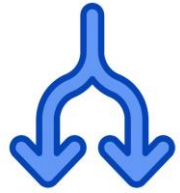
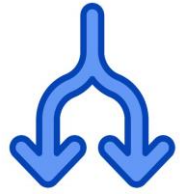


Practical Concurrent and Parallel Programming VII

Streams and Parallel Streams

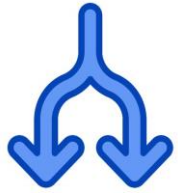
Jørgen Staunstrup

Agenda



- **Motivation and introduction**
- Lambda Expressions
- Java Stream
- Concurrency

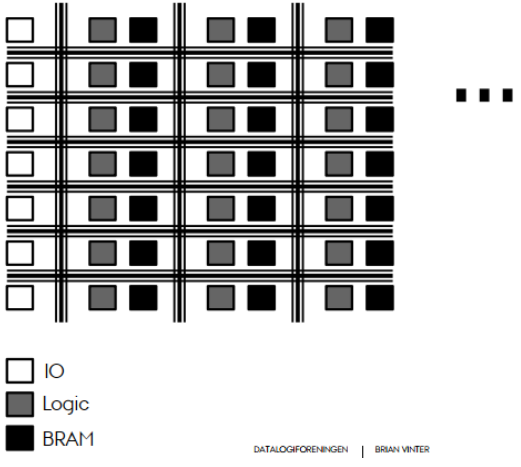
Motivation



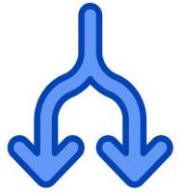
Data is often communicated as a stream

- from a file,
- from the internet
- from the user interface
- …

Supercomputer



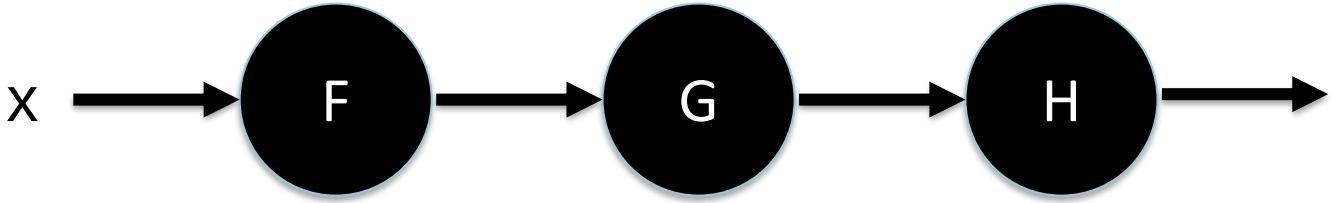
Streams and Reactive programming



In week 9 we will look at RxJava

Programming with streams and reactively have
- many similarities
- and some important differences

Java Stream

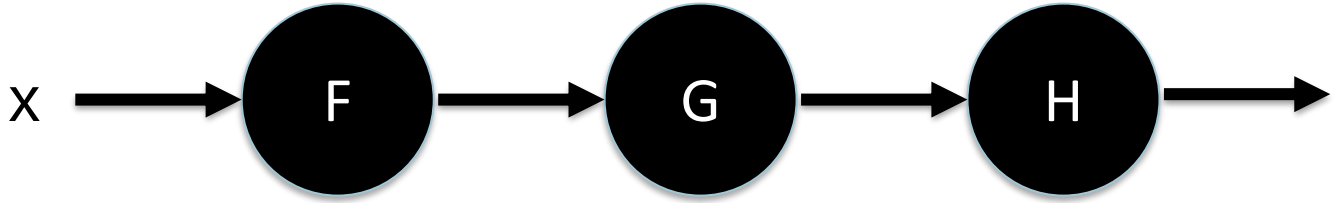



```
Java Stream
```

```
x().F().G().H()
```

