

Java vs Kotlin vs Scala

Functional programming showdown

Tomche Delev

31 March, JavaSkop '18

Modifying mutable variables

```
static String wordCount(String fileName) throws IOException {
    int lines = 0;
    int words = 0;
    int characters = 0;
    try (BufferedReader bufferedReader = new BufferedReader(new FileReader(fileName))) {
        String line;
        while ((line = bufferedReader.readLine()) != null) {
            lines++;
            String[] wordParts = line.split("\\s+");
            words += wordParts.length;
            characters += line.length() + 1;
        }
    }
    return String.format("%d %d %d", lines, words, characters);
}
```

Using assignments

```
static String wordCount(String fileName) throws IOException {
    int lines = 0;
    int words = 0;
    int characters = 0;
    try (BufferedReader bufferedReader = new BufferedReader(new FileReader(fileName))) {
        String line;
        while ((line = bufferedReader.readLine()) != null) {
            lines++;
            String[] wordParts = line.split("\\s+");
            words += wordParts.length;
            characters += line.length() + 1;
        }
    }
    return String.format("%d %d %d", lines, words, characters);
}
```

Control structures *(if-then-else, loops, break, continue, return)*

```
static String wordCount(String fileName) throws IOException {
    int lines = 0;
    int words = 0;
    int characters = 0;
    try (BufferedReader bufferedReader = new BufferedReader(new FileReader(fileName))) {
        String line;
        while ((line = bufferedReader.readLine()) != null) {
            lines++;
            String[] wordParts = line.split("\\s+");
            words += wordParts.length;
            characters += line.length() + 1;
        }
    }
    return String.format("%d %d %d", lines, words, characters);
}
```

What is Functional Programming?

Programming without mutable variables, assignments, loops and other imperative control structure

What is Functional Programming?

Focusing on the **functions** as **values** that can be:

- Produced
- Consumed
- Composed

All this becomes easier in a functional language

"A language that doesn't affect the way you think about programming is not worth knowing." - **Alan J. Perlis**

Why Functional Programming?

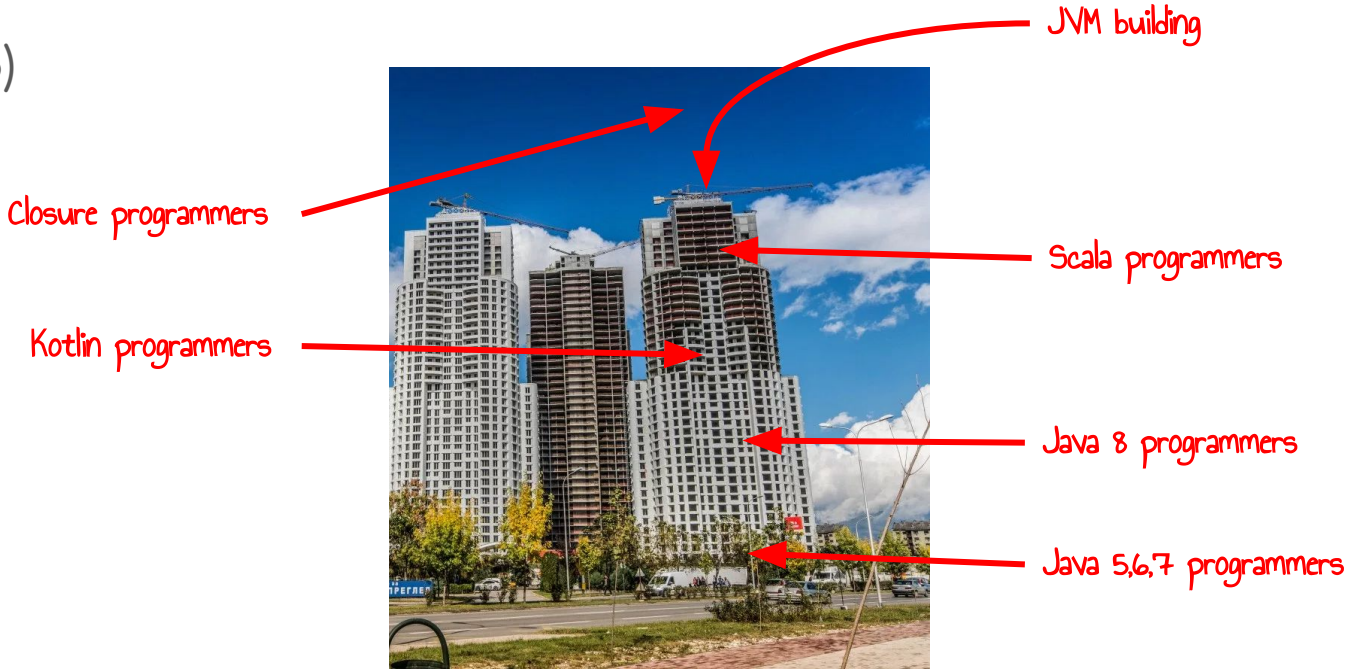
- Because it is programming for adults
- Pure functions and immutability
- Highly composable
- Lazy evaluation
- It shifts your perspective and it's more FUN
- Simple?

