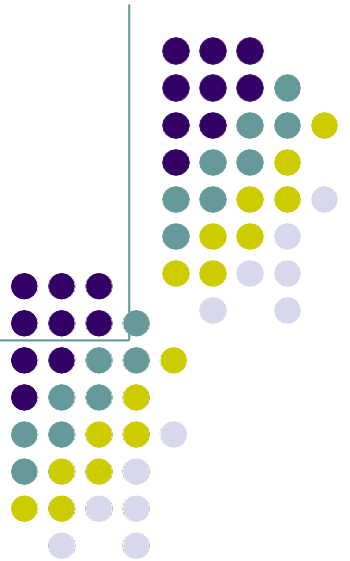
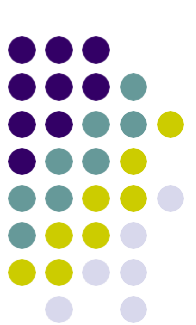


Lambdas & streams

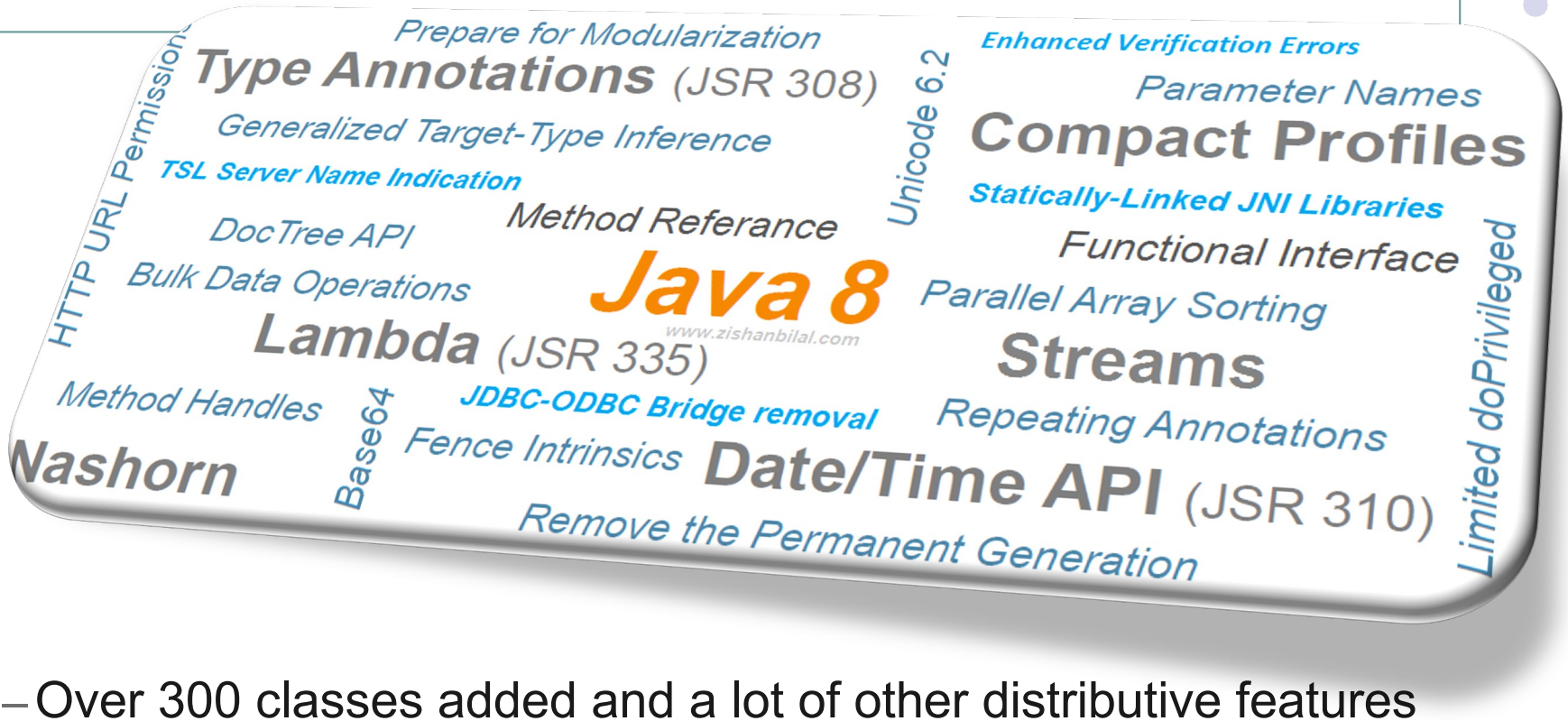
Ionut
Spalatelu

Outline



1. Get in touch with lambdas
2. Working with streams
3. Working with primitive streams
4. Advanced stream pipeline concepts

