# Lambdas and streams

## New cool features of Java 8

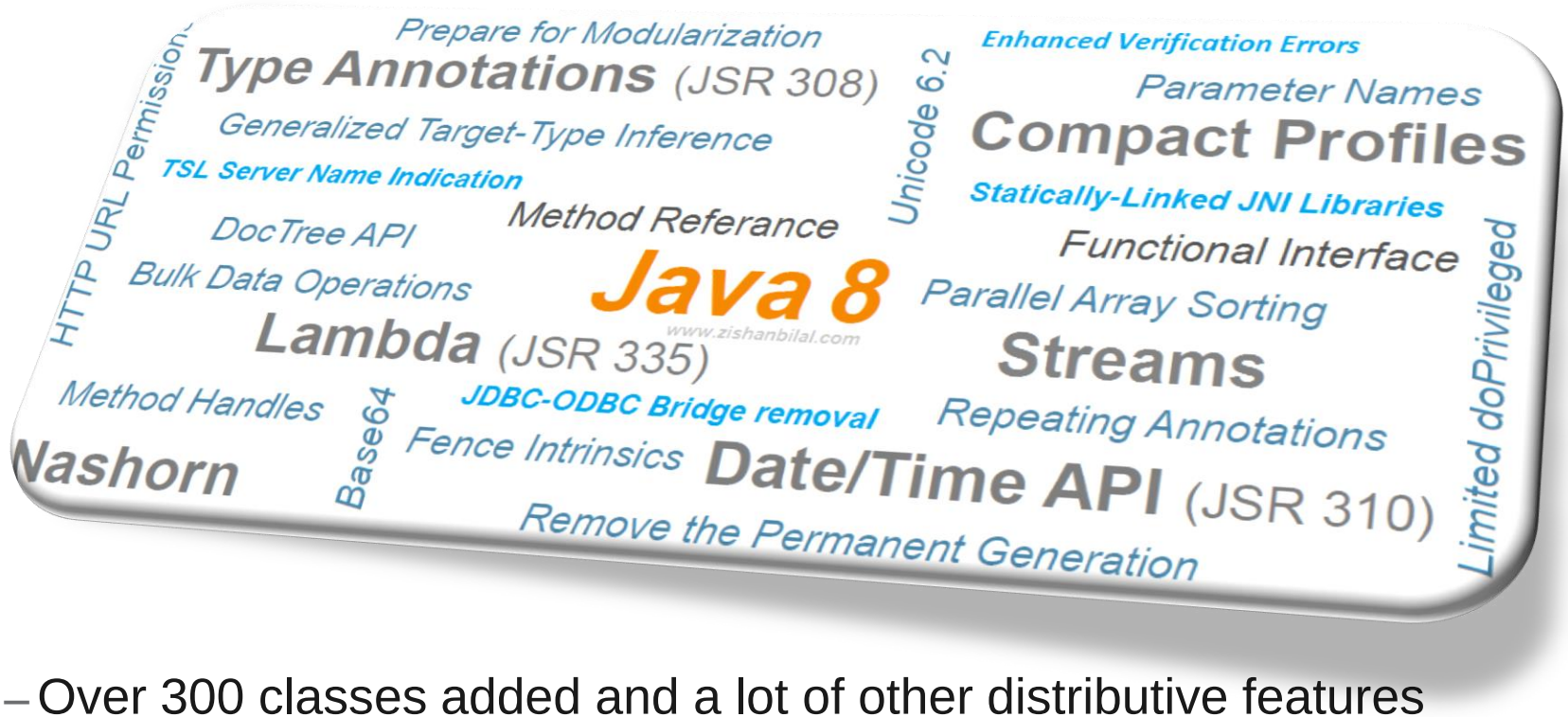
Ionuț Spălățelu, Java application developer
April 3, 2018

# Agenda

- Get in touch with lambdas
- Working with streams
- Working with primitive streams
- Advanced stream pipeline concepts

