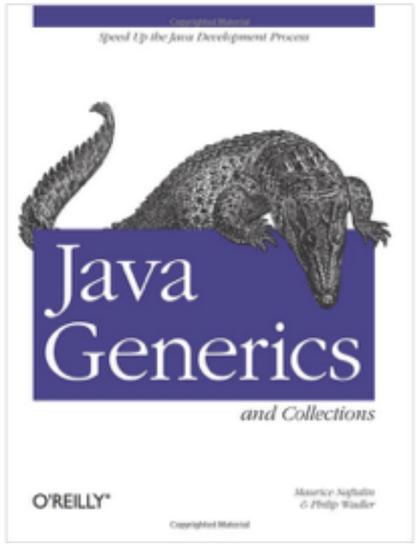
Journey's End

Collection and Reduction in the Stream API
JavaOne, October 2015

Maurice Naftalin

Developer, designer, architect, teacher, learner, writer



Co-author

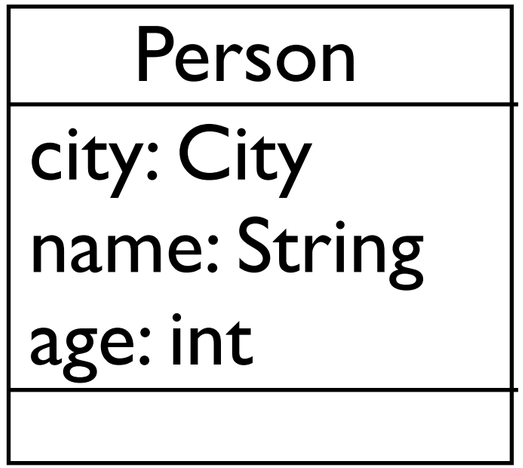
www.lambdafaq.org

Current Projects

Journey's End – Agenda

- Why collectors?
- Using the predefined collectors
  - Intermission!
- Worked problems
- Writing your own

Example Domain

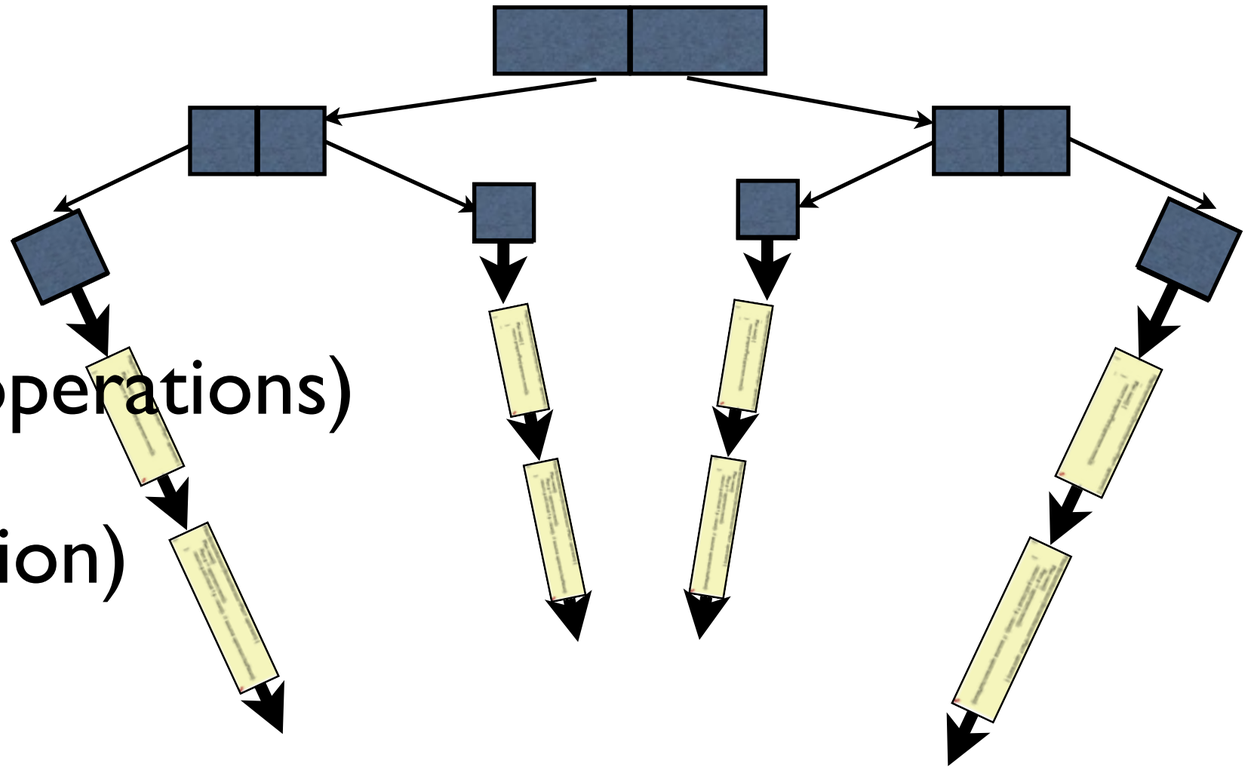



```
Person amy = new Person(Athens, "Amy", 21);
...

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

Collectors: Journey's End for a Stream

The Life of a Stream Element

- Born (at a spliterator)
- Transformed (by intermediate operations)
- Collected (by a terminal operation)



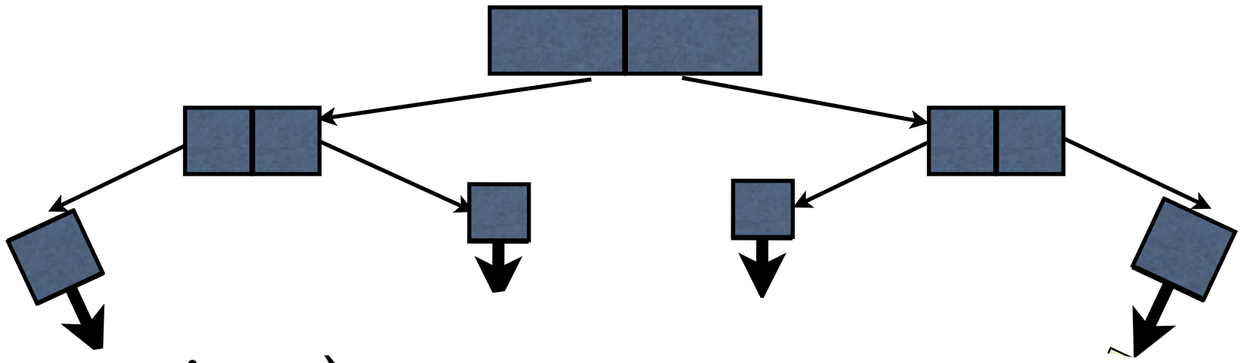
Collectors: Journey's End for a Stream

The Life of a Stream Element

- Born (at a spliterator)
- Transformed (by intermediate operations)
- Collected (by a terminal operation)

Collectors: Journey's End for a Stream

The Life of a Stream Element

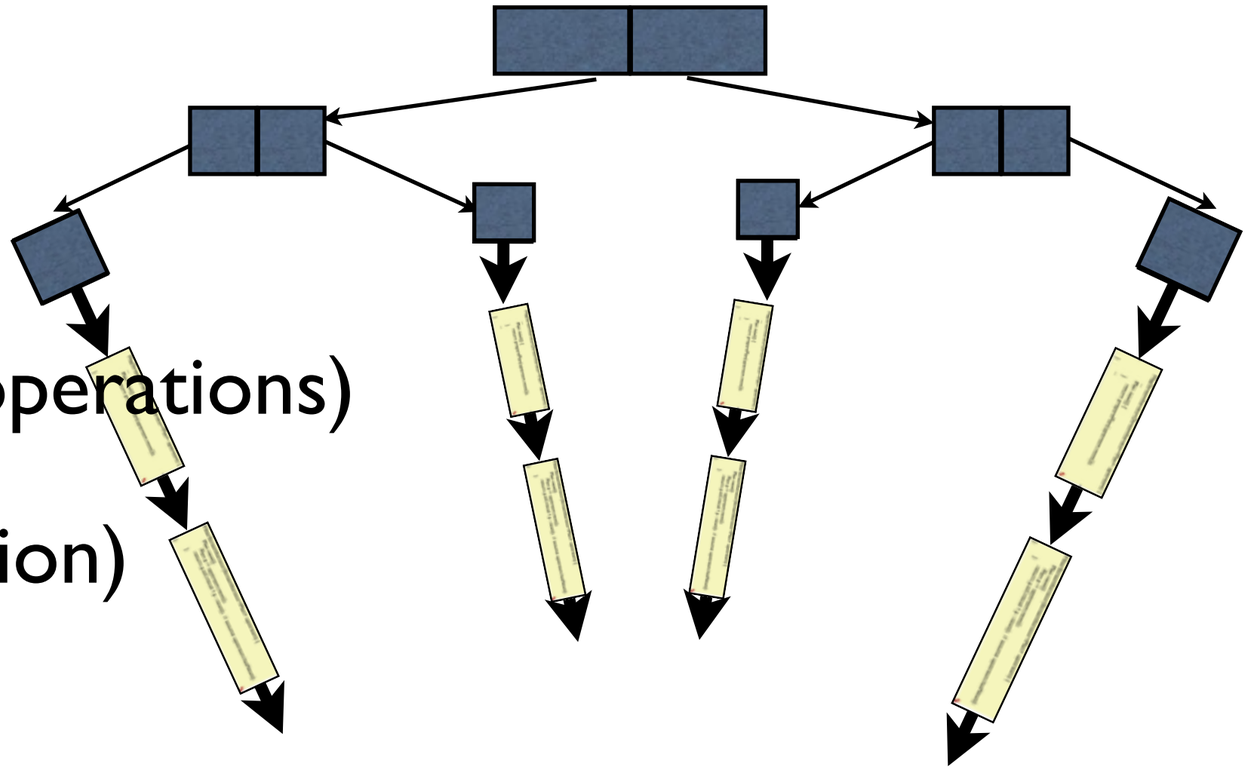


- Born (at a spliterator)
- Transformed (by intermediate operations)
- Collected (by a terminal operation)

Collectors: Journey's End for a Stream

The Life of a Stream Element

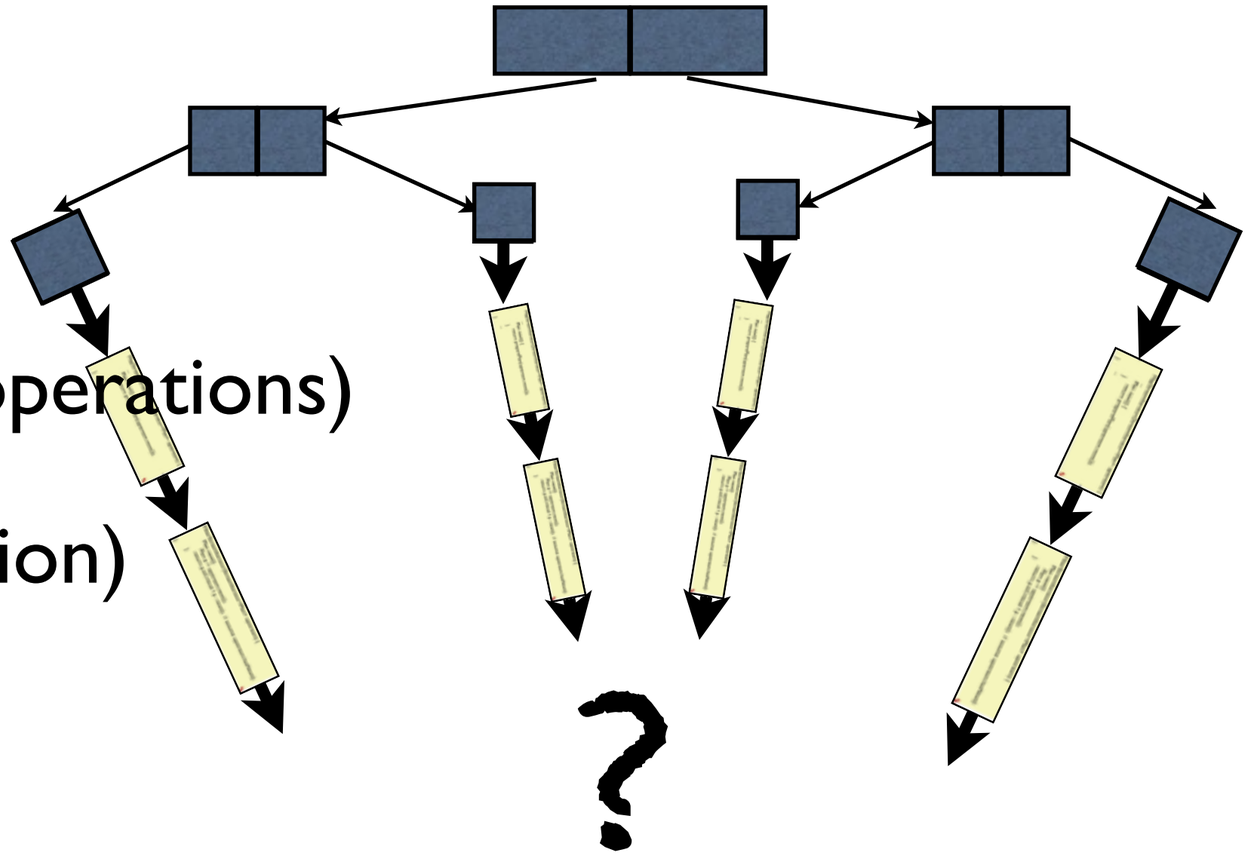
- Born (at a spliterator)
- Transformed (by intermediate operations)
- Collected (by a terminal operation)



Collectors: Journey's End for a Stream

The Life of a Stream Element

- Born (at a spliterator)
- Transformed (by intermediate operations)
- Collected (by a terminal operation)



Terminal Operations

– Search operations

– Side-effecting operations

– Reductions

Search Operations

Search operations -

- allMatch, anyMatch
- findAny, findFirst

```
boolean allAdults =
    people.stream()
        .allMatch(p -> p.getAge() >= 21);
```

Side-effecting Operations

Side-effecting operations

- forEach, forEachOrdered

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

- So could we calculate total ages like this?

```
int sum = 0;
people.stream()
      .mapToInt(Person::getAge)
      .forEach(a -> { sum += a; });
```

Side-effecting Operations

Side-effecting operations

- forEach, forEachOrdered

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

- So could we calculate total ages like this?

```
int sum = 0;
people.stream()
      .mapToInt(Person::getAge)
      .forEach(a -> { sum += a; });
```

Don't do this!

Reduction – Why?

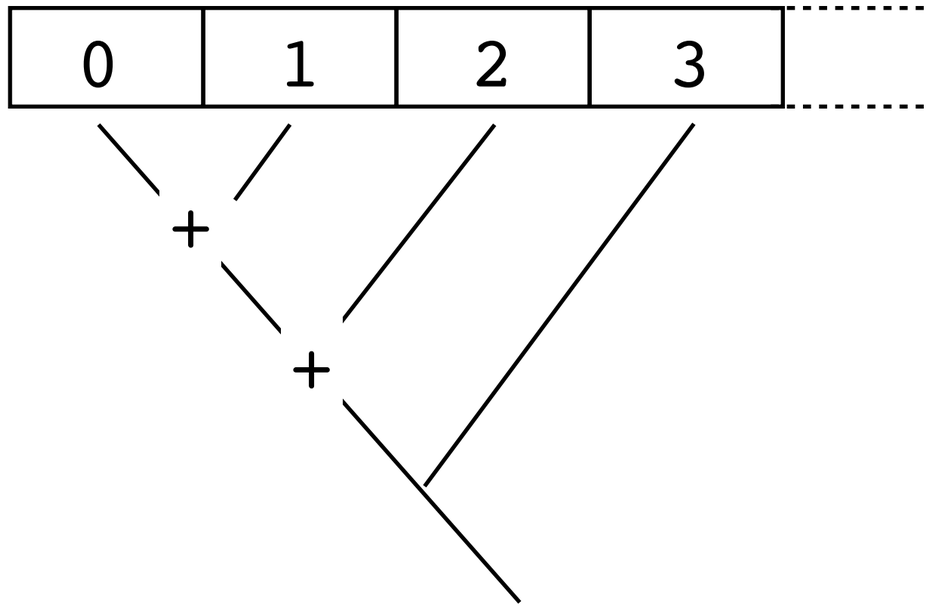
- Using an accumulator:

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

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

Reduction – Why?

- Using an accumulator:

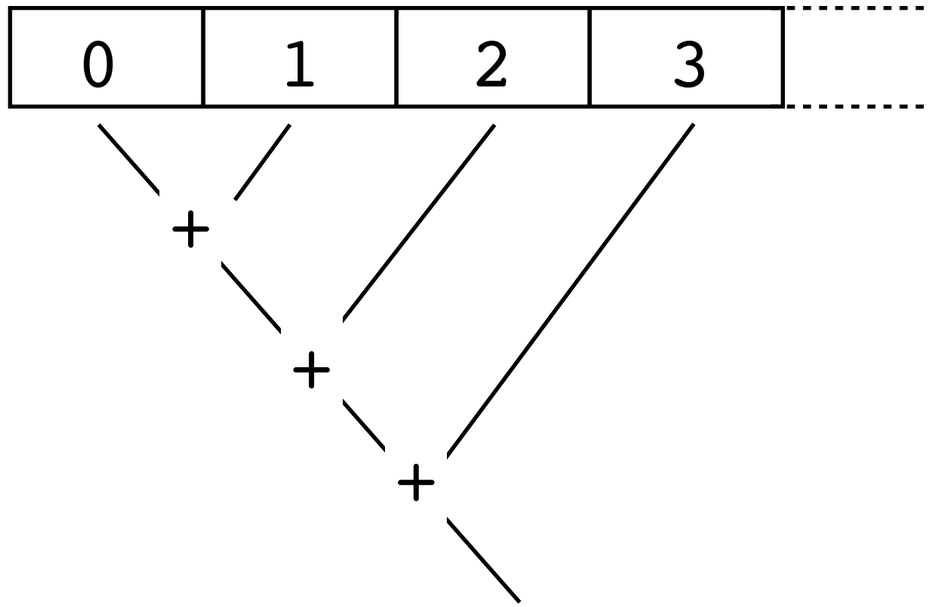
```
int[] vals = new int[100];
Arrays.setAll(vals, i -> i);

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



Reduction – Why?

- 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];
}
```



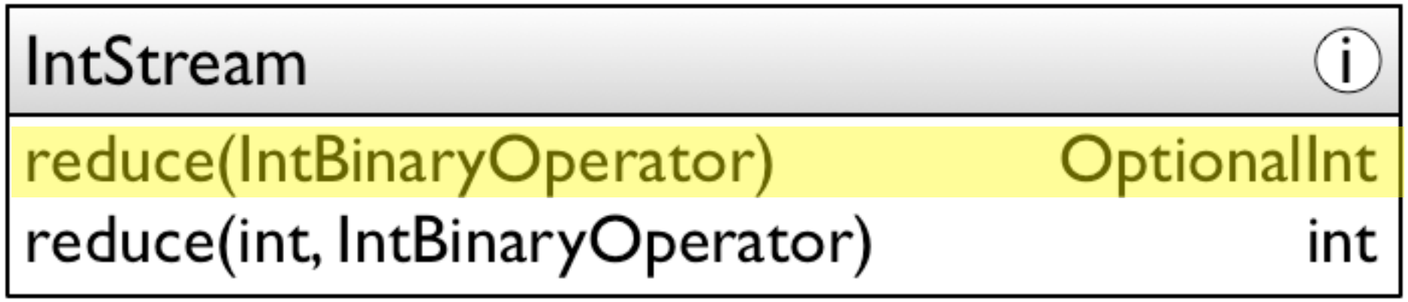
Reduction – Why?

- Avoiding an accumulator:

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

Reduction – Why?

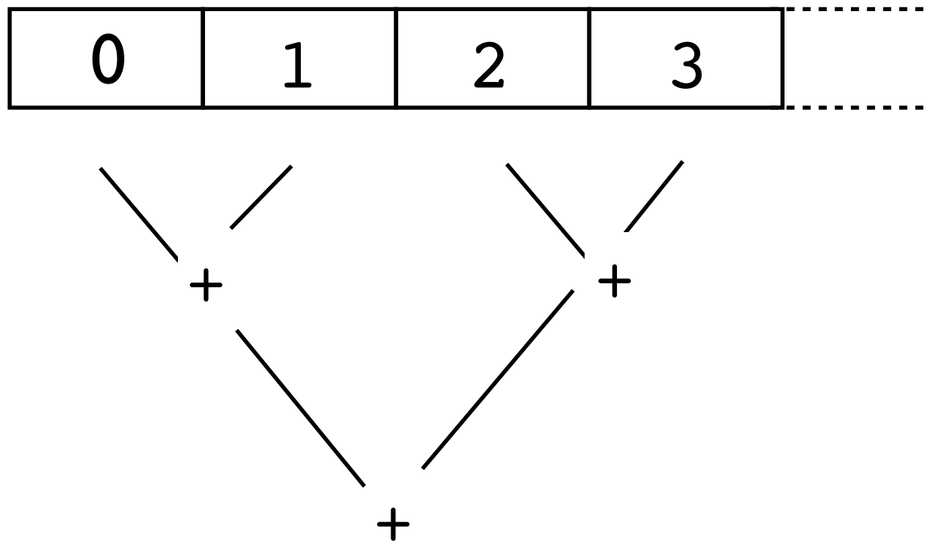
- Avoiding an accumulator:



Reduction – Why?

- Avoiding an accumulator:

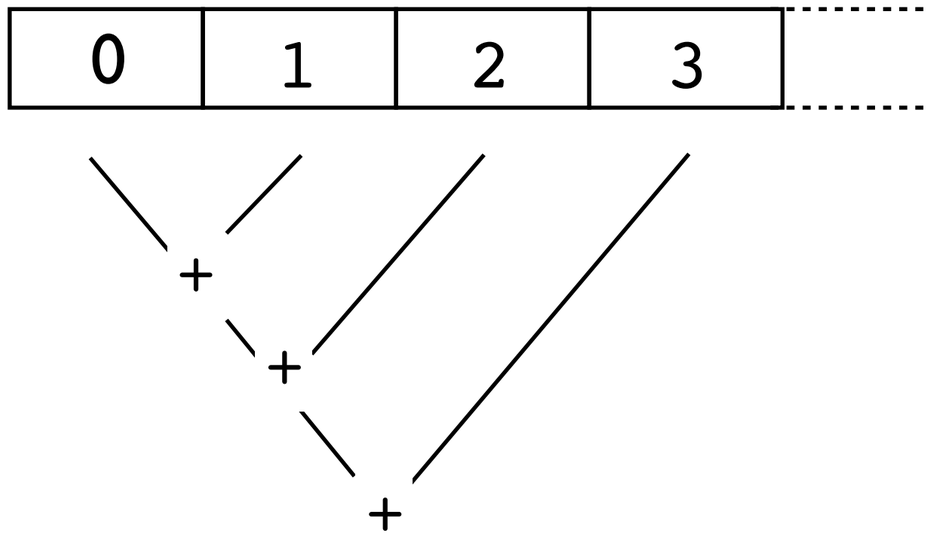
```
int[] vals = new int[100];
Arrays.setAll(vals, i -> i);

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



Reduction – Why?

- Avoiding an accumulator:

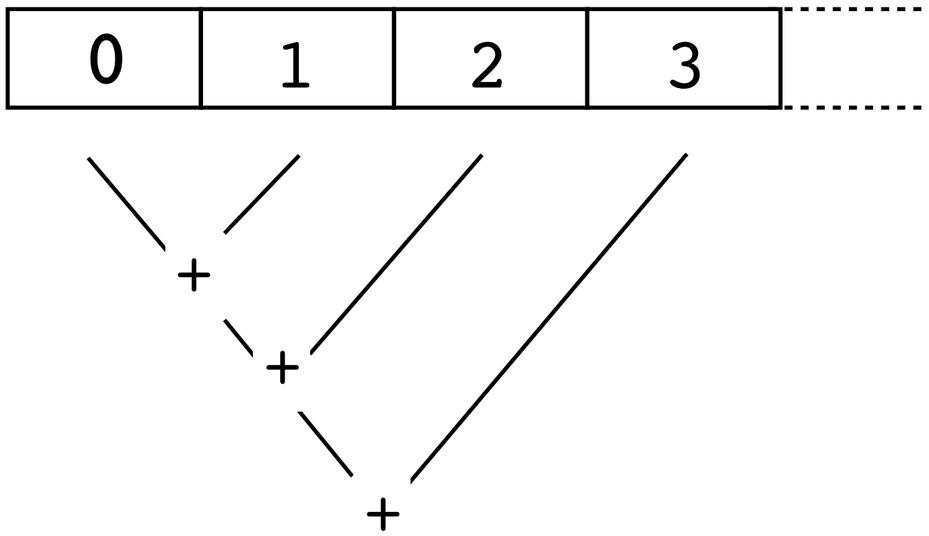
```
int[] vals = new int[100];
Arrays.setAll(vals, i -> i);

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



Reduction – Why?

- Avoiding an accumulator:

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

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

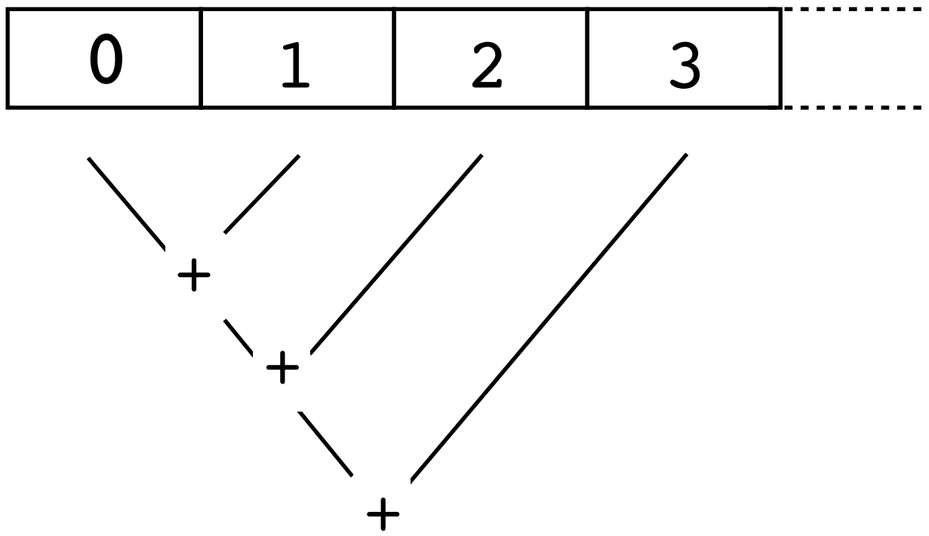


BinaryOperator must be associative!

Reduction – Why?

- Avoiding an accumulator:

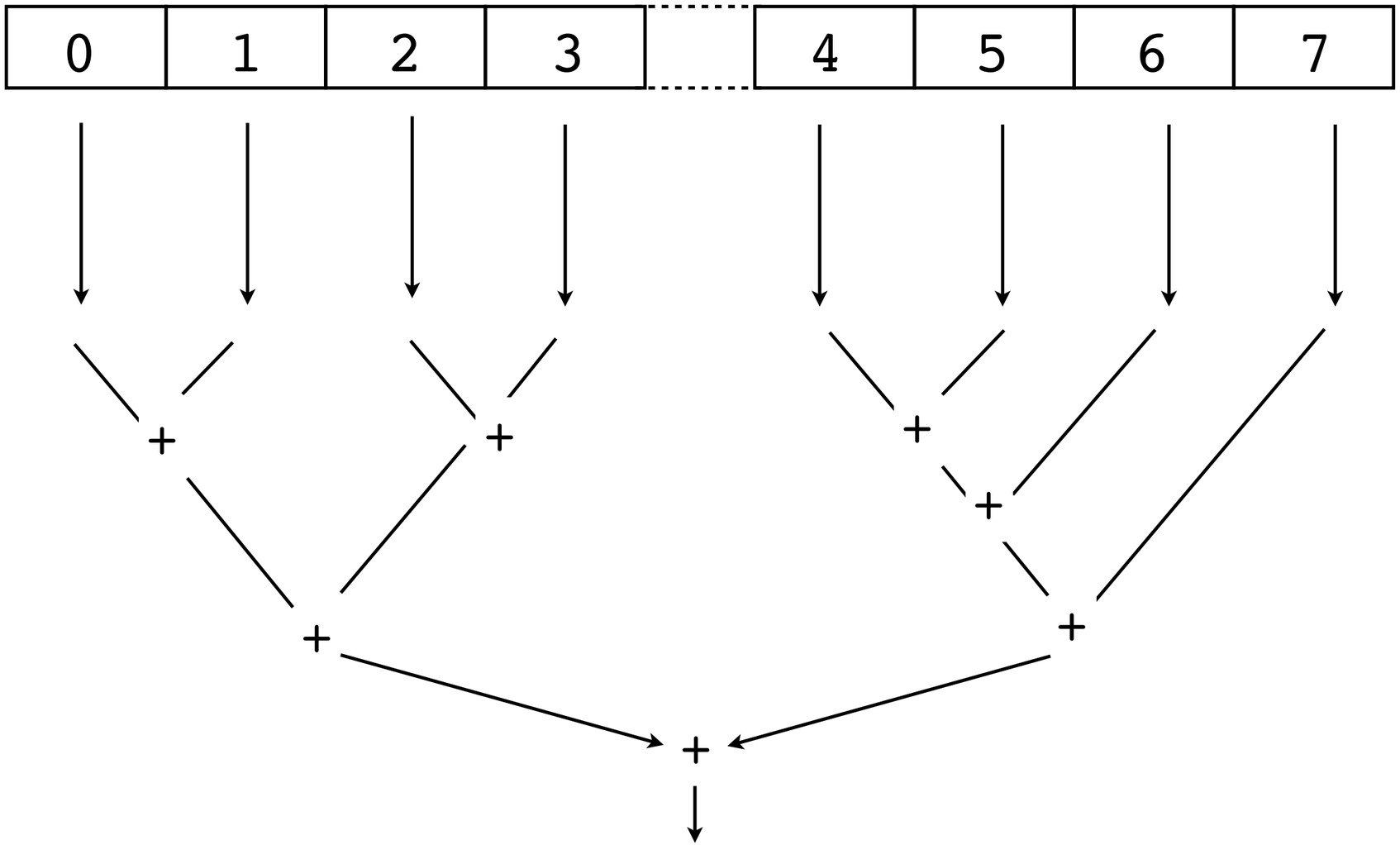
```
int[] vals = new int[100];
Arrays.setAll(vals, i -> i);

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



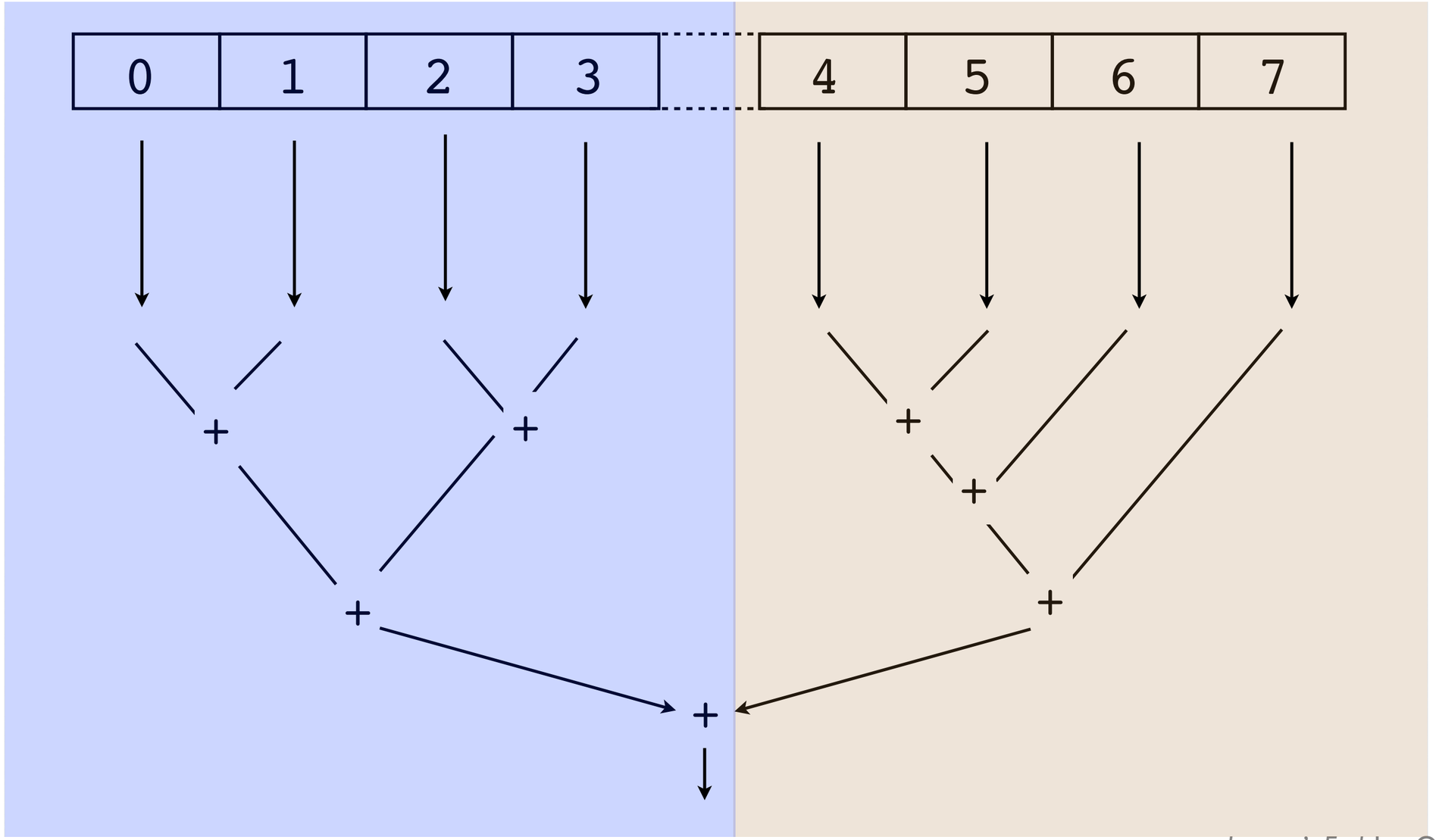
BinaryOperator must be associative!

a + (b + c) = (a + b) + c

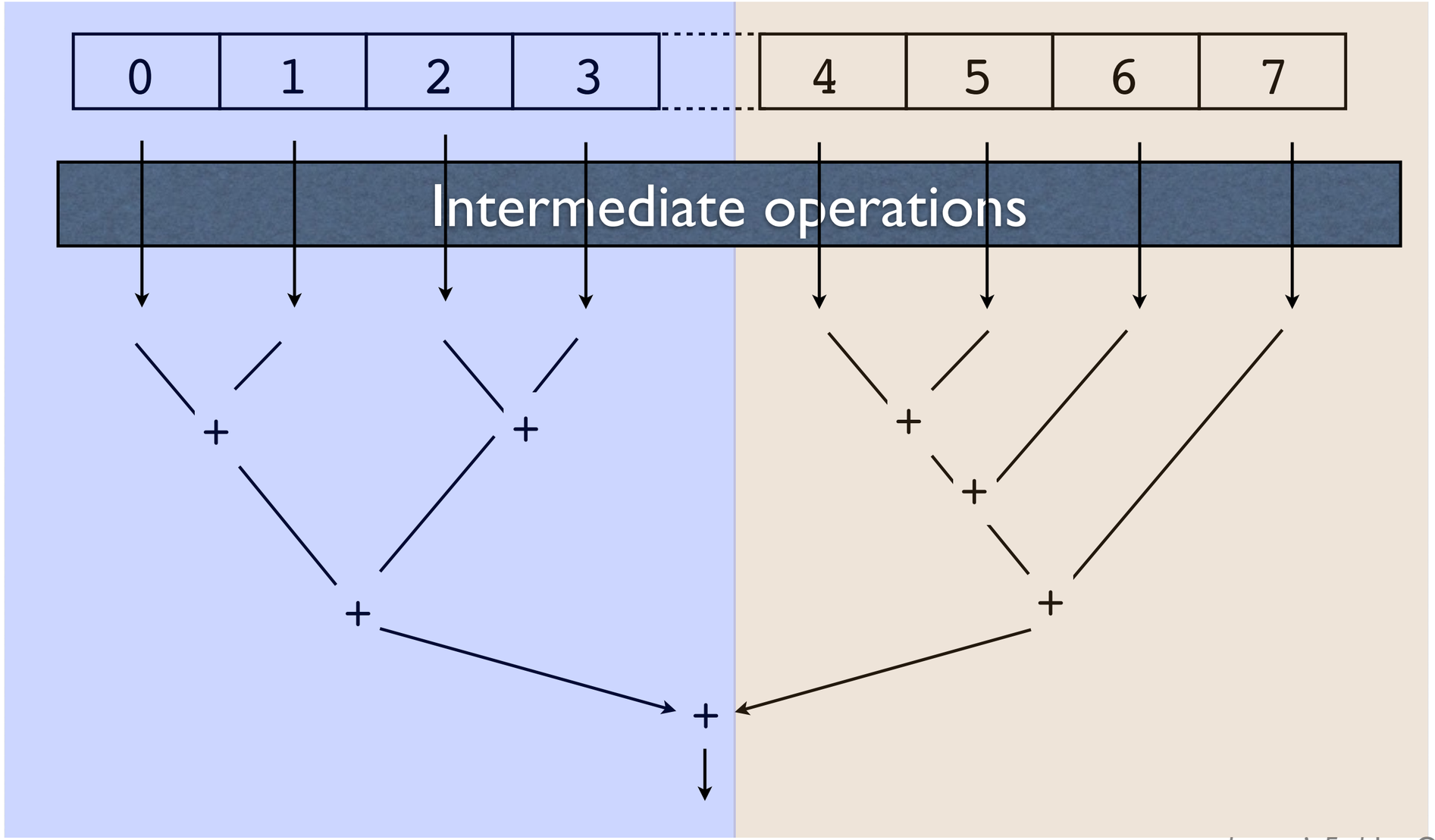
Why Reduction?