Closely related to: H(G(F(x))) functional programming

Stream example



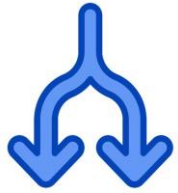
X is a stream of words: ….. ablaut able ableeze … zyme

F discards all words with 1 character

G discards all words that are capitalized

H counts number of words

Java

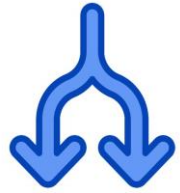


```
public static String F(String s) {
  return s.length() > 1 ? s : null;
}

public static String G(String s) {
  if (s==null) return null;
  return Character.isLowerCase(s.charAt(0)) ? s : null;
}

public static int H(String s) {
  return s==null ? 0 : 1;
}
...
while ((line= reader.readLine()) != null) {
  word= F(line);
  smallLetters= G(word);
  count+= H(smallLetters);
}
```

Java stream

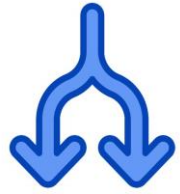


```
x().F().G().H()
```

```
count= readWords(filename)  // makes a stream of words
   .filter( w -> w.length()>1 ) // F
   .filter( w -> Character.isLowerCase(w.charAt(0)) ) // G
   .count(); // H
```

```
count= readWords(filename)
   .parallel()
   .filter( w -> w.length()>1 )
   .filter( w -> Character.isLowerCase(w.charAt(0)) )
   .count();
```

Java stream



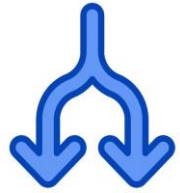
```
x().F().G().H()
```

```
count= readWords(filename)  // makes a stream of words
   .filter( w -> w.length()>1 ) // F
   .filter( w -> Character.isLowerCase(w.charAt(0)) ) // G
   .count(); // H
```

A sequence of stream method calls is commonly known as *stream pipeline*

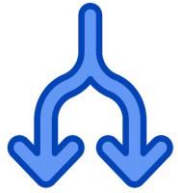
Lambda expressions

Agenda



- Motivation and introduction
- **Lambda Expressions**
- Java Stream
- Concurrency

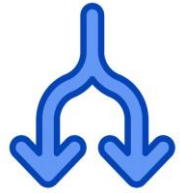
Java stream



```
count= readWords(filename)  // makes a stream of words
   .filter( w -> w.length()>1 )
   .count(); // H
```

Lambda expression

Lambda expressions

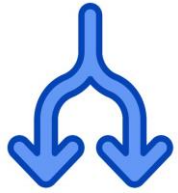


Argument type Result type Lambda expr.

- One argument
  - `Function<Integer, Integer> f = (x) -> x+1`
  - $f : \mathbb{Z} \rightarrow \mathbb{Z} \mid f(x) = x + 1$
  - $f(1)$ <-> `f.apply(1)`
- Two arguments
  - `BiFunction<Integer,Integer,Integer> f = (x,y) -> x+y`
  - $f : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \mid f(x,y) = x + y$
  - $f(1,2)$ <-> `f.apply(1,2)`

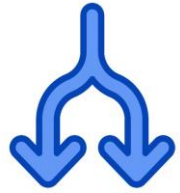
https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Lambda expressions (2)



- Zero arguments
  - `Supplier<Integer> f = () -> 2`
  - $f : \mathbb{Z} |\ \ f() = 2$
  - $f()$ <-> `f.get()`
- No return type (`void`)
  - `Consumer<Integer> f = (x) -> System.out.println(x);`
  - $f(2)$ <-> `f.accept(2)`

Method references

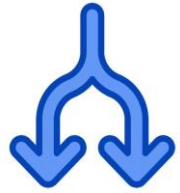


The methods of an object may also be referenced as follows

- `Class::method`
  - `BiFunction<String,Integer,Character> f = String::chartAt`
    - `f.apply(s,i) <-> s.charAt(i)`
  - `Function<Person,String> f = Person::getName`
  - `System.out::println`
  - …
- <Object instance>::method
  - `Function<Integer,Character> f = "01234"::charAt`
    - `f.apply(i) <-> "01234".charAt(i)`

