RICKER LYMAN
ROBOTIC

Parallel Functional & Streaming Programming with Scala

Week 1: Scala Language Pt. 1

Hello!

Main topics

Part 1. Intro to Big Data & parallel processing

Part 2. Scala

Part 3. Infrastructure & Cloud

Part 4. Kafka & Kafka Streams

Volodymyr Kondratenko
COO

Yurii Ostapchuk
Engineer

Jeffrey Ricker
CEO

Andrii Sirak
Architect

Andrii Yurkiv
Assistant

Let’s get acquainted with each other

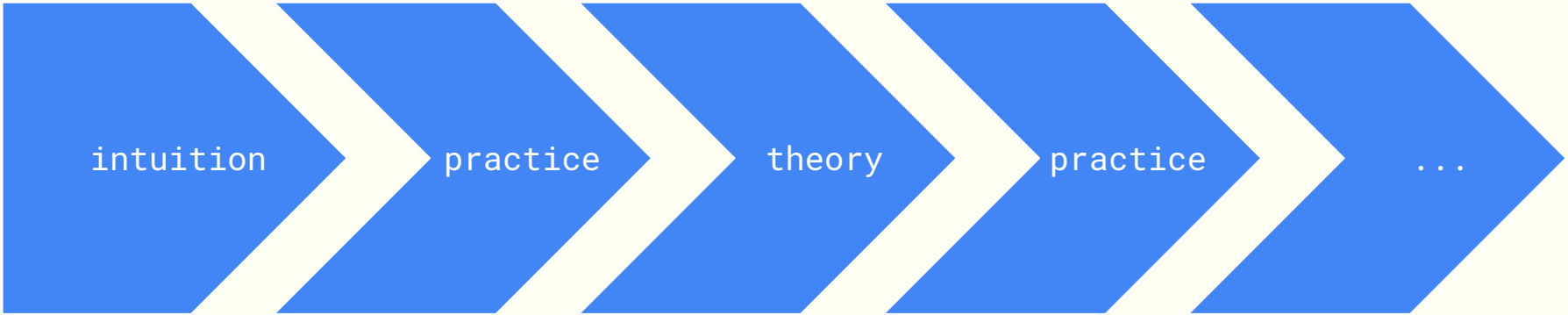
Goal

Get proficient in Scala to build streaming solution on Kafka, be able to deploy it locally and in the Cloud

Scala Part

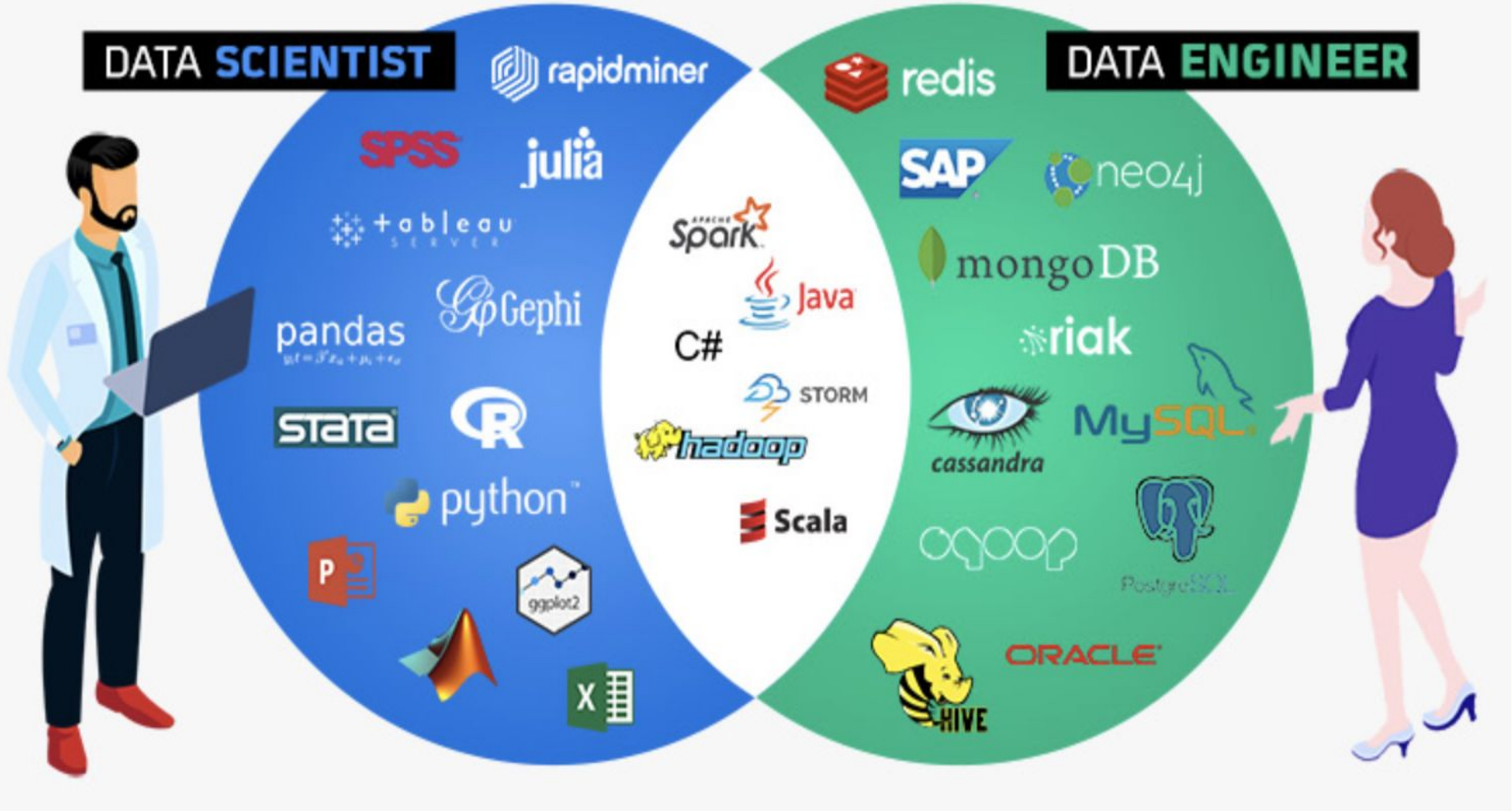
- Scala language
- Functional programming in Scala
- Parallel, Concurrent and Distributed programming in Scala
- Akka
- Practice, practice, practice

