Streams

James Gough
Raoul-Gabriel Urma
Richard Warburton

Streams Outline

- Collection Processing
- What is a Stream?
- Stream Operations & Patterns
- Stream Optimisation

Collection Processing

Collections

- Nearly every Java applications **makes** and **processes** collections
- However processing collections is far from perfect:
  - SQL like operations?
  - How to efficiently process large collections?

SQL-like operations

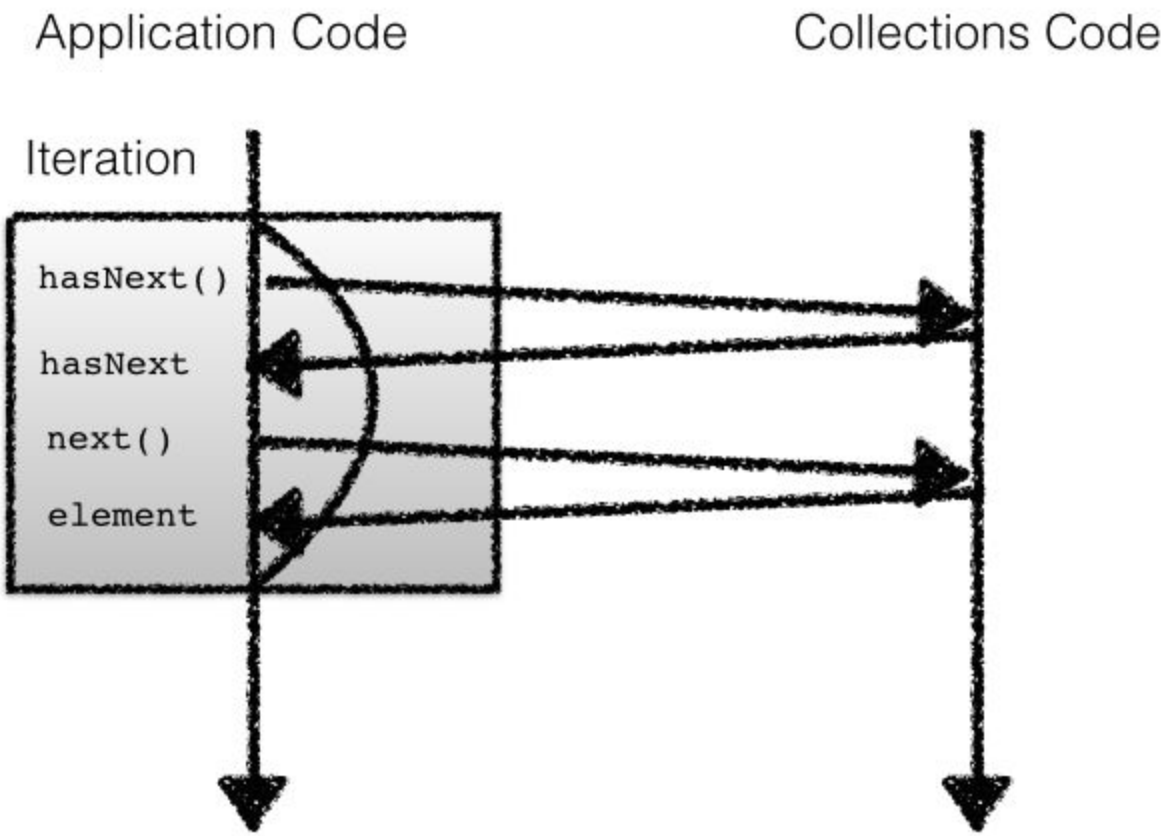
- Many processing patterns are SQL-like
  - **finding** a transaction with highest value
  - **grouping** transactions related to grocery shopping
- Re-implemented every time
- SQL is **declarative**
  - express **what** you expect not **how** to implement a query
  - SELECT id from transactions WHERE value > 1000

Internal and External Iteration

External Iteration

```
int count = 0;
for (Artist artist : artists) {
    if (artist.isFrom("London")) {
        count++;
    }
}
```