Why Reduction?



Why Reduction?



Reduction on Immutable Values

Reduction works on immutable values too

Stream<T>	ⓘ
reduce(BinaryOperator<T>)	Optional<T>
reduce(T, BinaryOperator<T>)	T
reduce(U, BiFunction<U,T,U>, BinaryOperator<U>)	U

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

Optional<BigDecimal> sum = Arrays.stream(vals)
    .reduce(BigDecimal::add);
```

Reduction on Immutable Values

Reduction works on immutable values too

Stream<T>	ⓘ
reduce(BinaryOperator<T>)	Optional<T>
reduce(T, BinaryOperator<T>)	T
reduce(U, BiFunction<U,T,U>, BinaryOperator<U>)	U

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

Optional<BigDecimal> sum = Arrays.stream(vals)
    .reduce(BigDecimal::add);
```

Reduction to Collections?

Reduction to Collections?

This also works with collections – sort of ...

- for each element, client code creates a new empty collection and adds the single element to it
- combines collections using `addAll`

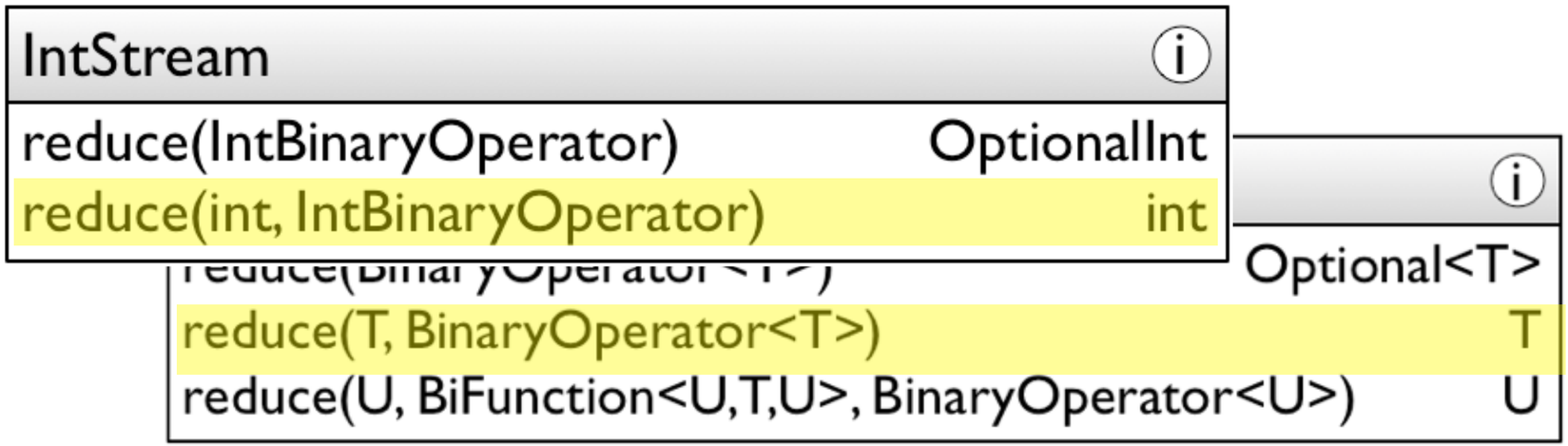
Reduction to Collections?

This also works with collections – sort of ...

- for each element, client code creates a new empty collection and adds the single element to it
- combines collections using `addAll`

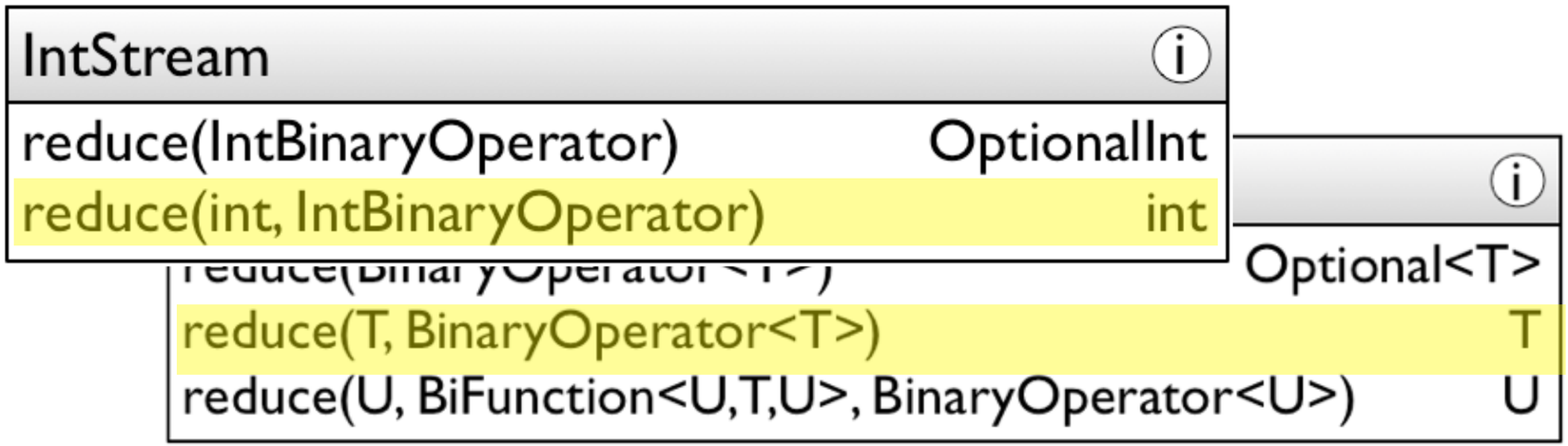
We can do better!

Reduction over an Identity




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

BigDecimal sum = Arrays.stream(vals)
    .reduce(BigDecimal.ZERO,BigDecimal::add);
```

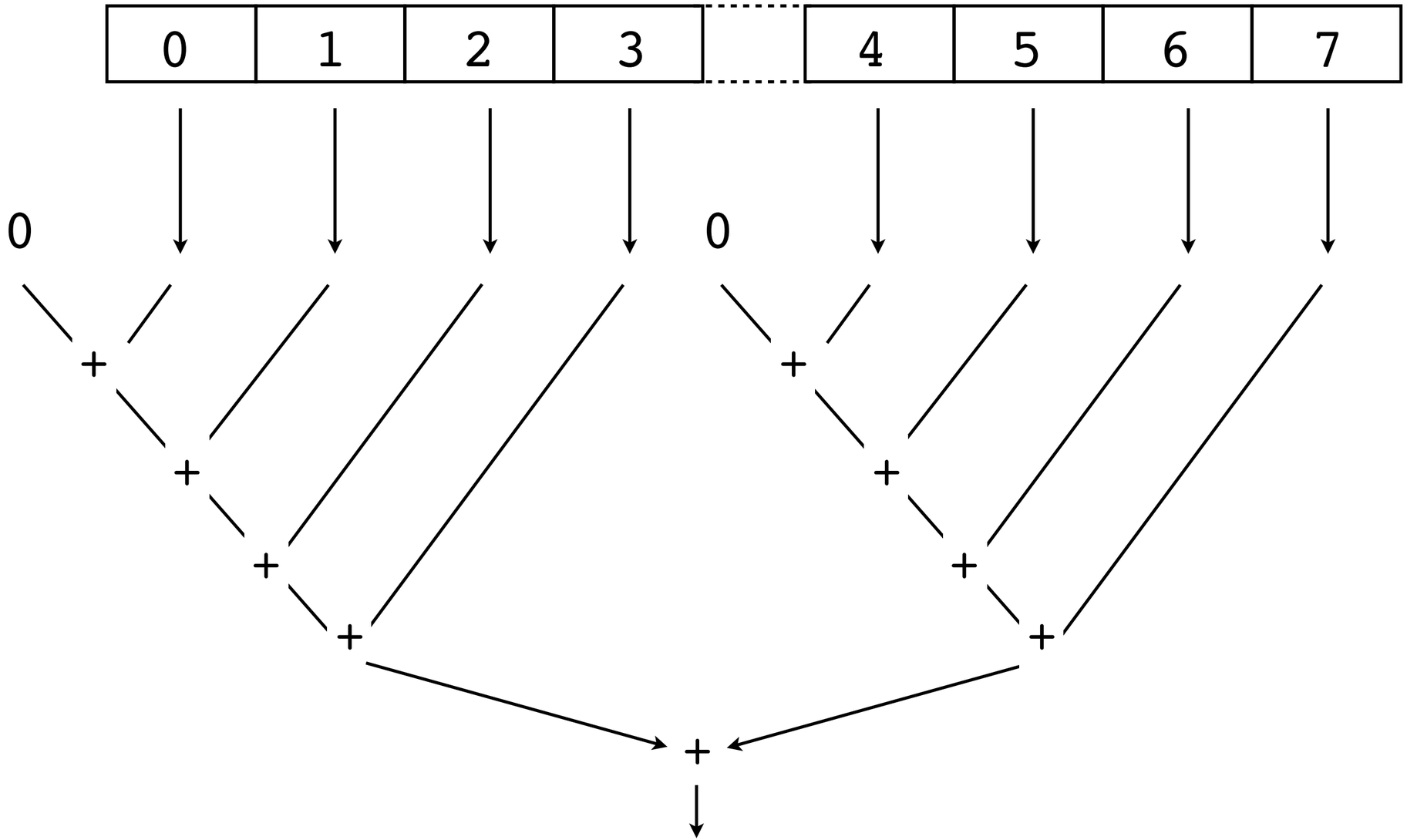
Reduction over an Identity



Works for immutable objects:

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

BigDecimal sum = Arrays.stream(vals)
    .reduce(BigDecimal.ZERO,BigDecimal::add);
```

Reduction over an Identity



Reduction to Collections?

Reduction to Collections?

Reduction over an identity doesn't work with collections *at all*

- reduction **reuses** the identity element

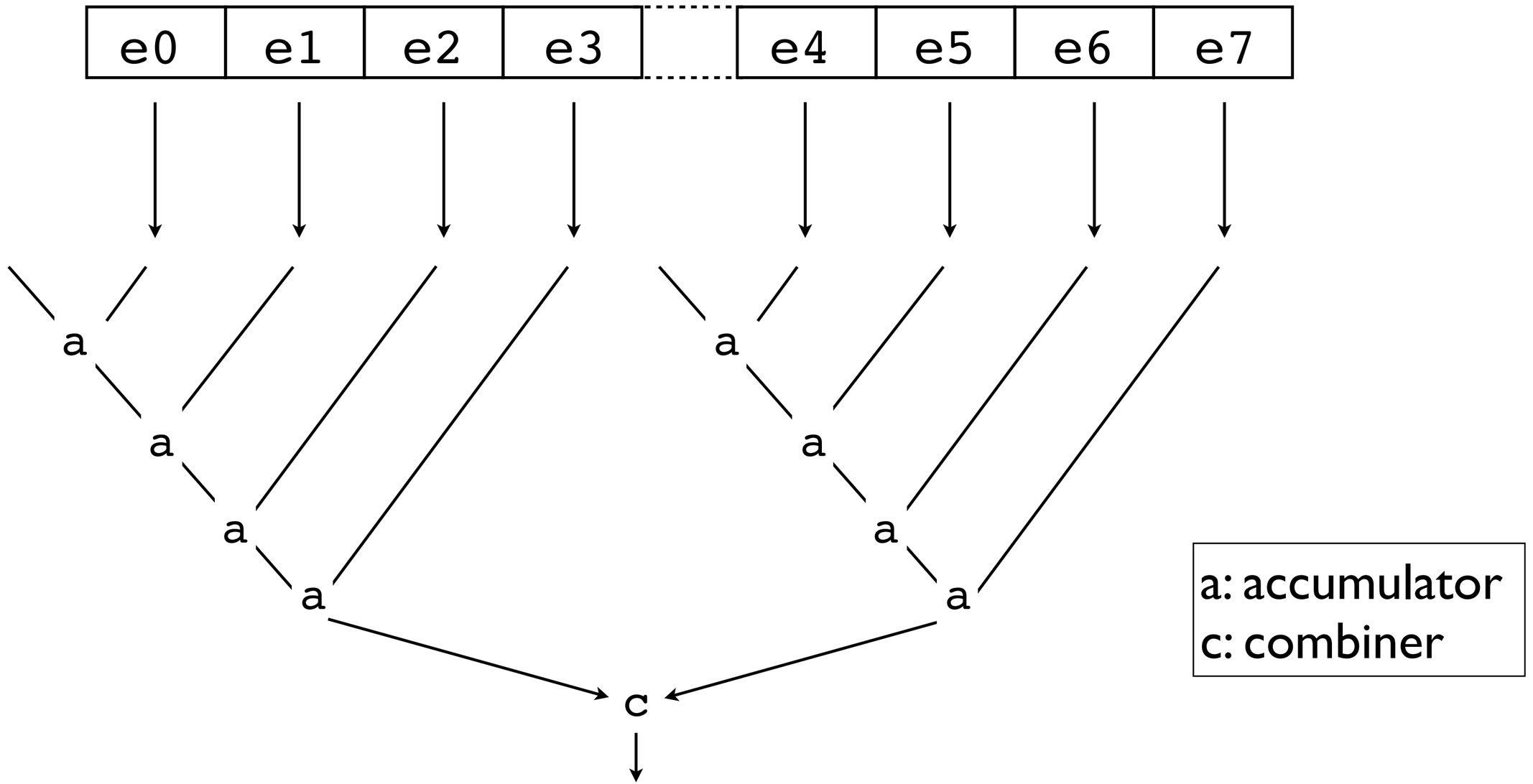
Reduction to Collections?

Reduction over an identity doesn't work with collections *at all*

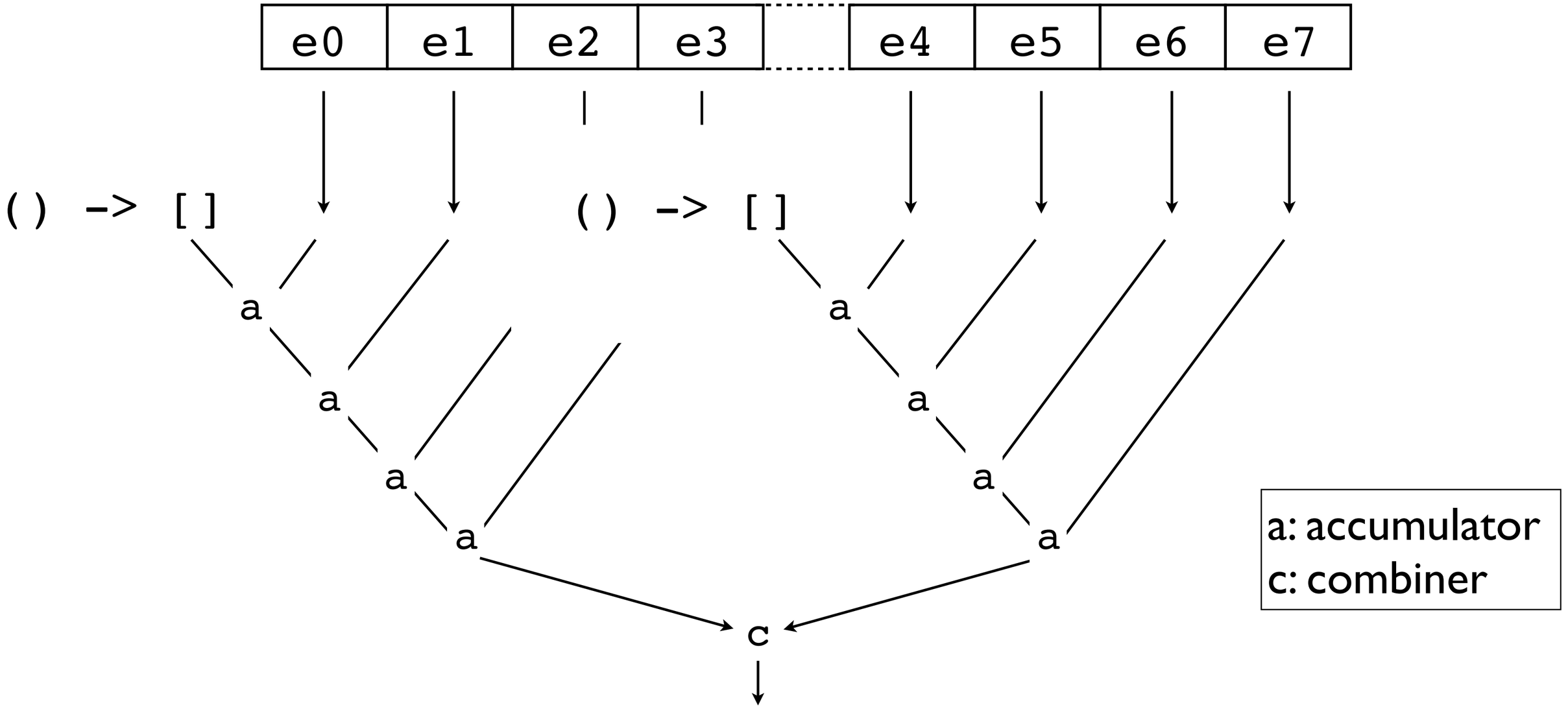
- reduction **reuses** the identity element

We've got to do better!

The Answer – Collectors



The Answer – Collectors



Collectors

So to define a collector, you need to provide

- Supplier
- Accumulator
- Combiner

That sounds really hard!

Good then that we don't have to do it

- – very often

Collectors API

Factory methods in the Collectors class. They produce standalone collectors, accumulating to:

- framework-supplied containers;
- custom collections;
- classification maps.

Journey's End – Agenda

- Why collectors?
- Using the predefined collectors
  - Intermission!
- Worked problems
- Writing your own

