Practical Concurrent and Parallel Programming V

Parallel Streams

Raúl Pardo

Exercise rooms

- Since rooms 2A12-14 have been sufficient for exercises in the past weeks, we have cancel the booking of the other two rooms.

Oral Feedback Sessions

- Oral feedback sessions are supposed to be fixed
  - Once you pick a slot it cannot change. Exceptionally some weeks you may negotiate alternative times with the TA/teacher you picked. But you should not change your choice in the scheduler.
  - Please note that for TAs/teachers to effectively prepare for exercise sessions we need stability and minimize changes

- Requests for written feedback have increased (too much)
  - Written feedback should be your last resource, you should use it only if it is not possible to attend oral feedback sessions.
    - We are open discuss alternative slots for oral feedback if this is the problem
  - Remember also that the exam will be like an oral feedback session. *There is no better preparation for the exam than attending oral feedback sessions.*

Q&A Forum

- I haven't forgotten about answering the comment on answering questions on the slides, it is coming this week
  - For this week, I already have a hidden version of the slides with answers to the questions

Previously on PCPP…

- Thread-safe classes
- Safe publication
- Immutability
- Instance confinement
- Synchronization primitives (synchronizers)
  - Semaphores
  - Barriers
- Producer-consumer problem

Agenda

- Data independence
- Lambda expressions
- Java Streams
- Parallel Java Streams

Thread-safety

"Writing thread-safe code is, at its core, about ***managing access to shared mutable data****"*

Goetz

Data independence among threads

- Partitioning data so that threads only access disjoint memory is another way to ensure thread-safety

Data independence among threads

- Partitioning data so that threads only access disjoint memory is another way to ensure thread-safety

```
int parties         = 10;
CyclicBarrier cb    = new CyclicBarrier(parties);
int[] shared_array = new int[parties];
…
for (int i = 0; i < parties; i++) {
  new SetterClass(i).start();
}
…
public class SetterClass extends Thread {
  int index;
  public SetterClass(int index) {this.index = index;}

  public void run() {
    shared_array[index] = index+1;
    cb.await();
    // After this point the array is initialized and it is safe to read it
  }
}
```

Remember this example from last week

Data independence among threads

- Partitioning data so that threads only access disjoint memory is another way to ensure thread-safety

```
int parties         = 10;
int[] shared_array = new int[parties];
…
for (int i = 0; i < parties; i++) {
  new SetterClass(i).start();
}
…
public class SetterClass extends Thread {
  int index;
  public SetterClass(int index) {this.index = index;}

  public void run() {
    // If the thread only works on share_array[index] the program is thread-safe
    shared_array[index] = index+1;
  }
}
```

If threads only access disjoint regions of the array the program is thread-safe

Data independence among threads

- Partitioning data so that threads only access disjoint memory is another way to ensure thread-safety

```
int parties         = 10;
int[] shared_array = new int[parties];
…
for (int i = 0; i < parties; i++) {
  new SetterClass(i).start();
}
…
public class SetterClass extends Thread {
  int index;
  public SetterClass(int index) {this.index = index;}

  public void run() {
    // If the thread only works on share_array[index] the program is thread-safe
    shared_array[index] = index+1;
  }
}
```

If threads only access disjoint regions of the array the program is thread-safe

BTW, this is the basis for GPU computing

Data independence among threads

- Partitioning data so that threads only access disjoint memory is another way to ensure thread-safety
- Java does not have intrinsic mechanisms to ensure this; it depends on the programmer

Data independence among threads

- Partitioning data so that threads only access disjoint memory is another way to ensure thread-safety
- Java does not have intrinsic mechanisms to ensure this; it depends on the programmer
- However, some parts of the standard library may help us
  - ThreadLocal
  - Parallel Streams
  - …

Lambda Expressions

Lambda expressions

- One argument
  - `Function<Integer, Integer> f = (x) -> x+1`
  - $f : \mathbb{N} \rightarrow \mathbb{N} \;\mid\; f(x) = x + 1$
  - $f(1)$ <-> `f.apply(1)`

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

- … (you see where we are going)

Lambda expressions

- Zero arguments
  - `Supplier<Integer> f = () -> 2`
  - $f : \mathbb{N} \mid f() = 2$
  - $f()$ <-> `f.get()`

- No return type (`void`)
  - `Consumer<Integer> f = (x) -> System.out.println(x);`
  - $f : \mathbb{N} \rightarrow Unit \mid f(x) = ???$
  - $f(2)$ <-> `f.accept(2)`

Lambda expressions

- Zero arguments
  - `Supplier<Integer> f = () -> 2`
  - $f : \mathbb{N} \mid f() = 2$
  - $f()$ <-> `f.get()`

What about zero arguments and no return type? For instance,

`() -> System.out.println("Hej")`

- No return type (`void`)
  - `Consumer<Integer> f = (x) -> System.out.println(x);`
  - $f : \mathbb{N} \rightarrow Unit \mid f(x) = ???$
  - $f(2)$ <-> `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)`

Higher-order functions

- A *higher-order* function is a function that
  - Takes functions as parameters, and/or
  - Returns a function

- Java streams support higher-order functions
  - map, reduce, filter, etc…

Java Streams

Java streams

- 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 java streams may be easily executed parallel

Java streams

- 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 java streams may be easily executed parallel

Why can operations be easily parallelized?