Learn



Today’s outline

- Why Scala?
- What is it?
- Language design
- What makes it special?
- Syntax
- Get our hands dirty


DATA SCIENTIST
rapidminer
SPSS
julia
+ableau SERVER
pandas
Gephi
STATA
R
python
P
ggplot2
X
Spark
Java
C#
STORM
hadoop
Scala
DATA ENGINEER
redis
SAP
neo4j
mongoDB
riak
cassandra
MySQL
HIVE
ORACLE

Scala can be scary / beautiful

```
trait ~>[F[_], G[_]] extends Poly1 {
 def apply[T](f : F[T]) : G[T]
 implicit def caseUniv[T]: Case.Aux[F[T], G[T]] = at[F[T]](apply(_))
}
object ~> {
 implicit def inst1[F[_], G[_], T](f : F ~> G) : F[T] => G[T] = f(_)
 implicit def inst2[G[_], T](f : Id ~> G) : T => G[T] = f(_)
 implicit def inst3[F[_], T](f : F ~> Id) : F[T] => T = f(_)
 }
type ~>>[F[_], R] = ~>[F, Const[R]#λ]
object identity extends (Id ~> Id) {
 def apply[T](t : T) = t
}
implicit def hconsSomeHelper[K <: Symbol, H, T <: HList, LabT <: HList, OutT <: HList]
(implicit tailHelper: Aux[T, LabT, OutT]): Aux[Some[H] :: T, K :: LabT, FieldType[K, H] :: OutT] =
 new Helper[Some[H] :: T, K :: LabT] {
   type Out = FieldType[K, H] :: OutT
   def apply(l: Some[H] :: T) = field[K](l.head.get) :: tailHelper(l.tail)
 }
```

