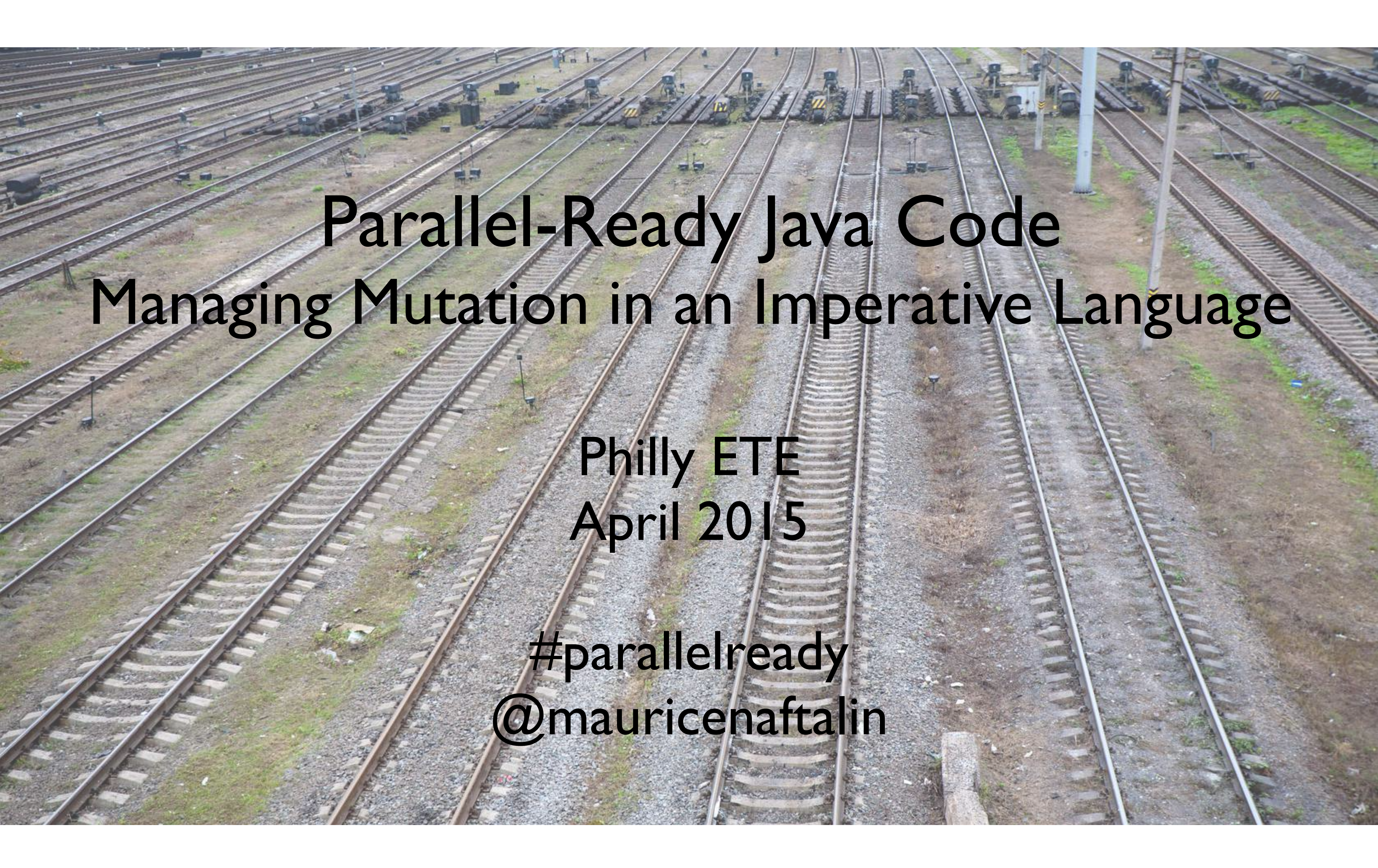
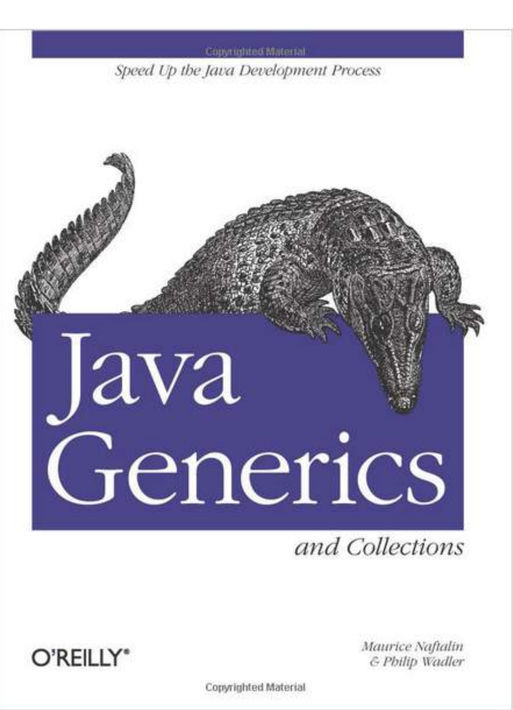

Parallel-Ready Java Code
Managing Mutation in an Imperative Language
Philly ETE
April 2015
#parallelready
@mauricenaftalin