Using the Predefined Collectors

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

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

`groupingBy(), partitioningBy(), groupingByConcurrent()`

Using the Predefined Collectors

Predefined Standalone Collectors – from methods in `Collections` class

framework provides the Supplier

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

`groupingBy(), partitioningBy(), groupingByConcurrent()`

Using the Predefined Collectors

Predefined Standalone Collectors – from methods in `Collections` class

framework provides the Supplier

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

user provides the Supplier

`groupingBy(), partitioningBy(), groupingByConcurrent()`

Using the Predefined Collectors

Predefined Standalone Collectors – from methods in `Collections` class

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

`groupingBy()`, `partitioningBy()`, `groupingByConcurrent()`

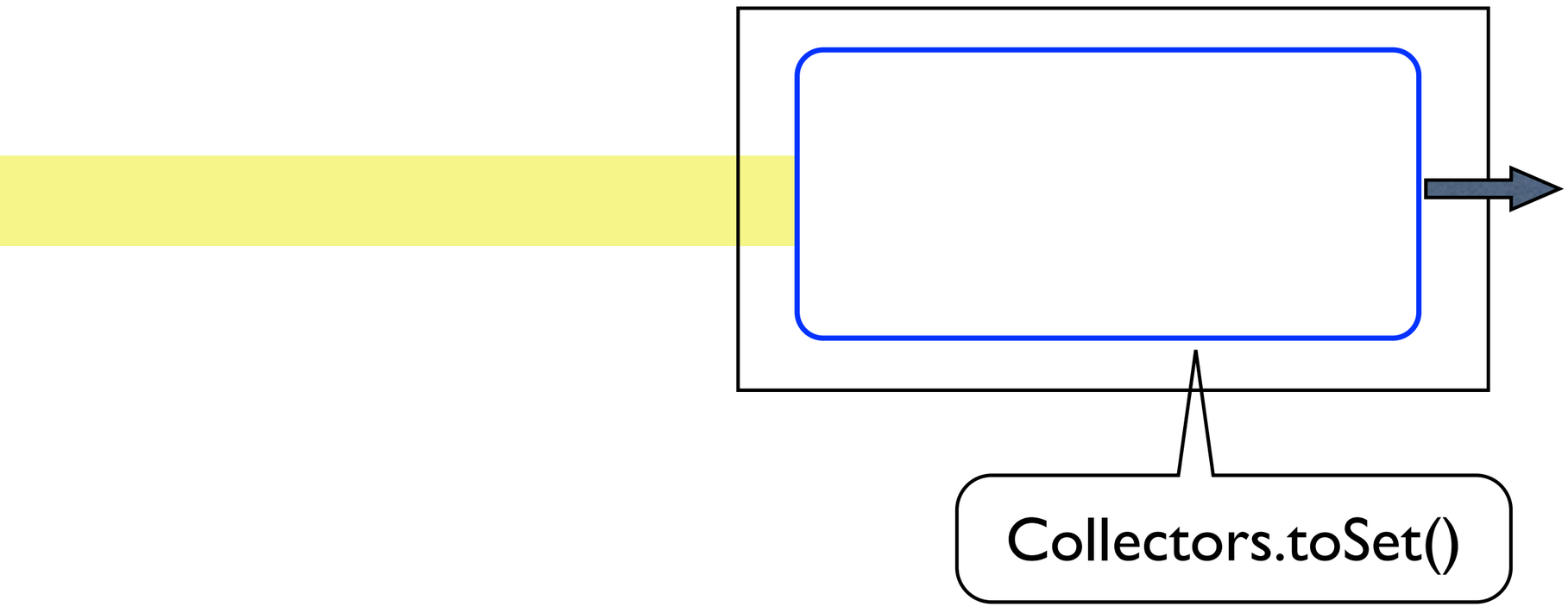
framework provides the Supplier

user provides the Supplier

produce a *classification map*

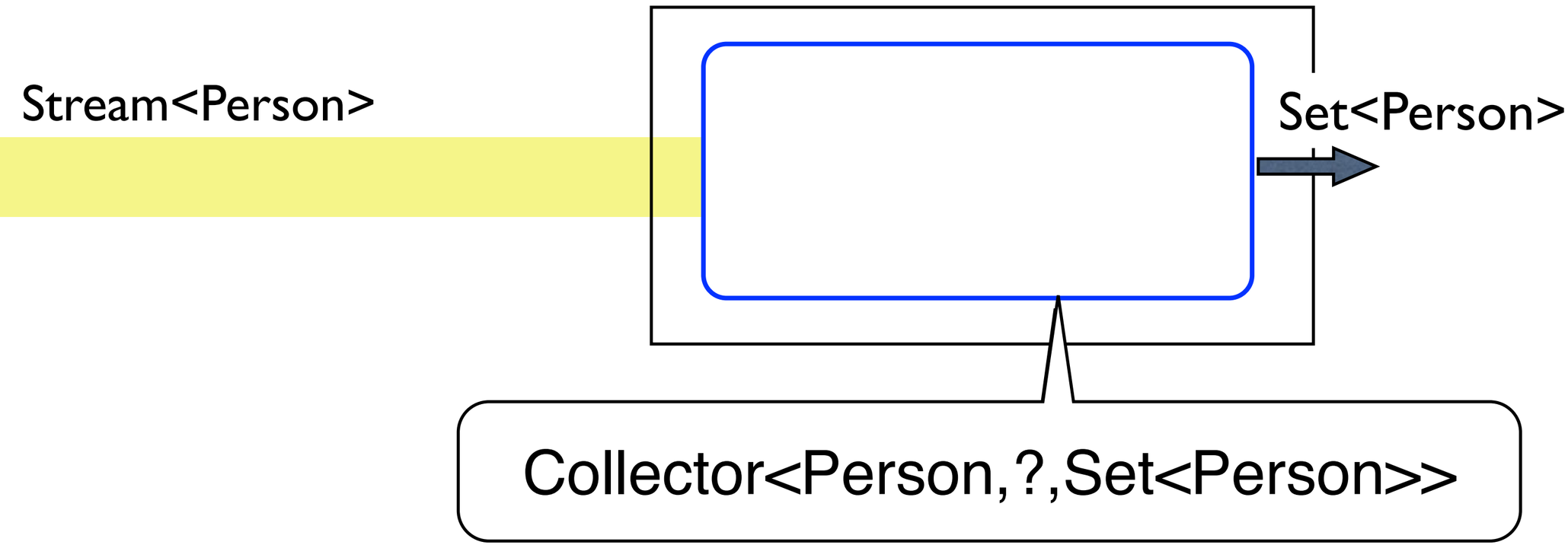
Simple Collector – `toSet()`

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



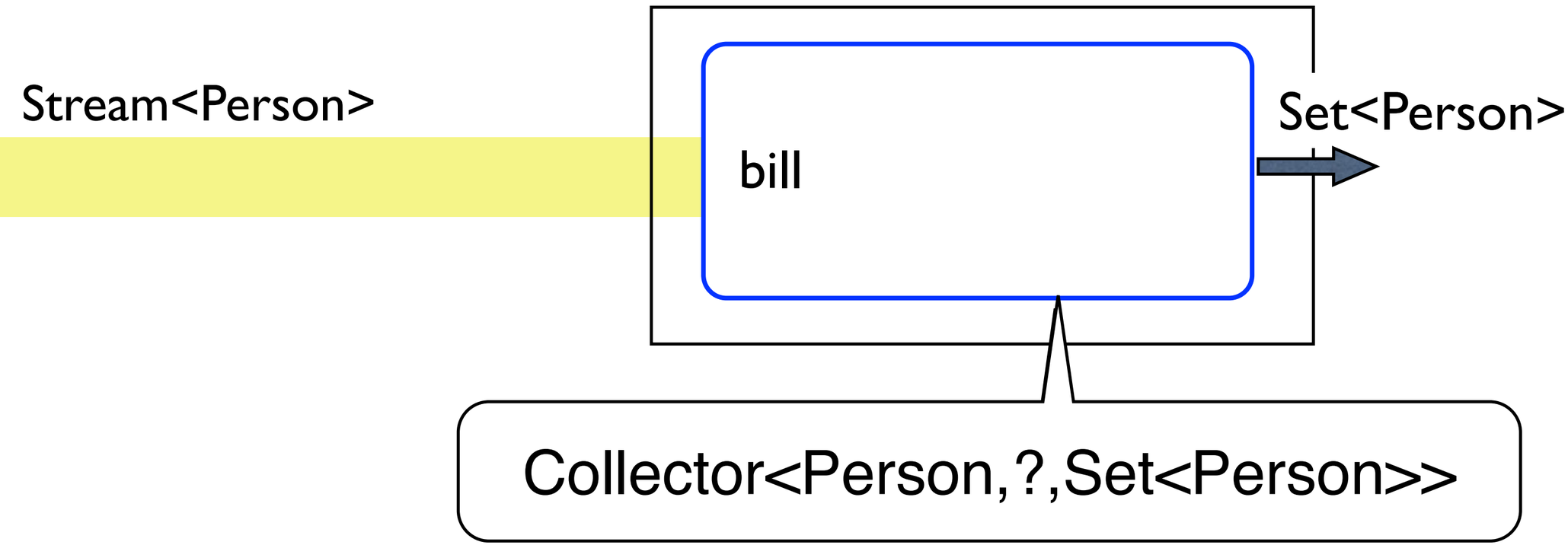
Simple Collector – `toSet()`

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



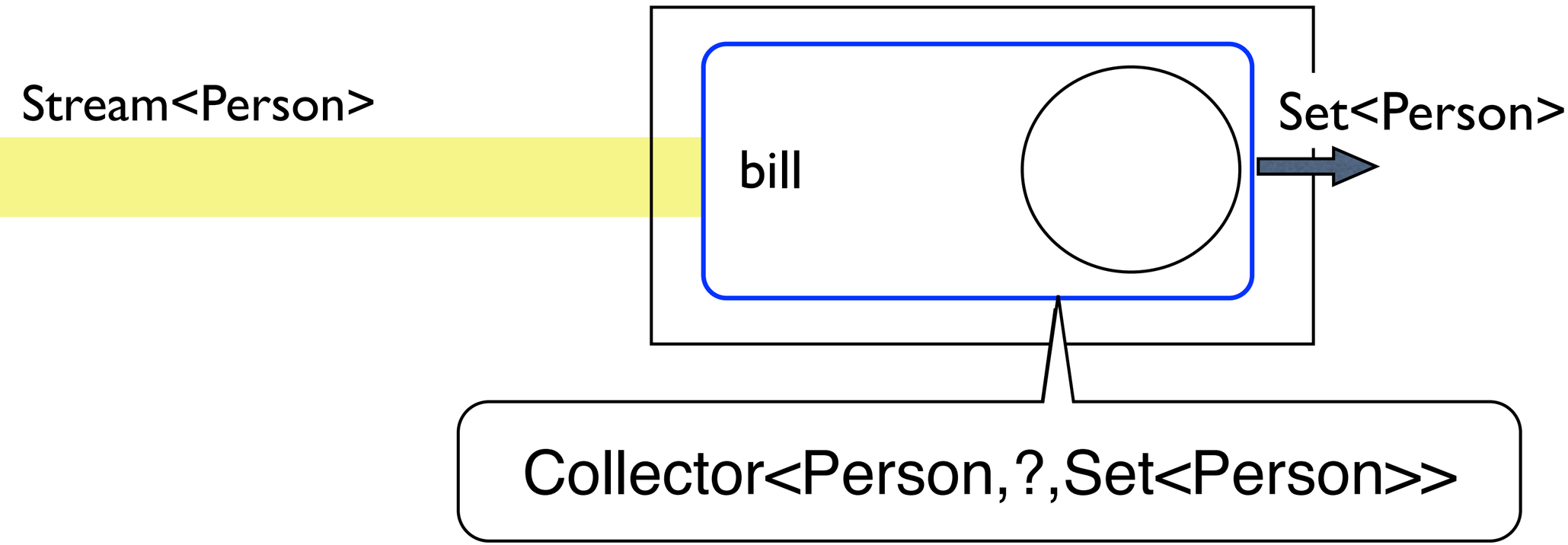
Simple Collector – `toSet()`

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



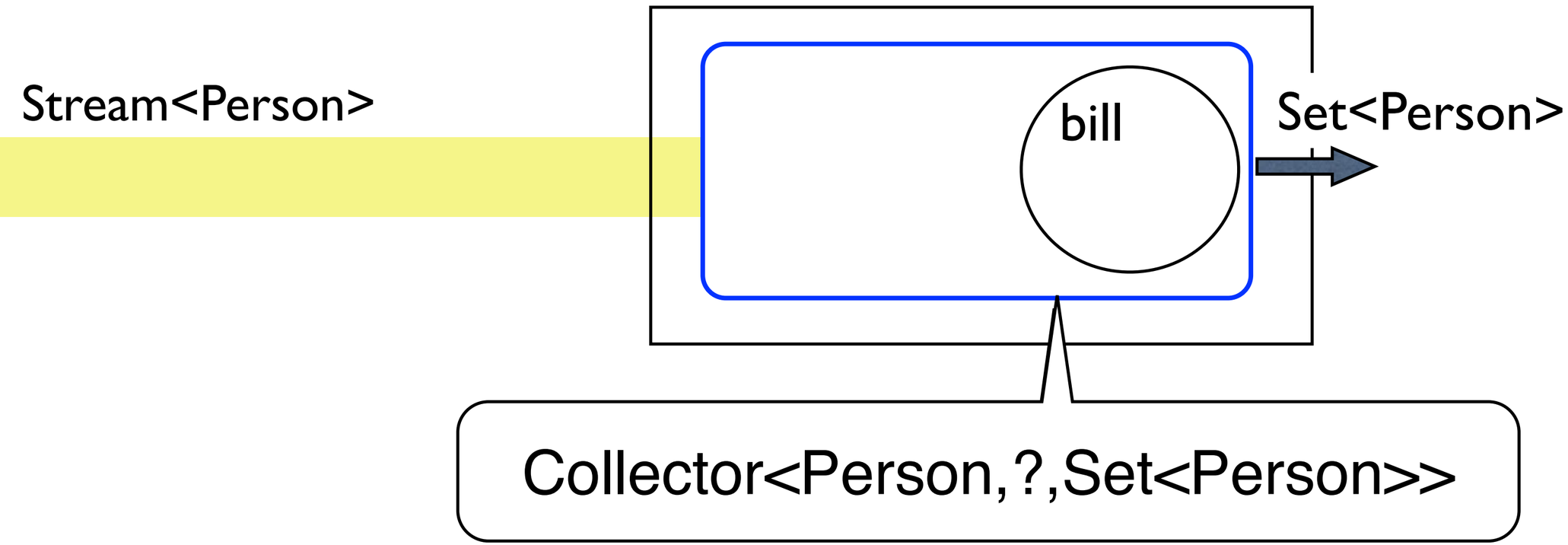
Simple Collector – `toSet()`

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



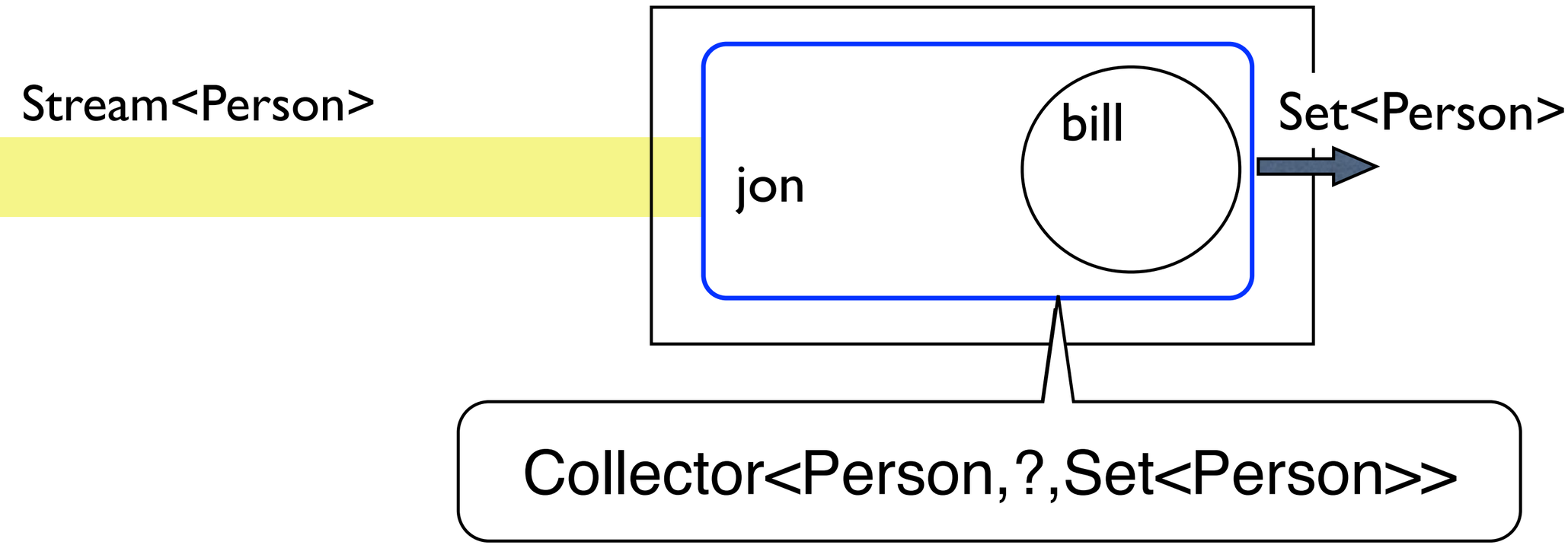
Simple Collector – `toSet()`

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



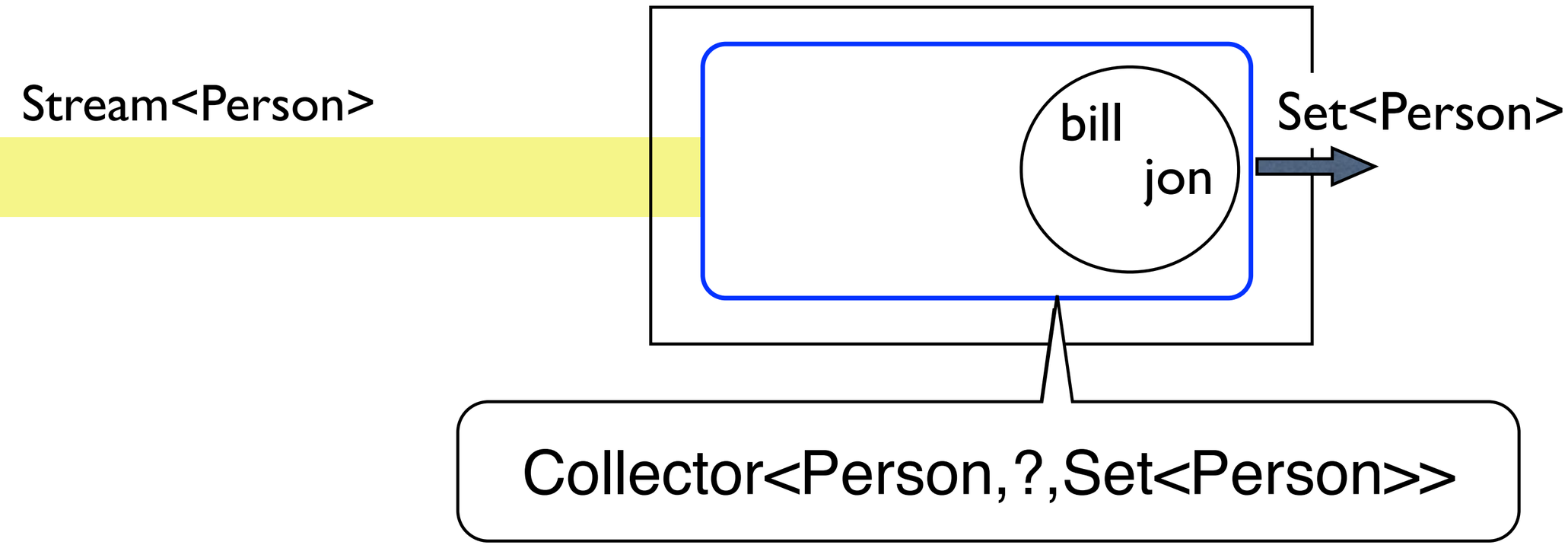
Simple Collector – `toSet()`

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



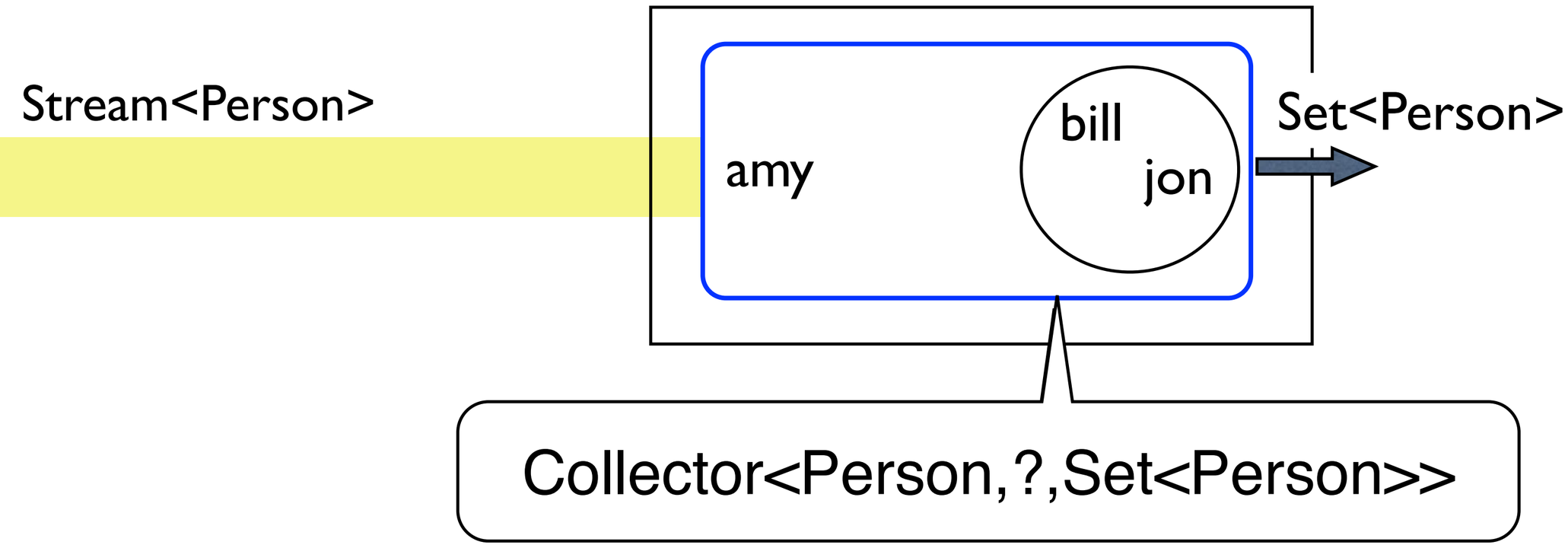
Simple Collector – `toSet()`

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



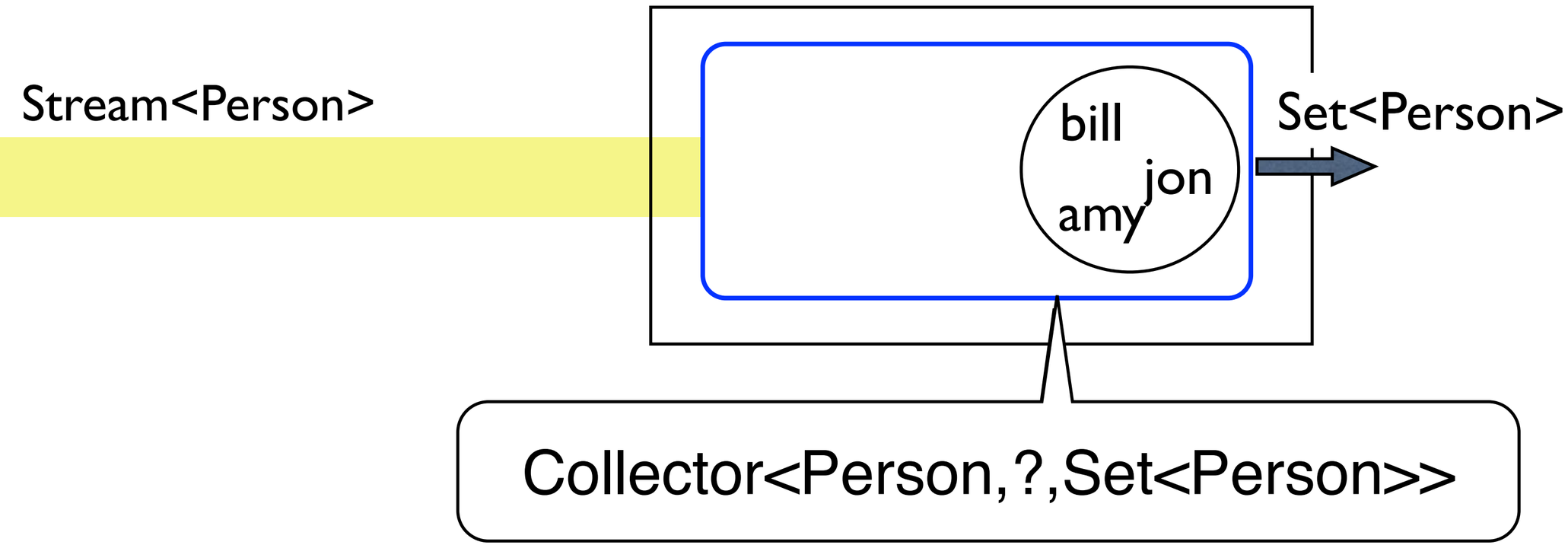
Simple Collector – `toSet()`

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



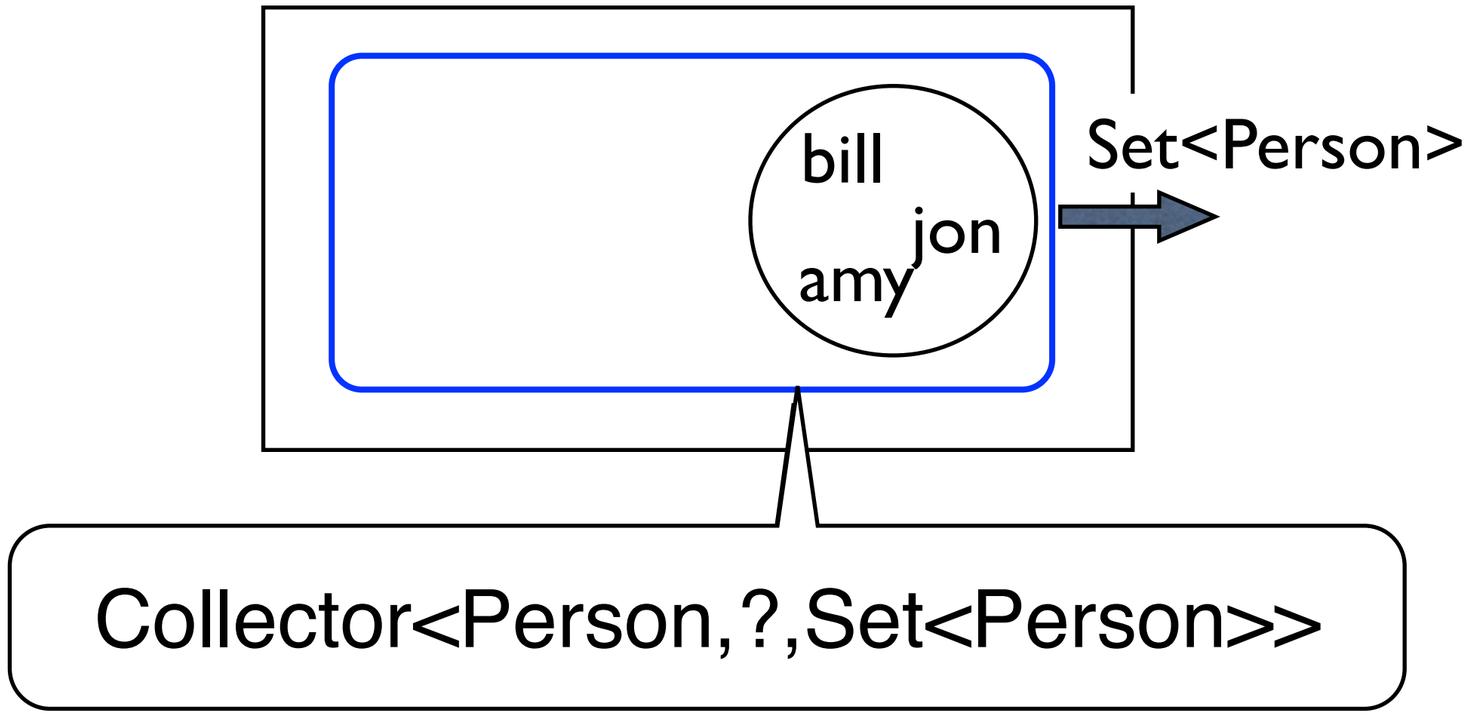
Simple Collector – `toSet()`

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



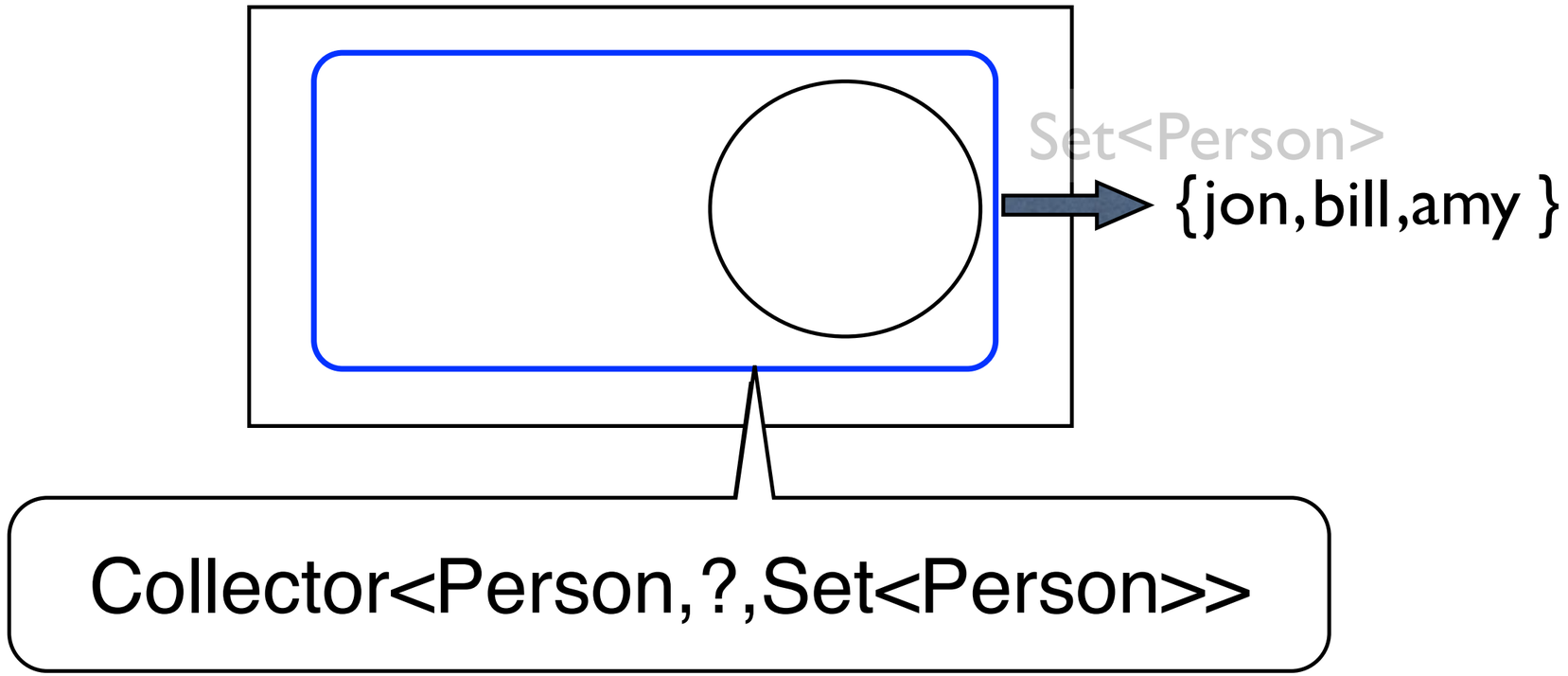
Simple Collector – `toSet()`

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



Simple Collector – `toSet()`

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

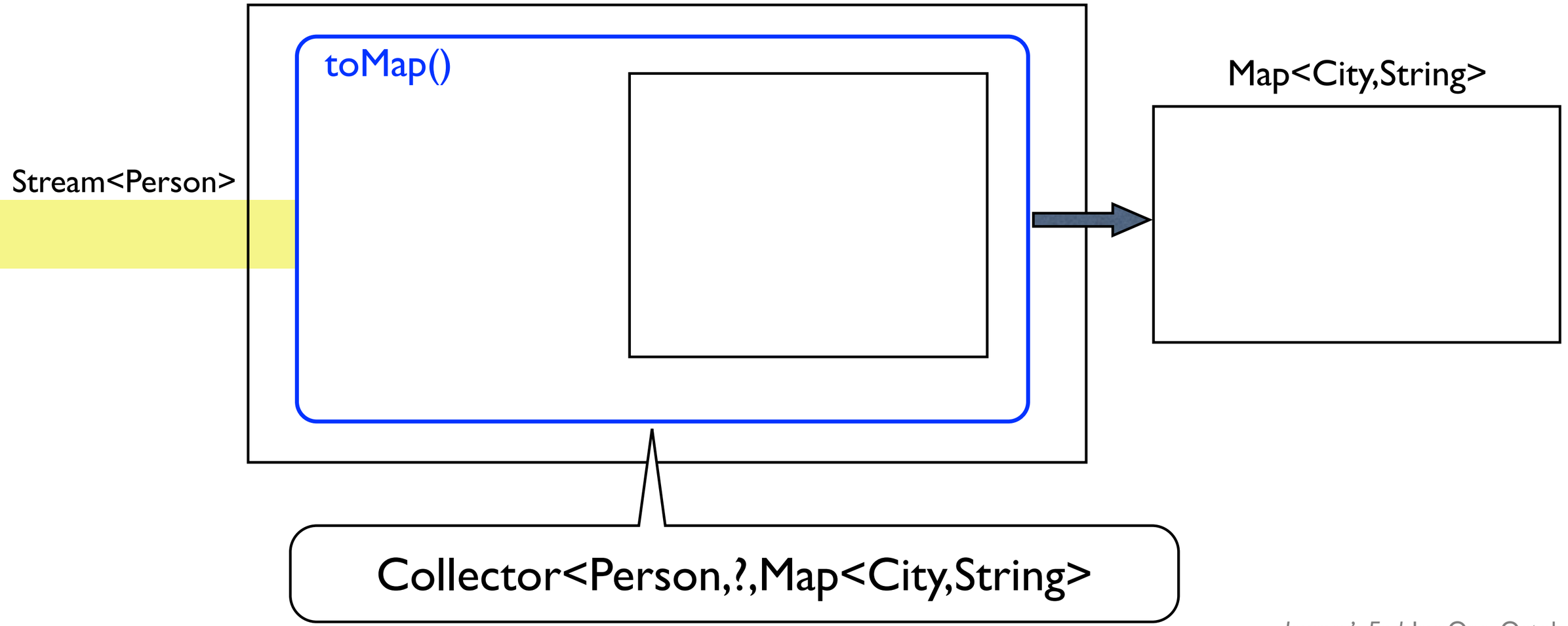


toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)

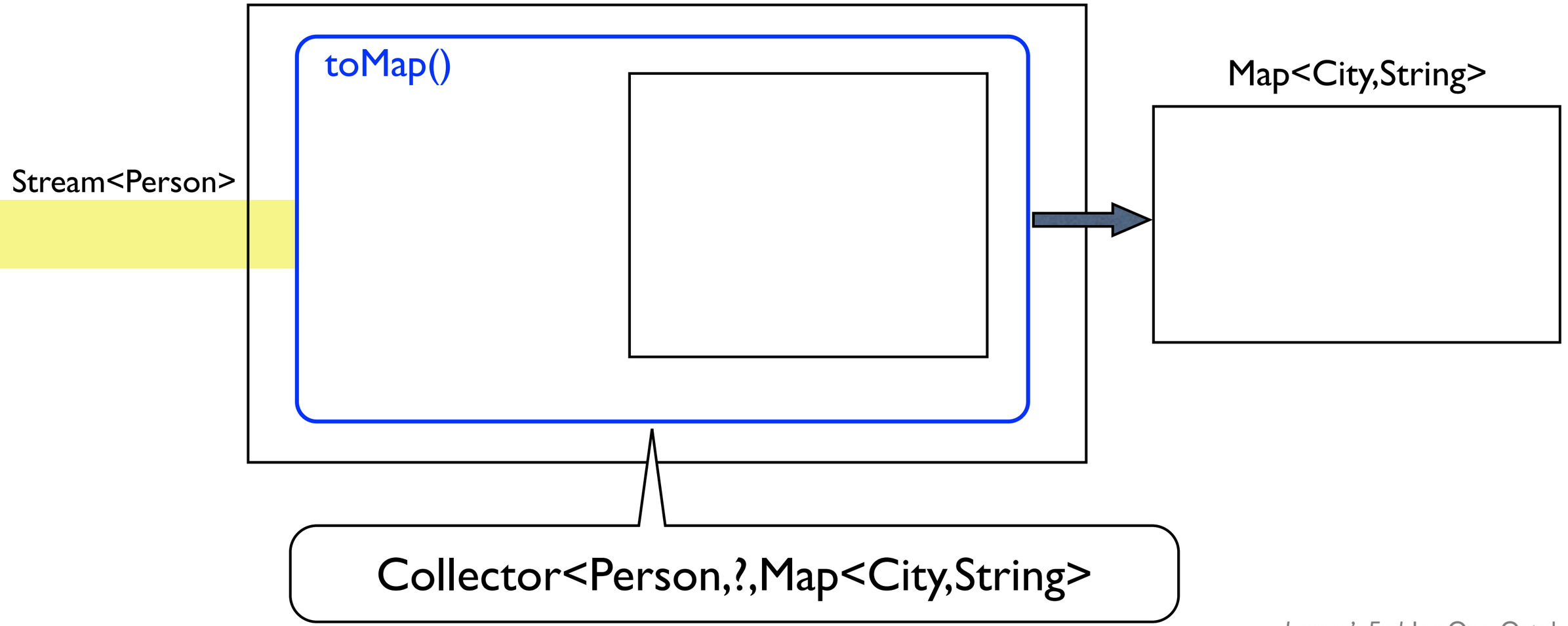
```
people.stream().collect(
     Collectors.toMap(
          Person::getCity,
          Person::getName))
```

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

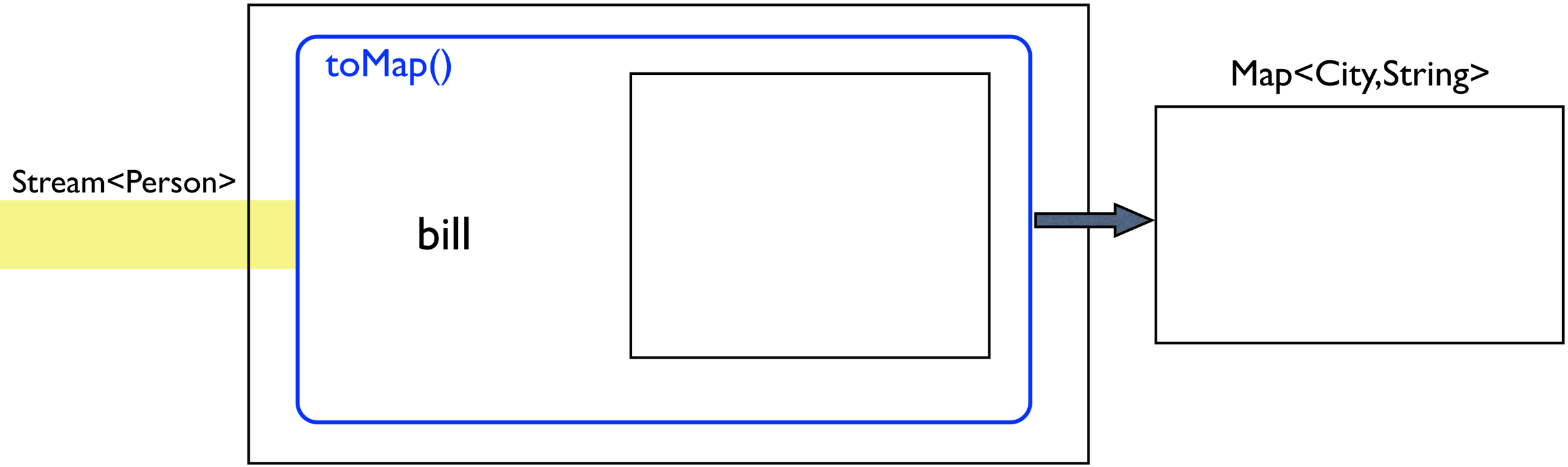
```
people.stream().collect(toMap(Person::getCity,Person::getName))
```



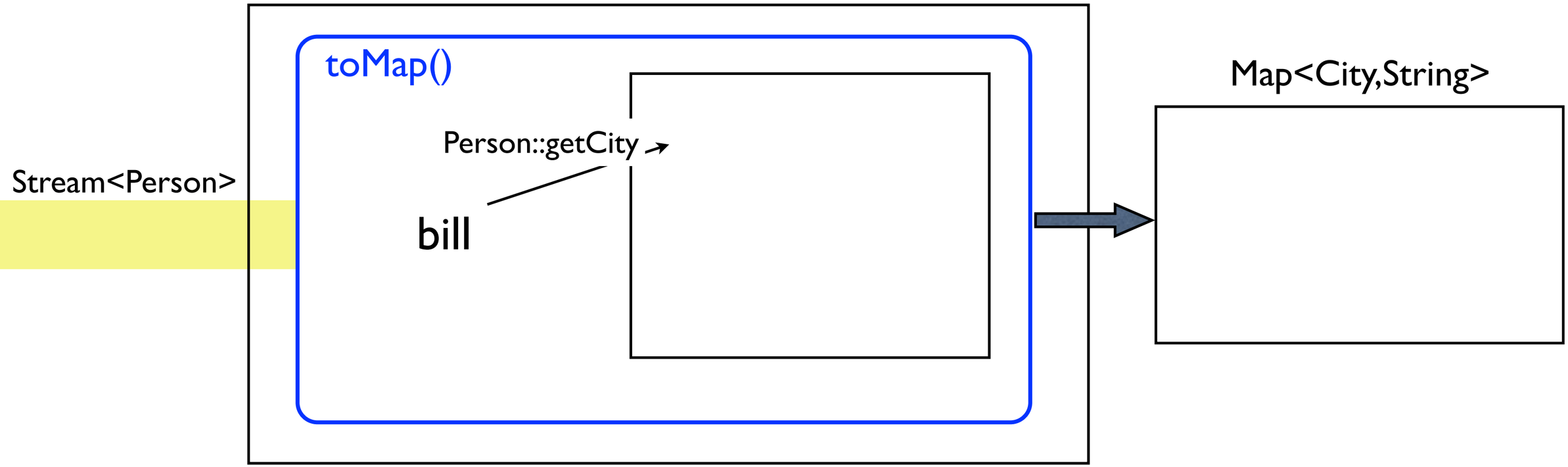
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



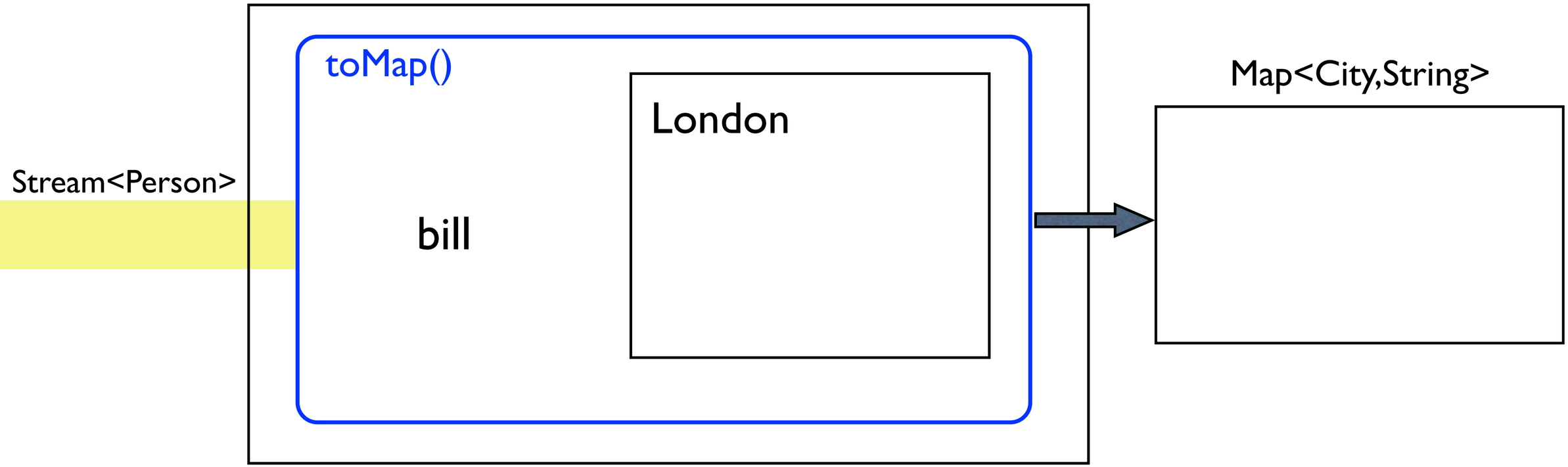
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



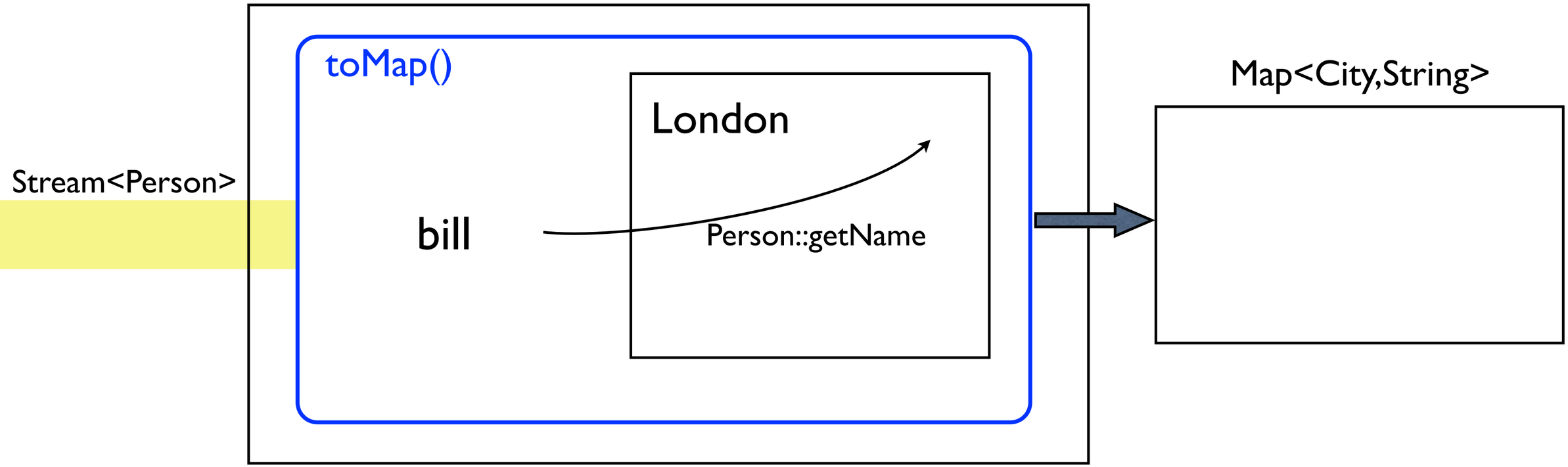
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



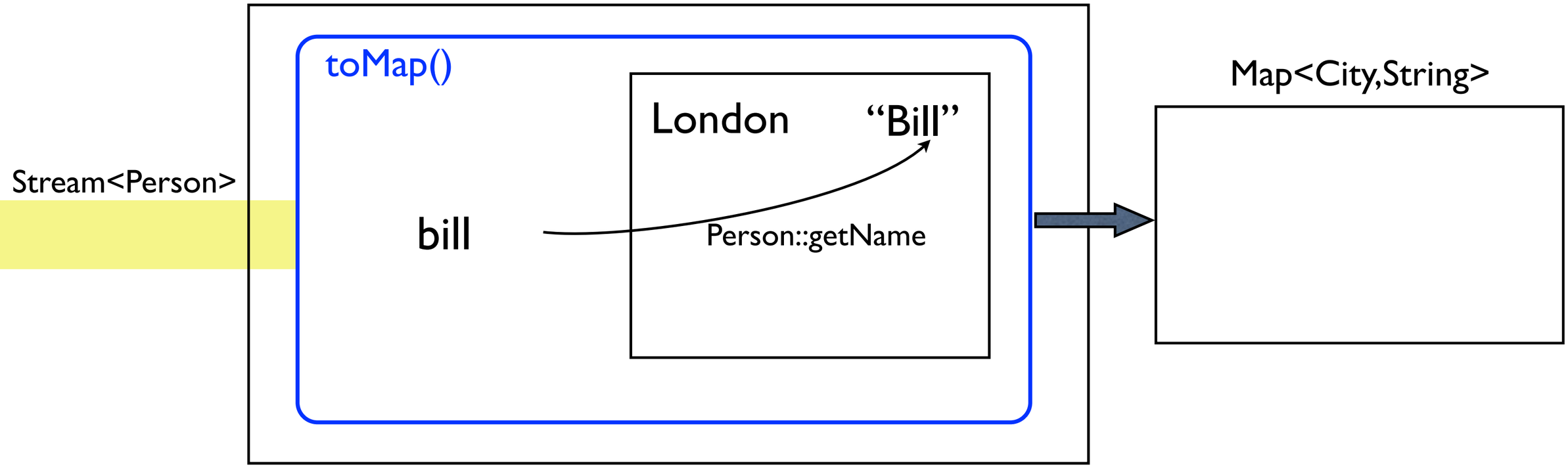
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



