CS 213 – Software Methodology

Spring 2019

Lecture 24: Apr 25

Streams (Java 8)

Example: Movie Stats

```java
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

```java
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

```java
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:

```java
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
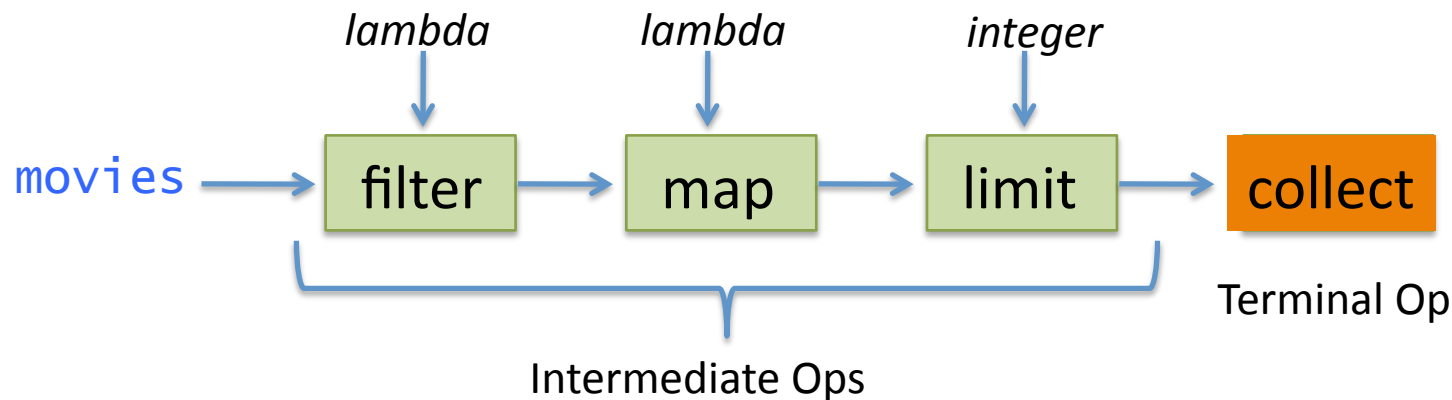
Stream Operation Types

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]

lambda　　　*lambda*　　　*integer*

movies → **filter** → **map** → **limit** → **collect**

Intermediate Ops

Terminal Op

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

```java
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);
```

```
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]
```

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.

Terminal Operations

Terminal operations can return a primitive, an object, or void

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

3

```java
// 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

Data Sources for Streams
(Aside from Collection.stream)

1. Stream of Values

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

```java
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

```java
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

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

```
1
4      IntStream
9      .rangeClosed(1,10)          Returned type is IntStream, not Stream<Integer>
16     .map(i -> i*i)
25     .forEach(System.out::println);
36
49
64     Static method java.util.stream.IntStream.range(1,10)
81     gives a right-open range 1..9
100
```

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