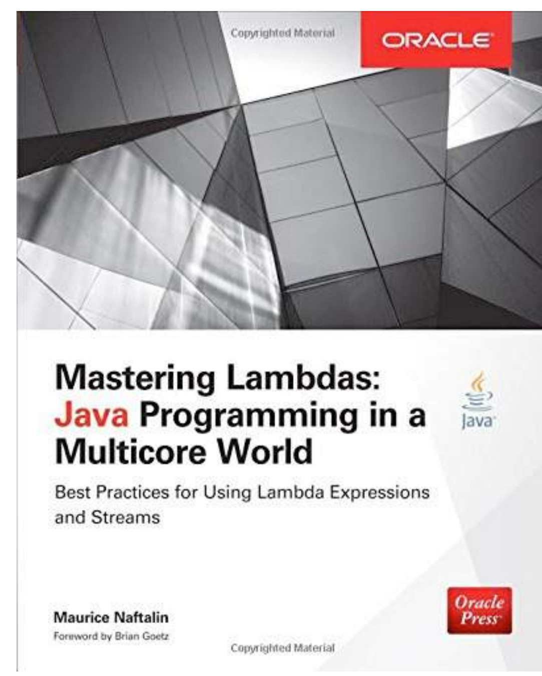
Maurice Naftalin

Developer, designer, architect, teacher, learner, writer

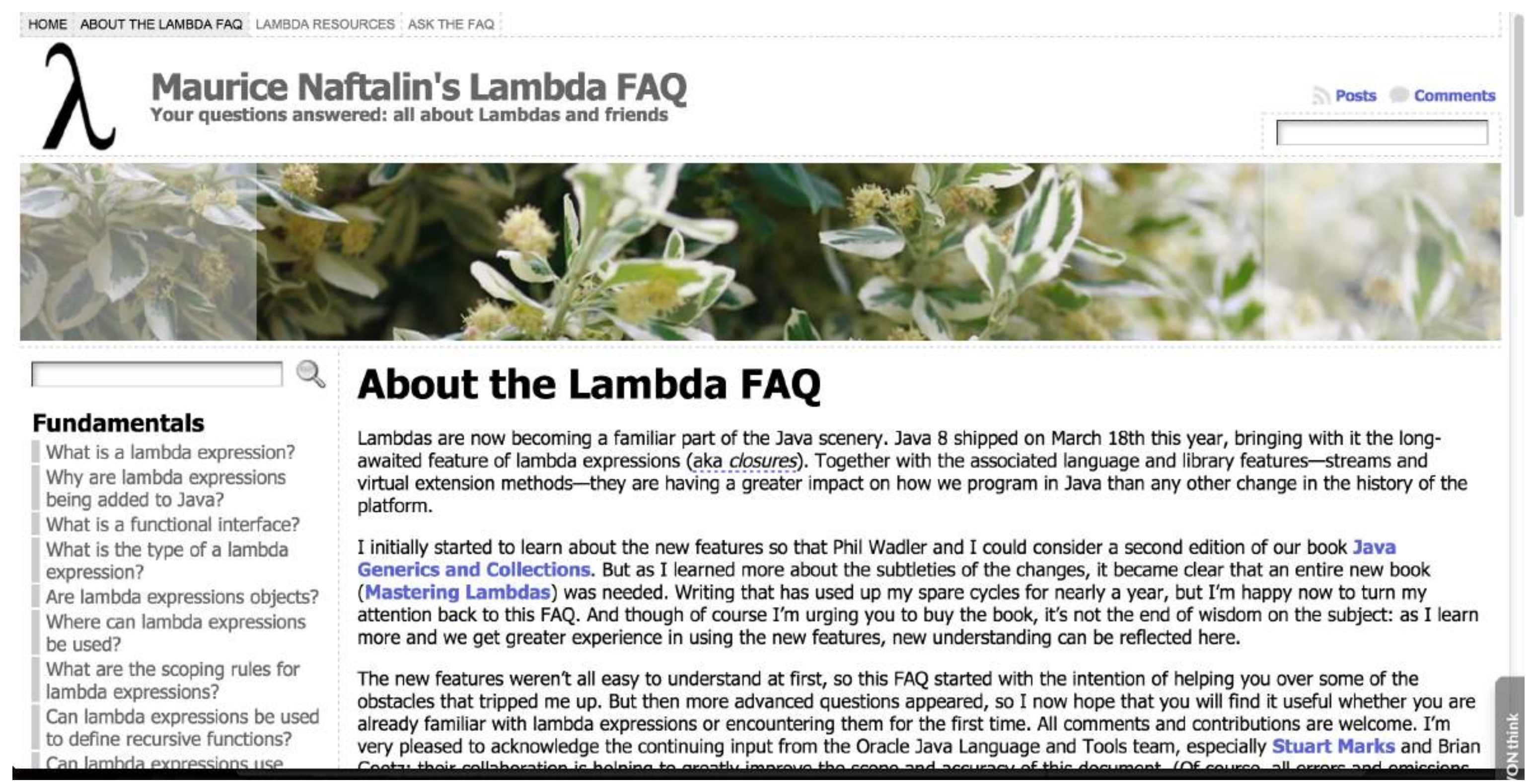
Maurice Naftalin

Repeat offender: Java 5 Java 8





The Lambda FAQ



www.lambdafaq.org

Ideas in the Stream API

– Pipes/Filters

– Parallelism

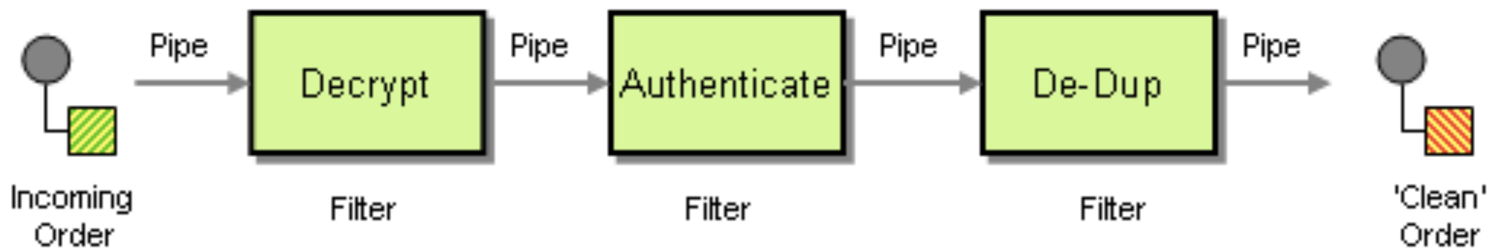
- Recursive decomposition
- Stream-transforming (aka intermediate) operations
- Parallel merging

Pipes and Filters

- Venerable Unix tool-building pattern:

```
ps -ef | grep login | cut -c 50- | head
```

- and in Enterprise Integration Patterns



Pipes and Filters

Advantages of this pattern

```
ps -ef | grep login | cut -c 50- | head
```

- no intermediate variables
- less (or no) intermediate storage
- lazy evaluation
- flexible tool-building:

Pipes and Filters

Doug McIlroy, inventor of the Unix pipe:

Write programs that do one thing well.

Write programs to work together.

~~*Write programs to handle text streams*~~

Example Domain

Person
city: City name: String age: int

```
Person amy = new Person(Athens, "Amy", 21);
...

List<Person> people = Arrays.asList(jon, amy, bill);
```

Pipes and Filters in Java

Is everyone an adult?

```
boolean allAdults = people.stream()
     .mapToInt(Person::getAge)
     .allMatch(a -> a >= 21);
```

Print the names of the inhabited cities

```
people.stream()
     .map(Person::getCity)
     .forEach(System.out::println);
```

Streams

Sequence of values

- Not a collection — may be partially evaluated or exhausted
- **Like an iterator**, yielding elements for processing
- Not like an iterator, **not associated with any storage mechanism**
- Sources: collections, arrays, generators, IO channels
- Can be
  - parallel
  - infinite
- Primitive specialisations: IntStream, LongStream, DoubleStream

Sources – sequential streams

Stream Sources:

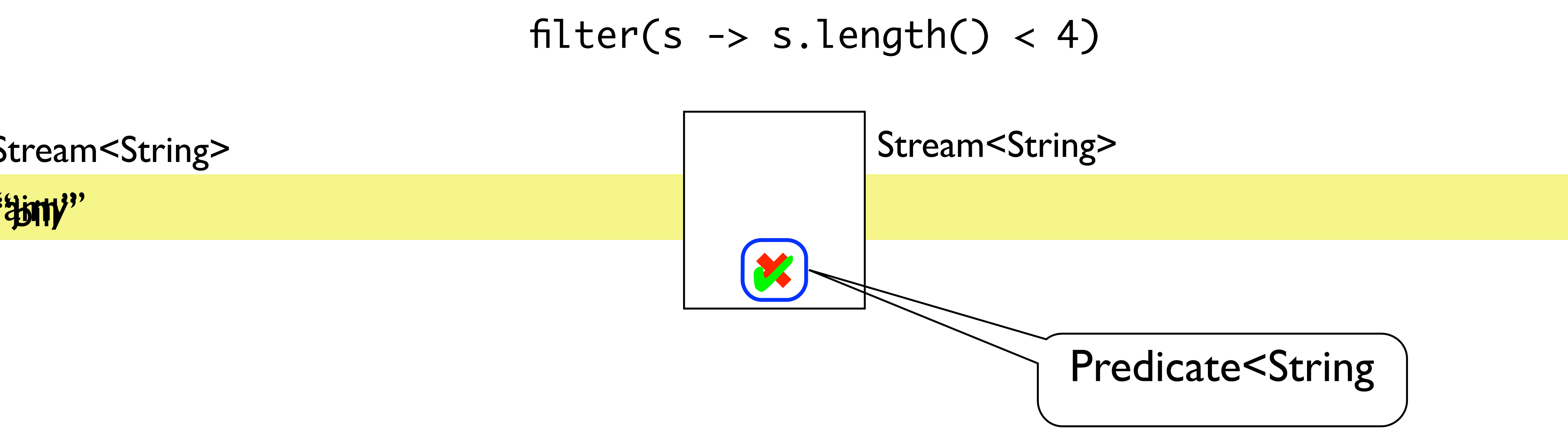
- supply elements one at a time
- implementations of `java.util.Spliterator`
- 40+ methods in platform classes
  - including `Collection.stream`

```
boolean allAdults = people.stream()
     .mapToInt(Person::getAge)
     .allMatch(a -> a >= 21);
```

Intermediate Operations

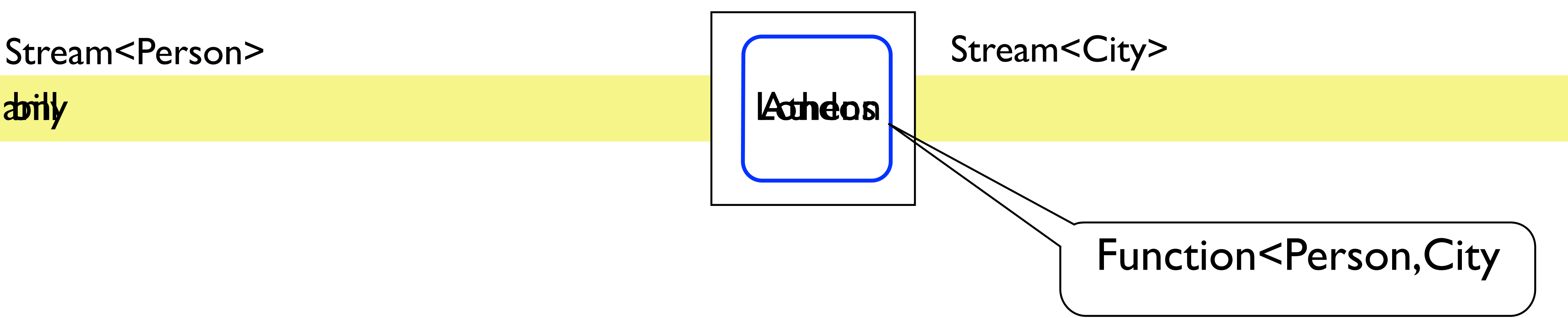
Filtering	`filter()`
Mapping	`map()`
One-to-Many Mapping	`flatMap()`
Debugging	`peek()`
Sorting/Deduplicating	`sorted(), distinct()`
Truncating	`skip(), limit()`

filter()



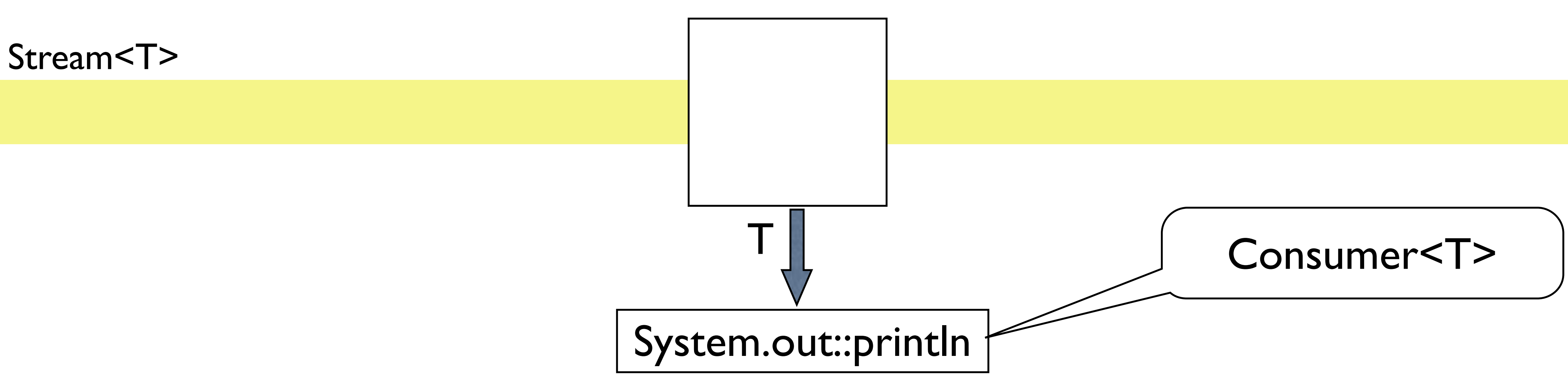
map()

map(Person::getCity)