"Simplicity does not precede complexity, but follows it." - **Alan Perlis**

How we do FP?

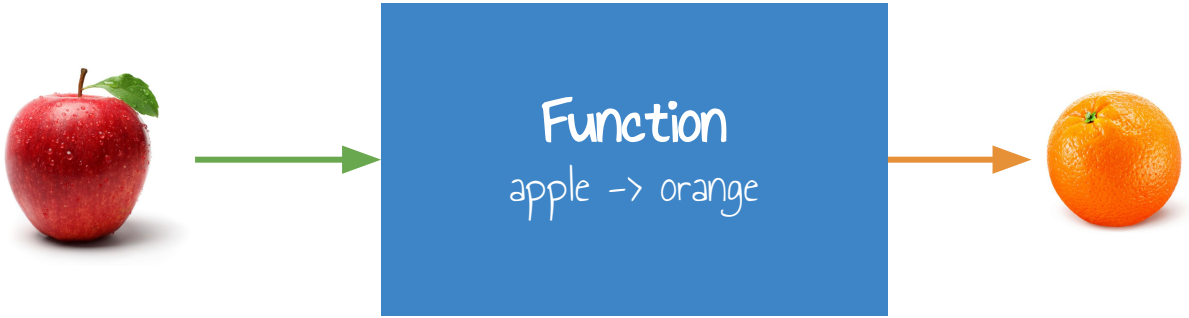
We need functional programming language?

- Java (>=8)
- Kotlin
- Scala



https://www.slideshare.net/ScottWlaschin/fp-patterns-ndc-london2014

Functions are things



(A) -> B

"Sometimes, the elegant implementation is just a function. Not a method. Not a class. Not a framework. Just a function." - **John Carmack**

Functions are things

```
int sum(int a, int b) {
    return a + b;
}
```

Not a thing

```
BiFunction<Integer, Integer, Integer> sumF = Functions::sum;
```

This is a THING

```
BiFunction<Integer, Integer, Integer>
```

(A, A) -> A

Functions are things

```
fun sum(a: Int, b: Int): Int {
  return a + b
}
```

```
fun sum(a: Int, b: Int) = a + b
```

Or without all the clutter

```
val sumF = ::sum
```

Method reference also works

```
(Int, Int) -> Int
```

Functions are things

```
def sum(a: Int, b: Int): Int = {
  a + b
}
```

```
def sum(a: Int, b: Int) = a + b
```

```
val sumF: (Int, Int) => Int = sum
```

```
(Int, Int) => Int
```

No method reference in Scala, but you can just pass the function name

Higher-order functions

```
Function<String, String> msgFun(int a, int b,
BiFunction<Integer, Integer, Integer> bf) {
    return msg -> msg + ": " + bf.apply(a, b);
}
```

Accepts function as argument

Or (and) returns function as result

(A, A, (A, A) -> A) -> (B) -> B

(Int, Int, (Int, Int) -> Int) ->
(String) -> String

Higher-order functions

```
fun msgFun(a: Int, b: Int, f: (Int, Int) -> Int): (String) -> String =
    { msg: String -> "$msg:${f(a, b)}" }
```

Kotlin has string interpolation

```
val resultFun = msgFun(5, 10, { a, b -> a + b })
resultFun("The sum is: ") // "The sum is: 15"
```

(A, A, (A, A) -> A) -> (B) -> B

(Int, Int, (Int, Int) -> Int) ->
(String) -> String

Higher-order functions

```
def mulFun(a: Int, b: Int, f: (Int, Int) => Int): Int => Int =
    x => x * f(a, b)
```