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)`**
- **`Stream::generate`**
  - See file **`StreamExample.java`**
- **`BufferedReader`** (important for exercises)

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

Intermediate vs Terminal operations

- Operations on Java streams may be
  - Intermediate
  - Terminal

- Intermediate operations return a new stream
  - Symbolic; they are recorded but the computation is postponed

- Terminal operations return a value
  - Execute the pipeline of intermediate operations to compute the value

Intermediate operations

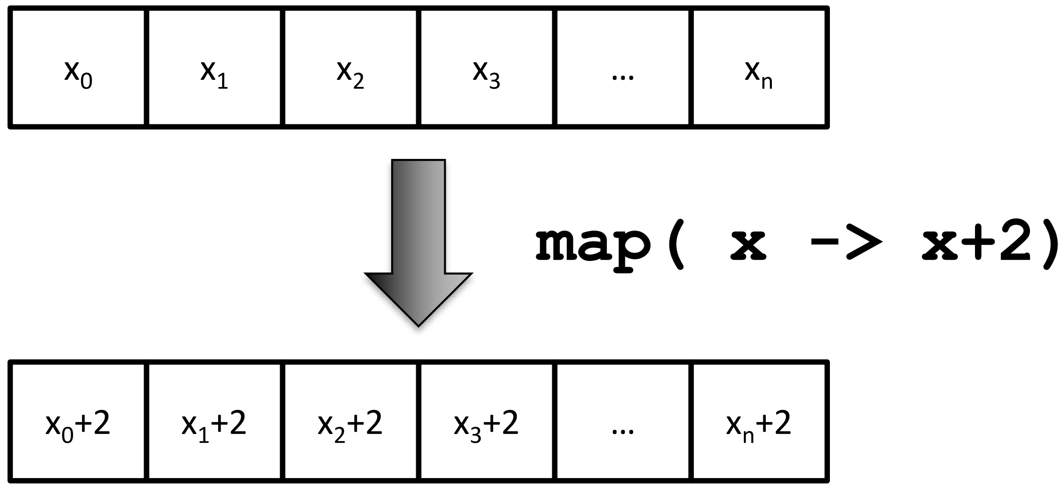
- Examples of intermediate operations are:
  - filter – takes a lambda expression lambda returning a boolean, if the boolean is true the element is included in the output stream
  - map – takes a lambda computing a value. The output stream consists of computed values.
  - limit(*n*) – takes an integer and returns a stream of the first *n* elements
  - skip(*n*) – takes an integer and 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

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 – takes a lambda computing a value. The output stream consists of computed values.
  - limit(*n*) – takes an integer and returns a stream of the first *n* elements
  - skip(*n*) – takes an integer and 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 | map



Intermediate operations | map

x_0	x_1	x_2	x_3	…	x_n

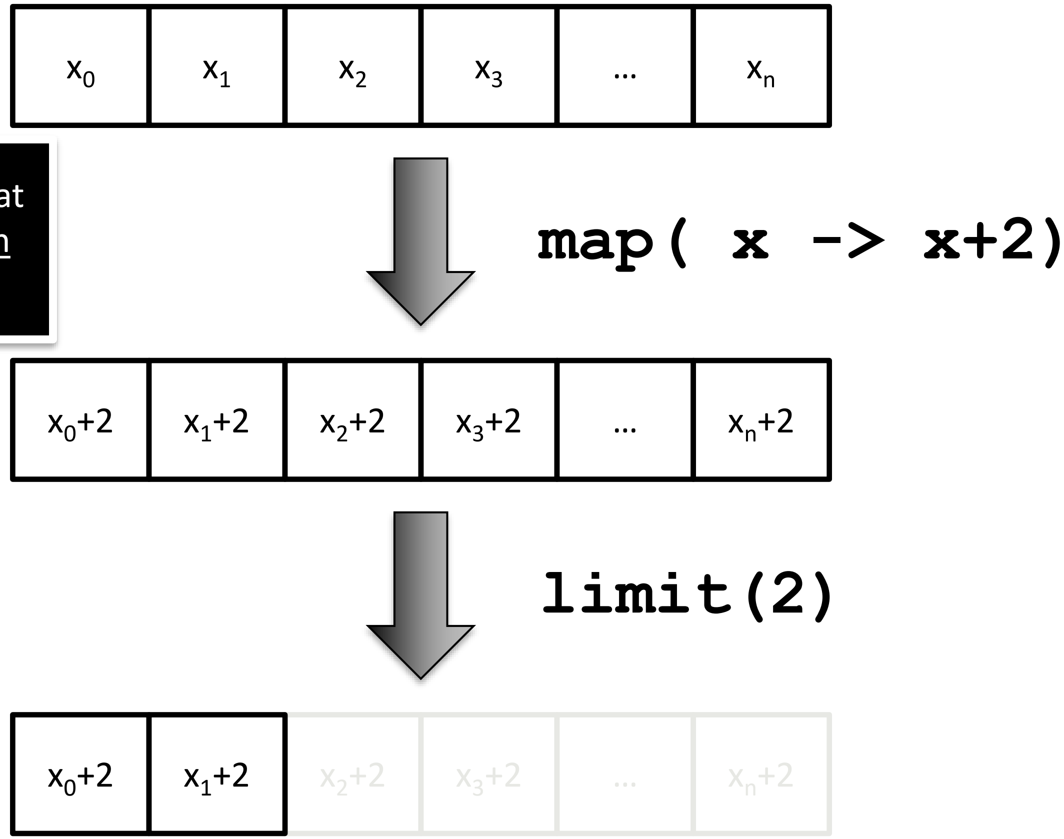
Here it is important to note that the computation has not been executed only recorded