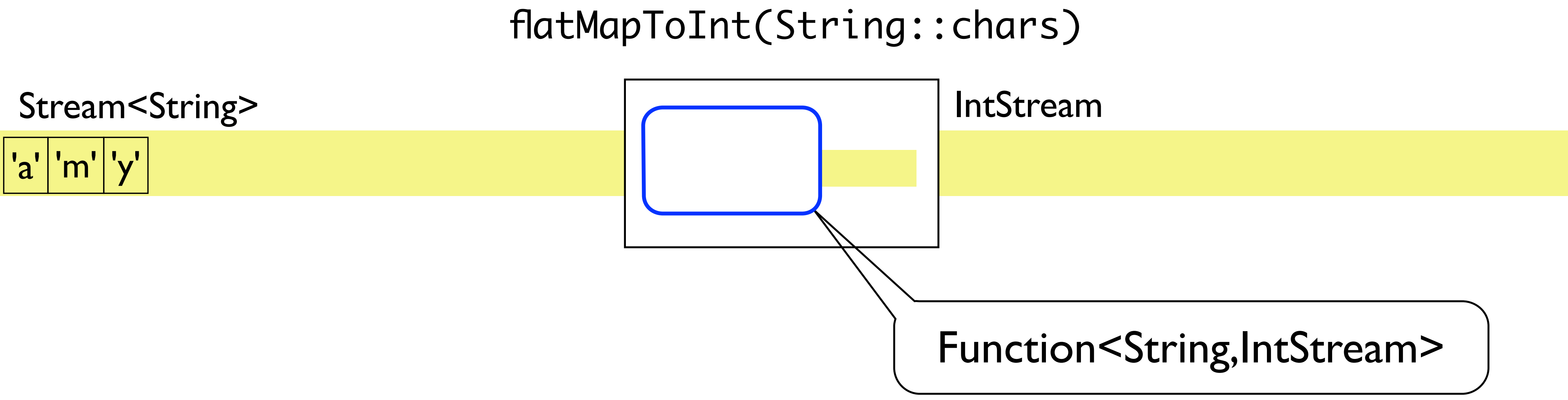


peek()

```
peek(System.out::println)
```



flatMapToInt()



Sinks – sequential streams

Stream Sinks (terminal operations):

Search	`findFirst(),findAny()` `anyMatch(), allMatch()`
Side-effecting	`forEach(), forEachOrdered()`
Reductions	...

Parallelism

"The best way to write parallel applications is not to have to think about parallelism."

Guy Steele

Automated Resource Management

Assemblers

Linkers

Register allocation

Heap management

Virtual memory

Parallelisation

Retooling

Some old habits won't work any more

minimising operation count (eg by reusing previously computed results)

- be prepared to "waste" operations to reduce communication

minimising space usage (by sharing data between operations)

- parallel programs trade space for reduced coupling

linear decomposition

- processing one thing at a time to accumulate results

No more accumulators!

Seriously, No More Accumulators??

No more accumulators!

Divide And Conquer algorithms instead

- symmetric merge operations

Major impact on terminal operations

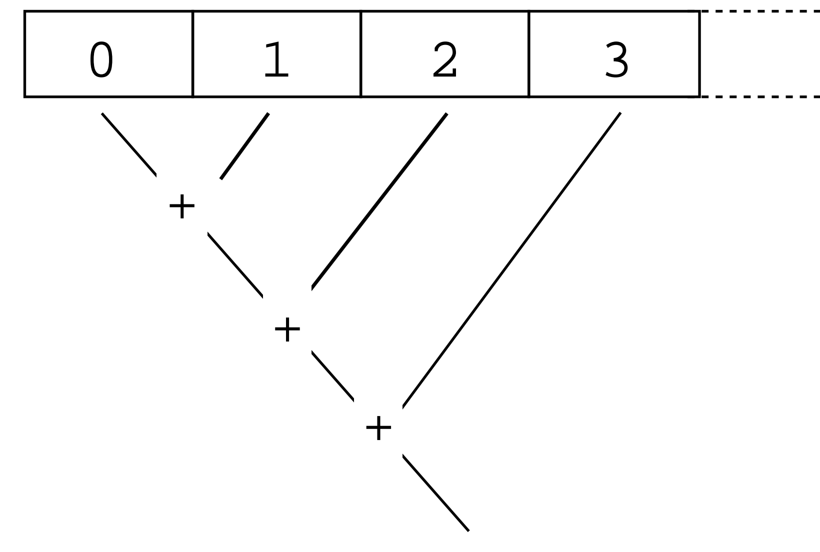
- almost all are accumulations of some kind

Simple Example – Summation

- Using an accumulator:

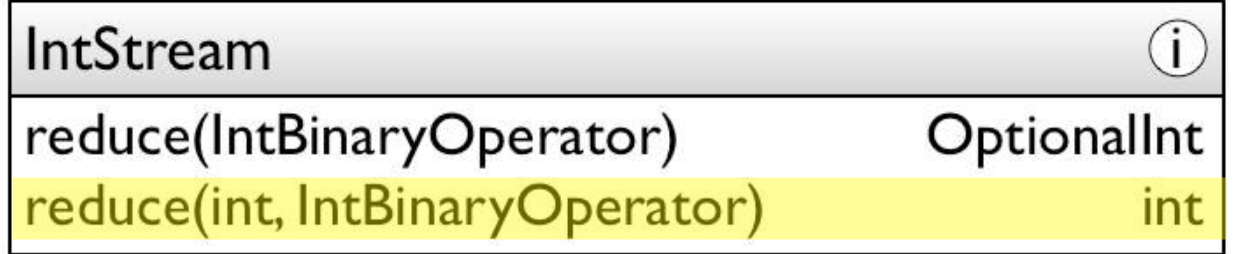
```
int[] vals = new int[100];
Arrays.setAll(vals, i -> i);

int sum = 0;
for (int i = 0 ; i < vals.length ; i++) {
    sum += vals[i];
}
```



Simple Example – Summation

- Avoiding an accumulator:



IntStream	ⓘ
reduce(IntBinaryOperator)	OptionalInt
reduce(int, IntBinaryOperator)	int

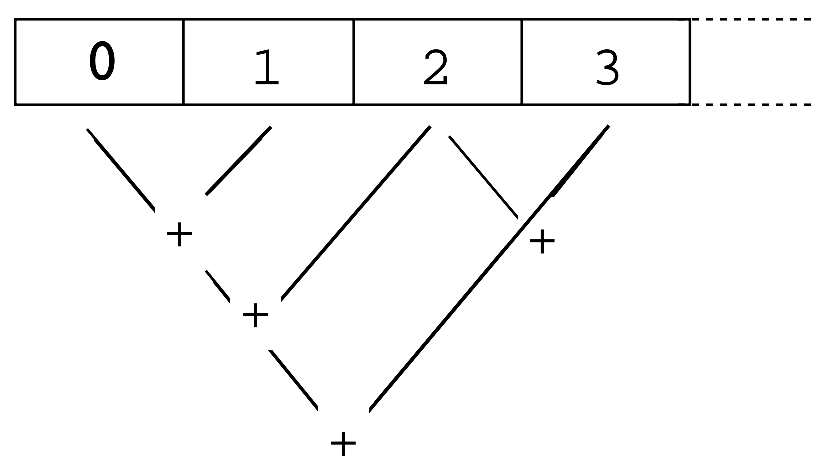
Simple Example – Summation

- Avoiding an accumulator:

