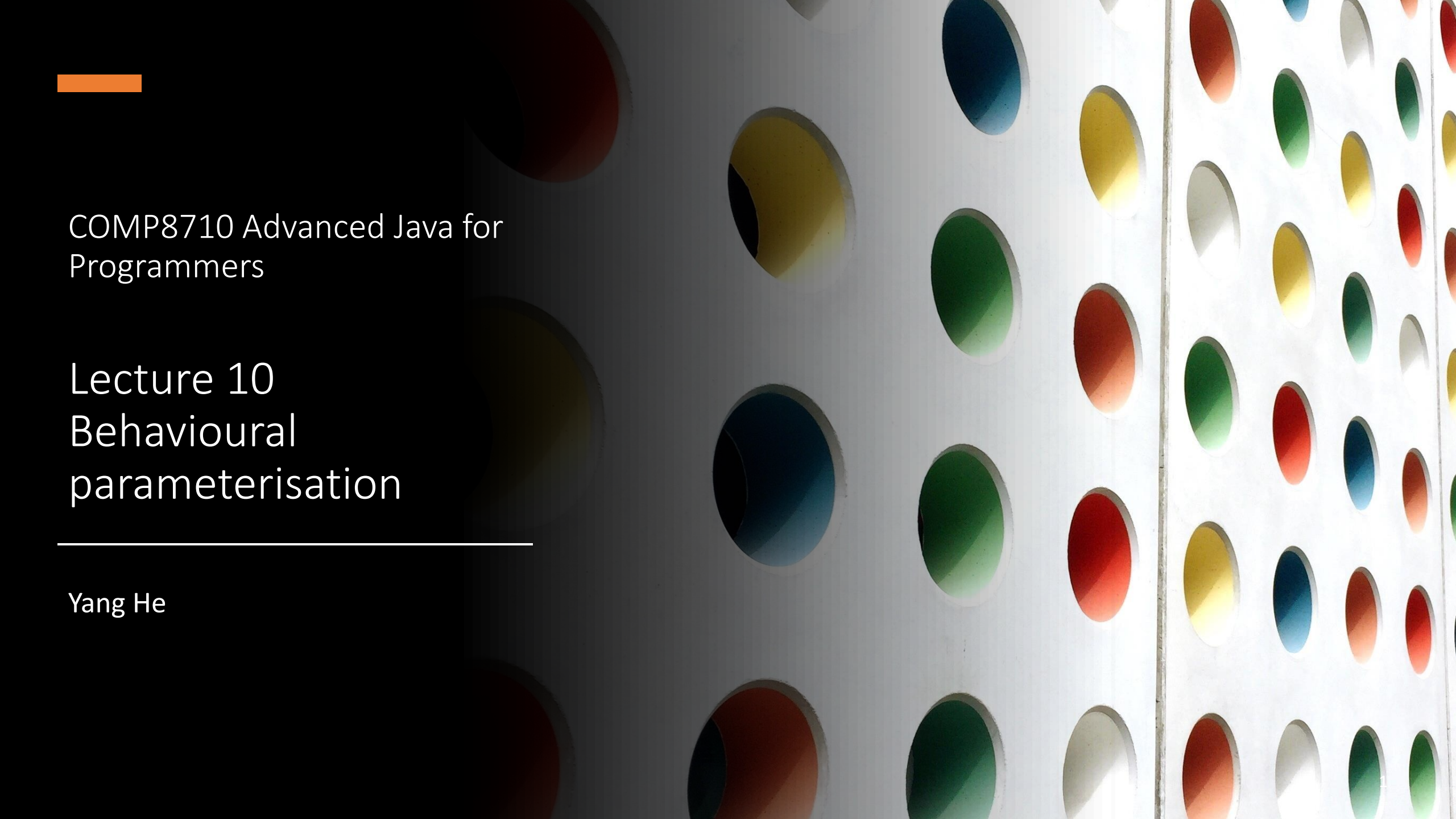
COMP8710 Advanced Java for Programmers

Lecture 10 Behavioural parameterisation

Yang He



Teaching plan (1)

- Better programming with Java ≥ 8
  - Introduction
  - Behaviour parameterisation
  - Lambda expressions
  - Functional interface
  - Method reference
  - Threads

Teaching plan (2)

- Better programming with Java ≥ 8
  - Introduction to streams
  - Stream operations
  - Collections vs. streams
  - Matching with Optional
  - Stateless vs. stateful
  - Infinite streams
  - Parallel streams

Teaching plan (3)

- GUI with JavaFX
  - Overview
  - Interacting with the user
  - Managing events
  - Properties & bindings
  - Link with application logic

