Navigating the Stream API

Maurice Naftalin
Morningside Light Ltd

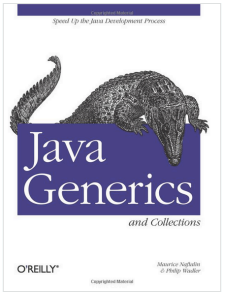
Code font??
Colour-emphasise example devt

Thank JFokus, thank you

How is this talk different from all other lambda talks? — not motivational, sharply focussed on exploring the API from the developer's viewpoint, test at the end.

Maurice Naftalin

Developer, designer, architect, teacher, learner, writer



Co-author

Current Projects

2

Another way of saying that I don't have a real job
8th April 2014, 208 pages

Navigating the Stream API

- Fundamentals
- API Overview
- Stream Operations
- Let's Push the Boat Out!

Lame jokes, test but it's of me

Streams – Why?

- Intention: replace loops for aggregate operations

instead of writing this:

```
Set<City> shortCities = new HashSet<>();

for (Person p : people) {
   City c = p.getCity();
   if (c.getName().length() < 4 ) {
       shortCities.add(c);
   }
}
```


Do code before 2nd bullet

I said not motivational, I'm assuming you have been to Brian's talks, Kool-Aid

Parallel version would be much longer - provide it?

SELECT P.NAME FROM PERSON P WHERE LENGTH(P.NAME) < 4

Streams – Why?

- Intention: replace loops for aggregate operations

instead of writing this:

```
Set<City> shortCities = new HashSet<>();

for (Person p : pe
   City c = p.getC
   if (c.getName()
       shortCities.add(c);
   }
}
```

we're going to write this:

```
Set<City> shortCities = people.stream()
    .map(Person::getCity)
    .filter(c -> c.getName().length() < 4)
    .collect(toSet());
```


Mention method reference

Streams – Why?

- Intention: replace loops for aggregate operations
  - more concise, more readable, composable operations, parallelizable

instead of writing this:

```
Set<City> shortCities = new HashSet<>();

for (Person p : pe
   City c = p.getC
   if (c.getName()
       shortCities.add(c);
   }
}
```

we're going to write this:

```
Set<City> shortCities = people.parallelStream()
    .map(Person::getCity)
    .filter(c -> c.getName().length() < 4)
    .collect(toSet());
```

6

I said not motivational, I'm assuming you have been to Brian's talks, Kool-Aid
Parallel version would be much longer - provide it?
Generate SQL, like LINQ?

On The Other Hand...

Instead of writing

```
Map<City,List<Name>> namesByCity = new HashMap<>();

for (Person p : people) {
   City c = p.getCity();
   if (! namesByCity.containsKey(c)) {
      namesByCity.put(c, new ArrayList<>());
   }
   namesByCity.get(c).add(p.getName());
}
```

On The Other Hand...

Instead of writing

```
Map<City,List<Name>> namesByCity = new HashMap<>();

for (Person p : people) {
   City c = p.getCity();
   if (!namesByCity.containsKey(c)) {
      namesByCity.put(c, new ArrayList<Name>());
   }
   namesByCity.get(c).add(p.getName());
}
```

We're going to write

```
Map<City, List<Name>> namesByCity = people.stream()
    .collect(groupingBy(Person::getCity,
          mapping(Person::getName,Collectors.toList())));
```

Does this mean that streams are hard to use?

No, it means that you've come to the right talk

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, filesystems, strings, IO buffers,...
- Can be
  - parallel
  - infinite
- Primitive specialisations: IntStream, LongStream, DoubleStream

Like Unix streams

Navigating the Stream API

- Fundamentals
- API Overview
- Stream Operations
- Let's Push the Boat Out!

The Life of a Pipeline

- Born at a source
- Successive intermediate operations
  - often use lambdas/method references to transform (or drop) values
  - operation itself returns a new stream that will carry transformed values
- Dies at a terminal operation
  - terminal operations "pull" values down the pipeline

Stream Operations – the Map

Intermediate

Stateless

```
filter
map
flatMap
peek
```

Stateful

```
distinct
limit
substream
sorted
```

Terminal

```
forEach
forEachOrdered
```

Reduction

```
reduce
min,max
count
```

MutableReduction

```
collect
toArray
```

Search

```
anyMatch
allMatch
findAny
findFirst
```

SM: sorted different because must collect all values before proceeding

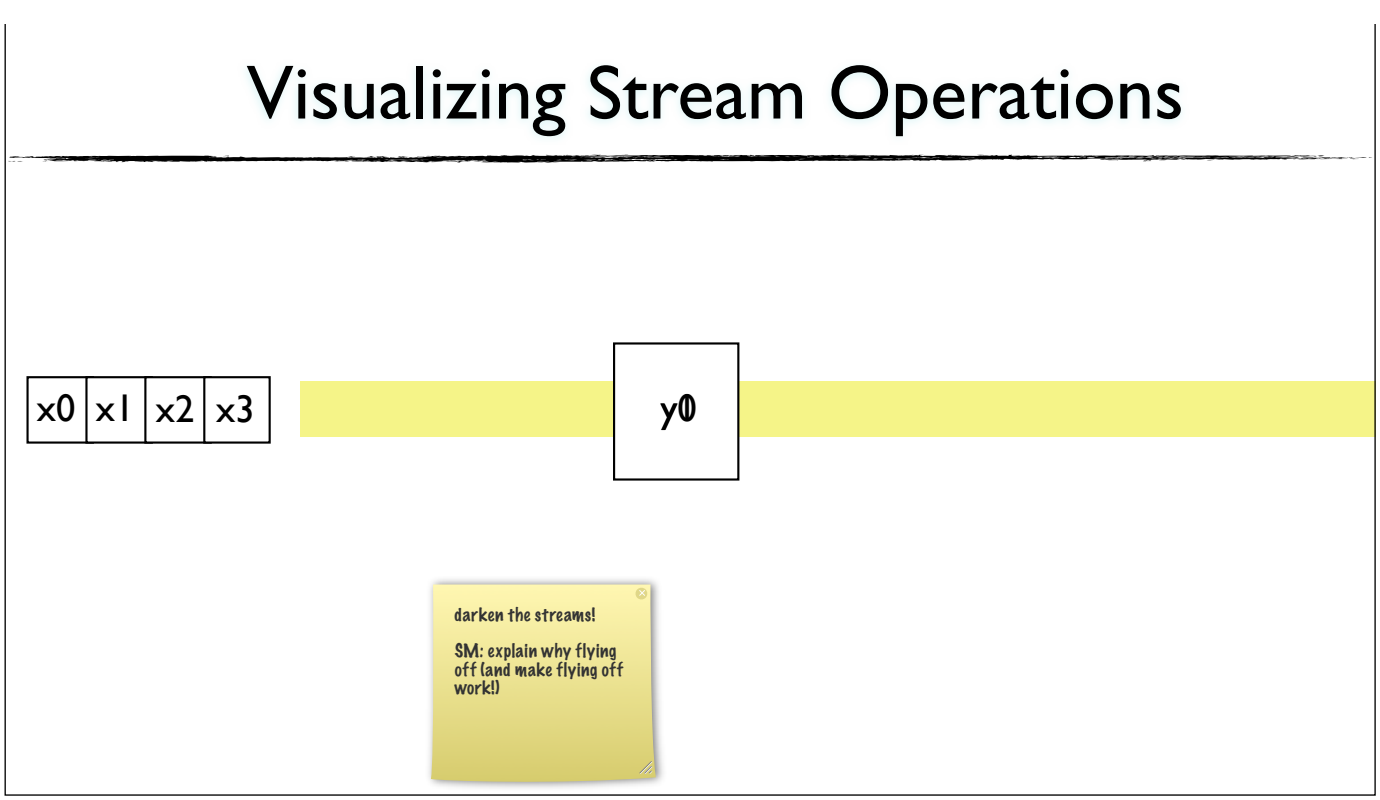
Navigating the Stream API

- Fundamentals
- API Overview
- Stream Operations
  - Intermediate Stateless Operations
- Let's Push the Boat Out!