`map( x -> x+2)`

x_0+2	x_1+2	x_2+2	x_3+2	…	x_n+2

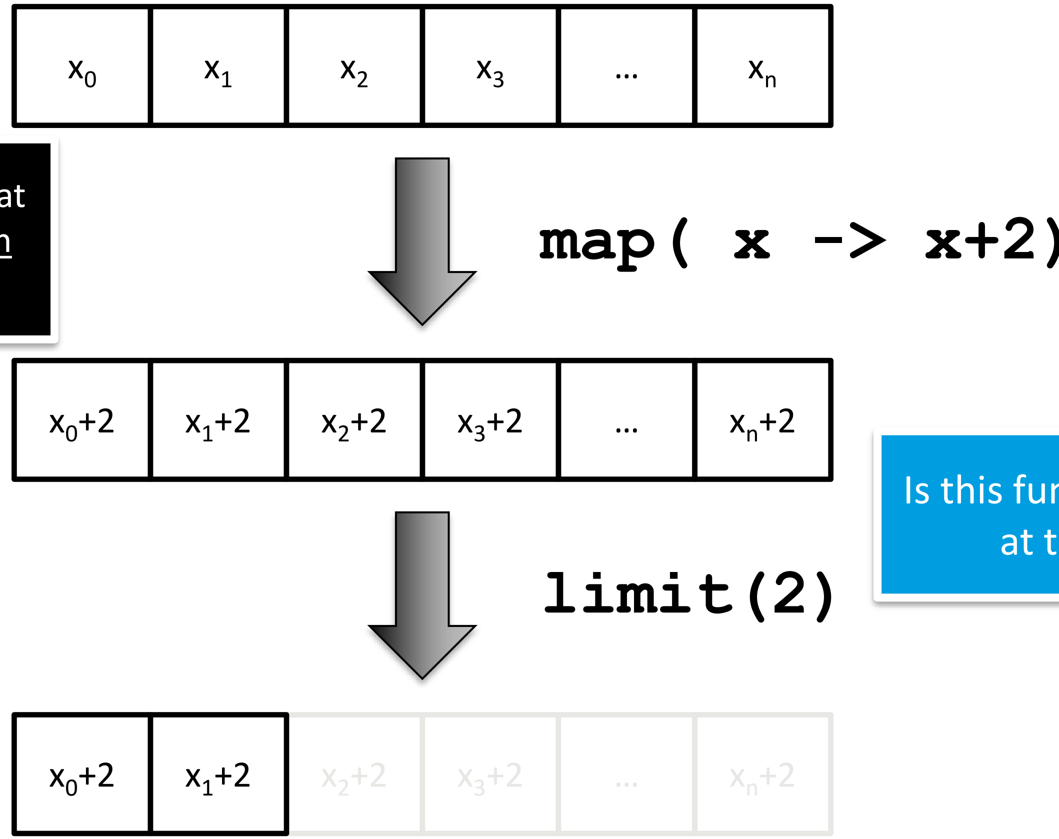
Intermediate operations | map & limit

Here it is important to note that the <u>computation has not been executed</u> only recorded



Intermediate operations | map & limit

Here it is important to note that the computation has not been executed only recorded



Is this function executed at this point?

