Lambdas and Streams

Develop Intelligence

Why Lambdas? Why Streams?

Develop Intelligence

Introduce a more functional style to Java programs;

- Cleaner, more expressive, more concise code
- Less arbitrary ordering of collection processing means more use of hardware parallelism

Agenda

- Lambdas and Method References
- Streams I – Intermediate Operations and Reductions
- Streams II – Collectors

Lambdas–taking values to a higher order

instead of supplying ***values*** to ***specific*** library methods

```
public interface Collection<E> {
    ...
    boolean removeAll(Collection<?> c);
    ...
}
```

we want to supply ***behaviour*** to ***general*** library methods:

```
public interface Collection<E> {
    ...
    boolean removeIf(Predicate<? super E> p);
    ...
}
```

`Predicate` is an interface with a single abstract boolean-valued method `test`. The method `removeIf` executes `test` for each element, and –

- if `test` returns `true`, `removeIf` removes that element

`Predicate` is an Interface

How can an instance of an interface represent behaviour?
A behaviour is implemented by a single method!

Functional Interface

Convention: if an interface has a *single abstract method*, like `Predicate`:

```
public interface java.util.functions.Predicate<T> {
     boolean test(T t);
}
```

then when the behaviour of an *instance of the interface* is wanted, the behaviour of *that single method* is used.

How to make an Interface Instance?

The old way was using an anonymous inner class:

```
// Predicate returns true for odd values of i
new Predicate<Integer>(){
   public boolean test(Integer i) {
      return i & 1 == 1;
});
```

and we *can* still supply that as a method argument:

```
// remove odd values from integerList
integerList.removeIf(new Predicate<Integer>(){
   public boolean test(Integer i) {
      return i & 1 == 1;
}));
```

Stripping Out the Boilerplate

```
// remove odd numbers from integerList
integerList.removeIf(new Predicate<Integer>(){
   public boolean test(Integer i) {
      return (i & 1) == 1;
   }
});
```

Why do we have to say we're supplying a `Predicate`?

Stripping Out the Boilerplate

```
// remove odd numbers from integerList
integerList.removeIf(new Predicate<Integer>(){
   public boolean test(Integer i) {
      return (i & 1) == 1;
   }
});
```

Why do we have to say we're supplying a `Predicate`?

Why do we have to say we're implementing `test`, the only abstract method?

Stripping Out the Boilerplate

```
// remove odd numbers from integerList
integerList.removeIf(new Predicate<Integer>(){
   public boolean test(Integer i) {
      return (i & 1) == 1;
   }
});
```

Why do we have to say we're supplying a `Predicate`?

Why do we have to say we're implementing `test`, the only abstract method?

Why do we have to say the type parameter is `Integer`?

Stripping Out the Boilerplate

```
// remove odd numbers from integerList
integerList.removeIf(new Predicate<Integer>(){
   public boolean test(Integer i) {
      return (i & 1) == 1;
   }
});
```

Why do we have to say we're supplying a `Predicate`?

Why do we have to say we're implementing `test`, the only abstract method?

Why do we have to say the type parameter is `Integer`?

Stripping Out the Boilerplate

```
// remove odd numbers from integerList
integerList.removeIf(i    (i & 1) == 1)
```

Why do we have to say we're supplying a `Predicate`?

Why do we have to say we're implementing `test`, the only abstract method?

Why do we have to say the type parameter is `Integer`?

All we need is one extra syntax element!

Stripping Out the Boilerplate

```
// remove odd numbers from integerList
integerList.removeIf(i -> (i & 1) == 1)
```

Why do we have to say we're supplying a `Predicate`?

Why do we have to say we're implementing `test`, the only abstract method?

Why do we have to say the type parameter is `Integer`?

All we need is one extra syntax element!

Stripping Out the Boilerplate

```
// remove odd numbers from integerList
integerList.removeIf(i -> (i & 1) == 1)
```

Why do we have to say we're supplying a `Predicate`?

Why do we have to say we're implementing `test`, the only abstract method?

Why do we have to say the type parameter is `Integer`?

All we need is one extra syntax element!

Rules for Writing Lambdas

```
() -> 42

a -> a + 1          // single untyped parameter – can omit parentheses

(a,b) -> a + b      // but they are needed for multiple parameters

(int a, int b) -> a + b;
                    // type either all parameters, or none

() -> { println(42); return 42; }
                    // lambda body can be a statement
```

Lots of detail on http://lambdafaq.org!

Target Typing

Context is needed to define a lambda's type:

```
Runnable r = () -> {}                 // compiles
Thread t = new Thread(() -> {})       // compiles
Object o = () -> {}                   // illegal
```

Functional Interfaces

Five types of functional interface implemented in `java.util.function`:

	Purpose	Example
`Supplier`	`Creates a value`	`() -> new StringBuilder()`
`Consumer`	`Works by side-effect, returning nothing`	`s -> System.out.println(s)`
`Function`	`Maps one or more values to another`	`(String s) -> new Integer(s)`
`Operator`	`Function with inputs and returns of the same type`	`(a,b) -> a + b`
`Predicate`	`Function returning a boolean`	`l -> l.isEmpty()`

Method References

Sometimes it's more convenient to write lambdas as *method references*. There are four kinds:

	example	lambda equivalent
constructor	`ArrayList::new`	`() -> new ArrayList()`
static	`Integer::max`	`(x,y) -> Integer.max(x,y)`
bound	`“abc"::concat`	`str -> "abc".concat(str)`
unbound	`String::length`	`str -> str.length()`

Lambdas Lab – Setup

- Clone or download the course repository:

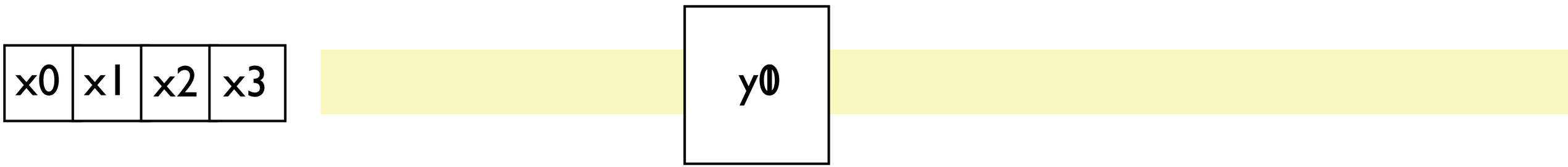
  https://github.com/MauriceNaftalin/Salesforce-Intermediate-Java

- Select *New* | *Project from Existing Sources*;
  - select the new directory `sfij-project`;
  - click *Open*, then *Import project from external model*, select *Maven*;
  - let IntelliJ guide you through the setup.

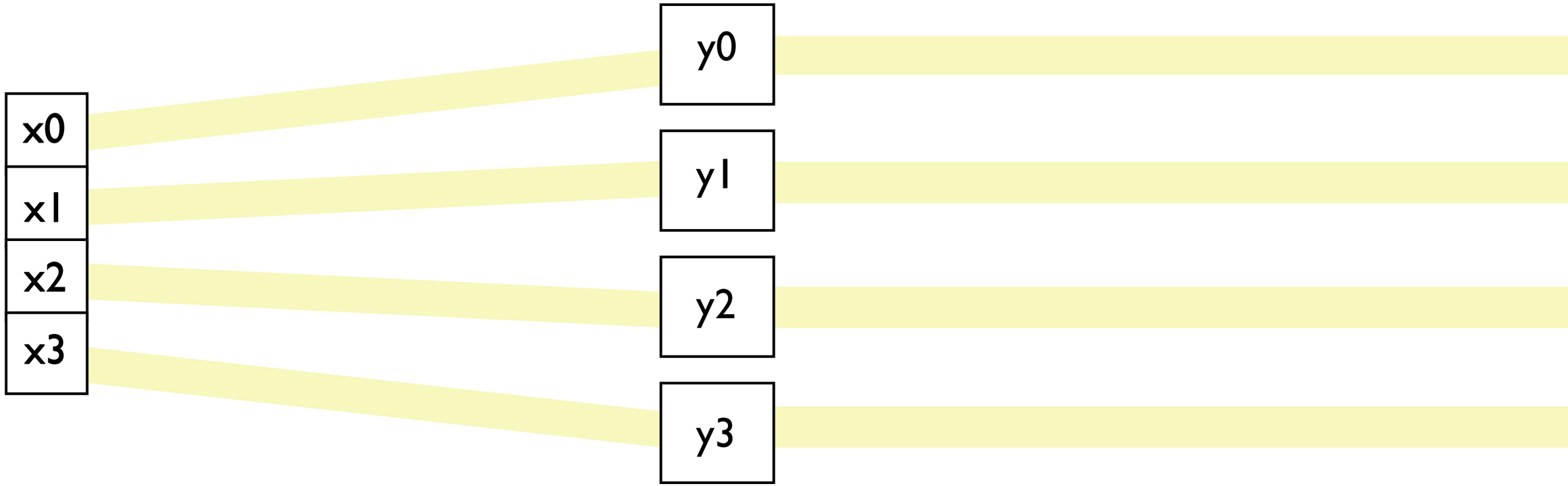
Agenda

- Lambdas and Method References
- Streams I – Intermediate Operations and Reductions
- Streams II – Collectors