Intermediate Operations

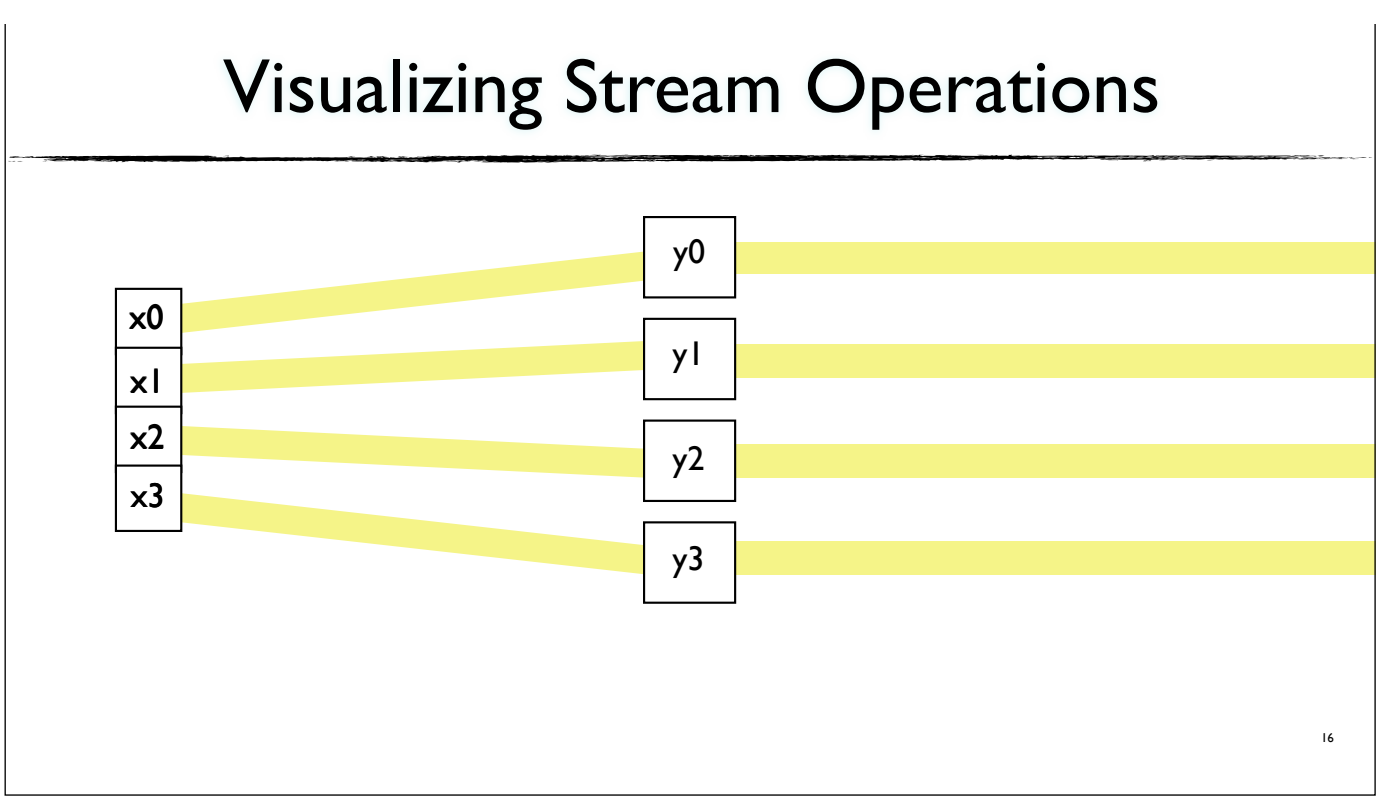
- return new **`Streams`**
- *lazy*: they create a new **`Stream`**, not new elements

```
Stream<City> cities =
     people.stream()
               .map(p -> p.getCity());
```



