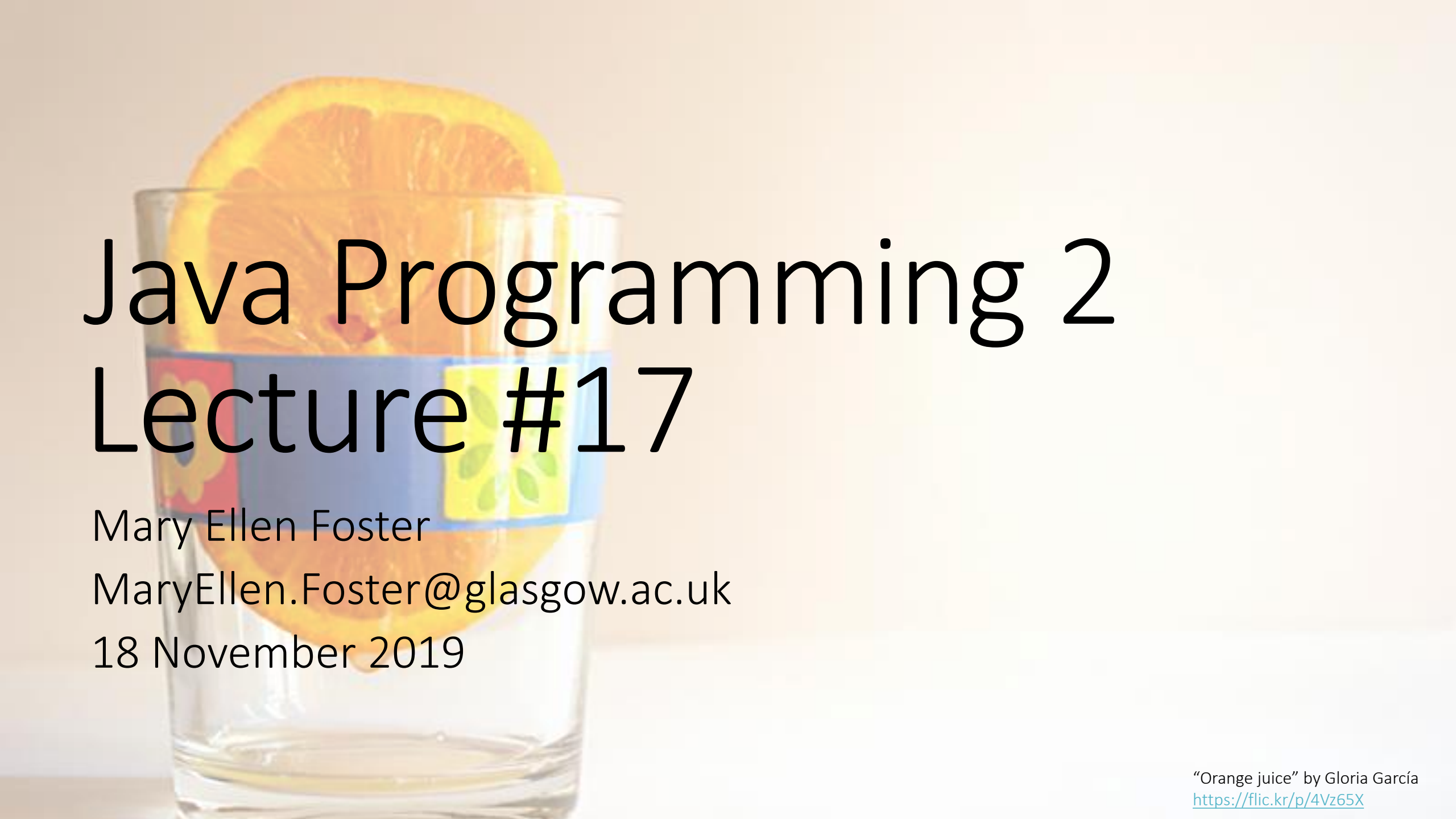


Java Programming 2
Lecture #17

Mary Ellen Foster
MaryEllen.Foster@glasgow.ac.uk
18 November 2019

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

Outline

Details of lab exam and rest of semester

Enumerations

Functional operations on `Lists`

Schedule for the rest of the semester

LECTURES/TUTORIALS

Week 9:

- 18 November: Lecture (Enum types, streams)
- 20 November: Lecture (JUnit, Javadoc)
- 22 November: No tutorial