Visualizing Stream Operations



Visualizing Stream Operations



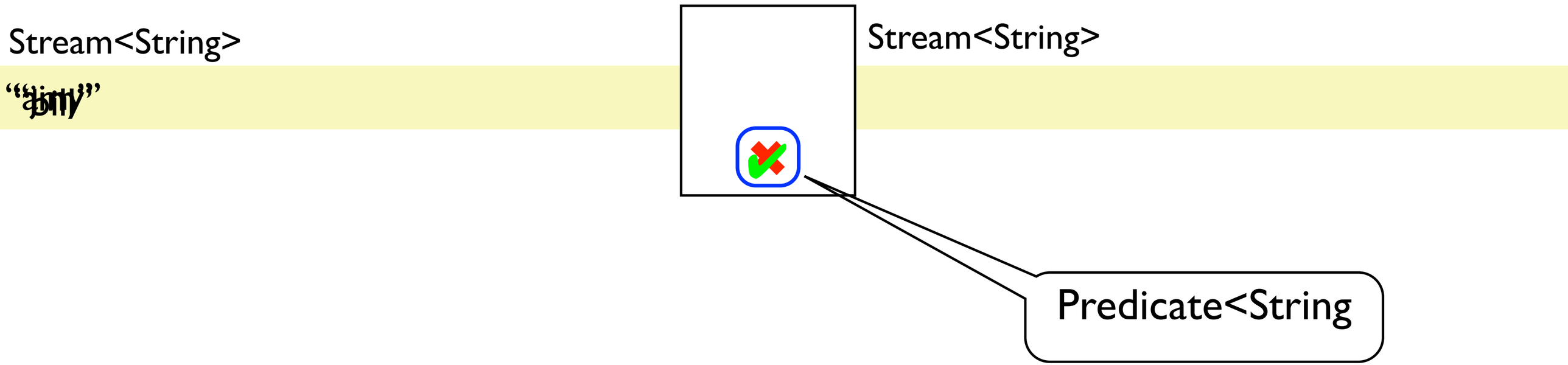
Stateless Intermediate Operations

name	returns	interface used	FI signature
filter	Stream<T>	Predicate<T>	T → boolean
map	Stream<U>	Function<T,U>	T → U
flatMap	Stream<R>	Function<T,Stream<R>>	T → Stream<R>
peek	Stream<T>	Consumer<T>	T → void

mapToInt	IntStream	ToIntFunction<T>	T → int
mapToLong	LongStream	ToLongFunction<T>	T → long
mapToDouble	DoubleStream	ToDoubleFunction<T>	T → double

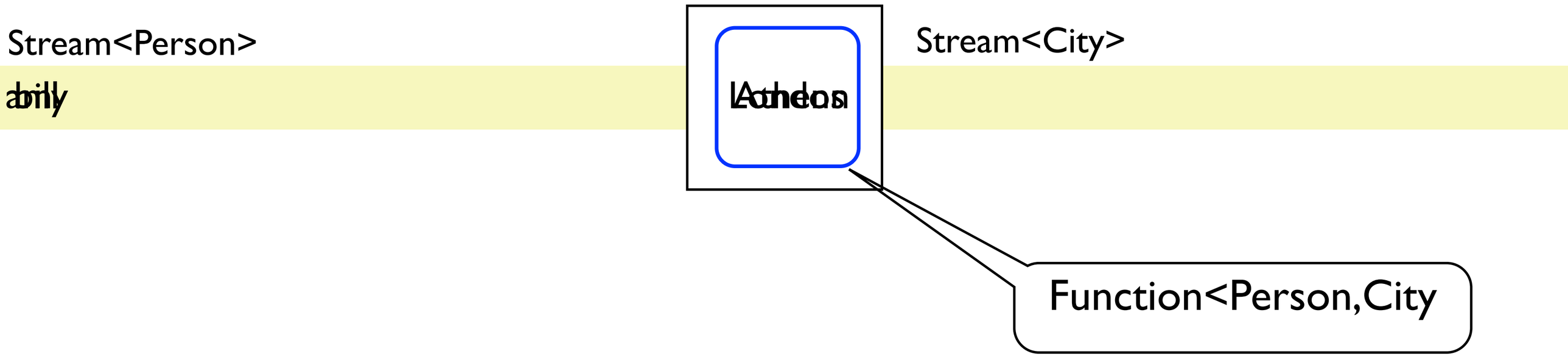
filter()

```
filter(s -> s.length() < 4)
```



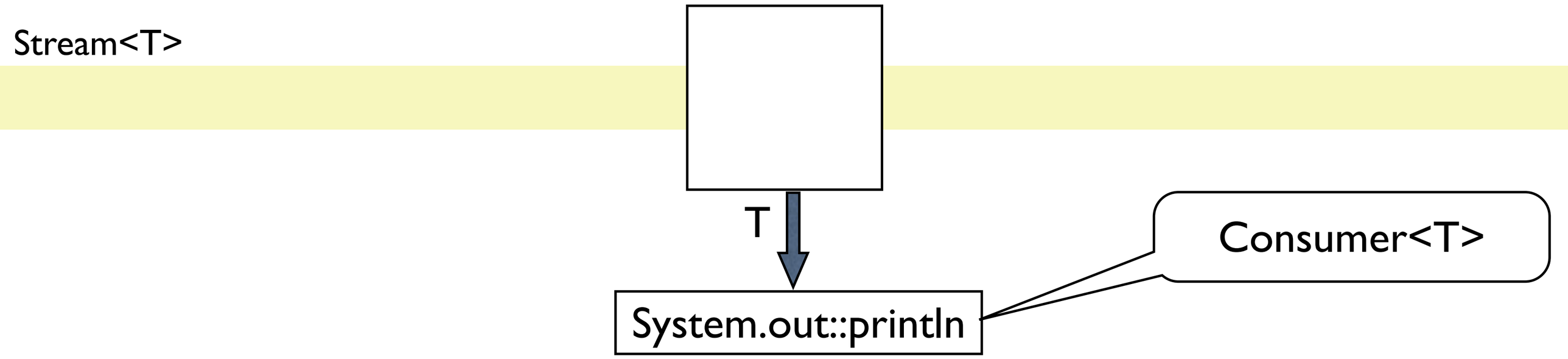
map()

```
map(Person::getCity)
```

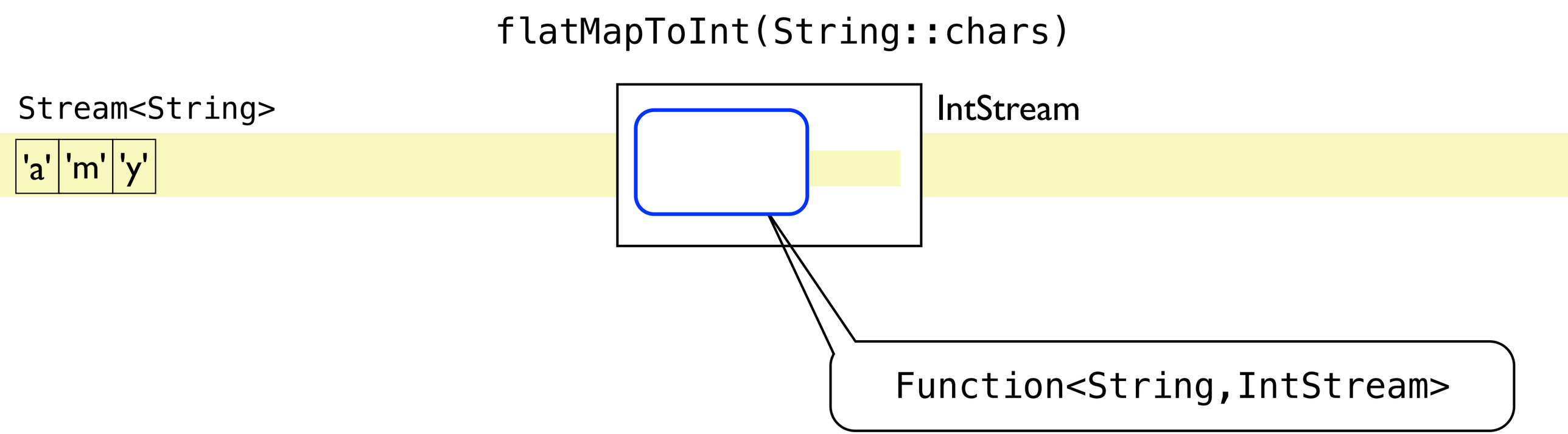


peek()