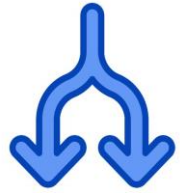
javaprecisely-3rd-draft-streams.pdf

Agenda



- Motivation and introduction
- Lambda Expressions
- **Java Stream**
- Concurrency

Java stream



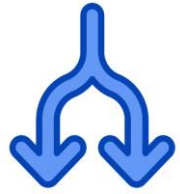
The streams are lazy (driven by the terminal operation)

```
count= readWords(filename)  // source
   .filter( w -> w.length()>1 ) // intermediate
   .filter( w -> Character.isLowerCase(w.charAt(0)) )
   .count(); // terminal
```

There are three different types of stream elements:

- *sources* (arrays, collections, IO, generators)
- *intermediate operations* (transforming one stream into another (e.g. filter)
- *terminal operations* (count, sum, forEach, ...)

Stream sources



Provides the data for the stream

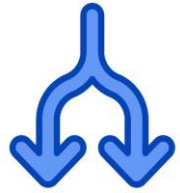
Examples of stream sources are:

- Input (files or network)
- The Arrays class has a number of utilities, for example:
  `Arrays.stream(arr)`

Other examples of collections?

https://howtodoinjava.com/java/stream/java-streams-by-examples/

Stream sources



Provides the data for the stream

Examples of stream sources are:

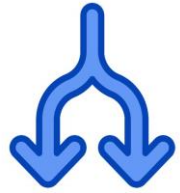
- Input (files or network)
- The Arrays class has a number of utilities, for example:
  `Arrays.stream(arr)`

List, Set, Map, …

- All java collections have a **`stream()`** method
- **`Stream.of("Huey", "Dewey" "Louie")`**
  returns a stream of the three strings

https://howtodoinjava.com/java/stream/java-streams-by-examples/

Intermediate operators



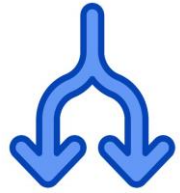
A computation/transformation on *each* element of the string

See Sestoft's Java precisely and the java documentation for a complete list

Examples of intermediate operations are:

- filter – takes a lambda expression lambda returning a boolean, if the boolean is true the element is included in the output stream
- map – transforms each element
- limit(*n*) – returns a stream of the first *n* elements
- skip(*n*) – returns a stream without the first *n* elements
- distinct – returns a stream without duplicated elements
- sorted - returns a stream with the elements sorted

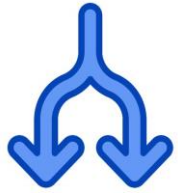
Intermediate operations



```
count= readWords(filename)  // makes a stream of words
   .filter( w -> w.length()>1 ) // F
   .filter( w -> Character.isLowerCase(w.charAt(0)) ) // G
   .count(); // H
```

Which operation(s) is/are intermediate?

Intermediate operations

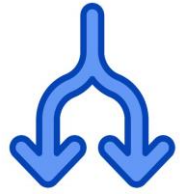


```
count= readWords(filename)  // makes a stream of words
   .filter( w -> w.length()>1 ) // F
   .filter( w -> Character.isLowerCase(w.charAt(0)) ) // G
   .count(); // H
```

filter

BufferedReader (file)

Exercise: 7.3



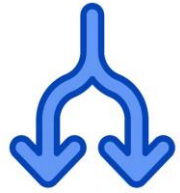
```
public static void main(String[] args) {
    String filename = "src/main/resources/english-words.txt";
    System.out.println( readWords(filename).count());
}

public static Stream<String> readWords(String filename) {
  try {
    BufferedReader reader= new BufferedReader(new FileReader(filename));
      return ... // TO DO: Implement properly
} catch (IOException exn) { return Stream.<String>empty();  }
}
```

https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html

`Stream<String>`	`lines()` Returns a `Stream`, the elements of which are lines read from this `BufferedReader`

