Java Fundamentals 2017

LAMBDAS & STREAMS

Arnel Pällo

06.02.2017

TODO

- Lambdas
  - Syntax and usage
  - Functional interfaces
  - Cute lambdas (λ composition)
  - Method references
  - Exception handling

TODO

- Streams
  - Intro & why should you care
  - Intermediate operations
  - Terminal operations
  - Grouping and mapping
  - Infinite streams

λ - anonymous function

```
new Thread(
    new Runnable() {
      public void run() {
          System.out.println("Hello world!");
      }
    }
).start();
```

λ - anonymous function

```
new Thread(
   new Runnable() {
     public void run() {
         System.out.println("Hello world!");
     }
   }
).start();

new Thread(
   () -> System.out.println("Hello")
).start();
```

Function components

- name
- parameters
- body
- return type
- enclosing class boilerplate

```
new Runnable() {
  public void run() {
    System.out.println("Hello!");
  }
};
```

λ components

- parameters
- body

```
Runnable runnable =
    () // parameters
    ->
    System.out.println("Hello!"); // body
```

- name - not used
- return type - inferred
- boilerplate - obliterated!

λ Syntax

```
(params) -> body
```

λ parameters

- Type inferred
  - `(int foo) -> ... vs (foo) -> ...`
- () optional for single param λ
  - `foo -> ...`
- () required otherwise
  - `() -> ...  (foo, bar) -> ...`

λ body

- multiline or single line

```
param -> {
  ...
  return param * 2;
}
// one line - return value inferred
param -> param * 2
```

λ body

- multiline or single line

```
param -> {
    ...
    return param * 2;
}
```

```
// one line - return value inferred
param -> param * 2
```

More lambdas

```
Runnable r = () -> println("hello");
Consumer<String> c = st -> println(st);
Supplier<String> s = () -> "hello";
Function<String, Boolean> f = str ->
                    str.startsWith("hello");
Predicate<String> p = str ->
                    str.startsWith("hello");
Comparator<Integer> cmp = (a, b) -> a-b;
```

Where can I use a λ ?

- everywhere
- where an instance of a functional interface can be used

Functional Inteface?

- Has exactly one abstract method
- λ must match that single method's signature
- Method reference points to a method that fits that single M.

Functional interfaces

```
interface Runnable {
  void run();
}
```

```
interface Callable<V> {
  V call() throws Exception;
}

interface ChangeListener extends EventListener {
  void stateChanged(ChangeEvent e);
}
```

Not functional interfaces

```
interface TooManyFunctions {
    int toInt();
    long toLong();
}
```

```
interface ExtendedRunnable extends Runnable{
  int count();
}

interface EventListener {
}
```

Back to functional interfaces

```
interface IAmFunctional {
    int toInt();
    default int getUltimateAnswer () {
        return 42;
    }
}

interface Derived extends IAmFunctional {
}
```

Type Safety

```
interface IntPredicate {
   boolean test(int value);
}

void foo(IntPredicate p) { ... }

obj.foo(val -> true); //OK
obj.foo(val -> val < 42); //OK
obj.foo((String val) -> val.length()); //E
obj.foo(val -> String.valueOf(val)); //ERR
```

Matching lambdas

```
Runnable runner = () -> println("hello");
void run();

Consumer<String> cons = string -> println(string);
void accept(T t);

Supplier<String> sup = () -> "hello";
T get();

Function<String, Boolean> f = string ->
                              string.startsWith("hello");
R apply(T t);
```

Using functional interfaces

```
class TaskProcessor {
  public void run(MyFunction func) {
     try {
       this.addValue(func.calculate();
     }
     catch (Exception e) {
        // cry
     }
  }
}
```

Using functional interfaces

```
taskProcessor.run(new MyFunction() {
  public int calculate() {
    return 42;
  }
});
```

```
taskProcessor.run ( () -> 42 );
```

Fluent API + λ

```
Stream.of(1.5, 3.5, 2.5)

  .filter(num -> num > 2)
  .filter(num -> num % 2 == 0)
  .map(num -> calculateArea(num))
  .map(num -> calculateWeight(num))
  .map(num -> getCost(num)
  .sorted((a, b) -> b - a)
  .collect(toList())
```

Lambdas as glue code

- use many small lambdas
- avoid multi line lambdas
  - split into several λ
  - move logic to separate method
    - call that method in λ