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



flatMapToInt()



Stateful Intermediate Operations

name	returns	type used	FI signature
`limit`	`Stream<T>`	`long`	
`skip`	`Stream<T>`	`long`	
`sorted`	`Stream<T>`	`Comparator<T>`	`(T, T) → int`
`distinct`	`Stream<T>`		
`takeWhile*`	`Stream<T>`	`Predicate<T>`	
`dropWhile*`	`Stream<T>`	`Predicate<T>`	

* added in Java 9

Collectors: Journey's End for a Stream

The Life of a Stream Element

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?

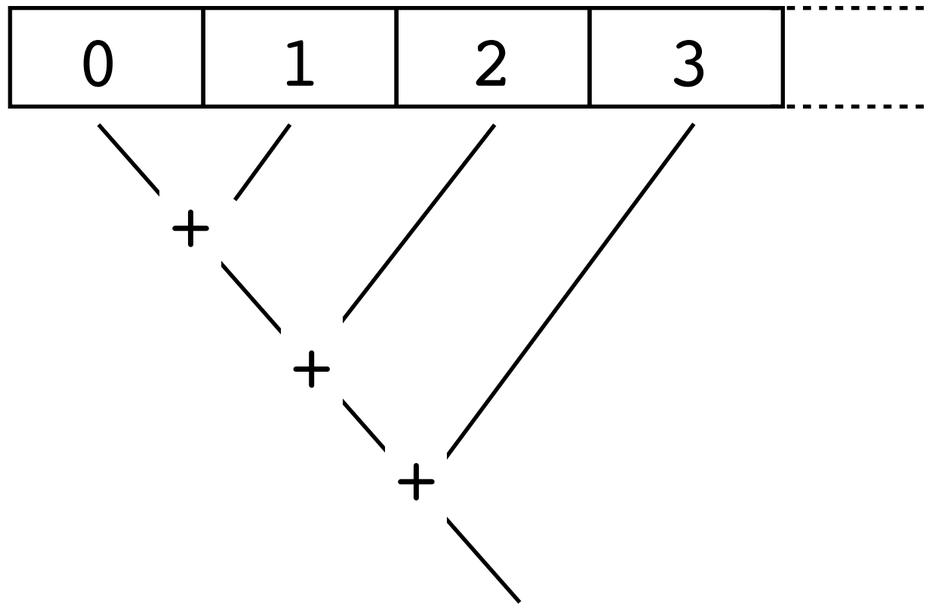
```
int sum = 0;
people.stream()
      .mapToInt(Person::getAge)
      .forEach(a -> { sum += a; });
```

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?

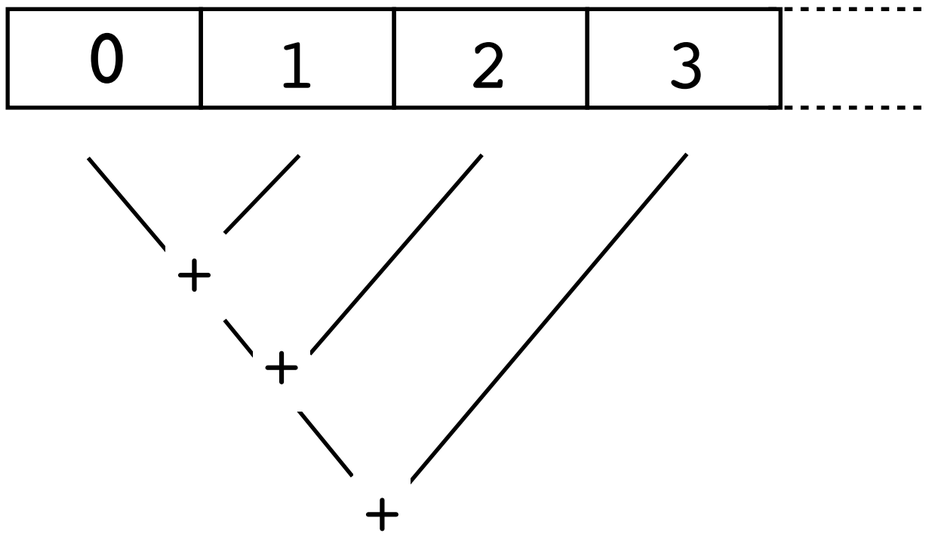
- Avoiding an accumulator:

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

Reduction – Why?

- Avoiding an accumulator:

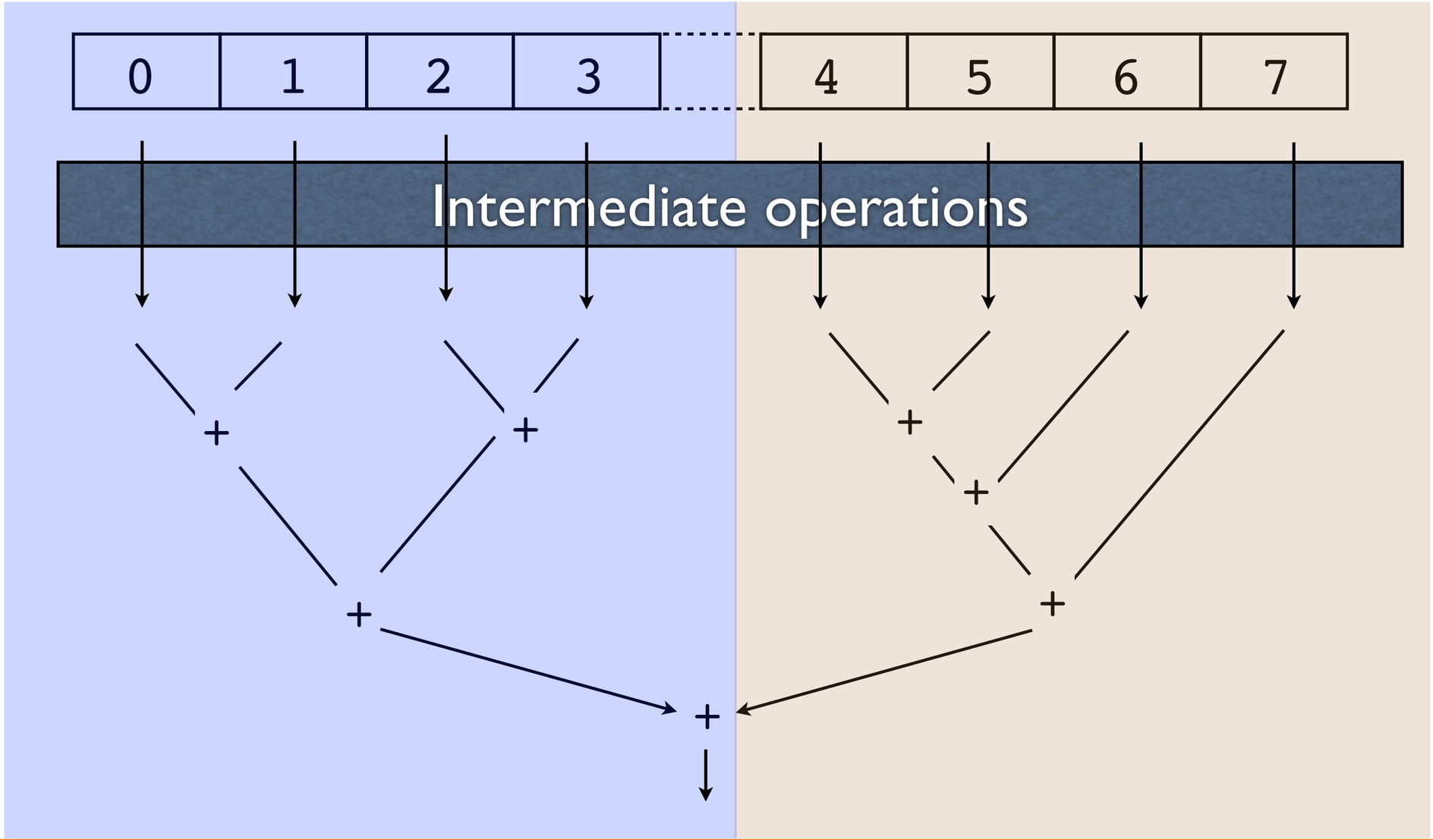
```
int[] vals = new int[100];
Arrays.setAll(vals, i -> i);

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



BinaryOperator must be associative!

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

Reduction – Why?



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);
```

