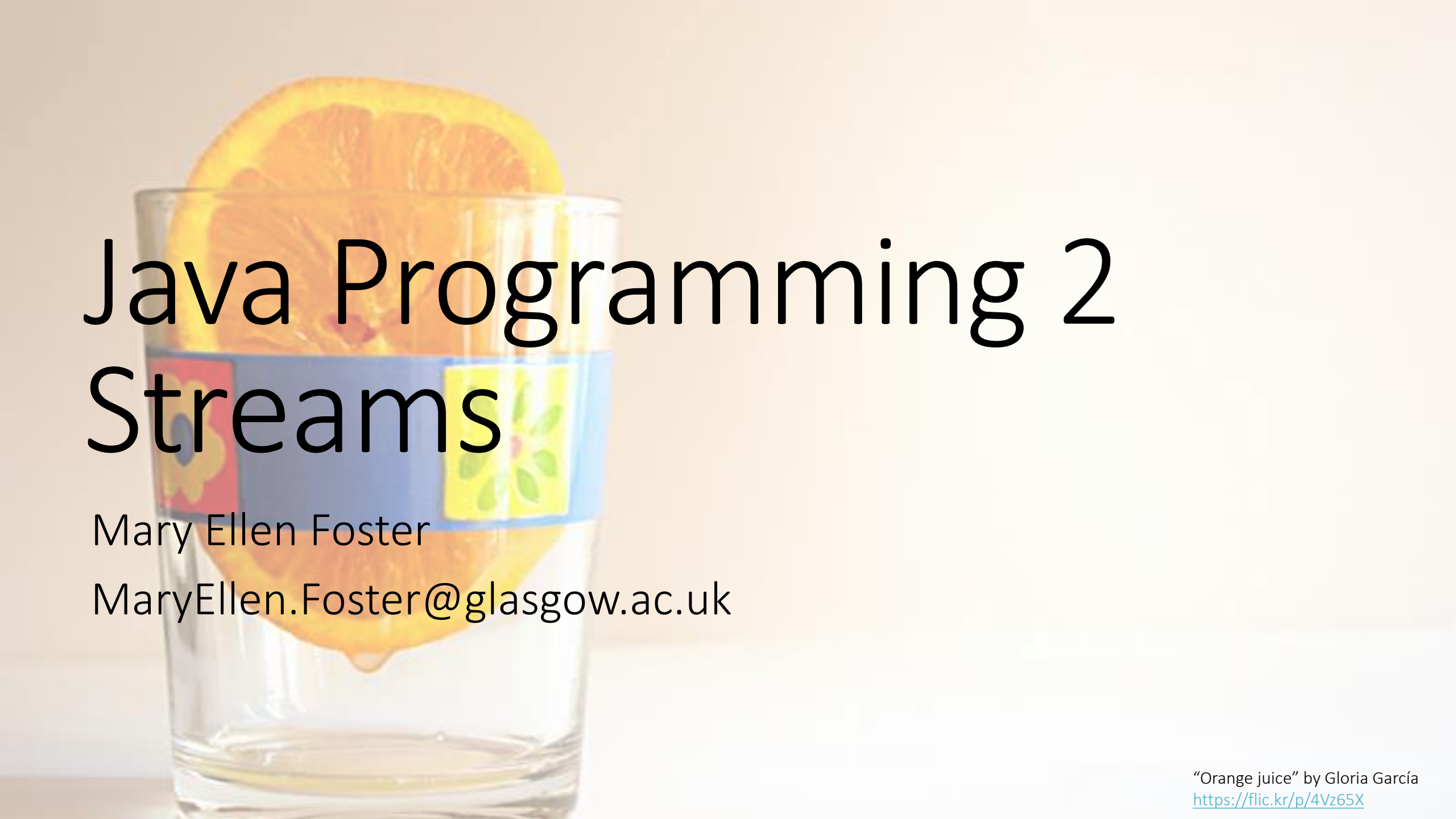
Java Programming 2
Streams

Mary Ellen Foster

MaryEllen.Foster@glasgow.ac.uk



"Orange juice" by Gloria García
https://flic.kr/p/4Vz65X

Streams overview

Package: java.util.stream (added in Java 8)