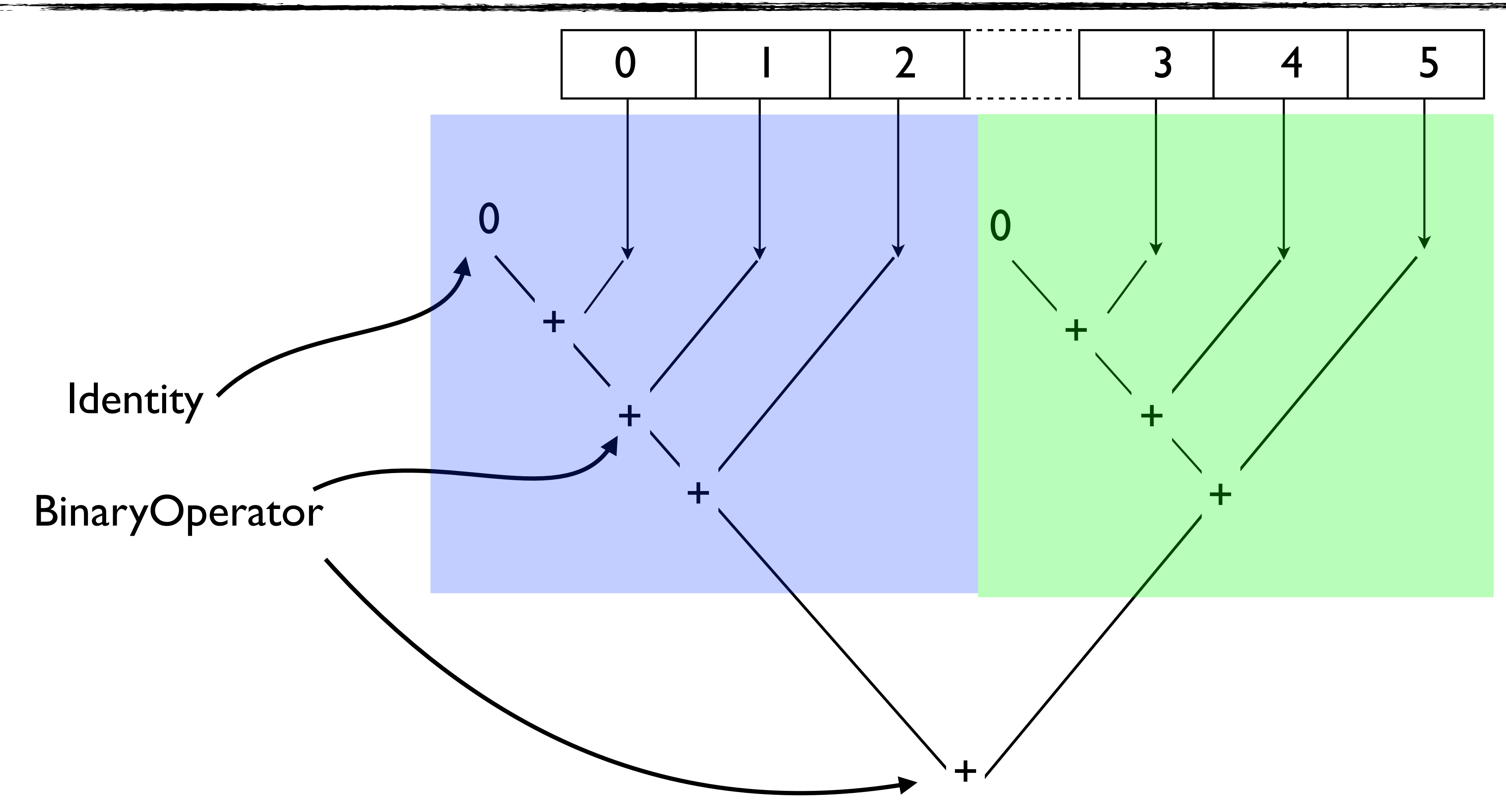
```
int[] vals = new int[100];
Arrays.setAll(vals, i -> i);

OptionalInt sum = Arrays.stream(vals)
    .reduce((a,b) -> a + b);
```

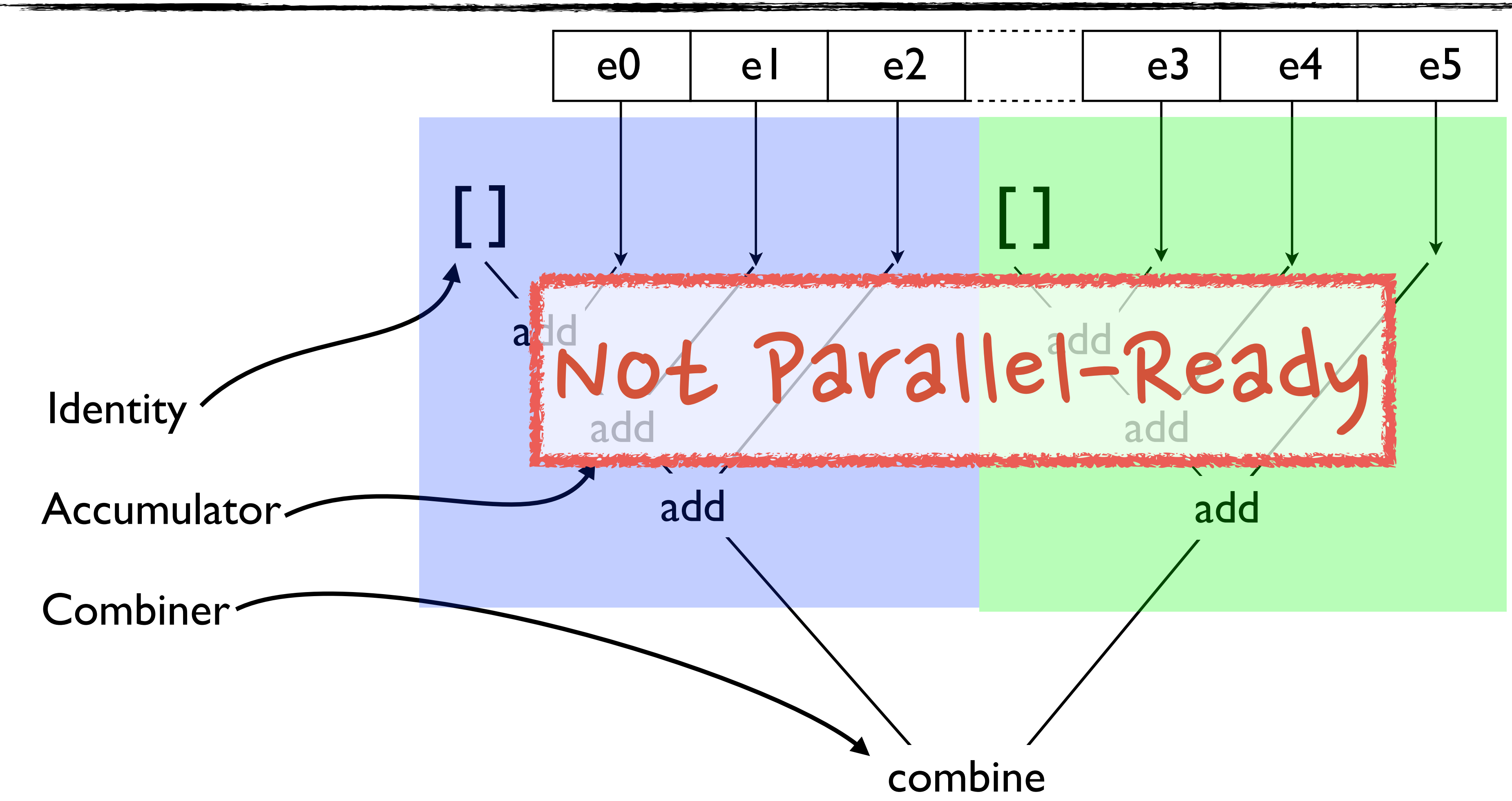
BinaryOperator must be associative!
a + (b + c) = (a + b) + c



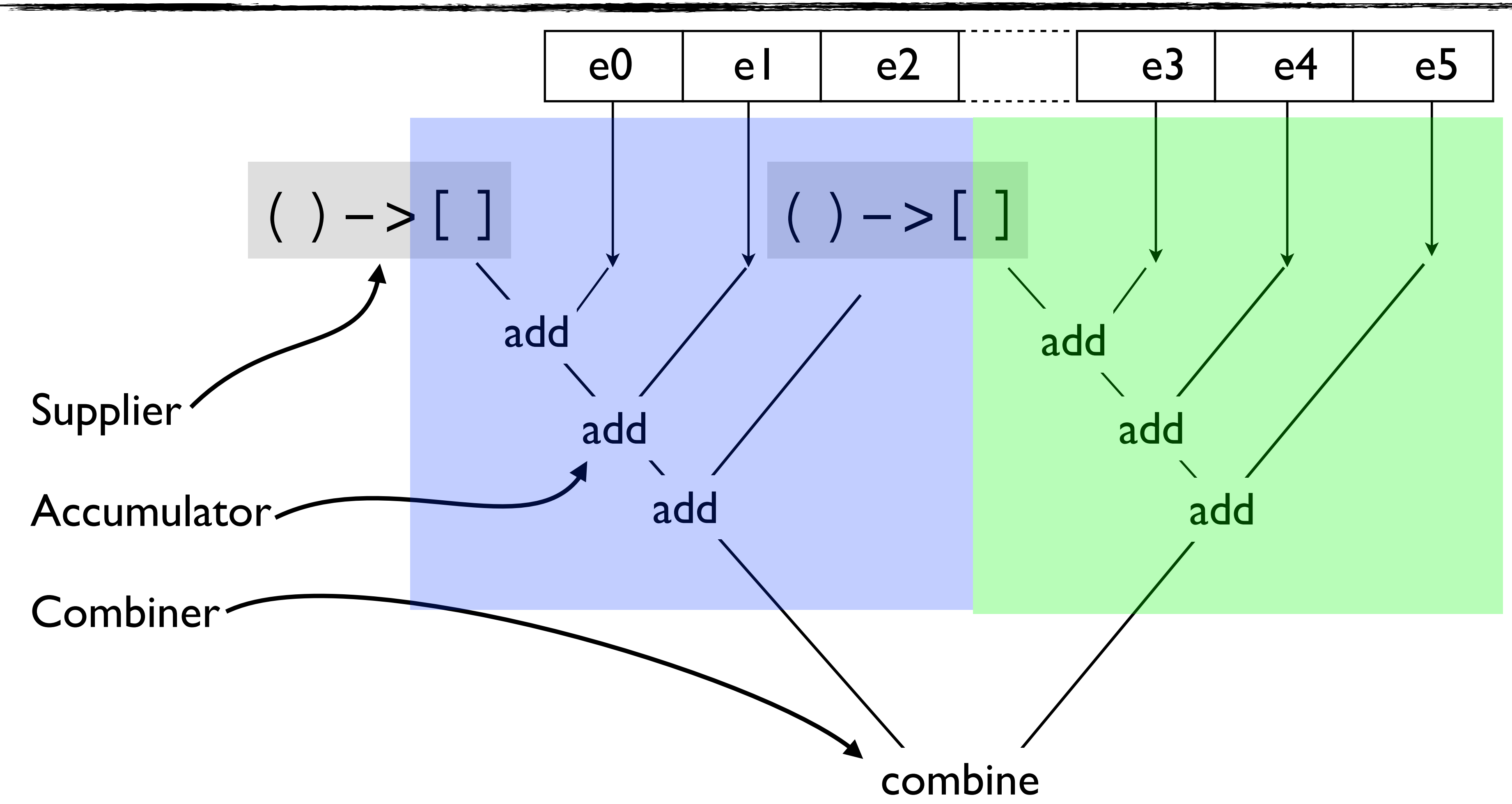
Reduction of Primitives



Reduction of Mutable Objects?



