COMP8710 Advanced Java for Programmers

Lecture 12
Method reference & Threads

Yang He

Topics

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

Method references (1)

- Method reference is a convenient shorthand of a lambda calling a method
- It can be used where the compiler can work out the full method call
- E.g. sorting red apples by weight

```
redApple.sort( Comparator.comparing(Apple::weight) );
```

is the same as

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

Method references (2)

- More examples:

```
ArrayList::new          <==>    () -> new ArrayList<>()

System.out::println     <==>    s -> System.out.println(s)

Math::abs               <==>    x -> Math.abs(x)

Apple::colour           <==>    (Apple a) -> a.colour()

String::substring       <==>    (str, i) -> str.substring(i)
```

Types of method references

- Reference to a static method, e.g.

```
Math::max      <==>   (a, b) -> Math.max(a, b)
```

- Reference to an instance method on a particular object, e.g.

```
Book::title    <==>   (Book book) -> book.title()
```

- Reference to an instance method on an object of a class type, e.g.

```
String::length  <==>   (text) -> text.length()
```

- Reference to a class constructor, e.g.

```
Random::new      <==>   () -> new Random()
```

Quiz

- What are equivalent method references for the following lambda expressions?

```
1)  Function<String, Integer> strToInteger = (String s) -> Integer.parseInt(s);

2)  BiPredicate<List<String>, String> contains = (list, str) -> list.contains(str);
```

Answer:

```
1)   Function<String, Integer> strToInteger = Integer::parseInt;
2)   BiPredicate<List<String>, String> contains = List::contains;
```

Homework – Sorting numbers using lambda

```
var myList = Arrays.asList(3,2,5,1,6,4);
```

1) Sort the list in ascending order => 1, 2, 3, 4, 5, 6
2) Sort the list in descending order => 6, 5, 4, 3, 2, 1
3) Sort the even numbers and odd numbers => 2, 4, 6, 1, 3, 5

Quiz

- Which of these interfaces are functional interfaces?

```
public interface Adder {
        int add(int a, int b);
}

public interface SmartAdder extends Adder {
        int add(double a, double b);
}

public interface Nothing { }
```

Answer:

Only Adder is a functional interface.

SmartAdder has two abstract methods called add (one is inherited from Adder).

Nothing has no abstract method at all.

Quiz

- Does the code below compile?

```
Object a = () -> System.out.println("Tricky example");
```

Answer:

No, because the context of the lambda expression is Object (the target type), but Object is a class, not a functional interface!

To fix this we can change the target type to Runnable, which represents a function descriptor

```
() -> void
```

i.e.

```
Runnable a = () -> System.out.println("Tricky example");
```

Examples – Composing lambda expressions (1)

- Reversing a comparator:

```
inventory.sort(comparing(Apple::weight).reversed());
```

Sort by weight in a descending order.

- Chaining Comparators:

```
inventory.sort(comparing(Apple::color).thenComparing(Apple::weight));
```

Sort by colour and then by weight within the same colour.

Examples – Composing lambda expressions (2)

- Composing predicates:

```
Predicate<Apple> redApplePred = a -> a.color().equals("red");
Predicate<Apple> heavyApplePred = a -> a.weight() > 100;
Predicate<Apple> notRedApplePred = redApplePred.negate();
Predicate<Apple> HeavyRedApplePred = redApplePred.and(heavyApple);
```

The interface Predicate provides these default methods

What are threads?

- Threads are like lightweight processes: they execute a block of code on their own
- Multithreading is a Java feature that allows *concurrent* execution of two or more parts of a program for maximum utilization of CPU
  - It enables a program to run more efficiently
  - It can be used to execute complicated tasks in the background without interrupting the main program
- In Java, you can use the `Runnable` interface to represent a block of code to be executed

Functional interface: `Runnable`

- "The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run."

https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

```
// java.lang.Runnable
// Function descriptor: () -> void
public interface Runnable {
        public void run();
}
```

Example - Multiple threads

```
var a = 6;
var b = 2;

// create new Threads
var ts = new Thread[4];
ts[0] = new Thread(() -> System.out.println("Thread 1: a+b = " + (a + b)));
ts[1] = new Thread(() -> System.out.println("Thread 2: a-b = " + (a - b)));
ts[2] = new Thread(() -> System.out.println("Thread 3: a*b = " + (a * b)));
ts[3] = new Thread(() -> System.out.println("Thread 4: a/b = " + (a / b)));

// start threads
for (var t: ts) {
    t.start();
}
```

The start() method is used to start the execution of a thread.
It internally calls the run() method of Runnable.

Using local variables (1)

- Lambda expressions can use/capture variables that are outside of their scope
  - Lambdas are allowed to capture *instance* and *static* variables without restriction
  - However, lambdas can only capture *local* variables that are declared *final* or are *effectively final* (assigned to only once)

Using local variables (2)

- This is will compile/run fine:

```
var counter = 0;
Runnable r = () -> System.out.println(counter);
new Thread(r).start();
```

- But this won't compile:

```
var counter = 0;
counter++;    // Not allowed to change the local variable
Runnable r = () -> System.out.println(counter);
new Thread(r).start();
```

Using local variables (3)

- We can use an object instead, e.g.

```
var obj = new Counter(0);  // local variable, effectively final
Runnable r1 = () -> System.out.println(obj.incrementAndGet());
new Thread(r1).start();
```

- `Counter` class:

```
public class Counter {
    private int value;
    public Counter(int n) { value = n; }
    public int incrementAndGet() { return ++value; }
    public int get() { return value; }
}
```

Using local variables (4)

- It’s better to use the built-in AtomicInteger class instead
- E.g.

```
var atomicInteger = new AtomicInteger();
Runnable r = () -> System.out.println(atomicInteger.incrementAndGet());
new Thread(r).start();
```

Concurrency issues

- Race condition occurs when 2 or more threads are accessing and changing the same variables at the same time
- It can cause run runtime errors or unexpected outcomes
- E.g.

```
var sharedCounter = new Counter(0);
var list = new ArrayList<Thread>();
for (var i = 0; i < 5; i++) {
    list.add(new Thread(() -> {
        var value = sharedCounter.incrementAndGet();
        System.out.println(currentThread().getName() + ": " + value
    }));
}
list.forEach(Thread::start);
```

One solution is to use a thread-safe object, e.g. a built-in AtomicInteger object instead

Review (1)

- A functional interface is an interface that declares *exactly one abstract method*
- Lambda expressions can be used only where a functional interface is expected
- Lambda expressions let us
  - pass code concisely
  - provide the implementation of the abstract method of a functional interface directly inline, and
  - treat the whole expression as an instance of a functional interface

Review (2)

- Method references let us reuse an existing method implementation and pass it around directly
- Java 8 provides a list of common functional interfaces in the `java.util.function` package,
  - E.g. `Predicate<T>, Function<T,R>, Supplier<T>, Consumer<T>`
- These functional interfaces have several default methods that can be used to combine lambda expressions
- Threads enable us to run our programs more efficiently by doing multiple tasks at the same time

Quiz on Thread