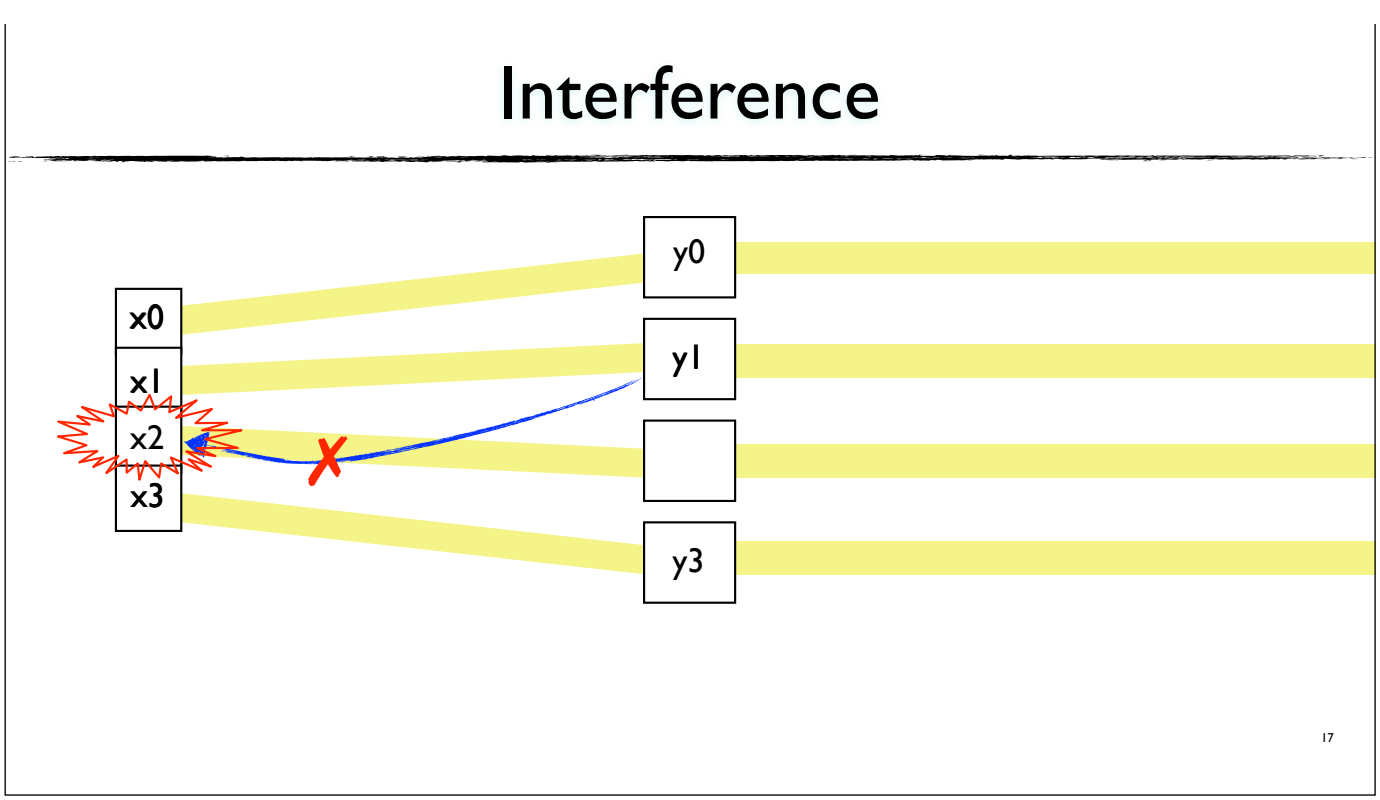
For visual learners

But note: principle of processing mode equality



This neat picture is not realistic!

No guarantees on ordering beyond what's required to maintain the semantics of the the operation.



API documentation says don't do this, ever. In fact, **no-one** should fritz with the source of an executing pipeline, unless it's concurrent.

Prime directive

Stateless Intermediate Operations

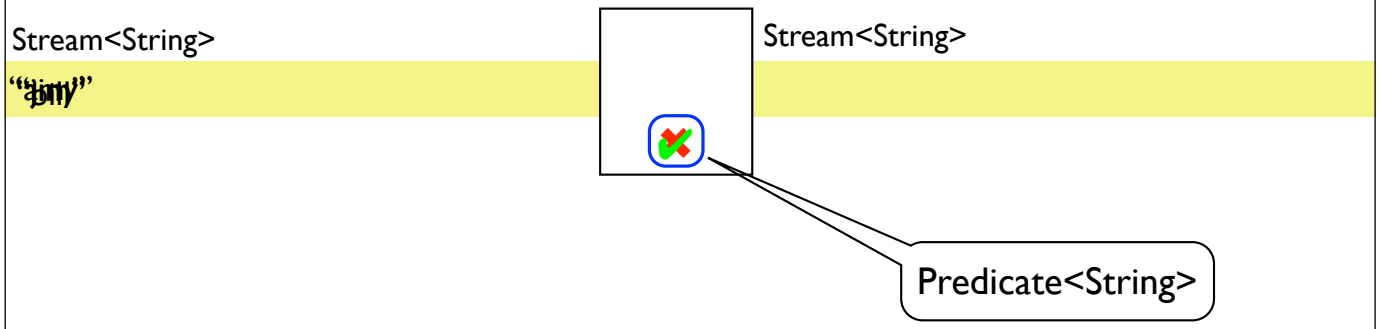
name	returns	interface used	λ signature
filter	Stream<T>	Predicate<T>	T → boolean
map	Stream<U>	Function<T,U>	T → U
flatMap	Stream<R>	Function<T,Stream<R>>	T → Stream<R>
peek	Stream<T>	Consumer<T>	T → void

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

only the type bounds, here and everywhere in the talk
filter takes Predicate<? super T> etc

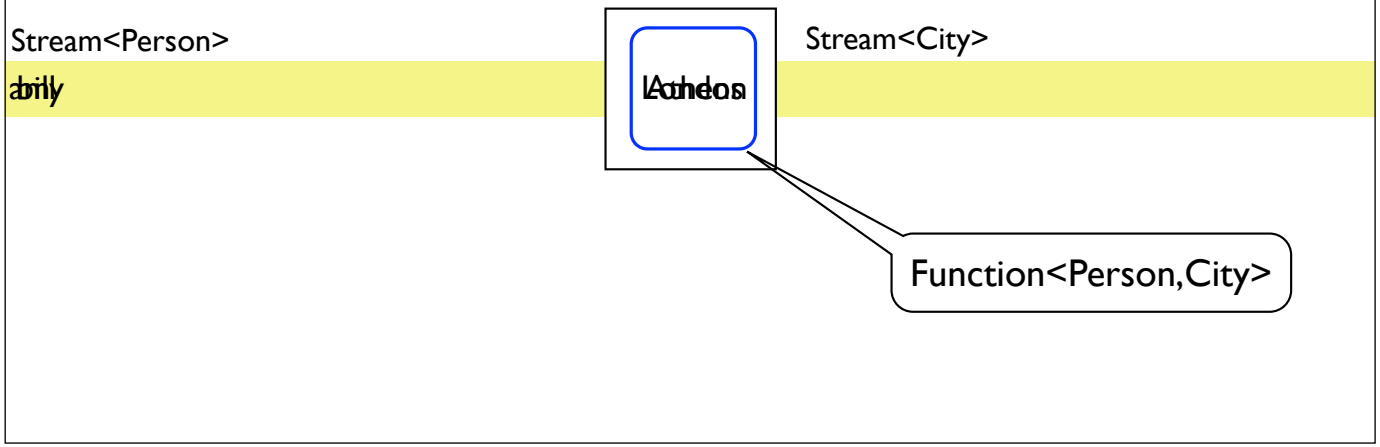
filter()