Avoid Multiline Lambdas

```
obj.process((a,b) -> {
   // do something
   // do moar stuff
    return answer;
});
// BAD! instead, use:
obj.process((a, b) -> doStuff(a,b));

int doStuff (int a, int b) {
   ...
}
```

Method references

- pointer to a method
- Its signature must fit ...
- the single abstract method ...
- of the functional interface

Method references

```
PrintStream out = System.out;
Stream.of(1.5, 3.5, 2.5)
      .filter(num -> num > 2)
      .map(num -> Math.round(num+1))
      .map(num -> num.intValue())
      .forEach(num -> out.println(num));
```

Method references

```
PrintStream out = System.out;
Stream.of(1.5, 3.5, 2.5)
      .filter(num -> num > 2)
      .map(num -> num + 1)
      .map(num -> Math.round(num))
      .map(num -> num.intValue())
      .forEach(num -> out.println(num));
```

ref static method

```
PrintStream out = System.out;
Stream.of(1.5, 3.5, 2.5)
      .filter(num -> num > 2)
      .map(num -> num + 1)
      .map(Math::round)
      .map(num -> num.intValue())
      .forEach(num -> out.println(num));
```

ref instance method

```
PrintStream out = System.out;
Stream.of(1.5, 3.5, 2.5)
      .filter(num -> num > 2)
      .map(num -> num + 1)
      .map(Math::round)
      .map(Long::intValue)
      .forEach(num -> out.println(num));
```

ref instance method

```
PrintStream out = System.out;
Stream.of(1.5, 3.5, 2.5)
      .filter(num -> num > 2)
      .map(num -> num + 1)
      .map(Math::round)
      .map(Long::intValue)
      .forEach(out::println);
```

Ref with two+ params

```
Stream.of(3, 2, 1).sorted((a, b) -> a-b)

//static method - Integer.compare(a, b)
Stream.of(3, 2, 1).sorted(Integer::compare)

//a.compareTo(b)
Stream.of(3, 2, 1).sorted(Integer::compareTo)

Comparator<Integer> cmp = (a, b) -> a-b;
Stream.of(3, 2, 1).sorted(cmp::compare)
```

Using method references

```
Stream.of(1, 2, 3)
  .forEach(num -> System.out.println(num+1));
```

Using method references

```
Stream.of(1, 2, 3)
  .forEach(num -> System.out.println(num+1));
```

```
Stream.of(1, 2, 3)
   .map(num -> num + 1)
   .forEach(System.out::println);
```

Exceptions

```
void oops(String s) throws Exception {
  ...
}
```

```
void run () {
   Stream.of("Hello", "World")
     .forEach(s->this.oops(s)); //Error
}
```

Throw RuntimeException

```
void oops(String s) throws RuntimeException{
  ...
}
```

Refactor Interface

```
interface MyConsumer {
  void accept(String s) throws Exception;
}
```

Handle Exceptions

```
Stream.of("Hello", "World")
  .forEach(s -> {
     try {
       this.oops(s);
     }
     catch (Exception e) {
      // handle
     }
  });
```

Wrap method call

```
void safeOops(String s) {
  try {
    obj.oops(s);
  }
  catch (Exception e) {
    // handle...throw RuntimeException
  }
}
Stream.of("Hello", "World")
    .forEach(s->this.safeOops(s));
```

Hack the interface

```
interface MyConsumer<T> extends Consumer<T> {
  default void accept(T t) {
    try {
      this.acceptThrows(t);
    }
    catch(Exception e) {
      // handle
    }
  }
  void acceptThrows(T elem) throws Exception;
}
```

Hack the interface

```
Stream.of("Hello", "World")
   .forEach(s->this.oops(s)); //ERROR
```

Hack the interface

```
Stream.of("Hello", "World")
   .forEach(
      s->this.oops(s)    //ERROR
   );
```

Hack the interface

```
Stream.of("Hello", "World")
   .forEach(
     (MyConsumer<String>) s->this.oops(s)
   );
```

Works, but is it worth it?

