

**Lambdas and the Streams API**

Pieter van den Hombergh  
Fontys Hogeschool voor Techniek en Logistiek  
May 6, 2018

HOM/FHTeL      Lambdas and the Streams API      May 6, 2018      1/18

Lambdas and the Streams API  
HOM

New in java8  
Streams a.k.a. pipeline  
MAP RED  
Demos

New in java8

Streams a.k.a. pipeline

Map reduce pipeline

Demos

HOM/FHTeL      Lambdas and the Streams API      May 6, 2018      2/18

Lambdas and the Streams API  
HOM

New in java8  
Streams a.k.a. pipeline  
MAP RED  
Demos

**New features in Java8**

Relevant to concurrency:

- Lambda expressions.
  - only just syntactic sugar
  - shorter source code
  - less generated code (no anonymous inner classes)
  - easy passing a function to methods. e.g, work on a collection
- Single Abstract Method (SAM) interfaces.
  - Help in defining use for lambda.
- Streams
  - New way to deal with data sources and operations on them
  - IO, collections, arrays
  - parallelisable
  - fluent coding pattern

HOM/FHTeL      Lambdas and the Streams API      May 6, 2018      3/18

Lambdas and the Streams API  
HOM

New in java8  
Streams a.k.a. pipeline  
MAP RED  
Demos

**New? not really. . .**

Lambda expressions also known as closures are no Java invention. In fact java was a bit late. But the implementation is very powerful when considering parallelism.

In the examples (on the net) you can see that using java is moving a bit towards **functional programming**.

HOM/FHTeL      Lambdas and the Streams API      May 6, 2018      4/18

Lambdas and the Streams API  
HOM

New in java8  
Streams a.k.a. pipeline  
MAP RED  
Demos

### Changes to class file

New class file format (similar to java6 to java7) so a few issues with e.g. instrumentation (coverage). The important new instruction `invokedynamic` is already in java7.

- Lambdas and the Streams API
- HOM
- New in java8
- Streams a.k.a. pipeline
- MAP RED
- Demos

HOM/FHTeL

Lambdas and the Streams API

May 6, 2018

5/18

### Lambda expressions

A lambda expression, or closure is all about defining a method in line with the code. An anonymous **function**.

What are functions?

- methods **without** side effects.
- that do **Not** change (mutate) the parameters that go in
- that produce a result

**No** Side Effects or **no** changes to objects works well with concurrency: Immutable objects.

Some languages, on the JVM, like scala already take this a step further.

```
// argument list      arrow token      body
(int x, int y) -> x + y
```

<http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html>

- Lambdas and the Streams API
- HOM
- New in java8
- Streams a.k.a. pipeline
- MAP RED
- Demos

HOM/FHTeL

Lambdas and the Streams API

May 6, 2018

6/18

### Examples

```
(int x, int y) -> x + y // function add
() -> 42 // source
(String s) -> { System.out.println(s); } // sink
```

- Lambdas and the Streams API
- HOM
- New in java8
- Streams a.k.a. pipeline
- MAP RED
- Demos

HOM/FHTeL

Lambdas and the Streams API

May 6, 2018

7/18

### Our friend Runnable

Runnable matches the requirement of functional interface or **single abstract method**.

```
public class RunnableTest {
    public static void main(String[] args) {
        System.out.println("==_RunnableTest_==");

        // Anonymous Runnable
        Runnable r1 = new Runnable(){

            @Override
            public void run(){
                System.out.println("Hello_world_one!");
            }
        };

        // Lambda Runnable
        Runnable r2 = () -> System.out.println("Hello_world_two!");

        // Run em!
        r1.run();
        r2.run();
    }
}
```

- Lambdas and the Streams API
- HOM
- New in java8
- Streams a.k.a. pipeline
- MAP RED
- Demos