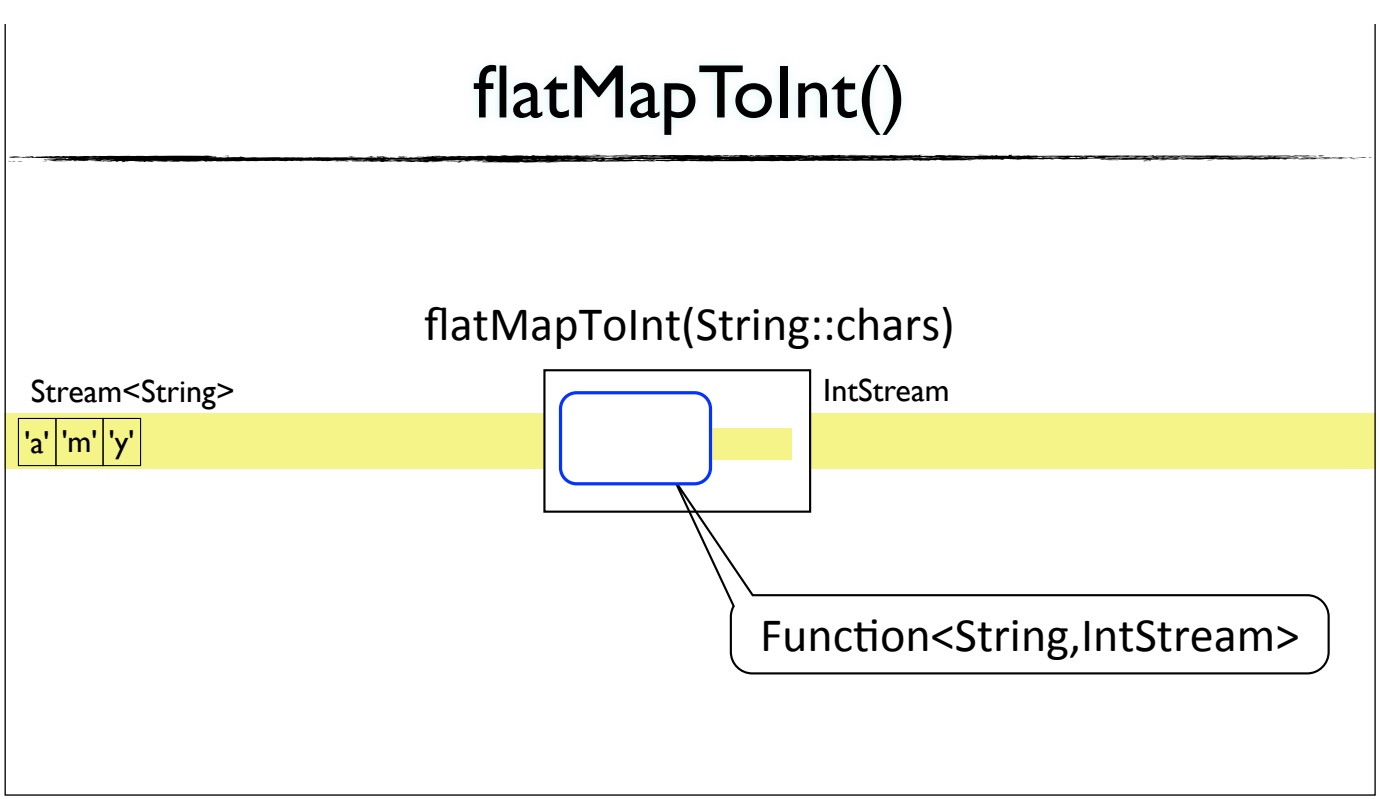
filter(s -> s.length() < 4)



map()

map(Person::getCity)





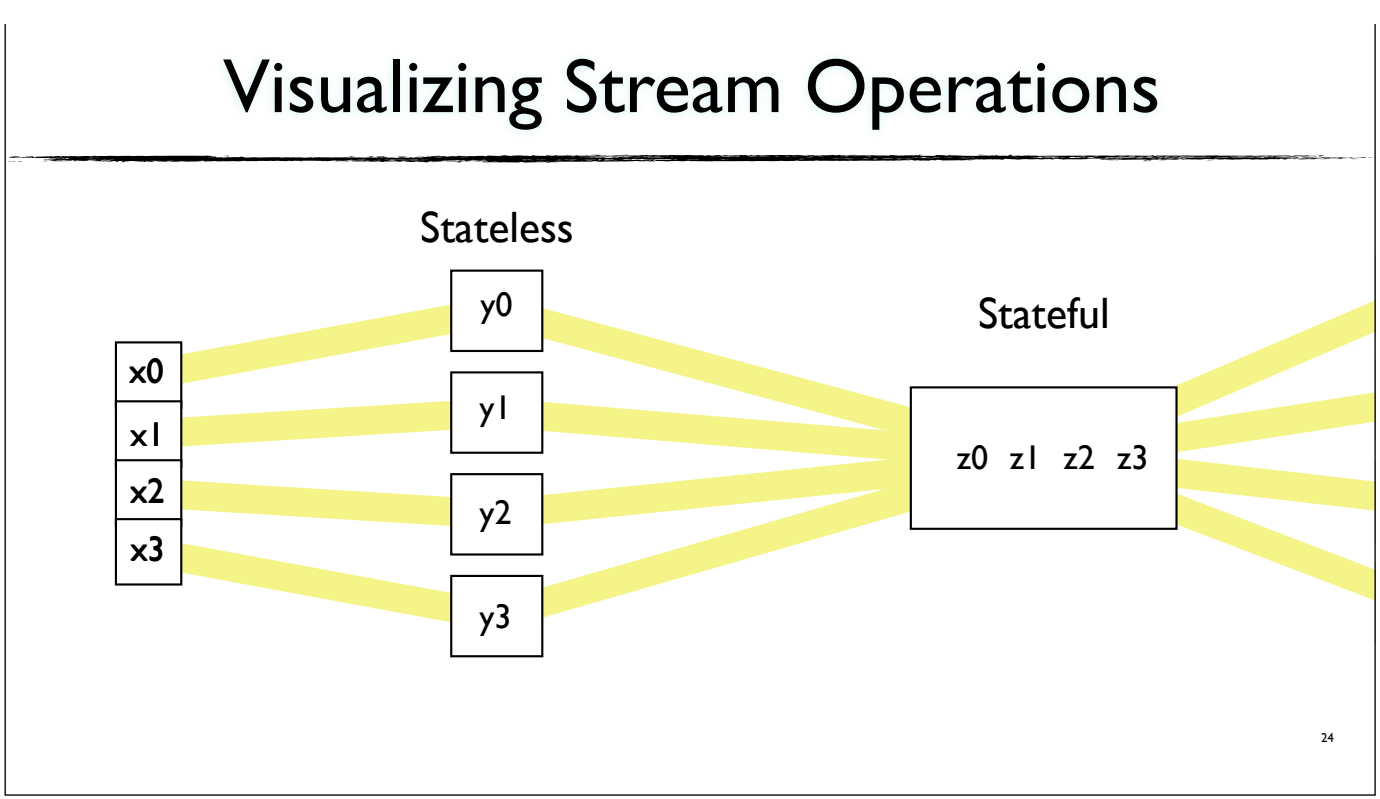
It’s ok to hold the input value inside the blue box temporarily — it’s never mutated, so just a convenient visual notation

