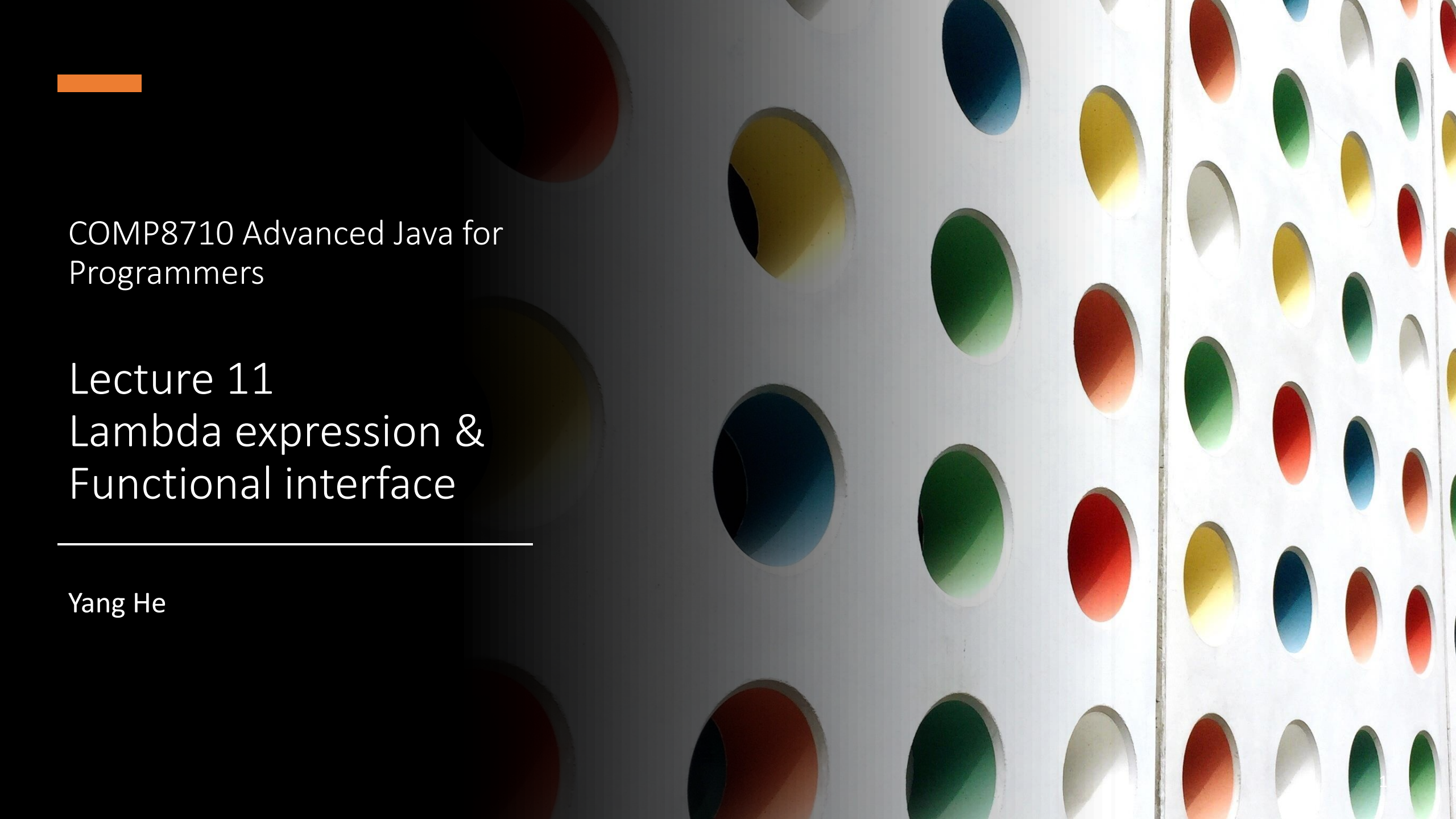
COMP8710 Advanced Java for Programmers

Lecture 11
Lambda expression & Functional interface

Yang He



Topics

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

Predicate by colour

```
public class AppleColourPredicate implements Predicate<Apple> {
    private final String colour;
    public AppleColourPredicate(String colour) {
        this.colour = colour;
    }

    @Override
    public boolean test(Apple a) {
        return a.color().equals(colour);
    }
}
```

- E.g. `new AppleColourPredicate("green")`

Predicate by weight

```
public class AppleWeightPredicate implements Predicate<Apple> {
    private final int threshold;
    public AppleWeightPredicate(int threshold) {
        this.threshold = threshold;
    }

    @Override
    public boolean test(Apple a) {
        return a.weight() > threshold;
    }
}
```

- E.g. `new AppleWeightPredicate(100)`

Logical ‘and’ predicate (1)

```
public class LogicalAndPredicate implements Predicate<Apple> {
    private final Predicate<Apple> predA;
    private final Predicate<Apple> predB;
    public LogicalAndPredicate(Predicate<Apple> predA, Predicate<Apple> predB) {
        this.predA = predA;
        this.predB = predB;
    }

    @Override
    public boolean test(Apple a) {
        return predA.test(a) && predB.test(a);
    }
}
```

Logical ‘and’ predicate (2)