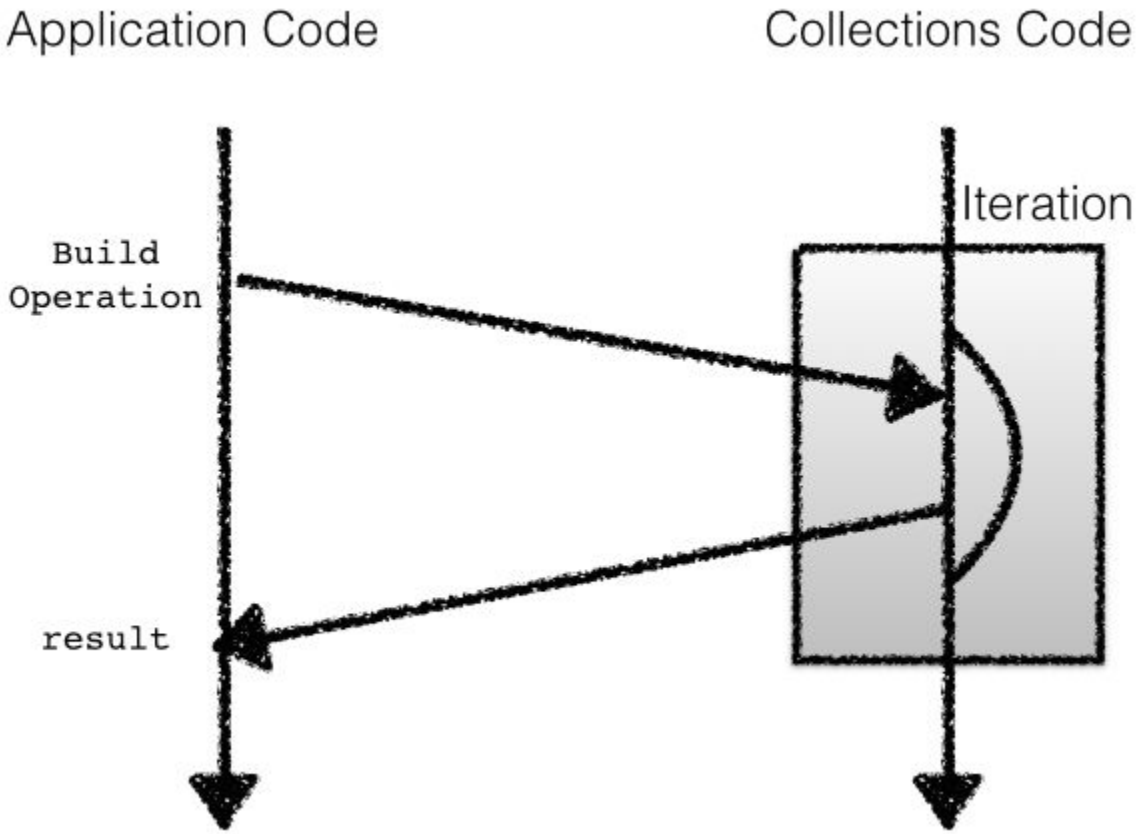
External Iteration



Internal Iteration

```
artists.stream()
        .filter(artist -> artist.isFrom("London"))
        .count();
```

Internal Iteration



Motivation for Internal Iteration

- Inversion of Control
- Decouples the operation from the application code
- Iterators hard-code a threading model

What is a Stream?

Ok cool – so what's a Stream?

- *Informally*: A fancy iterator with database-like operations
- *More formally*: A sequence of elements from a source that supports aggregate operations.

Ok cool – so what's a Stream?

- **Sequence of elements**: a stream provides an interface to a sequenced set of values of a specific element type. However, streams don't actually store elements, they are computed on demand.
  - You can turn a collection into a Stream by calling the method `stream()`

```
Stream<Integer> stream = numbers.stream();
```

Ok cool – so what's a Stream?

- **Source**: Streams consume from a data-providing source such as Collections, Arrays, or IO resources.

Ok cool – so what's a Stream?

- **Aggregate operations**: Streams support database-like operations and common operations from functional programing languages such as `filter`, `map`, `reduce`, `find`, `match`, `sorted` etc.

Two additional properties

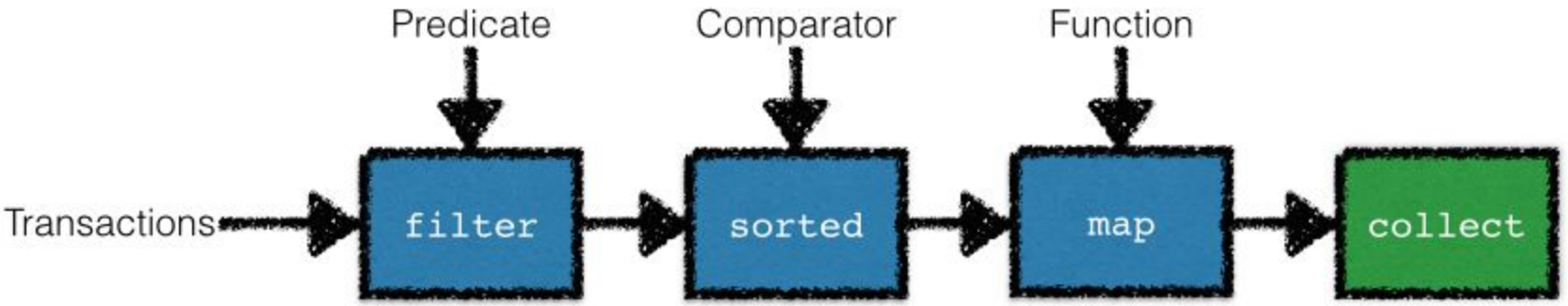
- **Pipelining**: Many stream operations return a stream themselves. This allows operations to be chained and form a larger pipeline as well as certain optimisations (more later).

- **Internal iteration**: In contrast to collections, that are iterated explicitly (“external iteration”), stream operations do the iteration behind the scene for you.