Streams I Lab

Agenda

- Lambdas and Method References
- Streams I – Intermediate Operations and Reductions
- Streams II – Collectors

What about Collections?

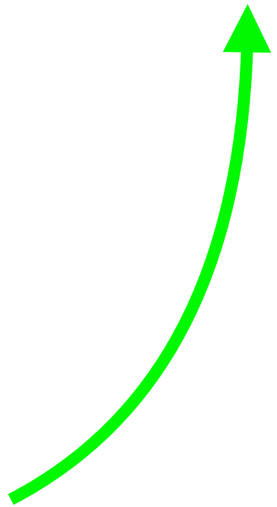
Reduction does work on collections – sort of … but then

- for each element, client code has to create a new empty collection and add the single element to it,
- and combine collections using `addAll`

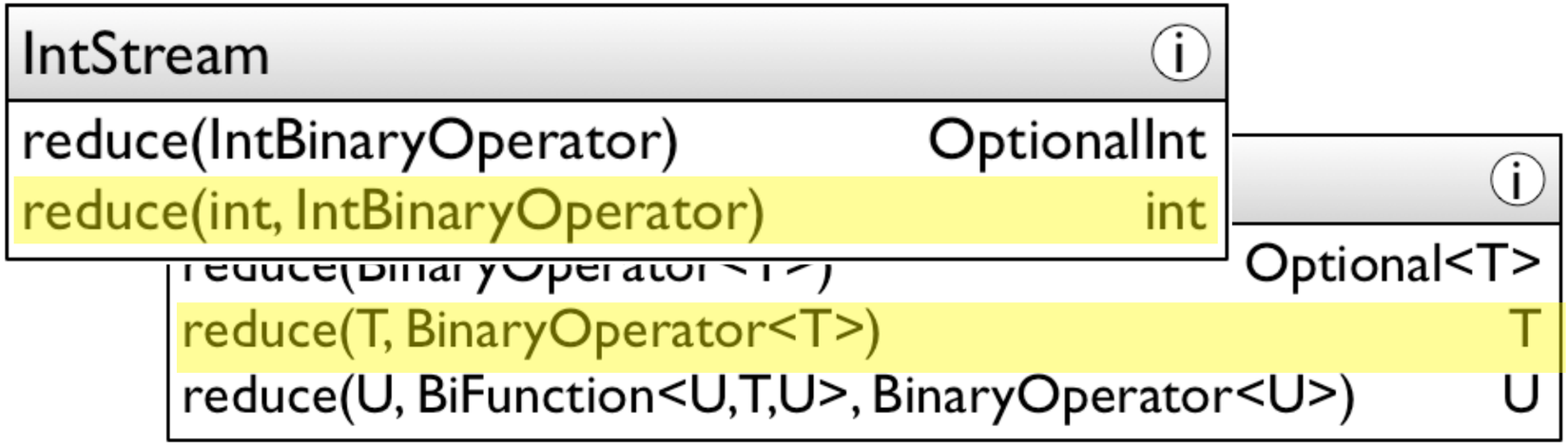
. . . it's hopelessly inefficient!

Reduction over an Identity

A better idea would be



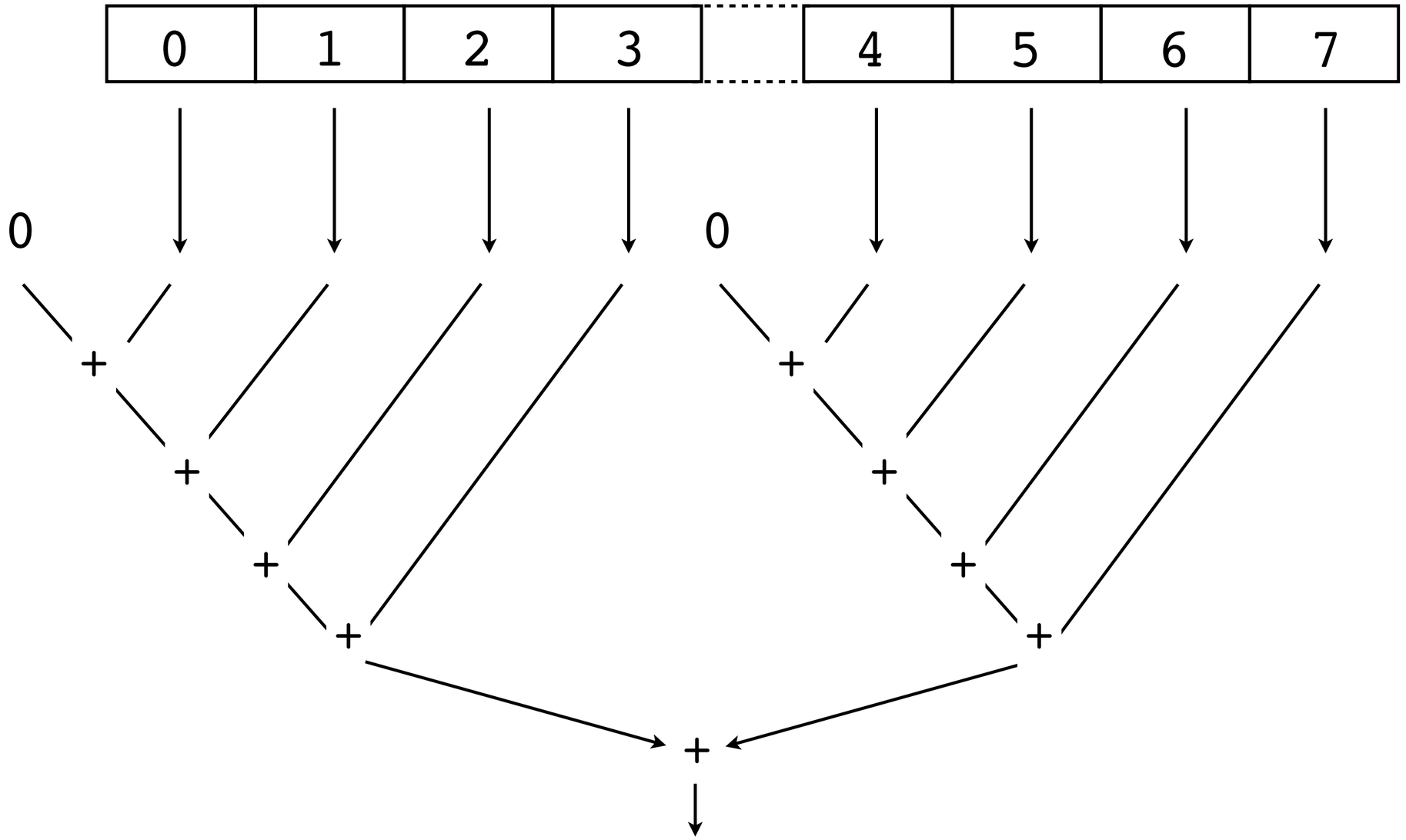
Reduction over an Identity



Works for primitives (of course) and also 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