Reference

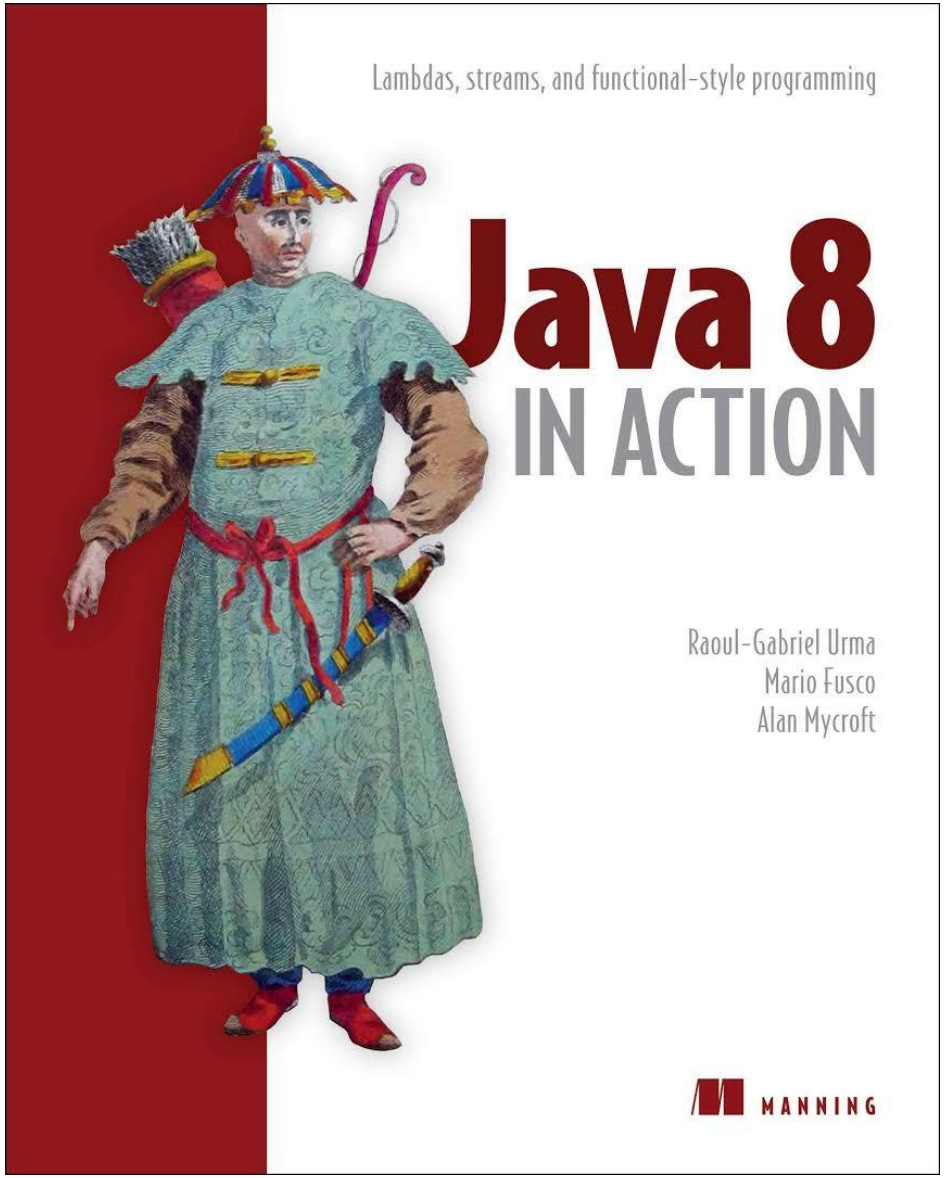
- The textbook used for this part of the course:

  “Java 8 In Action”

  (Chapters 2 to 7 in particular)

  by Urma, Fusco, and Mycroft

- It is available in the library
- There are more examples online



Demo:
Changing requirements

`FilteringAppleProject`

Demo – Example 1

- Implement a functionality to filter green apples from a list
- 1st attempt:

```
public List<Apple> filterGreenApple(List<Apple> inv) {
        var result = new ArrayList<Apple>();
        for (var a : inv) {
            if (a.color().equals("green")) {
                result.add(a);
            }
        }
        return result;
}
```

Demo – Example 2

- 2nd attempt: let user provides a colour using a parameter

```
public List<Apple> filterByColour(List<Apple> inv, String colour) {
        var result = new ArrayList<Apple>();
        for (var a : inv) {
            if (a.color().equals(colour)) {
                result.add(a);
            }
        }
        return result;
}
```