BufferedReader (URL)



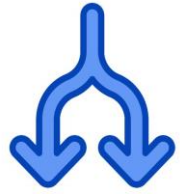
Exercise: 7.3

```
String filename = "https://staunstrups.dk/jst/english-words.txt";
System.out.println(readWordsFromURL(urlname).count());


public static Stream<String> readWordsFromURL(String urlname) {
  try {
    HttpURLConnection connection=
              (HttpURLConnection) new URL(urlname).openConnection();
    BufferedReader reader=
              new BufferedReader(new InputStreamReader(connection.getInputStream()));
    return reader.lines();
 } catch (IOException  exn) { return Stream.<String>empty(); }
}
```

https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html

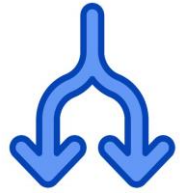
Defining a Java stream



- Using the Arrays class
  - **`Arrrays.stream(array)`**
- Most Java collections have a method **`stream()`** that turns the collection into a stream
- **`Stream.of(1,2,3,4)`** creates a stream with those elements
- Functional iterators for infinite streams
  - **`IntStream nats = IntStream.iterate(0, x->x+1)`**
- **`BufferedReader`** (important for exercises)

`Stream<String>`	`lines()` Returns a `Stream`, the elements of which are lines read from this `BufferedReader`

Terminal operations



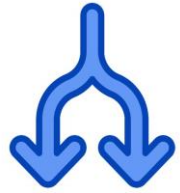
Provides the *result* of the stream computation

Examples of terminal operations are:

- min, max, sum, average, count (number streams)
- forEach e.g., forEach(System.out::println)
- reduce / collect (introduced shortly)

https://howtodoinjava.com/java/stream/java-streams-by-examples/

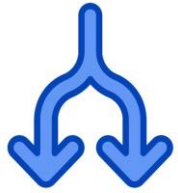
Terminal operations



```
count= readWords(filename)  // makes a stream of words
   .filter( w -> w.length()>1 ) // F
   .filter( w -> Character.isLowerCase(w.charAt(0)) ) // G
   .count(); // H
```

Which operation(s) is/are terminal?

