**LECTURE NOTES (DRAFT)**

Streams

Stream<String> s1 = Stream.*of*(**"Mariapia"**, **"Enrico"**, **"Stefano"**);

Stream<Integer> s2 = Stream.*of*(12, 34, 55);

s1.filter(x -> x.length() > 6)  
 .forEach(System.***out***::println);

different from collections: traversable only once, internal iteration

This is OK:

Stream s3 = s2.filter(x -> x > 30);  
s3.forEach(System.***out***::println);

This not (!!):

s2.filter(x -> x > 30);  
s2.forEach(System.***out***::println);

terminal: forEach, count, collect, ...

System.***out***.println(  
 s3.count()  
);

List<Integer> list1 = s2.filter(x -> x > 30).collect(Collectors.*toList*());  
list1.forEach(System.***out***::println);

Single pass:

List<Integer> list1 = s2.filter(x -> x > 30)  
 .filter(x -> x % 2 == 0)  
 .collect(Collectors.*toList*());

Execution starts with the terminal operator

filtering with predicate, unique elements: filter, distinct, ...

List<Integer> list2 = s2.filter(x -> x > 30)  
 .filter(x -> x % 2 == 0)  
 .distinct()  
 .collect(Collectors.*toList*());

List<String> list1 = s1.filter(x -> x.startsWith(**"M"**))  
 .collect(Collectors.*toList*());

truncating: limit, skipping, ...

List<String> list1 = s1.filter(x -> x.length() > 2)  
 .limit(2)  
 .collect(Collectors.*toList*());

List<String> list1 = s1.filter(x -> x.length() > 2)  
 .limit(2)  
 .skip(1)  
 .collect(Collectors.*toList*());

finding: findFirst, findAny, anyMatch(), allMatch(), noneMatch(), [SHORT CIRCUIT]

Optional<Integer> value = s2.filter(x -> x > 30)  
 .filter(x -> x % 2 == 0)  
 .findFirst();  
System.***out***.println(value.get());

findAny() is better for parallel execution

but if there is no value?

**if** (value.isPresent())  
 System.***out***.println(value.get());

or:

value.ifPresent(System.***out***::println);

if any value is OK

Optional<Integer> value = s2.filter(x -> x > 100)  
 .filter(x -> x % 2 == 0)  
 .findAny();  
value.ifPresent(System.***out***::println);

Optional. isPresent(), ifPresent(doSomething), get(), orElse(doSomething)

All match:

**boolean** value2 = s2.allMatch(x -> x > 2);

anyMatch:

**boolean** value2 = s2.anyMatch(x -> x > 32);

noneMatch:

**boolean** value2 = s2.noneMatch(x -> x > 32);

maping: map, flatMap

lower case:

List<String> list1 = s1.map(String::toLowerCase)  
 .collect(Collectors.*toList*());

first caracter:

List<Character> list1 = s1.map(x -> x.charAt(0))  
 .collect(Collectors.*toList*());

length:

List<Integer> list1 = s1.map(x -> x.length())  
 .collect(Collectors.*toList*());

sorted(Comparing(...))

lower case sorting:

List<String> list1 = s1.map(String::toLowerCase)  
 .sorted()  
 .collect(Collectors.*toList*());

sorting by length:

List<String> list1 = s1.map(String::toLowerCase)  
 .sorted(Comparator.*comparing*(String::length))  
 .collect(Collectors.*toList*());

List<String> list1 = s1.map(String::toLowerCase)  
 .sorted(Comparator.*comparing*(x -> x.))  
 .collect(Collectors.*toList*());

by surnames:

List<String> list1 = s1.map(String::toLowerCase)  
 .sorted(Comparator.*comparing*((Function<String, String>)(x -> x.substring(x.indexOf(**" "**)))))  
 .collect(Collectors.*toList*());

creation operations: .stream()

List<String> list4 = Arrays.*asList*(**"Stefano"**, **"Mariapia"**, **"Enrico"**);

list4.stream().count();

HashSet<String> set1 = **new** HashSet<>();  
set1.stream().count();

creation: Stream.empty(), Stream.of(......),

Stream s4 = Stream.*empty*();

Stream s5 = Stream.*of*(list1, list4);

Numeric streams:

numeric ranges: rangeClosed(), range().

IntStream ints1 = IntStream.*range*(0, 10);  
System.***out***.println(  
 ints1.filter(x -> x % 2 == 0).count()  
);

mapToInt(...) -> int, sum(), max(), min(),

Stream<Integer> s2 = Stream.*of*(12, 34, 34, 55, 102);  
System.***out***.println(  
 s2.mapToInt(x -> x + 1).sum()  
);

Stream<Integer> s2 = Stream.*of*(12, 34, 34, 55, 102);  
System.***out***.println(  
 s2.mapToInt(x -> x + 1).max()  
);

OptionalInt[103] ! Be careful!