Demo – Example 3

- Implement a functionality to filter apples by the colour and weight from a list

```
public List<Apple> filterByColourAndWeight(List<Apple> inv,
                                           String colour, int weight) {
        var result = new ArrayList<Apple>();
        for (var a : inv) {
            if (a.color().equals(colour) && a.weight() > weight) {
                result.add(a);
            }
        }
        return result;
}
```

Behaviour parameterisation

- We need a better way to cope with changing requirements. We can’t keep adding parameters and nested if-then-else!
- Behaviour parameterisation is a software development pattern that lets us handle frequent requirement changes

Behaviour parameterisation: predicates

- How can we model an abstract selection criteria on apples?
- A solution:

```
interface ApplePredicate {
    public boolean test(Apple a);
}
```

Instances of ApplePredicate (1)

- E.g. Predicate to identify *green* apples:

```
public class GreenApplePredicate implements ApplePredicate {
    @Override
    public boolean test(Apple a) {
        return a.color().equals("green");
    }
}
```

Instances of ApplePredicate (2)

- E.g. Predicate to identify *heavy* apples:

```
public class HeavyApplePredicate implements ApplePredicate {
    @Override
    public boolean test(Apple a) {
        return a.weight() > 100;
    }
}
```

Instances of ApplePredicate (3)

- Now, if you need to filter all the apples that are *green* and *heavy*, you need to define a new predicate:

```
public class GreenAndHeavyApplePredicate implements ApplePredicate {
    @Override
    public boolean test(Apple a) {
        return a.color().equals("green") && a.weight() > 100;
    }
}
```

Filtering by abstract criteria

- Now that we have an abstract way to characterise a selection criteria, i.e. via a predicate, we can pass it as a parameter to a method:

```
public List<Apple> filterApple(List<Apple> inv, ApplePredicate p) {
    var result = new ArrayList<Apple>();
    for (var a : inv) {
        if (p.test(a)) {
            result.add(a);
        }
    }
    return result;
}
```

Using the new filter

- E.g. green apples

```
var greenApples = filterApple(inventory, new GreenApplePredicate());
```

- E.g. heavy apples

```
var heavyApples = filterApple(inventory, new HeavyApplePredicate());
```

- E.g. green and heavy apples:

```
var greenHeavyApples = filterApple(inventory, new GreenAndHeavyAppePredicate());
```

On verbosity

- Currently the code is very verbose
  - We need to create a new `ApplePredicate` object for each test method!
- It is time-consuming process and unnecessary overhead
- Can we do better?

Anonymous classes (1)

- Anonymous classes are like local classes (i.e. a class defined in a block), but they don't have a name
- It allows us to declare and instantiate a class at the same time
- First attempt at a solution, e.g. filter red apples

```
var redApples = filterApple(inventory, new ApplePredicate() {
        public boolean test(Apple a) {
                return a.color().equals("red");
        }
});
```

Anonymous classes (2)

- It can be assigned to a variable for reuse:

```
var redApplePred = new ApplePredicate() {
        public boolean test(Apple a) {
                return a.color().equals("red");
        }
};

var redApples = filterApple(inventory, redApplePred);
```

Abstract more with generics (1)

- Using Java generics, we can abstract away even more
- `ApplePredicate` does not need to be Apple specific
- We can define a generic type:

```
interface Predicate<T> {
    public boolean test(T t);
}
```

From Java 8: the interface Predicate is defined in java.util.function

Java 21 API: https://docs.oracle.com/en/java/javase/21/docs/api/help-doc.html

Abstract more with generics (2)

- Similarly, we can define a generic filter method:

```
public <T> List<T> filter(List<T> inv, Predicate<T> p) {
    var result = new ArrayList<T>();
    for (var a : inv) {
        if (p.test(a)) {
           result.add(a);
        }
    }
    return result;
}
```

Abstract more with generics (3)

- E.g. green and heavy apples:

```
var greenHeavyApples = filter(inventory, new Predicate<>() {
            @Override
            public boolean test(Apple a) {
                return a.color().equals("green") && a.weight() > 100;
            }
});
```

Exercise

- What is the output of the main method of class MeaningOfThis?

```
public class MeaningOfThis {
    public final int value = 4;
    public void doIt() {
        var value = 6;
        Runnable r = new Runnable() {
            public final int value = 5;
            public void run() {
                var value = 10;
                System.out.println (this.value);
            }
        };
        r.run();
    }
    public static void main(String[] args) {
        var m = new MeaningOfThis();
        m.doIt(); // ???
    }
}
```