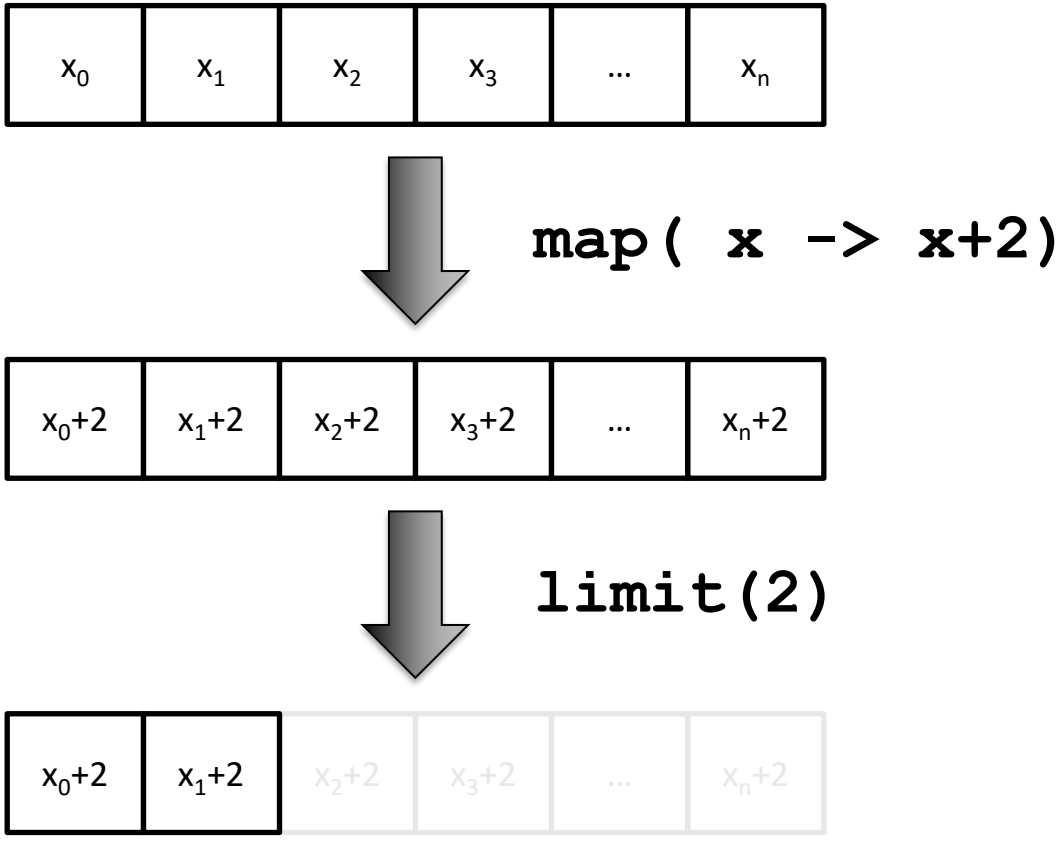
Intermediate operations



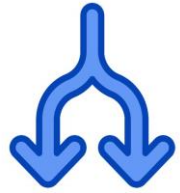
```
count= readWords(filename)  // makes a stream of words
   .filter( w -> w.length()>1 ) // F
   .filter( w -> Character.isLowerCase(w.charAt(0)) ) // G
   .count(); // H
```

count

Intermediate operations

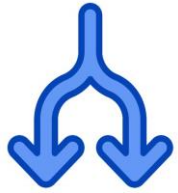


Terminal operation reduce



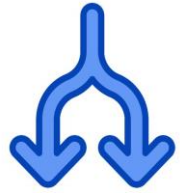
- Reduce all elements of the stream to a single value by applying a function
- **`reduce(identity, accumulator)`**
  - **`identity`**: The identity element is both the initial value of the reduction and the default result if there are no elements in the stream.
  - **`accumulator`**: The accumulator function takes two parameters: a partial result of the reduction and the next element of the stream.
- Example
  - Sum of squares of first 100 natural numbers
    - `IntStream.range(0,100).reduce(0, (a,b) -> a+b*b))`