# Quick intro

## Java 8 is massive




- Over 300 classes added and a lot of other distributive features
- Core changes are to incorporate functional language features
- Lambda Expressions, Functional Interfaces, Stream Collection Types

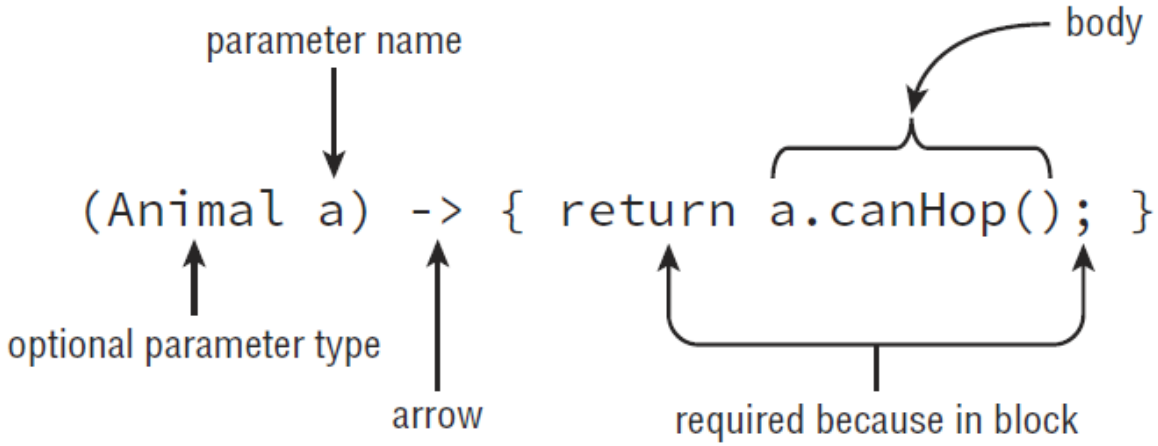
# Lambda expressions

- Anonymous implementation of a functional interface
- Can be passed to a method, like an object

Interface with a single **abstract** method

```
boolean test(Animal a);
```

Lambda syntax, including optional parts



# Let's see some lambdas

```
() -> { } good

() -> "Trex" good

(String a) -> { return a.startsWith("test"); } good

(String a) -> a.startsWith("test") good

a -> a.startsWith("test") good

(a,b) -> {a+b} bad

(Employee e1, Employee e2) ->
            e1.getFirstName().compareTo(e2.getFirstName()) good

(Employee e1,e2) -> e1.getName().compareTo(e2.getName()) bad
```

# Get some interfaces

| Functional Interfaces | # Parameters | Return Type | Single Abstract Method |
|---|---|---|---|
| `Supplier<T>` | 0 | T | get |
| `Consumer<T>` | 1 (T) | void | accept |
| `BiConsumer<T, U>` | 2 (T, U) | void | accept |
| `Predicate<T>` | 1 (T) | boolean | test |
| `BiPredicate<T, U>` | 2 (T, U) | boolean | test |
| `Function<T, R>` | 1 (T) | R | apply |
| `BiFunction<T, U, R>` | 2 (T, U) | R | apply |
| `UnaryOperator<T>` | 1 (T) | T | apply |
| `BinaryOperator<T>` | 2 (T, T) | T | apply |

# Let's implement them…

```
Supplier<Dog> s1 = () -> new Dog();
f():Dog

Consumer<String> c1 = s -> System.out.println(s);
f(String)

Predicate<ArrayList<String>> p1 = l -> l.contains("Gheorghe");
f(ArrayList):boolean

BiPredicate<String, String> bp1 = (a, b) -> a.startsWith(b);
f(String, String):boolean

Function<String, Integer> f1 = s -> s.length();
f(String):Integer

BiFunction<String, String, String> bf1 = (a,b) -> a.concat(b);
f(String,String):String
```

# Method references

- can be even shorter?? Yes, it can!