Partial application

Bake in on of the arguments

```
int sum5(int a) {
    return sum(5, a);
}
```

(A, A) -> A => (A) -> A

```
Function<Integer, Integer> sum5Partial = a -> sumF.apply(5, a);
```

Partially apply A on any function (A, B) -> C and convert to (B) -> C

```
<A, B, C> Function<B, C> partial(A a, BiFunction<A, B, C> f) {
    return b -> f.apply(a, b);
}
```

(A, B) -> C => (B) -> C

```
Function<Integer, Integer> sum10Partial = partial(10, sumF);
```

Partial application

```
fun sum5(a: Int): Int {
   return sum(5, a)
}

val sum5Partial = { a: Int -> sumF(5, a) }


fun <A, B, C> partial(a: A, f: (A, B) -> C): (B) -> C =
    { b -> f(a, b) }

val sum10Partial = partial(10, sumF)
```

Partial application

```
def sum5(a: Int): Int= {
  sum(5, a)
}

val sum5Partial: (Int) => Int = a => sumF(5, a)


def partial[A,B,C](a: A, f: (A, B) => C): (B) => C =
    b => f(a, b)

val sum10Partial = partial(10, sumF)
```

Curring

Transform any function with multiple arguments into new function with single argument

```
Function<Integer, Integer> sumA(int a) {
    return b -> sum(a, b);
}
```

(A, A) -> A => (A) -> (A) -> A

```
<A, B, C> Function<A, Function<B, C>> curry(BiFunction<A, B, C> f) {
    return a -> b -> f.apply(a, b);
}
```

Curry any function (A, B) -> C into (A) -> (B) -> C

```
Function<Integer, Function<Integer, Integer>> sumACurried =
    curry(sumF);
```

Curring

```
val sumA = { a: Int -> { b: Int -> sumF(a, b) } }
```

```
fun <A, B, C> curry(f: (A, B) -> C): (A) -> (B) -> C =
        { a -> { b -> f(a, b) } }
```

```
fun <A, B, C> ((A, B) -> C).curried(): (A) -> (B) -> C =
        curry(this)
```

Or using extension functions in Kotlin

```
val sumCurried = curry(sumF)
```

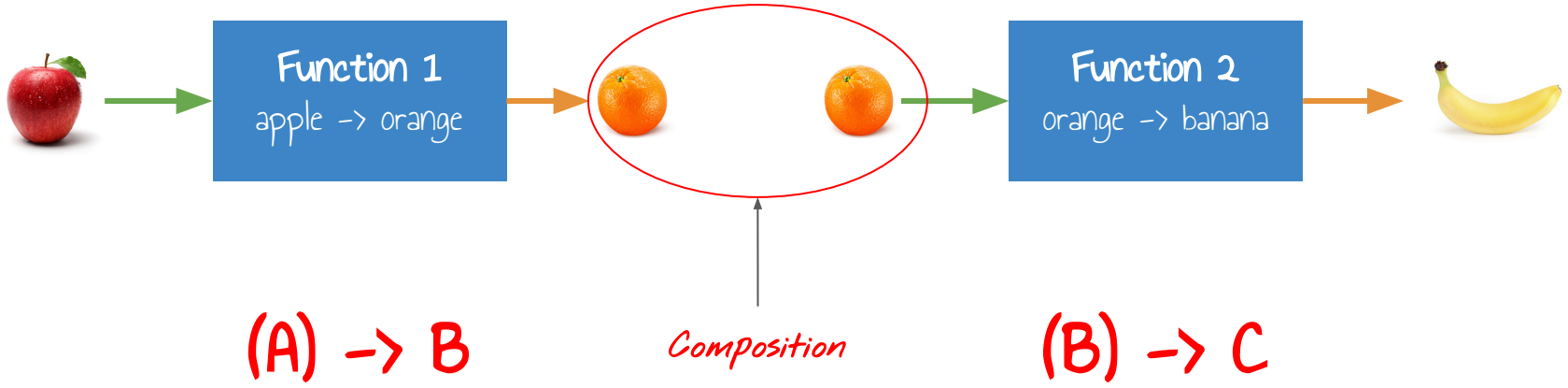
Curring

```
val sumA = (a: Int) => (b: Int) => sumF(a, b)
```

```
def curry[A, B, C](f: (A, B) => C): (A) => (B) => C =
    a => b => f(a, b)
```

```
val sumCurried = curry(sumF)
```

Composition



Composition

:(Not smoothie!