Reduce without identity



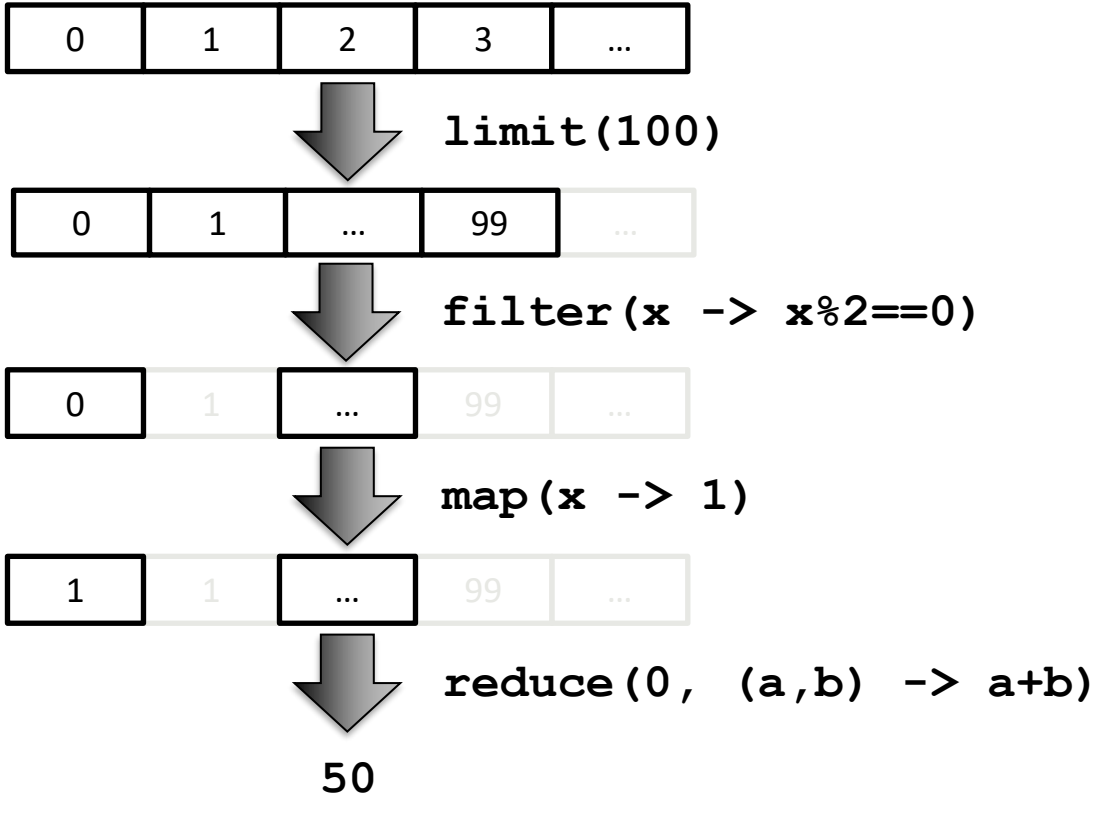
- Example
  - Sum of squares of first 100 natural numbers
    - `IntStream.range(0,100).reduce((a,b) -> a+b*b).  )`

Reduce without identity

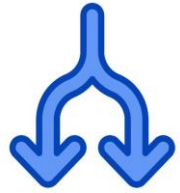


- Reduce can also be called without identity parameter
- Then it returns an `Optional` value
  - A container object which may or may not contain a non-null value.
  - Needed in case the reduction is performed on an empty stream.
- Example
  - Sum of squares of first 100 natural numbers
    - `IntStream.range(0,100).reduce((a,b) -> a+b*b).orElse(0))`
- There are other built-in reductions: sum, max, min, average, etc…

Example with everything so far



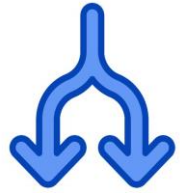
Example with everything so far



- Here is an example with everything we have seen so far
  - Amount of even numbers in the range 0 to 99

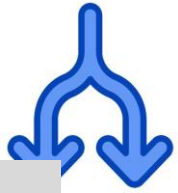
```
IntStream.iterate(0,x->x+1)
         .limit(100)
         .filter(x -> x%2==0)
         .map(x -> 1)
         .reduce(0, (a,b) -> a+b);
```

Agenda



- Motivation and introduction
- Lambda Expressions
- Java Stream
- **Concurrency**

Counting primes on Java 8 streams

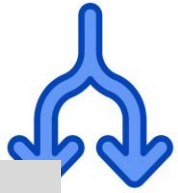


Our old standard Java for loop:

Classical efficient imperative loop

```
int count = 0;
for (int i=0; i<range; i++)
  if (isPrime(i))
    count++;
```

Counting primes on Java 8 streams



Our old standard Java for loop:

Classical efficient imperative loop

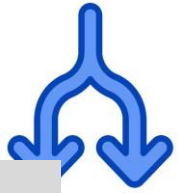
```
int count = 0;
for (int i=0; i<range; i++)
  if (isPrime(i))
    count++;
```

Sequential Java 8 stream:

Functional programming ...

```
IntStream.range(0, range)
.filter(i -> isPrime(i))
.count()
```

Counting primes on Java 8 streams



Our old standard Java for loop:

Classical efficient imperative loop

```
int count = 0;
for (int i=0; i<range; i++)
  if (isPrime(i))
    count++;
```

Sequential Java 8 stream:

Functional programming ...