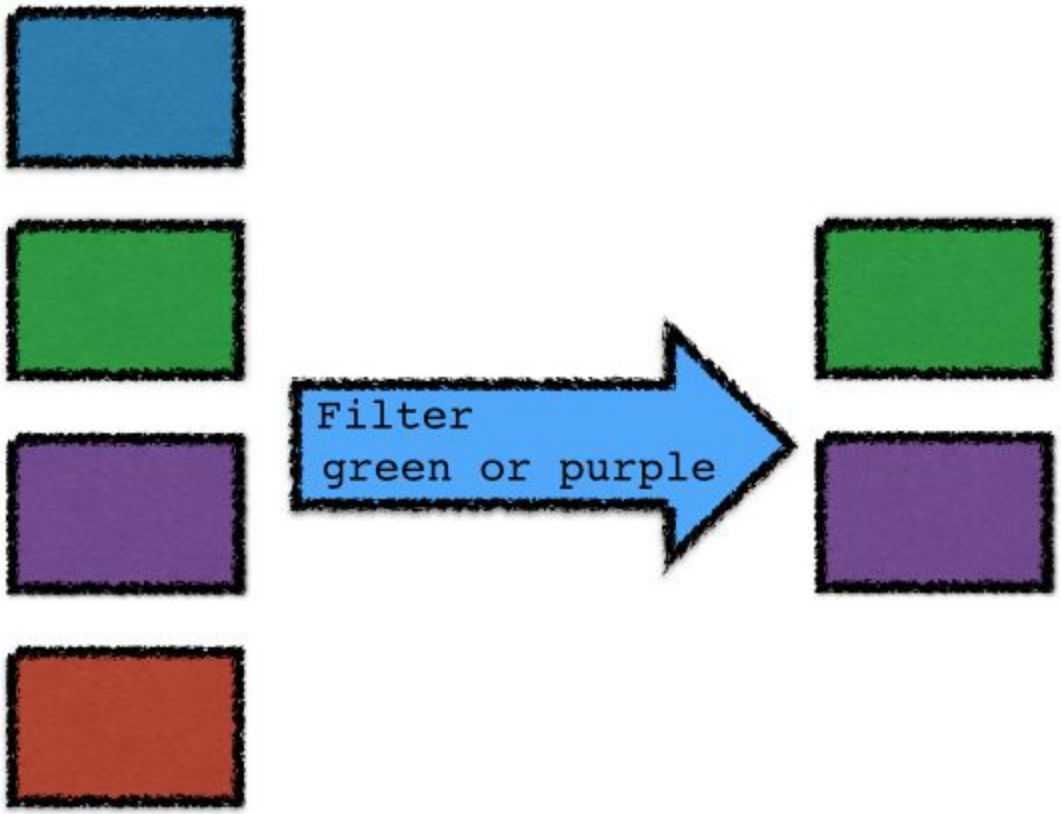
Streams vs Collections

- Collection is like a DVD: the whole movie is available before watching it
- Stream: you are streaming a movie over the internet, frames are computed on demand