Collector



Using the Predefined Collectors

Predefined Standalone Collectors – from factory methods in `Collectors` class

- `toList(), toSet(), toMap(), joining()`
- `toMap(), toCollection()`
- `groupingBy(), partitioningBy()`

framework provides the Supplier

user provides the Supplier

produce a *classification map*

Simple Collector – toSet()

```
people.stream().collect(Collectors.toSet())
```

Stream<Person>

Set<Person>

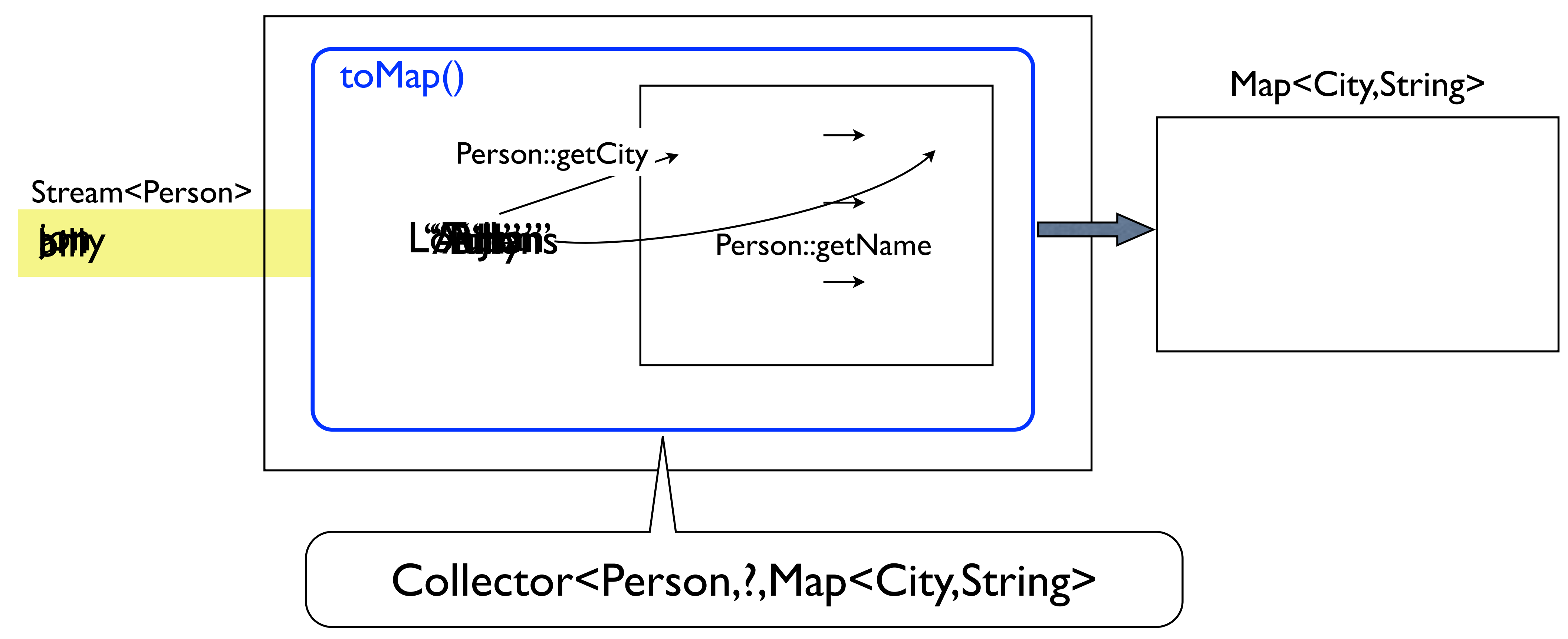
{ , , }

Collector<Person,?,Set<Person>>

toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)

```
people.stream().collect(
      Collectors.toMap(
            Person::getCity,
            Person::getName))
```

toMap(Function<T,K> keyMapper, Function<T,U> valueMapper)



groupingBy(Function<T,K> classifier)

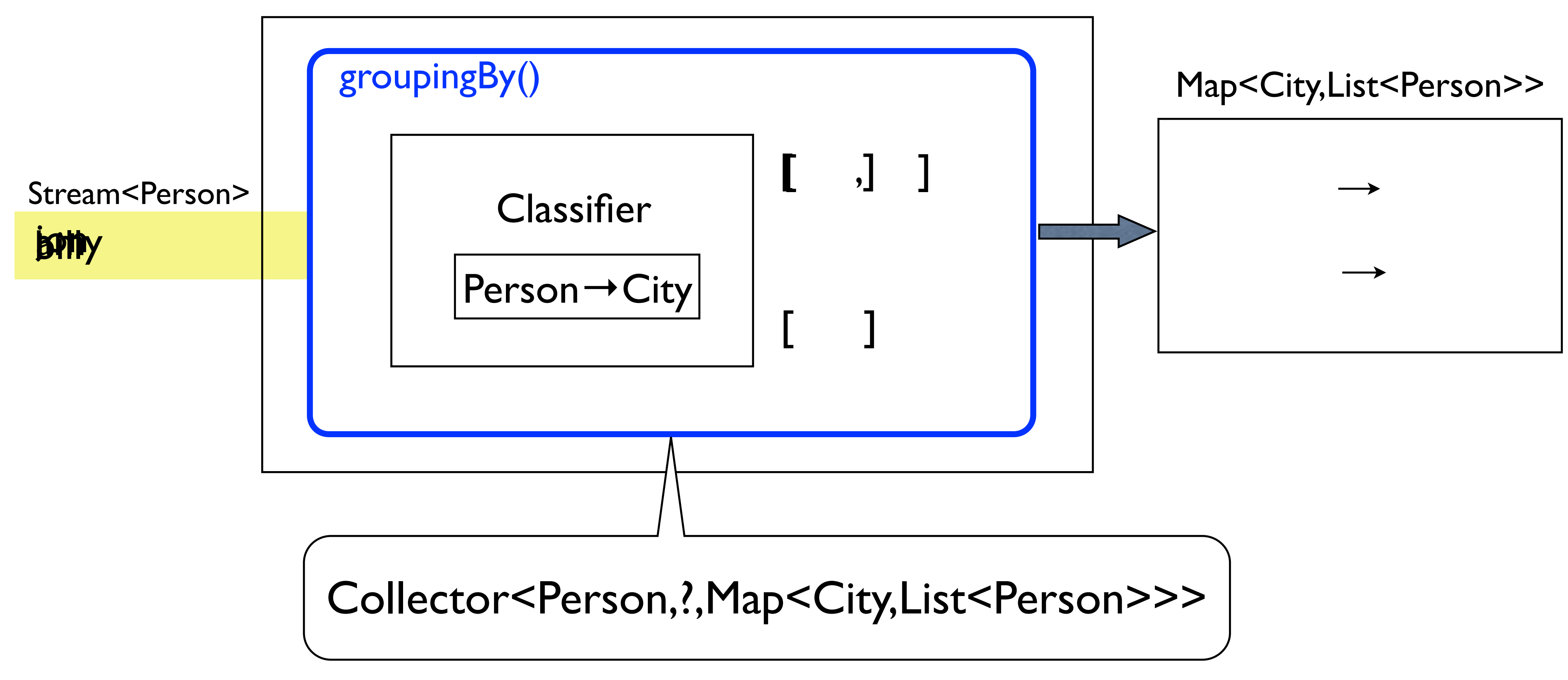
Uses the classifier function to make a *classification mapping*

Like toMap(), except that the values placed in the map are lists of the elements, one List corresponding to each classification key:

For example, use Person.getCity() to make a Map<City,List<Person>>

```
Map<City,List<Person>> peopleByCity = people.stream().
    collect(Collectors.groupingBy(Person::getCity));
```

groupingBy(Function<Person,City>)



Concurrent Collection

Thread safety is guaranteed by the framework

- Even for non-threadsafe containers!

But at a price... So what if your container is *already* threadsafe?

- a `concurrentMap` implementation?

Every overload of `toMap()` and `groupingBy()` has a dual

- toConcurrentMap(...)
- groupingByConcurrent(...)

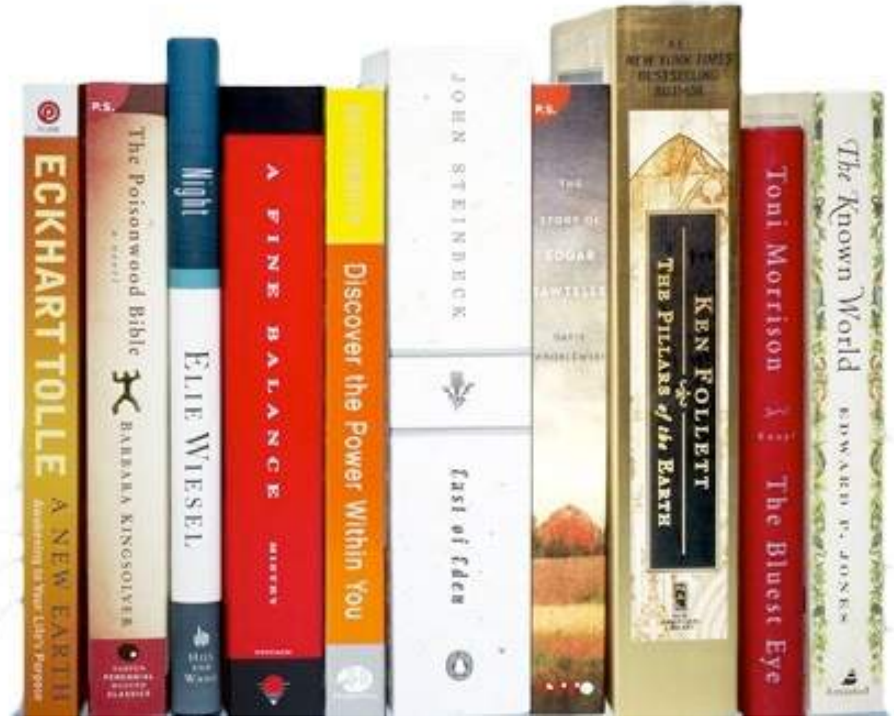
Writing a Collector

Why would you want to?