Week 10:

- 25 November: Quiz (1% credit available)
- 27 November: Revision lecture
- 29 November: Tutorial: lab exam prep

Week 11:

- 2 December: No lecture (lab exam)
- 4 November, 6 November: going over past exam problems

LABS

Lab 8

- 15 November: Lab 8 distributed
- 18/19 November: work on Lab 8 in lab
- 21 November: Lab 8 due

Lab exam

- 20 November: Lab exam practice problem distributed
- 25/26 November: work on practice problem with tutor help
  - *Highly recommended especially if you have been using your own computer, ESPECIALLY if you have not been using Eclipse*
- 2/3 December: Lab exam

"Poison" by freaktography
https://flic.kr/p/FKnqci

Lab exam

Timetable

Wednesday 20th: Practice problem distributed

Next week: Work on practice problem in lab with tutor help

2/3 December: lab exam in your scheduled time slot and location

- Starts at 5 past the hour, finishes at 5 to – e.g., 9:05 – 10:55
- If you have special exam arrangements, the office will communicate with you soon

You will be given:

- Specification (electronic, and on paper)
- Moodle link to submit your solution

During the lab exam

You will have access to **your normal lab account**

You must use one of the lab machines

You will have full access to the course Moodle site and to the Internet

You can bring any paper notes that you want

No phones, smartwatches, etc

No use of communication programs or apps

Different than previous years!

Practice problem is similar in difficulty and scale to actual lab exam

Each time slot will have **its own programming task** (6 separate lab exams in total)

All problems are of comparable difficulty

"Open-book" (i.e., open-Internet) but don't rely on searching to solve all problems

- E.g., you should remember how to define a class or initialise an ArrayList or similar without looking it up

Marking

Worth 20% of your mark – will be marked out of 20

Outline marking scheme included in lab sheet; will include:

- Correctness of each class
- Correctness of overall system
- Appropriate class design, including
  - *Well chosen data types*
  - *Well motivated methods*
  - *Access modifiers*
  - *Use of constructors, etc*
- Appropriate style (commenting where required, variable names, etc)

Your grade in this class

20% -- weekly lab exercises

- Based on **best 5** lab marks
- Will be computed out of 25 -> scaled to a mark out of 20

20% -- lab exam

- Marked out of 20 as described above

60% -- written exam (1 hour)

- Factual, definition questions
- Understanding fragments of Java code
- Writing Java code on paper (we will be lenient on syntax)

Enumerations

Enumerations

An enum type is a special data type that allows a variable to be one of a set of predefined constants

Common examples:

- Compass directions (NORTH, SOUTH, EAST, WEST)
- Days of week, months of year, etc.

Declaring an enum in Java

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY,
  }
```

Note: values are constants ==> conventionally written in ALL_CAPS

You use the `enum` keyword **instead of** `class`

An enum called Day should be in a class Day.java

An enum is a special class

It has **methods**

- Built-in static method values() that returns an array of all values
- Built-in static method valueOf() that parses a string into an enum constant
- Appropriate definitions of compareTo(), equals(), hashCode(), toString()
- Other methods:
  - ordinal() -- returns the position of this constant in the list
  - name() -- returns the name of this constant
- Any other methods that you define

You can define **fields** as well if necessary

Declaring a more complex enum

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);
    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
}
```

https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

Using an enum

It can be used in `switch` statements

You can create them with valueOf

You can iterate through them using values()

You can access their names and ordinal positions

```
switch (day) {
  case MONDAY:
    System.out.println("Monday");
    break;
}
```

```
Day day = Day.valueOf("MONDAY");
```

```
for (Day day : Day.values()) {
    System.out.println(day.ordinal()
        + " " + day.name());
}
```

Functional operations on Lists

Motivating example: sum of squares

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

Two separate loops:

- One creates the list of squares
- Second loop sums them

Could create a single loop, but code would still be kind of messy

For loop processes all elements, one at a time, in order

Lambda expressions (since Java 8)