Terminal operations

- Remember, terminal operations take a stream and produce value
  - Not lazy; they execute the pipeline of intermediate operations to compute the value

- Java streams divides further terminal operations into:
  - Reduce
  - Collect

Terminal operations | 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))`
- Implementation

```
T result = identity;
for (T element : this Stream)
  result = accumulator.apply(result, element)
return result;
```

Terminal operations | reduce

- 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 exist other built-in reductions: sum, max, min, average, etc…

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);
```

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

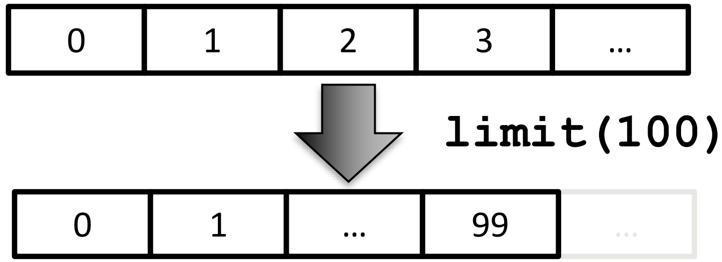
```
IntStream.iterate(0,x->x+1)
           .limit(100)
           .filter(x -> x%2==0)
           .map(x -> 1)
           .reduce(0, (a,b) -> a+b);
```

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

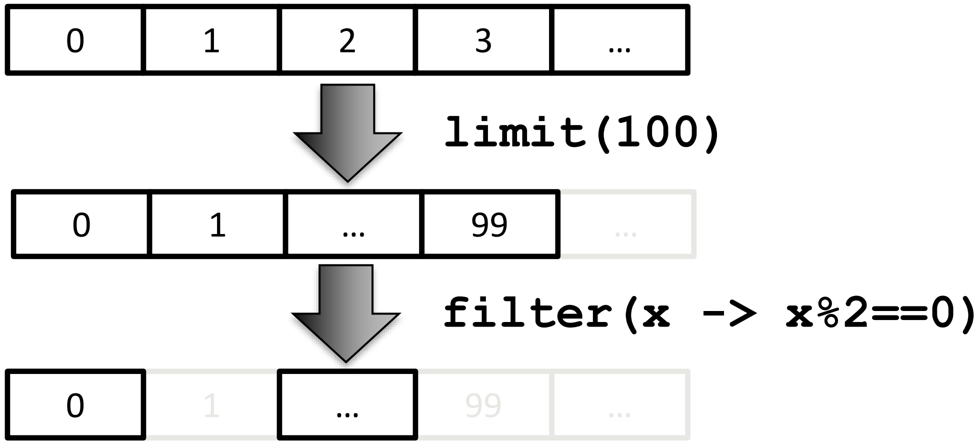
Example with everything so far

0	1	2	3	…

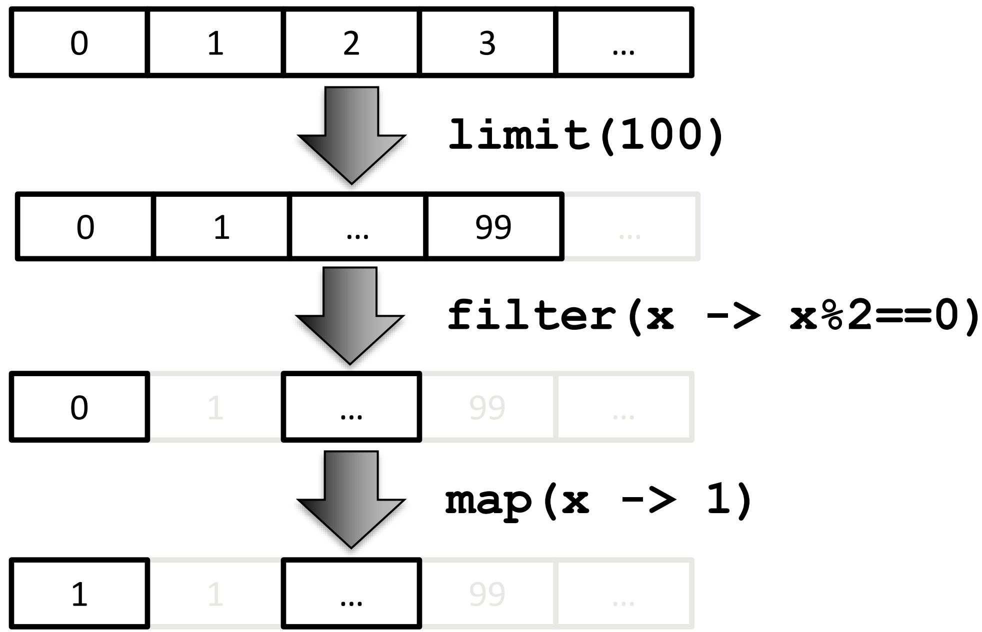
Example with everything so far



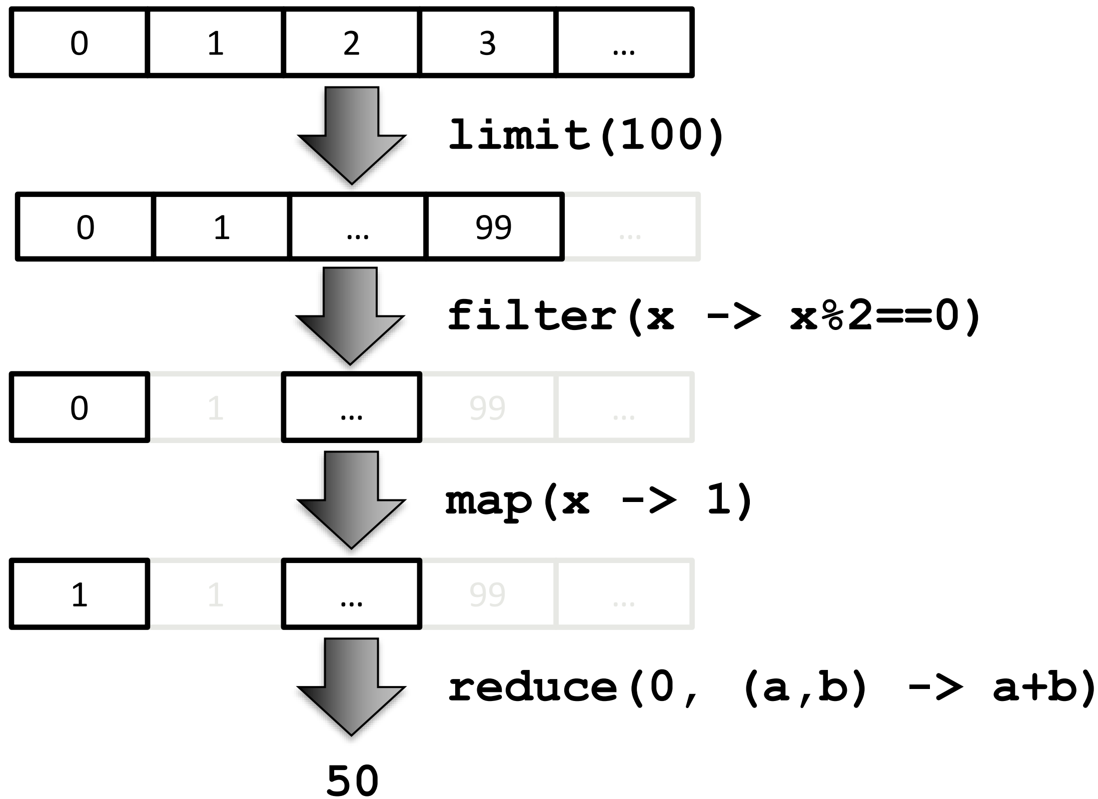
Example with everything so far



Example with everything so far



Example with everything so far



Terminal operations | 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());
```

