



 @AndrzejWasowski

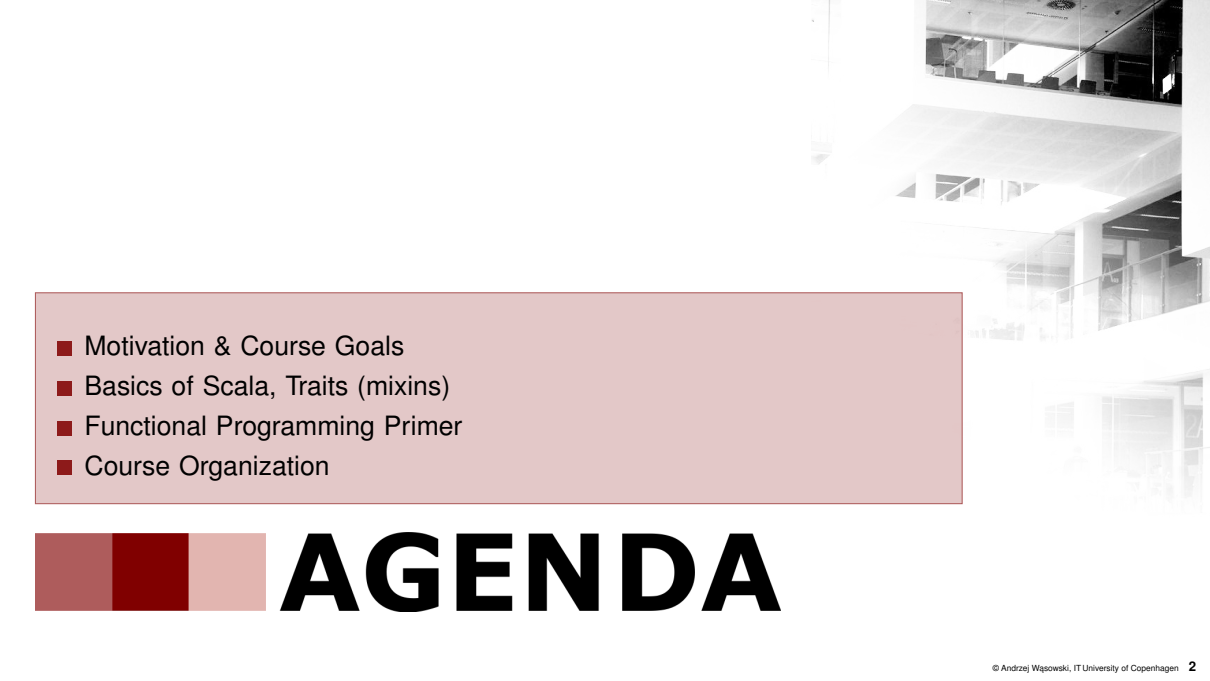
Andrzej Wąsowski
Florian Biermann

Advanced Programming

1. Introduction to Functional Programming

IT UNIVERSITY OF COPENHAGEN

S SOFTWARE
Q QUALITY
R RESEARCH

- 
- Motivation & Course Goals
 - Basics of Scala, Traits (mixins)
 - Functional Programming Primer
 - Course Organization



AGENDA

Apache Spark

A Motivating Example



- Open-source cluster-computing framework aimed at **Big Data processing**
- Compute queries on large amount of data in **distributed storage**
- **Simple interface**: like local data structures
- **Powerful semantics**: distribution and parallelization
- Originated in 2009 at UC Berkeley, run by **Apache Foundation**
- **600 devs from 200** companies contribute to spark
- **Implemented in Scala**, interfaced to Java, Python, and R
- Key reason of popularity of Scala for Big Data
- Much faster than Hadoop's MapReduce due to heavy use of in-memory operations
- **Some Users**: NBCUniversal, Netflix, Uber, Capital One, Baidu, Salesforce.com, ...

Count Word Frequency (File)

```
1 object StorageApp:
2
3   def main(args: Array[String]) =
4
5     val conf = SparkConf()
6       .setAppName("SimpleApp")
7       .setMaster("local[6]") // use 6 cores
8     val sc = SparkContext(conf)
9     val lines = sc
10      .textFile("/home/wasowski/opt/spark/README.md", 2)
11      .cache
12
13     val wordCounts = lines
14      .flatMap { line => line.split(" ") }
15      .map { word => (word, 1) }
16      .reduceByKey(_ + _)
17
18     println(wordCounts.collect.map(_._1).mkString)
19     sc.stop()
```



- Resilient distributed dataset (**RDD**)
- `lines` is an RDD of strings
- **Distributed** fault-tolerant processing
- L14 split each line into words, merge into RDD of words
- L15 RDD of words to RDD of pairs
- L16 merge pairs with same word, summing counters (map-reduce)
- We use **collection operations**
- Transformations (`flatMap`, `map`) build **representation of computation**.
- Transformations are **lazy**.
- Actions (`reduceByKey`) are **eager**: execute (**force**) representations
- **Pure** program, no vars&side effects
- `cache` only works if you have **referential transparency**