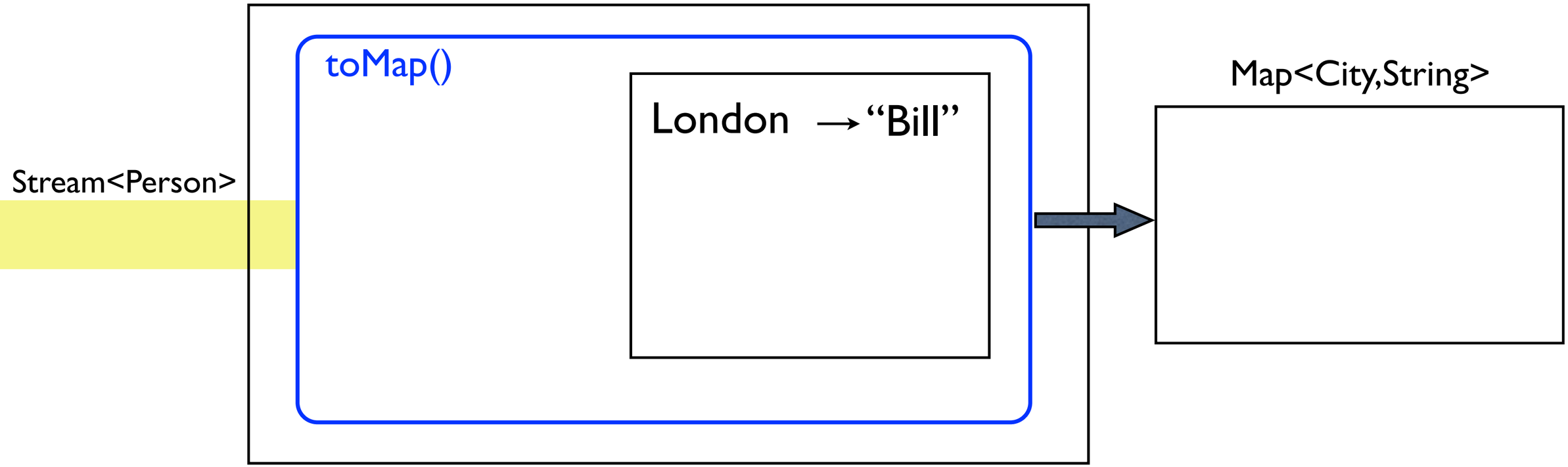
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



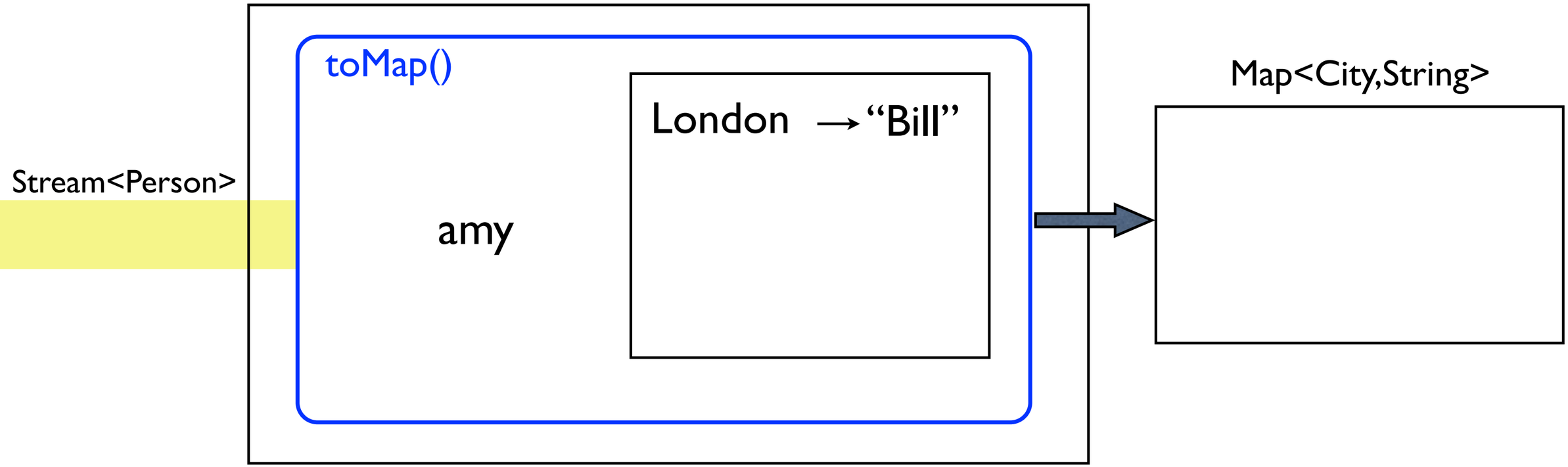
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



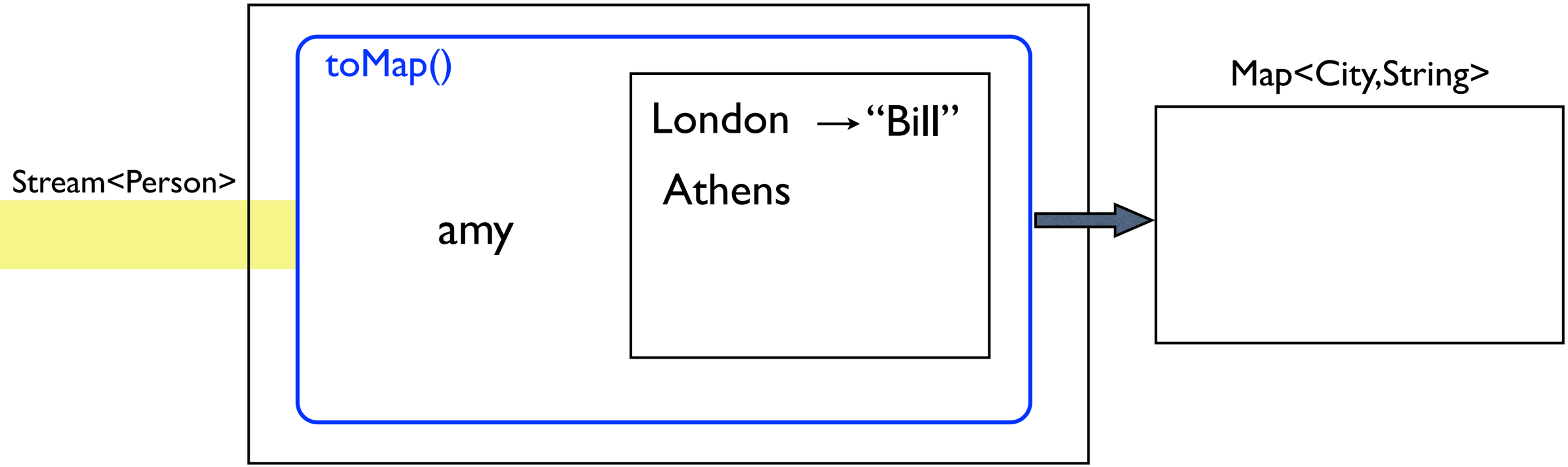
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



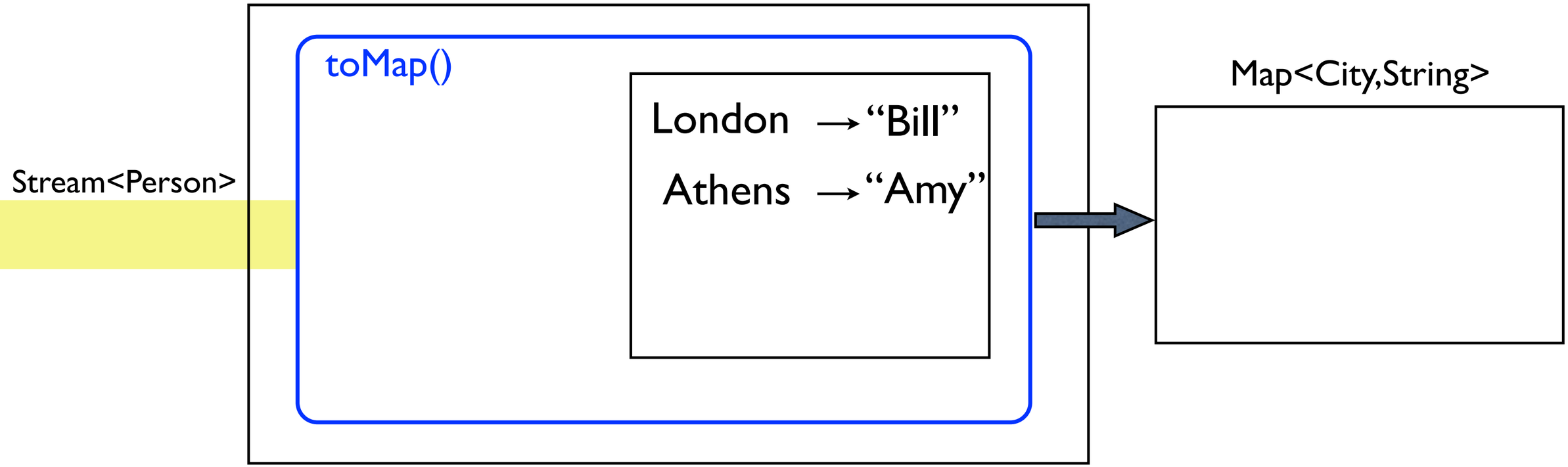
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



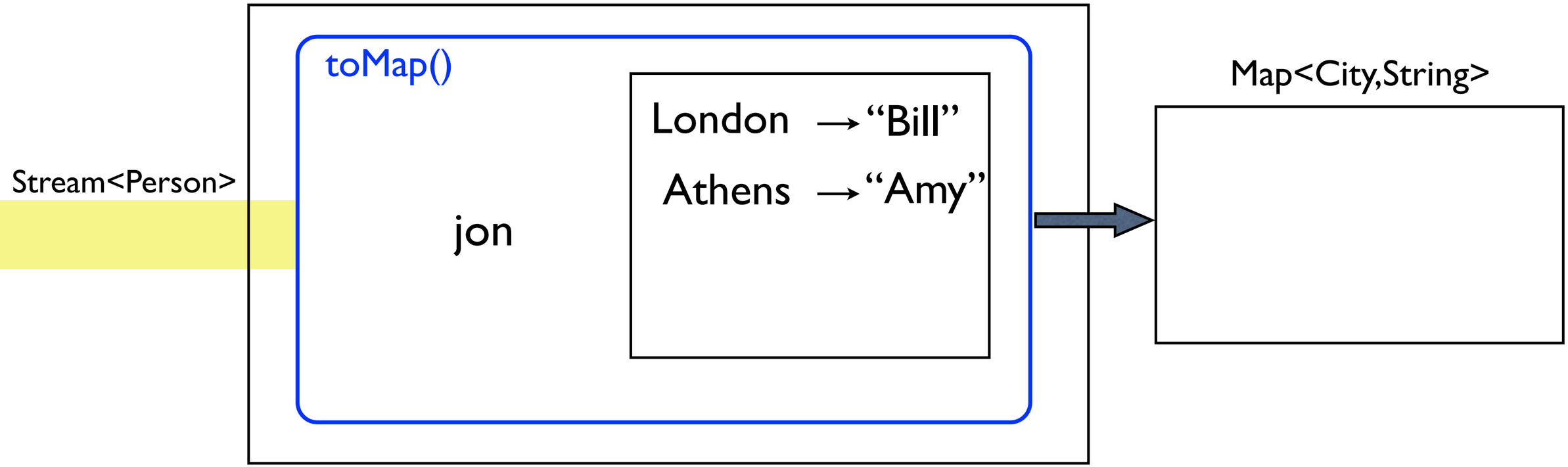
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



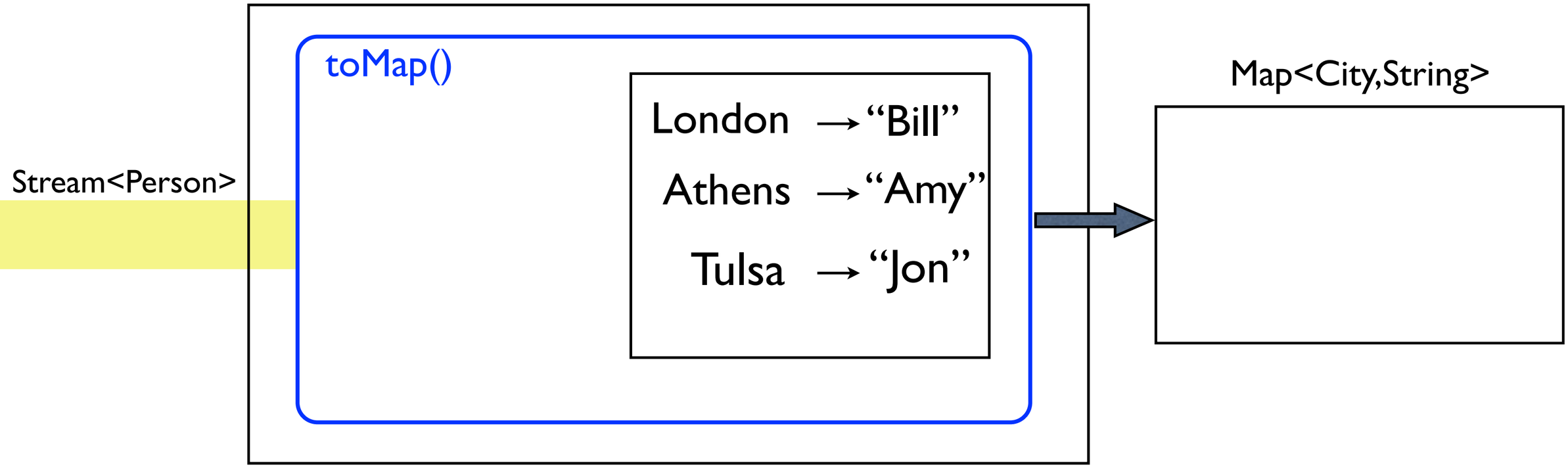
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



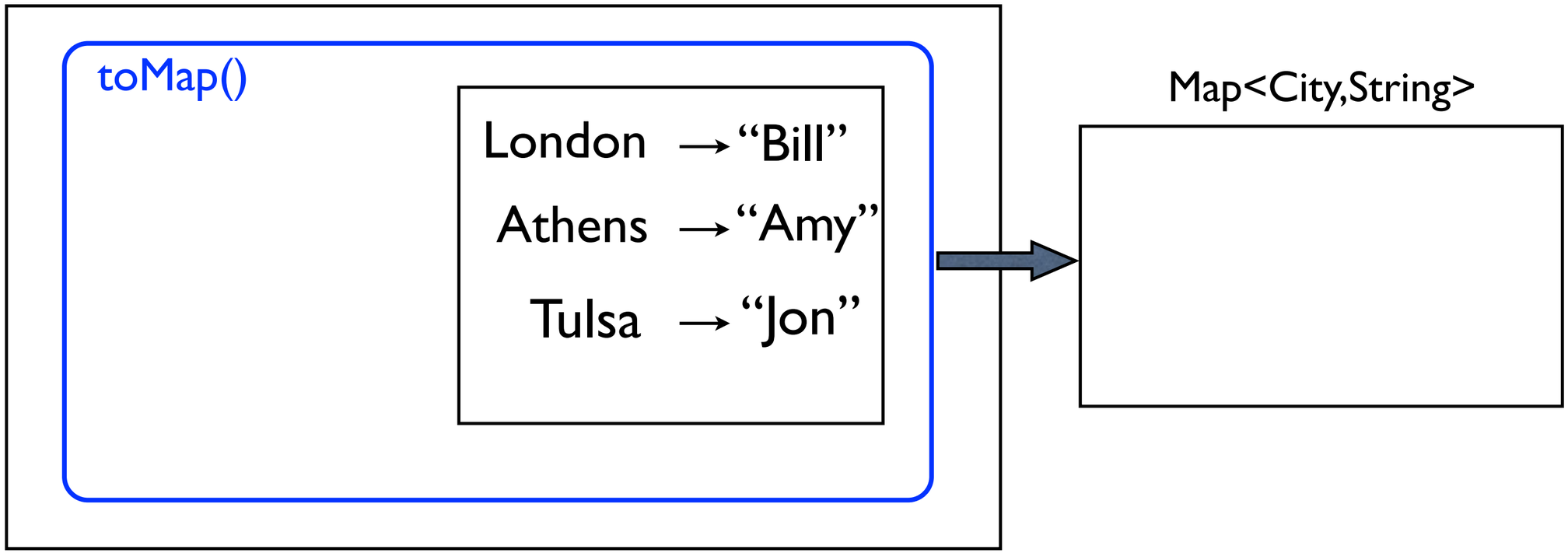
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



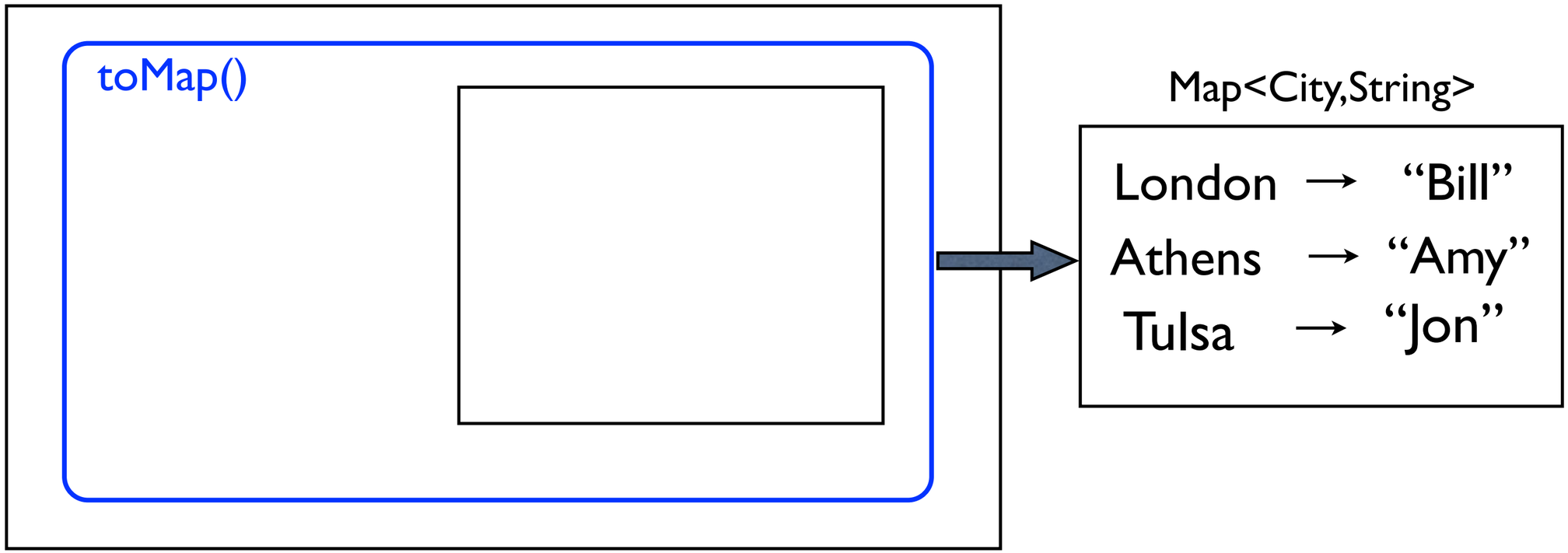
toMap(Function<T,K> keyMapper, Function<T,U> valueMapper)



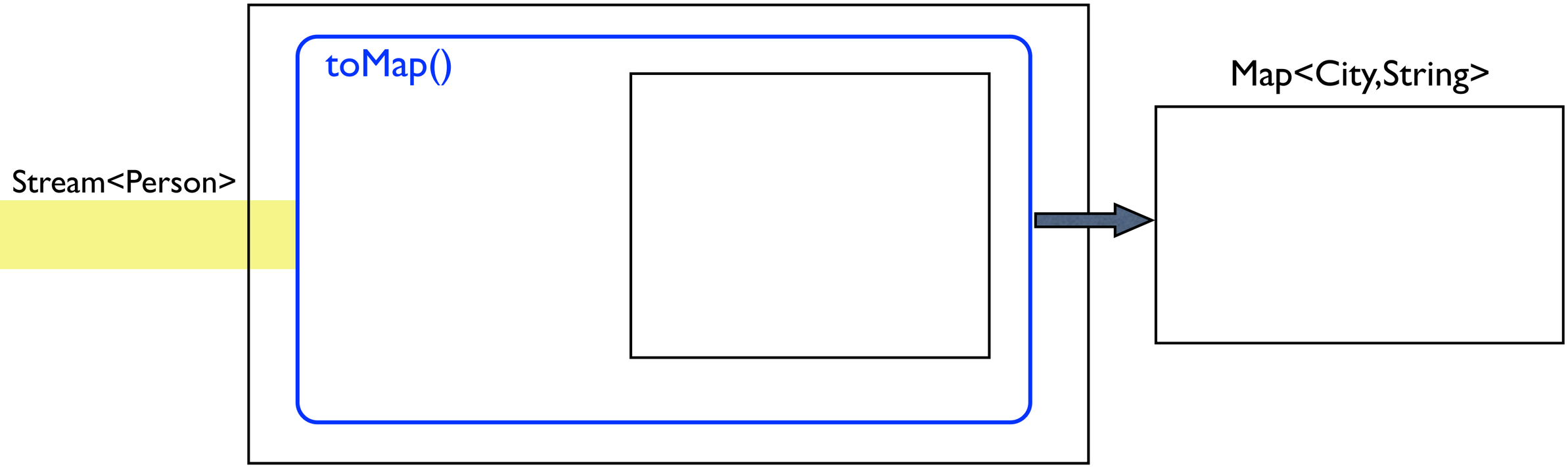
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



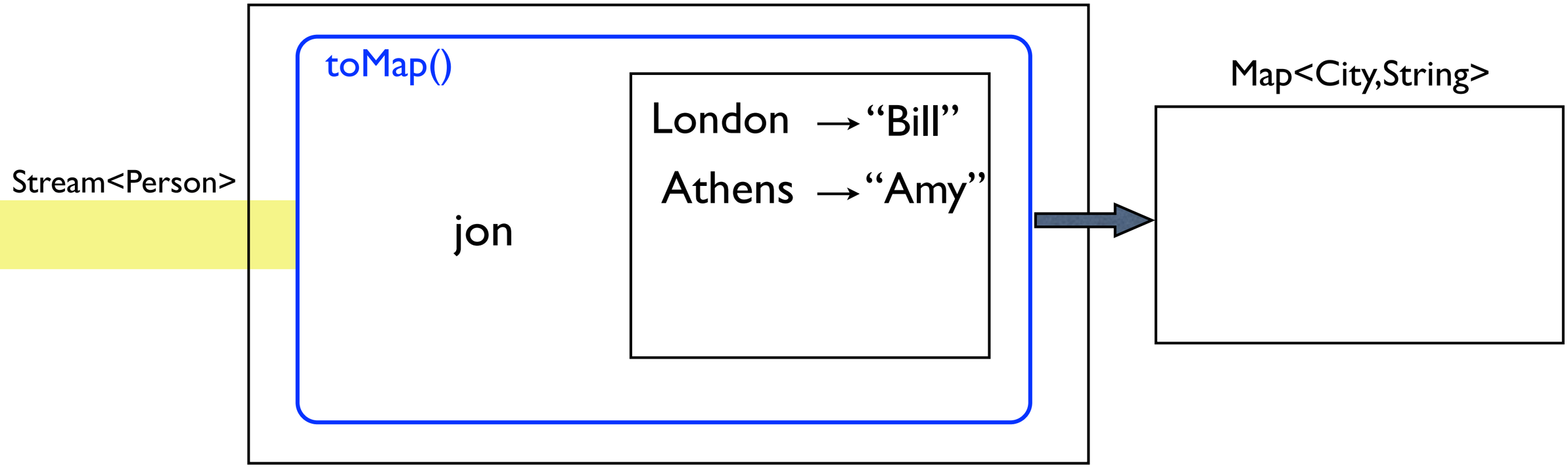
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



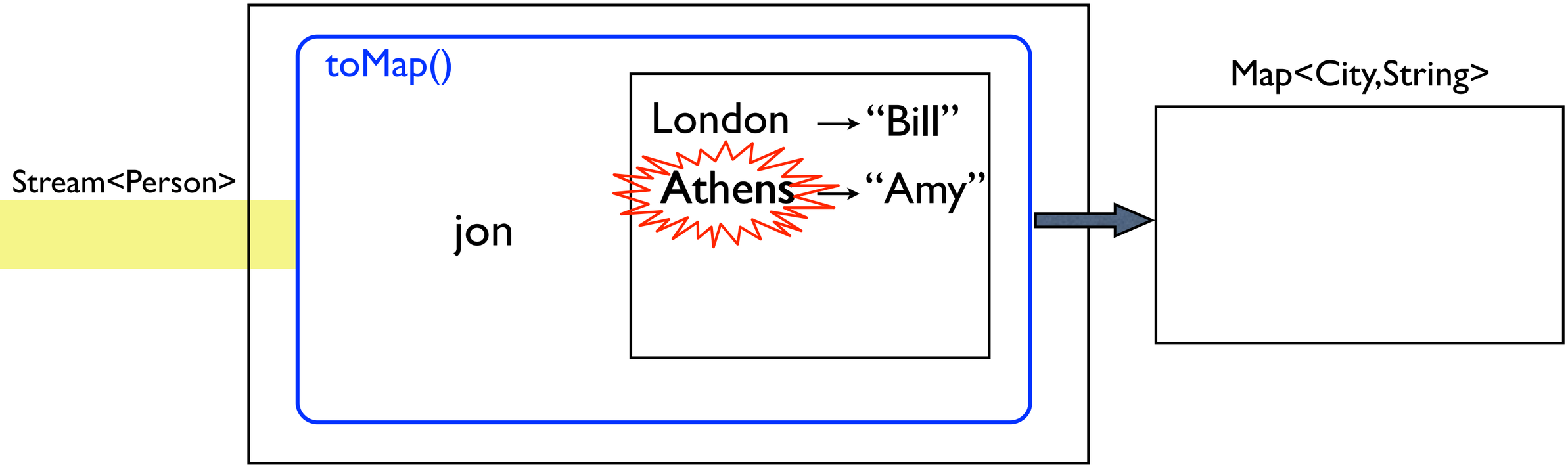
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



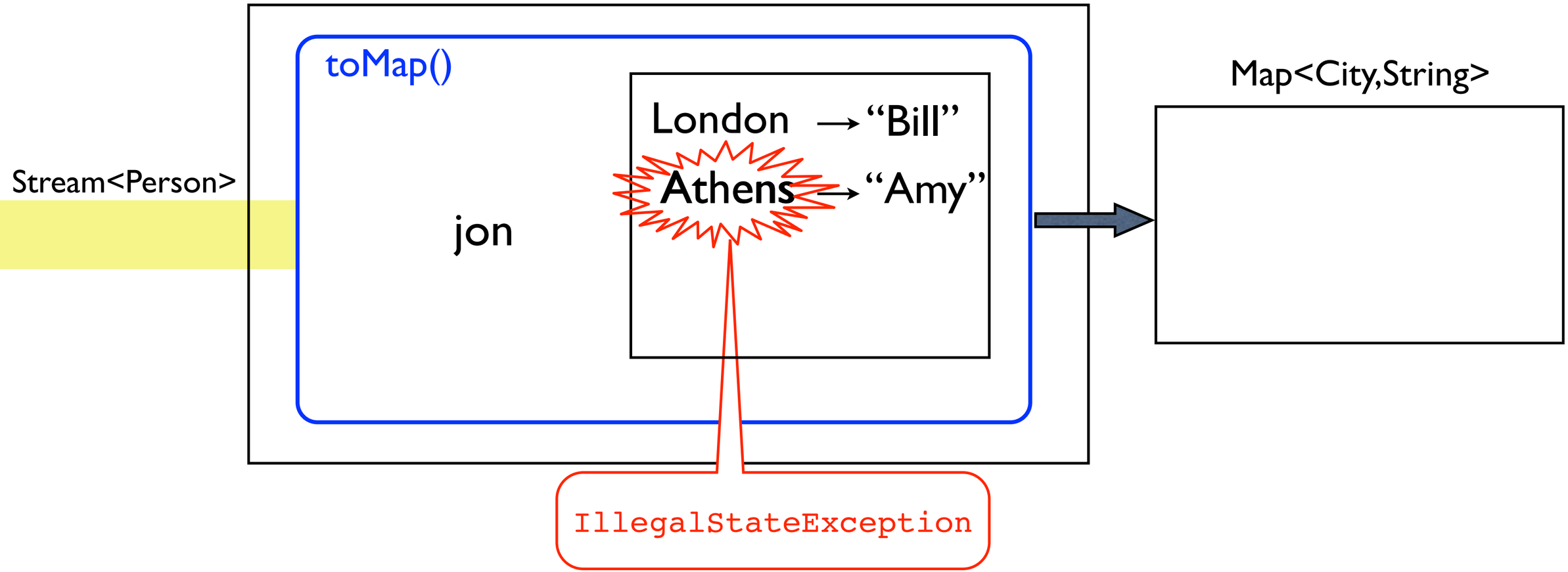
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



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



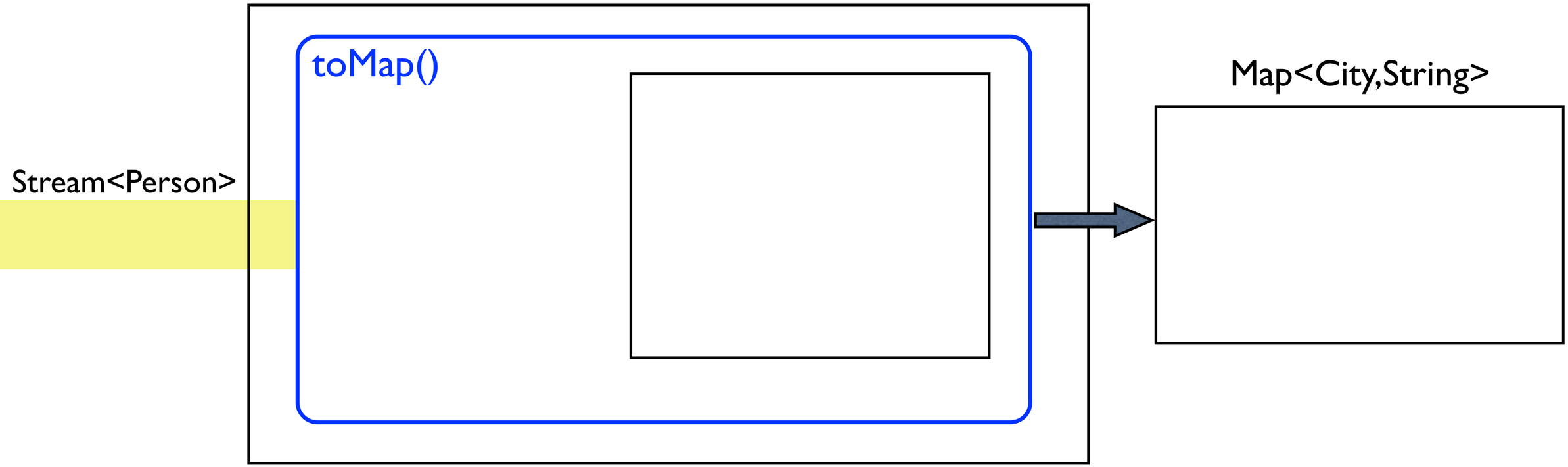
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



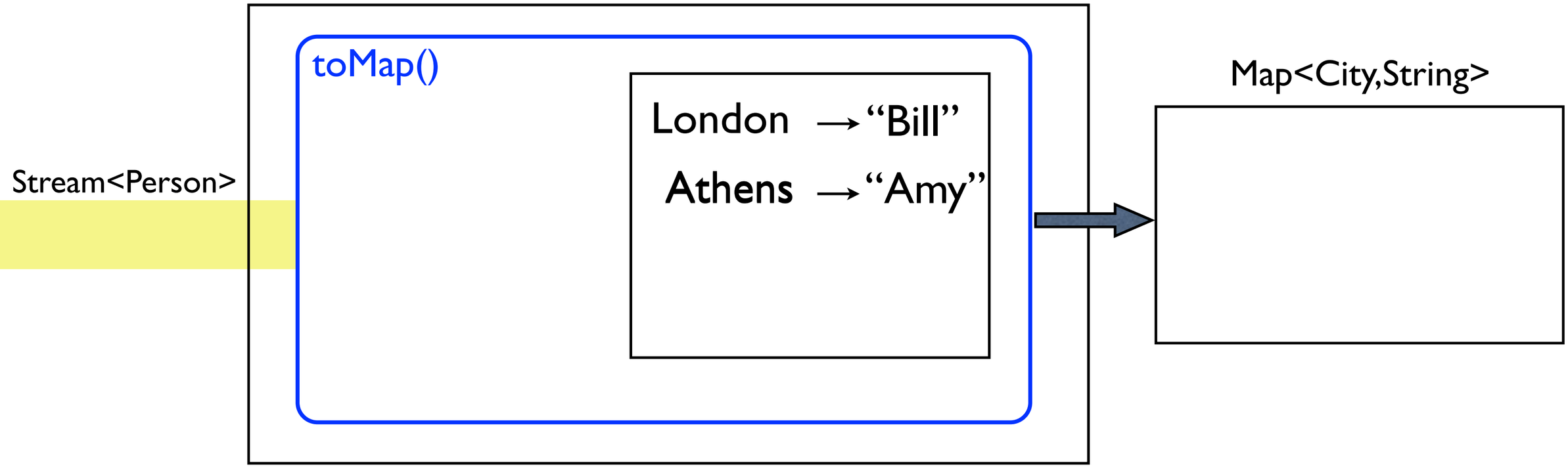
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper
BinaryOperator<U> mergeFunction)

```
people.stream().collect(
        Collectors.toMap(
                Person::getCity,
                Person::getName,
                (a,b) -> a.concat(b)))
```