λ λ λ

- anonymous functions
- great syntax
- they point to functions and
- require functional interfaces
- replace anonymous inner classes

λ λ λ

- Use (args) -> method body
- or ::methodname
- Compose simple λ's
- Avoid
  - multi line λ
  - checked exceptions

Streams

Flow direction

```
List<Integer> nums = Arrays.asList(
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

List<Integer> rslt = new ArrayList<>();
for (int num : nums) {
    if (num % 2 == 1) {
        rslt.add(num * 2);
    }
}
```

Flow direction

```
List<Integer> nums = Arrays.asList(
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

List<Integer> rslt = nums.stream()
.filter(num -> num % 2 == 1)
.map(num -> num * 2)
.collect(toList());
```

Streams

- do not contain data
- composition of functions
- filter-map-reduce
- logically defines computation as a sequence of steps

Streams

- Do not modify source
- As lazy as possible
- Can be used once
- Fast
- Possibly unlimited

Using a Stream

1. Connect to a source
   - collection, generator f(), IO
2. 0 or more intermediate steps
   - filter, map, distinct
3. one terminal operation
   - collectors, reducers, findFirst

Streams

- Intermediate operations
  - lazy
  - return a stream
- Terminal operations
  - eager
  - return end result

Intermediate operations

- For building a pipeline
- Does not execute the pipeline (lazy execution)
- filter(), map(), distinct(), limit()

Filter

```
Stream<Integer> str = Stream.of(1, 2, 3)
  .filter(num -> num % 2 == 0);
```

Configures the stream to let only even values through.

Takes a Predicate

return true to keep an element

does not modify or replace the element

Map

```
Stream<Puppy> p = Stream.of(1, 2, 3, 4, 5)
   .filter(num -> num % 2 == 0)
   .map(num -> new Puppy(num));
```

Replaces all elements with a different one - can even change element type

Uses Function<FROM, TO>
Does not change number of elements

FlatMap

```
List<List<String>> linesAndWords = ...;

Stream<String> s = linesAndWords.stream()
   .flatMap(line -> line.stream());
```

```
List<Stream<String>> listOfStreams = ...;

Stream<String> lst = listOfStreams.stream()
   .flatMap(stream -> stream);
```

FlatMap

```
Stream.of(1, 2, 3)
   .map(num->Stream.of("A", "B"))
// Stream<Stream<String>>
```

```
Stream.of(1, 2, 3)
   .flatMap(num->Stream.of("A", "B"))
   .forEeach(System.out::print);
//ABABAB
```

Distinct & limit

```
Stream.of(1, 2, 3, 2, 1)
   .distinct()
   .forEach(System.out::print);
//123
```

```
Stream.of(1,2,3,4,5)
   .limit(3)
   .forEach(System.out::print);
//123
```

Skip & Limit

```
Stream.of(1,2,3,4,5)
   .limit(3)
   .skip(2)
   .forEach(System.out::print);
   // 3 or 345?
```

Terminal operations

- eager - executes the pipeline
- returns the end value
- reducers
- collectors

Reducers

```
Integer sum = Stream.of(1, 2, 3, 4, 5)
  .reduce(0, (tmp, el) -> tmp + el);
```

- calculates new value based on intermediate result and current element
- that value becomes the intermediate val for next

Specialized reducers

```
Stream.of("Hello", "world")
  .mapToInt(String::length)
  .sum(); // 10
```

```
Stream.of("Hello", "world")
  .count(); // 2
```

Collectors

```
Stream.of("Hello", "World")
  .map(String::length)
  .collect(Collectors.toList()); // [5, 5]
```

```
Stream.of("Hello", "World")
  .map(String::length)
  .collect(Collectors.toSet()); // [5]
```

Mapping

```
Stream.of("Hola", "World")
   .collect(
     Collectors.toMap(s->s, String::length)
   );

// {Hola=4, World=5}
```

Grouping and mapping

```
Stream.of("hello", "world", "is", "nice")
  .collect(groupingBy(s->s.length()));

Stream.of("hello", "world", "is", "nice")
   .collect(groupingBy(
     s->s.length(),
     mapping(s->s, Collectors.toList())
    ));

//{2=[is], 4=[nice], 5=[hello, world]}
```

Grouping and mapping

```
Stream.of("hello", "world", "is", "nice")
   .collect(groupingBy(s->s.length(),
     Collectors.counting()));

//{2=1, 4=1, 5=2}
```

Reducing Collectors

```
Stream.of("Hello", "World")
  .collect(Collectors.joining());
// HelloWorld
```

```
Stream.of("Hello", "World")
  .collect(Collectors.counting());
// 2
```

```
Stream.of("Hello", "World")
  .collect(Collectors.reducing(...));
```

java.util.stream.Collectors

- Many helpful collectors
- Often duplicates capabilities of intermediate steps
- Use static imports

```
.collect(toList());
```

java.util.stream.Collectors

- toList()/toSet()/toMap(key, val)
- toCollection(collectionSupplier)
- groupingBy()
- mapping()
- counting()
- joining()

java.util.stream.Collectors

- reducing(initial, BinaryOperator)
- minBy() / maxBy()
- partitioningBy()
- summingDouble/Int/Long()
- etc

Numeric Streams

- IntStream, LongStream, DoubleStream
- Primitives only
- Aggregate functions(max, sum, average)
- normalStream.mapToInt(...)