Composed function
apple -> banana

(A) -> C

We have no idea if it's composed of other functions

Composition

```
String result(int a) {
    return String.format("Result is: %d", a);
}

String resultSum(int a, int b) {
    return result(sum(a, b));
}
```

We can (usually) compose by passing the result

```
BiFunction<Integer, Integer, String> resultComposed =
     sumF.andThen(Functions::result);
```

Composition

```
<A, B, C> Function<A, C> compose(Function<B, C> f, Function<A, B> g) {
    return a -> f.apply(g.apply(a));
    // return g.andThen(f);
}
```

Compose any two functions (B) -> C and (A) -> B into new function (A) -> C

```
Function<Integer, String> square =
    compose(Functions::result, a -> a * a);

square(5) // "Result is: 25"
```

Composition

```
fun result(a: Int) = "Result is: $a"

fun resultSum(a: Int, b: Int): String {
    return result(sum(a, b))
}

fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C {
    return { a -> f(g(a)) }
}

val square = compose(::result, { a: Int -> a * a })
```

Composition

```
def result(a: Int) = s"Result is: $a"

def resultSum(a: Int, b: Int): String = {
 result(sum(a, b))
}

def compose[A, B, C](f: (B) => C, g: (A) => B): (A) => C = {
 a => f(g(a))
}

val square = compose(result, (a: Int) => a * a)
```

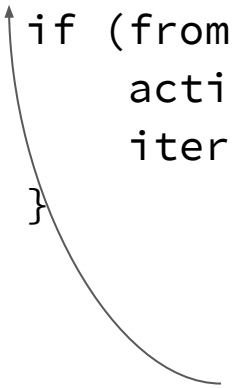
Iteration

```
void iterate(int from, int to, Consumer<Integer> action) {
    if (from < to) {
        action.accept(from);
        iterate(from + 1, to, action);
    }
}
```

What will happen for ranges in many thousands?

Iteration

```
tailrec fun iterate(from: Int, to: Int, action: (Int) -> Unit) {
    if (from < to) {
        action(from)
        iterate(from + 1, to, action)
    }
}
```

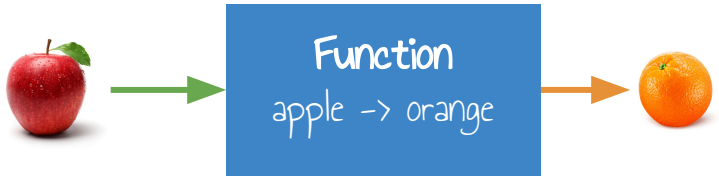


Will convert this function into TAIL RECURSIVE function

Iteration

```
@tailrec
def iterate(from: Int, to: Int, action: (Int) => Unit) {
 if (from < to) {
   action(from)
   iterate(from + 1, to, action)
 }
}
```

Total functions

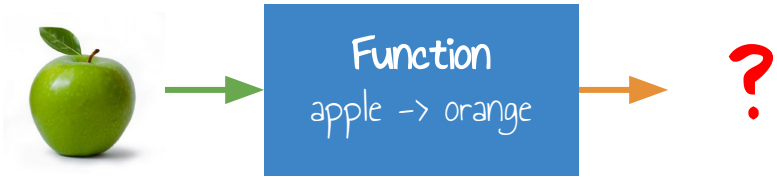


*For every **apple** there is an **orange***

```
String intToString(int i) {
    return String.valueOf(i);
}
```

(Int) -> String

Total functions



What we do when we can't find an orange for some apple?

```
int div(int number, int n) {
    if(n == 0) ???
    else return number / n;
}
```

Exceptions?

