# CS 213 – Software Methodology

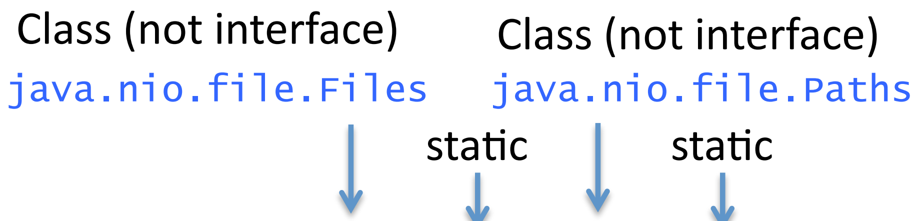
# Spring 2019

Lecture 25: Apr 30

Streams – Part 2

# Data Sources for Streams (Continued from previous lecture)

# 4. File



```
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:

```java
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)
.limit(5)
.forEach(System.out::println);
```

infinite sequence of random numbers
`(Stream<Double>)`

`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)

  .map(mm -> mm[0])

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

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

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

?

```
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

## 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());
}
```

This version of `reduce` 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. The return type of this reduce is `Optional<T>`

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

# Reduce – `mapToInt/sum`

Product – Using an identity element as seed

E.g. find the factorial of n

```
IntStream is = IntStream.rangeClosed(1,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
      .mapToInt(line -> line.split(" ").length)
      .sum()
   );
} catch (IOException e) {
   System.out.println(e.getMessage());
}
```

Returns an `IntStream`

Can also do max and min reductions on `IntStream`

## 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

# Reduce – Averaging with IntStream

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

```
OptionalDouble optDbl =
    movies.stream()
          .mapToInt(Movie::getRating)
          .average();

System.out.println(optDbl.orElse(0));
```

← mapToInt returns IntStream

# flatMap (Funky Stream Operation)

# 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(s -> s.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(s -> s.forEach(System.out::println));`~~

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

Print contents of each `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(s -> s.forEach(a -> System.out.println(Arrays.toString(a))));~~

```
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]
```

## 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.

# 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
```

# flatMap

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

So, “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
...
```

# 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
```