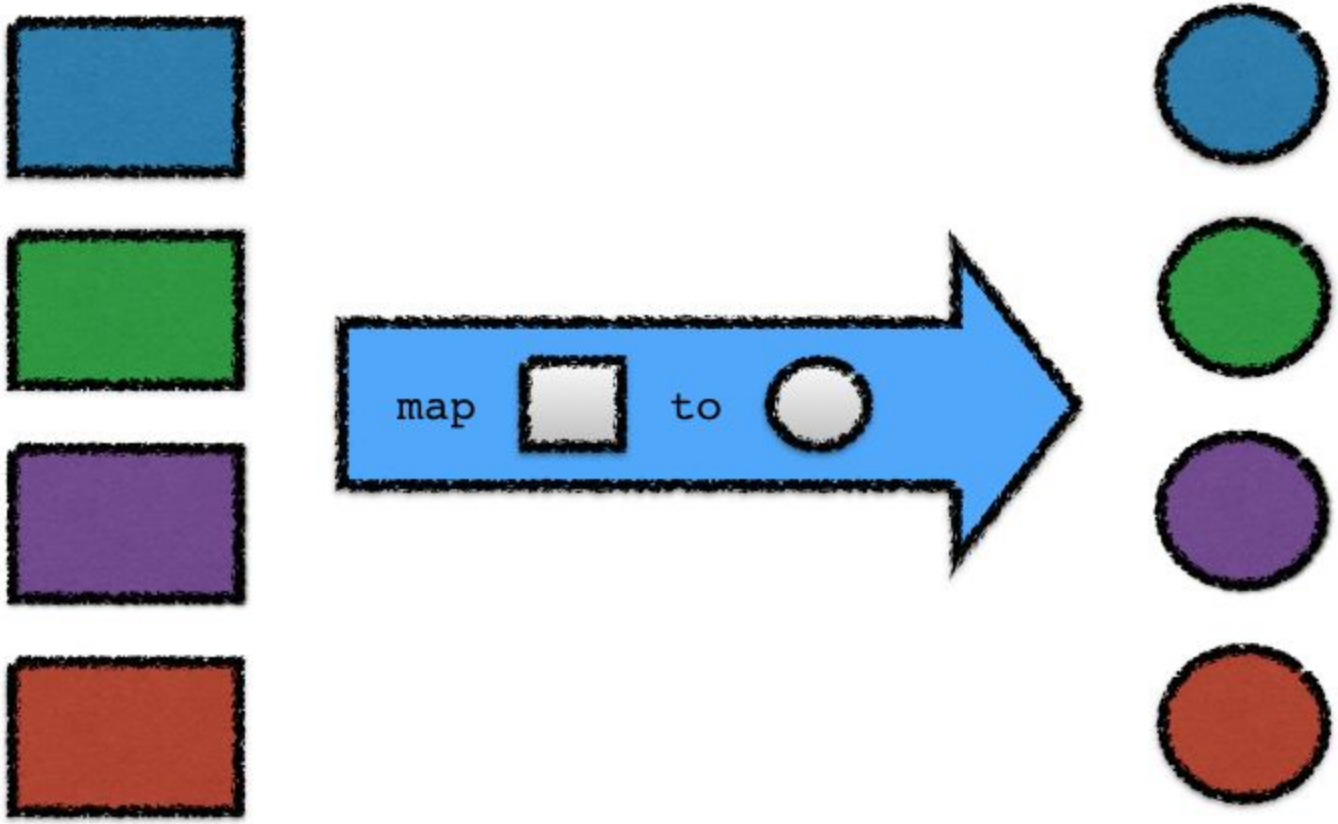
Pipelining


Predicate
Comparator
Function
Transactions
filter
sorted
map
collect

Common Stream Operations


Filter
green or purple

```
List<String> beginningWithNumbers =
    Stream.of("a", "1abc", "abc1")
          .filter(value -> isDigit(value.charAt(0)))
          .collect(toList());

assertEquals(
   asList("1abc"),
   beginningWithNumbers);
```


map
to

```
List<String> collected =
  Stream.of("a", "b", "hello")
        .map(value -> value.toUpperCase())
        .collect(toList());

assertEquals(
   asList("A", "B", "HELLO"),
   collected);
```

Exercise

1) Find the even numbers up to 10:
`com.java_8_training.problems.streams.FilterExerciseTest`

2) Double the stream of numbers given to you
`com.java_8_training.problems.streams.MapExerciseTest`

Checking a predicate matches all/no elements

```
boolean isHealthy =
    menu.stream()
        .allMatch(d -> d.getCalories() < 1000);

boolean isReallyBad =
    menu.stream()
        .noneMatch(d -> d.getCalories() < 1000);
```

Finding an element

```
Optional<Dish> dish =
    menu.stream()
        .filter(Dish::isVegetarian)
        .findAny();

Optional<Dish> dish =
    menu.stream()
        .filter(Dish::isVegetarian)
        .findFirst();
```

Quiz: Finding an element

- Find the first square that is divisible by 3 from a list of number

Quiz: Finding an element

- Find the first square that is divisible by 3 from a list of number

```
List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> firstSquareDivisibleByThree =
    someNumbers.stream()
               .map(x -> x * x)
               .filter(x -> x % 3 == 0)
               .findFirst(); // 9
```

The Reduce Pattern

The reduce pattern takes a stream of values and produces a single value at the end.

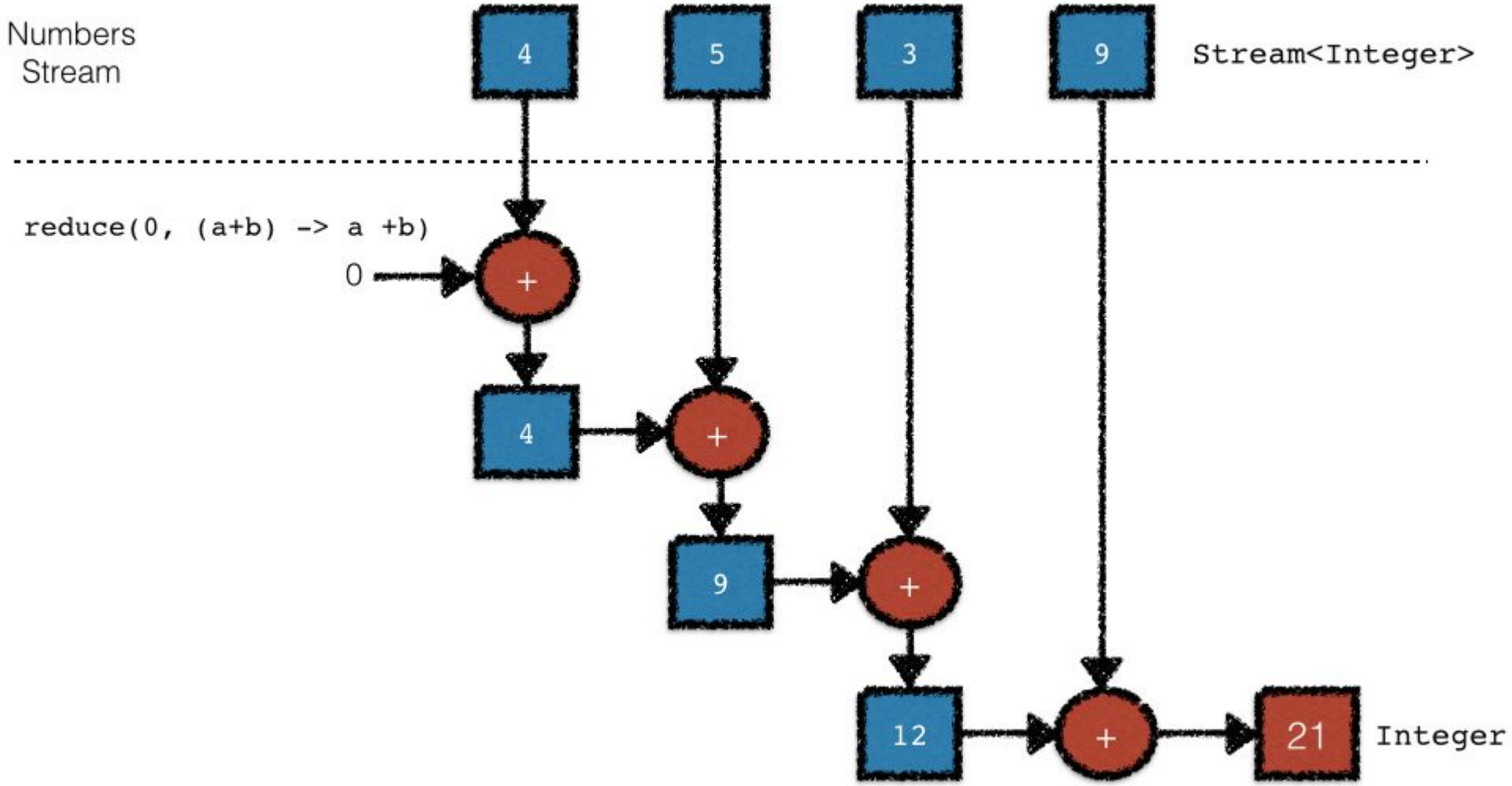
The pattern in code

```
Object accumulator = initialValue;

for(Object element : collection) {

    accumulator = combine(accumulator, element);

}

return accumulator
```