"So much complexity in software comes from trying to make one thing do two things."
– Ryan Singer

Total functions

```
sealed class Option<out T>


object None : Option<Nothing>()

data class Some<out T>(val value: T) : Option<T>()
```

Absence of value

Presence of value

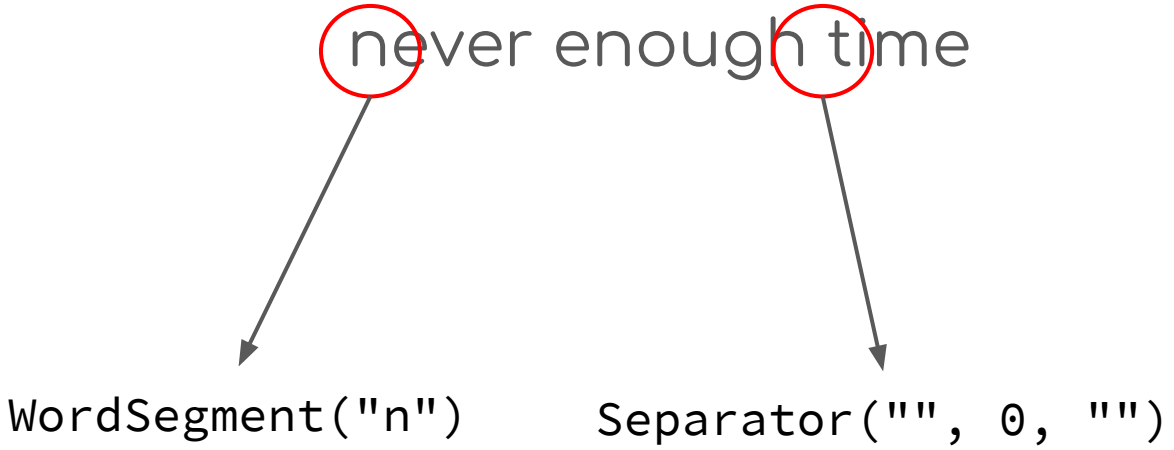
```
fun div(number: Int, div: Int) {
    if(div == 0) None
    else return Some(number / div)
}
```

Word count

"There's never enough time to design the right solution, but somehow always an infinite amount of time for supporting the wrong solution."

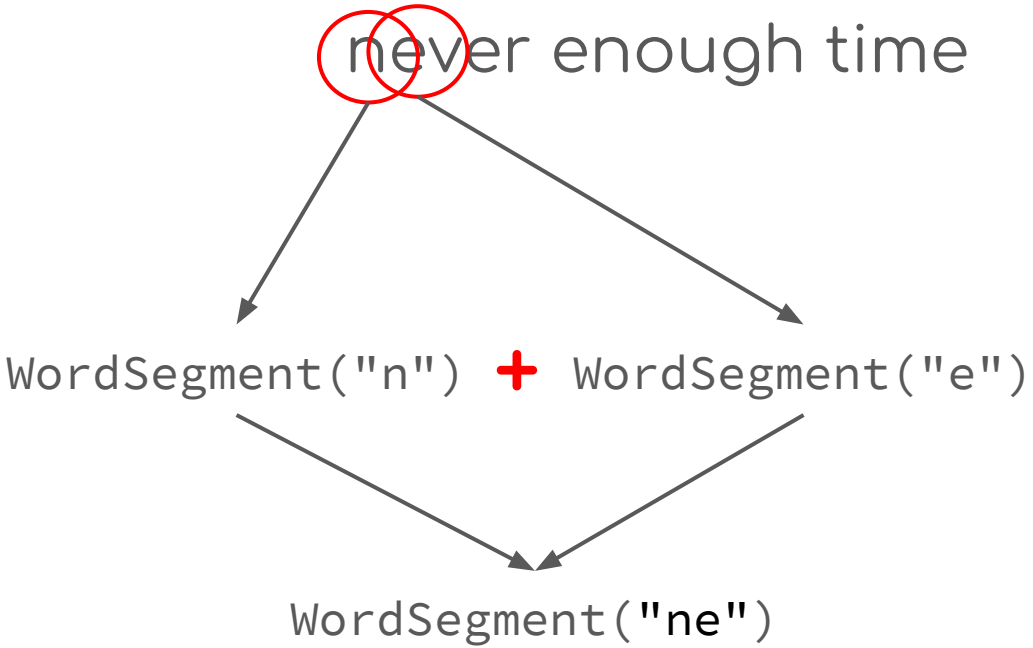
We want to count the number of words in a sentence

