Scala - the scalable language

Daniel Ploetzer and Eric Heder

Seminar on scientific languages
AICES, RWTH Aachen University

19.12.2013

Table of contents

Motivation

What is scientific computing?

transdisciplinary

- mathematics
- informatics
- field of application

use of computers to analyze and solve scientific problems

- computer simulation
- numerical computations
- data analysis
- computational optimization (HPC)

$\rightarrow$ way to obtain knowledge apart from experiment

Motivation

What do we want from a language?

Must-haves

- fast and easy prototyping
- high performance
- low memory usage
- support of
  - parallel programming
  - mathematical calculations/expressions
  - graphic plotting

Nice-to-haves

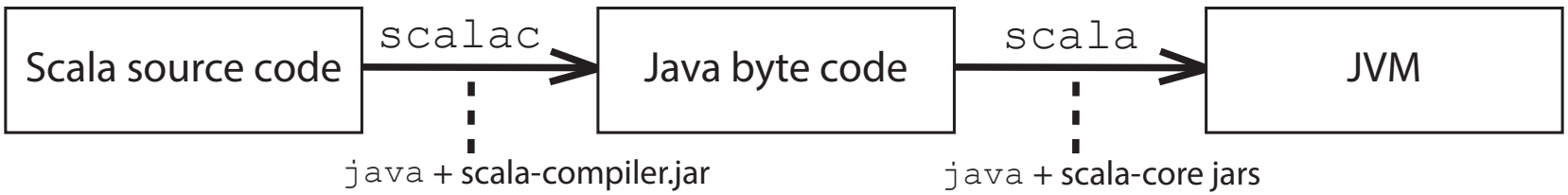
- native parallel programming support
- portable
- free license
- active community

- italian for stairway
- scalable language

- under BSD license
- created by Martin Odersky
  - german computer scientist
  - professor in the programming reasearch group at *Swiss Federal Institute of Technology* in Lausanne
  - developer of the current version of `javac`
- history in a nuthsell
  - 2001: project started
  - 2004: first version on the java plattform
  - 2006: version 2.0
  - 2011: launch of *Typesafe Inc.*
    $\Rightarrow$ commercial support, services and training for Scala
  - 2013: latest version 2.10.3

What is Scala?

- programming language, which extends `java`
  $\Rightarrow$ any existing java library can be used



- also usable as interactive shell

- hybrid language:
  - functional
  - object-oriented

- based on the *lambda calculus* $\rightarrow$ formal system in logic and cs from the 1930s
- everything is a function (esp. values are 0-ary functions)
- results only depend on input parameters
- no side effects
- avoid reuse of variables

- based on the *lambda calculus* $\rightarrow$ formal system in logic and cs from the 1930s
- everything is a function (esp. values are 0-ary functions)
- results only depend on input parameters
- no side effects
- avoid reuse of variables

$\Rightarrow$ treat functions like mathematical functions
e.g. `x1 = x + 1` instead of `x = x + 1`

- common way: code depends on how to do it

```
def sumOfEquals(xs: List[Int]): Int = {
    var sum = 0
        for (x <- xs)
            if (x%2 == 0)
                sum += x
    sum
}
```

- common way: code depends on how to do it

```
def sumOfEquals(xs: List[Int]): Int = {
    var sum = 0
        for (x <- xs)
            if (x%2 == 0)
                sum += x
    sum
}
```

- functional way: code depends on what to do

```
def sumOfEquals(xs: List[Int]) = xs filter{_%2 == 0} sum
```

Scala

Functional programming features in Scala

- first-class and higher-order $\rightarrow$ functions as parameter and return value

Scala

Functional programming features in Scala

- first-class and higher-order $\rightarrow$ functions as parameter and return value
- currying

```
def addInts(x: Int, y: Int, z: Int) = x + y + z // uncurried
def addInts(x: Int)(y: Int)(z: Int) = x + y + z // curried
```

Scala

Functional programming features in Scala

- first-class and higher-order $\rightarrow$ functions as parameter and return value
- currying

```
def addInts(x: Int, y: Int, z: Int) = x + y + z // uncurried
def addInts(x: Int)(y: Int)(z: Int) = x + y + z // curried
```

- lazy evaluation

```
var a = 2
lazy val b = 8 * a
a = 3
println(b) // Evaluates to 24 instead of 16
```