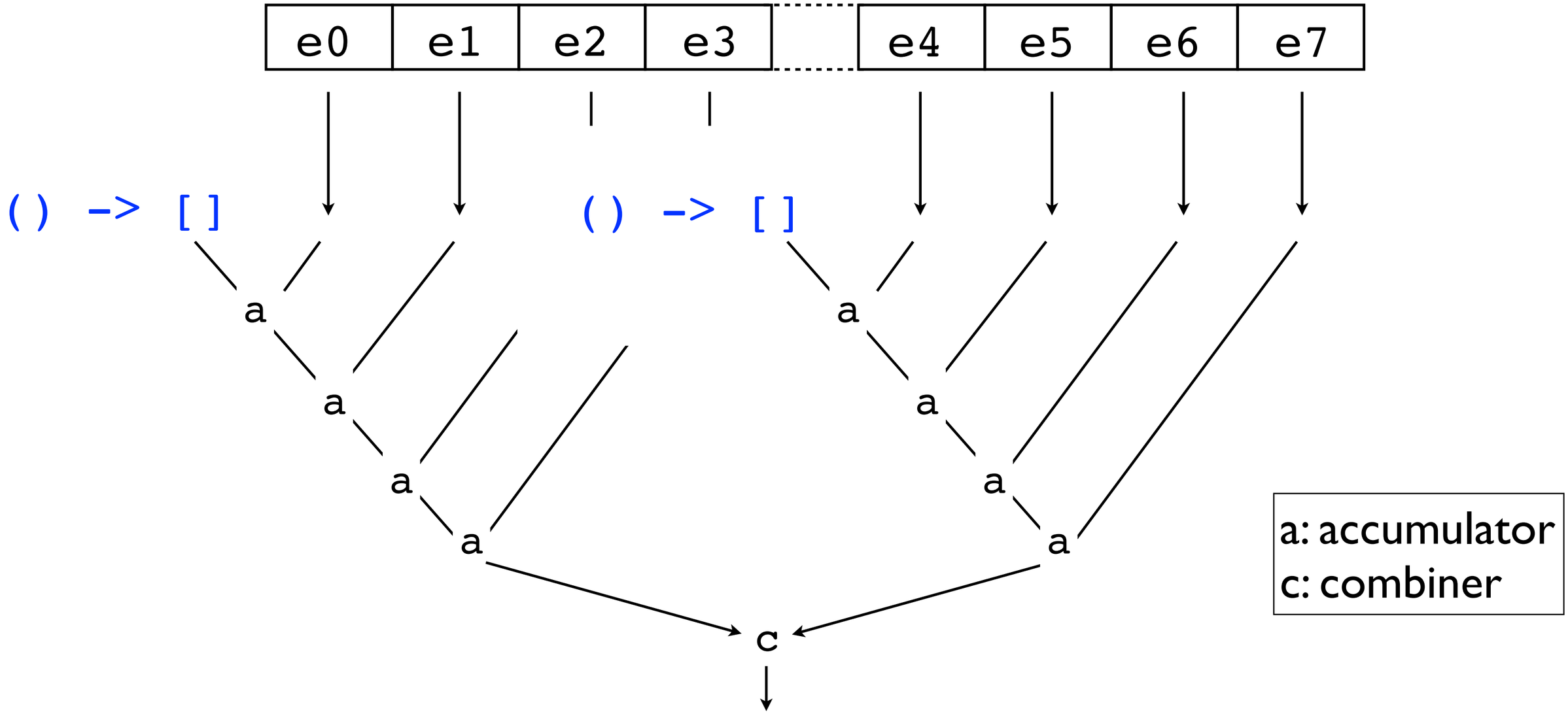
So, Reduction to Collections?

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

- reduction **reuses** the identity element

We need something better!

The Answer – Collectors



Collectors

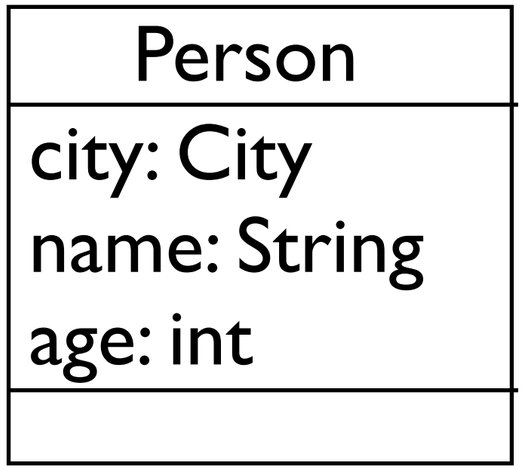
So to define a collector, you need to provide

- Supplier
- Accumulator
- Combiner

That sounds really hard…

Good then that we don't have to do it!

