CS 213 – Software Methodology

Spring 2019

Lecture 26: May 2

Streams – Part 3

`flatMap` (Funky Stream Operation)

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 compile because the map function to `IntStream` must result in another `IntStream`, but here we are trying for 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	