Navigating the Stream API

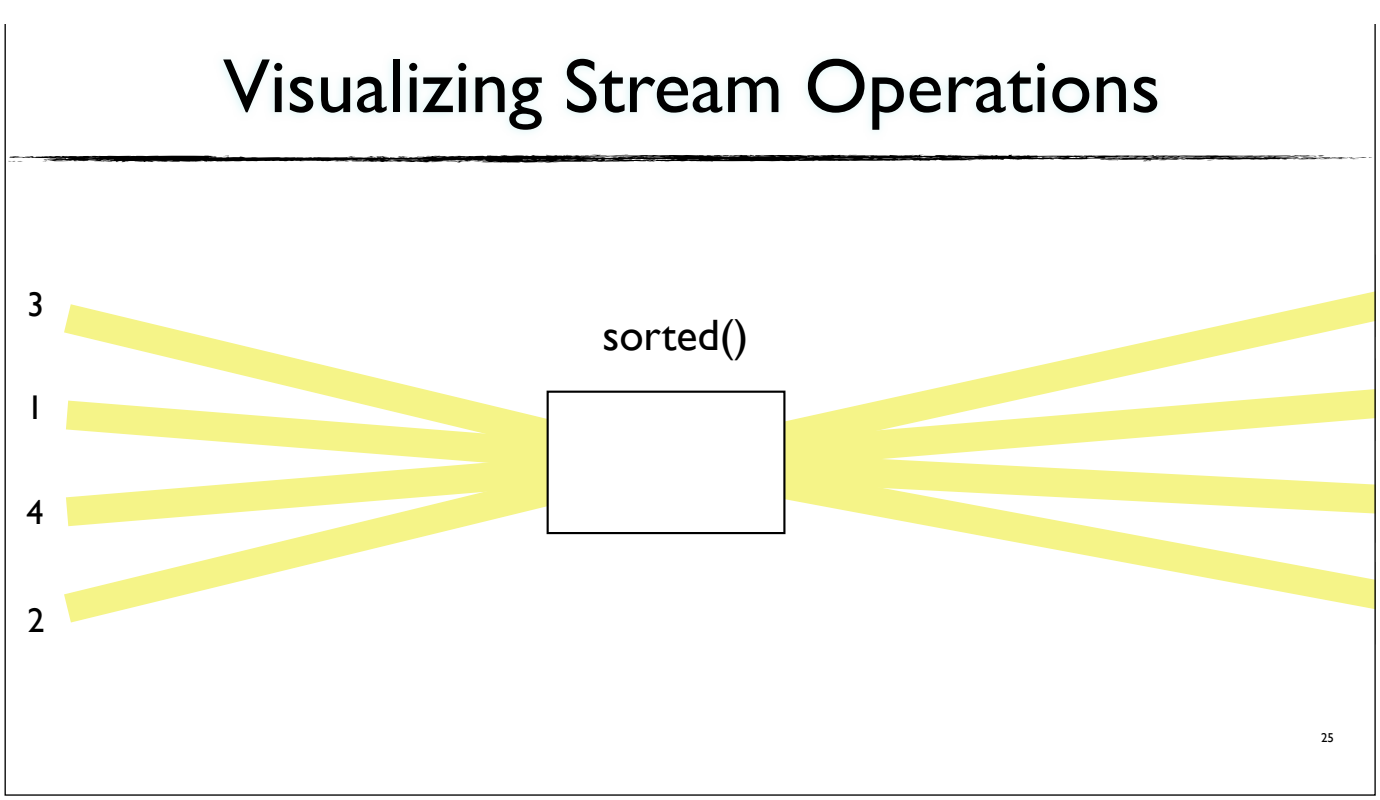
- Fundamentals
- API Overview
- Stream Operations
  - Intermediate Stateful Operations
- Let's Push the Boat Out!

Stateful Intermediate Operations

name	returns	type used	λ signature
`limit`	`Stream<T>`	`long`	
`substream`	`Stream<T>`	`(long, long)`	
`sorted`	`Stream<T>`	`Comparator<T>`	`(T, T) → int`
`distinct`	`Stream<T>`		



Not all of them, of course



Sorted is always a barrier
Not all of them, of course

Navigating the Stream API

- Fundamentals
- API Overview
- Stream Operations
  - Terminal Operations
- Let's Push the Boat Out!

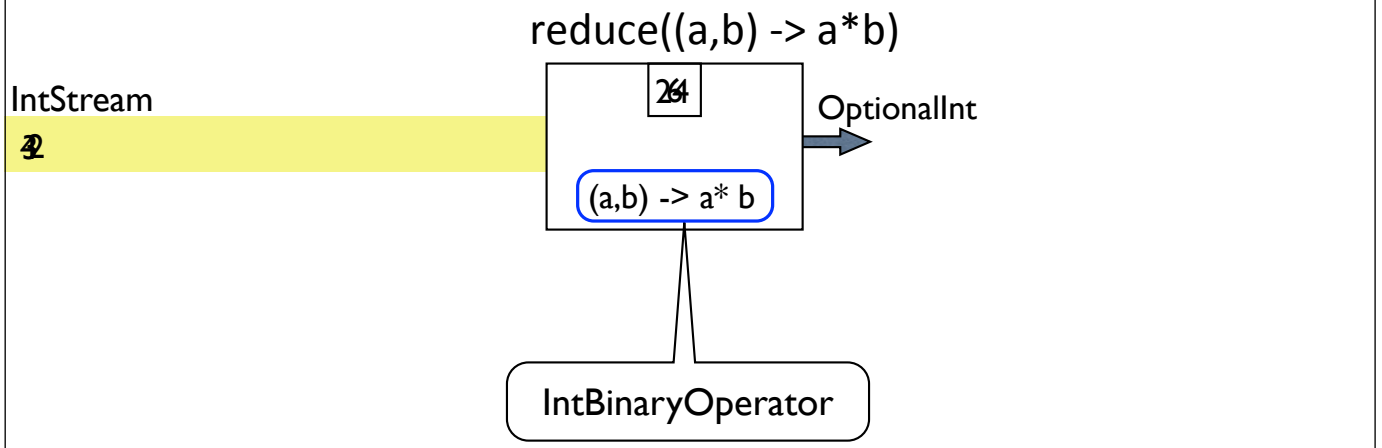
Terminal Operations

- return non-Stream values
- typically *eager*: they force evaluation of their stream

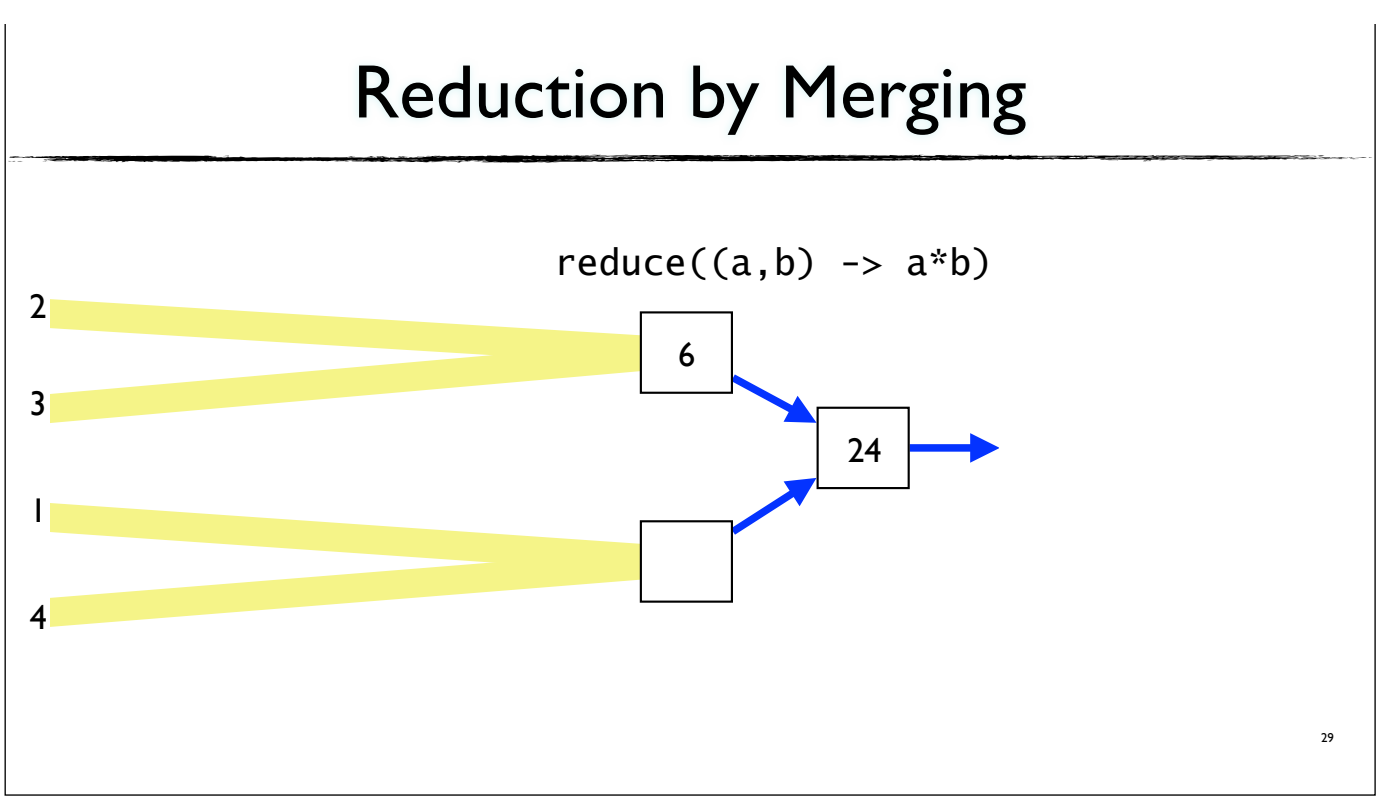
```
Set<City> cities =
     people.stream()
               .map(p -> p.getCity())
               .collect(toSet());
```