Word count

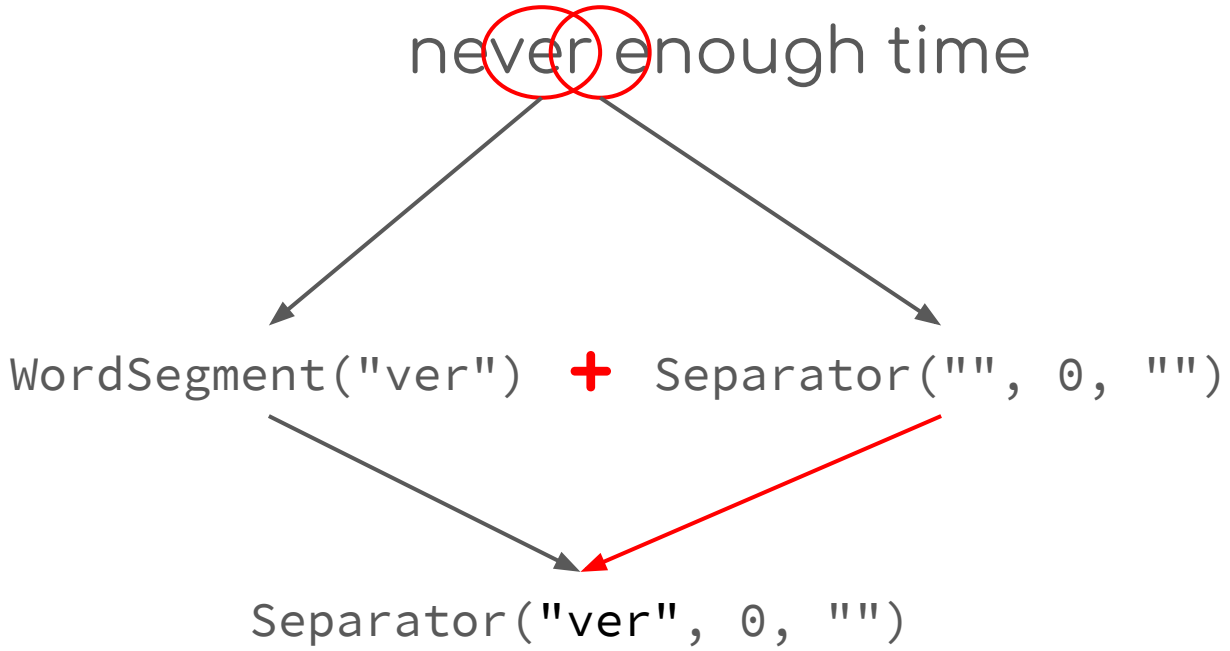


we map each character into some type

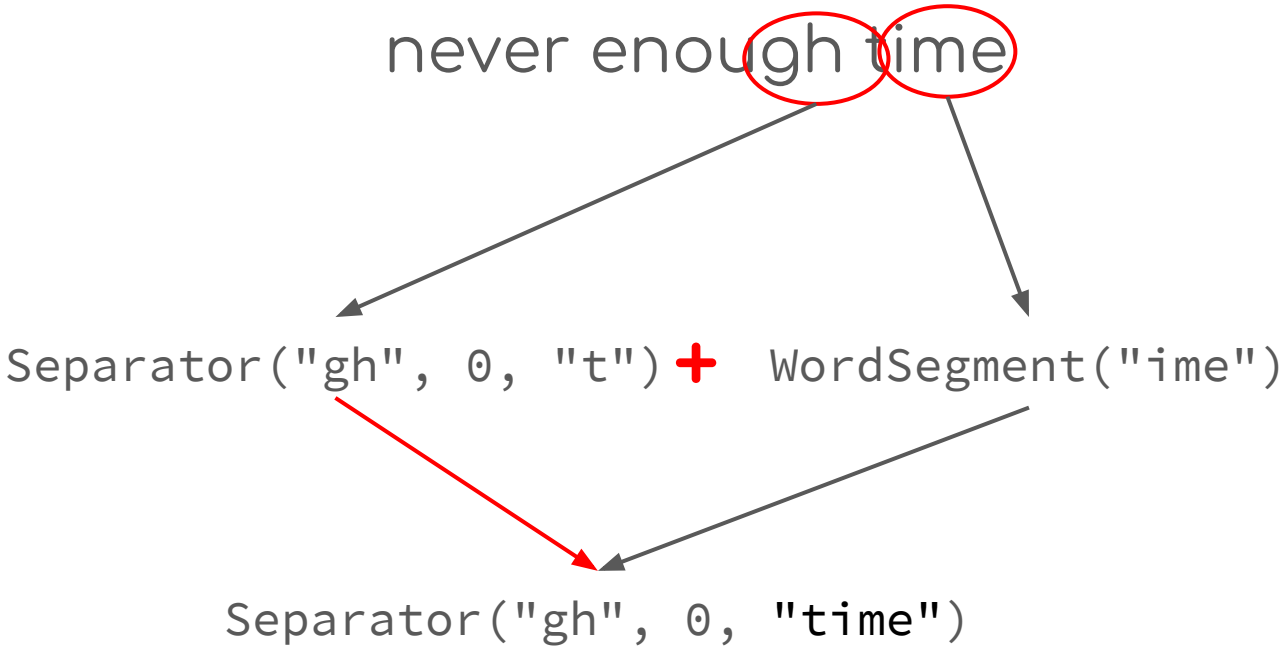
Word count


never enough time