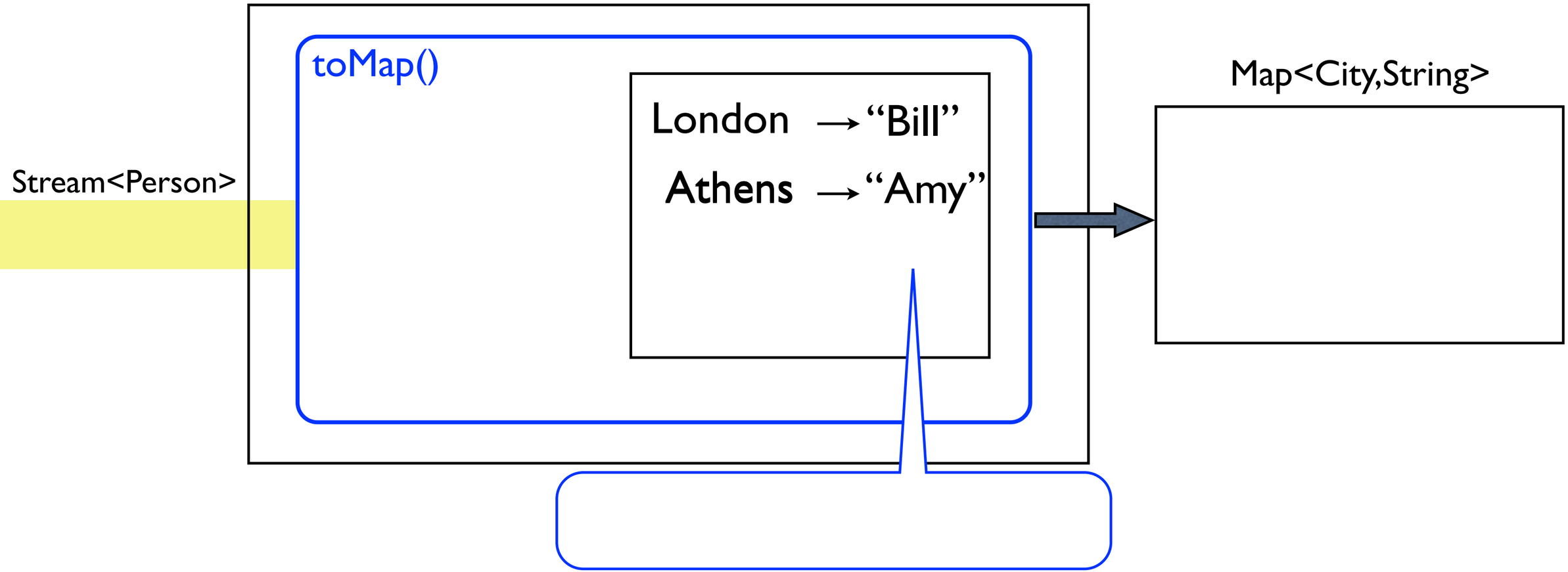
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper
BinaryOperator<U> mergeFunction)



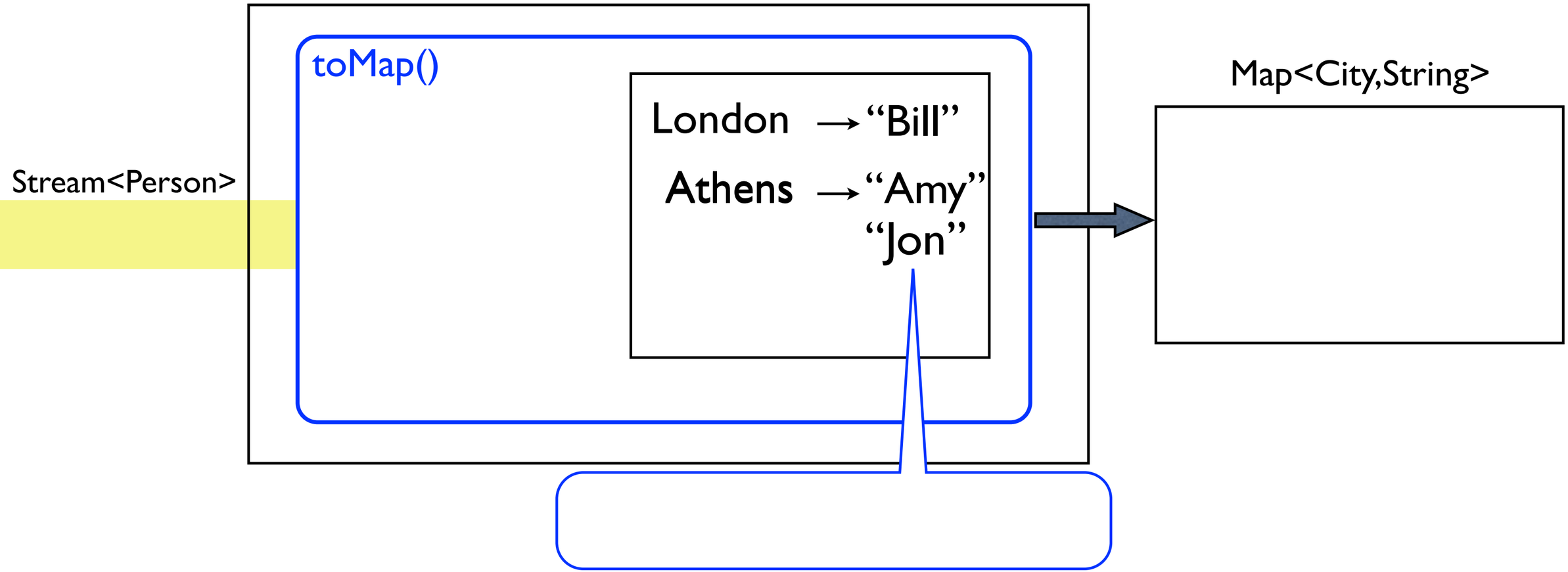
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper
BinaryOperator<U> mergeFunction)



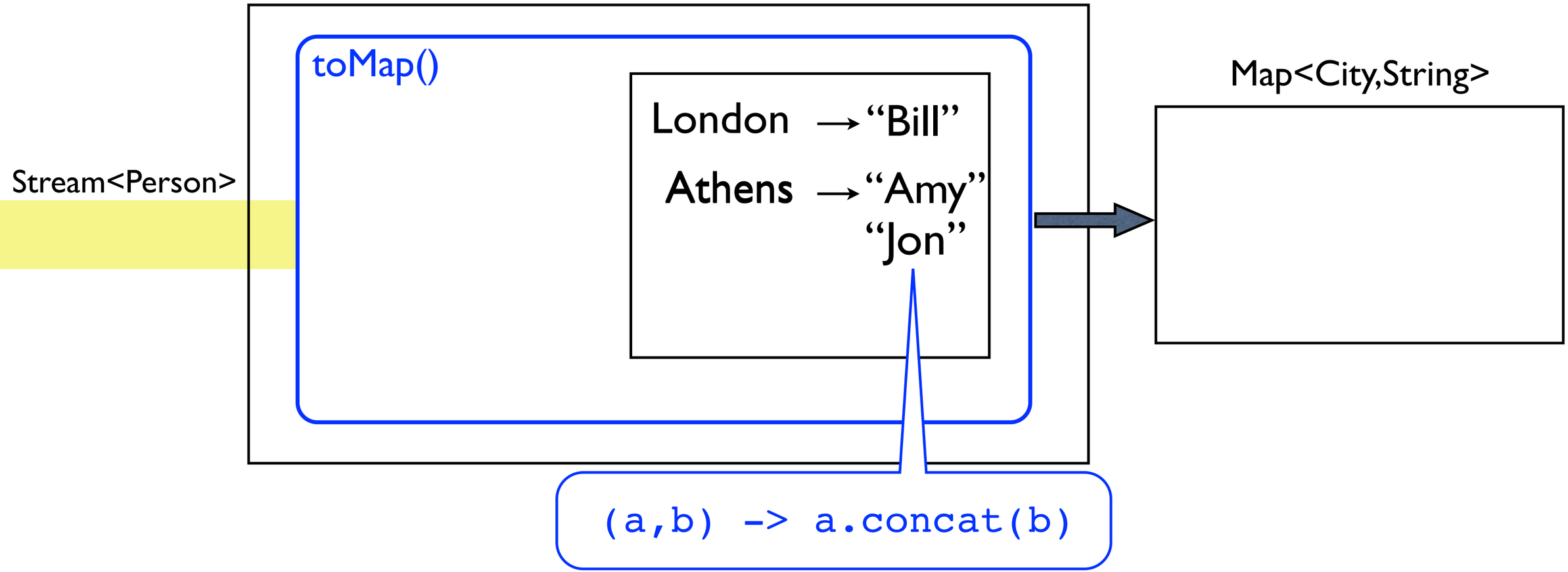
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper
BinaryOperator<U> mergeFunction)



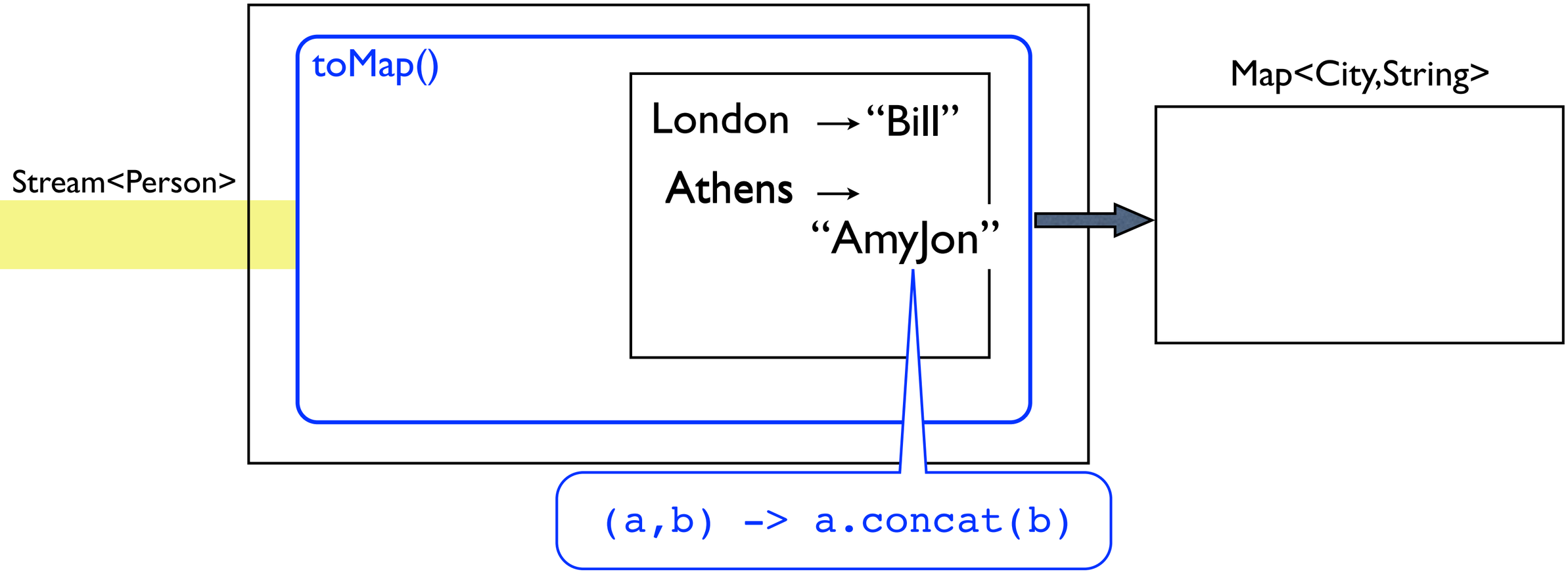
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper
BinaryOperator<U> mergeFunction)



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



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



Collectors API

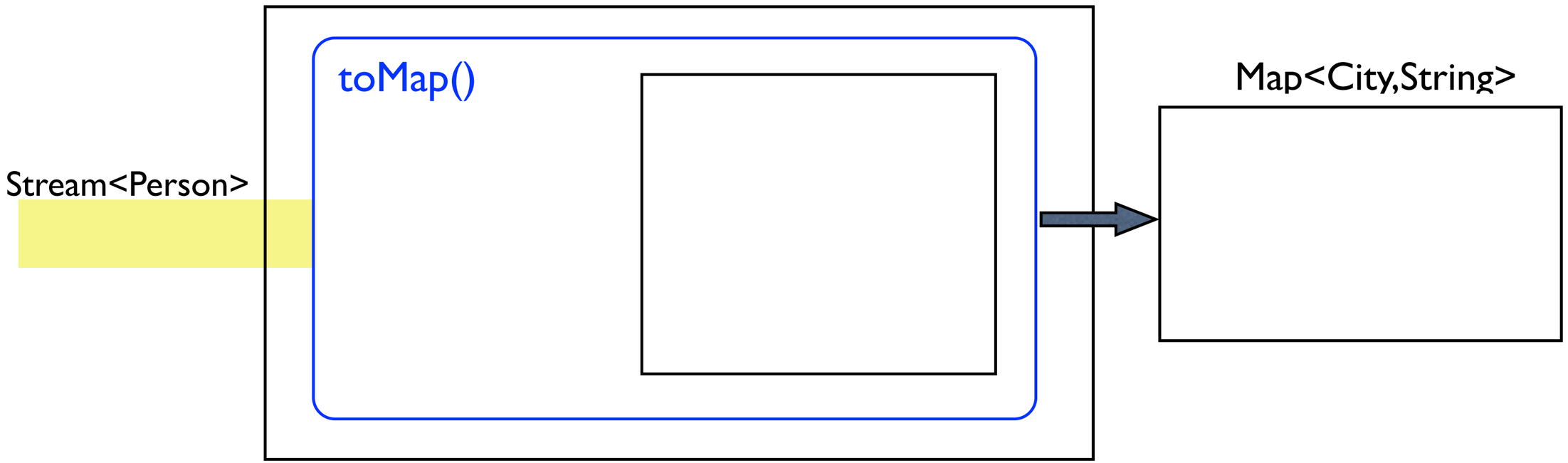
Factory methods in the Collectors class. They produce standalone collectors, accumulating to:

- framework-supplied containers;
- custom collections;
- classification maps.

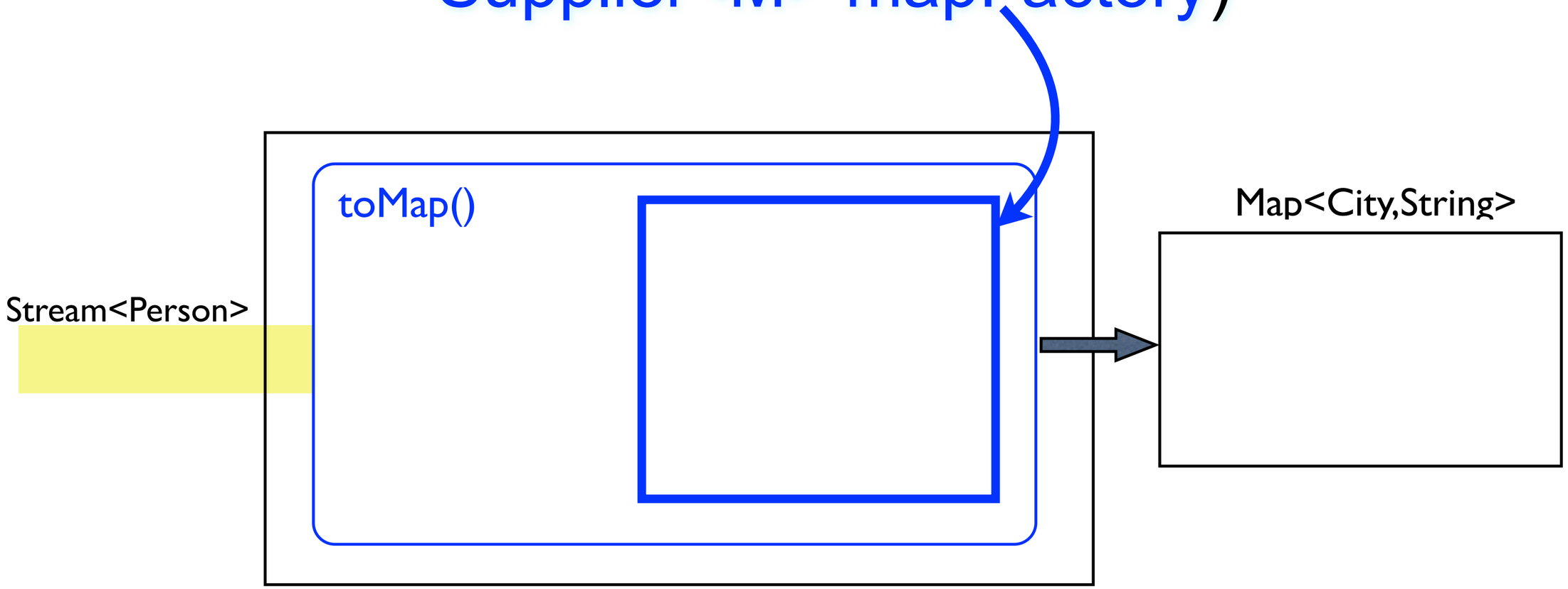
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper
BinaryOperator<U> mergeFunction,
Supplier<M> mapFactory)

```
people.stream().collect(
    Collectors.toMap(
        Person::getCity,
        Person::getName,
        (a,b) -> a.concat(b),
        TreeMap::new))
```

```
toMap(Function<T,K> keyMapper,
      Function<T,U> valueMapper
      BinaryOperator<U> mergeFunction,
      Supplier<M> mapFactory)
```



```
toMap(Function<T,K> keyMapper,
      Function<T,U> valueMapper
      BinaryOperator<U> mergeFunction,
      Supplier<M> mapFactory)
```



Collecting to Custom Collections

```
<C extends Collection<T>>
```

```
NavigableSet<String> sortedNames = people.stream()
        .map(Person::getName)
        .collect(toCollection(TreeSet::new));
```

Collecting to Custom Collections

```
<C extends Collection<T>>
toCollection(Supplier<C> collectionFactory)
```

```
NavigableSet<String> sortedNames = people.stream()
        .map(Person::getName)
        .collect(toCollection(TreeSet::new));
```

Collectors API

Factory methods in the Collectors class. They produce standalone collectors, accumulating to:

- framework-supplied containers;
- custom collections;
- classification maps.

Collecting to Classification Maps

groupingBy

- simple
- with downstream
- with downstream and map factory

partitioningBy

groupingBy(Function<T,K> classifier)

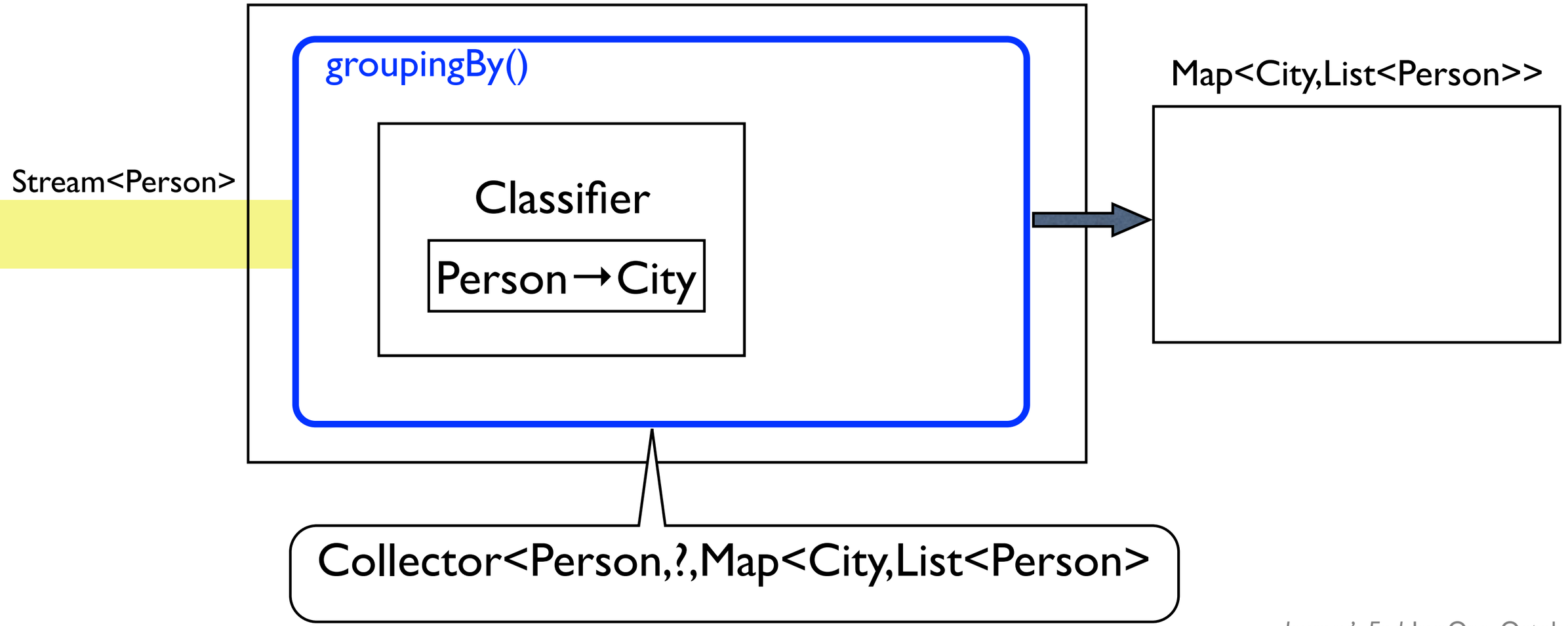
Uses the classifier function to make a *classification mapping*

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

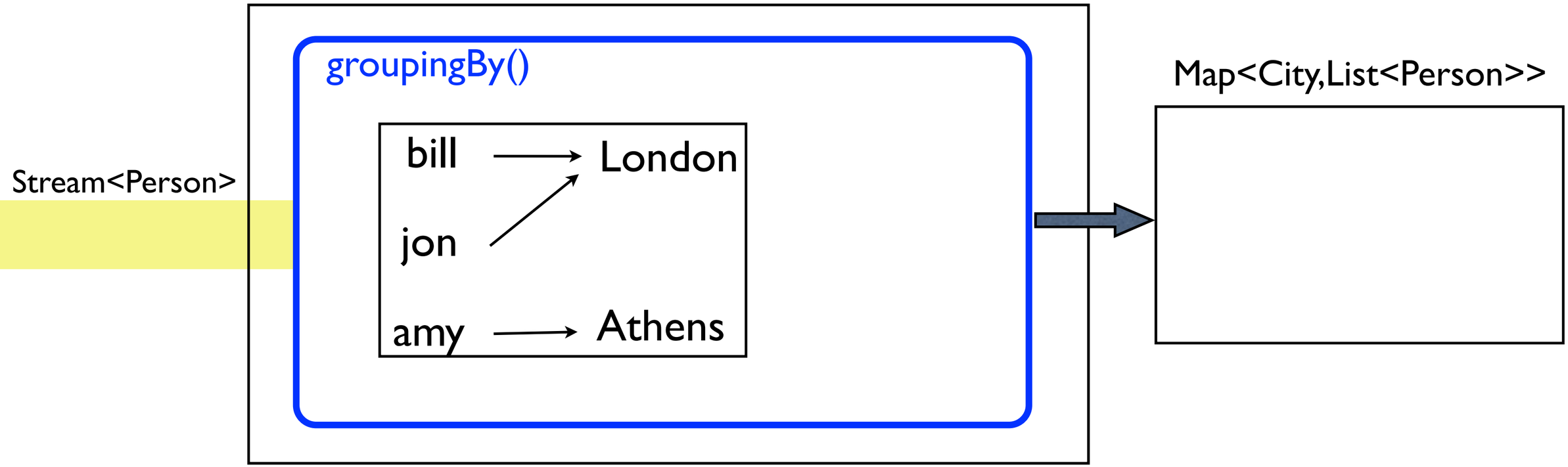
For example, use Person.getCity() to make a Map<City,List<Person>>

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

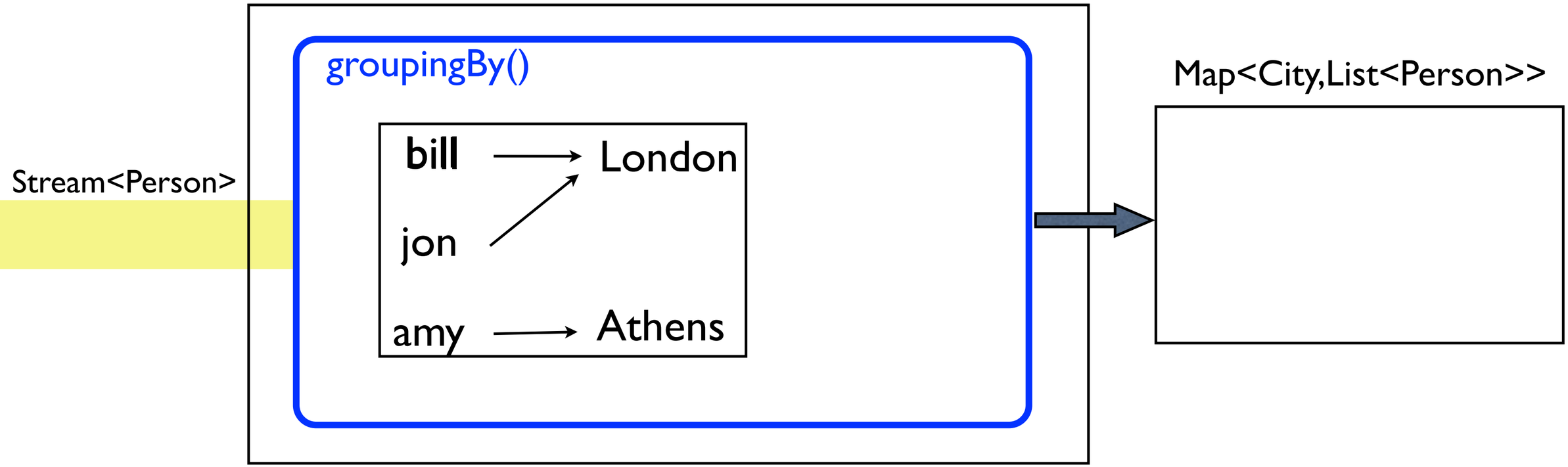
groupingBy(Function<Person,City>)



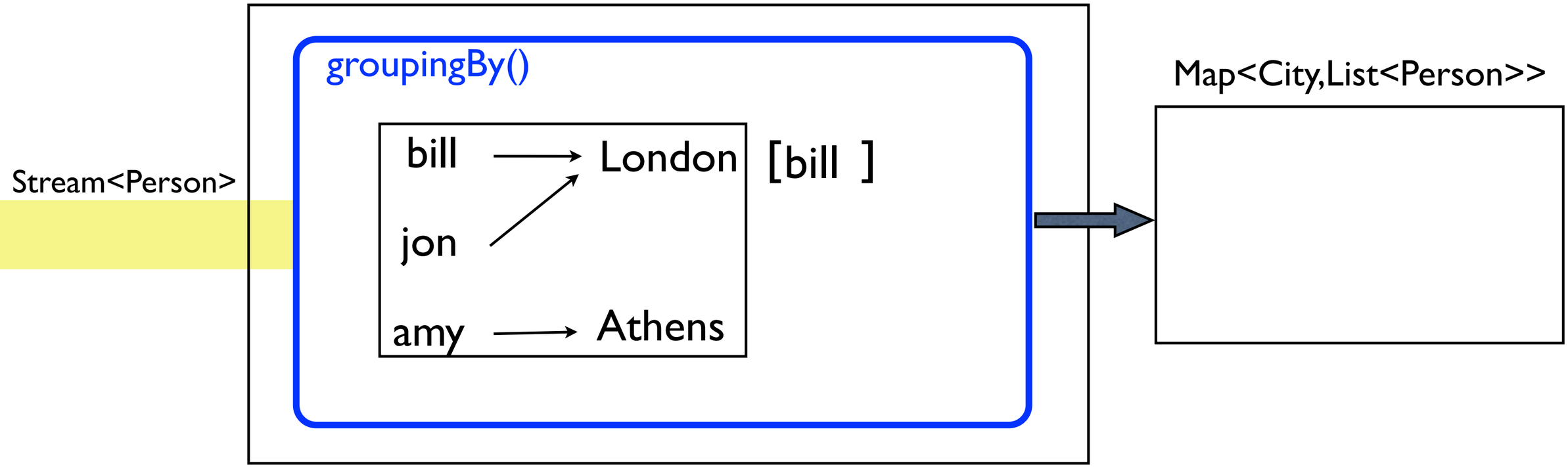
groupingBy(Function<Person,City>)



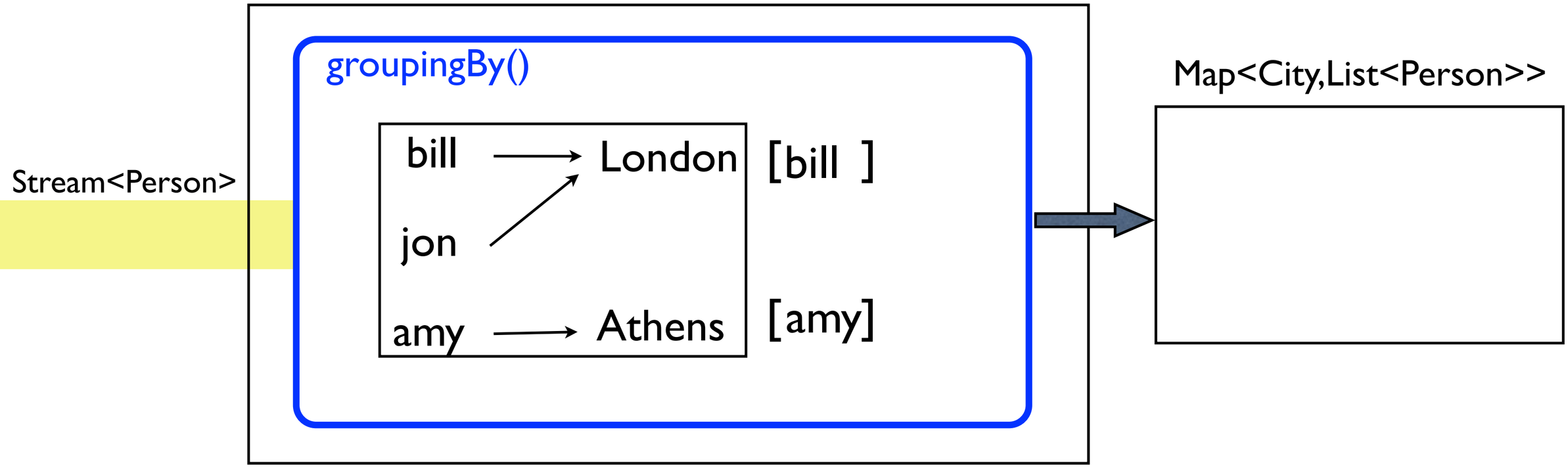
groupingBy(Function<Person,City>)



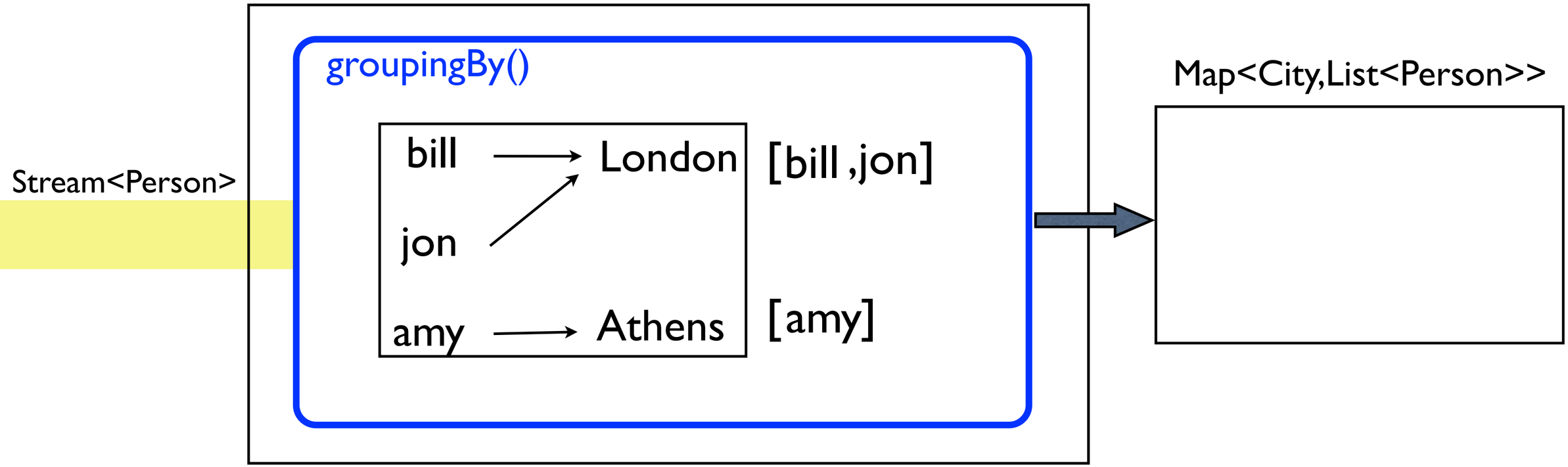
groupingBy(Function<Person,City>)



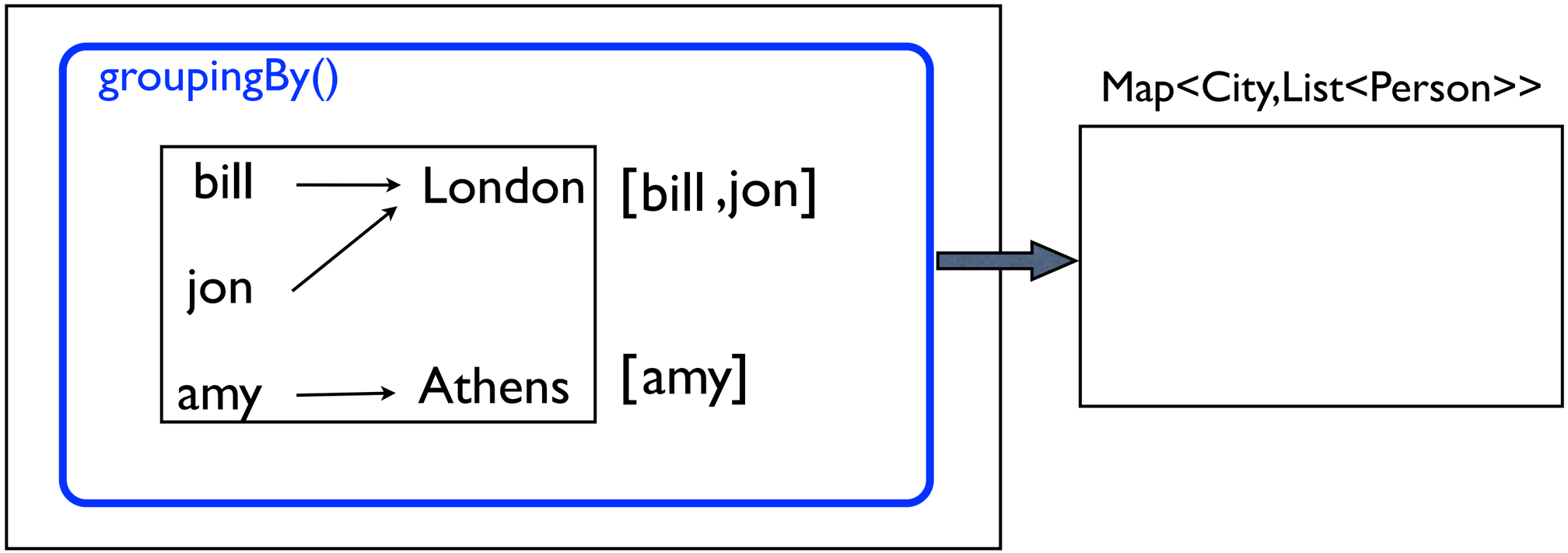
groupingBy(Function<Person,City>)