See `StreamExample.java`

Terminal operations | 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());
```

Is this list symbolic like a stream or are values evaluated and stored in memory?

See `StreamExample.java`

Terminal operations | collect | groupingBy

- groupingBy is a special type of collector returning a Map where
  - Keys are generated based on some grouping criteria
  - Values are lists of elements (or operations on these lists of elements) of the stream matching the key

Terminal operations | collect | groupingBy

- Group employees by department

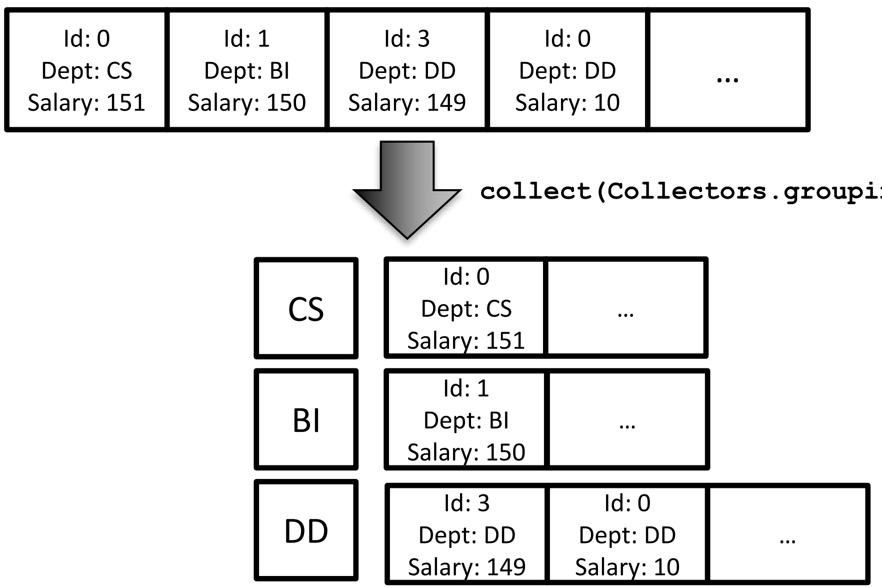
```
Map<String,List<Employee>> m = randomEmployees()
          .limit(50)
          .collect(Collectors.groupingBy(Employee::getDept));