WordSegment("n") + WordSegment("e")
WordSegment("ne")

Word count

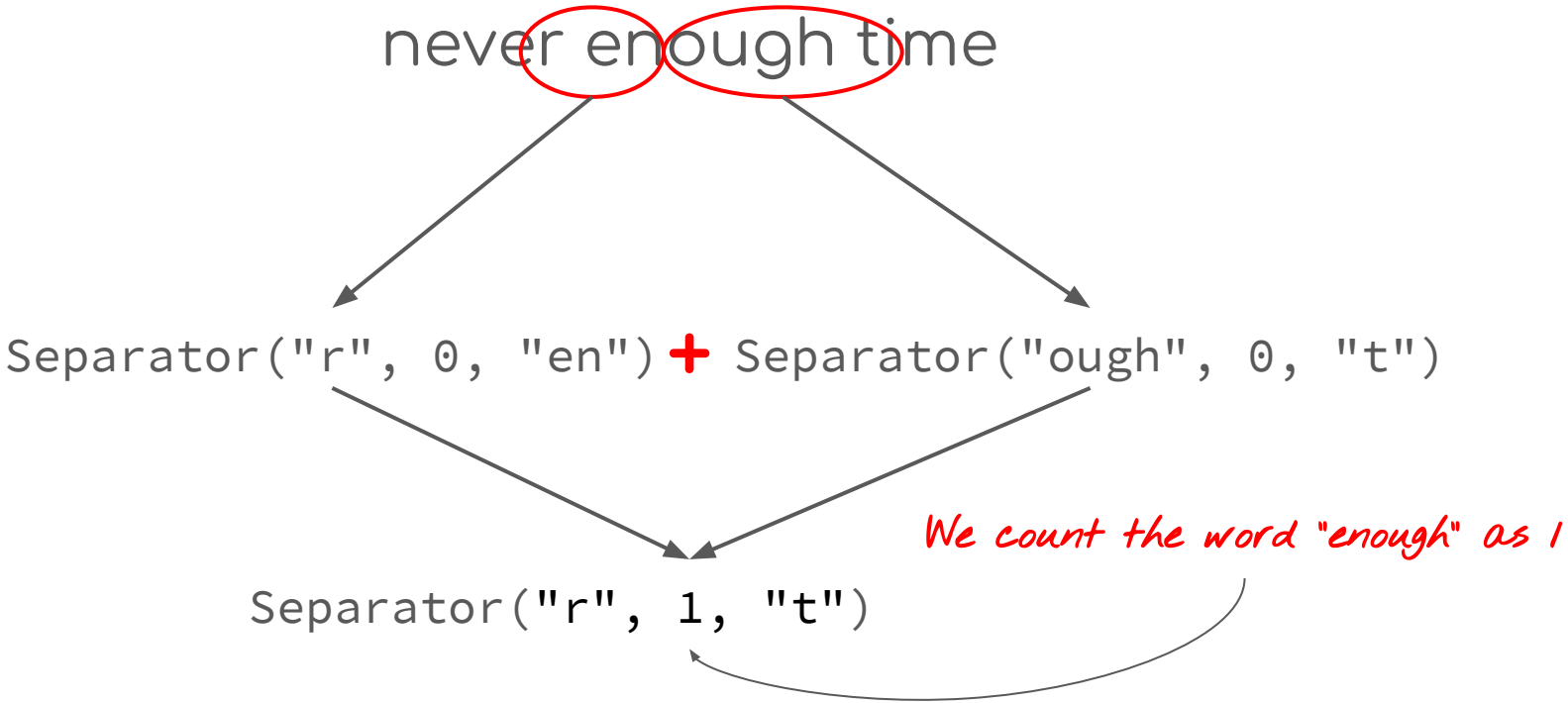

never enough time
WordSegment("ver") + Separator("", 0, "")
Separator("ver", 0, "")