Primitives Only

- map() maps to same type
- `IntStream.of(1,2,3)...`
  - `.map(i->i*2);  // OK`
  - `.map(i->" ");  // ERROR`
  - `.mapToObj(i->" "); // OK`
  - `.mapToDouble(i->i * 2.0) // OK`

Lazy Streams

```
IntStream str = IntStream.rangeClosed(0,9)
   .filter(LazyStr::largerThanFour)
   .filter(LazyStr::isEven)
   .map(LazyStr::doubleNum);

System.out.println("Stream created");

OptionalInt result = str.findFirst();
System.out.println("Stream processed");
System.out.println(result.getAsInt());
```

Lazy Streams

```
private static boolean largerThanFour(int n) {
    System.out.println("largerThanFour: " + n);
    return n > 4;
}
private static boolean isEven(int n) {
    System.out.println("isEven: " + n);
    return n % 2 == 0;
}
private static int doubleNum(int n) {
    System.out.println("doubleNum: " + n);
    return n * 2;
}
```

Lazy Streams

```
Stream created
largerThanFour: 0
largerThanFour: 1
largerThanFour: 2
largerThanFour: 3
largerThanFour: 4
largerThanFour: 5
isEven: 5
largerThanFour: 6
isEven: 6
doubleNum: 6
Stream processed
12
```

```
.filter(LazyStr::largerThanFour)


.filter(::isEVen)

.map(::doubleNum)
str.findFirst()
```

∞ Infinite streams

- Streams that never run out of values
- Whole stream can not be collected
- Needs a short-circuit
  - findFirst()
  - limit(n)
  - allMatch(p) & noneMatch(p)

Creating Infinite Streams

Creating Infinite Streams

Use a generator function

```
Stream<Double> soRandom =
                    Stream.generate(Math::random);

Stream<Integer> geometricalSequence =
        Stream.iterate(2, prev -> prev * 2);
```

∞

Finding prime numbers

```
LongStream.iterate(2, i -> i + 1)
      .filter((x) -> isPrime(x))
      .filter(x -> x > 1000)
      .findFirst()
      .getAsLong();

boolean isPrime(long x) {
   return LongStream
     .rangeClosed(2, (long) Math.sqrt(x))
     .allMatch(n -> x % n != 0);
}  //http://thestandardoutput.com/posts/check-number-primality-using-java-8-streams/
```

Ordering of steps

```
LongStream.iterate(2, i -> i + 1)
      .filter(x -> x > 1000)
      .filter((x) -> isPrime(x))
      .findFirst()
      .getAsLong();

boolean isPrime(long x) {
   return LongStream
     .rangeClosed(2, (long) Math.sqrt(x))
     .allMatch(n -> x % n != 0);
}  //http://thestandardoutput.com/posts/check-number-primality-using-java-8-streams/
```

Ordering of steps

```
LongStream.iterate(2, i -> i + 1)
      .skip(1000)
      .filter((x) -> isPrime(x))
      .findFirst()
      .getAsLong();

boolean isPrime(long x) {
   return LongStream
     .rangeClosed(2, (long) Math.sqrt(x))
     .allMatch(n -> x % n != 0);
} //http://thestandardoutput.com/posts/check-number-primality-using-java-8-streams/
```

Ordering of steps

```
LongStream.iterate(1000, i -> i + 1)
        .filter((x) -> isPrime(x))
        .findFirst()
        .getAsLong();
```