Example Domain




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

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

Collectors API

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

- framework-supplied containers
- custom collections
- classification maps

Using the Predefined Collectors

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

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

framework provides the Supplier

user provides the Supplier

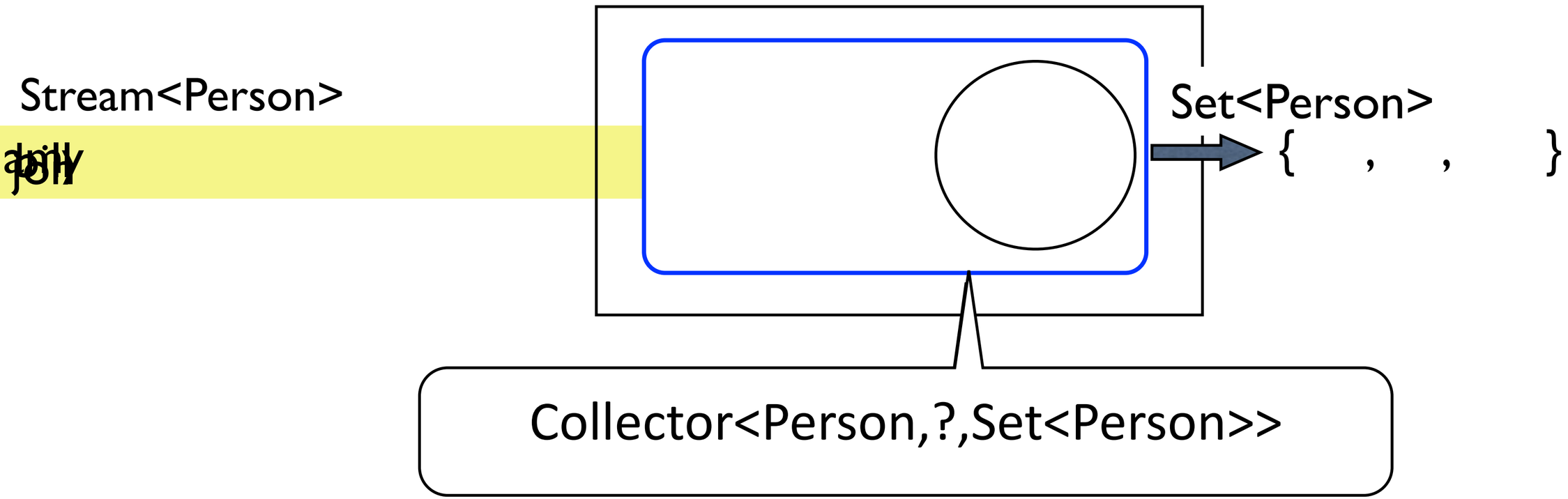
produce a *classification map*

Simple Collector – `toSet()`

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

Simple Collector – `toSet()`

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

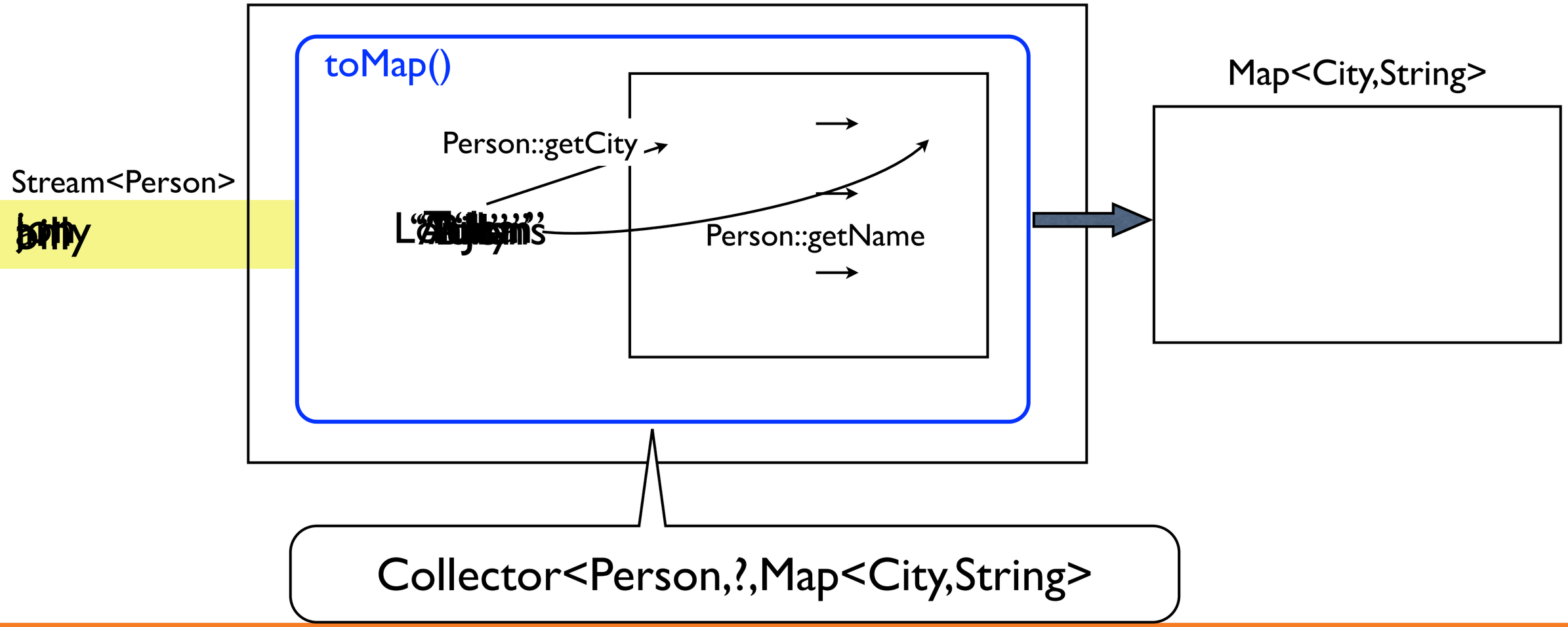


toMap(Function<T,K> keyMapper, Function<T,U> valueMapper)

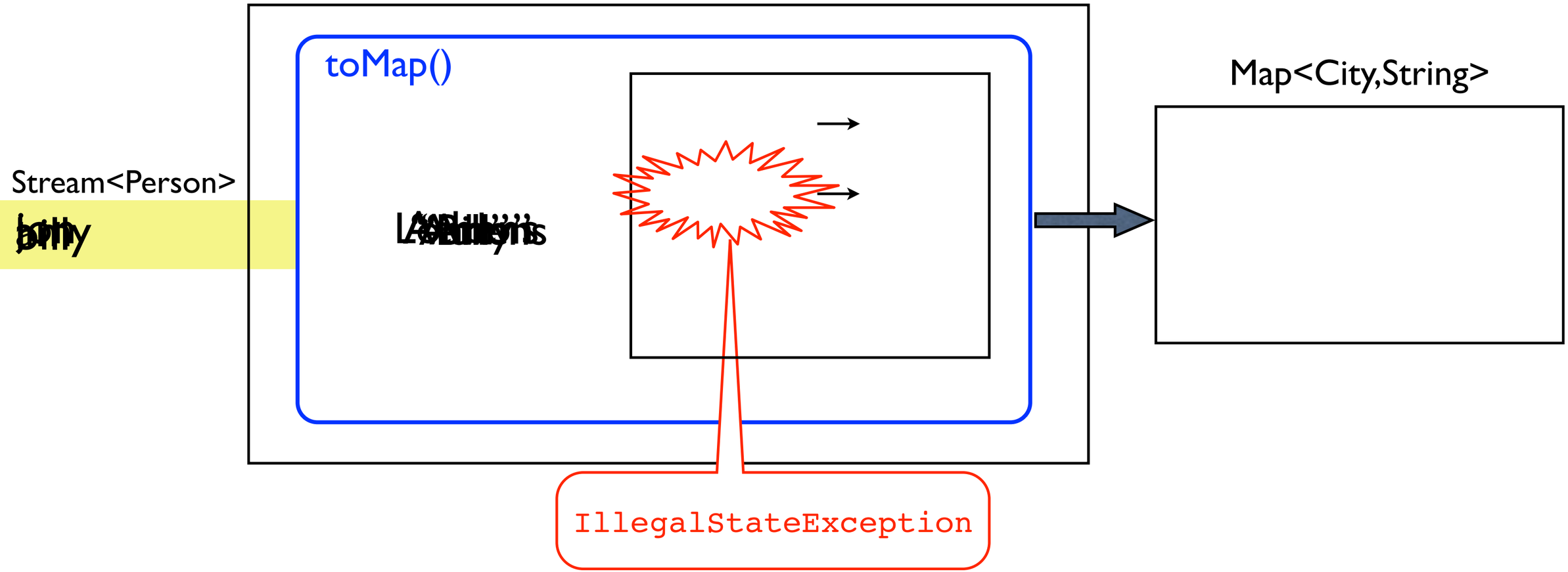
```
people.stream().collect(
      Collectors.toMap(
            Person::getCity,
            Person::getName))
```

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

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



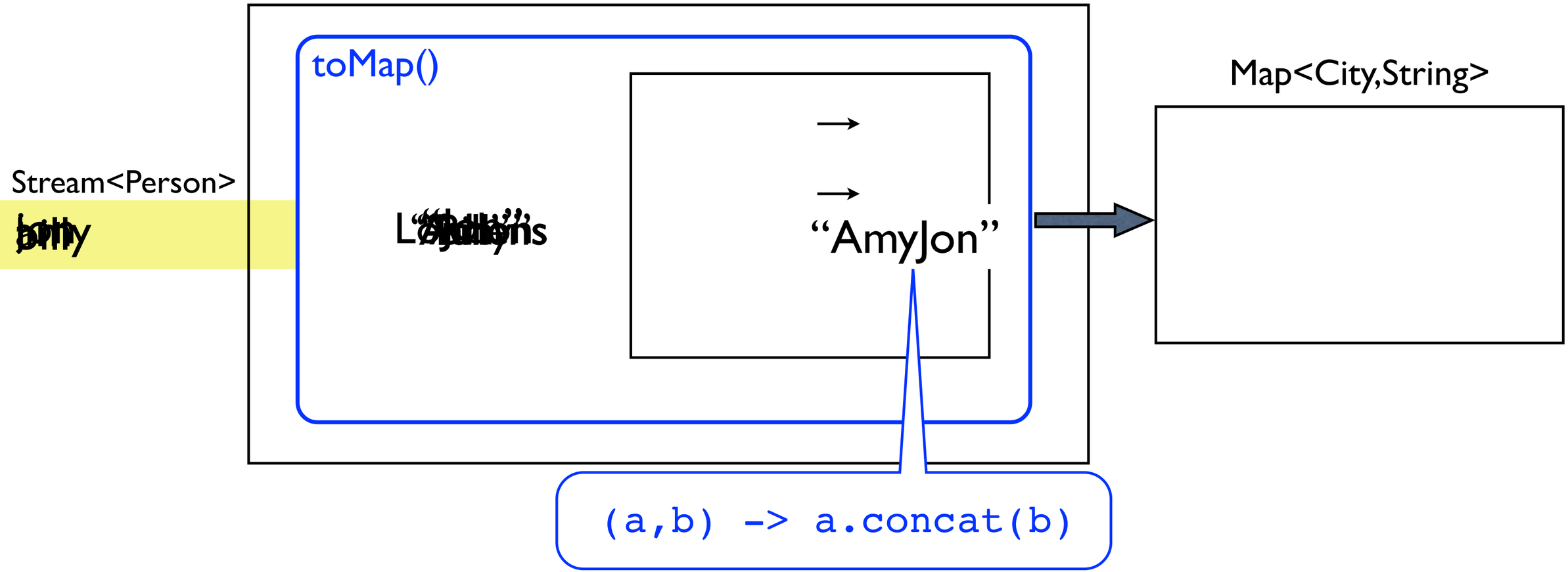
toMap(Function<T,K> keyMapper,
Function<T,U> valueMapper)