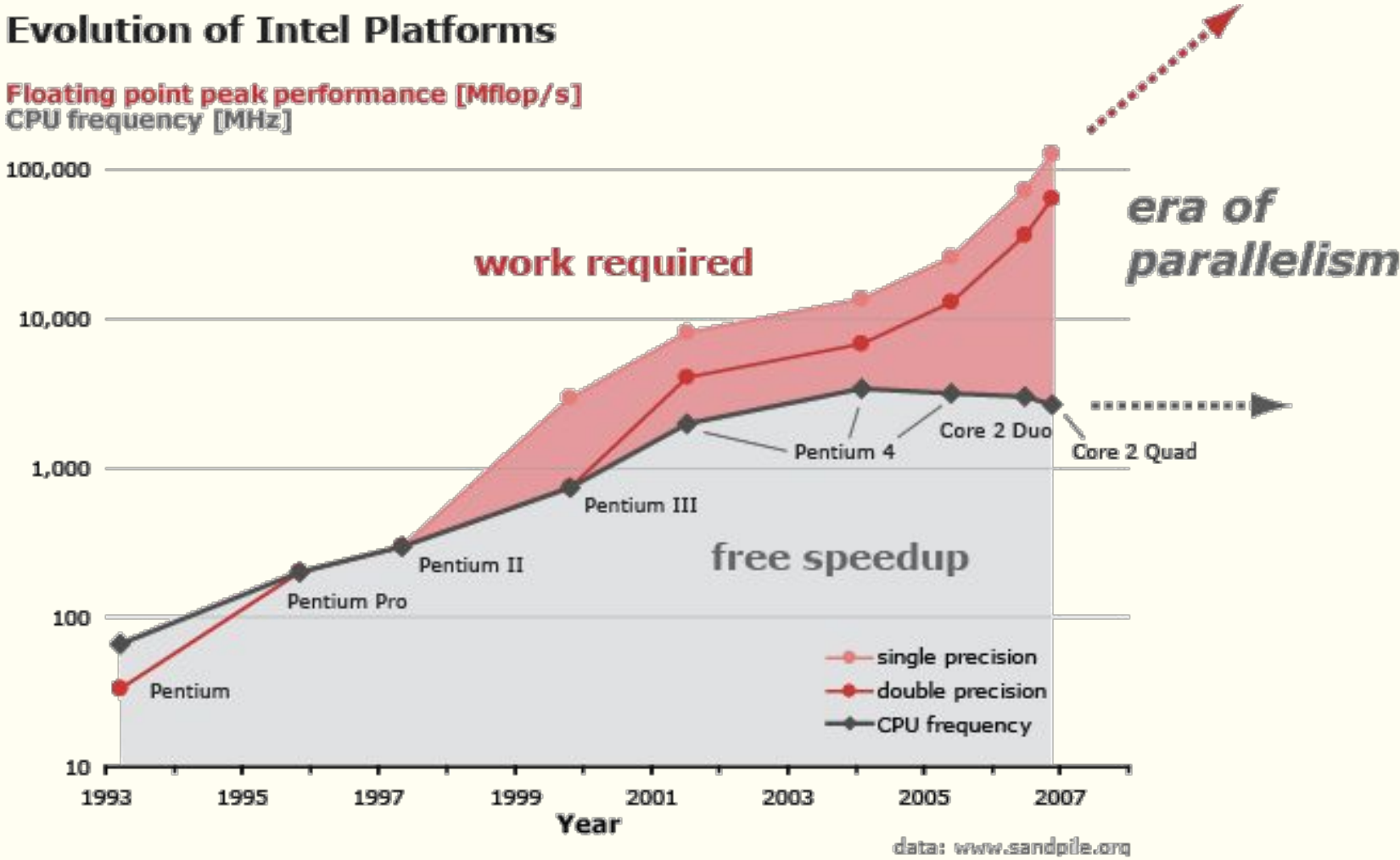
Scala

Scalable language

History

- Designed in 2004
- by Martin Odersky, a professor at the Ecole Polytechnique Federale de Lausanne, in Switzerland
- formerly worked on javac compiler, and Java generics

Moor's law


Evolution of Intel Platforms
Floating point peak performance [Mflop/s]
CPU frequency [MHz]
100,000
10,000
1,000
100
10
work required
era of parallelism
free speedup
Pentium
Pentium Pro
Pentium II
Pentium III
Pentium 4
Core 2 Duo
Core 2 Quad
single precision
double precision
CPU frequency
1993
1995
1997
1999
2001
2003
2005
2007
Year
data: www.sandpile.org

Scala is **Functional language**

Scala is also **Object-oriented** language

Scala is **statically typed** language

Runs on **JVM**, compiles to **bytecode**

Special features

Scala is **Functional language**

Scala is also **Object-oriented** language

Scala is **statically typed** language

Runs on **JVM**, compiles to **bytecode**

Special features

Programming Paradigms

Paradigm: In science, a paradigm describes distinct concepts or thought patterns in some scientific discipline.

Programming Paradigms

More intuitive:

Programming style

Language is an **instrument**

Programming Paradigms

Main programming paradigms:

- Imperative programming
  - Procedural
  - Object oriented
- Declarative programming
  - Functional programming
  - Logic programming
  - Dataflow
  - …
- Event-driven
- Parallel computing
- Metaprogramming
- Dynamic/scripting
- ...