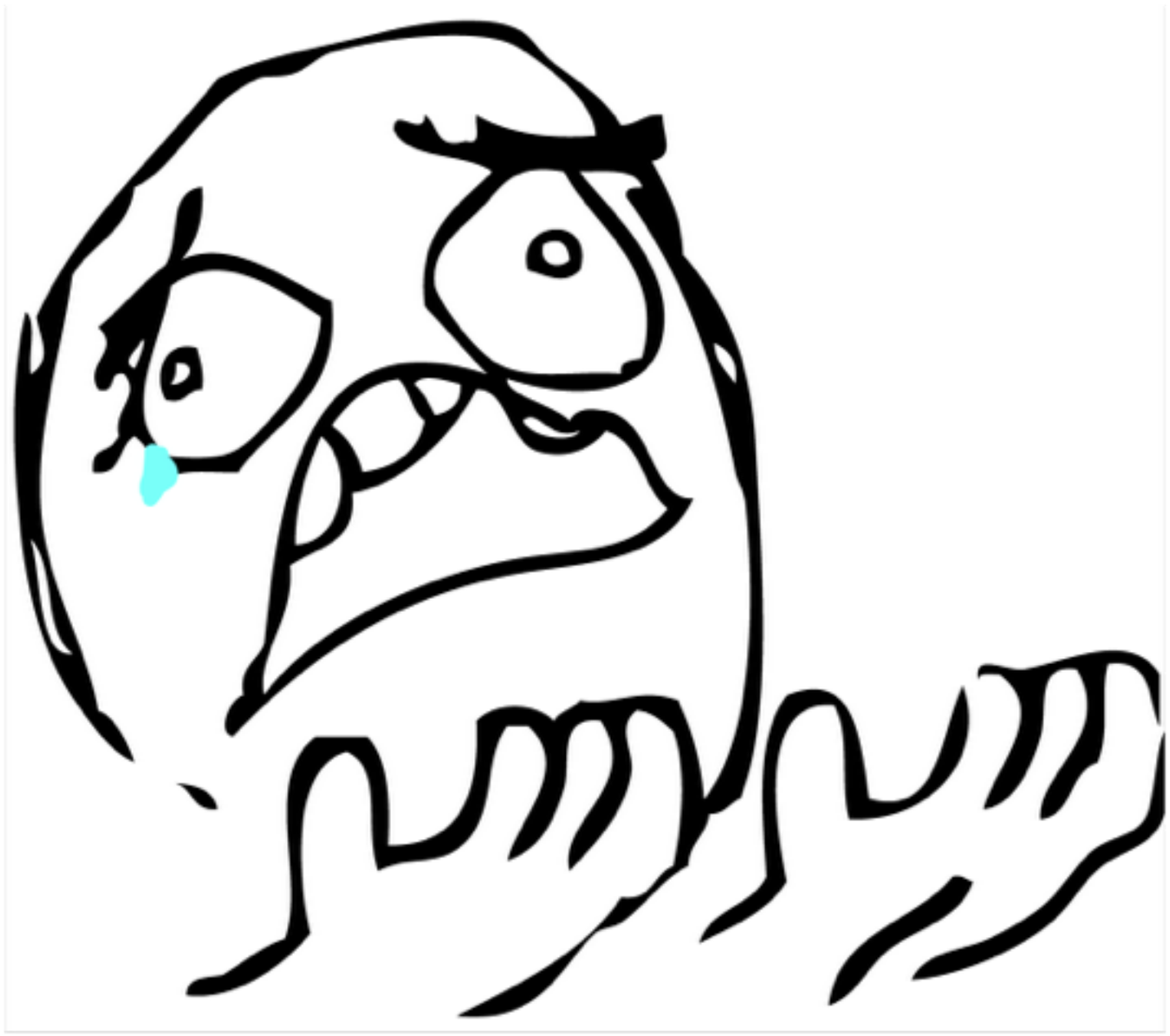
```
boolean isPrime(long x) {
   return LongStream
     .rangeClosed(2, (long) Math.sqrt(x))
     .allMatch(n -> x % n != 0);
}
```

```
//http://thestandardoutput.com/posts/check-number-primality-using-java-8-streams/
```

Conclusion

- Powerful way to transform data
- using functional programming concepts (map, filter, reduce).
- Leverages the power of λ
- but keep them cute and mind the ordering

Homework



Homework

- Find five most used words in a file
- Case insensitive
- Skip one-letter words
- Must Use stream API & λ
- Return a map, results in descending order

Homework

- Use the following to read lines:

```
Files.lines(Paths.get("lorem-ipsum.txt"))
```

(Returns a Stream<String>)

Unit test with expected output is included in the homework skeleton.

Homework tips

- Consider edge cases!
  - must work with any file, not only lorem-ipsum.txt
  - punctuation matters

Homework tips

- Don't forget the tests
- One is included, add more!
- Format your code!

Homework in GitHub

- clone or download the skeleton for homework 2

- **https://github.com/JavaFundamentalsZT/jf-hw-2-lambdas**