- accumulate to a container that doesn't implement `Collection`
- share state between values being collected
- to show how to parallelize "iterative-looking" problems

Oprah’s Problem

How to find a book?

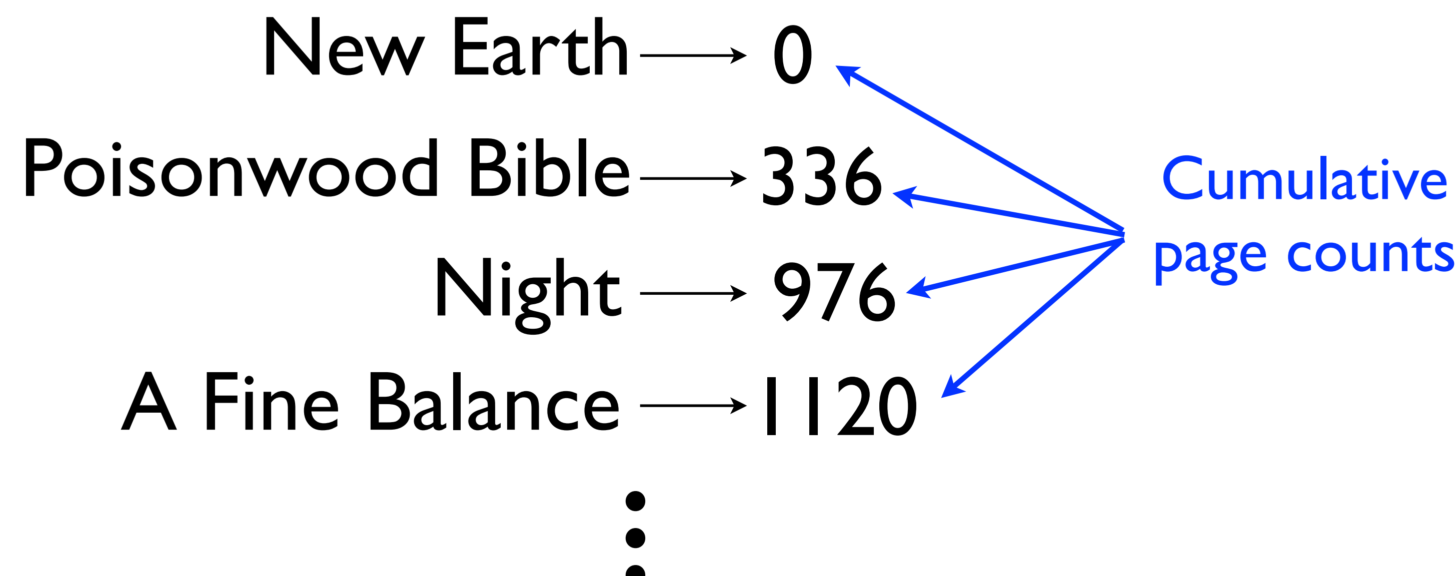


Page count →

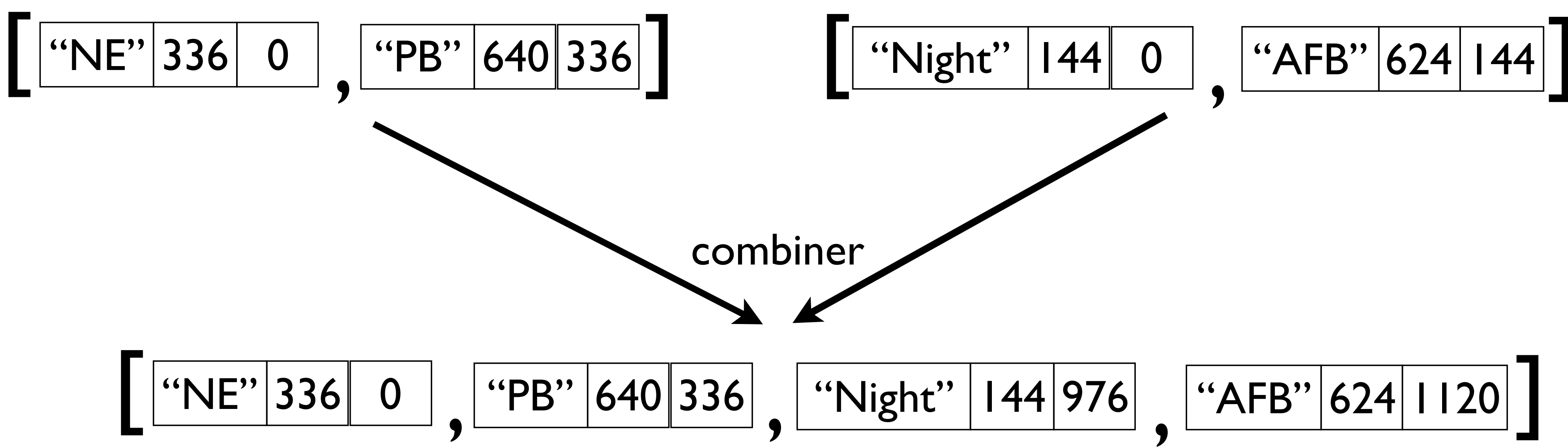
336 144 239 384 172

640 624 640 1104 400

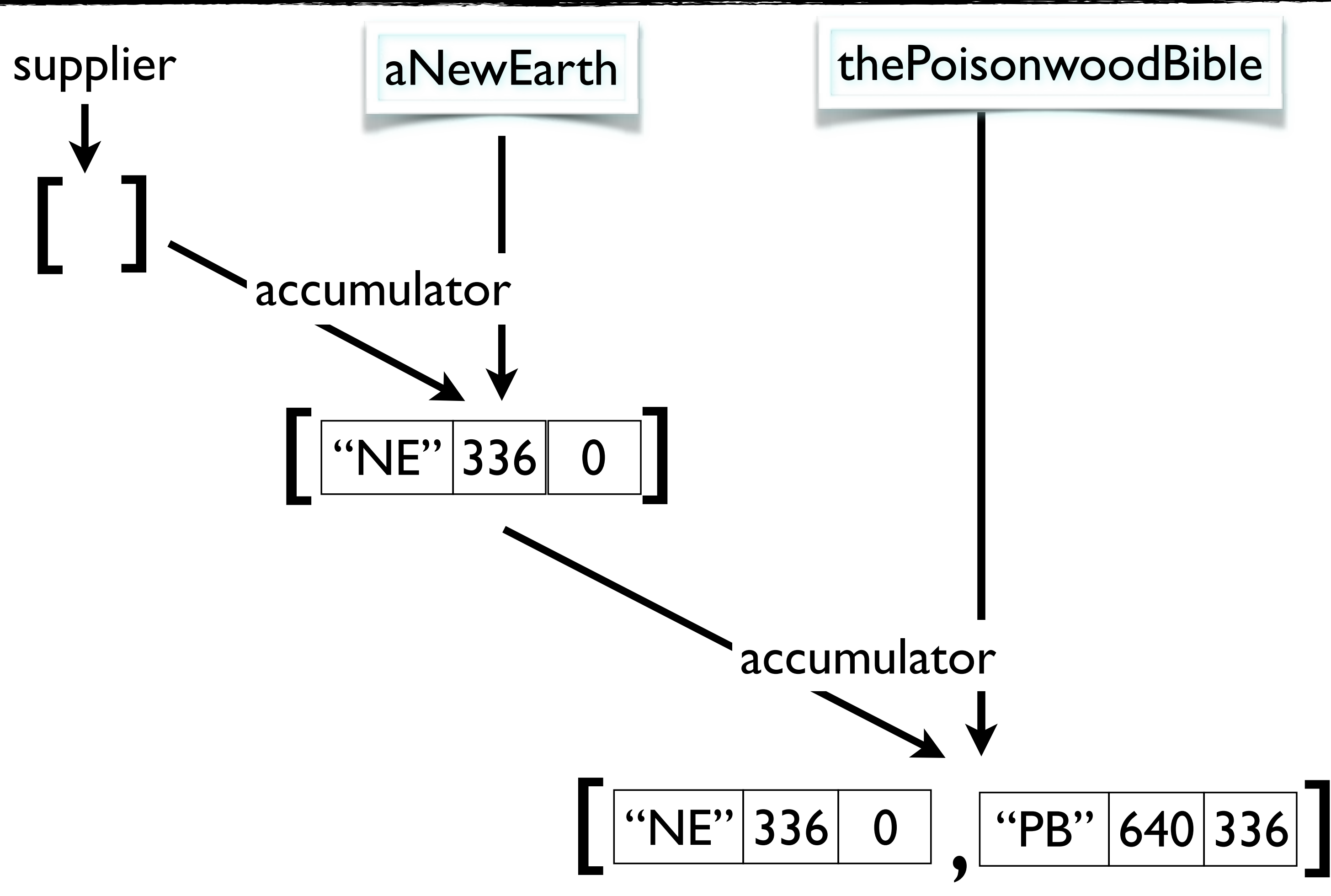
What Oprah Needs



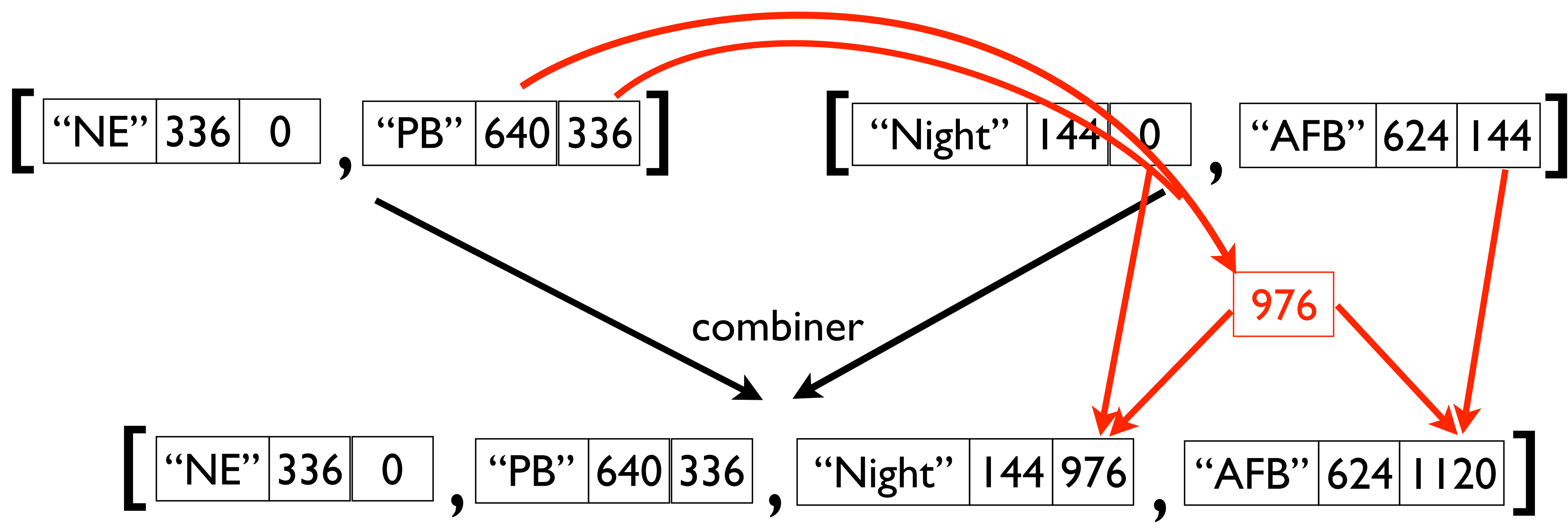
Creating A Partial Solution



Supplier and Accumulator

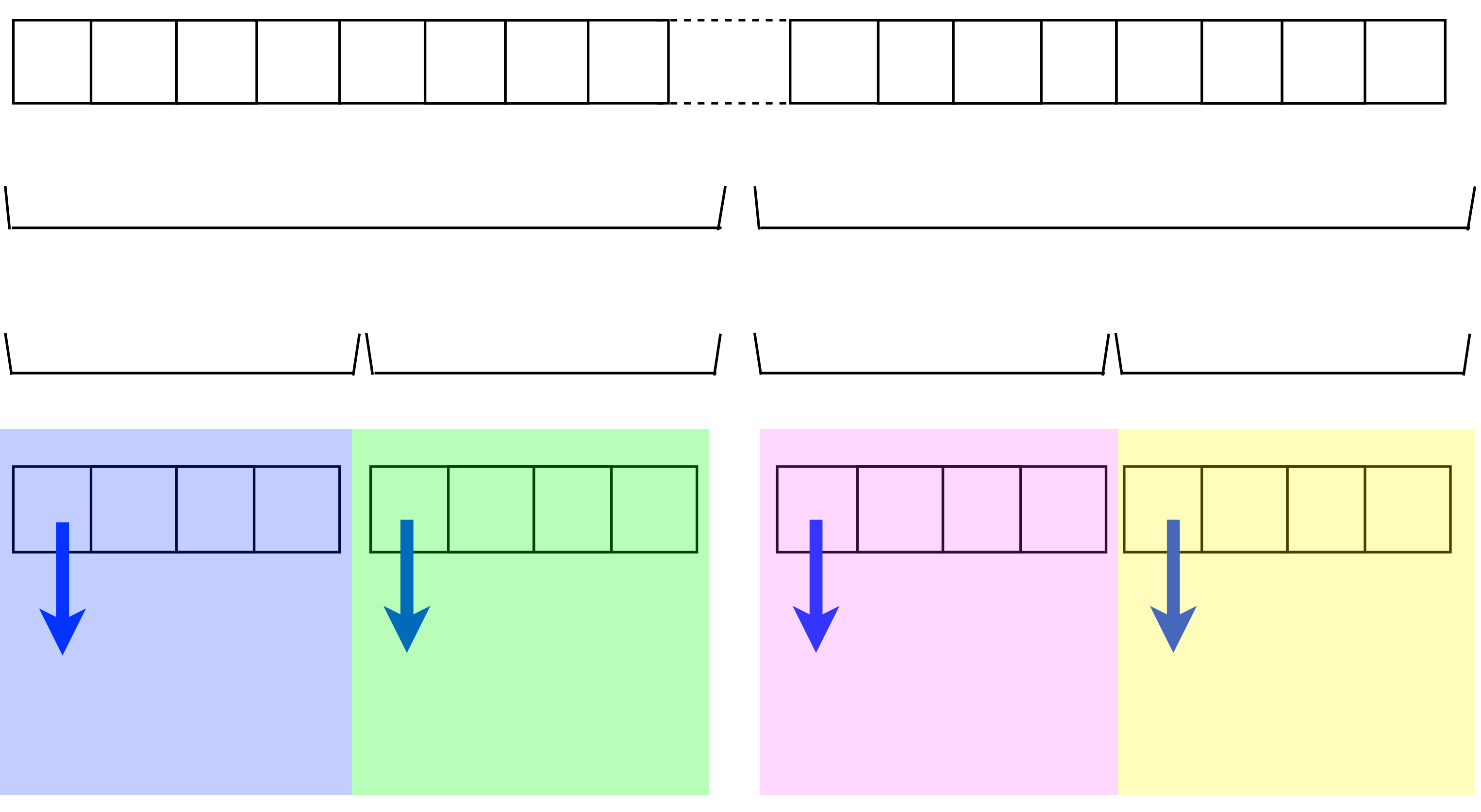


Combiner



Sources – parallel streams

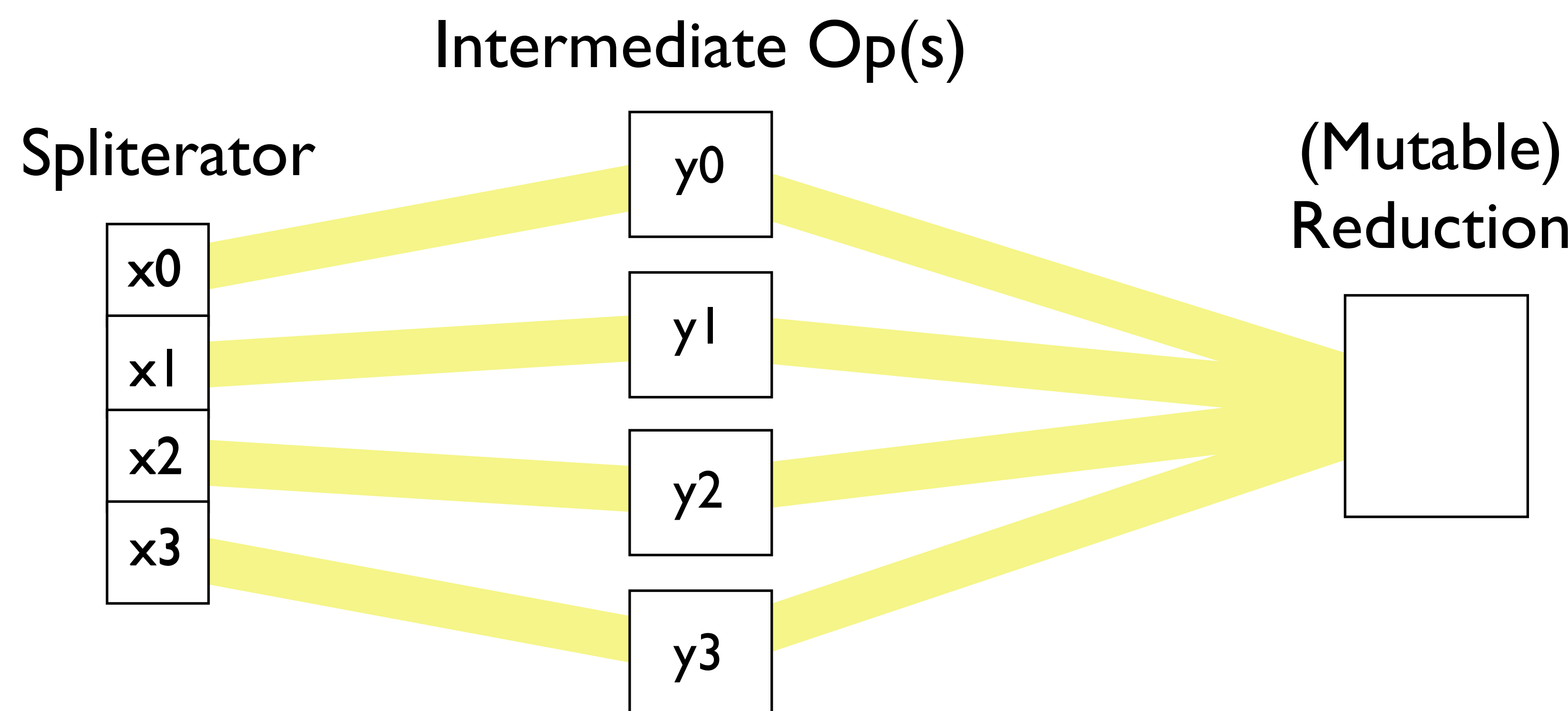
Implemented by a `Spliterator`



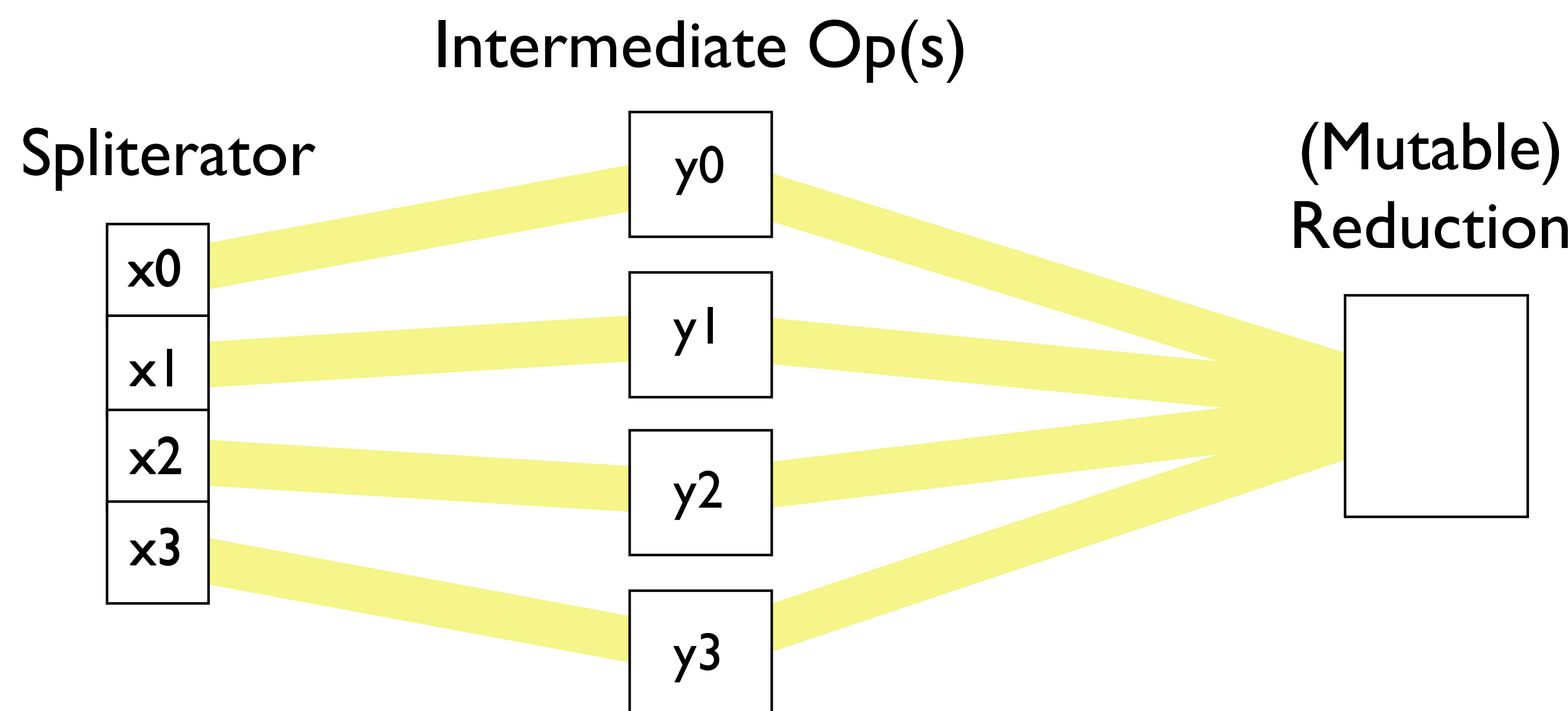
#parallelready

@mauricenaftalin

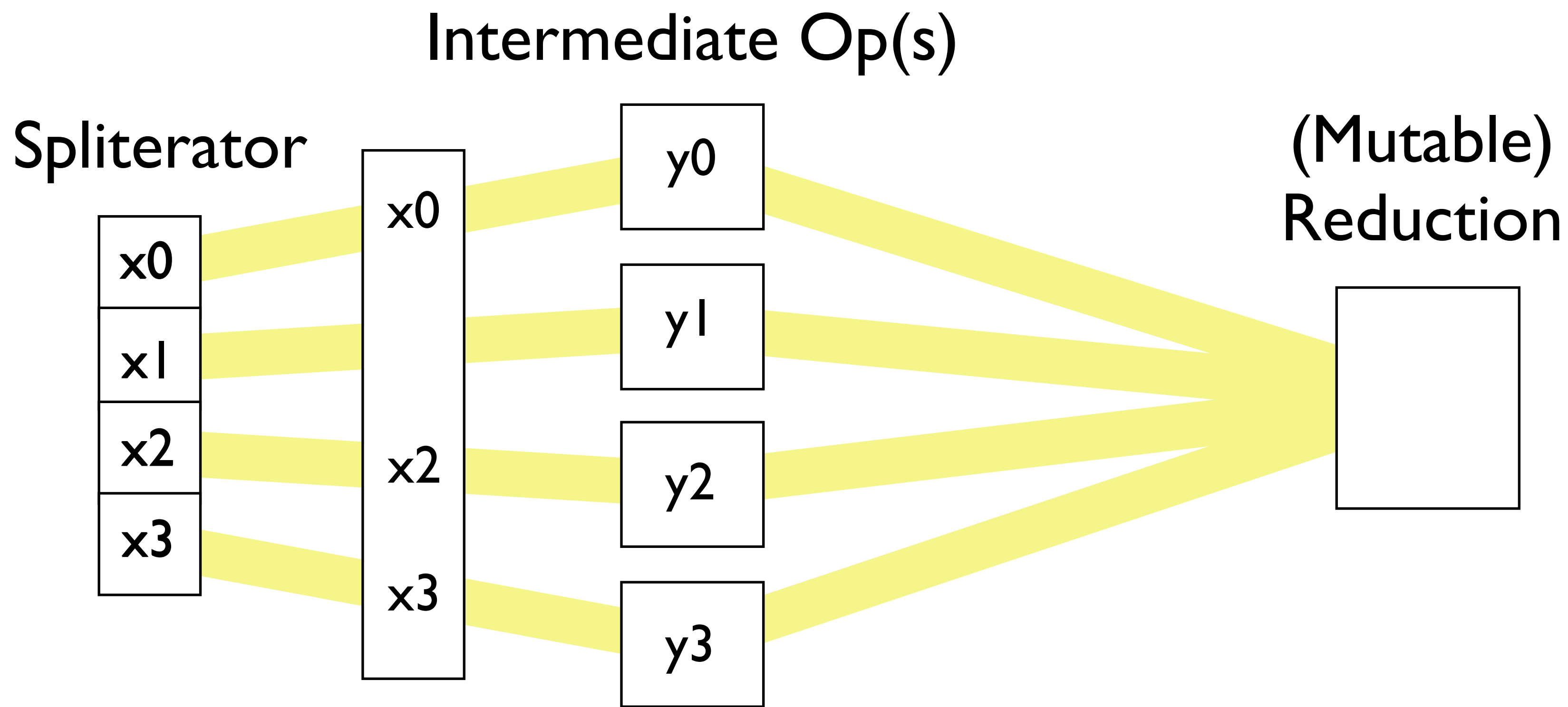