1. **Constructor references**

```
() -> new Dog();
```

Lambda

```
Dog::new;
```

Method reference

2. **Static methods**

```
list -> Collections.sort(list);
```

Lambda

```
Collections::sort;
```

Method reference

3. **Instance methods**

```
s -> str.startsWith(s);
```

Lambda

```
str::startsWith;
```

Method reference

```
s -> s.isEmpty();
```

Lambda

```
String::isEmpty;
```

Method reference

KEEP
CALM
AND
CODE

# Agenda

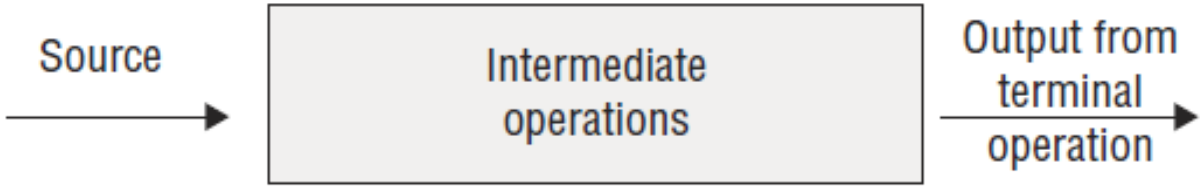
- Get in touch with lambdas
- Working with streams
- Working with primitive streams
- Advanced stream pipeline concepts

# What is a stream?

➔A stream is a sequence of elements from a source that supports data processing operations

- **source:** collections, arrays, I/O, generated…
- **pipelining:** lazy evaluation and short-circuiting
- **internal iteration:** iteration behind the scenes for you

Stream pipeline



# Creating stream sources

## Stream<T> (java.util.stream.Stream)

- **.empty():** returns an empty stream

```
Stream<String> empty = Stream.empty();
```

- **.of(T …):** returns a stream of specified elements

```
Stream<Integer> fromArray = Stream.of(1, 2, 3);
```

- **.stream() and .parallelStream()**

```
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> fromList = list.stream();
Stream<String> fromListParallel = list.parallelStream();
```

- **.generate(Supplier<T>) and .iterate(T seed, UnaryOperator<T>)**

```
Stream<Double> randoms = Stream.generate(Math::random);
Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);
```

# Using terminal operations

## Counting and finding

- .count()

```
Stream<String>  finit = Stream.of("monkey", "cat", "lion");
System.out.println(finit.count());              //  3
```

- . min() and .max(Comparator<T>)

```
Optional<String> min = finit.min(
                  (s1, s2) -> s1.length() - s2.length());
min.ifPresent(System.out::println);             // cat
max.ifPresent(a -> System.out.println(a));  // monkey
```

- . findAny() and findFirst()

```
Stream<String> infinite = Stream.generate(() -> "chimp");
finit.findAny().ifPresent(System.out::println);       // monkey
infinite.findAny().ifPresent(System.out::println);    // chimp
```

# Using terminal operations

## Matching and iterating

- .allMatch() , anyMatch() and noneMatch(Predicate<T>)

```
Stream <String> finit = Arrays.asList("str", "2", "num").stream();
Stream<String> infinite = Stream.generate(() -> "chimp");
Predicate<String> pred = x -> Character.isLetter(x.charAt(0));
finit.anyMatch(pred);                    //  true
finit.allMatch(pred);                    //  false
finit.noneMatch(pred);                   //  false
infinite.anyMatch(pred);                 //  true
```

- .forEach(Consumer<T>)

```
finit.forEach(System.out::print);              //str 2 num
```

# Using terminal operations

## Reducing and collecting

- .reduce(T identity, BinaryOperator<T> accumulator)

```
//pre Java 8 way
String[] array = new String[] { "w", "o", "l", "f" };
String result = "";
for (String s: array) result = result + s;

//Java 8 way
Stream<String> stream = Arrays.asList(array).stream();
String word = stream.reduce("", (s, c) -> s + c);
String word1 = stream.reduce("", String::concat);

int sum = numbers.stream().reduce(1, (a,b)-> a+b);
```