Stream<Integer> s2 = Stream.*of*(12, 34, 34, 55, 102);  
System.***out***.println(  
 s2.mapToInt(x -> x + 1).min()  
);

be careful with return type: OptiionalInt

Creation: Arrays.stream(...),

**int** [] a = {1, 2, 3};  
System.***out***.println(Arrays.*stream*(a).sum());

Streams from functions, infinite streams:

Stream.iterate(...), Stream.generate(...)

LongStream st1 = LongStream.*iterate*(2, x -> x \* x);  
**long** []b = st1.limit(5).toArray();  
Arrays.*stream*(b).forEach(System.***out***::println);

Stream.*generate*(Math::*random*)  
 .limit(5)  
 .forEach(System.***out***::println);

flatMap:

**int** []c = IntStream.*rangeClosed*(0, 2).  
 flatMap(x -> IntStream.*rangeClosed*(0, 2).  
 map(y -> x + y))  
 .toArray();  
Arrays.*stream*(c).forEach(System.***out***::println);

List<Integer> m2 = Stream.*of*(Arrays.*asList*(1,2,3), Arrays.*asList*(4,5,6))  
 .flatMap(x -> x.stream())  
 .collect(Collectors.*toList*());

Collectors

A nice example for strings:

Stream m4 = Stream.*of*(**"Stefano"**, **"Mariapia"**, **"Enrico"**);  
System.***out***.println(  
 m4.collect(Collectors.*joining*(**","**,**"["**,**"]"**))  
);

We can count:

Stream m3 = Stream.*of*(12, 34, 34, 55, 102);  
System.***out***.println(  
 m3.collect(Collectors.*counting*())  
);

We can average:

Stream m5 = Stream.*of*(12, 34, 34, 55, 102);  
System.***out***.println(  
 m5.collect(Collectors.*averagingInt*(x -> (**int**)x))  
);

We can summarize:

Stream m6 = Stream.*of*(12, 34, 34, 55, 102);  
System.***out***.println(  
 m6.collect(Collectors.*summarizingInt*((x -> (**int**) x)))  
);

Also applyes to integer streams:

System.***out***.println(  
 ints2.sum()  
);  
  
IntStream ints3 = IntStream.*range*(0, 10);  
System.***out***.println(  
 ints3.average()  
);  
  
IntStream ints4 = IntStream.*range*(0, 0);  
System.***out***.println(  
 ints4.average()  
);

Note the optional! Do not forget it!

We can also group:

Book b1 = **new** Book(**"Java 8 lambdas"**, **"Richard Warbuton"**, 182, **"O'Reilly"**);  
Book b2 = **new** Book(**"Java 8 in action"**, **"Raoul-Gabriel Urma"**, 497, **"Manning"**);  
Book b3 = **new** Book(**"Functional thinking"**, **"Neal Ford"**, 179, **"O'Reilly"**);  
Book b4 = **new** Book(**"Learning scala"**, **"Jason Swartz"**, 255, **"O'Reilly"**);  
Book b5 = **new** Book(**"Parallel and concurrent programming in Haskell"**, **"Simon Marlow"**, 321, **"O'Reilly"**);  
Book b6 = **new** Book(**"Presentation patterns"**, **"Neal Ford"**, 265, **"Addisson Wesley"**);  
  
List<Book> books = Arrays.*asList*(b1, b2, b3, b4, b5, b6);

And group by author:

Map<String, List<Book>> book1 = books.stream()  
 .collect(Collectors.*groupingBy*(Book::getAuthor));

by publisher:

Map<String, List<Book>> book2 = books.stream()  
 .collect(Collectors  
 .*groupingBy*(Book::getPublisher));

books about Java:

Map<String, List<Book>> book3 = books.stream()  
 .collect(Collectors.  
 *groupingBy*(x -> (x.getTitle().contains(**"Java"**))

? **"Java"** : **"Other"**));

Two levels: by author and publisher:

Map<String, Map<String, List<Book>>> book4 = books.stream()  
 .collect(Collectors  
 .*groupingBy*(Book::getAuthor, Collectors  
 .*groupingBy*(Book::getPublisher)));

Counting:

Map<String, Long> book5 = books.stream().collect(Collectors  
 .*groupingBy*(Book::getAuthor, Collectors.*counting*()));

Getting information about the total number of pages:

Map<String, Integer> book6 = books.stream()  
 .collect(Collectors  
 .*groupingBy*(Book::getAuthor, Collectors  
 .*summingInt*(Book::getNumberOfPages)));

maximun number of pages:

Map<String, Optional<Book>> book7 = books.stream()  
 .collect(Collectors  
 .*groupingBy*(Book::getAuthor, Collectors  
 .*maxBy*(Comparator  
 .*comparingInt*(Book::getNumberOfPages))));