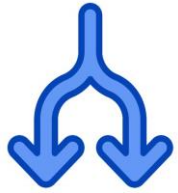
```
IntStream.range(0, range)
.filter(i -> isPrime(i))
.count()
```

Parallel Java 8 stream:

... and thus parallelizable and thread-safe

```
IntStream.range(0, range)
.parallel()
.filter(i -> isPrime(i))
.count()
```

Performance results



Counting the primes in 2 ... 100.000

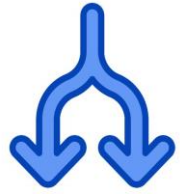
Using Mark7

Sequential	4891635.9 ns	21879.73	64
Stream	4953867.6 ns	63873.82	64
ParallelStream	1363886.8 ns	10621.99	256

PCPP 2019

Intel i7 (4 cores) speed-up: 3.6 x

Ordering



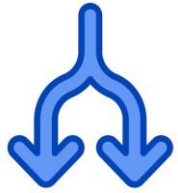
Streams may or may not have a defined *encounter order*

- **`List`** and **`Arrays`** are intrinsically ordered
- **`HashSet`** is not ordered

An intermediate operation e.g., **`sorted`** transforms an unordered **`Stream`** into an ordered **`Stream`**

https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Ordering

Parallel (intermediate)



```
IntStream.range(0, range)
.parallel()
.filter(i -> isPrime(i))
.count()
```

← ordered

← ?

`IntStream.range` produces an *ordered* stream

`parallel` is an operation in `BaseStream`

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util.stream

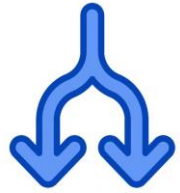
Interface BaseStream<T,S extends BaseStream<T,S>>

`parallel()`

Returns an equivalent stream that is parallel.

https://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html

Interfering streams

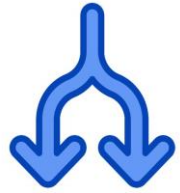


- If you try to modify a stream you are operating you will get a **`ConcurrentModificationException`** at runtime
  - So don't do it ☺

- Cannot be detected at compile time. It depends on the programmer.

- From the Java documentation
  - ***Streams enable you to execute possibly-parallel aggregate operations*** *over a variety of data sources, including even non-thread-safe collections such as ArrayList.* ***This is possible only if we can prevent interference with the data source during the execution of a stream pipeline****. […] For most data sources, preventing interference means ensuring that the data source is not modified at all during the execution of the stream pipeline.*

Example: stream of objects

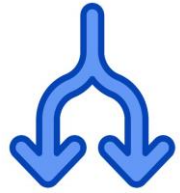


```
class Employee {
  int id;
  String dept;
  int salary;

  public Employee(int id, String dept, int salary) {
    this.id     = id;
    this.dept   = dept;
    this.salary = salary;
  }
  public int getId() { return this.id; }
  public String getDept() { return this.dept; }
  public int getSalary() { return this.salary; }
}

static private Stream<Employee> randomEmployees() {
...
}
```

Terminal operation collect

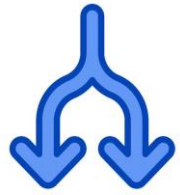


- It is a reduction operation that allows to collect the results of a stream into a Java collection or summarize them using complex criteria

