JDK 8 MOOC: Functional Programming in Java with Lambdas and Streams

Simon Ritter
Java Technology Evangelist

The Problem: External Iteration

```
List<Student> students = ...
double highestScore = 0.0;

for (Student s : students) {
  if (s.getGradYear() == 2011) {
    if (s.getScore() > highestScore)
      highestScore = s.getScore();
  }
}
```

- Our code controls iteration
- *Inherently serial:* iterate from beginning to end
- Not thread-safe
  - Business logic is stateful
  - Mutable accumulator variable

Internal Iteration With Lambda Expressions

```
List<Student> students = ...
double highestScore = students
            .filter(Student s -> s.getGradYear() == 2011)
            .map(Student s -> s.getScore())
            .max();
```

- More readable
- More abstract
- Less error-prone

This slide is intended to be conceptual. A little more work is needed to get this code to compile.

Lambda Expression Types

- A Lambda expression is an anonymous function
  - It is not associated with a class
- But Java is a strongly typed language
  - So what is the type of a Lambda expression?
- A Lambda expression can be used wherever the type is a functional interface
  - This is a single abstract method type
  - The Lambda expression provides the implementation of the abstract method

- Variable assignment

```
Callable c = () -> process();
```

- Method parameter

```
new Thread(() -> process()).start();
```

Functional Interface Definition

- An interface
- Has only one abstract method
- Before JDK 8 this was obvious
  - Only one method
- JDK 8 introduces default methods
  - Multiple inheritance of behaviour for Java
- JDK 8 also now allows static methods in interfaces
- `@FunctionalInterface` annotation

Functional Interfaces

Examples

```
interface FileFilter     { boolean accept(File x); }
interface ActionListener { void actionPerformed(…); }
interface Callable<T>    { T call(); }
```

Is This A Functional Interface?

```
@FunctionalInterface
public interface Runnable {
  public abstract void run();
}
```

Is This A Functional Interface?

```
@FunctionalInterface
public interface Predicate<T> {
  ~~default Predicate<T> and(Predicate<? super T> p) {…};~~
  ~~default Predicate<T> negate() {…};~~
  ~~default Predicate<T> or(Predicate<? super T> p) {…};~~
  ~~static <T> Predicate<T> isEqual(Object target) {…};~~
  boolean test(T t);
}
```

java.util.function Package

Consumer<T>

Operation That Takes a Single Value And Returns No Result

```
String s -> System.out.println(s)
```

Supplier

A Supplier Of Results

```
() -> createLogMessage()
```

Function<T,R>

A Function That Accepts One Argument And Returns A Result

```
Student s -> s.getName()
```

Predicate

A Boolean Valued Function Of One Argument

```
Student s -> s.graduationYear() == 2011
```

Method References

```
FileFilter x = (File f) -> f.canRead();
FileFilter x = File::canRead;
```

Lambda

Method Ref

```
(args) -> ClassName.staticMethod(args)
ClassName::staticMethod
```

Lambda

Method Ref

```
(arg0, rest) -> arg0.instanceMethod(rest)
instanceOf
ClassName::instanceMethod
```

Lambda

Method Ref

```
(args) -> expr.instanceMethod(args)
expr::instanceMethod
```

Lambda

Method Ref

```
(String s) -> Integer.parseInt(s);
Integer::parseInt
```

Lambda

Method Ref

```
(String s, int i) -> s.substring(i)
String::substring
```

Lambda

Method Ref

```
Axis a -> getLength(a)
this::getLength
```

ORACLE®

Lesson 2: Introduction To The Streams API

Simon Ritter
Java Technology Evangelist

Imperative Programming

Names And Values

- Use variables as an association between names and values
- Use sequences of commands
  - Each command consists of an assignment
  - Can change a variable's value
  - Form is <variable_name> = <expression>
  - Expressions may refer to other variables
    - Whose value may have been changed by preceding commands
  - Values can therefore be passed from command to command
  - Commands may be repeated through loops

VS

Functional Programming

Names And Values

- Based on structured function calls
- Function call which calls other functions in turn (composition)

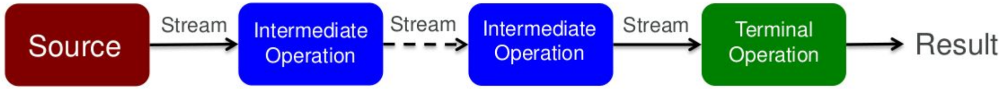
```
<function1>(<function2>(<function3> … ) … )
```

