Kata Retrospective in Java 11

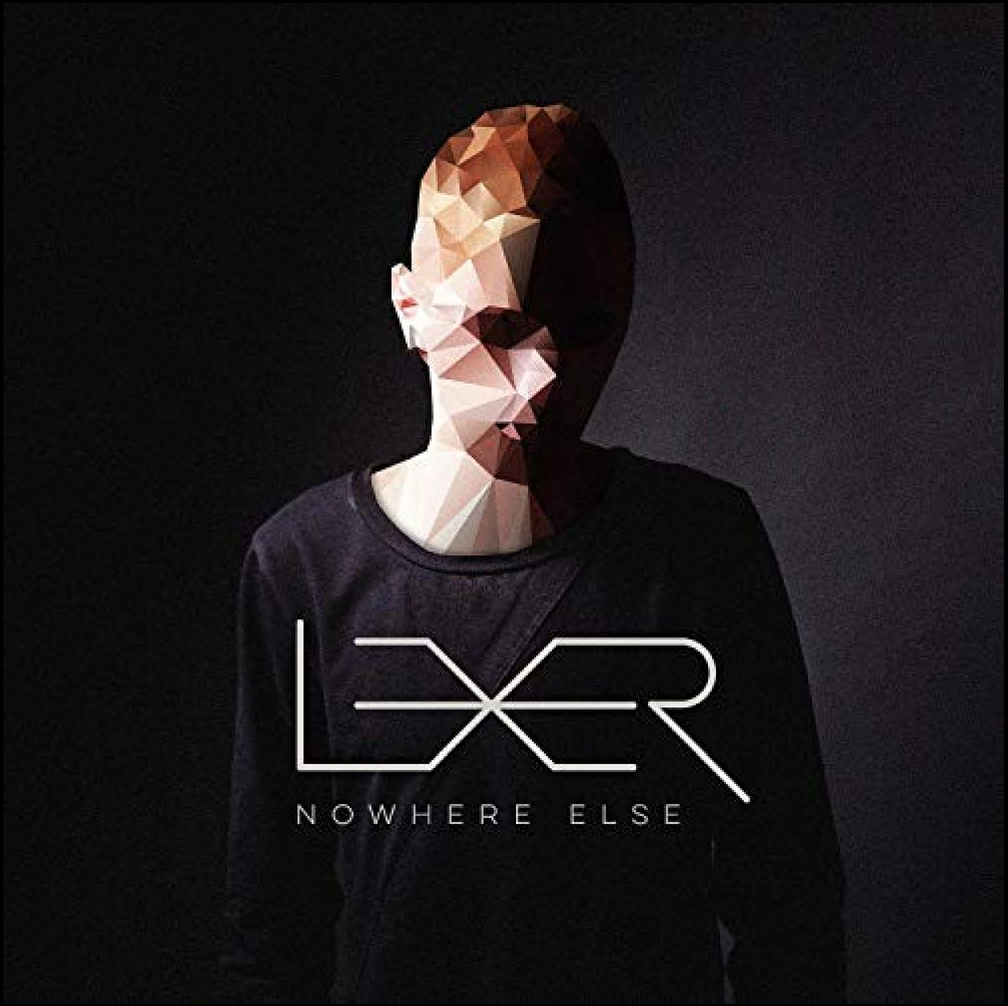
Rémi Forax – Devoxx 2019

Me, Myself and I

UPEM
UNIVERSITÉ PARIS-EST
MARNE-LA-VALLÉE

Esipe
INGÉNIEURS PARIS EST

Java Champion,
OpenJDK amber & valhalla
ex: jigsaw, lambda, invokedynamic


LEXER
NOWHERE ELSE

Goal

```
var lexer = Lexer.create()
      .with("([0-9]+)",              Integer::parseInt)
      .with("([0-9]+\\.[0-9]*)", Double::parseDouble)
      .with("([a-zA-Z]+)",           Function.identity());

lexer.tryParse("foo")    // the String foo
lexer.tryParse("12.3")   // the double 12.3
lexer.tryParse("200")    // the int 200
```

Functional form of ...

An if/else cascade

```java
static Object parse(String text) {
  if (recognize(text, "([0-9]+)")) {
    return Integer.parseInt(text);
  }
  if (recognize(text, "([0-9]+\\.[0-9]*)")) {
    return Double.parseDouble(text);
  }
  if (recognize(text, "([a-zA-Z]+)")) {
    return text;
  }
  throw new ...
}
```

Is it a function ?

a function that takes one input and has two outputs

```
if (recognize(text, "([0-9]+)")) {
  return Integer.parseInt(text);
} else {
  ...
}
```

so it’s a not function ??