Things go wrong with side-effects

Side effects are officially banned in this course!

```
1 var counter = 0
2 var rdd = sc.parallelize(data)
3 // Wrong: Don't do this!!
4 rdd.foreach(x => counter += x)
```



- Line 2: we parallelize a computation
- Line 4: we sum values from an RDD incrementing a counter
- This cannot be done in a distributed way!!!
- Each node gets a **closure** containing the counter, the closure is sent to nodes.
- Each node increments a different copy of the counter!
- This is why we use functions like map and reduceByKey instead of variables

Count Word Frequency in a Real-Time Data Stream

```
1 object StreamingApp:
2
3   def main(args: Array[String]) =
4     val sparkConf = SparkConf()
5       .setAppName("StreamingApp")
6       .setMaster("local[6]")
7     // Sample every second
8     val ssc = StreamingContext(sparkConf, Seconds(1))
9     val lines = ssc.socketTextStream("localhost", 9999,
10                                     StorageLevel.MEMORY_AND_DISK_SER)
11
12     val wordCounts = lines
13       .flatMap {line => line.split(" ")}
14       .map { word => (word, 1) }
15       .reduceByKey(_ + _)
16
17     wordCounts.print // Nothing gets printed!
18     ssc.start        // The computation starts here
19     ssc.awaitTermination
```



- L12-L15: identical **algorithm**
- Because the DStream **interface** is the same as RDD's
- RDDs and Streams are **monads**
- We will understand this style of API really deeply
- `reduceByKey` needs a commutative associative operator (a **monoid**)
- `+` is a monoid on integers
- L12-L17 builds a **representation of computation**
- L18: streaming starts, before this **nothing happens**
- L17 printing every 1s until killed

Introduction to Scala I

- A rich modern OO programming language with a functional part; eager by default, statically typed
- Compiles to JVM, compatible with Java on byte code level
- Designed by prof. Martin Odersky at EPFL in Lausanne
- First official release in 2004
- **This is not a course about Scala**, we use Scala to learn concepts that apply to other languages (F#, Haskell, OCaml, Clojure, Java, C#, Python, Ruby, etc.)

Basics of Scala

A singleton class and its only instance

object creates a name space; used to build modules. Access the namespace with navigation: `MyModule.abs(42)`

```
1 object MyModule:
3   def abs(n: Int): Int =
4     if n < 0 then -n else n
6   private def formatAbs(x: Int) =
7     s"The absolute value of $x is ${abs (x)}"
9   val magic: Int = 42
10  var result: Option[Int] = None
12  def main(args: Array[String]): Unit =
13    assert (magic - 84 == magic.-(84))
14    println (formatAbs (magic-100))
```

def Defines a function (l.3)

A body **expression** (statements secondary in Scala)

Use indentation if more expressions needed.

A named **value** declaration (final, immutable). Use this a lot.

A **variable** declaration. Avoid.

Instantiation of a generic type

`None` is a singleton "constructor". Construct objects without **new**

Operators are functions, can be overloaded:
minus is `Int.-(Int) :Int`
Unary methods can be used infix: `MyModule abs -42` legal

Every value is an object

Line 6 shows an **interpolated** character string

Traits: Rich or Fat Interfaces

Scala idiom: decompose classes into traits

```
1 // A class with a final property 'name' and
2 // a unary constructor. You can still add
3 // more members like in Java in braces.
4 abstract class Animal(val name: String)

6 // concrete methods
7 trait HasLegs:
8   def run(): String = "after you!"
9   def jump(): String = "hop!"

11 // abstract method
12 trait Audible: def makeNoise(): String
13 // field
14 trait Registered: var id :Int = 0

16 // multiple traits mixed in
17 class Frog(name: String) extends
18   Animal(name) with HasLegs with Audible:
19   def makeNoise(): String = "croak!"
20   // Frog concrete, so provide makeNoise
```

```
1 // Mix directly into an object
2 val f = new Frog("Kaj") with Registered
3 // f: Frog with Registered = $anon$1@88f0bea
4 f.id = 42
5 println(s"My name is ${f.name}")
6 println("I'm running " + f.run())
7 println("I'm saying " + f.makeNoise())
```

	concrete class	abstract class	trait
multiple inheritance	—	—	+
data	+	+	+
concrete methods	+	+	+
abstract methods	—	+	+
constructor arguments	+	+	—

[Horstmann 2012, chpt. 10], [Odersky et al. 2014, chpt. 12] have more info than [Pilquist, Bjarnasson, Chiusano 2022]

Pure Functions

Def. Referentially transparent expression (e)

Expression e is RT iff replacing e by its value in programs does not change their semantics

(Java) append an element to a list

`a.add(5)` // non RT

value void; substitution is pointless; the meaning is in the references reachable from a (change over time for the same a)

(Scala) append to an immutable list

`val b = Cons(5, a)` // RT

The value is a list b , identical to a , modulo the added head element

Def. Pure function (f)

Iff every expression $f(e)$ is referentially transparent for all referentially transparent expressions e .