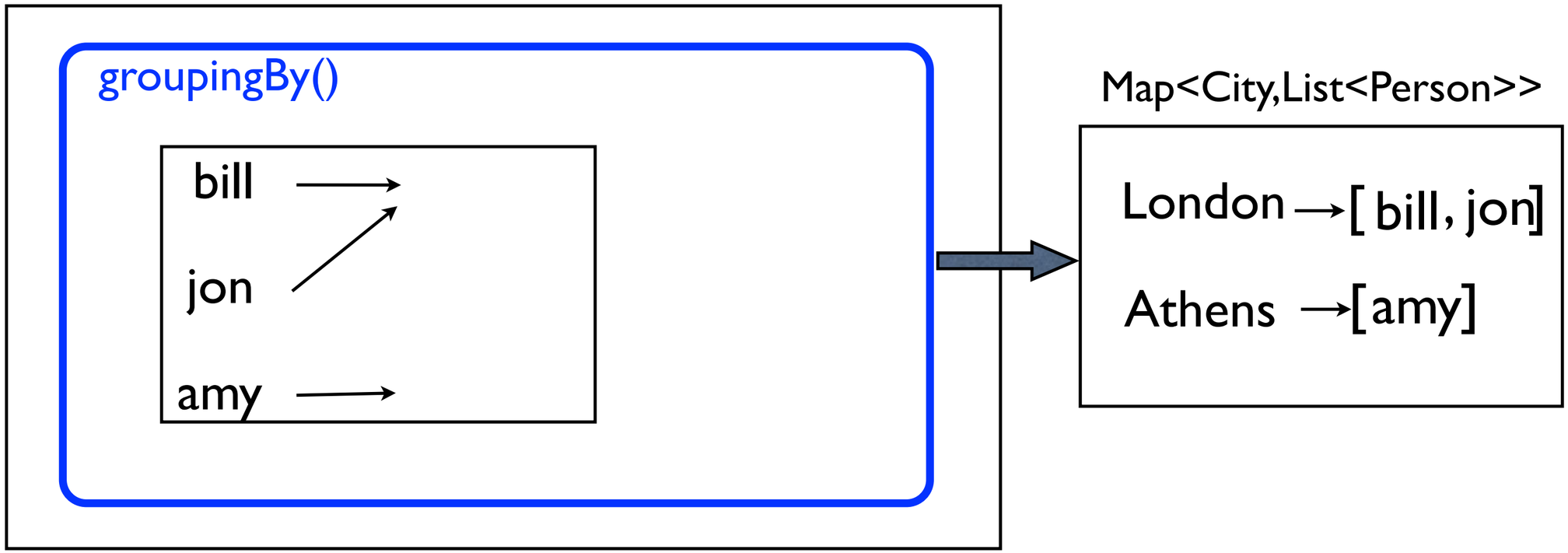
groupingBy(Function<Person,City>)



groupingBy(Function<Person,City>)



groupingBy(Function<Person,City>)



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(classifier, downstream)

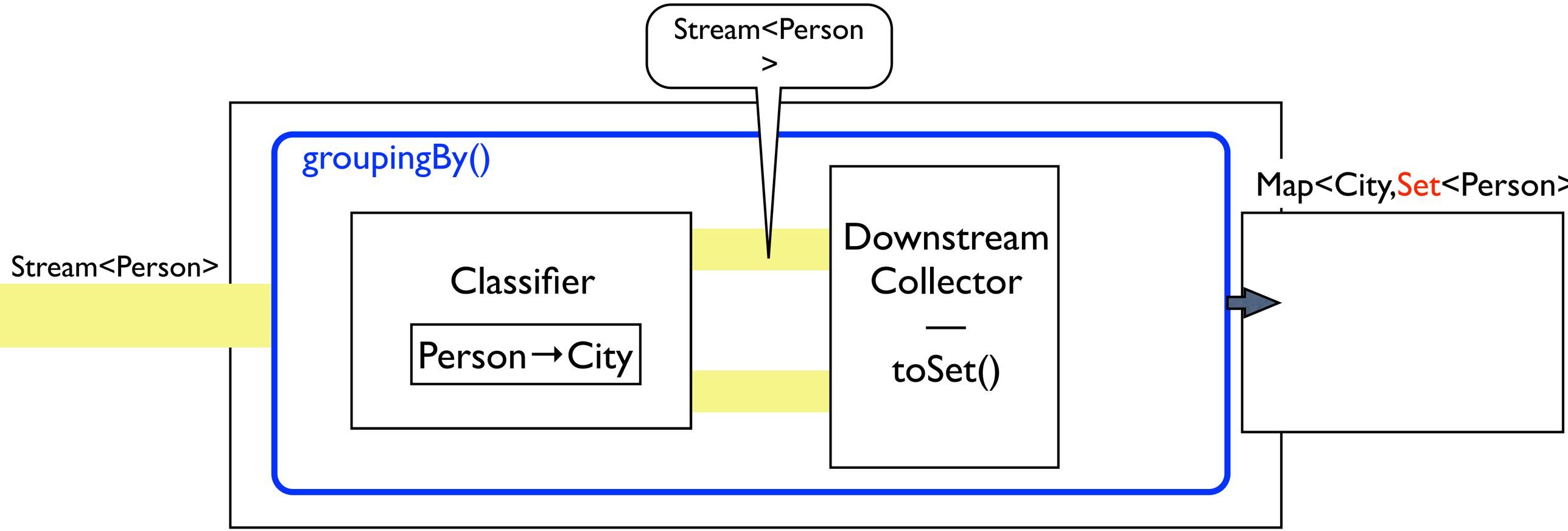
Uses the classifier function to make a *classification mapping* into a container defined by a downstream Collector

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

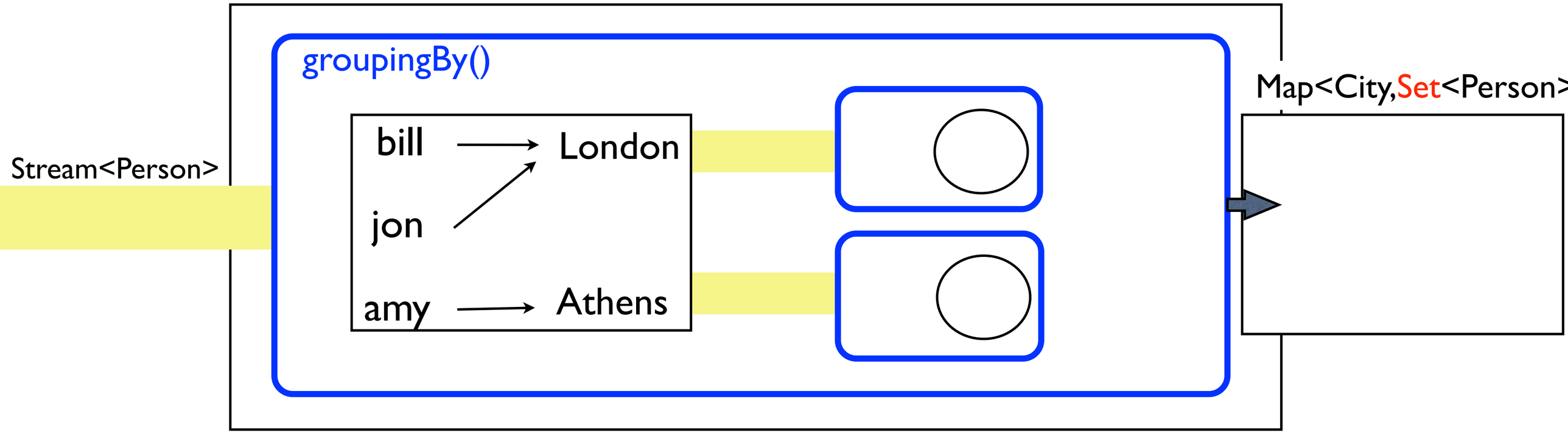
For example, use Person.getCity() to make a Map<City,Set<Person>>

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

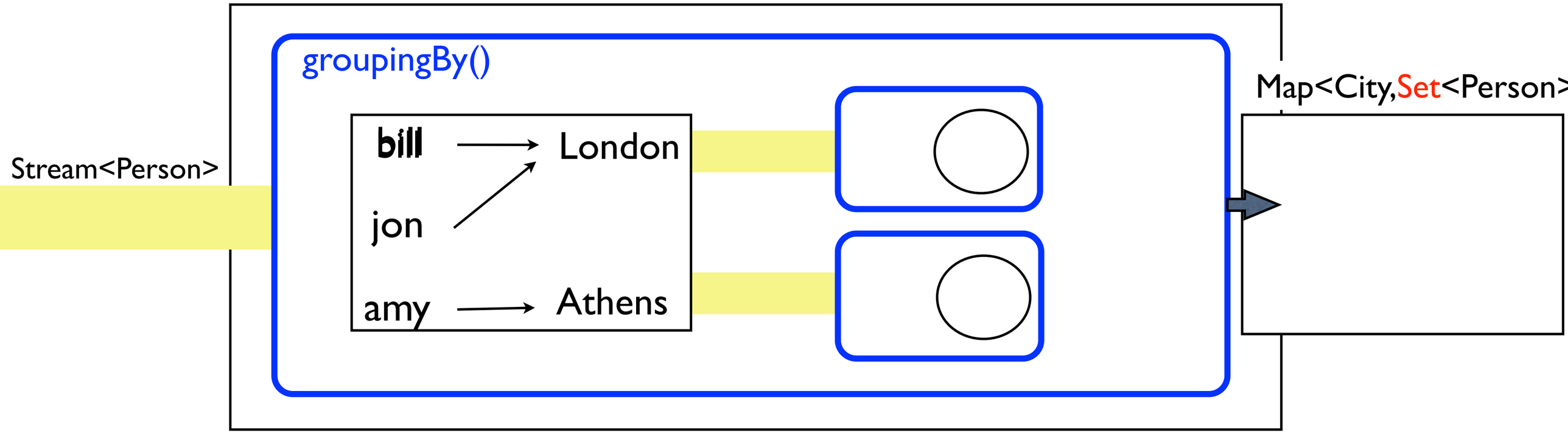
groupingBy(Function classifier,Collector downstream))



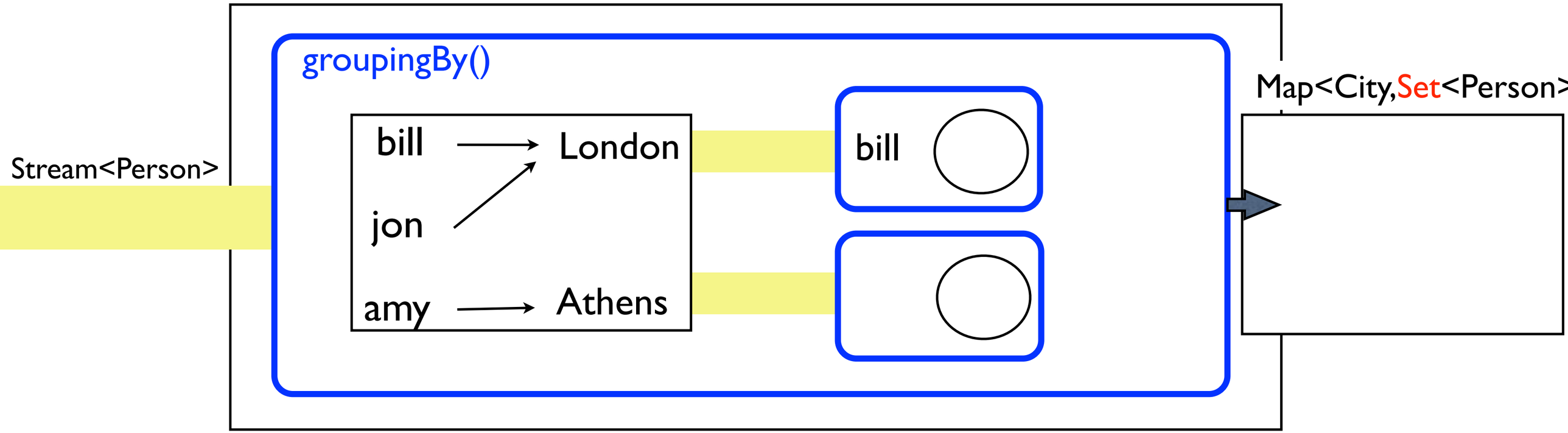
groupingBy(Function classifier,Collector downstream))



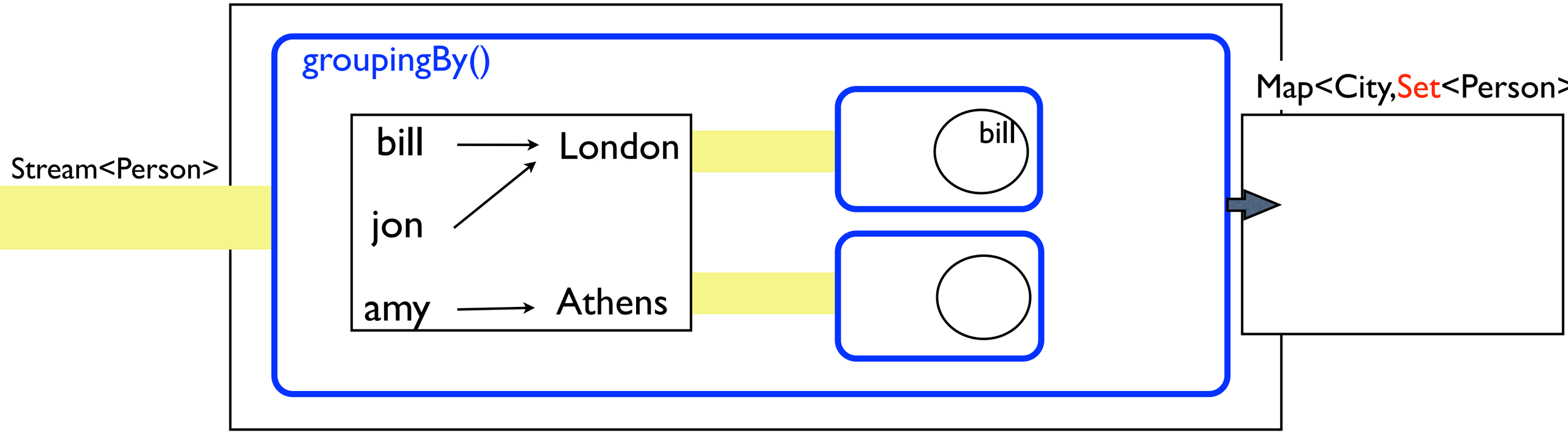
groupingBy(Function classifier,Collector downstream))



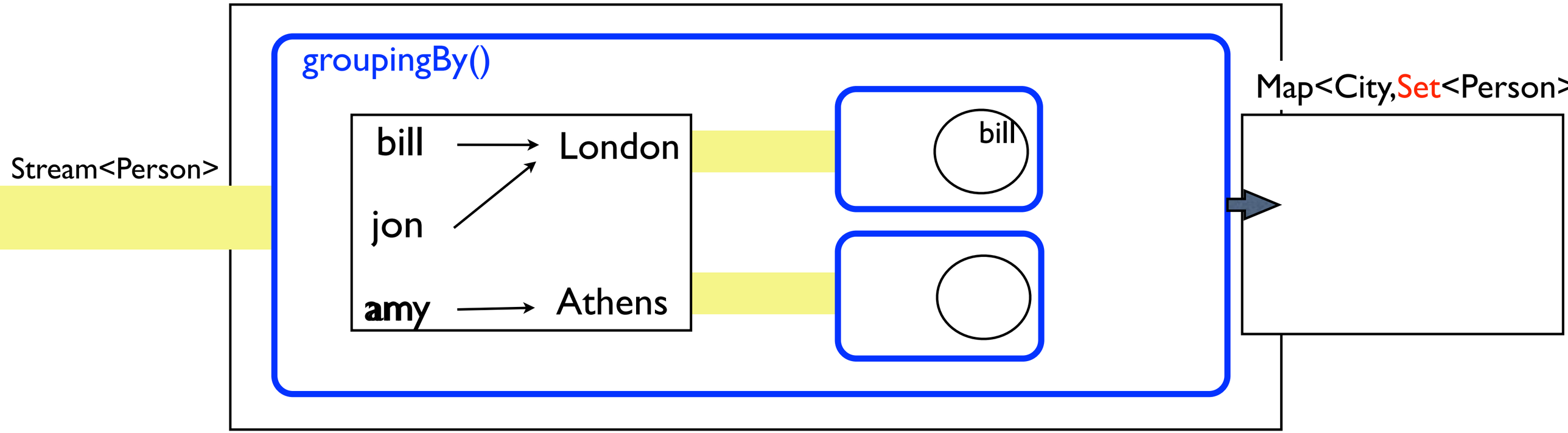
groupingBy(Function classifier,Collector downstream))



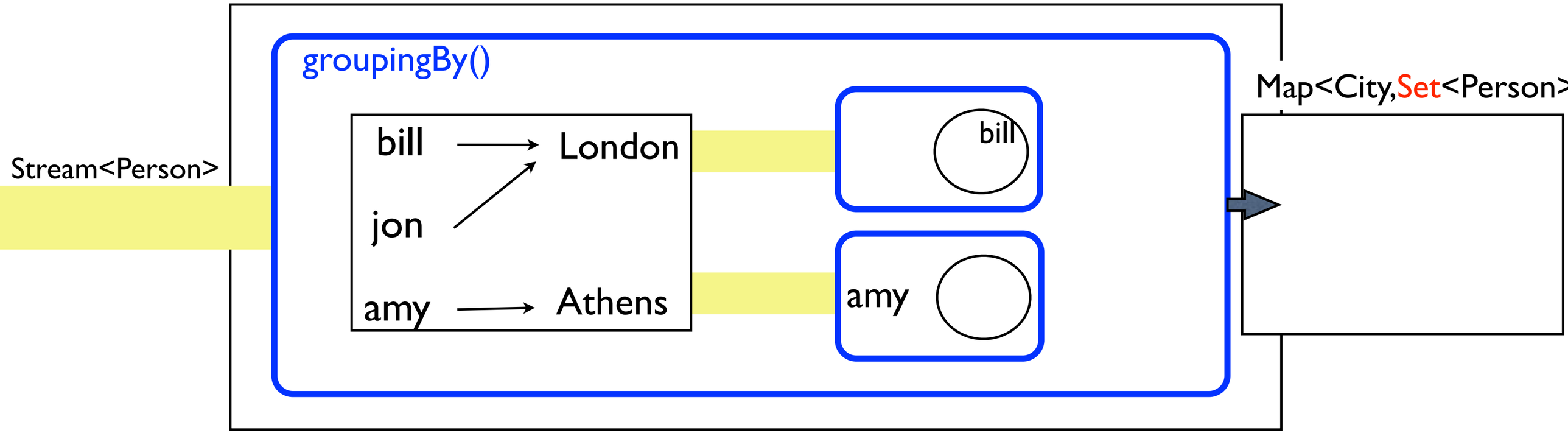
groupingBy(Function classifier,Collector downstream))



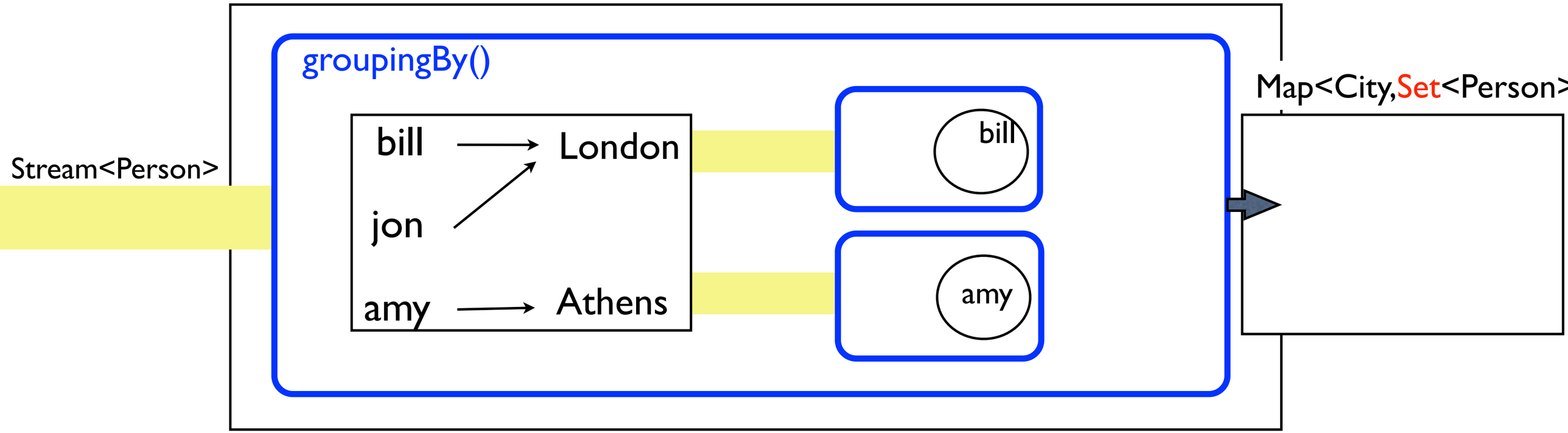
groupingBy(Function classifier,Collector downstream))



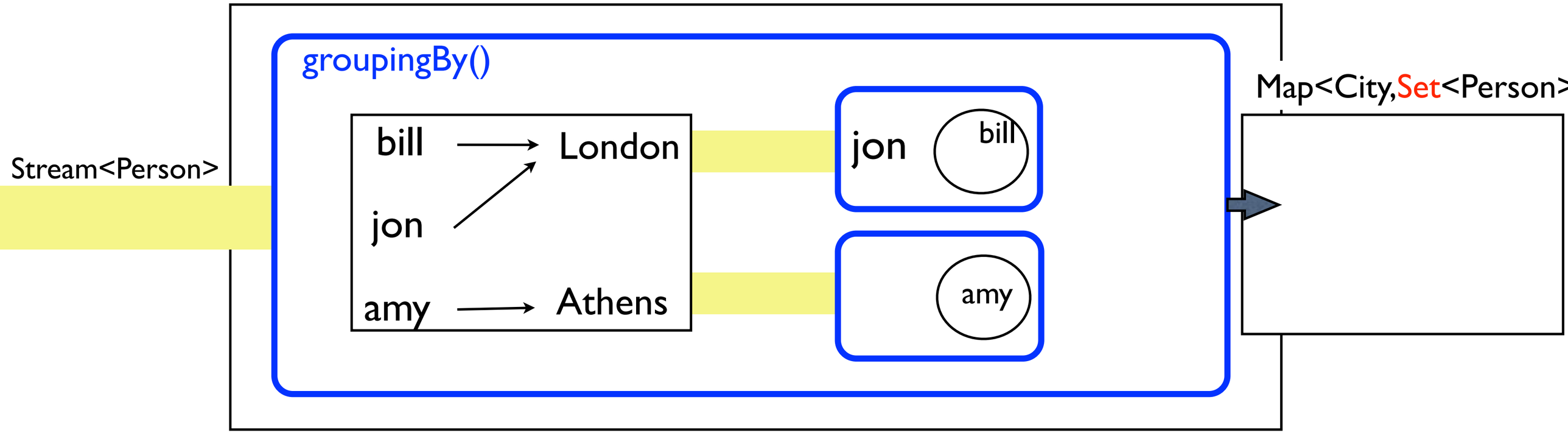
groupingBy(Function classifier,Collector downstream))



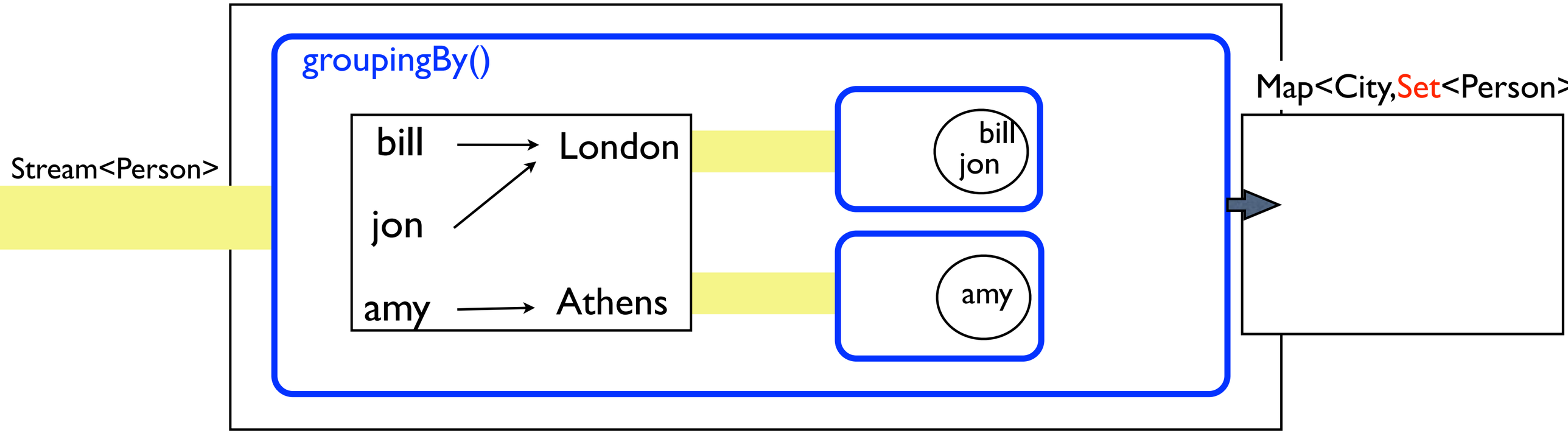
groupingBy(Function classifier,Collector downstream))



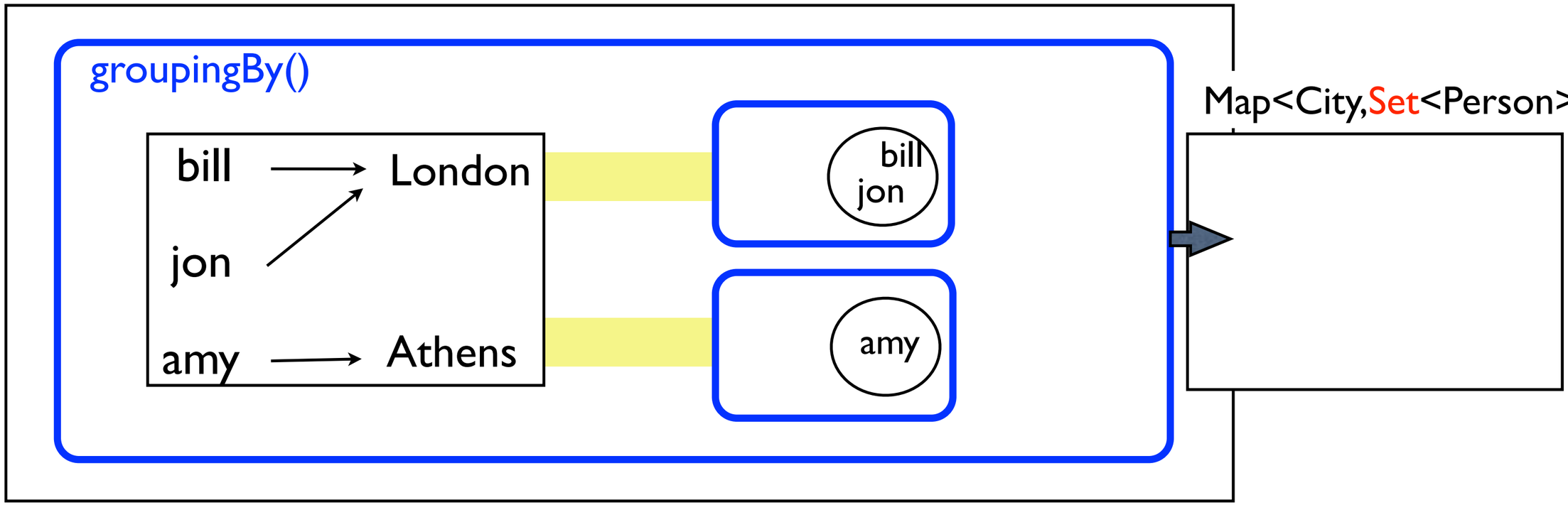
groupingBy(Function classifier,Collector downstream))



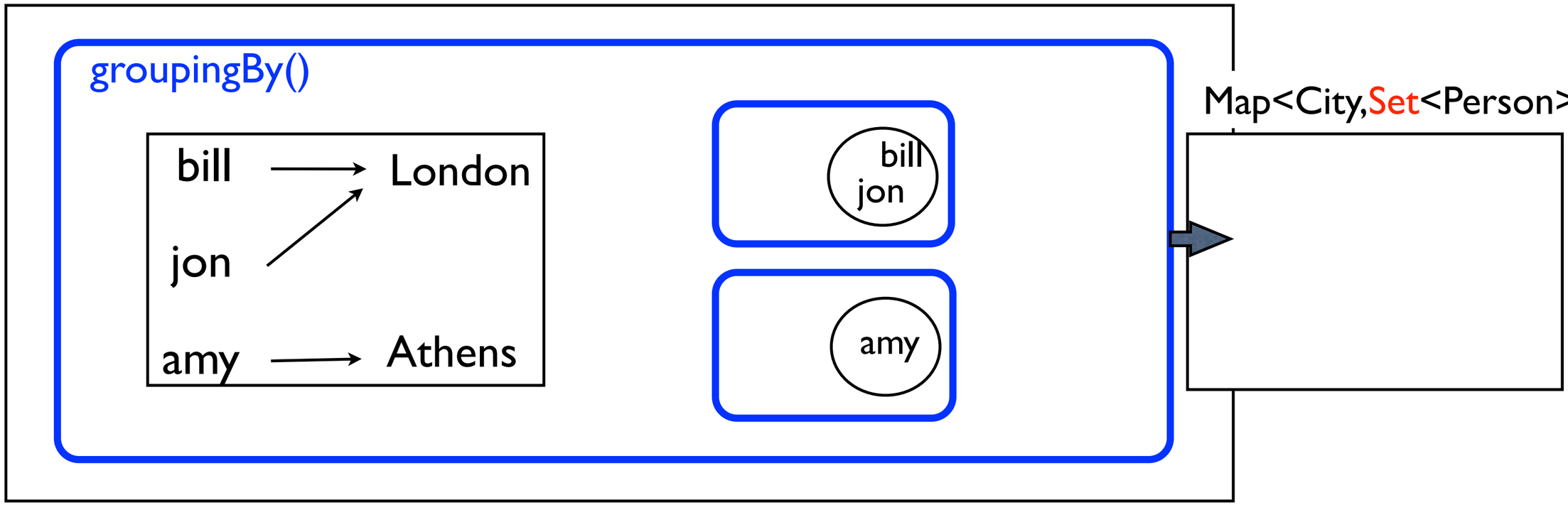
groupingBy(Function classifier,Collector downstream))



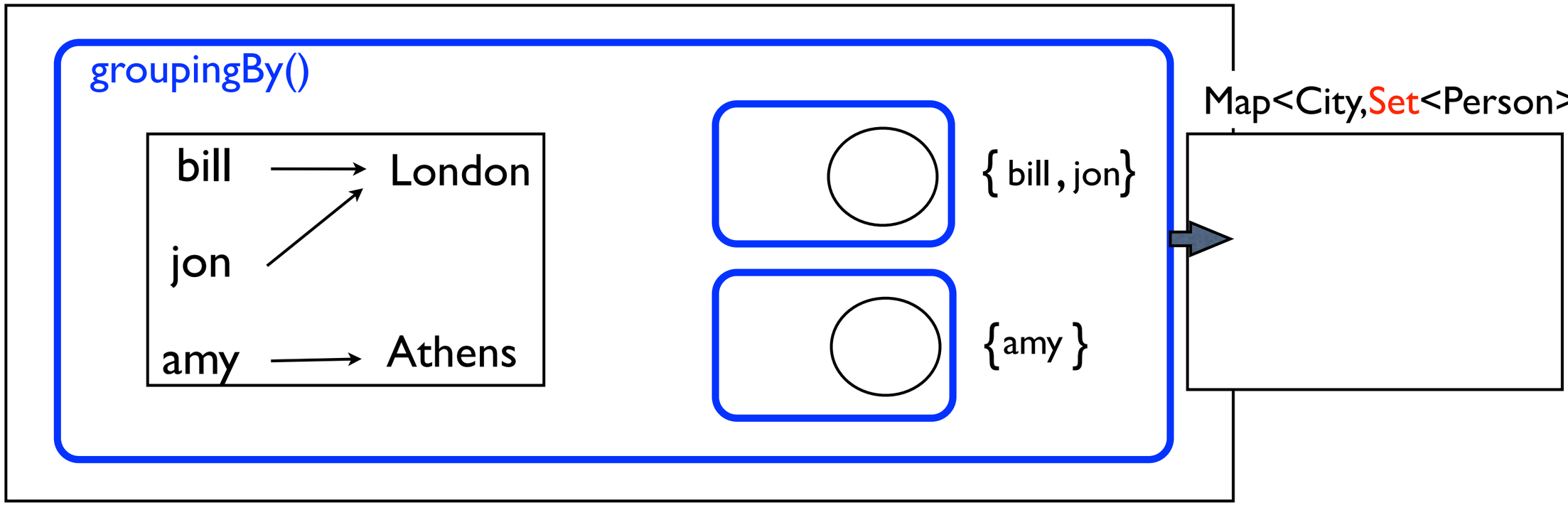
groupingBy(Function classifier,Collector downstream))



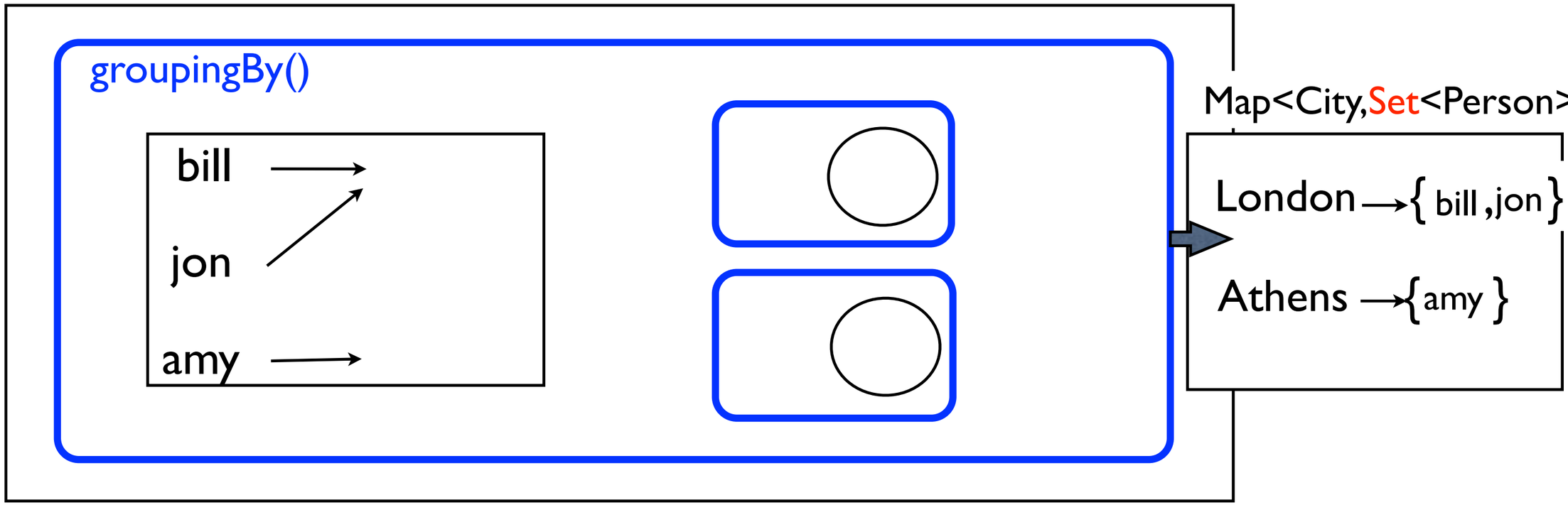
groupingBy(Function classifier,Collector downstream))



groupingBy(Function classifier,Collector downstream))



groupingBy(Function classifier,Collector downstream))



mapping(Function mapper,Collector downstream))

Applies a mapping function to each input element before passing it to the downstream collector.