The railroad switch pattern

term coined by Scott Wlaschin (fsharpforfunandprofit.com)

```
?? tryParse(String text)
```

```
if (recognize(text, "([0-9]+)")) {
  return Integer.parseInt(text);
} else {
  ...
}
```

The Option monad to the rescue

Monad

- Superposition of states using a common interface

java.util.Optional

- Optional.of(…)
- Optional.empty()

The railroad switch pattern

The result is either a value or nothing

```
Optional<V> tryParse(String text)
```

```
if (recognize(text, "([0-9]+)")) {
  return Integer.parseInt(text);
} else {
  ...
}
```

This talk !

3 levels !

- Any behavioral/creational patterns in Java
- Railroad Switch Pattern
- Lexer implementation / Java 11

The railroad switch kata in Java 11

Rémi Forax – Devoxx 2019

Question 1

Write a Lexer that doesn’t recognize any pattern

```
var lexer = Lexer.create();
lexer.tryParse("a_keyword").isEmpty()  // true
```

Demo Question 1

A Lexer

```
@FunctionalInterface
interface Lexer<V> {
  Optional<V> tryParse(String text);
}
```

A simple lambda is enough

```
static <T> Lexer<T> create() {
  return text -> {
    requireNonNull(text);
    return Optional.empty();
  };
}
```

Question 2

Write a Lexer that recognizes a pattern

```
var lexer = Lexer.from(Pattern.compile("([a-z]o)o"));
lexer.tryParse("zoo").orElseThrow()  // zo
lexer.tryParse("bar").isEmpty()  // true


var lexer2 = Lexer.from("([a-z]o)o");
```

Demo Question 2

Lexer.from()

```
private static void requireOneCaptureGroup(Pattern pattern) {
  if (pattern.matcher("").groupCount() != 1) {
    throw new IAE(pattern + " has not one captured group");
  }
}

static Lexer<String> from(Pattern pattern) {
  requireOneCaptureGroup(pattern);
  return text -> Optional.of(pattern.matcher(text))
                         .filter(Matcher::matches)
                         .map(matcher -> matcher.group(1));
}

static Lexer<String> from(String regex) {
  return from(compile(regex));
}
```

Question 3

Write a method map() that transforms the result of a Lexer

```
var lexer =
  Lexer.from("([0-9]+)").map(Integer::parseInt);

lexer.tryParse(404).orElseThrow()  // 404
```

Demo Question 3

Lexer.map()

```
 default <U> Lexer<U> map(
     Function<? super T, ? extends U> mapper) {

  requireNonNull(mapper);
  return text -> tryParse(text).map(mapper);
}
```

Question 4

Write a method or() that combines two Lexers

```
var lexer1 = Lexer.from("([0-9]+)")
      .map(Integer::parseInt);
var lexer2 = Lexer.from("([0-9]+\\.[0-9]*)")
      .map(Double::parseDouble);
var lexer3 = lexer1.or(lexer2);
```

Demo Question 4

Lexer.or()

```java
default Lexer<T> or(Lexer<? extends T> lexer) {
  requireNonNull(lexer);
  return text -> tryParse(text)
    .or(() -> lexer.tryParse(text));
}
```

Lexer.with()

Add a method with() that combines a pattern and a transformation

```
var lexer =
  Lexer.create()
    .with("([0-9]+)", Integer::parseInt)
    .with("([0-9]+\\.[0-9]*)", Double::parseDouble);
```

Demo Question 5

Lexer.with()

```
 default Lexer<T> with(String regex,
   Function<? super String, ? extends T> parser) {

  return or(from(regex).map(parser));
}
```

Lexer.from(List, List)

Write an overload of from() that takes couples of pattern/action

```
var lexer =
  Lexer.from(
    List.of("([0-9]+)",         "([0-9]+\\.[0-9]*)"),
    List.of(Integer::parseInt, Double::parseDouble));
```

Demo Question 6

Lexer.from(List, List)

```java
static class FastLexer<T> implements Lexer<T> {
  private final List<String> regexes;
  private final List<Function<? super String, ? extends T>> mappers;
  ...

  @Override
  public Optional<T> tryParse(String text) {
    requireNonNull(text);
    if (regexes.isEmpty()) {
      return Optional.empty();
    }
    var matcher = compile(join("|", regexes)).matcher(text);
    if (!matcher.matches()) {
      return Optional.empty();
    }
    for(var i = 0; i < matcher.groupCount(); i++) {
      var group = matcher.group(i + 1);
      if (group != null) {
        return Optional.of(group).map(mappers.get(i));
      }
    }
    return Optional.empty();
```