Result is a non-stream value

Reduction by Accumulation



Note exhaustion of input stream
Traditional FP way of thinking about reduction
BUT: every reduction must be doable in parallel



With larger set, lots of parallel working, (finally must be serialised, of course)

There must be a symmetric variant of any reduction

Reduction Operations

name	returns	interface used	λ signature
`reduce`	`Optional<T>`	`BinaryOperator<T>`	`(T, T) → T`
`min, max`	`Optional<T>`	`Comparator<T>`	`(T, T) → int`
`count`	`long`		

another overload of reduce also takes an accumulator function, but there is no overload with *only* an accumulator function

primitive versions have others eg sum and statistics

Mutable Reductions

Mutable reductions collect the values of a stream into a "container"

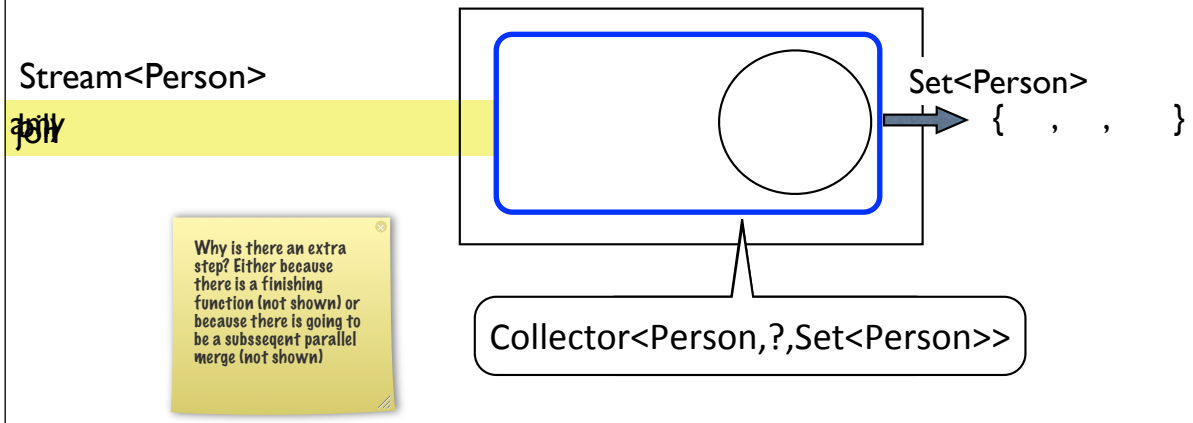
```
public <R,A> R collect(Collector<T,A,R>)
```

- What is a collector?
  - aggregates values of type T into a container, of type R
  - A is an intermediate type, used to accumulate T values before they are "finished" into an R
    - for example, StringBuilder is used an intermediate type when Strings are joined

also toArray
A is often an implementation detail

Collecting

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



Why is there an extra step? Either because there is a finishing function (not shown) or because there is going to be a subsseqent parallel merge (not shown)

Collectors

- Mostly you'll use predefined Collectors
  - Factory methods in the java.util.stream.Collectors
    - counting
    - summing/averaging/summarizing (for each of int, long, double)
    - joining (for Strings)
    - toList/Map/Set
    - reducing
    - groupingBy(x3)
    - mapping

Think of the container in each case

Collectors.groupingBy(Function classifier)

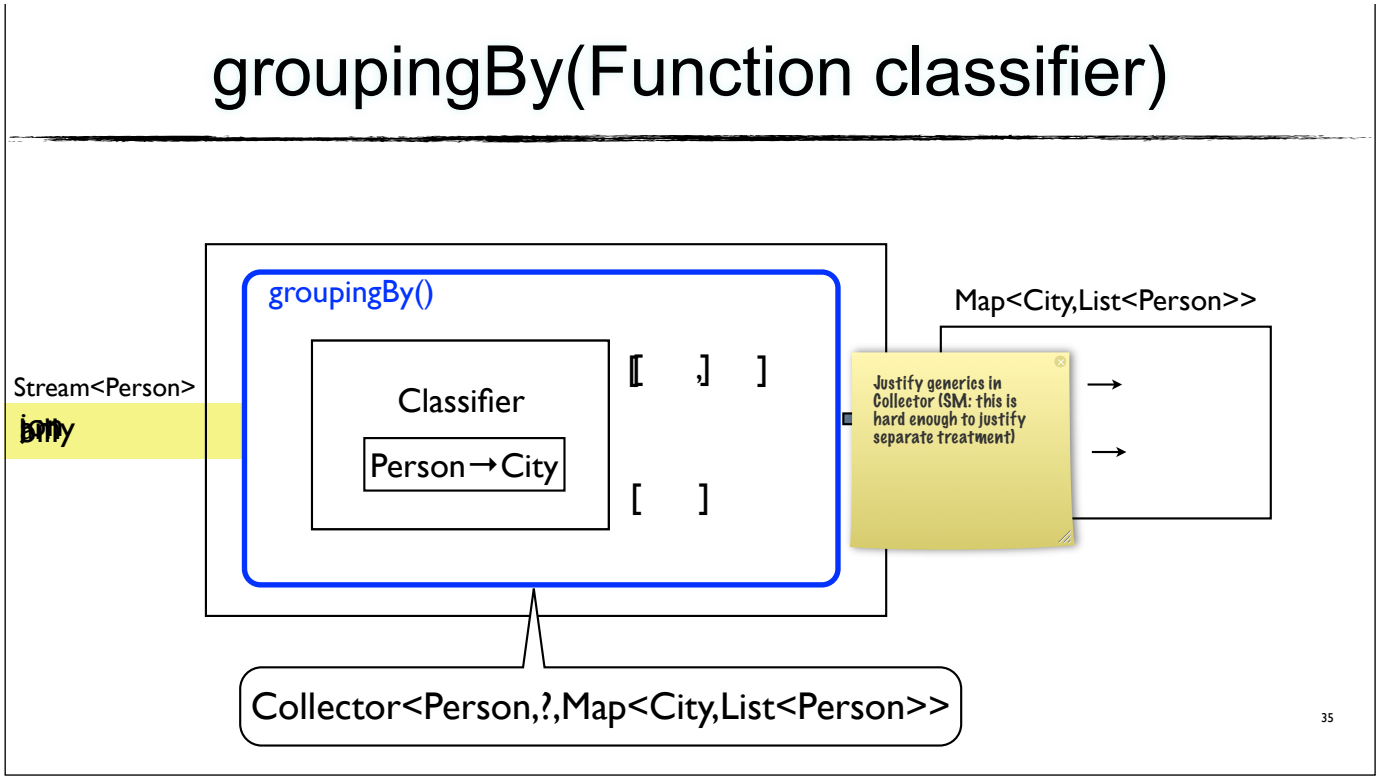
Uses the classifier function to make a *classification mapping*