Programming paradigms

- Imperative
  - Java, Python, C++, C#
- Declarative
  - SQL, HTML, XML, CSS

Imperative

How?

Declarative

What?

liberties constrain, constraints liberate

Programming paradigms

- Imperative
  - Java, Python, C++, C#
- Functional
  - Lisp, Haskell, F#, Clojure, Scala, Erlang

Imperative programming

is about

▶️ modifying mutable variables, -> (memory cells)

▶️ reference variables -> (load instructions)

▶️ using assignments -> (store instructions)

▶️ and control structures such as if-then-else, loops, break, continue, return. -> (jump instructions)

Imperative programming

Problem: Scaling up.

“How can we avoid conceptualizing programs word by word?”

Reference: John Backus, Can Programming Be Liberated from the von. Neumann Style?, Turing Award Lecture 1978.

Scaling Up

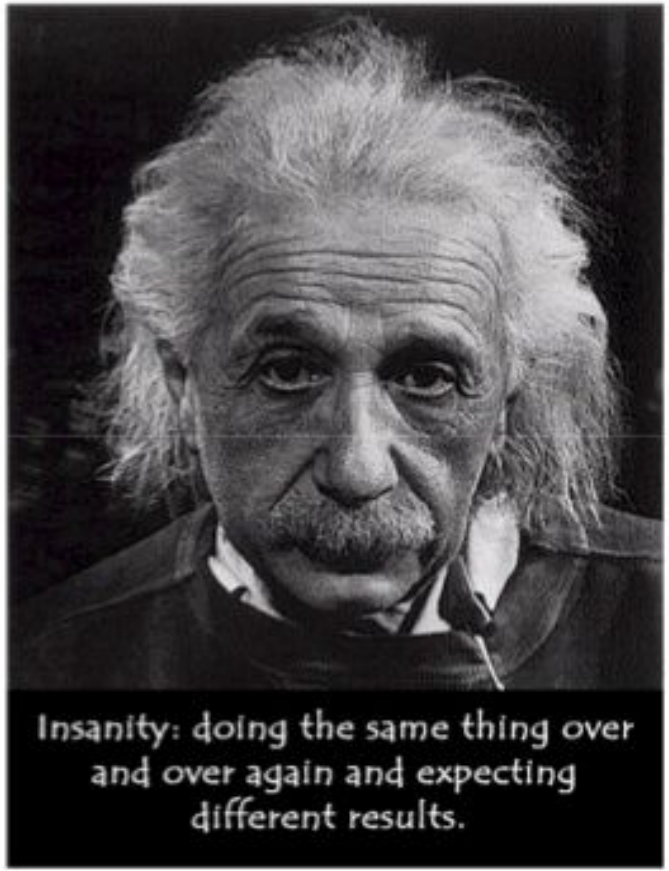
- Vertical scaling
- Horizontal scaling

Questions:

- Where the execution should happen?
- How shared state has to be managed?
- ..

The need for higher level of programming abstractions rises

The cause of the problem …



Mutable state +
Parallel processing =
Non-determinism

Define some variables

- Scala has two kinds of variables, **val**ues and **var**iables

```
// Variable

var greeting = "Hello, world!"

greeting = "Leave me alone, world!" // OK

// Value

val msg = "Hello, world!"

msg = "Goodbye cruel world!" // error: reassignment to val
```

Functional programming

Simplified:

- **No mutable** shared state
- Functions as a **values** (first-class citizens)
- Program composed of functions (**composition**)
- Program satisfy particular **laws**

Function definition

```
def square(x: Int): Int = x * x
```

Object-oriented

- Encapsulation & Inheritance & Polymorphism
- Everything is an object
- Every function call is also a method call
- Functions ⇔ Methods

Functional and Object Oriented are orthogonal

In other words

Both styles can be used simultaneously

Scala is **Functional language**

Scala is also **Object-oriented** language

Scala is **statically typed** language

Runs on **JVM**, compiles to **bytecode**

Special features

Static vs Dynamic

```
var numb = 1

numb = "1"// you cannot do that in statically typed
language
```

It’s all about contract between components. Exposing it and maintaining it

Which leads to better **scalability** as well

Static vs Dynamic, Compiled vs Interpreted

```
var numb = 1

numb = "1"// you cannot do that in statically typed
language
```

It’s all about contract between components. Exposing it and maintaining it

Which leads to better **scalability** as well

Static vs Dynamic, Compiled vs Interpreted

- **Statically typed** – the type of names (variables, fields etc.) is known at compile time.
- **Dynamically typed** – the type is associated with values at runtime.