```

Id: 0 Dept: CS Salary: 151	Id: 1 Dept: BI Salary: 150	Id: 3 Dept: DD Salary: 149	Id: 0 Dept: DD Salary: 10	...

Terminal operations | collect | groupingBy

- Group employees by department

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

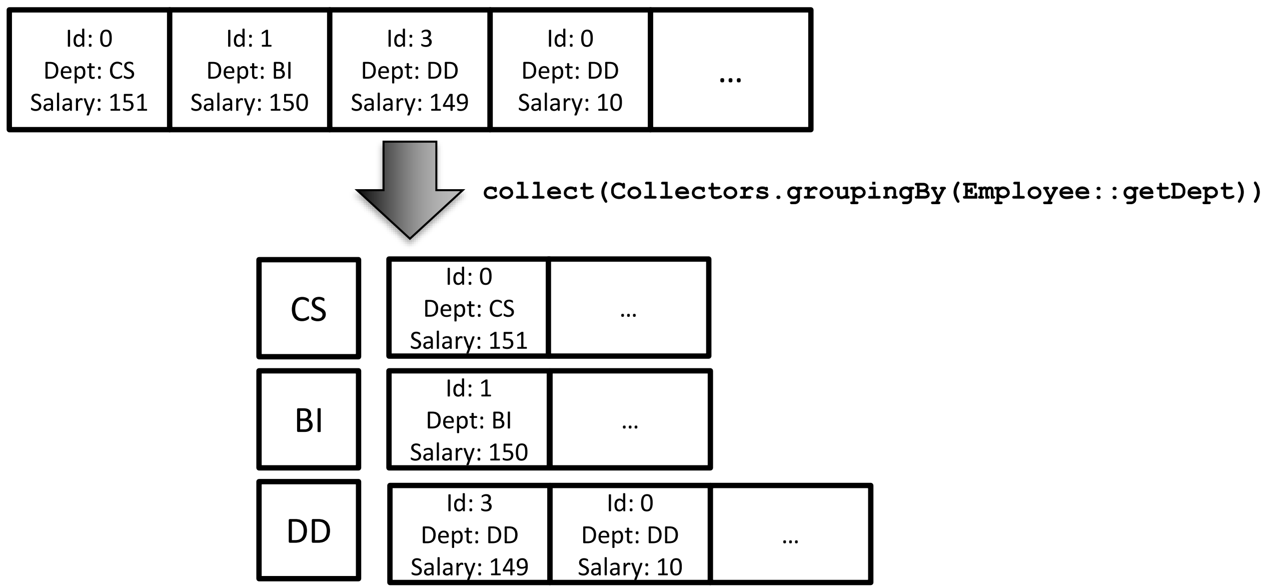


Terminal operations | collect | groupingBy

Is this map symbolic like a stream or are values evaluated and stored in memory?

- Group employees by department

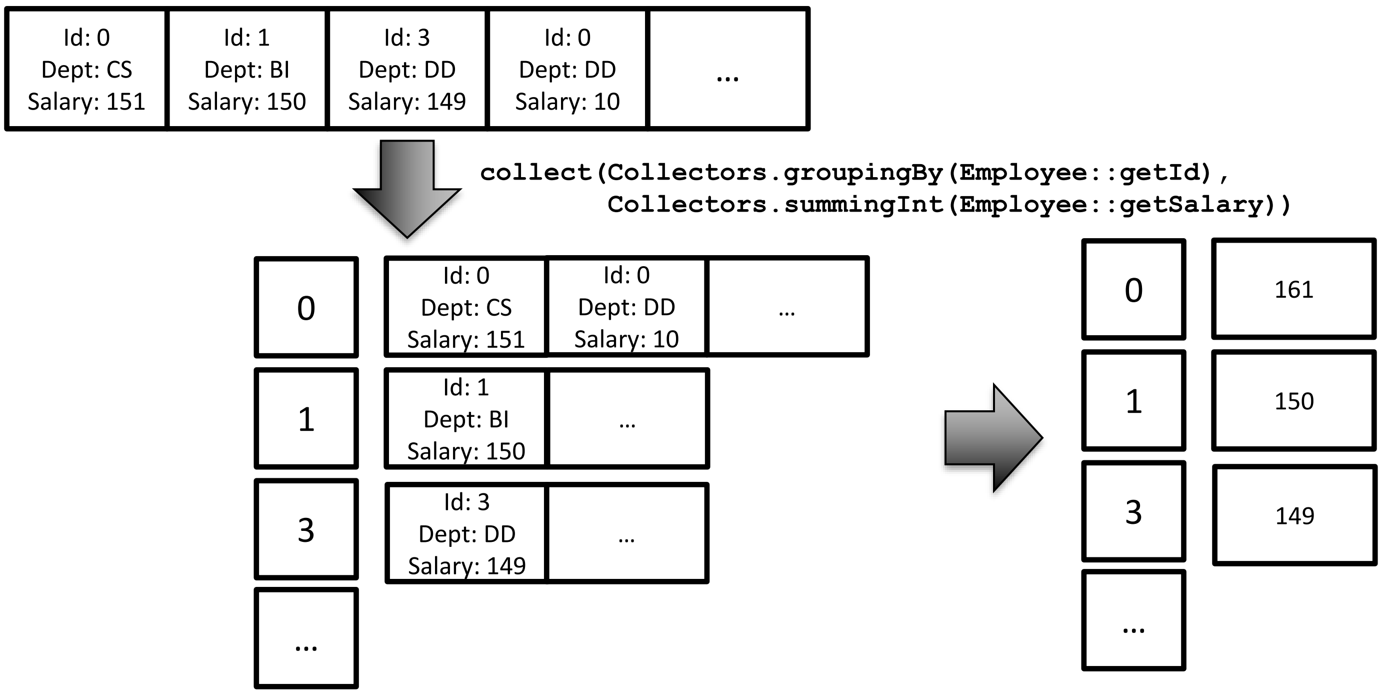
```
Map<String,List<Employee>> m = randomEmployees()
        .limit(50)
        .collect(Collectors.groupingBy(Employee::getDept));
```



Terminal operations | collect | groupingBy

- Get total salary per employee