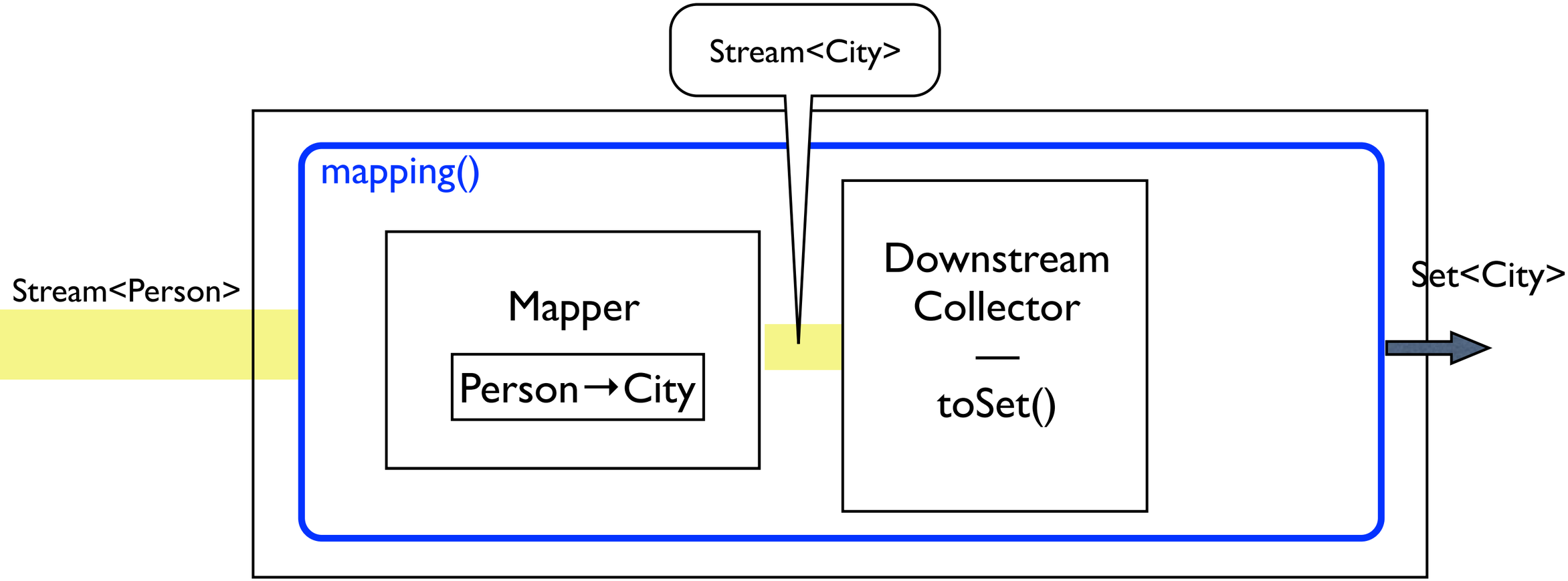
Animation example

```
Set<City> inhabited = people.stream()
   .collect(mapping(Person::getCity,toSet()));
```

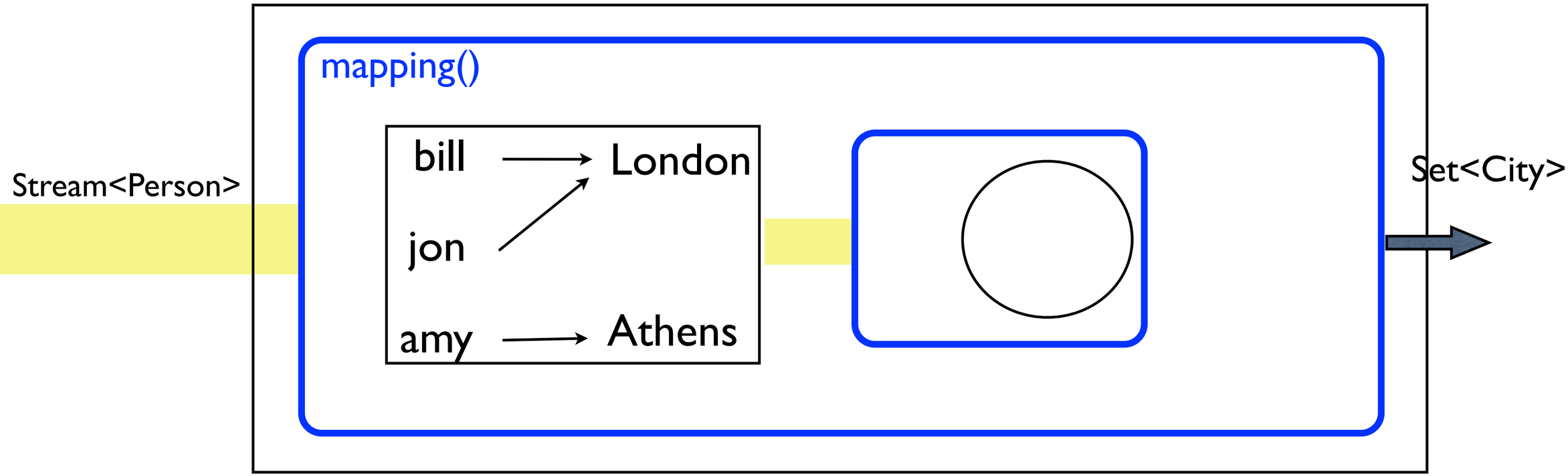
Sensible example

```
Map<City,String> namesByCity = people.stream()
   .collect(groupingBy(Person::getCity,
                       mapping(Person::getName,joining())));
```

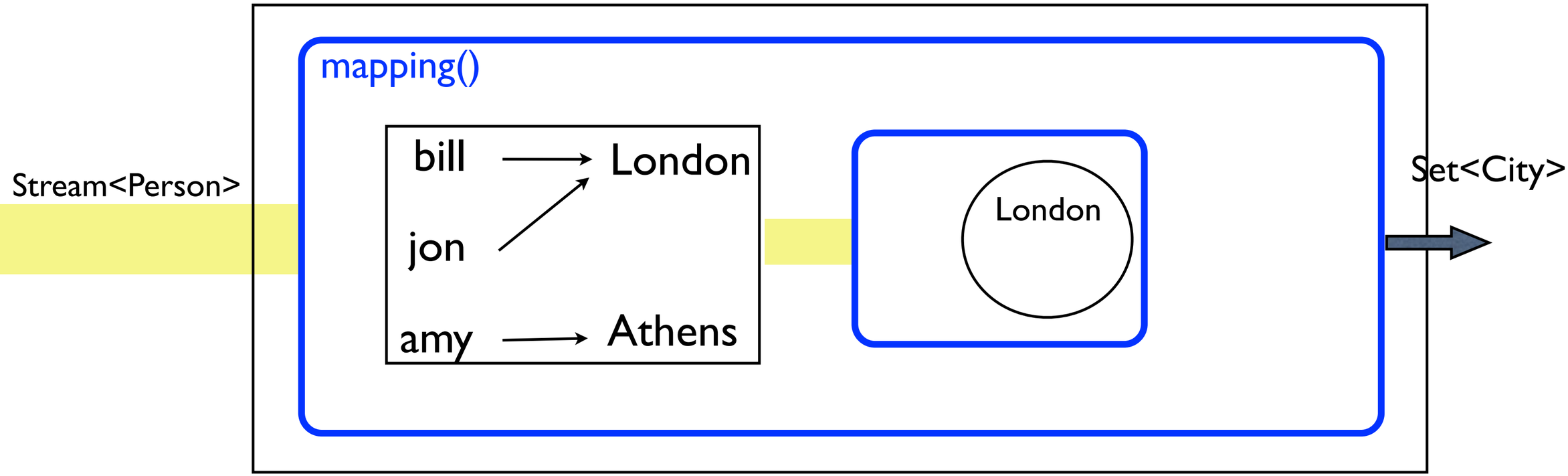
mapping(Function mapper,Collector downstream))



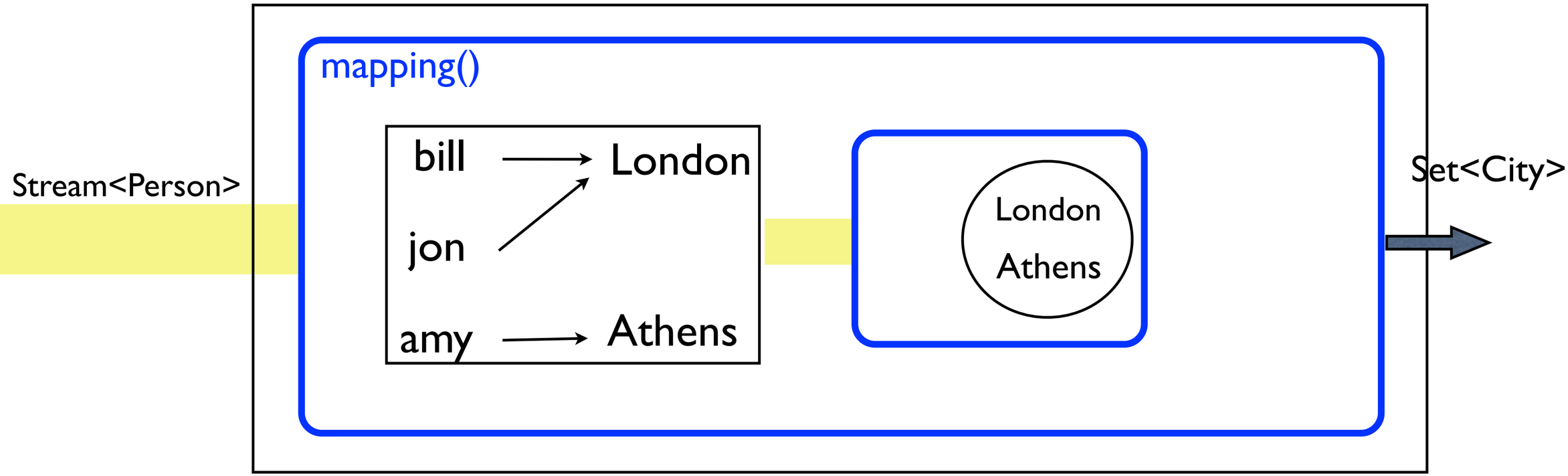
mapping(Function mapper,Collector downstream))



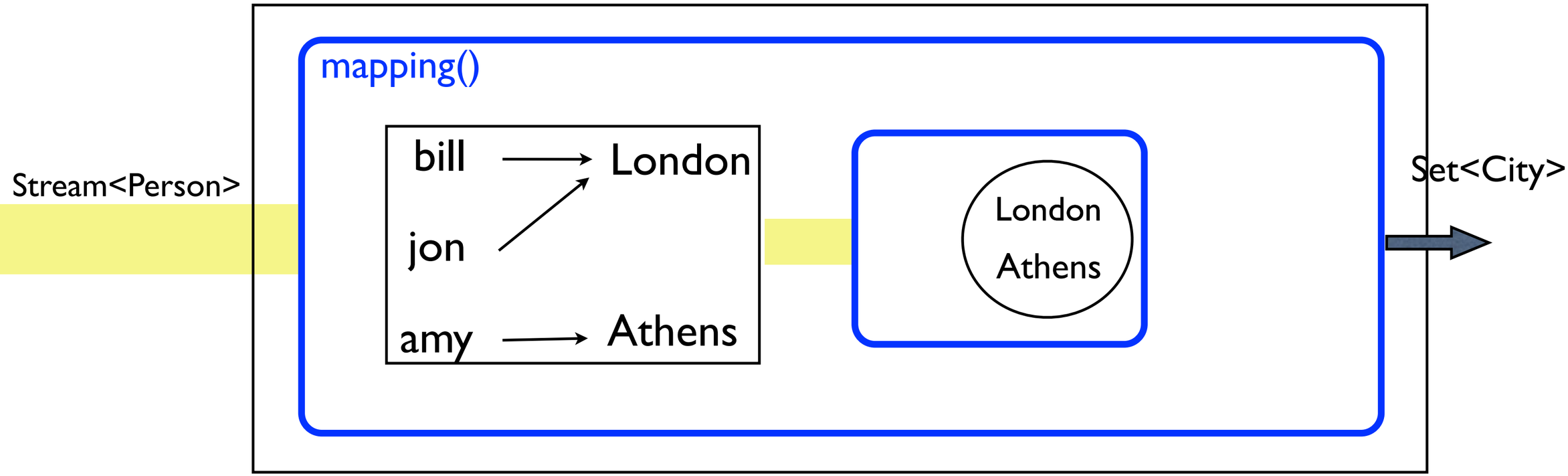
mapping(Function mapper,Collector downstream))



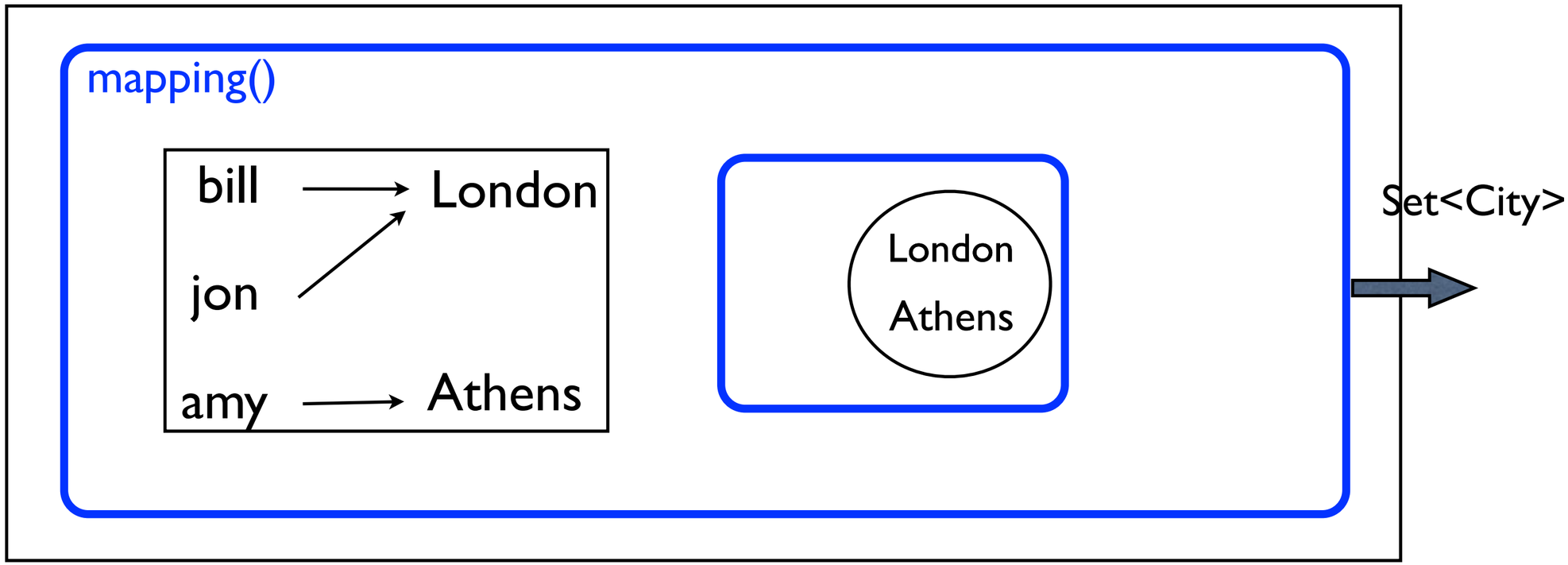
mapping(Function mapper,Collector downstream))



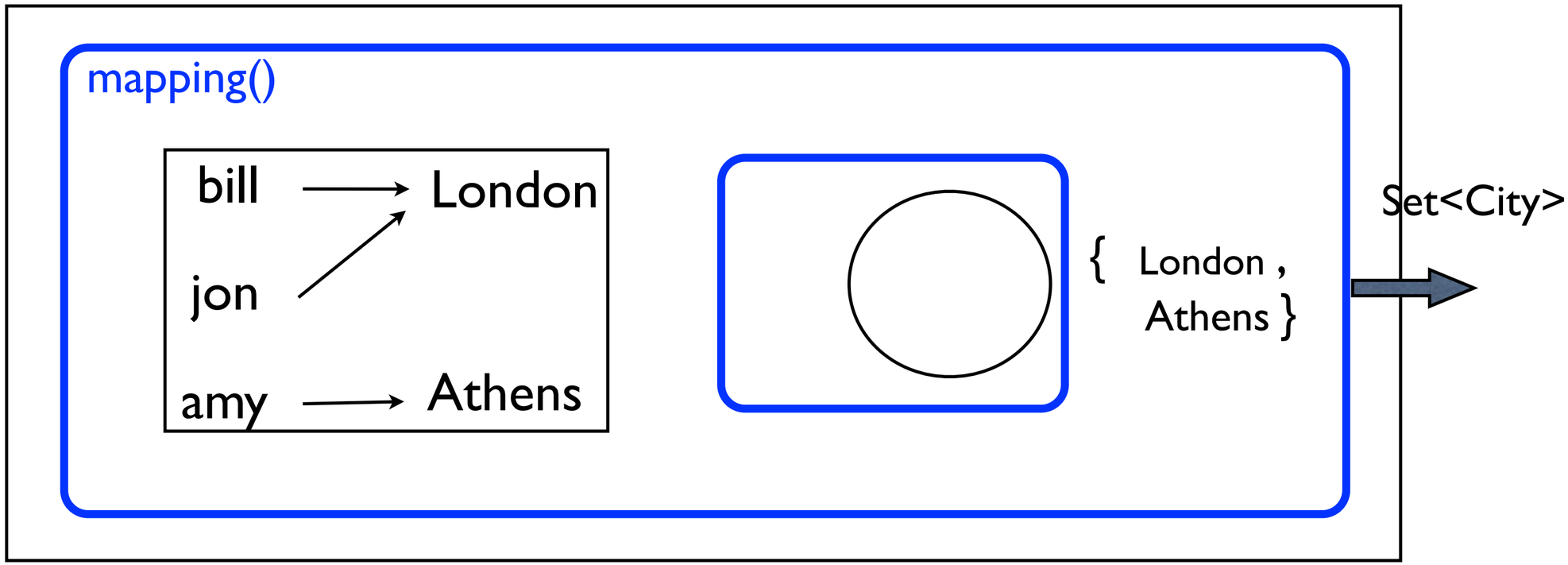
mapping(Function mapper,Collector downstream))



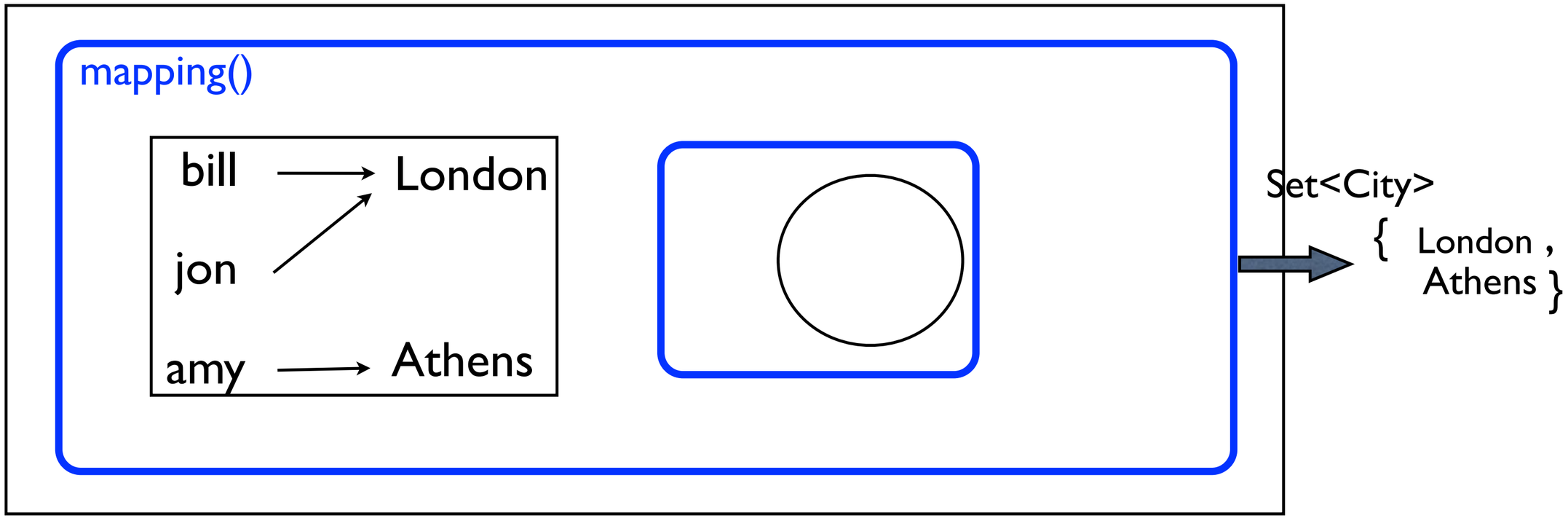
mapping(Function mapper,Collector downstream))



mapping(Function mapper,Collector downstream))



mapping(Function mapper,Collector downstream))



Dualling Convenience Reductions

- Collectors.counting()
  - Collectors.maxBy
  - Collectors.minBy
- Collectors.summarizingXXX

Concurrent Collection

Concurrent Collection

Thread safety is guaranteed by the framework

- Even for non-threadsafe containers!

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?

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, for example?

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, for example?

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

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

Journey's End – Agenda

- Why collectors?
- Using the predefined collectors
  - Intermission!
- Worked problems
- Writing your own

Journey's End – Agenda

- Why collectors?
- Using the predefined collectors
  - Intermission!
- Worked problems
- Writing your own

Writing a Collector

Why would you want to?

Writing a Collector

Why would you want to?

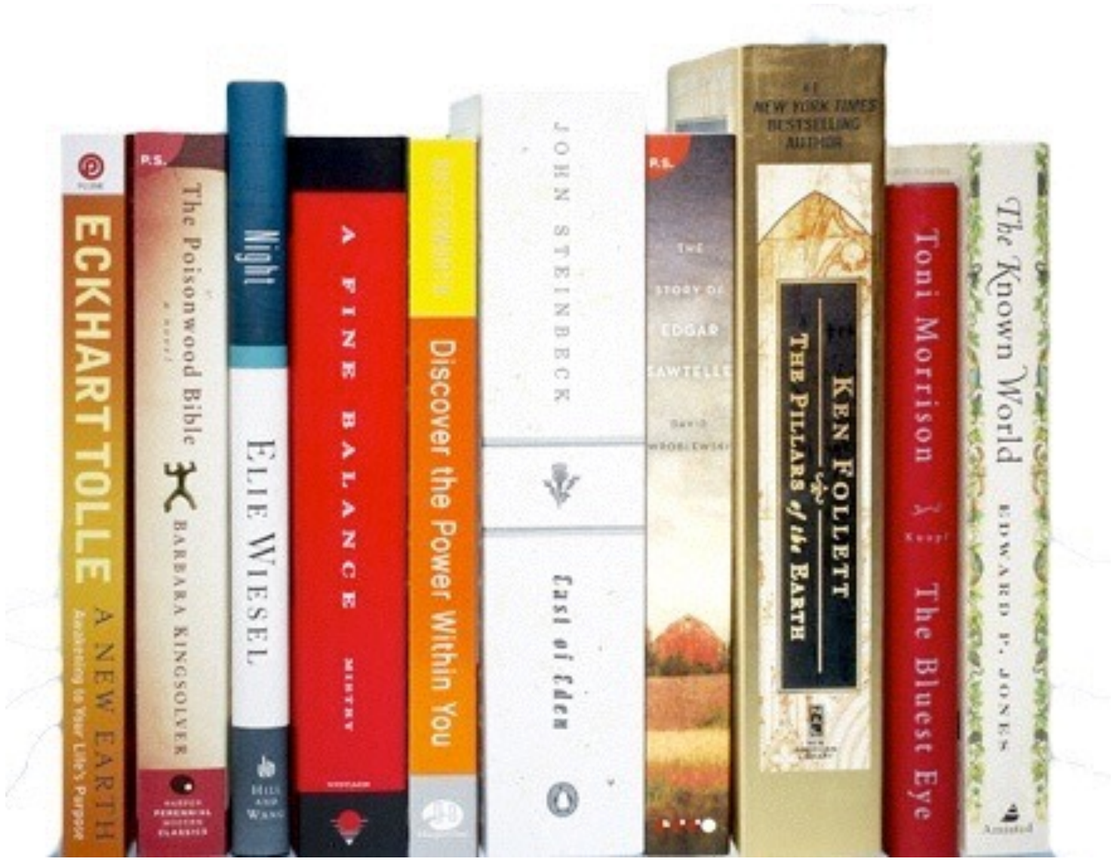
- accumulate to a container that doesn't implement `Collection`

Writing a Collector

Why would you want to?

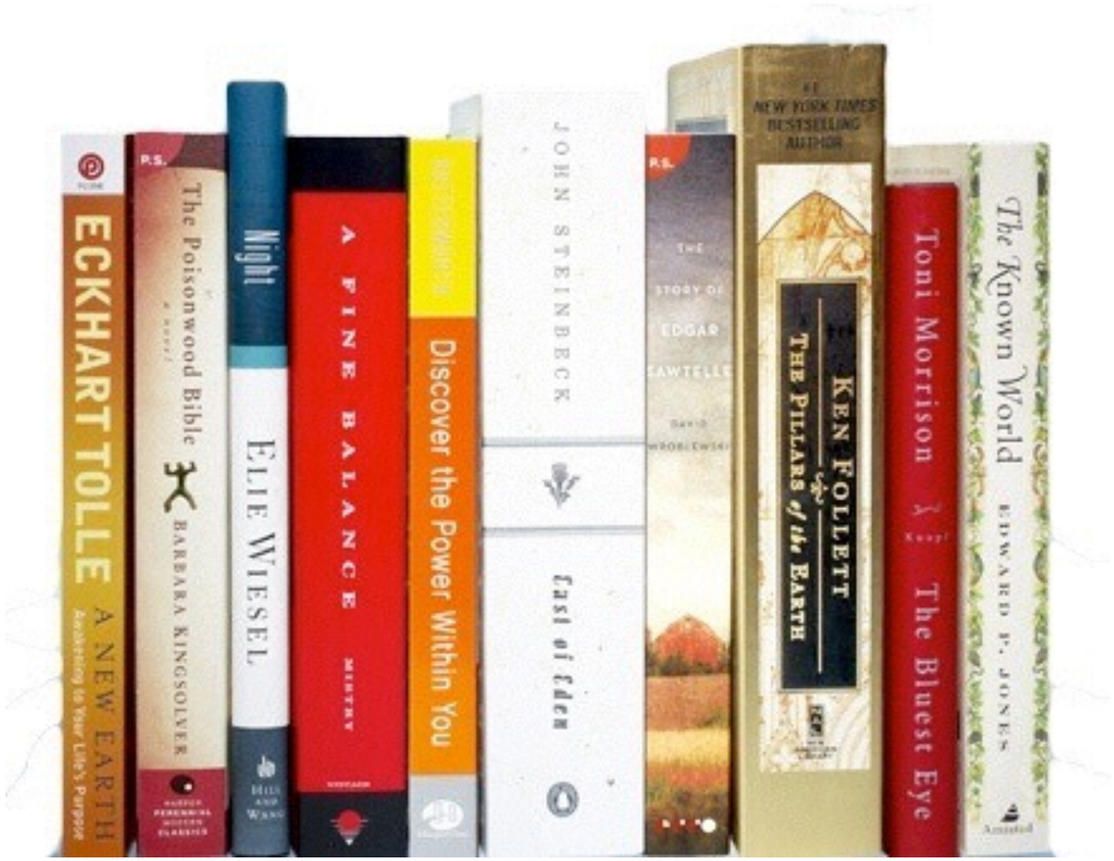
- accumulate to a container that doesn't implement `Collection`
- share state between values being collected

Oprah's Problem



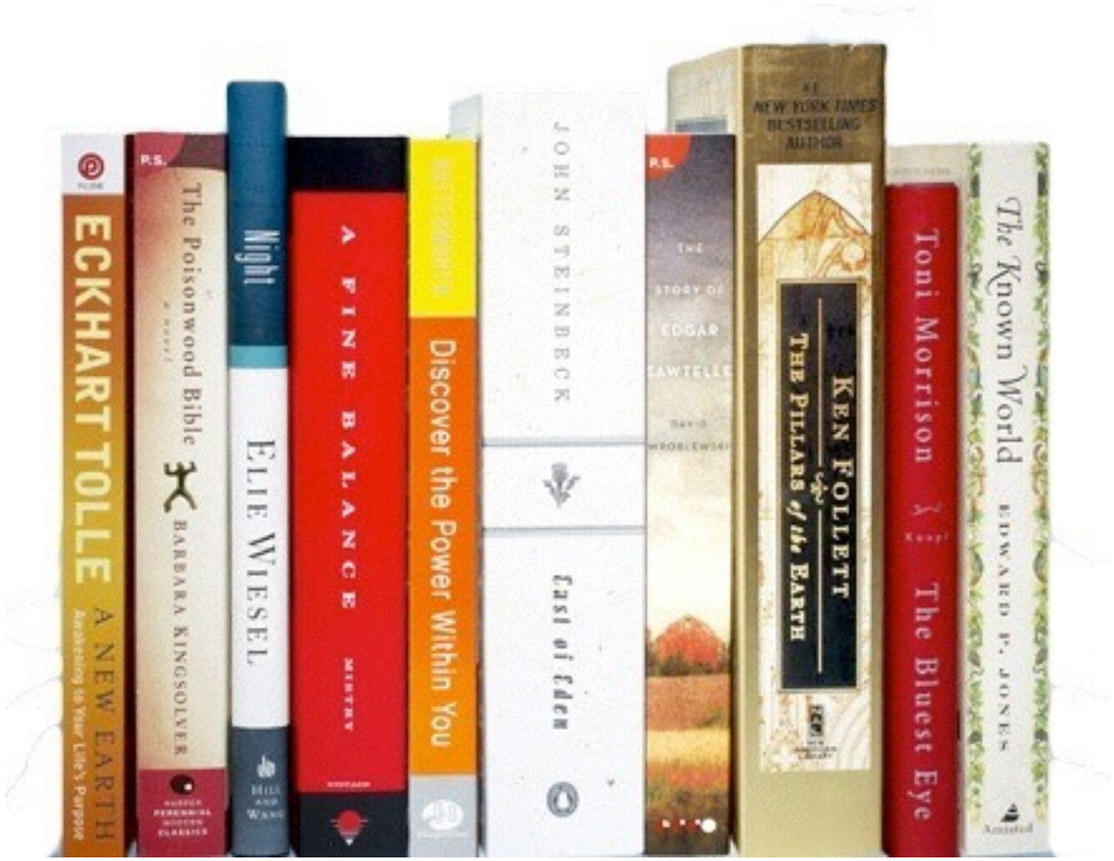
Oprah's Problem

How to find a book?



Oprah's Problem

How to find a book?