Summing: an example of reduce

```
int sum =
    Stream.of(4, 5, 3, 9)
          .reduce(0, (acc, x) -> acc + x);

assertEquals(21, sum);
```

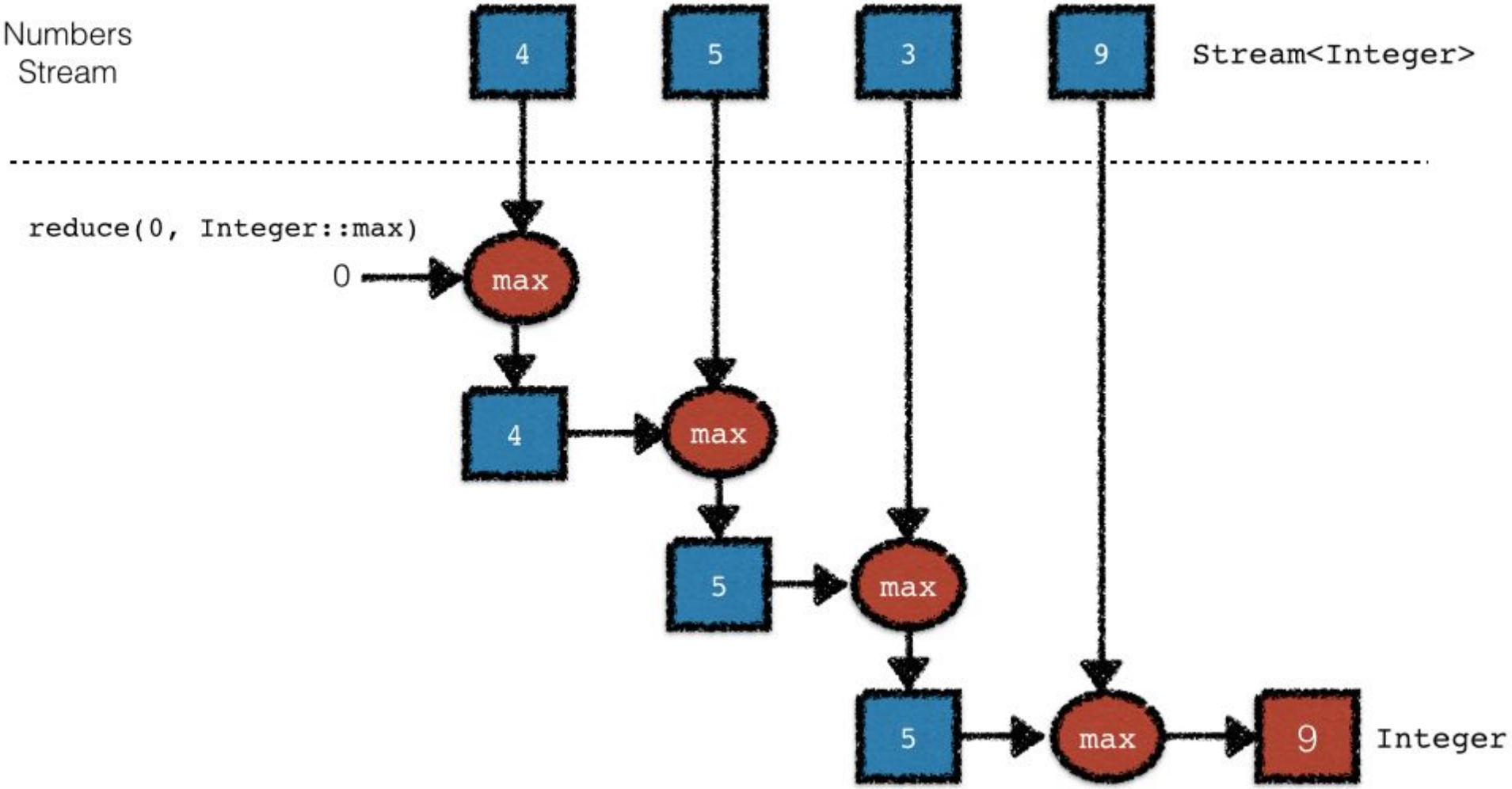
Summing with reduce



Reduce more examples

```
int product = numbers.stream()
                     .reduce(1, (a, b) -> a * b);

int max = numbers.stream()
                 .reduce(Integer.MIN_VALUE,
                         (a, b) -> Integer.max(a, b));

int max = numbers.stream()
                 .reduce(Integer.MIN_VALUE,
                          Integer::max);
```