- .collect(Supplier<R>, BiConsumer<T,R>, BiConsumer<T,R>)

```
Stream<String> stream = Stream.of("w", "o", "l", "f");
StringBuilder word = stream.collect(StringBuilder::new,
                StringBuilder::append, StringBuilder::append); //wolf

TreeSet<String> set = stream.collect(TreeSet::new, TreeSet::add,
        TreeSet::addAll);
System.out.println(set); // [f, l, o, w]
```

# Intermediate operations

➔somewhere in the middle of the things 😊

- **.filter(Predicate<T>):** keeps only the matching elements
  .filter(e -> e.getAge()>30)
- **.distinct():** removes duplicate elements
- **.skip(n):** skips the first n elements
- **.limit(n):** keeps only the first n elements of the stream
- **.sorted(Comparator<T>)**
- **.map(Function<T,R>):** maps the elements of the stream of type T to another type, R

# Intermediate operations

- **.flatMap()**

How could you return a list of all the *unique characters* for a list of words?

- first attempt:

```
words.stream()
     .map(word -> word.split(""))//Stream<String[]>
     .distinct()
     .collect(toList());
```

- second attempt:

```
words.stream()
     .map(word -> word.split(""))
     .map(Arrays::stream) //Stream<Stream<String>>
     .distinct()
     .collect(toList());
```