HOM/FHTeL

Lambdas and the Streams API

May 6, 2018

8/18

### Comparable

- Comparable as example
- Note: *take care when using simple subtraction to implement comparable.*

```
X arr = ...;
Arrays.sort( arr, ( a, b )-> a.intValue() - b.intValue() );
```

- comparison based on comparable members
- Note that `java.util.List` has default methods like `sort(...)`.

```
List<Student> list = ...;
list.sort( ( a, b )-> a.getFirstName().compareTo( b.getFirstName() ) );
or, even shorter ( with import static java.util.Comparable.comparing; )
list.sort( comparing( Student::getFirstName ) );
```

HOM/FHTed

Lambdas and the Streams API

May 6, 2018

Lambdas and the Streams API  
HOM

New in java8  
Streams a.k.a. pipeline  
MAP RED  
Demos

### Map Reduce Pipeline

**Task** Find all transactions of type `grocery` and return a list of transaction IDs sorted in decreasing order of transaction value.

Three steps are involved:

- 1 filter for GROCERY
- 2 sort by value
- 3 map transAction to id

HOM/FHTed

Lambdas and the Streams API

May 6, 2018

Lambdas and the Streams API  
HOM

New in java8  
Streams a.k.a. pipeline  
MAP RED  
Demos

### Java SE 7 code

The legacy java way of doing it:

```
List<Transaction> groceryTransactions
= new ArrayList<>();
for(Transaction t: transactions){ //filter
    if(t.getType() == Transaction.GROCERY){
        groceryTransactions.add(t);
    }
}
Collections.sort(groceryTransactions, new Comparator(){
    public int compare(Transaction t1, Transaction t2){
        return t2.getValue().compareTo(t1.getValue());
    }
});
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: groceryTransactions){ // map and collect
    transactionIds.add(t.getId());
}
```

HOM/FHTed

Lambdas and the Streams API

May 6, 2018

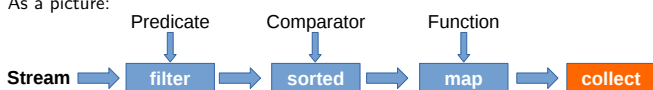
Lambdas and the Streams API  
HOM

New in java8  
Streams a.k.a. pipeline  
MAP RED  
Demos

### Java 8 Pipeline

```
List<Integer> transactionIds =
    transactions.stream()
        .filter(t -> t.getType() == Transaction.GROCERY)
        .sorted(comparing(Transaction::getValue))
        .map(Transaction::getId)
        .collect(toList());
```

As a picture:



- The operations are done 'at once'.
- The stream starts as soon as there is a terminal operation, as in this case the `collect`.
- There are no intermediate results (collections).
- Filtering trims down the remaining work.
- Doing this in parallel is trivially easy: replace `stream()` with `parallelStream()`.

HOM/FHTed

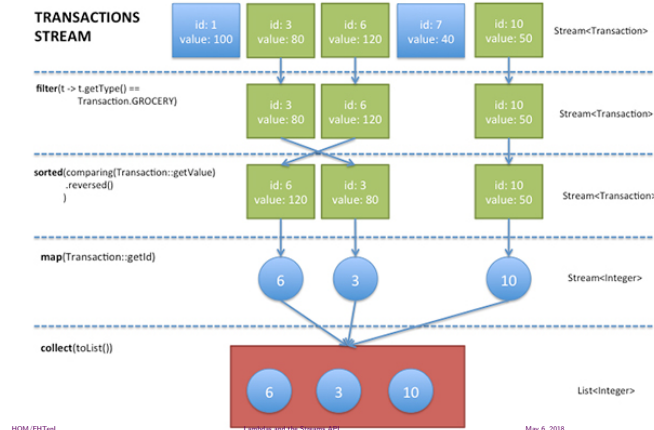
Lambdas and the Streams API

May 6, 2018

Lambdas and the Streams API  
HOM

New in java8  
Streams a.k.a. pipeline  
MAP RED  
Demos

Pipeline in action



Lambdas and the Streams API  
 HOM  
 New in java8  
 Streams a.k.a. pipeline  
 MAP RED  
 Demos  
 13/18

Stream vs Iterable

- All Java collections are **Iterable**.
- New is that **Iterable** provides a **void** `forEach(Consumer<? super T> action)`.
- The streams also provide a sequence of elements.
- Both are **one time use** only. You can neither restart as sequence nor a stream. They are 'consumed'.