Reduce: max


Numbers
Stream
4
5
3
9
Stream<Integer>
reduce(0, Integer::max)
0
max
4
max
5
max
5
max
9
Integer

One Argument Reduce

```
// Leaves out the initial value

Optional<Integer> sum =
    Stream.of(1, 2, 3, 4)
          .reduce((acc, x) -> acc + x);

assertEquals(10, sum.get());
```

Three Argument Reduce

```
int sum =
    Stream.of(1, 2, 3, 4)
          .reduce(0,
                  (acc, x) -> acc + x,
                  (l, r) -> l + r);

assertEquals(10, sum.get());
```

Summary of operations

Operation	Type	Argument Type	Argument function descriptor	Result
filter	intermediate	Predicate<T>	T -> boolean	Stream<T>
distinct	intermediate			Stream<T>
skip	intermediate	long		Stream<T>
limit	intermediate	long		Stream<T>
map	intermediate	Function<T, T>	T -> R	Stream<R>
flatMap	intermediate	Function<T, Stream<R>>	T -> Stream<R>	Stream<R>
sorted	intermediate	Comparator<T>	(T, T) -> int	Stream<T>
anyMatch	terminal	Predicate<T>	T -> boolean	boolean
noneMatch	terminal	Predicate<T>	T -> boolean	boolean
allMatch	terminal	Predicate<T>	T -> boolean	boolean

Summary of operations

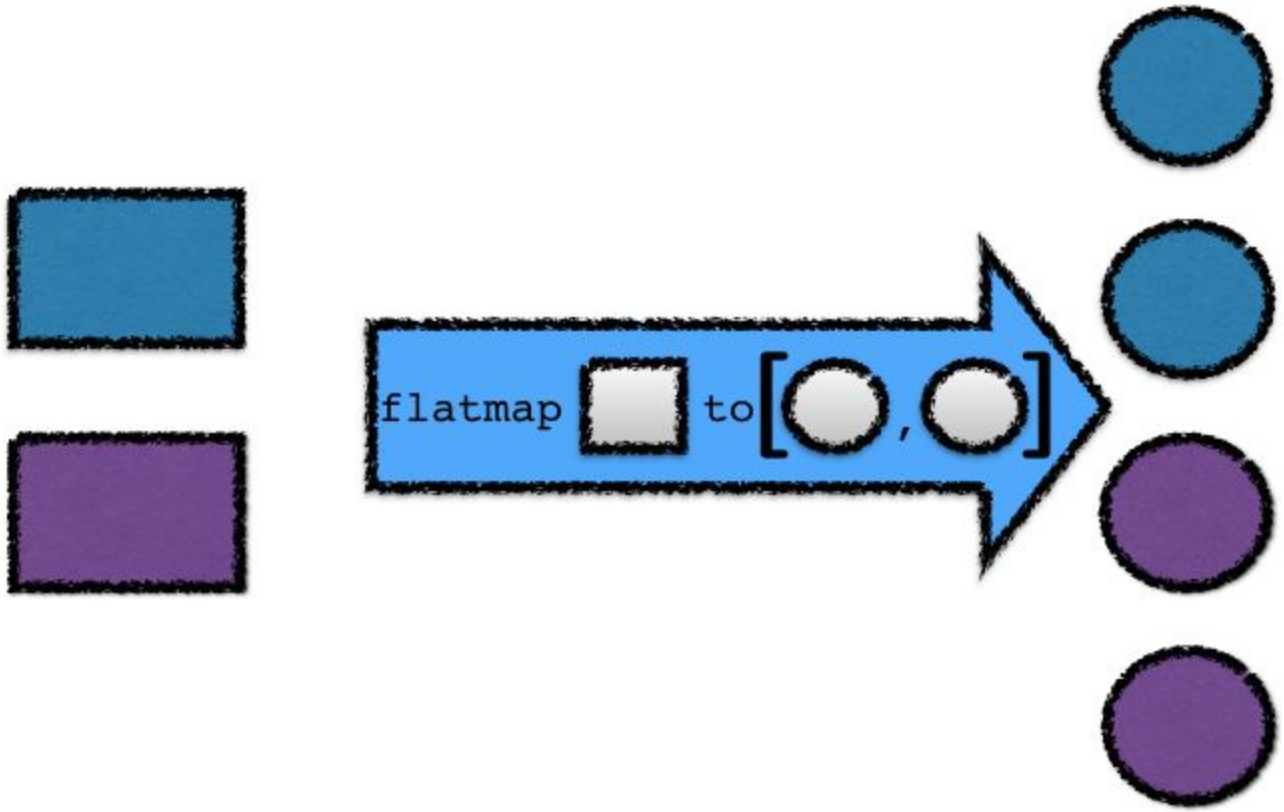
Operation	Type	Argument Type	Argument function descriptor	Result
findAny	terminal			Optional<T>
findFirst	terminal			Optional<T>
max/min	terminal	Comparator<T>	(T, T) -> int	Optional<T>
forEach	terminal	Consumer<T>	T -> void	void
collect	terminal	Collector<T, A, R>		R
reduce	terminal	BinaryOperator<T>	(T, T) -> T	Optional<T>
reduce	terminal	(T, BinaryOperator<T>)	(T , T) -> T	T
count	terminal			long