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



Collectors API

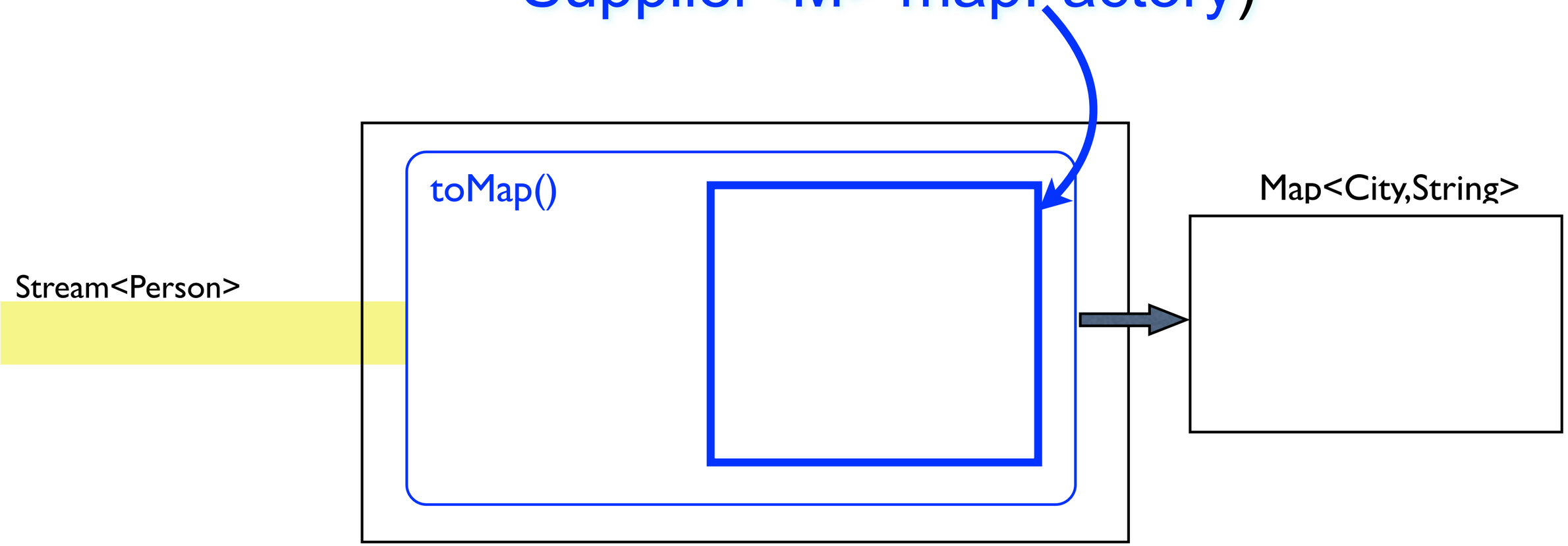
Factory methods in the Collectors class. They produce standalone collectors, accumulating to:

- 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)
```



Collecting to Custom Collections

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

returns

```
Collector<T,?,C>
```

```
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)

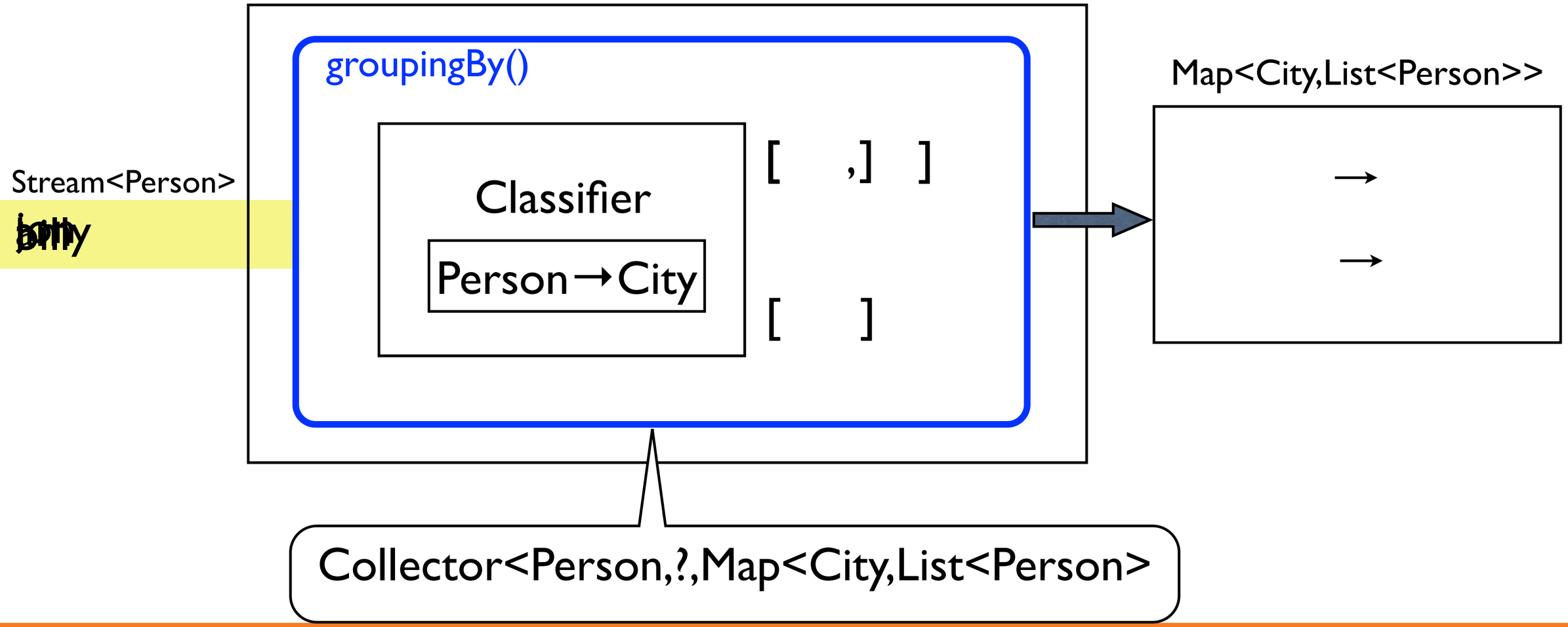
Uses the classifier function to make a *classification mapping*

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<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)

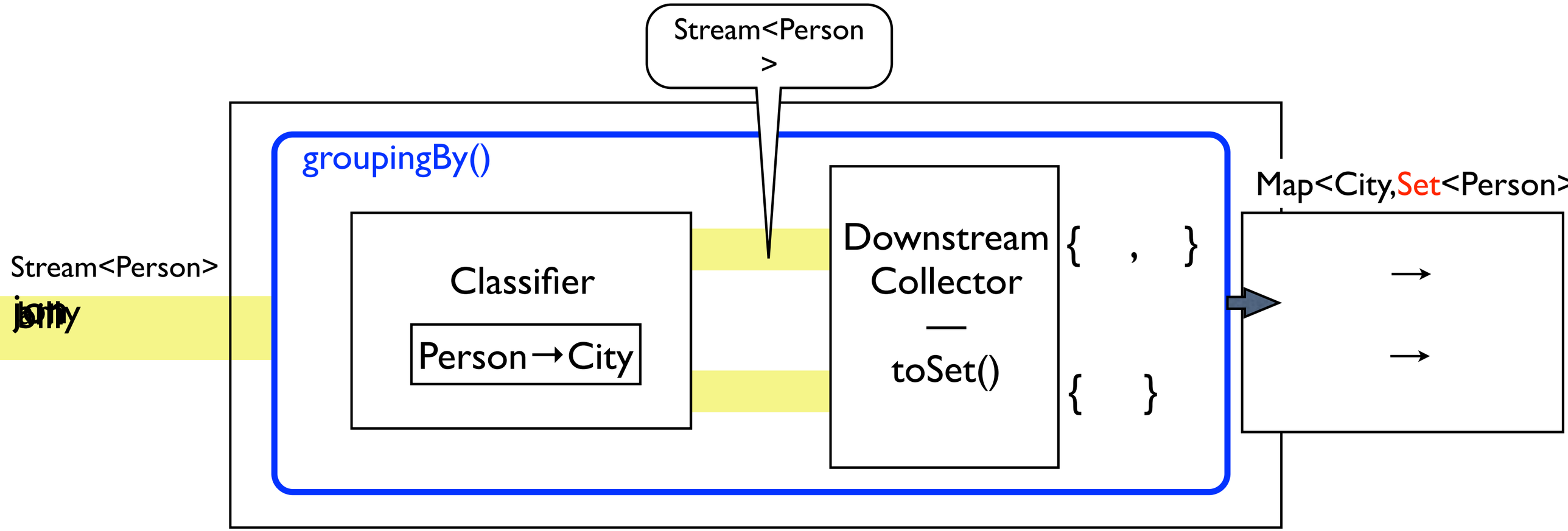
Uses the classifier function to make a *classification mapping* into a container defined by a downstream Collector