Visualizing Stream Operations



Visualizing Stream Operations



Visualizing Stream Operations



peek(System.out::print)

What Does Parallel-Ready Mean?

- sequential execution is no longer *the* default
- it’s still often more efficient, but that will change
- code must be agnostic about its execution mode
- operations in the Stream API have equivalent effect
  - sequential or parallel
  - may be equally non-deterministic
- *decouples* execution mode from functional behavior
- parallel-ready is not only about parallel execution!
  - or even mainly, for now

So, Is Your Code Parallel-Ready?

Most unlikely. But it can be!

The Stream API provides a framework and a lot of support for writing PR code. You just need to co-operate with it.

- it depends on you behaving sanely

Using Streams Safely

Rules in the JavaDoc for `java.util.stream`

Stateless behavior:

- avoid stateful behavioral parameters
  - “one whose result depends on any state which might change during the execution of the stream pipeline”

…

Using Streams Safely

Rules in the JavaDoc for `java.util.stream`

...

Non-interference

- don’t modify a stream’s data source during execution
  - you may if it’s a thread-safe class

Don’t rely on side-effects

- use for debugging only

Conclusion

The Streams API offers a framework for

- programming bulk data operations in parallel-ready style
- easy parallellism, with thread-safe access to non-threadsafe containers

All we have to do is co-operate with it!

Parallel-Ready code starts today!

Resources

www.lambdafaq.org

www.masteringlambdas.org