So what's the difference?

- Collections are about data. Iterators typically produce all values in the collection. (for each loop)
- and streams are about computations (functions).
- A stream can apply a sequence of functions to each element passes, and evaluates these functions only when **really** needed. This is called **Lazy evaluation**.

Lambdas and the Streams API  
 HOM  
 New in java8  
 Streams a.k.a. pipeline  
 MAP RED  
 Demos  
 14/18

Peek: Looking a stream go by

```

List<String> l = Stream.of( "one", "two", "three", "four" )
    .peek( e
        -> System.out.println( "all values: " + e ) )
    .filter( e
        -> e.length() > 3 )
    .peek( e
        -> System.out.println( "Filtered value: " + e ) )
    .map( String::toUpperCase )
    .peek( e
        -> System.out.println( "Mapped value: " + e ) )
    .collect( Collectors.toList() );
  
```

this produces:  
 all values: one  
 all values: two  
 all values: three  
 Filtered value: three  
 Mapped value: THREE  
 all values: four  
 Filtered value: four  
 Mapped value: FOUR

- The elements one, two ... are "pulled" through the pipe, one after the other, but no further then needed.
- element one and two never come past the filter stage.

Lambdas and the Streams API  
 HOM  
 New in java8  
 Streams a.k.a. pipeline  
 MAP RED  
 Demos  
 15/18

Filter and map example

```

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
List<Integer> twoEvenSquares =
    numbers.stream()
        .filter(n -> {
            System.out.println("filtering " + n);
            return n % 2 == 0;
        })
        .map(n -> {
            System.out.println("mapping " + n);
            return n * n;
        })
        .limit(2)
        .collect(toList());
  
```

- Note the curly braces and return in the lambda bodies.
- Each element runs through the **complete** stream before the next is considered
- So the whole process stops when the limit operation is 'satisfied'.

Lambdas and the Streams API  
 HOM  
 New in java8  
 Streams a.k.a. pipeline  
 MAP RED  
 Demos  
 16/18

Now that is lazy...

### Some Stream functions

#### Intermediate (filter)

- **Stream**<T> filter(Predicate<? super T> predicate),
- **Stream**<T> limit(long maxSize)

#### Map or transform

- <R> **Stream**<R> map(Function<? super T,? extends R> mapper)

#### Terminal (non shortcut)

- **Optional**<T> reduce(BinaryOperator<T> accumulator)
- **Optional**<T> max(Comparator<? super T> comparator)
- **Optional**<T> min(Comparator<? super T> comparator)

#### Terminal (shortcut)

- **boolean** allMatch(Predicate<? superT>)
- **boolean** anyMatch(Predicate<? superT>)
- **Optional**<T> findAny()

HOM/FHTeL

Lambdas and the Streams API

May 6, 2018

17/18

Lambdas and the Streams API  
HOM

New in java8  
Streams a.k.a pipeline

MAP RED

Demos

### Creating streams is easy

```
List<String> myList
    = Arrays.asList( "a1", "a2", "b1", "c2", "c1" );

myList
    .stream()
    .filter( s -> s.startsWith( "c" ) ).
    map( String::toUpperCase ).
    sorted().
    forEach( System.out::println );
```

This, just as the remaining example com from the tutorial by Benjamin Winterberg. See <http://winterbe.com/>  
So lets dive in and discuss things using netbeans.

HOM/FHTeL

Lambdas and the Streams API

May 6, 2018

18/18

Lambdas and the Streams API  
HOM

New in java8  
Streams a.k.a pipeline

MAP RED

Demos