Explicit Types

```scala
val sum: Int = 1 + 2 + 3

val nums: List[Int] = List(1, 2, 3)

val map: Map[String, List[Int]] = ...
```

Type Inference

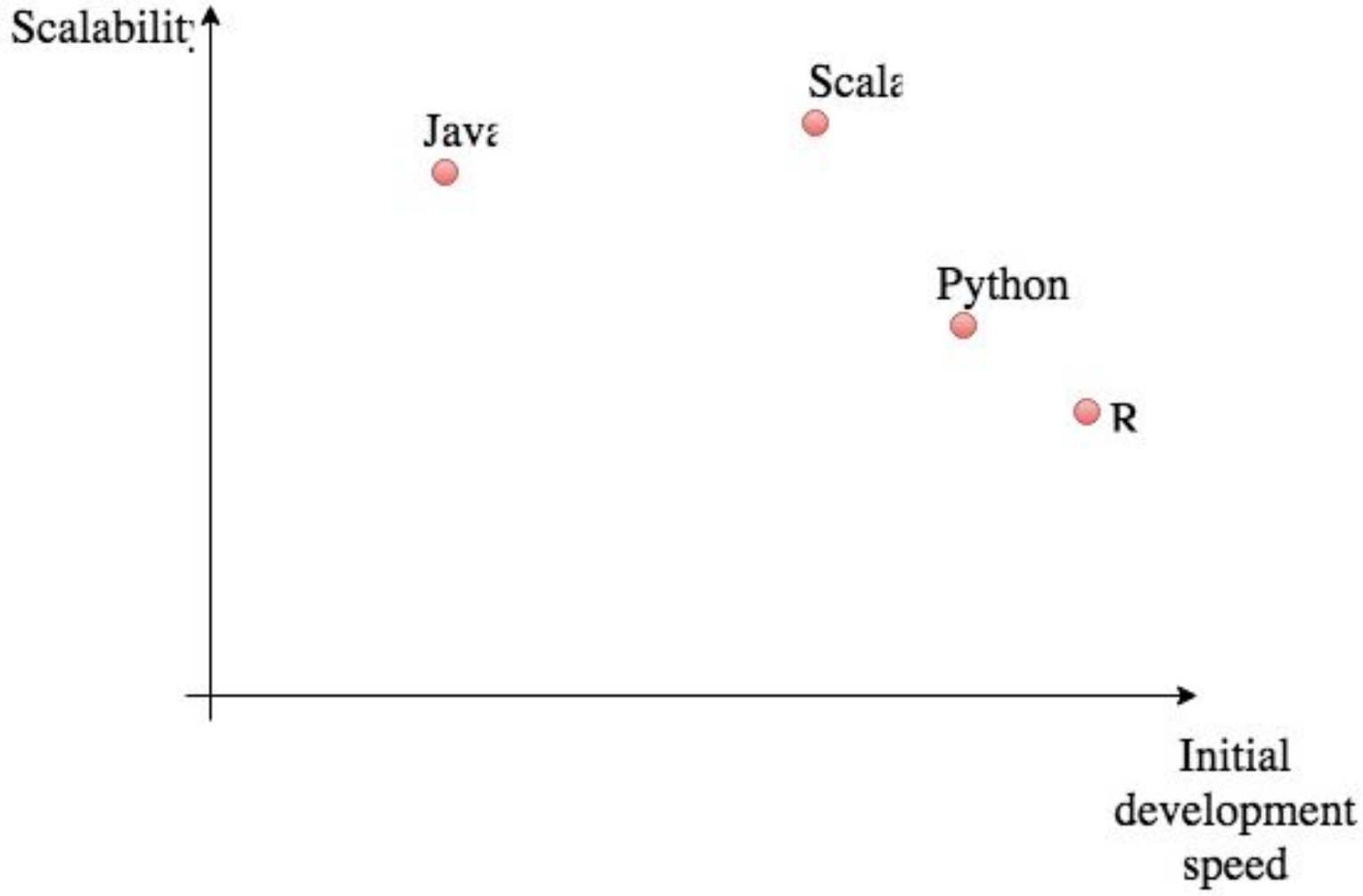
```
val sum = 1 + 2 + 3

val nums = List(1, 2, 3)

val map = Map("abc" -> List(1,2,3))
```

Statically-typed programming language

- The compiler as validity check
  - Verifies types of objects
  - Application compiles => ready for production (almost)
  - First level of defence against errors
- Type inference
  - Scala compiler infers the types
  - You don't need to type type declarations, but the program is type-checked
  - Safer applications with less typing


Scalabilit
Java
Scala
Python
R
Initial
development
speed

Everything is an Expression (almost everything)

In programming language terminology, an "**expression**" is a combination of values and functions that are combined and interpreted by the compiler to create a new value, as opposed to a "**statement**" which is just a standalone unit of execution and doesn't return anything.