Primary class: **Stream<T>** (generic)

- Represents a sequence of elements
- Allows operations to be performed on the sequence without needing to write, e.g., loops or other control structures
- All **intermediate** operations return a new Stream to allow operations to be **chained**
- All **terminal** operations traverse the stream to produce a **result** or a **side effect**

You can get a Stream from a collection via the **stream()** method – also many other ways to create one depending on the source (e.g., Stream.of, Arrays.stream, …)

Collection vs stream

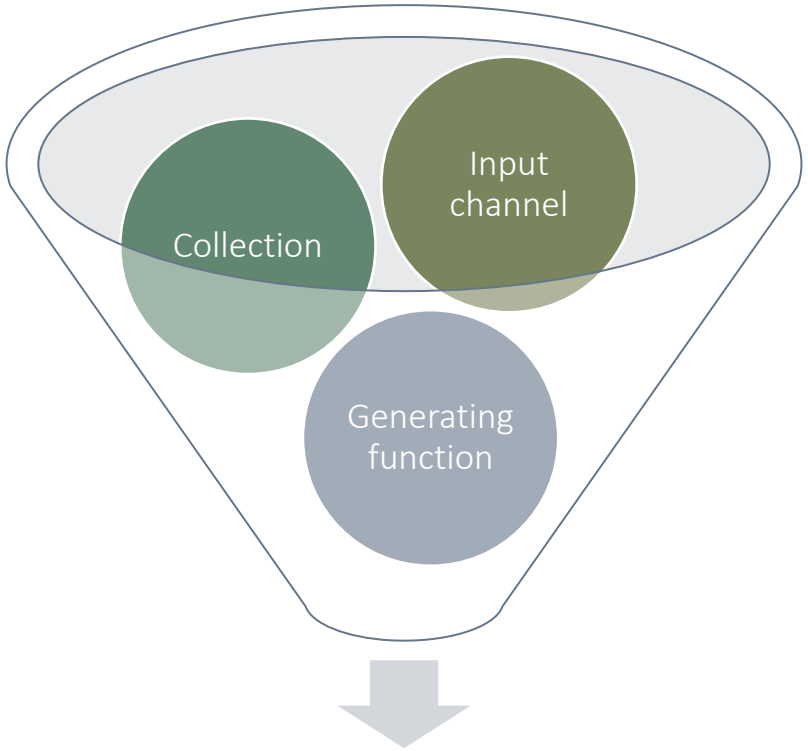
COLLECTION

- Stores data internally
- Operations modify data directly
- Must be finite in size

STREAM

- No storage – carry values from a source through a pipeline
- Operations produce a new (modified) stream
- Permit **laziness**
  - Elements can be returned on demand
  - Can represent infinite streams (e.g., all integers)

Possibly instructive image



Sum of squares with standard Java

```
List<Integer> list, squareList;
int sum = 0;
for (int i : list) {
  squareList.put(i*i);
}
for (int square : squareList) {
  sum += square;
}
```

Sum of squares with streams

```
int sum = list.stream()
                // New stream of squares
                .map(x -> x*x)
                // Convert Integers to primitive ints
                .mapToInt(x -> x)
                // Sum the stream
                .sum();
```

Details of lambda expressions (->)

Formally: let you “express instances of single-method classes more succinctly”

What it looks like:

- Comma-separated list of formal parameters (can omit data type; can omit parens if only one parameter)
- An arrow token ->
- A body consisting of a **single expression** or a **statement block**

The argument to stream operations is a **lambda expression**

You can also use a **method reference** instead of a lambda expression

- Syntax: `ClassName::methodName`

Some useful intermediate operations

map(), mapToInt(), mapToDouble(), etc

Produces a new stream where the given operation has been applied to each element

sorted()

Produces a new stream where the elements are sorted (assumes stream element implements Comparable)

filter()

Produces a new stream containing only the elements that match

limit()

Produces a new stream of at most the given number of elements

distinct()

Produces a new stream of only the unique objects according to equals()