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

Persons are classified according to the City that classifier gives them;
same-classified Persons are put into a List

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


List is the default



What if you don't want to put them into a List, though?

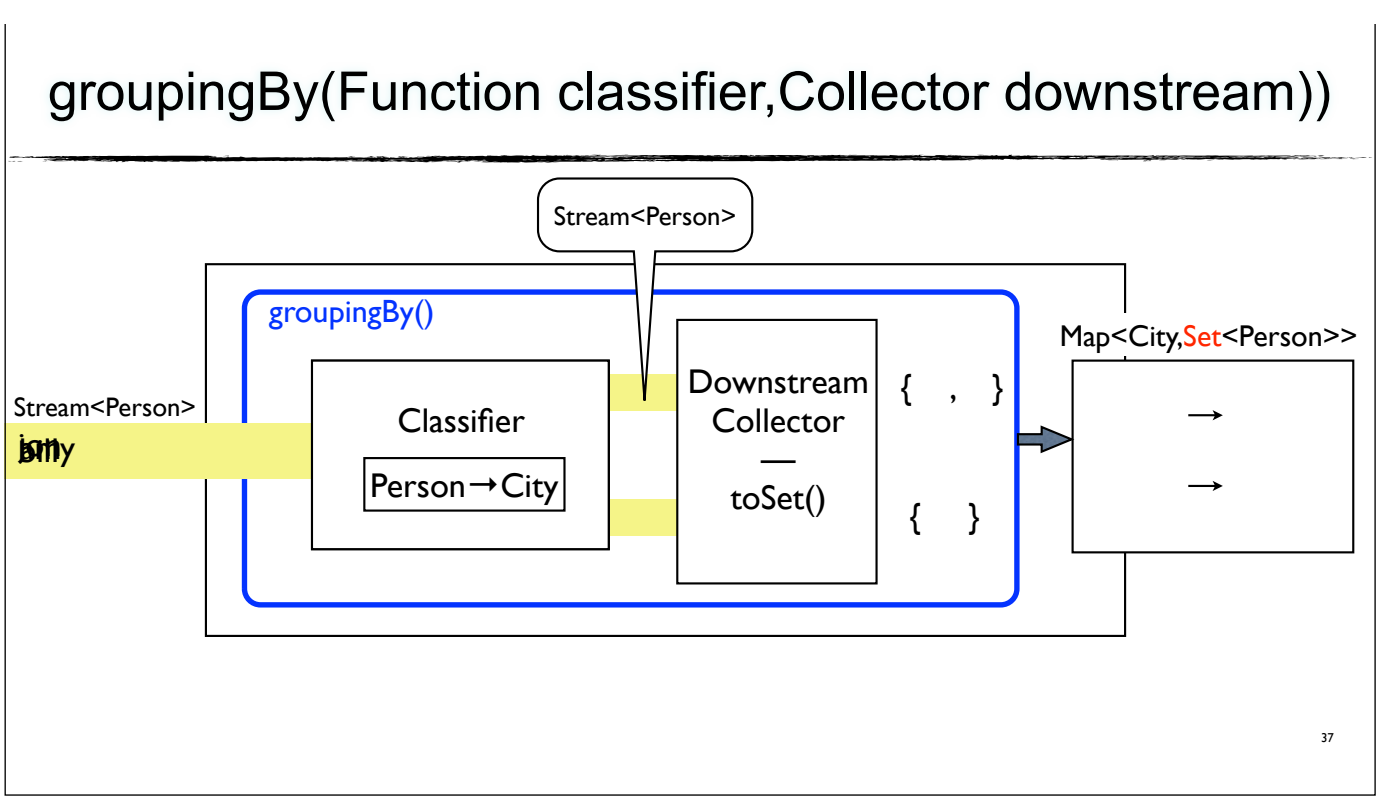
groupingBy(Function classifier,Collector downstream))

Uses the classifier function to make a *classification mapping*

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

Persons are classified according to the City that classifier gives them;
same-classified Persons are put into
a container defined by the downstream collector

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



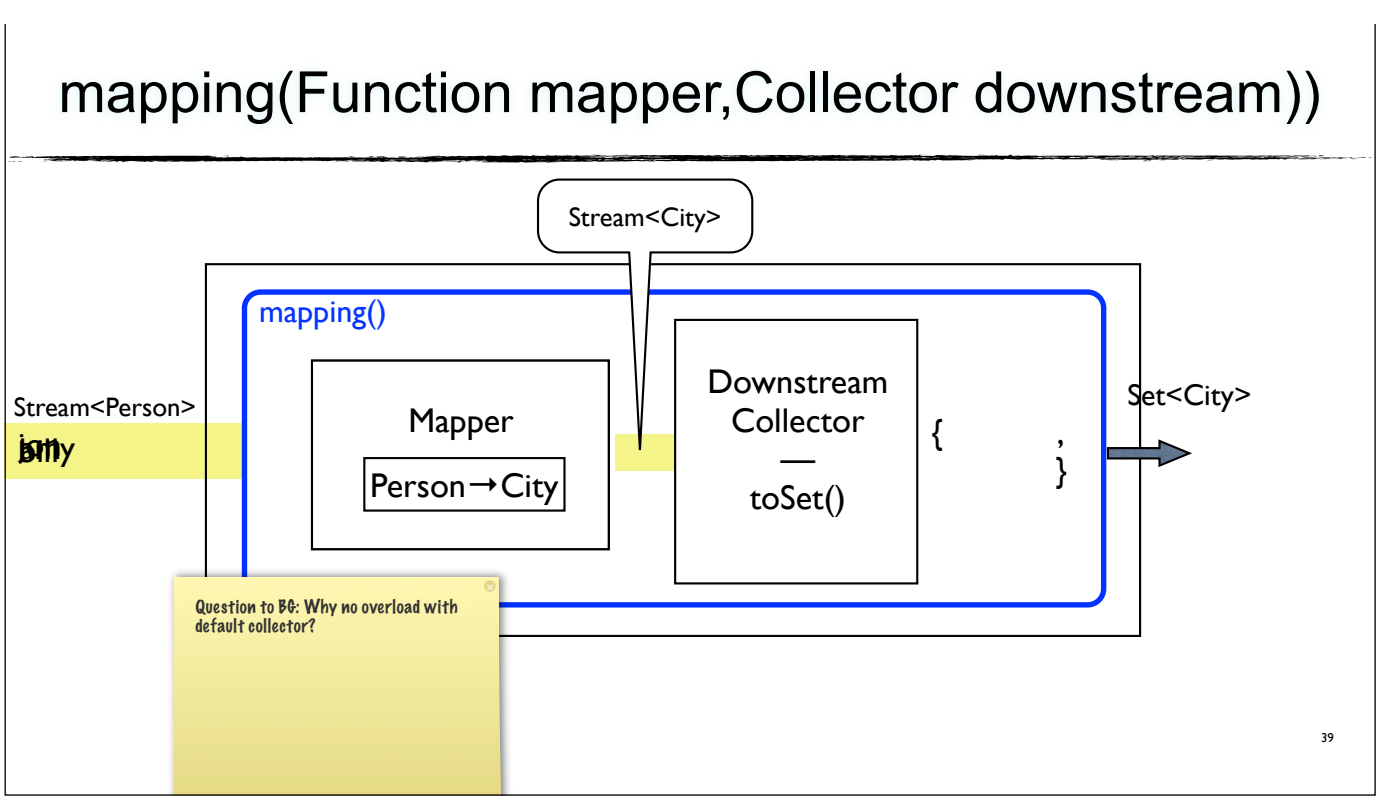
Each T goes to the collector for its classification key

