CS 213 – Software Methodology

Spring 2017

Lecture 26: Apr 27

Streams – Part 2

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)
    );
```

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

```
[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
```

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

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

Example without `flatMap`

```
List<Integer> l1 = Arrays.asList(2,3,7,9);
Stream<int[]> strm =
      l1.stream()
        .map(i -> new int[]{1,i});
strm.forEach(a -> System.out.println(Arrays.toString(a)));
```

```
[1,2]
[1,3]
[1,7]
[1,9]
```

Useful Stream Operations

Example without `flatMap`

```
List<Integer> l1 = Arrays.asList(2,3,7,9);
List<Integer> l2 = Arrays.asList(4,5,8);

Stream<Stream<int[]>> strm2 =
      l1.stream()
        .map(i -> l2.stream()
                    .map(j -> {new int[]{i,j}));

strm2.forEach(System.out::println);
```

```
java.util.stream.ReferencePipeline$3@53d8d10a
java.util.stream.ReferencePipeline$3@e9e54c2
java.util.stream.ReferencePipeline$3@65ab7765
java.util.stream.ReferencePipeline$3@1b28cdfa
```

Each item in strm2 is a stream of `int[]`

```
[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

Example without `flatMap`

```
List<Integer> l1 = Arrays.asList(2,3,7,9);
List<Integer> l2 = Arrays.asList(4,5,8);

Stream<Stream<int[]>> strm2 =
      l1.stream()
        .map(i -> l2.stream()
                    .map(j -> {new int[]{i,j}));
```

~~`strm2.forEach(System.out::println);`~~

```
strm2.forEach(a -> a.forEach(System.out::println));
```

Each item output is an `int[]`

```
[I@1b28cdfa
[I@eed1f14
[I@7229724f
[I@4c873330
[I@119d7047
[I@776ec8df
[I@4eec7777
[I@3b07d329
[I@41629346
[I@404b9385
[I@6d311334
[I@682a0b20
```

```
[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

Example without `flatMap`

```
List<Integer> l1 = Arrays.asList(2,3,7,9);
List<Integer> l2 = Arrays.asList(4,5,8);

Stream<Stream<int[]>> strm2 =
      l1.stream()
        .map(i -> l2.stream()
                    .map(j -> {new int[]{i,j}));
```

~~`strm2.forEach(a -> a.forEach(System.out::println));`~~

```
strm2.forEach(a -> a.forEach(b -> System.out.println(Arrays.toString(b))));
```

Print contents of each array `int[]`

```
[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

With `flatMap`

```
List<Integer> l1 = Arrays.asList(2,3,7,9);
List<Integer> l2 = Arrays.asList(4,5,8);

Stream<int[]> strm2 =
      l1.stream()
        .flatMap(i -> l2.stream()
                     .map(j -> {new int[]{i,j}));
```

Nested `Stream<int[]>` has been flattened into a sequence of `int[]`

~~`strm2.forEach(a -> a.forEach(b -> System.out.println(Arrays.toString(b))));`~~

```
strm2.forEach(a -> System.out.println(Arrays.toString(a));
```

Print contents of each array `int[]`

```
[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

Try with `IntStream` instances:

```
int[] arr1 = {2,3,7,9};
int[] arr2 = {4,5,8};
IntStream is1 = Arrays.stream(arr1);
IntStream is2 = Arrays.stream(arr2);

is1.map(i -> new int[]{1,i})
   .forEach(a -> System.out.println(Arrays.toString(a)));
```

Won't work because the map function to `IntStream` must result in another `IntStream`, but here we want a `Stream<int[]>`

Useful Stream Operations

flatMap

Convert to `stream<Integer>` instead with `boxed()`, then apply `Stream.map`

```
IntStream is1 = Arrays.stream(arr1);
IntStream is2 = Arrays.stream(arr2);

Stream<int[]> pairs =

is1.boxed()
   .flatMap(i ->
                is2.boxed()
                 .map(j -> new int[]{i,j}));
```

returns `Stream<Integer>`

Won't work because the stream `is2` is used up for the first item of `is1`, and will be closed.

A new stream will have to be opened on `arr2` for every item in `is1`

Useful Stream Operations

flatMap

Convert to `stream<Integer>` instead with `boxed()`, then apply `Stream.map`

```
IntStream is1 = Arrays.stream(arr1);

Stream<int[]> pairs =
is1.boxed()
   .flatMap(i ->
                Arrays.stream(arr2).boxed()
                   .map(j -> new int[]{i,j}));

  pairs
     .forEach(p -> System.out.println(Arrays.toString(p)));
```

A new stream is opened on `arr2` for every item in `is1`

Useful Stream Operations

flatMap

Alternatively, can apply `IntStream.mapToObj` to second stream, without having to box

```
IntStream is1 = Arrays.stream(arr1);

Stream<int[]> pairs =
is1.boxed()
   .flatMap(i ->
                Arrays.stream(arr2)
                   .mapToObj(j -> new int[]{i,j}));

  pairs
     .forEach(p -> System.out.println(Arrays.toString(p)));
```

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[])
    movies.stream()
          ...
          .toArray();
```

This cast does not work

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	