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))



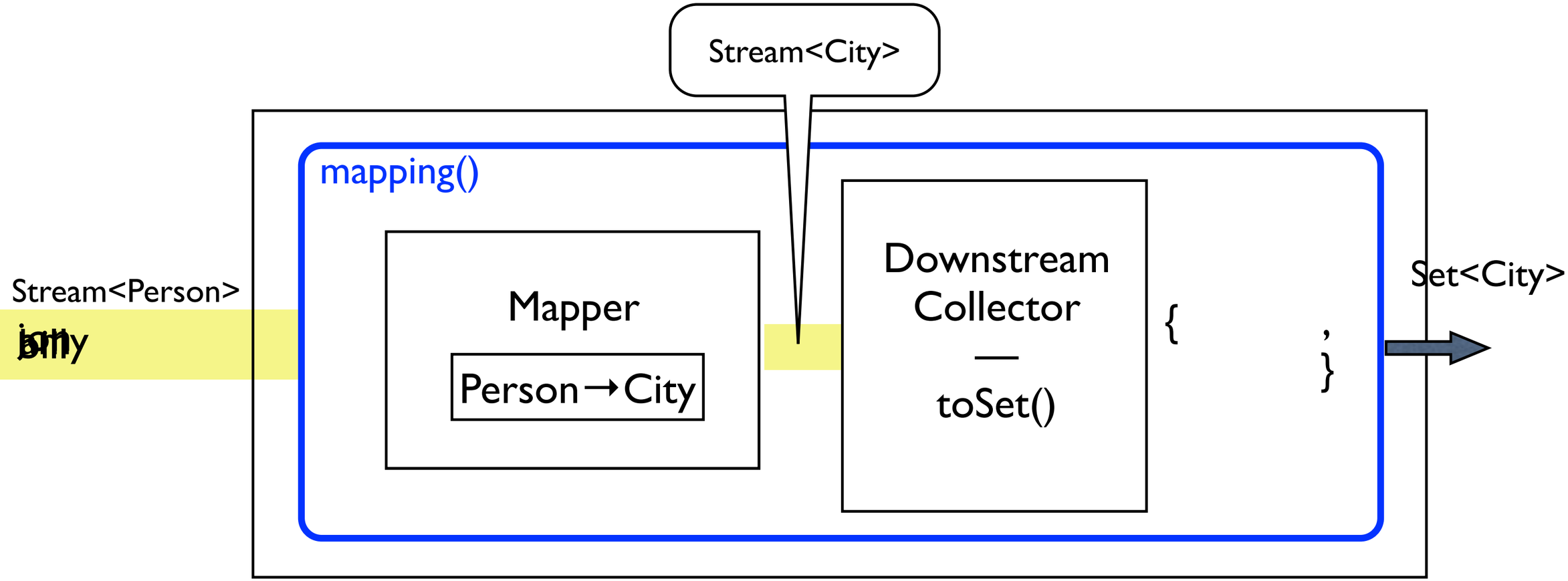
mapping(Function mapper,Collector downstream))

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()));
```

mapping(Function mapper,Collector downstream))



mapping(Function mapper,Collector downstream))

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

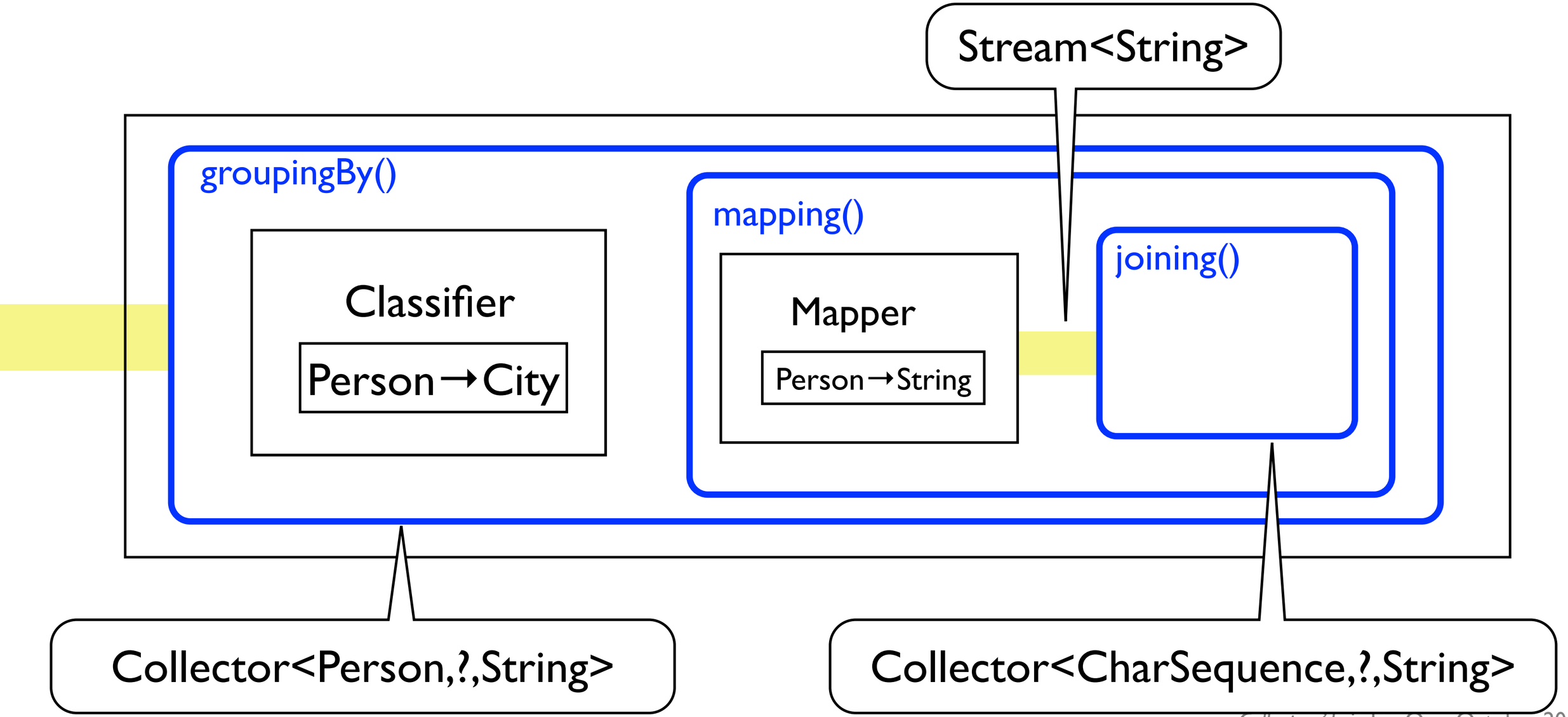
Animation example

```
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())));
```

Nesting Collectors



Dualling Convenience Reductions

Terminal operation	Collectors factory method
`count()`	`counting()`
`max()` `min()`	`maxBy()` `minBy()`
`sum()` `average()`	`summingXxx()` `averagingXxx()`
`summaryStatistics()`	`summarizingXxx()`

`reduce(accumulator)`
`reduce(identity, accumulator)`
`reduce(identity, accumulator, combiner)`
	`reducing(accumulator)`
	`reducing(identity, accumulator)`
	`reducing(identity, accumulator, combiner)`

Streams II

Demo and labs