The process of combining and interpreting, as for mathematical expressions, is called evaluation.

In scala, many constructs like code-blocks, if-else, method bodies are expressions

```
val ifThenElseExpression = if (aBool) 42 else 0
```

Expression vs Statement

- What is difference in between:

```
val isEven = if (number > 0) "true" else "false"
```

vs

```
var isEven: String
if(number > 0)
  isEven = "true"
else isEven = "false"
```

- Or difference in between (Java):

```
List<Integer> ls = new ArrayList<Integer>();
ls.add(1);
ls.add(2);
```

Vs

```
List<Integer> ls = Arrays.asList(1,2);
```

Value definition

```
val <name>: <Type> = <expression>
```

Example:

```
val positive: Int = {
 if (numb > 0) 1 else -1
}
```

If - Else

Structure:

```
if (<boolean expression>) <expression>
else <expression>
```

Example:

```
val num = 23
val ans2 = if(num > 0) {
  val i = 1
  val j = 2
  i + j
} else {
  val i = 1
  val j = 2
  i - j
}
```

if-else is itself also an expression

Code blocks

- In blocks, the last statement of the code-block becomes the return value

```
val c = {
    val i = 1
    val j = math.pow(2, 10)
    i - j
}
```

Above i-j is the last statement, so it becomes the value of c

- The type of the last statement becomes the type of the target variable

```
val sqr: Boolean = {
      val a = 1
      val b:Long = 23L
      val hi = "hi"
      true
  }
```

Function bodies

```
def <name>(<arg1>: <Type1>, <arg2>: <Type2>,..): <TypeR> =
<expression>
```

Function bodies

```
def square(x: Int): Int = {
 val result = x * x
 return result
}
```

Scala is **Functional language**

Scala is also **Object-oriented** language

Scala is **statically typed** language

Runs on **JVM**, compiles to **bytecode**

Special features

Runs on JVM

- Java interoperability
  - Jython, Clojure, Groovy, Kotlin..
- Java Ecosystem & Tooling
  - Big Data ecosystem
- Cross-platform
  - You just need to install Java
  - Simple containerized deployment

Scala is **Functional language**

Scala is also **Object-oriented** language

Scala is **statically typed** language

Runs on **JVM**, compiles to **bytecode**

Special features

What makes it really special

- REPL (Read Evaluate Print Loop)
- Implicits
- Macro
- Advanced type-level programming
  - Shapeless
- Duck typing (structural typing)
- scala.language.dynamics
- string interpolation
  - sql”select * from user where age > 20”
- XML Literals
- Type Specialization

Not enforced, but very powerful (FP purity as an option - as well)

Syntax

RICKER LYMAN ROBOTIC

Get your hands dirty with Scala

REPL

- Supports tab completing, imports
- Type **>scala** to start
- Special commands
  - Commands with : at the front

    :help to get started

    :quit to quit

    :save

Imports

```
import scala.util.Random

// method

import scala.util.Random.nextInt

// wildcard

import scala.util._

// named

import scala.util.Random.{nextInt => next}
```

Imports

```
scala> import scala.util.Random.{nextInt => next}

import scala.util.Random.{nextInt=>next}


scala> next

res0: Int = -859814294
```

println

Output the result of computing the expression

```
> println(expression)
```

Worksheet

AKA Jupyter notebooks

Syntax

- Primitives
- Control Flow
  - if-else
  - loops
- Functions
- Lists
- Tuples
- Pattern matching
- Recursion

Primitive data types

Data Type	Definition
Boolean	true or false
Byte	8-bit signed two's complement integer (-2^7 to 2^7-1, inclusive) -128 to 127
Short	16-bit signed two's complement integer (-2^15 to 2^15-1, inclusive) 32,768 to 32,767
Int	32-bit two's complement integer (-2^31 to 2^31-1, inclusive) 2,147,483,648 to 2,147,483,647
Long	64-bit two's complement integer (-2^63 to 2^63-1, inclusive) -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
Float	32-bit IEEE 754 single-precision float 1.40129846432481707e-45 to 3.40282346638528860e+38 (positive or negative)
Double	64-bit IEEE 754 double-precision float 4.94065645841246544e-324d to 1.79769313486231570e+308d (positive or negative)
Char	16-bit unsigned Unicode character (0 to 2^16-1, inclusive) 0 to 65,535
String	a sequence of Chars