Lexer.from(List, List)

```
static <T> Lexer<T> from(List<String> regexes,
     List<? extends Function<? super String, ? extends T>> mappers) {

  if (regexes.size() != mappers.size()) {  // implicit nullchecks
    throw new IllegalArgumentException("lists with different sizes");
  }
  regexes.forEach(regex -> requireOneCaptureGroup(compile(regex)));

  return new FastLexer<>(List.copyOf(regexes), List.copyOf(mappers));
}
```

Lexer.from(List, List) with a Stream

```java
return Optional.of(regexes)
  .filter(not(List::isEmpty))
  .map(regexes -> compile(join("|", regexes)).matcher(text))
  .filter(Matcher::matches)
  .flatMap(matcher -> range(0, matcher.groupCount())
              .boxed()
              .flatMap(i -> ofNullable(matcher.group(i + 1))
                              .map(mappers.get(i))
                              .stream())
              .findFirst());
```

Lexer.from(List, List).map|or()

Provide a better map() implementation for the FastLexer

```
var lexer = Lexer.<Integer>from(
     List.of("([0-9]+)"), List.of(Integer::parseInt));
lexer = lexer.map(x -> x * 2);

var lexer2 = Lexer.from(
     List.of("..."), List.of(Double::parseDouble));
lexer2.or(lexer);
```

Demo Question 7

Lexer.from(List, List).map()

```
@Override
public <U> Lexer<U> map(
  Function<? super T, ? extends U> mapper) {

  return new FastLexer<>(regexes,
    mappers.stream()
      .map(mapper::compose)
      .collect(toUnmodifiableList()));
}
```

Lexer.from(List, List).or()

```java
@Override
public Lexer<T> or(Lexer<? extends T> lexer) {
  if (lexer instanceof FastLexer) {
    var fastLexer = (FastLexer<? extends T>)lexer;
    return new FastLexer<>(
      concat(regexes, fastLexer.regexes),
      concat(mappers, fastLexer.mappers));
  }
  return Lexer.super.or(lexer);
}

private static <T> List<T> concat(List<? extends T> l1, List<? extends T> l2) {
  return Stream.of(l1, l2).flatMap(List::stream).collect(toUnmodifiableList());
}
```

Java 8 / 11

More functional (language)

Fake structural type + inference

- Functional interface/lambda

Less names !

- var + anonymous class

More functional (API)

- Immutable collections
  - List/Set/Map.of()
  - List/Set/Map.copyOf()
  - Collectors.toUnmodifiableList/Set/Map()
- Monads
  - Optional, Stream

but ...

- Too many wildcards
  - JEP 300: Augement Use-Site Variance with Declration-Site Defaults

- Generics are still not reified
  - Reified generics for primitives: valhalla

Railroad Switch Pattern

Railroad Switch Pattern

```java
@FunctionInterface
public interface Switch<T, R> {
  Optional<R> apply(T t);

  static <T, R> Switch<T, R> lift(Function<? super T, ? extends R> function) {
    return t -> Optional.of(function.apply(t));
  }

  default Switch<T, R> filter(Predicate<? super R> filter) {
    return t -> apply(t).filter(filter);
  }

  default <V> Switch<T, V> map(Function<? super R, ? extends V> mapper) {
    return t -> apply(t).map(mapper);
  }

  default Switch<T, R> or(Switch<? super T, ? extends R> switz) {
    return t -> apply(t).or(() -> switz.apply(t));
  }
```

FizzBuzz

Imperative implementation

```
for (int i = 1; i <= 100; i++) {
  if ((i % 15) == 0)
    System.out.println("fizzbuzz");
  else if ((i % 3) == 0)
    System.out.println("fizz");
  else if ((i % 5) == 0)
    System.out.println("buzz");
  else
    System.out.println(i);
}
```

Functional Implementation

```
 public static void main(String[] args) {
   var fizz = lift((Integer i) -> i)
     .filter(i -> i % 3 == 0).map(i -> "fizz");
   var buzz = lift((Integer i) -> i)
       .filter(i -> i % 5 == 0).map(i -> "buzz");
   var fizzbuzz = lift((Integer i) -> i)
       .filter(i -> i % 15 == 0).map(i -> "fizzbuzz");
   var all = fizzbuzz.or(buzz).or(fizz);

   rangeClosed(1, 100)
     .mapToObj(i -> all.apply(i).orElseGet(() -> "" + i))
     .forEach(System.out::println);
}
```

Questions ?