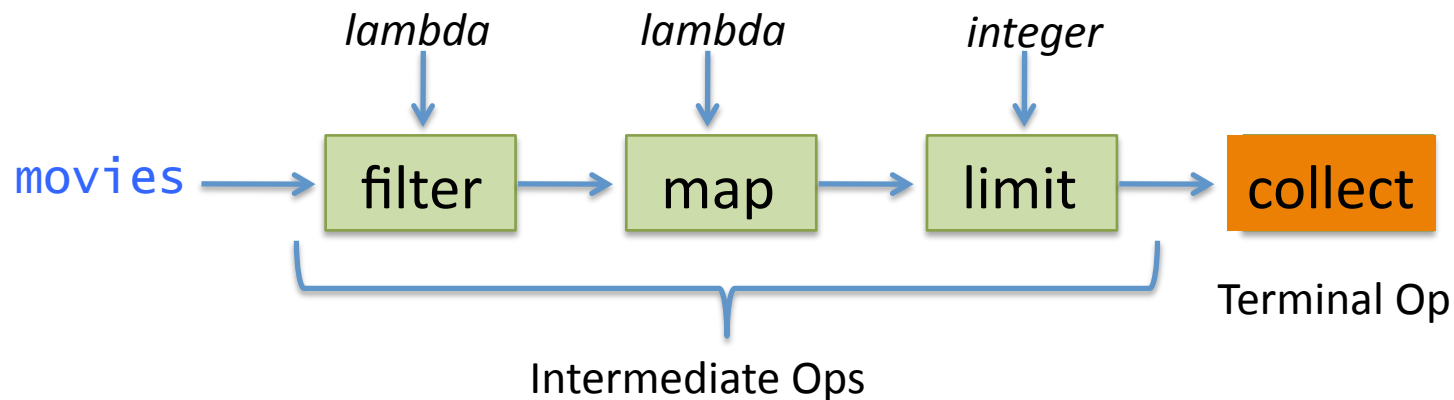
Streams operations are either **intermediate** or **terminal**

An intermediate operation results in a stream

A terminal operation produces a non-stream result

```
List<String> badMovies =  
    movies.stream()  
        .filter(m -> m.getRating() < 3)  
        .map(Movie::getName)  
        .limit(2)  
        .collect(toList());
```

[Fifty Shades of Grey, Transcendence]



Breaking it Down

```
Stream<Movie> movieStream = movies.stream();  
movieStream = movieStream.filter(m -> m.getRating() < 3);  
Stream<String> movieNameStream = movieStream.map(Movie::getName);  
List<String> movieNameList = movieNameStream.collect(toList());
```


Short-Circuiting of Operations

```
List<String> names =  
movies.stream()  
    .filter(m -> {  
        System.out.println("filtering " + m.getName());  
        return m.getRating() < 3;  
    })  
    .map(m -> {  
        System.out.println("mapping " + m.getName());  
        return m.getName();  
    })  
    .limit(2)  
    .collect(toList());  
System.out.println(names);
```

NOT every item in the list is processed.
As soon as the limit is reached,
processing stops (short-circuiting).
Also, filtering and mapping do not
happen in strict sequence—they are
interleaved

```
filtering Mad Max: Fury Road  
filtering Straight Outta Compton  
filtering Fifty Shades of Grey  
mapping Fifty Shades of Grey  
filtering American Sniper  
filtering Transcendence  
mapping Transcendence  
[Fifty Shades of Grey, Transcendence]
```

Terminal Operations

Terminal operations can return a primitive, a structure, or void

```
// count operation returns a long int
int adventureMoviesCount = (int)
movies.stream()
    .filter(m -> m.getCategory() == Genre.ADVENTURE)
    .count();
```

3

```
// forEach operation consumes the stream
movies.stream()
    .filter(m -> m.getCategory() == Genre.ACTION)
    .sorted(comparing(Movie::getName).reversed())
    .map(Movie::getName)
    .forEach(System.out::println);
```

Max Max: Fury Road
American Sniper

Static method
`java.util.Comparator.comparing`

Returns a `Comparator` that
Reverses the comparison order
of `Comparator` on which it is
applied

Sources for Streams

1. Values

Static method
`java.util.stream.Stream.of`

```
Stream<String> gimme =  
    Stream.of("Spotlight", "Mad Max", "Martian",  
             "Revenant", "Big Short", "The Danish Girl");  
  
gimme.map(String::toUpperCase)  
    .forEach(System.out::println);
```

```
SPOTLIGHT  
MAD MAX  
MARTIAN  
REVENANT  
BIG SHORT  
THE DANISH GIRL
```

2. Array

```
int[] primes = {2,3,5,7,11,13,19,23,29};  
  
IntStream primeStream = Arrays.stream(primes);  
  
System.out.println(primeStream.sum());
```


112

Static method
`java.util.Arrays.stream`

Sources for Streams

3. Numerical range

Static method
`java.util.stream.IntStream.rangeClosed`

`IntStream` 
`.rangeClosed(1,10)`
`.map(i -> i*i)`
`.forEach(System.out::println);`

Static method `java.util.stream.IntStream.range(1,10)`
gives a right-open range 1..9

1
4
9
16
25
36
49
64
81
100

Typed Streams

There are three typed streams: `IntStream`, `DoubleStream`, and `LongStream`, with slightly different sets of methods.