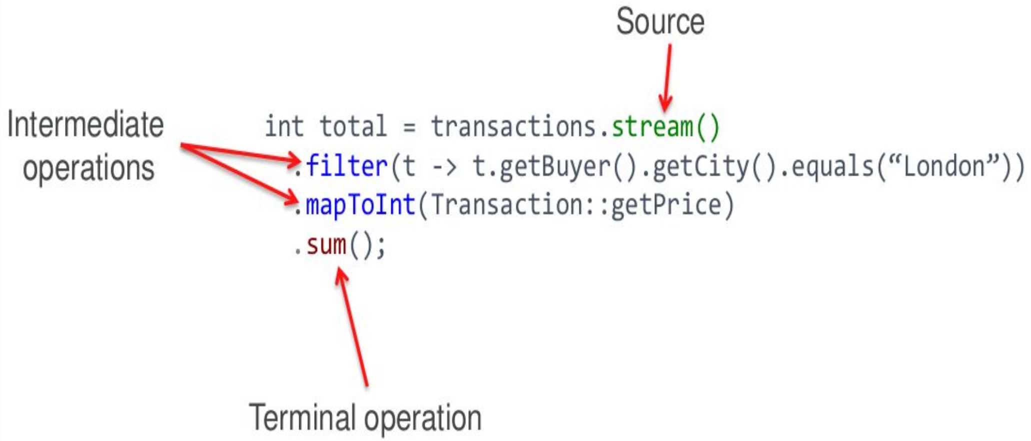
- Each function receives values from, and passes values back the calling function
- Names are only used as formal parameters
  - Once value is assigned it can not be changed
- No concept of a command, as used in imperative code
  - Therefore no concept of repetition
- Functional programming allows functions to be treated as values
  - This is why Lambda expressions were required in JDK 8
  - To make this much simpler than anonymous inner classes

Stream Overview

Pipeline

- A stream pipeline consists of three types of things
  - A source
  - Zero or more intermediate operations
  - A terminal operation
    - Producing a result or a side-effect





```
int total = transactions.stream()
        .filter(t -> t.getBuyer().getCity().equals("London"))
        .mapToInt(Transaction::getPrice)
        .sum();
```

- `stream()`
  - Provides a sequential stream of elements in the collection
- `parallelStream()`
  - Provides a parallel stream of elements in the collection
  - Uses the fork-join framework for implementation

Filtering And Mapping

- `distinct()`
  - Returns a stream with no duplicate elements
- `filter(Predicate p)`
  - Returns a stream with only those elements that return true for the `Predicate`
- `map(Function f)`
  - Return a stream where the given `Function` is applied to each element on the input stream
- `mapToInt()`, `mapToDouble()`, `mapToLong()`
  - Like `map()`, but producing streams of primitives rather than objects

- `skip(long n)`
  - Returns a stream that skips the first *n* elements of the input stream
- `limit(long n)`
  - Returns a stream that only contains the first *n* elements of the input stream
- `sorted(Comparator c)`
  - Returns a stream that is sorted with the order determined by the `Comparator`
  - `sorted()` with no arguments sorts by natural order
- `unordered()`
  - Inherited from `BaseStream`
  - Returns a stream that is unordered (used internally)
  - Can improve efficiency of operations like `distinct()` and `groupingBy()`

findFirst() and findAny(), you can narrow your search with a filter() statement upstream.

- `findFirst()`
  - The first element that matches
- `findAny()`
  - Works the same way as `findFirst()`, but for a parallel stream
- `boolean allMatch(Predicate p)`
  - Whether all the elements of the stream match using the `Predicate`
- `boolean anyMatch(Predicate p)`
  - Whether any of the elements of the stream match using the `Predicate`
- `boolean noneMatch(Predicate p)`
  - Whether no elements match using the `Predicate`
- `collect(Collector c)`
  - Performs a mutable reduction on the stream
- `toArray()`
  - Returns an array containing the elements of the stream

- `count()`
  - Returns how many elements are in the stream
- `max(Comparator c)`
  - The maximum value element of the stream using the `Comparator`
  - Returns an `Optional`, since the stream may be empty
- `min(Comparator c)`
  - The minimum value element of the stream using the `Comparator`
  - Returns an `Optional`, since the stream may be empty
- `average()`
  - Return the arithmetic mean of the stream
  - Returns an `Optional`, as the stream may be empty
- `sum()`
  - Returns the sum of the stream elements
- `forEach(Consumer c)`
  - Performs an action for each element of this stream
- `forEachOrdered(Consumer c)`
  - Like `forEach`, but ensures that the order of the elements (if one exists) is respected when used for a parallel stream

JDK 8 Libraries

- There are 95 methods in 23 classes that return a `Stream`
  - Many of them, though are intermediate operations in the `Stream` interface
- 71 methods in 15 classes can be used as practical `Stream` sources