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));
```

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

Streams

As of Java 8, all `Collection` objects have a method `stream()`

- Returns an instance of `java.util.stream.Stream`

A Stream

- Represents a sequence of values
- Exposes a set of **aggregate operations** to express common manipulations easily and clearly
- 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**

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 aggregate operations is a **lambda expression**

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

- Syntax: `ClassName::methodName`

Colouring only blue objects red

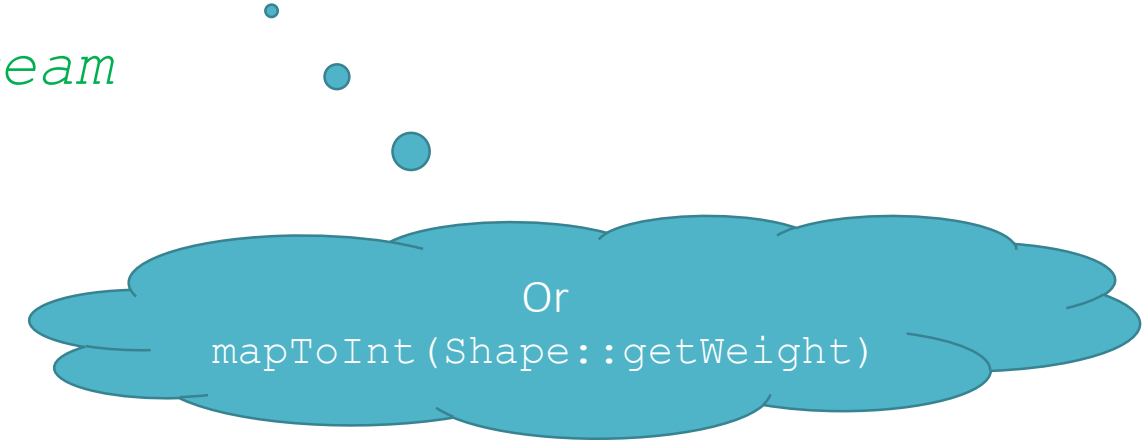
```
// Produce a stream view of the Collection
shapes.stream()
      // Create a new stream of only blue objects
      .filter(s -> s.getColor() == BLUE)
      // Colour all objects in the new stream red
      .forEach(s -> s.setColor(RED));
```

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();
```



Collection vs stream

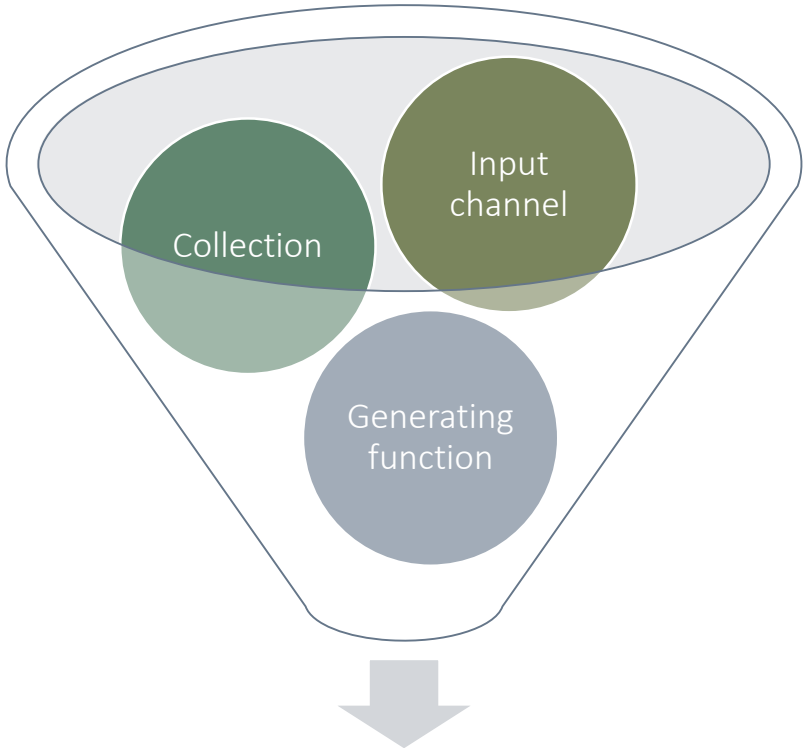
COLLECTION

- 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 stream and lambda

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

Some useful stream operations

```
.count()
.distinct()
.filter()
.findFirst()
.forEach()
.map()
.max()
.min()
```

```
.reduce()
.skip()
.sorted()
.toArray()

.average()
.sum()
.min() / .max()
```

Numeric streams only

https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#StreamOps for full documentation

Next time

Threads in Java