Otherwise **impure** or **effectful**.

In practice: **A function is pure if it does not have side effects** (writes/reads variables, files or other streams, modifies data structures in place, sets object fields, throws exceptions, halts with errors, draws on screen, draws a random number)

Pure code lists dependencies in interface, good for mocking, testable

Referential Transparency Poll

Which of the following computations are referentially transparent [in Java]?

- 1 `a = a + 42`
- 2 `a[x] == 42`
- 3 `println("42")`
- 4 `throw DivideByZero()`
- 5 `f(f(x))` if `f` is pure
- 6 `z = z + f(f(x))` if `f` is pure

Loops and Recursion

An imperative factorial

```
1 def factorial(n : Int) : Int =  
2   var result = 1  
3   for i <- 2 to n do  
4     result *= i  
5   return result
```

Loops compute with effects;
cannot be used in pure code

Tail recursive, pure factorial

```
1 def factorial(n : Int) : Int =  
2   def f(n : Int, acc : Int) : Int =  
3     if n <= 1 then acc  
4     else f(n - 1, n * acc)  
5   f(n, 1)
```

accumulator parameter

call in tail position

Call tails are automatically compiled to loops with $O(1)$ space overhead

A pure recursive factorial

```
1 def factorial(n : Int) : Int =  
2   if n <= 1 then 1  
3   else n * factorial(n - 1)
```

Example execution

factorial(5)

↪ 5 * (factorial(4))

↪ 5 * (4 * (factorial(3)))

↪ 5 * (4 * (3 * (factorial(2))))

↪ 5 * (4 * (3 * (2 * (factorial(1)))))

↪ 5 * (4 * (3 * (2 * 1)))

↪ 5 * (4 * (3 * 2))

↪ 5 * (4 * 6)

↪ 5 * 24

↪ 120

call not in tail position

Uses $O(n)$ stack space;
Technically exponential
(for this example!)

Def. Call in tail position

The caller immediately returns the value of the call

Anonymous Functions (Values, Literals)

Literals

```
val l = List(1, -2, 3)
val a = Array(-1, 2, -3)
val z = (42, false)
```

Function Literals (Anonymous Functions)

We need the same for functions

```
val negative = (x: Int) => x < 0
negative(-42) ==> true
```

Use to create functions in place:

```
l.filter { (x: Int) => x < 0 } ==> ?
a.filter { (x: Int) => x > 0 } ==> ?
```

Alternative concise syntax

```
(abs) ==> (x: Int) => MyModule.abs(x)
```

Scala distinguishes functions and methods.

We used this syntax before to turn a method into a function (like above).

Currying and partial application

```
val add2 = (x: Int, y: Int) => x + y
val add = (x: Int) => (y: Int) => x + y
```

..... [a curried function]

What is the type of add? What is the value of add(2)(3) ==> ?

Curried functions can be partially applied: val incr = add(1) [a partial application]

Type of incr? Value of incr(7) ==> ?

Methods can also be curried: def add(x: Int)(y: Int): Int = x + y

This Course is About ...

- This is **not a course in which you learn to implement one particular thing** (a data base, a neural network, an IoT controller)
- This is a course that makes you a **better programmer of anything**. Anything can be programed in a pure manner.
- You learn to program **concisely**, in an **organized readable way**
- You learn to use **types** to increase both **safety** and **reuse** in your programs
- You learn to **design uniform standard APIs** that are familiar to other experienced programmers like you
- You learn to **test** your programs much more intensively, with better coverage.
- Concurrency (Akka, ZIO), Computational Graphs (Tensor Flow), Big Data (Spark), Probabilistic Programming (Figaro, PyMC3), Reactive Programming (Rx) are all examples of concrete contexts in which these skills will be useful.
- Sometimes it hurts, but this is because you are growing.

Course Organization

- Our **website** is LearnIT + a git-repo
- **Reading:** read prescribed book chapters and papers **before class**.
- **Lectures:** 15 weeks. Summarize the main points, but may skip details needed in exercises
- **Exercises (Homeworks):** ca. 13 weeks, (10 graded) same days as lectures
- Individual hand ins, but allowed, even encouraged to discuss with others (**mix across BSc-programs**). You must type your hand-in yourself
- **Exam:** written, need to pass 5 homeworks to be admitted, old exams will be published
- **To pass a homework:** (1) the code must compile and (2) at most 5 tests can fail
- Communicate **in class** on Thursday, daily on the **Teams**, Thursdays in **CS Study Lab**
- Andrzej and Florian are **terrible with email**. Use email only for sensitive personal matters.
- Teachers are **available during exercises for the first 30 minutes**. Each group is strongly encouraged to discuss at least one hand in to get a sense of requirements. Discuss with teachers even if you think you are doing fine.

In the next episode ...

- Functional Programming 101 continued
- Algebraic data types, pattern matching, higher order functions, polymorphism, folding
- The reading should be relatively easy, so remember to read before class !