Quick intro



- 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

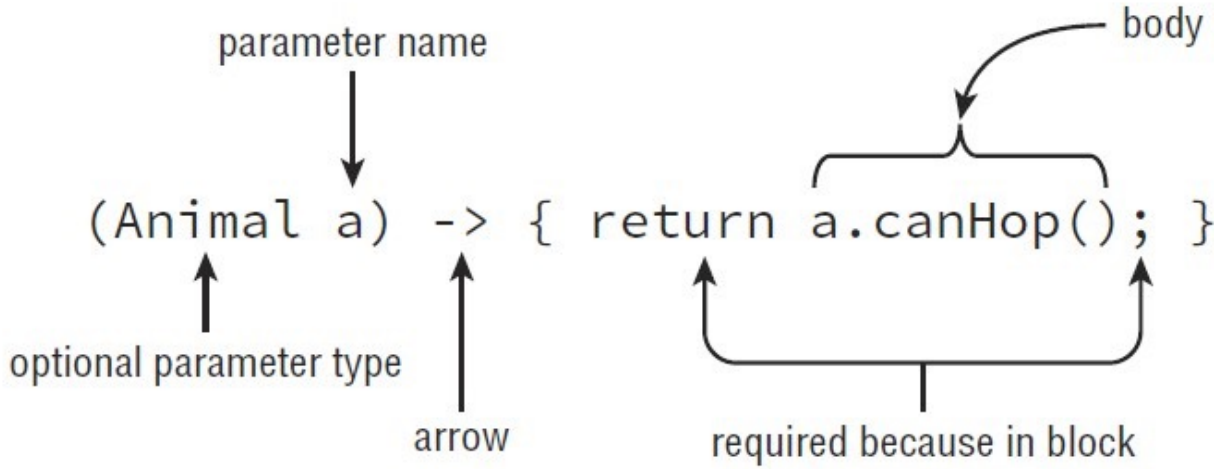
Lambda expressions

- 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



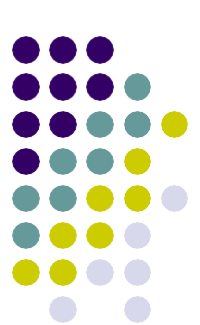
Let's see some lambdas

```
() -> { }

() -> "Trex"

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

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

a -> a.startsWith("test")

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

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

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

Java's functional 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

- 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);
```

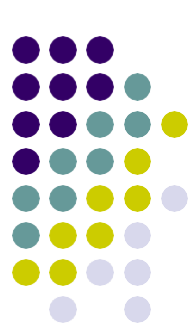
- **.stream() and .parallelStream()**

```
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

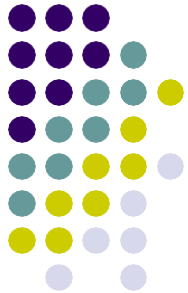
- .allMatch() , anyMatch() and noneMatch(Predicate<T>)

```
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
```

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

Putting together the pipeline

Get first two names of four letters sorted alphabetically

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

Putting together the pipeline

Get first two names of four letters sorted alphabetically

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

Putting together the pipeline

Get first two names of four letters sorted alphabetically

```
Arrays.asList("Toby", "Anna", "Leroy", "Alex", "Jamie")
        .stream()
        .filter(n -> n.length() == 4)
```

Putting together the pipeline

Get first two names of four letters sorted alphabetically