Exercise

Use reduce to find the the minimum number in the stream

`com.java_8_training.problems.streams.ReduceExerciseTest`

`flatMap` lets you replace a value with a Stream and concatenates all the streams together


flatmap
to
[
,
]

flatMap

```
// The Album class has a method:
// Stream<Track> trackStream()

List<Album> albums = loadAlbums();

albums.stream()
      .flatMap(album -> album.trackStream())
      .collect(toList())
```

Putting the operations together

Putting it Together

for a given an album, find the nationality of every band playing on that album

Putting it Together

1. get all the artists for an album,
2. figure out which artists are bands,
3. find the nationalities of each band
4. put together a list of these values.

Putting it Together

```
List<String> origins =
  album.getMusicians()
       .filter(artist -> artist.getName().startsWith("The"))
       .map(Artist::getNationality)
       .distinct()
       .collect(toList());
```

Example: Java 7

```
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish d: dishes){
    if(d.getCalories() < 400){
        lowCaloricDishes.add(d);
    }
}
List<String> lowCaloricDishesName = new ArrayList<>();
Collections.sort(lowCaloricDishes,
   new Comparator<Dish>() {
    public int compare(Dish d1, Dish d2){
        return Integer.compare(d1.getCalories(),
           d2.getCalories());
    }
});
for(Dish d: lowCaloricDishes){
    lowCaloricDishesName.add(d.getName());
}
```

filtering low calories

sorting by calories

Extract names

Example: Java 8

```
List<String> lowCaloricDishesName =
                dishes.stream()
                      .filter(dish -> dish.getCalories() < 400)
                      .sorted(comparing(Dish::getCalories))
                      .map(Dish::getName)
                      .collect(toList());
```

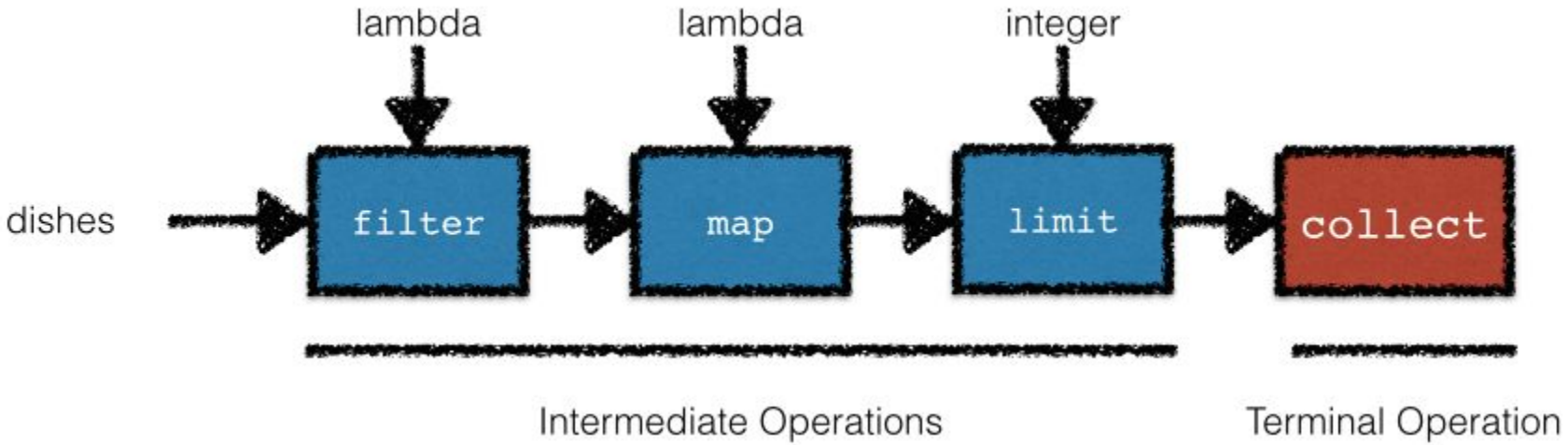
Example: Java 8 - parallel

```
List<String> lowCaloricDishesName =
                dishes.parallelStream()
                      .filter(dish -> dish.getCalories() < 400)
                      .sorted(comparing(Dish::getCalories))
                      .map(Dish::getName)
                      .collect(toList());
```

Eager and Lazy evaluation

Two types of operations

- intermediate: return a Stream and can be “connected”
- terminal: return a non-Stream value (e.g. int, String,...)