mapping(Function mapper,Collector downstream))

Adapts a Collector accepting elements of one type
to one accepting elements of another
by applying a mapping function to each input element
before accumulation by the downstream collector.

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


API documentation



SM: Stupid example, you would just use an upstream map(). Maybe leave, but explain useful as downstream collector, see examples at end.

Navigating the Stream API

- Fundamentals
- API Overview
- Stream Operations
- Let's Push the Boat Out!

Some Problems

From a stream of Person, compute

- List of the adults
- Set of ages of the adults
- Listing of people by age
- Population by age
- Names by age
- Most popular age
- Bonus: Most popular ages

First 5 doable from presentation, 6th needs something more

Bonus is a “problem for the reader”

Mavens: help out, but only if...

Problem 1

18

~~List of adults (age > 21)~~

```
List<Person> adults =
        people.stream()
                .filter(p -> p.getAge()  > 18 )
                .collect(toList());
```

Assuming static imports of Collectors factory methods
Building next problems up from previous ones

Problem 2

Set of ages of the adults

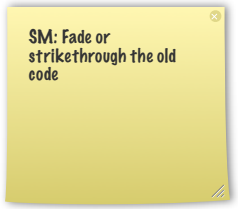
```
Set<Integer> adultAges =
        people.stream()
                .filter(p -> p.getAge() > 21)
                .collect(toList());
```

Problem 2

Set of ages of the adults

```
Set<Integer> adultAges =
        people.stream()
                .filter(p -> p.getAge() > 21)

                .collect(toList());
```



Problem 2

Set of ages of the adults

```
Set<Integer> adultAges =
        people.stream()
                .filter(p -> p.getAge() > 21)
                .map(Person::getAge)
                .collect(toList());
```

Problem 2

Set of ages of the adults

```
Set<Integer> adultAges =
        people.stream()
                .filter(p -> p.getAge() > 21)
                .map(Person::getAge)
                .collect(toSet());
```

Problem 3

People by Age

```
Map<Integer,List<Person>> peopleByAge =
        people.stream()
                .filter(p -> p.getAge() > 21)
                .map(Person::getAge)
                .collect(toSet());
```

Can we keep any of this? (No)

Problem 3

People by Age

```
Map<Integer,List<Person>> peopleByAge =
        people.stream()
                .filter(p -> p.getAge() > 21)
                .map(Person::getAge)
                .collect(groupingBy( ?? ));
```

Problem 3

People by Age

```
Map<Integer,List<Person>> peopleByAge =
        people.stream()
                .filter(p -> p.getAge() > 21)
                .map(Person::getAge)
                .collect(groupingBy(Person::getAge));
```

Problem 3

People by Age

```
Map<Integer,List<Person>> peopleByAge =
        people.stream()
                .collect(groupingBy(Person::getAge));
```

Problem 4

Population by Age

```
Map<Integer,Long> populationbyAge =
        people.stream()
                .collect(groupingBy(Person::getAge));
```

Problem 4

Population by Age

```
Map<Integer,Long> populationByAge =
        people.stream()
                .collect(groupingBy(Person::getAge,
                                    ?? ));
```

Problem 4

Population by Age

```
Map<Integer,Long> populationByAge =
        people.stream()
                .collect(groupingBy(Person::getAge,
                                    counting()));
```

Problem 5

Names by Age

```
Map<Integer,List<Name>> namesByAge =
        people.stream()
                .collect(groupingBy(Person::getAge,
                                    counting()));
```

Problem 5

Names by Age

```
Map<Integer,List<Name>> namesByAge =
        people.stream()
                .collect(groupingBy(Person::getAge,
                                    ??      ));
```

Problem 5

Names by Age

```
Map<Integer,List<Name>> namesByAge =
        people.stream()
                .collect(groupingBy(Person::getAge,
                        mapping(  ?? , ??  )));
```

Problem 5

Names by Age

```
Map<Integer,List<Name>> namesByAge =
        people.stream()
                .collect(groupingBy(Person::getAge,
                        mapping(Person::getName,toList())));
```

Problem 6

Most Popular Age

```
Optional<Integer> modalAge = populationByAge
                .entrySet()
                .stream()
                .max(Entry.comparingByValue())
                .map(Entry::getKey);
```

populationByAge is Map<Integer,Long>

Problem 7

Most Popular Ages

```
Optional<Set<Integer>> modalAges = populationByAge
        .entrySet()
        .stream()
        .collect(groupingBy(Entry::getValue, mapping(Entry::getKey, toSet())))
        .entrySet()
        .stream()
        .max(Entry.comparingByKey())
        .map(Entry::getValue);
```

Summary

- Collections processing with streams looks very different
- But the payoff is huge!
- Need to learn to think differently – but it's not that hard!

If you're invested in Java, this is a great development
Speaking as a person who's written on the two big language changes Java 5 and Java 8

Resources

- Brian Goetz talk @ JavaOne 2014 (parleys.com) http://goo.gl/OEjk1h
- State of the Lambda, Libraries Edition

  http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html
- java.util.stream API package documentation
- Lambda FAQ (http://lambdafaq.org) – collections material coming real soon!
- Out now: *Functional Programming in Java,* Venkat Subramaniam
- Out soon—honestly!
  - *Java 8 Lambdas in Action,* Raoul Urma, Mario Fusco, Alan Mycroft (Mannning, early access)
  - *Java 8 Lambdas: Pragmatic Functional Programming,* Richard Warburton
  - *Mastering Lambdas: Java Programming in a Multicore World,* Maurice Naftalin

Navigating the Stream API

Questions?