Some useful terminal operations

max(), min(), average(), sum()
Only on numeric streams

count()

collect(), toArray(), reduce()
Various ways of combining together all the elements of the stream

findFirst(), findAny()
Tries to find an element matching the argument (returns an **Optional**)

forEach
Implements internal iteration

forEach and internal iteration

Instead of using a for loop and implementing the body …

External iteration

… you just specify what should happen to each element, and let the collection manage the details of processing the elements

Internal iteration

```
for (Shape s : shapes) {
    s.setColor(RED);
}
```

```
shapes.forEach
    (s -> s.setColor(RED));
```

Example drawn from http://www.drdobbs.com/jvm/lambdas-and-streams-in-java-8-libraries/240166818

Colouring only blue objects red

```
// Produce a stream view of the Collection
shapes.stream()
      // Create a new stream of only blue objects
      .filter(s -> s.getColor() == BLUE)
      // Colour all objects in the new stream red
      .forEach(s -> s.setColor(RED));
```

Some useful things to use in collect()

All are defined in the **Collectors** class

- **toList()** – convert into a list
- **toCollection()** – convert to a specific collection type
- **joining()** – concatenate all elements as strings
- **summingInt()** – sum all integer values without going through an integer stream
- **groupingBy()** – collect into a Map (very powerful, read documentation)
- **partitioningBy()** – collect into a Map, dividing into several lists (again, read docs)

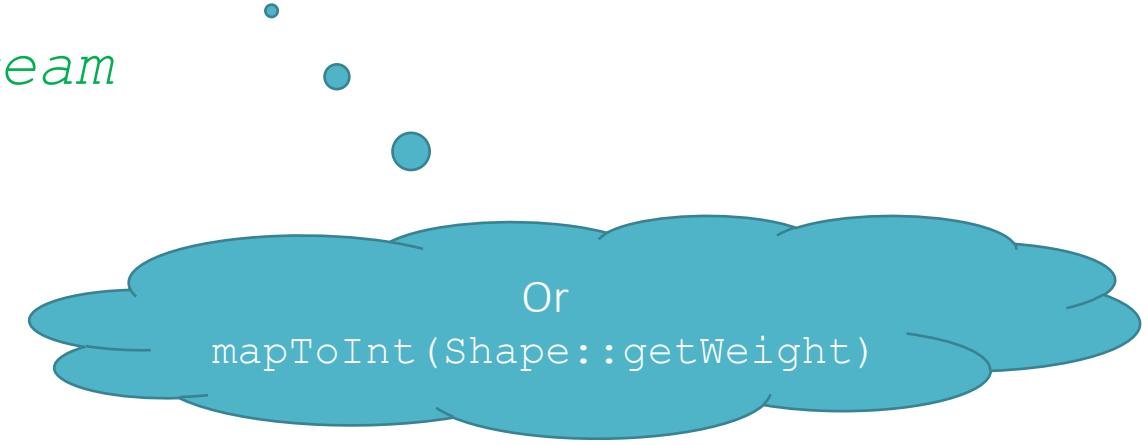
Documentation of all Collectors is at https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/stream/Collectors.html

Collecting blue shapes into new List

```
List<Shape> blue = shapes.stream()
                        // Select blue shapes
                        .filter(s -> s.getColor() == BLUE)
                        // Turn stream into a new list
                        .collect(Collectors.toList());
```

Compute total weight of blue shapes

```
int sum = shapes.stream()
                .filter(s -> s.getColor() == BLUE)
                // New stream of the object weights
                .mapToInt(s -> s.getWeight())
                // Sum the stream
                .sum();
```



Dealing with optional values

Some terminal operations return an **Optional<T>** (e.g., findAny, findFirst)

This represents a situation where the value might or might not exist, but returning null would be problematic

How to deal with an **Optional**

- First, check **isPresent()**
- If true, then you can access value with **get ()**
- Other option – use **orElse()** to represent default value

```
shapes.stream().filter(s -> s.getColor == YELLOW).findFirst();
```

Lambdas are useful beyond Streams

… not going to discuss them in this course though ☺

Some possible starting points to read more for those who are interested:

- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- https://www.baeldung.com/java-8-lambda-expressions-tips

If you're excited about this topic, there's an entire course on Functional Programming in Level 4