Short-circuiting

- no need to process the whole stream to find a result
- stop as soon as a result can be produced
- similar to boolean arithmetic
  - many expressions chained with the and operator
  - we know the result is false as soon as one expression is false

Eager vs Lazy

- eager evaluation: evaluate first operation and start the next one **only** when completed
- lazy evaluation: evaluate **only** when we actually **need** to iterate the result of an operation
- *intermediate operations* are lazy because it enables optimisations
  - a pipeline of three operations could be merged to one

Quiz: Eager vs Lazy

```
Set<String> origins =
  album.getMusicians()
       .filter(artist -> artist.getName().startsWith("The"))
       .map(artist -> {
         String nation = artist.getNationality());
         System.out.println(nation);
         return nation;
       })

// What's printed at this point?
       .collect(toSet());
```

Laziness & short-circuiting

```
List<Integer> numbers =
    Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);

List<Integer> twoEvenSquares =
    numbers.stream()
           .filter(n -> n % 2 == 0)
           .map(n -> n * n)
           .limit(2)
           .collect(toList());
```

Laziness & short-circuiting

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
List<Integer> twoEvenSquares =
    numbers.stream()
           .filter(n -> n % 2 == 0)
           .map(n -> n * n)
           .limit(2)
           .collect(toList());
```

- filtering 1
- filtering 2
- mapping 2
- filtering 3
- filtering 4
- mapping 4

Quiz: Laziness

Take a look at the signatures of these Stream methods. Are they eager or lazy?

a. boolean anyMatch(Predicate<? super T> predicate);

b. Stream<T> limit(long maxSize);

Exercise

A series of examples related to traders and transactions.

Look at test:

`com.java_8_training.problems.streams.TransactionsAndTradesPart1Test`

Primitive Streams

Numeric streams

This isn't possible

```
int calories = menu.stream()
                   .map(Dish::getCalories)
                   .sum();
```

- Stream<T>: **T** would need to be summable!
- Streams API bring primitive specialisations to fix this problem

Primitive streams (1)

- IntStream, LongStream, DoubleStream specialise the elements for int, long and double (more efficient as no boxing!)

- Can convert a normal stream to a primitive streams using mapToInt, mapToLong or mapToDouble

```
int calories = menu.stream()
                   .mapToInt(Dish::getCalories)
                   .sum();
```

Primitive streams to normal Stream

- use boxed()

```
IntStream intStream =
    menu.stream()
        .mapToInt(Dish::getCalories);

Stream<Integer> stream = intStream.boxed();
```

Primitive streams to normal Stream

- Or better mapToObj()

```
IntStream intStream =
    menu.stream()
        .mapToInt(Dish::getCalories);


intStream.mapToObj(cals ->
 “Wow you’ve eaten “ + cals + “ calories”)
         .forEach(System.out::println);
```

Quiz: Numeric streams

- Given a list of words as List<String>, how would you calculate the sum of the length of each word?

Quiz: Numeric streams

- Given a list of words as List<String>, how would you calculate the sum of the length of each word?

```
int sum = list.stream()
               .mapToInt(String::length)
               .sum();
```

Numeric ranges

- Two static methods on [Int|Long]Stream
  - exclusive: **range**(start, end)
  - inclusive: **rangeClosed**(start, end)

```
IntStream evenNumbers =
    IntStream.rangeClosed(1, 100)
             .filter(n -> n % 2 == 0);
```

Numeric ranges + mapToObj

```
Stream<Pair<Integer, Integer>> pairs =
    IntStream.rangeClosed(1, 10)
             .mapToObj(n -> new Pair(n,n));
```

Building Streams

Building streams

- You've seen how to create a stream from a collection
- You've seen how to create numeric ranges
- You can also create streams from
  - Values
  - Arrays
  - File
  - Function: infinite streams!

Building streams - values

- Stream.of

```
Stream<String> stream =
    Stream.of("Java", "8", "Training");

stream.map(String::toUpperCase)
      .forEach(System.out::println);
```

- Empty stream:

```
Stream<String> emptyStream = Stream.empty();
```

Building streams - arrays

```
int[] numbers = {2, 3, 5, 7, 11, 13};
int sum = Arrays.stream(numbers).sum();
```

Building streams - file

```
long uniqueWords =
    Files.lines(Paths.get("data.txt"), Charset.defaultCharset())
         .flatMap(line -> Arrays.stream(line.split(" ")))
         .distinct()
         .count();
```

Infinite streams

- A stream that doesn’t have a fixed size like when we create a stream from a fixed collection.

- Streams produced by **iterate** and **generate** create values on demand given a function and can therefore calculate values forever!

Iterate

- Takes a starting value and a lambda of type `UnaryOperator<T>`
- Applies the lambda successively to the result

```
Stream.iterate(0, n -> n + 2)
      .limit(10)
      .forEach(System.out::println);
```

Iterate

```
Stream.iterate(0, n -> n + 2)
      .limit(10)
      .forEach(System.out::println);
```

0, 2, 4, 6, 8 …

- The limit allows us to turn an infinite stream into a finite sized stream

Generate

- Lets you produce an infinite stream of values computed on demand
- Takes a lambda of type () -> T (a Supplier<T>)
- However, does not apply the lambda passed successively to the result. It just calls it every time and produce a new value.

```
Stream.generate(Math::random)
        .limit(5)
        .forEach(System.out::println);
```

Generate

```
Stream.generate(Math::random)
        .limit(5)
        .forEach(System.out::println);
```

```
0.9410810294106129
0.6586270755634592
0.9592859117266873
0.13743396659487006
0.3942776037651241
```

Quiz: infinite streams

- How would you produce a stream of all numbers that are both divisible by 5 and 3?

Quiz: infinite streams

```
Stream<Integer> s =
    Stream.iterate(0, n -> n + 1)
          .filter(n -> n % 3 == 0 && n % 5 == 0);
```

```
Stream<Integer> s =
    Stream.iterate(0, n -> n + 15);
```

Summary

Streams

- Powerful new abstraction
- Enhanced Iterator with inversion of control
- Supports lots of operations