Recap: Values

Value definitions

```
val a = 1

var b = 2

b = -2

val c, d, e = "wow"
```

Recap: Values

Value definitions

```
val a = 1L

val b: Double = 1d

val c = 1f

val hex = 0x5

val hex2 = 0x00FF

val magic = 0xcafebabe
```

Recap: Inference

Inference

```
val result = if (true) 1D else 1L

res0: Double = 1.0
```

You can also cast

```
1: Long

1.asInstanceOf[Long]
```

But:

```
"1".asInstanceOf[Long]

    java.lang.ClassCastException

"1": Long

    error: type mismatch;
```

Instead:

```
"1".toInt
```

Recap: If-Else

- `if (a > 0) b = 2 else b = -2`

- ```
  if (a > -1 & a <= 4 && a != 0 | (1 == 2 || "1" != "2")) {
      println("b = 2")
      b = 2
  } else {
      b = -2
  }
  ```

- & and | are strict while && and || are short-circuiting
  ```

List

- List definition (with type inference)

```
val list = List(1, 2, 3, 4, 5)
```

- Adding an element to the head of a list

```
val list1 = 0 :: list
```

- Adding an element to the tail of a list

```
val list2 = list1 :+ 6
```

- Concatenating lists

```
val list3 = list1 ++ list2
```

for loops

```
for ( <i> <- <s> ) <expression>
```

Remark: <s> has to be a subtype of Traversable (Arrays, Collections, Tables, Lists, Sets, Ranges, . . .)

Ranges for for-loops can be built using **.to(...)**

```
(1).to(5)
```

```
1 to 5
```

Equals to:

```
Range(1, 2, 3, 4, 5)
```

for loops

Example:

```
for (
   i <- 1 to 5
) println(i)
```

for yield

for (seq;) **yield** expr

You can ‘yield’ an expression to produce an output sequence

```
for {
   i <- 1 to 10
   j <- 1 to 100
   if j == i * i
} yield j
```

Exercise:
Given **val** *lb*=*List*(1,2,3,4,5) and using **for**, build the list of squares of *lb*.

while

```
var i: Int = 0

while(i < 10) {

 print(i)

 i += 1

}
```

Function definition

RICKER LYMAN ROBOTIC

Function definition

```
def <fName>(<arg1>: <Type1>,..., <argn>:<Typen>): <Typef> = { <expr> }
```

- Remark 1: type of <expr> (the type of the last expression of <expr>) is Typef
- Remark 2: Typef can be inferred for non recursive functions
- Remark 3: The type of **fName** is : (Type1,. . . ,Typen) => Typef

Function definition

```
def getFullName(firstName: String, lastName: String): String = {
   firstName + " " + lastName
}
```

A type of this function:

```
(String, String) => String
```

Default Arguments

```
def getUser(
            firstName: String = "John",
            lastName: String = "Smith",
            age: Int = -1
            ): Unit = {
 // ...
}
```

Named Arguments

```
createUser(user, true, false, false, true, false, false)
```

Better?

```
createUser(
 user = user,
 encryptPassword = true,
 admin = false,
 ldapAuth = false,
 suspicious = true,
 blocked = false,
 visible = false)
```

Infix/Dot notation

```
1 + 2

(1).+(2)

result.append(a).append(b).append(c)

result append a append b append c
```

Anonymous functions

```
(x: Int) => x + 2 // full version
```

Anonymous functions

```
(x: Int) => x + 2 // full version

x => x + 2 // type inferred
```

Anonymous functions

```
(x: Int) => x + 2 // full version

x => x + 2 // type inferred

_ + 2 // placeholder syntax (each argument must be used
                                                exactly once)
```

Anonymous functions

```
(x: Int) => x + 2 // full version

x => x + 2 // type inferred

_ + 2 // placeholder syntax (each argument must be used
                                        exactly once)

x => { // body is an expression
   val numberToAdd = 2
   x + numberToAdd
}
```

Operations with List

```
val list = List(1, 2, 3)
list.foreach(x => println(x)) // prints 1, 2, 3
list.foreach(println) // same
list.map(x => x + 2) // returns a new List(3, 4, 5)
list.map(_ + 2) // same
list.filter(x => x % 2 == 1)// returns a new List(1, 3)
list.filter(_ % 2 == 1) // same
list.reduce((x, y) => x + y) // => 6
list.reduce(_ + _) // same
list.sorted // sorted
list.sortWith((x, y) => x < y)
```

Functions

```
// Regular function definition
def addTwo(x: Int): Int = x + 2
// Value definition
val addTwo = (x: Int) => x + 2
// In these cases a type of the value is
// Int => Int
val addTwo: Int => Int = _ + 2
```

Closures

- An anonymous function is just a function that has no name; nothing more.
- A closure is a function that captures the state of the surrounding environment.

Nested functions

You can nest function definitions. As well as code blocks - as any other kind of expression.