```
randomEmployees().limit(50)
              .collect(Collectors.groupingBy(Employee::getId, Collectors.summingInt(Employee::getSalary)))
```



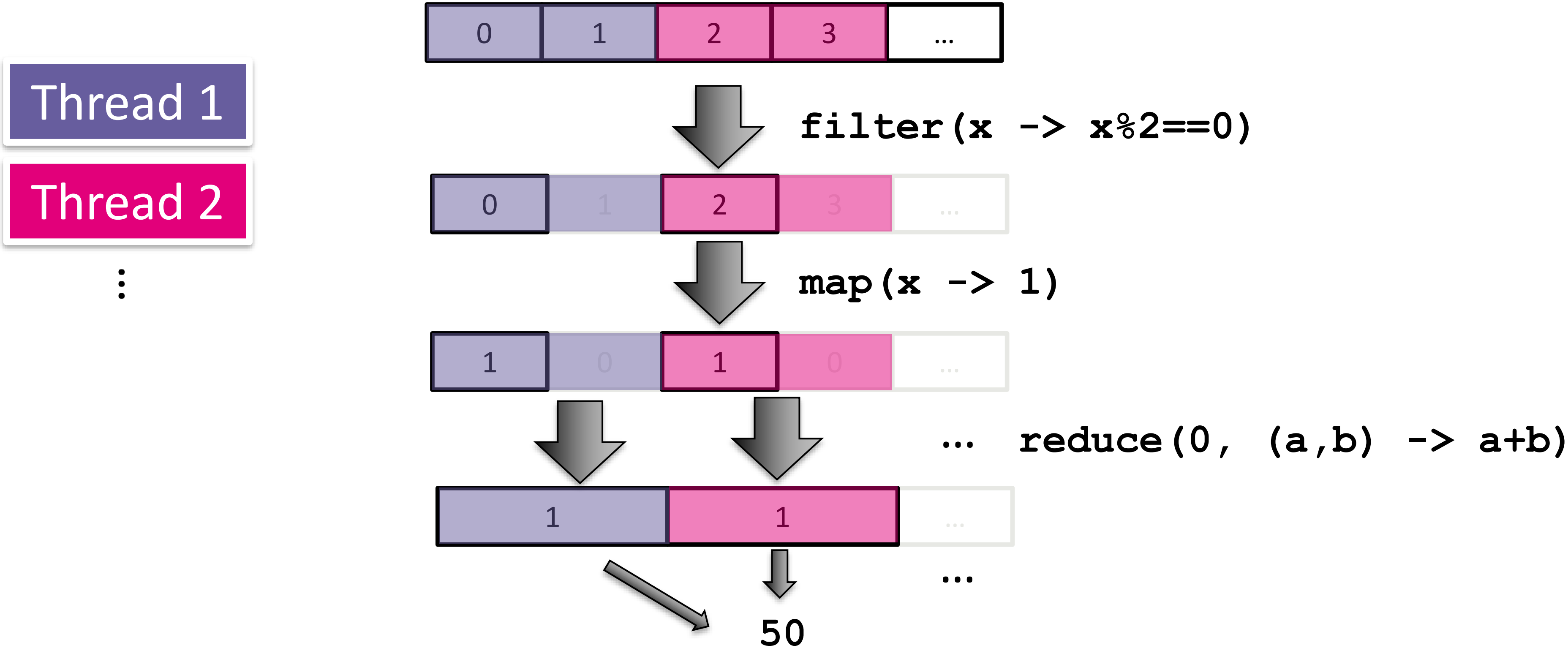
Parallelization of Streams

Java Parallel Streams

- You can create a parallel stream by calling
  - **`parallelStream()`** on, e.g., a collection, or
  - **`parallel()`** on a stream

Java Parallel Streams

- Parallelization of streams is very easy (remember the beginning of the lecture). Disjoint streams (from the original stream) are assigned to distinct threads from a thread pool (next lecture)



Java Parallel Streams | Processing order

- Since execution is parallel the processing of the stream is not guaranteed to in order

- For instance, run this program
  - `IntStream.range(0,10).parallel().forEach(System.out::println);`

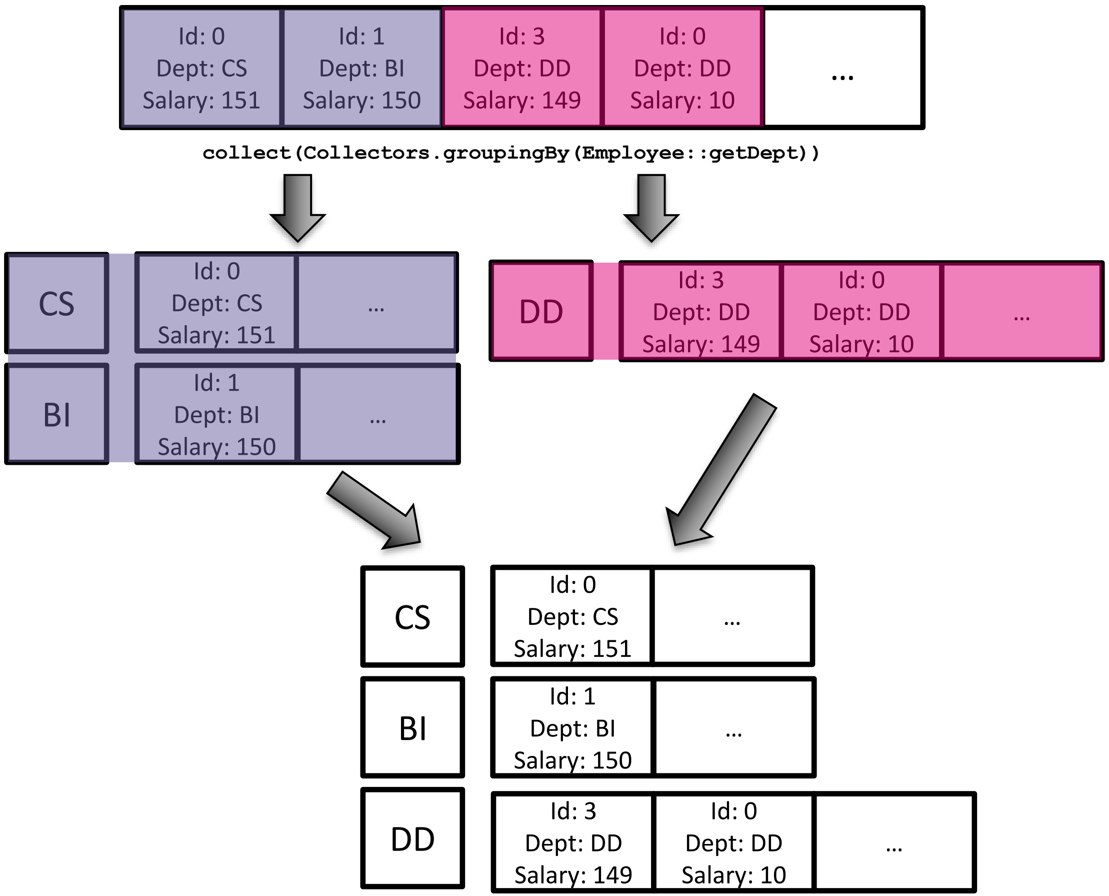
- In this case, it may be mitigated with **`forEachOrdered`**
  - `IntStream.range(0,10).parallel().forEachOrdered(System.out::println);`

Java Parallel Streams | Grouping

Would it be ok if parallel streams work concurrently on the Map resulting from a groupingBy operation?

Java Parallel Streams | Grouping

- In the local reduce operation, each thread creates locally its own “container” (collection or value) for the partial results
- In a final step, a single thread merges the local containers, so no race conditions arise
- However this step is very memory waste full, as it needs to create one container per partition of stream



Java Parallel Streams | Grouping

- A possible solution is to use **`groupingByConcurrent`**
- This function uses a ConcurrentHashMap that all threads can access concurrently.

```
collect(Collectors.groupingByConcurrent(Employee::getDept))
```

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 a 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.*

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:

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

Parallel Java 8 stream:

```
IntStream.range(0, range)
.parallel()
.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:

Pure functional programming ...

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

Parallel Java 8 stream:

```
IntStream.range(0, range)
.parallel()
.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:

Pure 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 0 ...99,999

Method	Intel i7 (ms)	AMD Opteron (ms)
Sequential for-loop	9.9	40.5
Sequential stream	9.9	40.8
Parallel stream	**2.8**	**1.7**

Functional streams give the simplest solution

Nearly as fast as tasks and threads, or faster:

- Intel i7 (4 cores) speed-up: 3.6 x
- AMD Opteron (32 cores) speed-up: 24.2 x

The future is parallel – and functional ☺

Agenda

- Data independence
- Lambda expressions
- Java Streams
- Parallel Java Streams