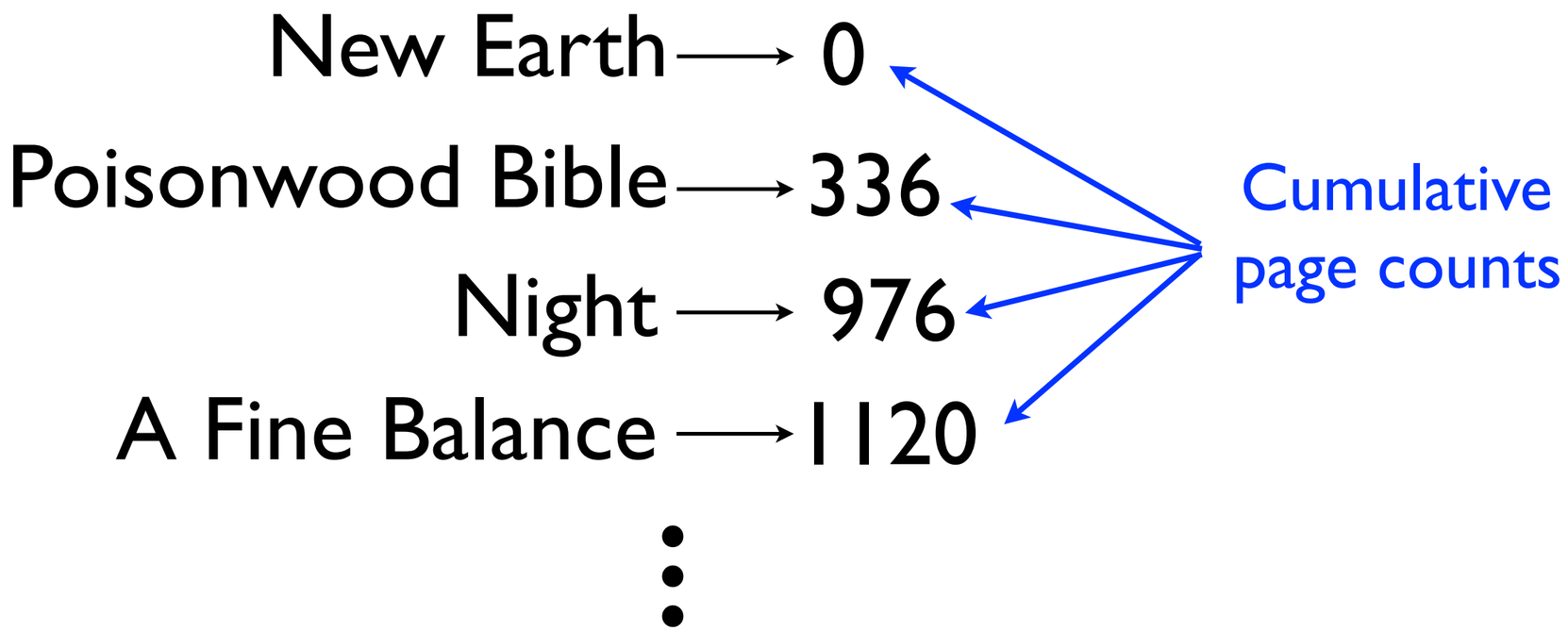


Page count ⟶

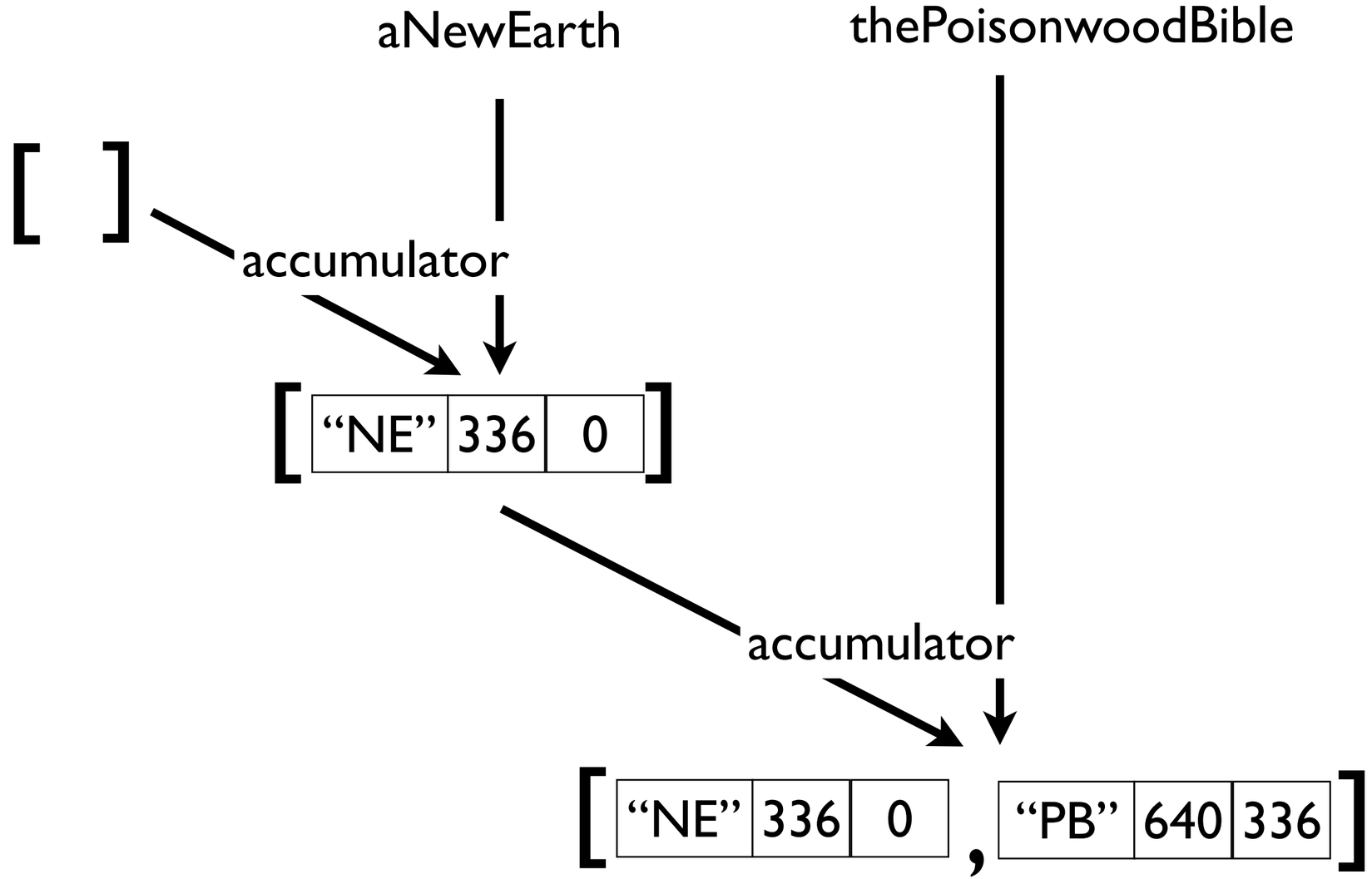
336 144 239 384 172
640 624 640 1104 400

Journey's End, JavaOne, October 2015

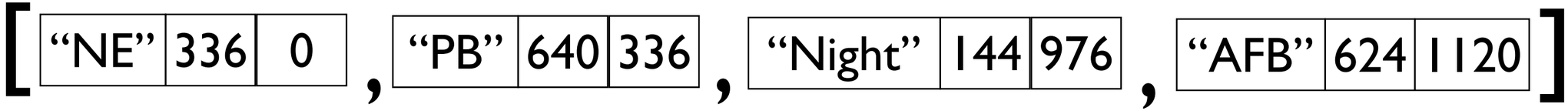
What Oprah Needs



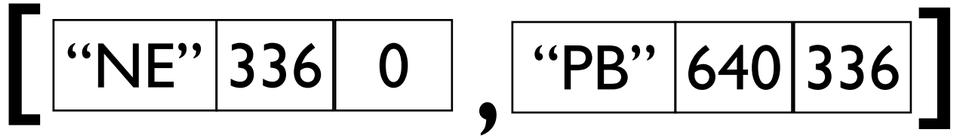
Supplier and Accumulator

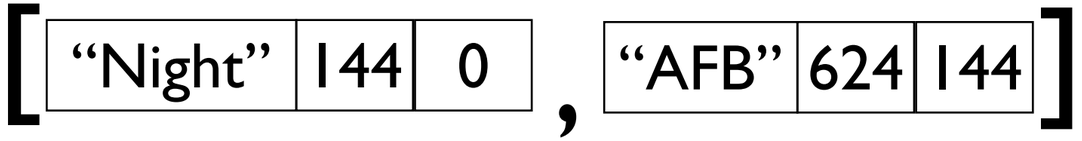


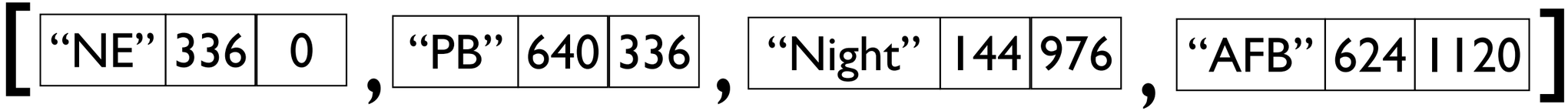
Combiner



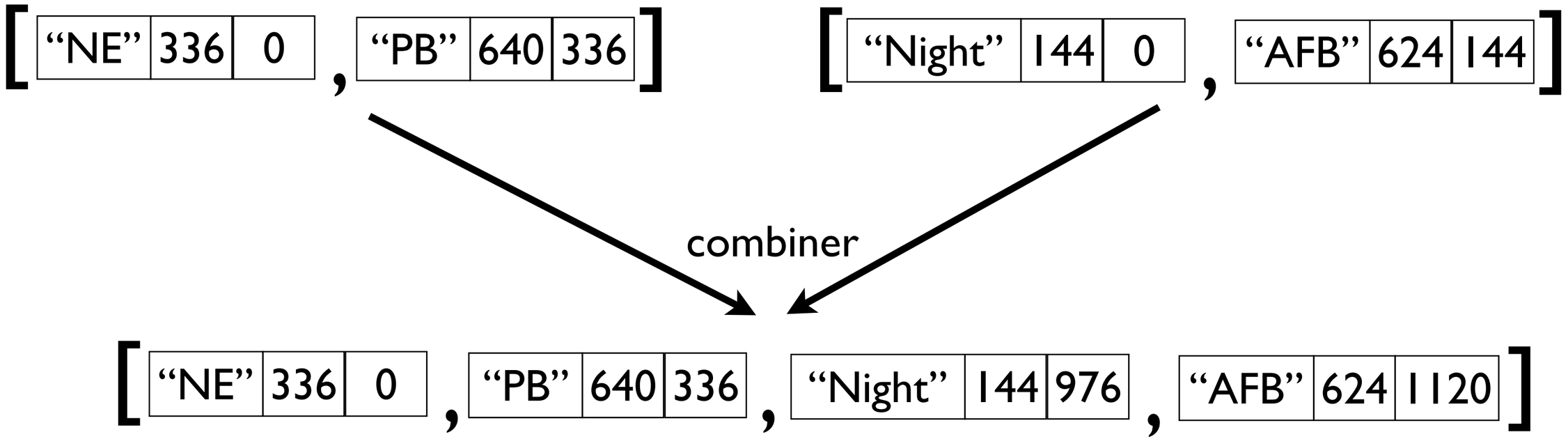
Combiner


["NE" 336 0 , "PB" 640 336]

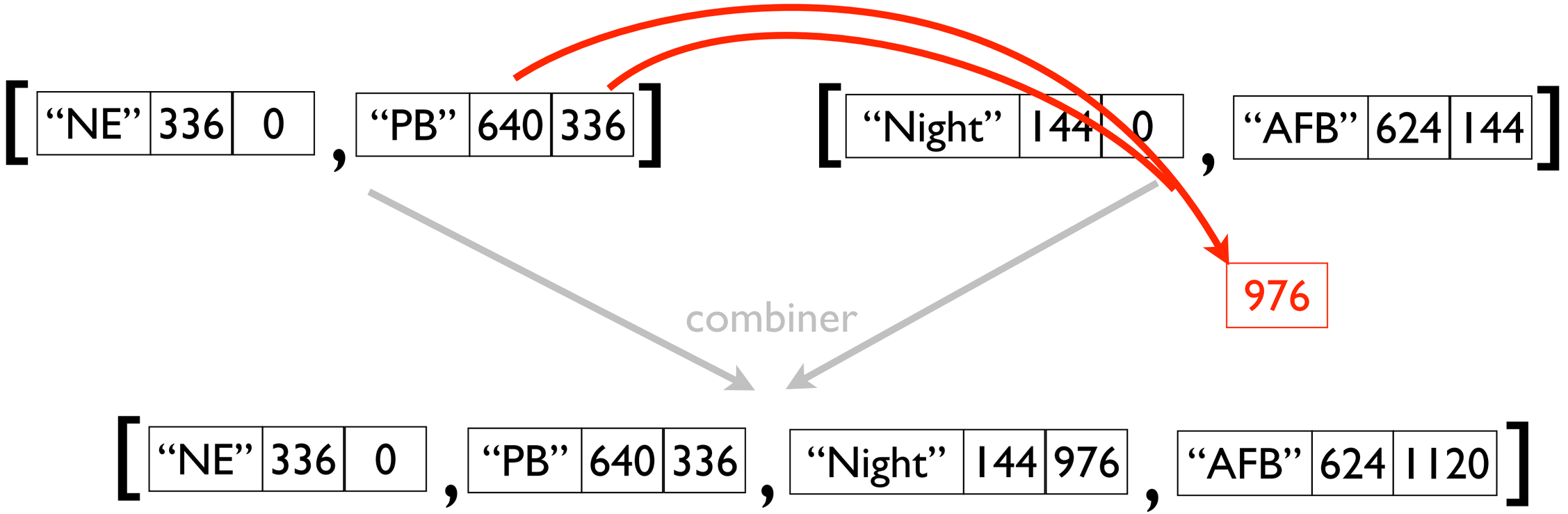

["Night" 144 0 , "AFB" 624 144]


["NE" 336 0 , "PB" 640 336 , "Night" 144 976 , "AFB" 624 1120]

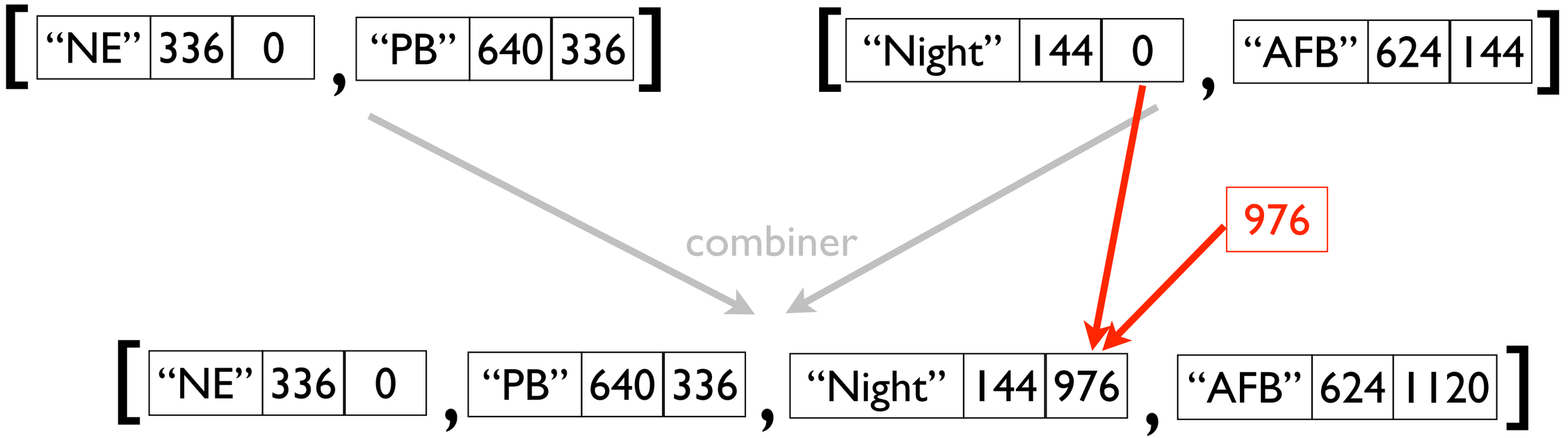
Combiner


["NE" 336 0 , "PB" 640 336]
["Night" 144 0 , "AFB" 624 144]
combiner
["NE" 336 0 , "PB" 640 336 , "Night" 144 976 , "AFB" 624 1120]

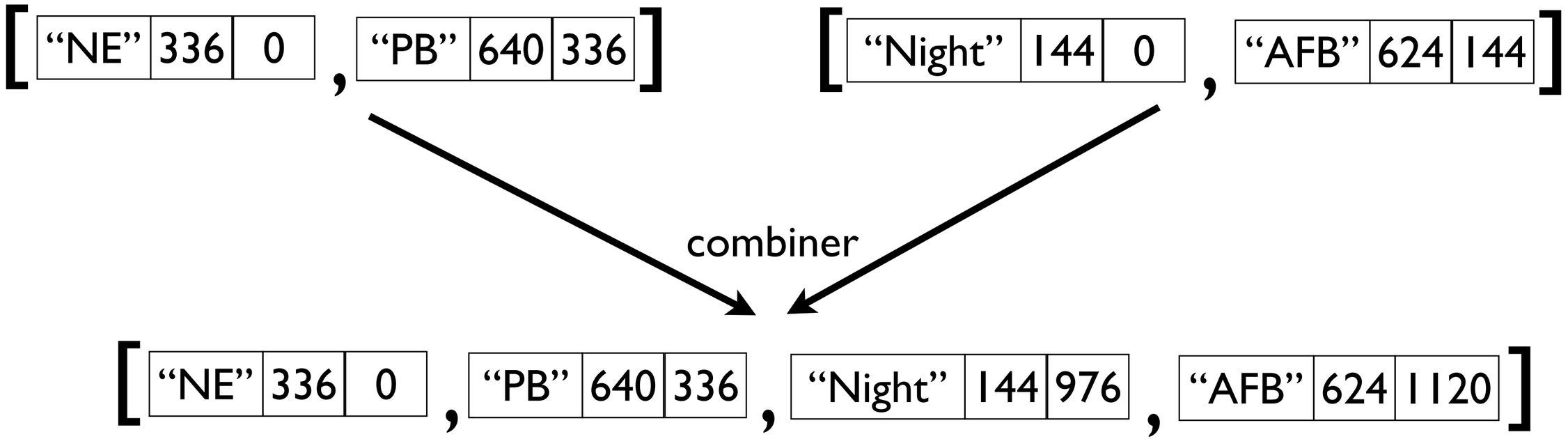
Combiner


["NE" 336 0 , "PB" 640 336]
["Night" 144 0 , "AFB" 624 144]
976
combiner
["NE" 336 0 , "PB" 640 336 , "Night" 144 976 , "AFB" 624 1120]

Combiner


["NE" 336 0 , "PB" 640 336]
["Night" 144 0 , "AFB" 624 144]
combiner
976
["NE" 336 0 , "PB" 640 336 , "Night" 144 976 , "AFB" 624 1120]

Combiner

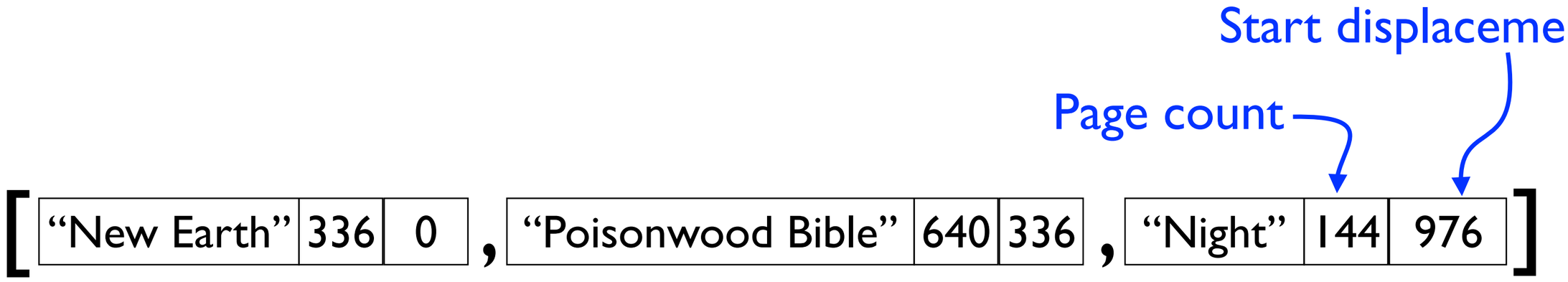

["NE" 336 0 , "PB" 640 336]
["Night" 144 0 , "AFB" 624 144]
combiner
["NE" 336 0 , "PB" 640 336 , "Night" 144 976 , "AFB" 624 1120]

Using Reduction Instead...

newEarth	poisonwood	night

Using Reduction Instead...

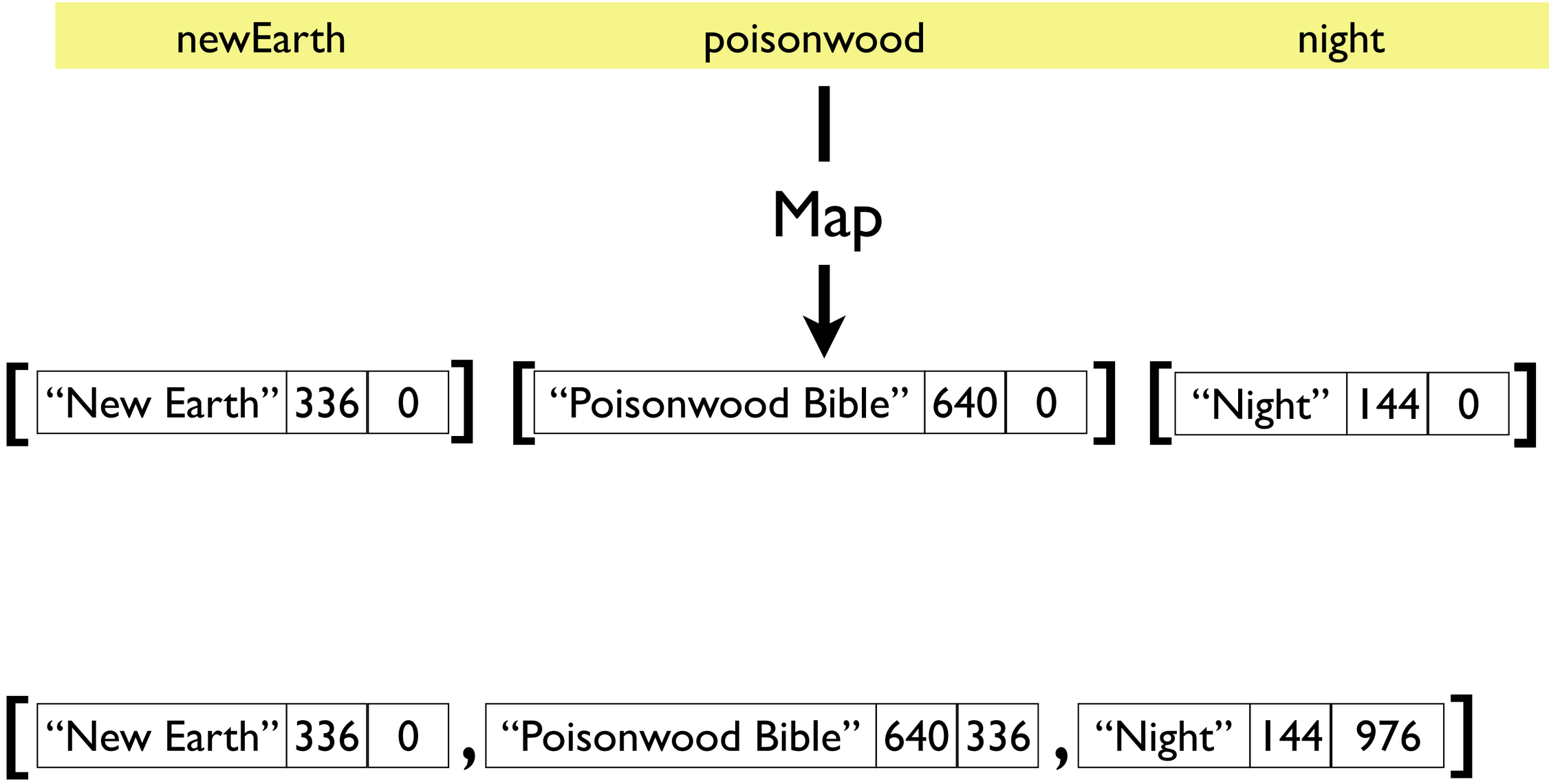
newEarth poisonwood night



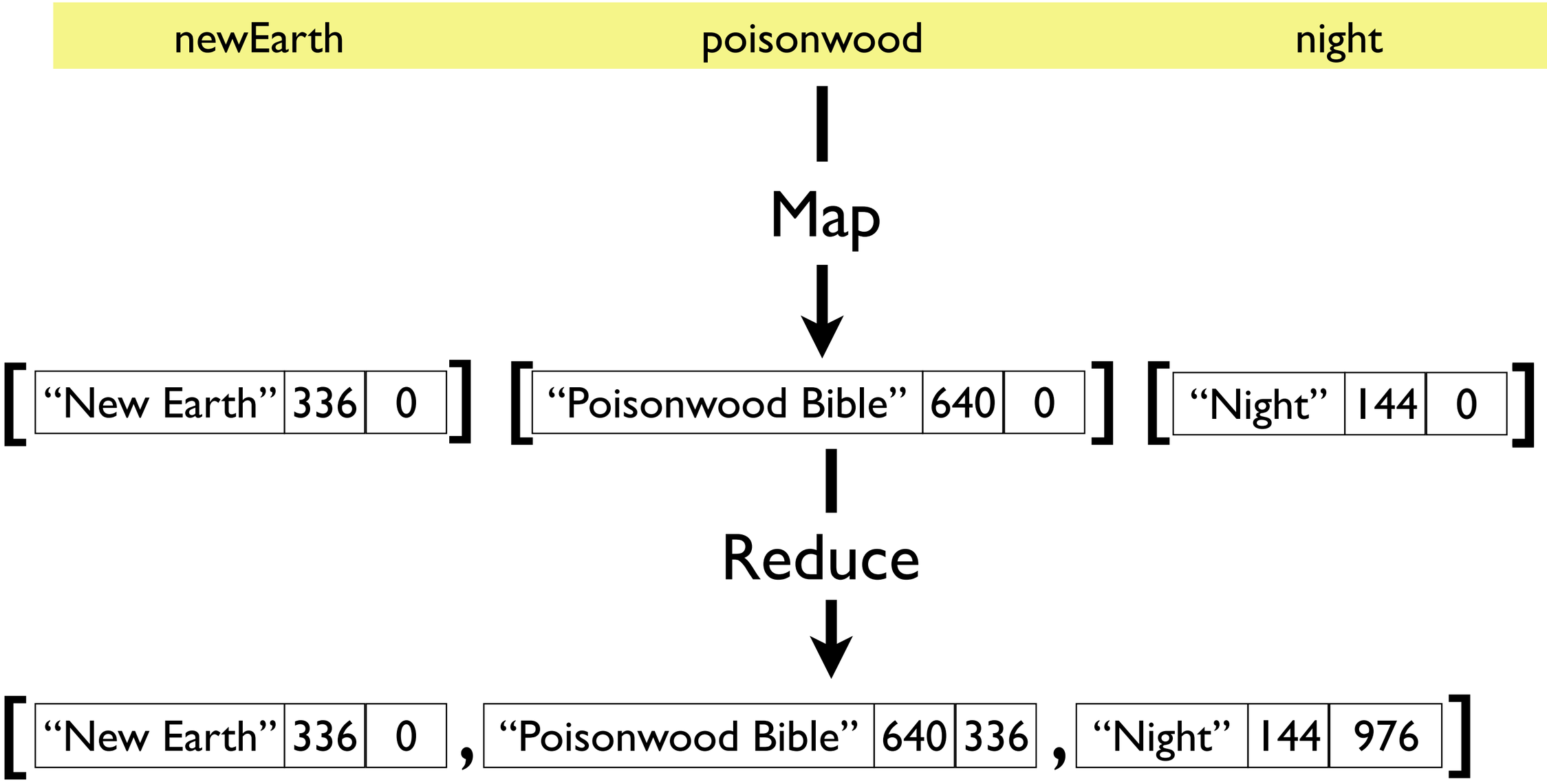
Using Reduction Instead...

newEarth	poisonwood	night

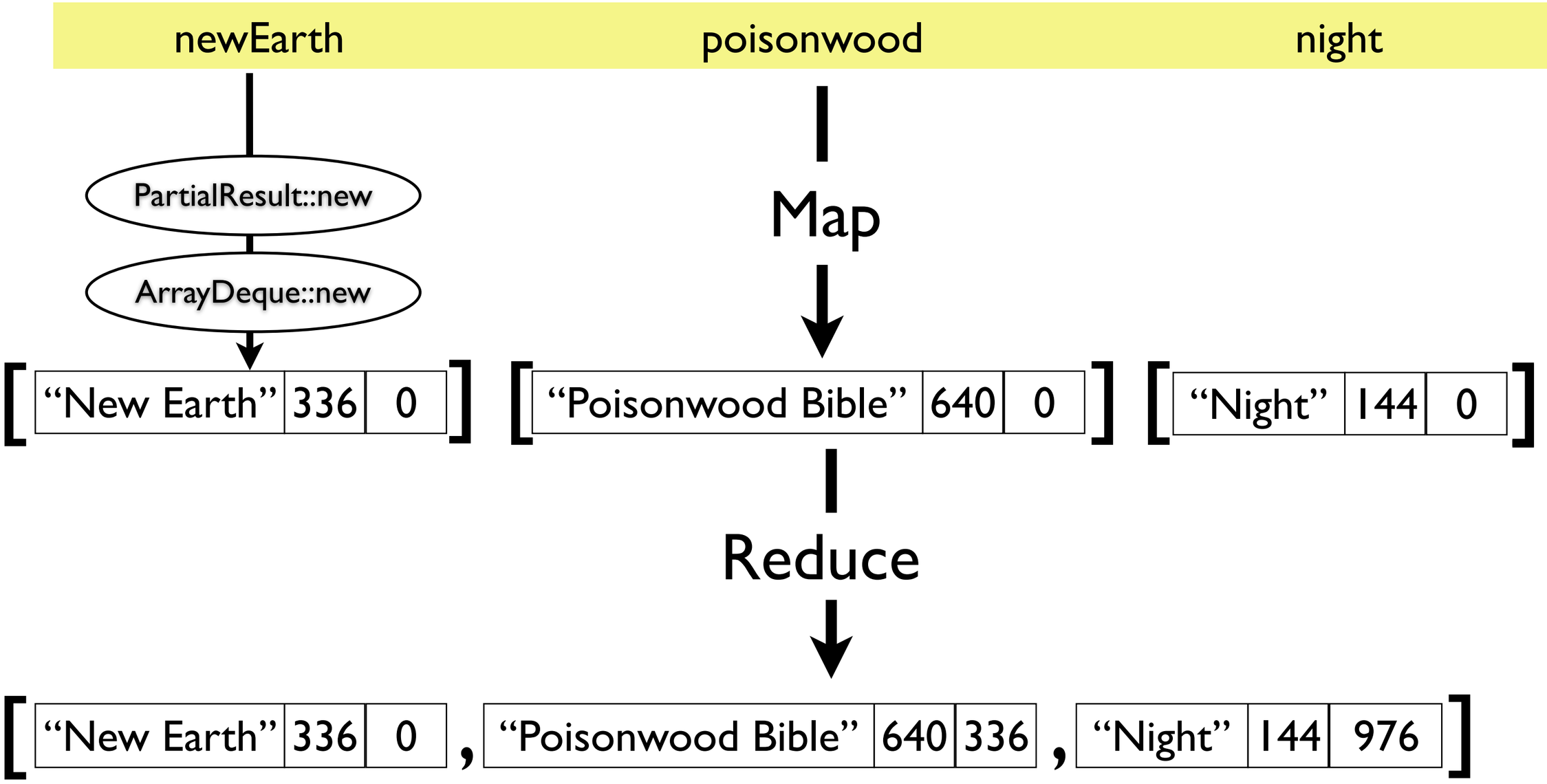
Using Reduction Instead...



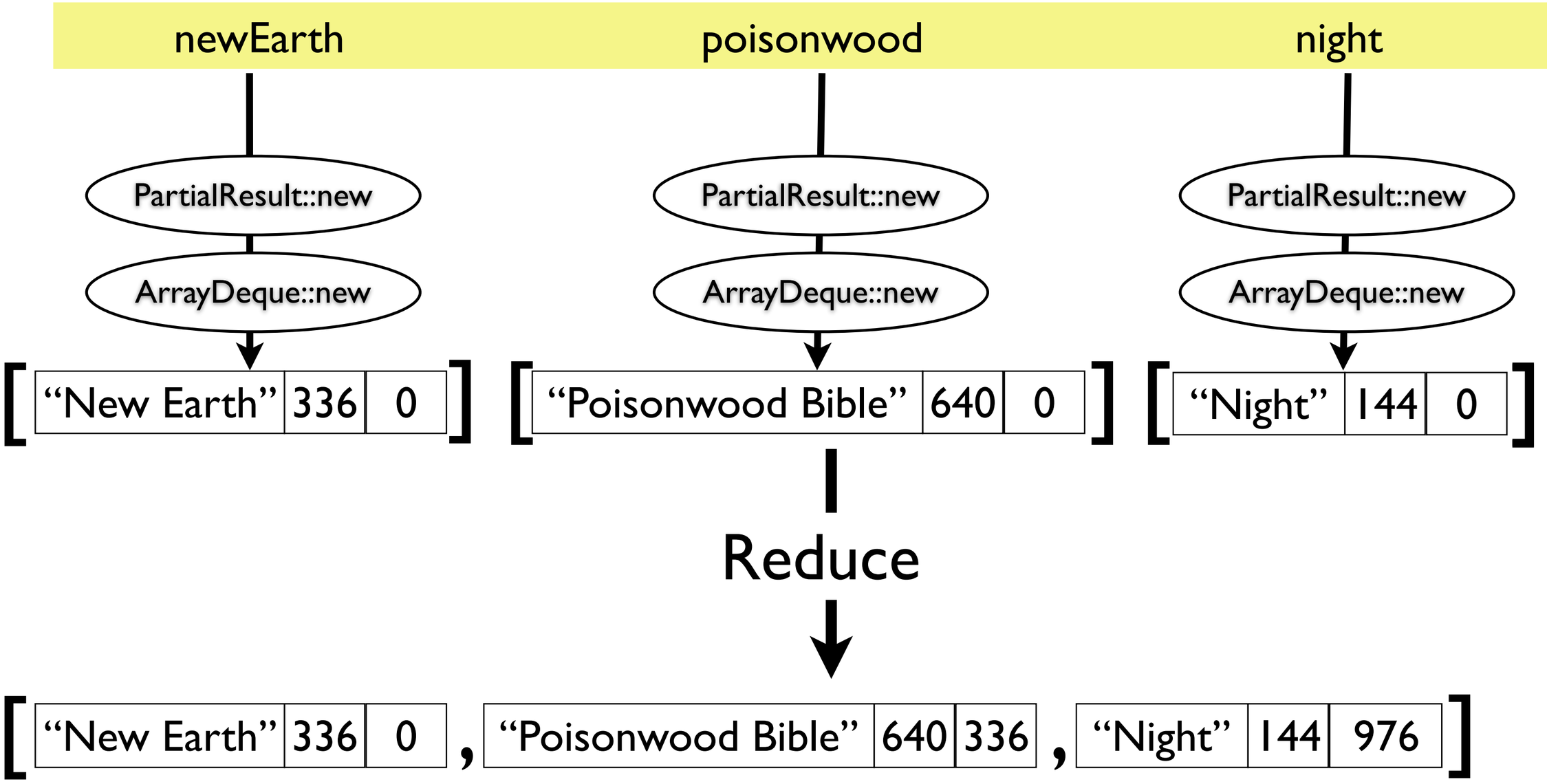
Using Reduction Instead...



Using Reduction Instead...



Using Reduction Instead...



Performance of Collectors

Depends on the performance of accumulator and combiner functions

- toList(), toSet(), toCollection – performance will probably be dominated by accumulator, but remember that the framework will need to manage multithread access to non-threadsafe containers for the combine operation

toMap(), toConcurrentMap()

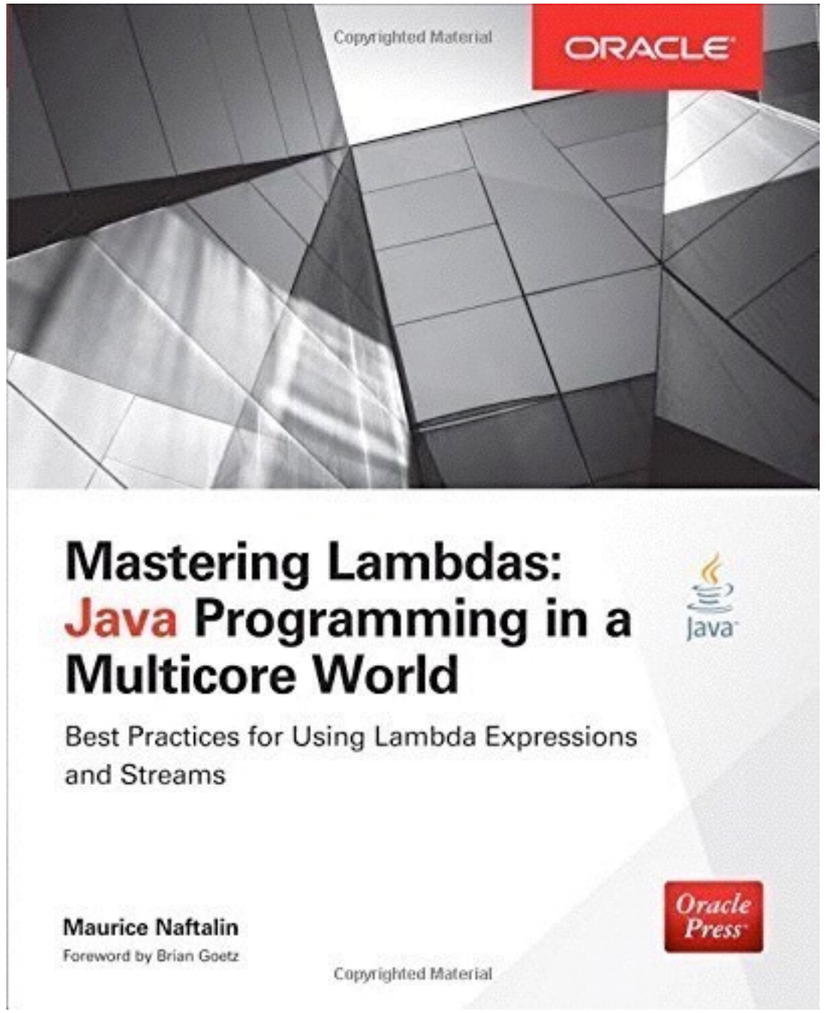
- map merging is slow. Resizing maps, especially concurrent maps, is particularly expensive. Whenever possible, presize all data structures, maps in particular.

Conclusion

Collectors

- generalisation of reduction
- allow a functional style while continuing to work with a language based on mutation
- designed for composition
- flexible and powerful programming tool
- take a bit of getting used to! – but they're worth it!

Questions?


Copyrighted Material
ORACLE
Mastering Lambdas:
Java Programming in a
Multicore World
Best Practices for Using Lambda Expressions
and Streams
Java
Maurice Naftalin
Foreword by Brian Goetz
Oracle
Press
Copyrighted Material