Word count


never enough time
Separator("gh", 0, "t")
+
WordSegment("ime")
Separator("gh", 0, "time")

Word count

(A + B) + C = A + (B + C)



Word count

```
sealed class WordCount

data class WordSegment(val chars: String) : WordCount()

data class Separator(
    val left: String,
    val words: Int,
    val right: String
): WordCount()
```

Word count

```
fun wc(c: Char): WordCount =
   if (c.isWhitespace())
        Separator("", 0, "")
   else
        WordSegment(c.toString())
```

Word count

```
fun combine(a: WordCount, b: WordCount) = when (a) {
  is WordSegment -> when (b) {
    is WordSegment -> WordSegment(a.chars + b.chars)
  ...
```

Word count

```
fun combine(a: WordCount, b: WordCount) = when (a) {
  is WordSegment -> when (b) {
    is WordSegment -> WordSegment(a.chars + b.chars)
    is Separator -> Separator(a.chars + b.left, b.words, b.right)
  }
  ...
```

Word count

```
fun combine(a: WordCount, b: WordCount) = when (a) {
  is WordSegment -> when (b) {
    is WordSegment -> WordSegment(a.chars + b.chars)
    is Separator -> Separator(a.chars + b.left, b.words, b.right)
  }
  is Separator -> when (b) {
    is WordSegment -> Separator(a.left, a.words, a.right + b.chars)
  ...
```

Word count

```
fun combine(a: WordCount, b: WordCount) = when (a) {
  is WordSegment -> when (b) {
    is WordSegment -> WordSegment(a.chars + b.chars)
    is Separator -> Separator(a.chars + b.left, b.words, b.right)
  }
  is Separator -> when (b) {
    is WordSegment -> Separator(a.left, a.words, a.right + b.chars)
    is Separator -> Separator(a.left, a.words + b.words +
        if ((a.right + b.left).isNotEmpty()) 1 else 0,
        b.right)
            }
  }
}
```

Word count

```
fun count(text: String): Int {
    val result = text.chars()
        .mapToObj { it.toChar() }
        .map(::wc)
        .reduce(wcCombiner.unit(), wcCombiner::combine)

    fun unstub(s: String) = min(s.length, 1)

    return when (result) {
        is WordSegment -> unstub(result.chars)
        is Separator ->
           unstub(result.left) + result.words + unstub(result.right)
}
```

Is there something special about this?

```
public interface Combiner<A> {
    A combine(A left, A right); // mappend

    A identity(); // zero // mempty
}
```

combine(combine(a, b), c) == combine(a, combine(b, c))

combine(a, identity()) == a

It is referred as a "Monoid"

Or this?

```
public interface Option<A> {

    default <B> Option<B> map(Function<A, B> f) {
        return this.flatMap(a -> unit(f.apply(a)));
    }

    <B> Option<B> flatMap(Function<A, Option<B>> f);

    static <A> Option<A> unit(A value) {
        return new Some<>(value);
    }
}
```

Is there a common pattern?

```
class Optional<T> {
    Optional<U> flatMap(Function<? super T, Optional<U>> mapper)

    Optional<T> of(T value);

    Optional<T> empty()
}
```

Is there a common pattern?

```
interface Stream<T> {
    <R> Stream<R> flatMap(Function<? super T, ? extends Stream<?
extends R>> mapper);

    Stream<T> of(T t);

    Stream<T> empty();
}
```

It is referred as a "Monad"

But what is a Monad?

Monad it's just a monoid in the category of endofunctors

References

- https://github.com/tdelev/fp-java-kotlin-scala
- https://www.coursera.org/learn/progfun1
- Functional Programming in Scala, Paul Chiusano and Runar Bjarnason
https://www.amazon.com/Functional-Programming-Scala-Paul-Chiusano/dp/1617290653
- https://www.slideshare.net/ScottWlaschin/fp-patterns-ndc-london2014

Thank You for your attention!

The most precious thing you can ask from others is not their money nor their time; it's their attention.

https://twitter.com/venkat_s/status/972906986558824448

This work is supported by

sorsix

Questions?