- For instance, converting a stream into a list

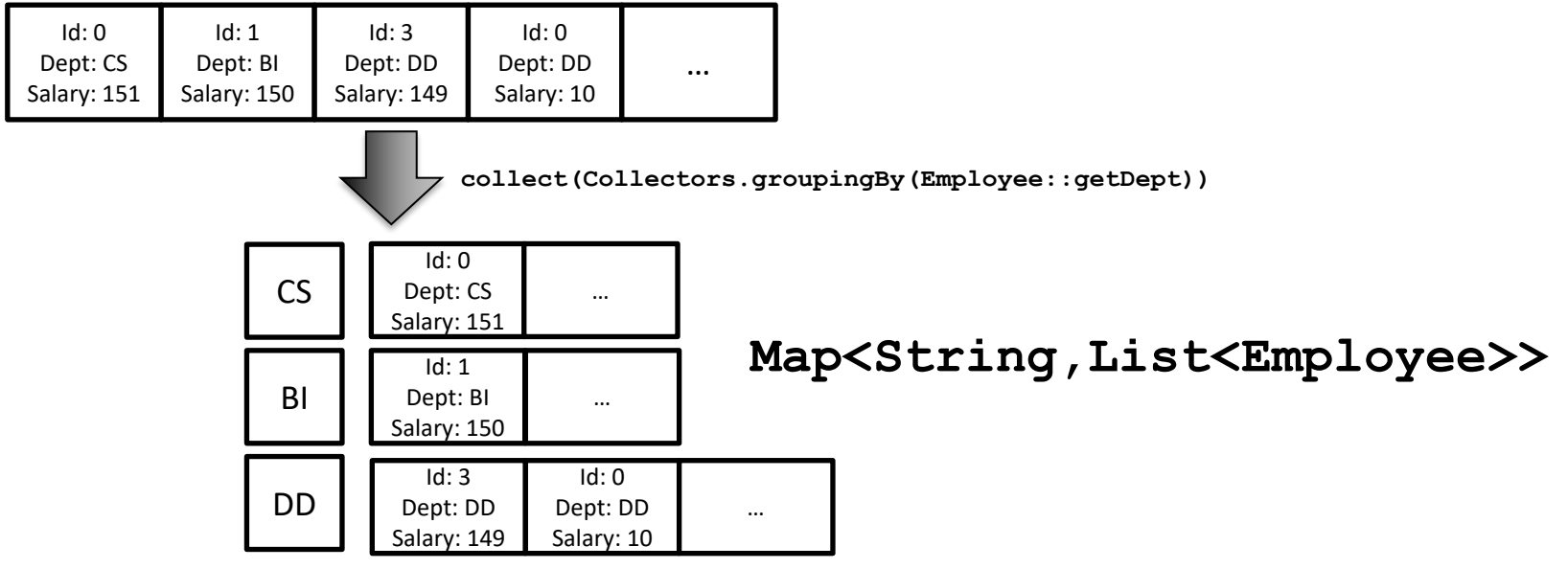
```
List<Integer> l = randomEmployees()
        .limit(50)
        .map(Employee::getId)
        .collect(Collectors.toList());
```

Terminal operation: collect + groupingBy

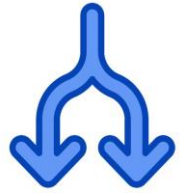


- Group employees by department

```
Map<String,List<Employee>> m = randomEmployees()
        .limit(50)
        .collect(Collectors.groupingBy(Employee::getDept));
```

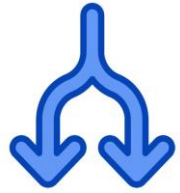


Printing the result



```
randomEmployees()
   .limit(50)
   .collect(Collectors.groupingByConcurrent(Employee
   ::getDept))
     .forEach((k,v) ->
      System.out.println(k+":     "+v
         .stream()
         .map(Employee::getId)
         .collect(Collectors.toList())
      ));
```

Summary



- A Java stream is a finite or infinite sequence of Objects
- Java streams use lazy evaluation
- The execution of operations over Java streams is typically efficient
- Operations on (unordered)Java streams may be easily executed parallel