```
Arrays.asList("Toby", "Anna", "Leroy", "Alex", "Jamie")
        .stream()
        .filter(n -> n.length() == 4)
        .sorted()
```

Putting together the pipeline

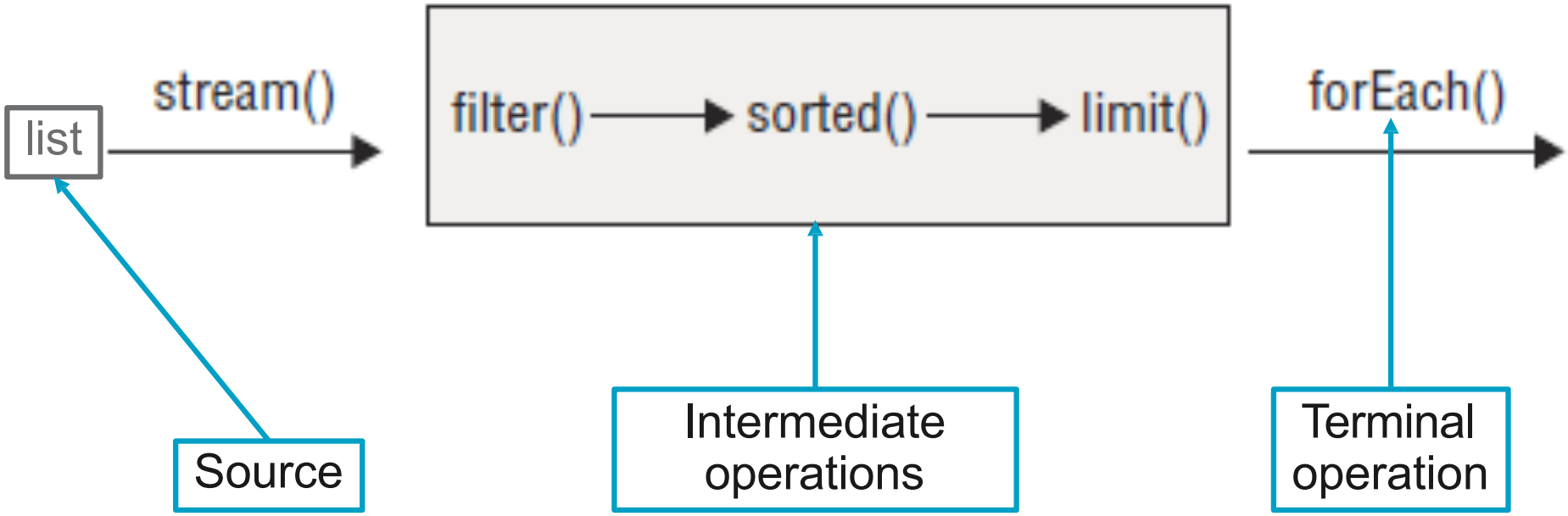
Get first two names of four letters sorted alphabetically

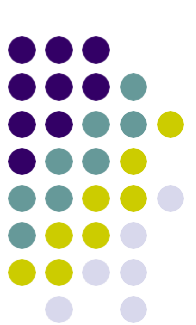
```
Arrays.asList("Toby", "Anna", "Leroy", "Alex", "Jamie")
        .stream()
        .filter(n -> n.length() == 4)
        .sorted()
        .limit(2)
        .forEach(System.out::println);
```

Putting together the pipeline

Get first two names of four letters sorted alphabetically

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





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

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);
```

- 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());
```

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]
```

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]}
```

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}
```

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}
```

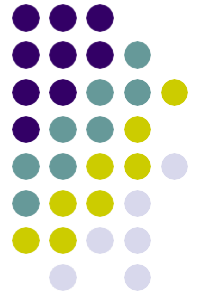
NIO 2 additions

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...
}
```

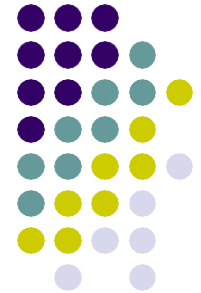
NIO 2 additions



- .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...
}
```

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...
}
```

Summary

- Lambdas and method references syntax

- Creating, processing and consuming streams

Questions

Bibliography

- https://docs.oracle.com/javase/tutorial/java/concepts
- **Java 8 in Action**, by Alan Mycroft, Mario Fusco.
- **Thinking in Java 4th Edition**, by Bruce Eckel
- http://beginnersbook.com/2013/04/oops-concepts/
- https://introcs.cs.princeton.edu/java/home/
- https://ocw.mit.edu/