`DoubleStream`, for instance, does not have a range method

Sources for Streams

4. File

Static method
`java.nio.file.Files.lines`



```
try {  
    Stream<String> lines = Files.lines(Paths.get("file.txt"));  
    lines  
        .map(line -> line.split(" ").length)  
        .forEach(System.out::println);  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

?

number of words in
each line of file.txt

Class `java.nio.file.Files` consists exclusively of static methods that operate on files and directories

Class `java.nio.file.Paths` consists exclusively of (two) static methods that create file or URI path objects out of strings

Sources for Streams

5. Functions

a. iterate

Static method
`java.util.stream.Stream.iterate`

↓

Stream
`.iterate(1, n -> n+3)` infinite sequence 1,4,7,10,...
`.limit(10)`
`.forEach(System.out::println);`

`iterate` takes a seed parameter of type `T`, and a `UnaryOperator<T>` (which is a special kind of the `Function` interface that has same result type as input, i.e. `Function<T, T>`, and inherits the `apply` method from `Function`)

The function is applied on each successive value, so the sequence:
`seed, f(seed), f(f(seed)) ...`

Stream
`.iterate("*", s -> s + "*")`
`.limit(6)`
`.forEach(System.out::println);`

```
*  
**  
***  
****  
*****  
*****
```

Sources for Streams

5. Functions

b. generate

Static method

`java.util.stream.Stream.generate`

`Stream`



```
.generate(Math::random)    infinite sequence of random numbers  
.limit(5)  
.forEach(System.out::println);
```

`generate` takes a `Supplier<T>` as parameter and generates an infinite sequence of type `T` elements

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

Useful Stream Operations

Some additional ops aside from the ones we have already seen

Identifying distinct occurrences

```
String[][] cars =  
{  
    {"Honda","Civic","2016"},  
    {"Toyota","Camry","2016"},  
    {"Ford","Fusion","2016"},  
    {"Subaru","Forrester","2016"},  
    {"Honda","Accord","2016"},  
    {"Ford","Focus","2016"},  
    {"Honda","Pilot","2016"}  
};
```

Arrays

```
.stream(cars)  
.map(mm -> mm[0])  
.distinct()  
.forEach(System.out::println);
```

mapping array to
array element

?

Honda
Toyota
Ford
Subaru

distinct car makes

Useful Stream Operations

Finding and Matching

1. Find any – version 1

E.g. find any 1-star rated movie in `movies` list

```
movies
    .stream()
    .filter(m -> m.getRating() == 1)
    .map(Movie::getName)
    .findAny()
    .ifPresent(System.out::println);
```

Fifty Shades of Grey

`findAny` returns a `java.util.Optional<T>` object

`Optional` is a container that may or may not contain a null value

The `ifPresent` method in `Optional` accepts a `Consumer` that is applied to the contained value, if any. If not, the method does nothing

Useful Stream Operations

Finding and Matching

1. Find any – version 2

E.g. find any 2014 movie in `movies` list that was 5-star rated

```
System.out.println(  
    movies  
        .stream()  
        .filter(m -> m.getYear() == 2014 && m.getRating() == 5)  
        .map(Movie::getName)  
        .findAny()  
        .orElse("No match"));
```

No match

The `orElse` method in `Optional` returns the contained value, if any. If not, it returns the supplied value

Short Circuiting

```
movies
  .stream()
  .filter(m -> {
    System.out.println("filtering" + m.getName());
    return m.getRating() == 1;
  })
  .map(m -> {
    System.out.println("mapping " + m.getName());
    return m.getName();
  })
  .findAny()
  .ifPresent(System.out::println);
```

```
filtering Max Max: Fury Road
filtering Straight Outta Compton
filtering Fifty Shades of Grey
mapping Fifty Shades of Grey
Fifty Shades of Grey
```

Useful Stream Operations

Finding and Matching

2. Find first

E.g. find the first movie in `movies` list that got a 4-star rating

```
System.out.println(  
    movies  
        .stream()  
        .filter(m -> m.getRating() == 4)  
        .map(Movie::getName)  
        .findFirst()  
        .orElse("No match"));
```

American Sniper

Useful Stream Operations

Finding and Matching

3. Predicate Matching

a. Is there any item that matches a predicate?

