

# CS 213 – Software Methodology

*Sesh Venugopal*

Streams – Part 1

# 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("Mad 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 list names of movies with rating < 3

## Iterator Version:

Implement a filter + mapper that will filter `Movie` instances on some predicate, and map these instances to the associated movie names

```
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 list names of movies with rating < 3

## Stream Version:

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

```
List<String> badMovies =  
    movies.stream() ← Returns an instance of interface type  
                java.util.stream.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

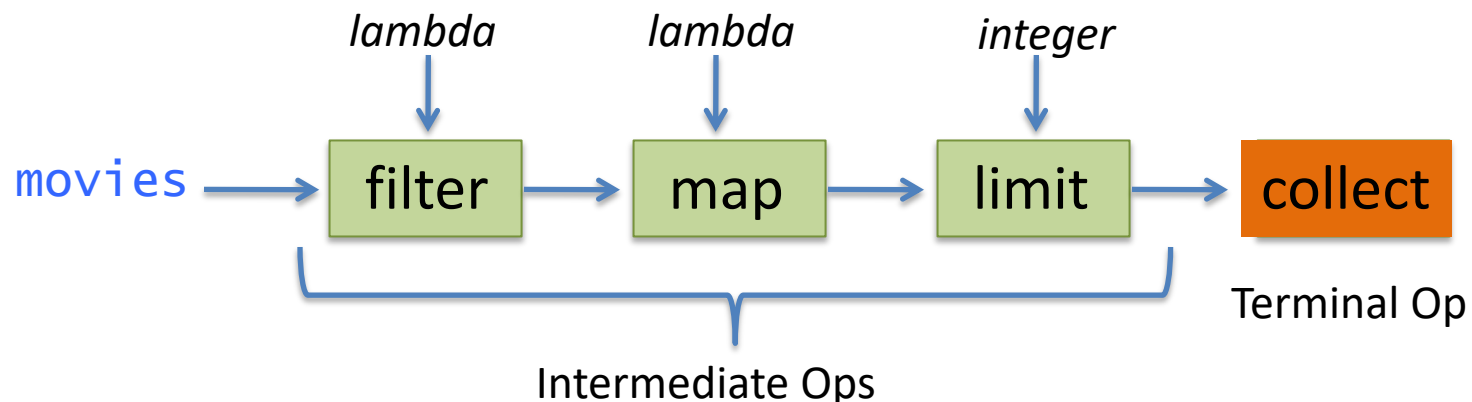
Streams operations are either **intermediate** or **terminal**

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, With Full Typing of all Intermediate Structures

```
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, an object, or void

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


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);
```

Static method  
`java.util.Comparator.comparing`



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




Mad Max: Fury Road  
American Sniper

# Data Sources for Streams (Aside from `Collection.stream`)

# 1. Stream of 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

interface  
`java.util.stream.IntStream`  
for streams that hold primitive int values

Static method  
`java.util.Arrays.stream`  
“Reduction” method in  
`java.util.stream.IntStream`

(`DoubleStream` for `double[]` and `LongStream` for `long[]`,  
but `Stream<T>` for `T[]`)

### 3. Numerical range

```
1  
4  
9  
16  
25  
36  
49  
64  
81  
100
```

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

`IntStream`  
`.rangeClosed(1,10)` ← Returned type is `IntStream`, not `Stream<Integer>`  
`.map(i -> i*i)`  
`.forEach(System.out::println);`

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

### Typed Streams

There are three typed streams: `IntStream`, `DoubleStream`, and `LongStream`, with slightly different sets of methods. (All of these hold values of the corresponding primitive type.)

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

## 4. File

Class (not interface)      Class (not interface)  
`java.nio.file.Files`      `java.nio.file.Paths`

static      static

```
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

## 5. Functions - iterate

### a. iterate

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

Stream ↓

```
.iterate(1, n -> n+3)    infinite sequence 1,4,7,10,... (Stream<Integer>)  
.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, resulting in the sequence:  
`seed, f(seed), f(f(seed)) ...`



## 5. Functions - iterate

### a. iterate

Q. How could you use iterate to print this pattern:

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

A:

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

## 5. Functions - generate

### b. generate

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

`Stream` ↓

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

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

The typed streams `IntStream`, `DoubleStream`, and `LongStream`, also have generate methods, that return an instance of that typed stream:

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

↑  
Lambda for functional interface  
`java.util.function.IntSupplier`

# Additional Useful Stream Operations

## Identifying distinct occurrences - distinct

```
String[][] cars =  
{  
    {"Honda", "Civic", "2019"},  
    {"Toyota", "Camry", "2019"},  
    {"Ford", "Fusion", "2019"},  
    {"Subaru", "Forrester", "2019"},  
    {"Honda", "Accord", "2019"},  
    {"Ford", "Focus", "2019"},  
    {"Honda", "Pilot", "2019"}  
};
```

Arrays

`.stream(cars)` ← gives `Stream<String[]>`

`.map(mm -> mm[0])` ← mapping array to its first element

`.distinct()`  
`.forEach(System.out::println);`

?

Honda  
Toyota  
Ford  
Subaru

distinct  
car makes

## Finding and Matching - `findAny`

### 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

## Finding and Matching - `findAny`

### 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
```

Stream processing is cut short  
as soon as there is an instance  
in the stream before `findAny`

## Finding and Matching - `findFirst`

### 2. Find first – returns `Optional`

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



## Finding and Matching – anyMatch/allMatch/noneMatch (boolean)

### 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("2019")));
```

? true

```
String[][] cars =  
{  
    {"Honda", "Civic", "2019"},  
    {"Toyota", "Camry", "2019"},  
    {"Ford", "Fusion", "2019"},  
    {"Subaru", "Forrester", "2019"},  
    {"Honda", "Accord", "2019"},  
    {"Ford", "Focus", "2019"},  
    {"Honda", "Pilot", "2019"}  
};
```

c. There's also a `noneMatch` method