- E.g. red and heavy apples

```
Predicate<Apple> greenPred = new AppleColourPredicate("green");
Predicate<Apple> heavyPred = new AppleWeightPredicate(100);

var greenAndHeavyApples = filter(inventory,
                                 new LogicalAndPredicate(greenPred, heavyPred));
```

From Java 8, we can simply use the ‘and’ method of Predicate, e.g. greenPred.and(heavyPred)

Boilerplate code (1)

- Anonymous classes are still not good enough: there are too much boilerplate code!

```
var redApples = filter(inventory, new Predicate<Apple>() {
        public boolean test(Apple a) {
            return a.color().equals("red");
        }
});
```

Boilerplate code (2)

- Anonymous classes are still not good enough: there are too much boilerplate code!

```
var redApples = filter(inventory, new Predicate<Apple>() {
        public boolean test(Apple a) {
            return a.color().equals("red");
        }
});
```

This is all we want as a second argument of filter method!

Lambda expression (1)

- Using lambda expressions (Java 8 or later), we can write less boilerplate code
- e.g.

```
var redApples = filter(inventory,
                       (Apple a) -> { return a.color().equals("red"); } );
```

Or simply

```
var redApples = filter(inventory, a -> a.color().equals("red"));
```

- *Parameter type can be omitted as it can be inferred from the context*
- *With only one parameter, brackets () can be omitted.*
- *With only one statement, both curly brackets {} and the keyword return can be omitted*

Lambda expression (2)

- A lambda expression can be understood as a concise representation of an anonymous function
- It can be passed around as an argument to a method or stored in a variable
- It doesn’t have a name

Lambda expression (3)

- A lambda expressions consists of
  - A list of parameters
  - An arrow (->)
  - The body of the lambda (an expression or a list of statements)
- The basic syntax of a lambda:

```
(parameters) -> expression
```

or

```
(parameters) -> { statements; }
```

More examples of lambda expressions

- `(String s) -> s.length()`
- `(Apple a) -> { System.out.println(a.weight()); }`
- `(int x, int y) -> x * y`
- `() -> 42`
- `() -> new Apple(100, "green")`
- `(Apple a1, Apple a2) -> a1.weight().compareTo(a2.weight())`

- *A lambda expression can have zero, one or more parameters*
- *The type of the parameters can be explicitly declared or inferred from the context*

Where can we use lambdas?

- The Lambda expression can be used to provide an implementation of an interface which has only *one abstract method*
- E.g.

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

```
List<Apple> redApples = filter(inventory, a -> a.color().equals("red"));
```

Functional interface

- A functional interface is defined as an interface that specifies *exactly one abstract method* (annotation `@FunctionalInterface`)
- Examples of Java functional interfaces:

```
public interface Comparator<T> {
        int compare(T o1 , T o2);
}

public interface Runnable {
        void run();
}

public interface ActionListener extends EventListener {
        void actionPerformed (ActionEvent e);
}
```

Note: `EventListener` is an empty interface, i.e. it has no method, so it is not a functional interface!

Function descriptors (1)

- The signature of the abstract method of the functional interface essentially describes the signature of the lambda expression
- We call the signature of this abstract method a function descriptor
- E.g.

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

Function descriptors (2)

- Examples of functional interfaces:

Functional Interface	Method	Function Descriptors
`Predicate<T>`	`test`	`T -> boolean`
`Function<T,R>`	`apply`	`T -> R`
`BiFunction<T,U,R>`	`apply`	`(T, U) -> R`
`Consumer<T>`	`accept`	`T -> void`
`Supplier<T>`	`get`	`() -> T`
`UnaryOperator<T>`	`apply`	`T -> T`
`BinaryOperator<T>`	`apply`	`(T, T) -> T`
`Comparator<T>`	`compare`	`(T, T) -> int`

Sorting apples using lambda expression (1)

- The Java provides sort method on a list; it takes a Comparator:

```
// Interface Comparator is a functional interface
// Function descriptor: (T, T) -> int
void sort( Comparator<T> c )
```

- E.g. If we want to sort a List redApples by weight:

```
redApples.sort( (a1, a2) -> a1.weight().compareTo(a2.weight()) );
```

Sorting apples using lambda expression (2)

- Alternatively, use the static method comparing of Comparator:

```
redApples.sort( Comparator.comparing(a -> a.weight());
```

Record patterns (JDK 21)

- In pattern matching, we can destructure record instances, making it more concise and less error-prone
- E.g.

```
record Point(int x, int y){};
record Circle(Point centre, int radius){};


var obj = new Circle(new Point(2,3), 5);
if (obj instanceof Circle(Point(int x, int y), int radius)) {
    System.out.println("Centre: " + x + ", " + y);    // Centre: 2,3
    System.out.println("Radius: " + radius);          // Radius: 5
}
```

Pattern Matching for `switch` (JDK 21)

- It simplifies switch statements to be more concise and readable
- E.g.

```
var result = ""
switch (obj) {
    case null      -> result = "null";
    case String s  -> result = "String: " + s;
    case Integer i -> result = "Integer: " + i;
    case List t    -> result = "List of size " + t.size();
    case Point(int x, int y) -> result = "Point: " + x + ", " + y;
    default -> result = "Something else";
}
System.out.println(result)
```