```
System.out.println(  
    movies  
        .stream()  
        .anyMatch(m -> m.getCategory() == Genre.MYSTERY && m.getRating() > 3));
```

true

b. Do all items match a predicate?

```
System.out.println(  
    Arrays  
        .stream(cars)  
        .map(mmy -> mmy[2])  
        .allMatch(y -> y.equals("2016")));
```

? true

```
String[][] cars =  
{  
    {"Honda", "Civic", "2016"},  
    {"Toyota", "Camry", "2016"},  
    {"Ford", "Fusion", "2016"},  
    {"Subaru", "Forrester", "2016"},  
    {"Honda", "Accord", "2016"},  
    {"Ford", "Focus", "2016"},  
    {"Honda", "Pilot", "2016"}  
};
```

c. There's also a `noneMatch` method

Useful Stream Operations

Reduce

Sum

E.g. find the number of words in an input file

```
try {  
    Stream<String> lines = Files.lines(Paths.get("file.txt"));  
    lines  
        .map(line -> line.split(" ").length)  
        .reduce(Integer::sum)  
        .ifPresent(System.out::println);  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

The reduce method takes as parameter a `BinaryOperator<T>` instance, which serves as an associative accumulator. In this example, the associative accumulator is the sum method in the `Integer` class.

Useful Stream Operations

Reduce

`Optional<T> reduce (BinaryOperator<T> accumulator)`

Here's the equivalent code for `reduce`, according to the Java doc:

```
boolean foundAny = false;
T result = null;
for (T element: this stream) {
    if (!foundAny) {
        foundAny = true;
        result = element;
    } else {
        result = accumulator.apply(result, element);
    }
}
return foundAny ? Optional.of(result) : Optional.empty();
```

The accumulator function must be an associative function because the accumulation process is not guaranteed to work through the stream items sequentially

Useful Stream Operations

Reduce

Product – Using an identity element as seed

E.g. find the factorial of n

```
IntStream is = IntStream.rangeClosed(n);  
int fact = is.reduce(1, (x,y) -> x*y);
```



identity

Sum method, numeric stream

```
try {  
    Stream<String> lines = Files.lines(Paths.get("file.txt"));  
    System.out.println(  
        lines  
        Returns an IntStream → .mapToInt(line -> line.split(" ").length)  
        .sum()  
    );  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

Can also do max and
min reductions on
IntStream

Useful Stream Operations

Reduce

E.g. find the average star rating of all movies in `movies` list

```
Optional<Integer> opt =  
movies.stream()  
    .map(Movie::getRating)  
    .reduce(Integer::sum);  
try {  
    System.out.println(opt.get()*1f/movies.stream().count());  
} catch (NoSuchElementException e) {  
    System.out.println("No movies in list");  
}
```

The `Optional` class's `get` method returns the contained value, or throws a `NoSuchElementException` if none exists

Useful Stream Operations

flatMap

E.g. Find the average word length in an input file

The rabbit-hole went straight on like a tunnel for some way, and then dipped suddenly down, so suddenly that Alice had not a moment to think about stopping herself before she found herself falling down a very deep well. Either the well was very deep, or she fell very slowly, for she had plenty of time as she went down to look about her and to wonder what was going to happen next. First, she tried to look down and make out what she was coming to, but it was too dark to see anything; then she looked at the sides of the well, and noticed that they were filled with cupboards and book-shelves; here and there she saw maps and pictures hung upon pegs.

Useful Stream Operations

flatMap

We need to extract words from each line, then get their lengths

```
try {  
    Stream<String> lines = Files.lines(Paths.get("alice.txt"));  
    lines  
        .map(line -> line.split(" "))  
        .forEach(System.out::println);  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

What does this print?

Each line of output is an
array of words in the lines
of the input file

The map function in the code converts
Stream<String> to Stream<String[]>

```
[Ljava.lang.String;@7cc355be  
[Ljava.lang.String;@6e8cf4c6  
[Ljava.lang.String;@12edcd21  
[Ljava.lang.String;@34c45dca  
[Ljava.lang.String;@52cc8049  
[Ljava.lang.String;@5b6f7412  
[Ljava.lang.String;@27973e9b  
[Ljava.lang.String;@312b1dae  
[Ljava.lang.String;@7530d0a  
[Ljava.lang.String;@27bc2616  
[Ljava.lang.String;@3941a79c
```

Useful Stream Operations

flatMap

But we need a `Stream<String>` of individual words, so we may get their lengths, then average