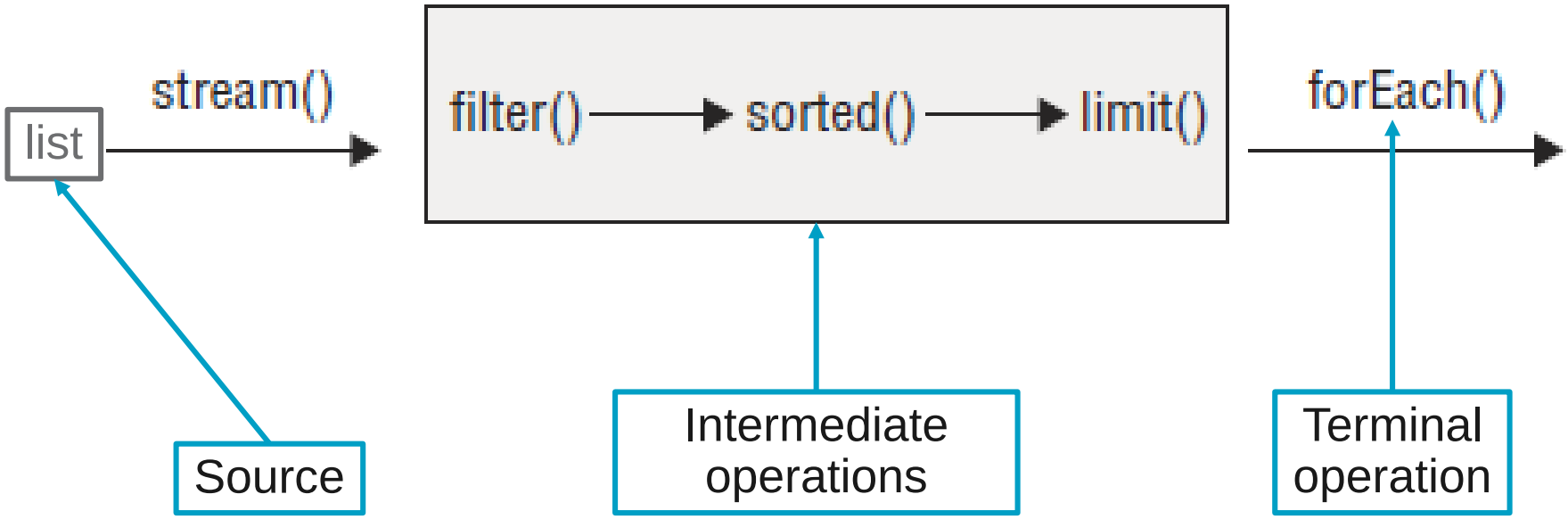
Still doesn't work!!! ☹

# Intermediate operations

# Putting together the pipeline


Get first two names of four letters sorted alphabetically

Arrays.*asList("Toby", "Anna", "Leroy", "Alex", "Jamie")*



# Agenda

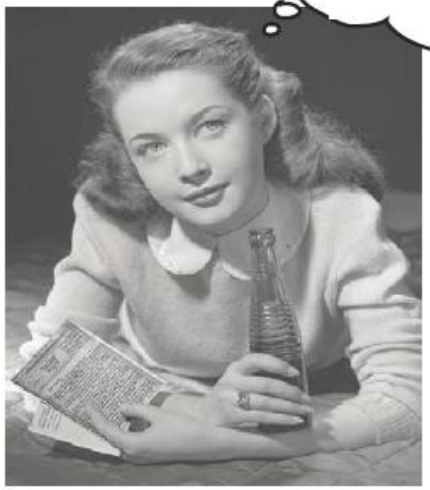
- Get in touch with lambdas
- Working with streams
- Working with primitive streams
- Advanced stream pipeline concepts

# Why primitive streams?

Let's calculate the total number of calories from the menu…
This is one approach:

```
int calories = menu.stream()
                   .map(Dish::getCalories)
                   .reduce(0, Integer::sum);
```

If only I could:
int calories = menu.stream()
.map(Dish::getCalories)
.**sum**();
That would be dreamy. But I know it's just a fantasy…



**What else we can do???**

# Primitive streams

**IntStream** to the rescue…

```
int calories = menu.stream()
                   .mapToInt(Dish::getCalories)
                   .sum();
```

Returns an `IntStream`

Returns a `Stream<Dish>`

You can convert it back to a general stream:

```
menu.stream().mapToInt(Dish::getCalories);
n = intStream.boxed();
```

Converting the numeric stream to a `Stream`

Converting a `Stream` to a numeric stream

# Primitive streams

## Create and use primitive streams

### .range(int start, int end)

```
IntStream s = IntStream.range(1,5);
s.forEach(System.out::println);   //1,2,3,4
```

### .rangeClosed(int start, int end)

```
IntStream s = IntStream.rangeClosed(1,5);
s.forEach(System.out::println);   //1,2,3,4,5
```

### .average()

```
OptionalDouble opt = IntStream.of(8, 3, 12, 4, 45, 88, 93)
                              .filter(n -> n%3==0)
                              .limit(5)
                              .sorted()
                              .average();

opt.ifPresent(System.out::println);
```

# Agenda

- Get in touch with lambdas
- Working with streams
- Working with primitive streams
- Advanced stream pipeline concepts

# Linking streams to source

What do you think it happens here?

```
List<String> cats = new ArrayList<>();
cats.add("Annie");
cats.add("Ripley");
Stream<String> stream = cats.stream();
cats.add("KC");
System.out.println(stream.count());      // 3
```

Remember that streams are lazily evaluated!

The stream pipeline runs first, looking at the source.

# Collecting Using Basic Collectors

## Collector vs Collectors

- .joining(CharSequence cs)

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
String result = ohMy.collect(joining(", "));
System.out.println(result); // lions, tigers, bears
```

- .toCollection(Supplier<T> s)

```
TreeSet<String> set = ohMy.collect(toCollection(TreeSet::new));
System.out.println(result); // [bears, lions, tigers]
```

# Collecting into Maps

- .toMap(Function k, Function v)

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
Map<String, Integer> map = ohMy.collect(
                                toMap(s -> s, String::length));
System.out.println(map); // {lions=5, bears=5, tigers=6}
```

**What if we want to reverse things…?**

```
Map<Integer, String> map = ohMy.collect(
                                toMap(String::length, k -> k));
                // BAD
```

**We have to handle the duplicate keys!**

```
Map<Integer, String> map = ohMy.collect(
    toMap(String::length, k -> k, (s1, s2) -> s1 + "," + s2));
System.out.println(map); // {5=lions,bears, 6=tigers}
```