```
// Can nest multiple levels of functions
def outer() {
 var msg = "foo"
 val f = (x: Int) => {
   println(msg)
 }
 def one() {
   def two() {
     def three() {
       println(msg)
     }
     three()
   }
   two()
 }
 one()
}
```

Lexical scoping

In a Scala program, an inner variable is said to shadow a like-named outer variable, because the outer variable becomes invisible in the inner scope.

Local Definitions → Explicits imports → Wild card imports → Packages

Pattern matching

RICKER LYMAN ROBOTIC

Pattern matching

match - case expressions (like switch-case):

```
<expr> match {
 case <pattern1> => <r1> //patterns can be constants
 case <pattern2> if <bool_expr> => <r2> //or terms with variables
   ... //or terms with holes: '_'
 case _ => <rn> // default case
}
```

- A series of cases
- Cases checked until first one matches
- First match wins
- No other cases checked
- No matching case leads to scala.MatchError

Pattern matching

match - case expressions (like switch-case):

```
<expr> match {
 case <pattern1> => <r1> //patterns can be constants
 case <pattern2> if <bool_expr> => <r2> //or terms with variables
   ... //or terms with holes: '_'
 case _ => <rn> // default case
}
```

Example:

```
x match {
 case "bonjour" => "hello"
 case "au revoir" => "goodbye"
 case _ => "don't know"
}
```

Remark: the type of this expression is the supertype of r1, r2, . . . rn

Pattern matching List

Doing pattern-matching over lists

```
val list = List(1,2,3,4,5)
val head = list match {
  case Nil => 0
  case head :: tail => head
}
```

Pattern matching on assignment (unapply)

```
val (x, y) = (3, 5)

val x :: xs = List(1, 2, 3, 4)

val List(a, b, c) = List(1, 2, 3)
```

Tuple

RICKER LYMAN ROBOTIC

Tuples

Tuples are immutable containers of values

```
val t = (1, "toto", 18.3)
// t: (Int, String, Double) = (1,toto,18.3)
```

Tuple getters: t. 1, t. 2, etc.

```
t._1 // 1: Int
t._2 // toto: String
```

... or with match - case:

```
t match {
   case (2, "toto", _) => "found!"
   case ( _, x, _) => x
}
```

The above expression evaluates in "toto"

Tail Recursion

RICKER LYMAN ROBOTIC

Recursion

```
// basic recursive factorial method
def factorial(n: Int): Int = {
 if (n == 0) 1 else n * factorial(n-1)
}

factorial(4)
if (4 == 0) 1 else 4 * factorial(4 - 1)
4 * factorial(3)
4 * (3 * factorial(2))
4 * (3 * (2 * factorial(1)))
4 * (3 * (2 * (1 * factorial(0))))
4 * (3 * (2 * (1 * 1)))
24
```

Recursion

```
def gcd(a: Int, b: Int): Int =
 if (b == 0) a else gcd(b, a % b)
```

This is how it expands:

```
gcd(14, 21)
if (21 == 0) 14 else gcd(21, 14 % 21)
if (false) 14 else gcd(21, 14 % 21)
gcd(21, 14 % 21)
gcd(21, 14)
if (14 == 0) 21 else gcd(14, 21 % 14)
if (false) 21 else gcd(14, 21 % 14)
gcd(14, 7)
gcd(7, 14 % 7)
gcd(7, 0)
if (0 == 0) 7 else gcd(0, 7 % 0)
if (true) 7 else gcd(0, 7 % 0)
7
```

Tail recursion

```scala
import scala.annotation.tailrec

// tail-recursive gcd method
 @tailrec
def gcd(a: Int, b: Int): Int =
 if (b == 0) a else gcd(b, a % b)
```

Various

- ; is usually omitted

  but if you write multiple statements into one line - ; acts like a separator

  val one = 1; val two = 2
- ??? - useful placeholder for non-implemented-yet methods/values
- scala.Predef

Questions

THANK YOU!