What we want to do is to “flatten” the `Stream<String[]>` to `Stream<String>`

```
try {  
    Stream<String> lines = Files.lines(Paths.get("alice.txt"));  
    lines  
        .map(line -> line.split(" "))  
        .flatMap(Arrays::stream)  
        .forEach(System.out::println);  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

The arrays produced in the first map is flattened out into their constituent words by the second

```
The  
rabbit-hole  
went  
straight  
on  
like  
A  
tunnel  
...
```

Useful Stream Operations

flatMap

So now we can map the words to their lengths, and get the average

```
try {  
    Stream<String> lines = Files.lines(Paths.get("alice.txt"));  
    Optional<Double> avg =  
        lines  
            .map(line -> line.split(" "))  
            .flatMap(Arrays::stream)  
            .mapToInt(String::length)  
            .average();  
    avg.ifPresent(System.out::println);  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

4.224

Useful Stream Operations

flatMap

Given two arrays, print all cross-pairs of items

```
List<Integer> L1 = Arrays.asList(2,3,7,9);  
List<Integer> L2 = Arrays.asList(4,5,8);  
L1.stream()  
    .flatMap(i ->  
        L2.stream()  
            .map(j -> new int[]{i,j}))  
    .forEach(a -> System.out.println(Arrays.toString(a)));
```

```
[2, 4]  
[2, 5]  
[2, 8]  
[3, 4]  
[3, 5]  
[3, 8]  
[7, 4]  
[7, 5]  
[7, 8]  
[9, 4]  
[9, 5]  
[9, 8]
```

Useful Stream Operations

flatMap

Given two arrays, get all cross-pairs of items –
try with arrays instead of lists

```
int[] arr1 = {2,3,7,9};  
int[] arr2 = {4,5,8};  
Arrays.stream(arr1)  
    .flatMap(i ->  
        Arrays.stream(arr2)  
            .map(j -> new int[]{i,j})));
```

Useful Stream Operations

flatMap

Using `IntStream` instead

```
Stream<int[]> pairs =  
    IntStream  
        .rangeClosed(1,5)  
        .boxed()  
        .flatMap(i ->  
            IntStream  
                .rangeClosed(i+1, 5)  
                .mapToObj(j -> new int[]{i,j}));  
  
pairs  
    .forEach(p -> System.out.println(Arrays.toString(p)));
```


Breaking it Down

flatMap

```
IntStream is = IntStream.rangeClosed(1,5);  
is.flatMap(i -> is.map(j -> new int[]{i,j}));
```

Doesn't work: cannot convert from int[] to int
(IntStream.flatMap must return an IntStream)

```
Stream<Integer> si = is.boxed();
```

```
Stream<int[]> pairs =  
si.flatMap(i -> si.map(j -> new int[]{i,j}));
```

(si is a Stream<Integer>, so no mapToObj – in fact
mapToObj does not exist in Stream interface)

```
pairs  
    .forEach(p -> System.out.println(p[0] + “,” + p[1]));
```

Useful Stream Operations

flatMap

Using `IntStream` instead

```
Stream<int[]> pairs =  
    IntStream  
        .rangeClosed(1,5)  
        .boxed()  
        .flatMap(i ->  
            IntStream  
                .rangeClosed(i+1, 5)  
                .mapToObj(j -> new int[]{i,j}));  
                .boxed()  
                .map(j -> new int[]{i,j}));
```

Converting a Stream to an Array

The `Stream` method `toArray()` converts a stream to an array:

```
String[] badMovies =  
    movies.stream()  
        .filter(m -> m.getRating() < 3)  
        .map(Movie::getName)  
        .toArray(String[]::new);
```

Without the generator parameter, `toArray` will produce an array of `Object` instances, which cannot be cast to an array of another type:

```
String[] badMovies = (String[]) ← This cast does  
    movies.stream()              not work  
    ...  
    .toArray();
```

Numeric Stream to an Array

The `IntStream` method `toArray()` does not accept a parameter, and returns an `int[]`

```
int[] squares =  
    Arrays.stream(new int[]{1,2,3,4,5})  
        .map(i -> i*i)  
        .toArray();
```

The `DoubleStream` and `LongStream()` numeric streams work similarly, with `toArray()` returning `double[]` and `long[]`, respectively.

Useful Stream Operations

Operation	Return Type	Type Used
filter	Stream<T>	Predicate<T>
distinct	Stream<T>	
limit	Stream<T>	long
map	Stream<R>	Function<T,R>
flatMap	Stream<R>	Function<T, Stream<R>>
sorted	Stream<T>	Comparator<T>
anyMatch/noneMatch/ allMatch	boolean	Predicate<T>
findAny/findFirst	Optional<T>	
forEach	void	Consumer<T>
collect	R	Collector<T,A,R>
reduce	Optional<T>	BinaryOperator<T>
count	long	