**Chose whatever Map you like…**

```
TreeMap<Integer, String> map = ohMy.collect(
                    toMap(String::length, k -> k,
                    (s1, s2) -> s1 + "," + s2, TreeMap::new));
System.out.println(map); // // {5=lions,bears, 6=tigers}
```

# Grouping

- .groupingBy(Function<T,R> f)

```
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, List<String>> map = ohMy.collect(
                                    groupingBy(String::length));
System.out.println(map); // {5=[lions, bears], 6=[tigers]}
```

**Suppose you prefer a Set instead of List**

```
Map<Integer, Set<String>> map = ohMy.collect(
                                  groupingBy(String::length, toSet()));
System.out.println(map); // {5=[lions, bears], 6=[tigers]}
```

**…and a TreeMap**

```
TreeMap<Integer, Set<String>> map = ohMy.collect(
        groupingBy(String::length, TreeMap::new, toSet()));
System.out.println(map); // {5=[lions, bears], 6=[tigers]}
```

# Partitioning

- .partitioning(Predicate<T> p)

```
Map<Boolean, List<String>> map = ohMy.collect(
                          partitioningBy(s -> s.length() <= 5));
System.out.println(map); // {false=[tigers], true=[lions, bears]}
```

**We can change the List with a Set here too**

```
Map<Boolean, Set<String>> map = ohMy.collect(
              partitioningBy(s -> s.length() <= 7, toSet()));
System.out.println(map);//{false=[], true=[lions, tigers, bears]}
```

# NIO 2 additions

## Files.* (java.nio.file.Files)

- .walk(Path path)

```
Path path = Paths.get("/bigcats");
try {
Files.walk(path)          //Stream<Path>
     .filter(p -> p.toString().endsWith(".java"))
     .forEach(System.out::println);

} catch (IOException e) {
// Handle file I/O exception...
}
```

# NIO 2 additions

- .list(Path path)

```
try {
    Path path = Paths.get("dummy");
    Files.list(path)
        .filter(p -> !Files.isDirectory(p))
        .map(p -> p.toAbsolutePath())
        .forEach(System.out::println);
} catch (IOException e) {
    // Handle file I/O exception...
}
```

# NIO 2 additions

- .lines(Path path)

```
Path path = Paths.get("/logs/sdm.log");
try {

Files.lines(path)  //Stream<String>
    .forEach(System.out::println);

} catch (IOException e) {
// Handle file I/O exception...
}
```

# Hands on. Putting all into practice

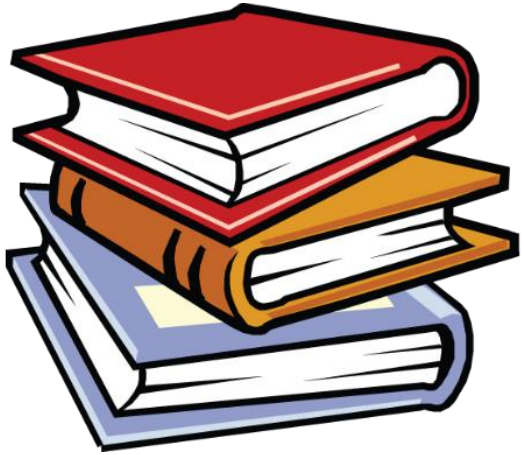
1. Find all transactions in the year 2011 and sort them by value (small to high).
2. What are all the unique cities where the traders work?
3. Find all traders from Cambridge and sort them by name.
4. Return a string of all traders' names sorted alphabetically.
5. Are any traders based in Milan?
6. Print all transactions' values from the traders living in Cambridge.
7. What's the highest value of all the transactions?
8. Find the transaction with the smallest value.

# References


1. Java 8 in Action – Alan Mycroft, Mario Fusco.

2. Oracle Certified Professional Java SE 8 Programmer II Study Guide – Jeanne Boyarsky & Scott Selikoff

3. Head First Java – Bert Bates & Kathy Sierra


To be Continued