Scala

Functional programming features in Scala

- first-class and higher-order $\rightarrow$ functions as parameter and return value
- currying

```
def addInts(x: Int, y: Int, z: Int) = x + y + z // uncurried
def addInts(x: Int)(y: Int)(z: Int) = x + y + z // curried
```

- lazy evaluation

```
var a = 2
lazy val b = 8 * a
a = 3
println(b) // Evaluates to 24 instead of 16
```

- tail recursion $\rightarrow$ last command is recursion call

Scala

Functional programming features in Scala

- first-class and higher-order → functions as parameter and return value
- currying

```
def addInts(x: Int, y: Int, z: Int) = x + y + z // uncurried
def addInts(x: Int)(y: Int)(z: Int) = x + y + z // curried
```

- lazy evaluation

```
var a = 2
lazy val b = 8 * a
a = 3
println(b) // Evaluates to 24 instead of 16
```

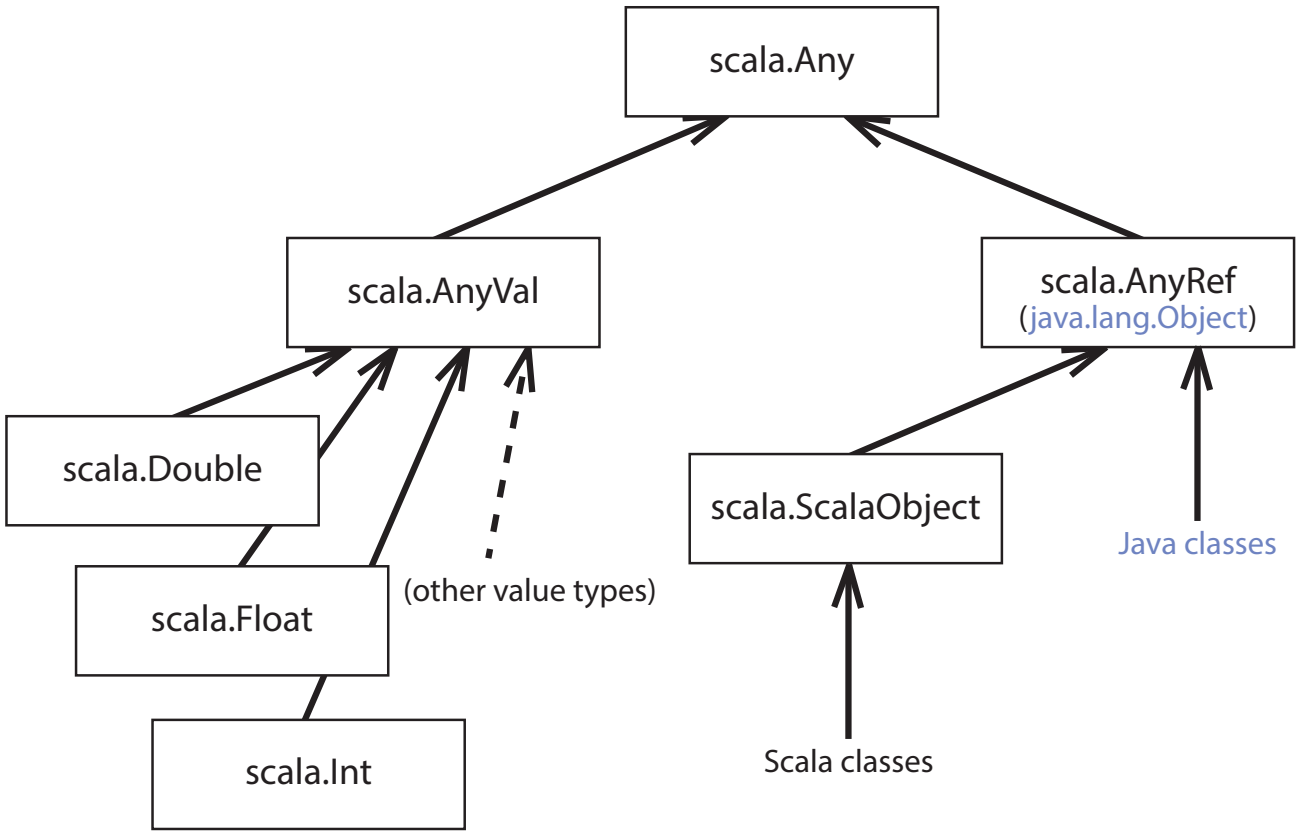
- tail recursion → last command is recursion call
- pattern matching
  → likewise `switch command` in java

Scala

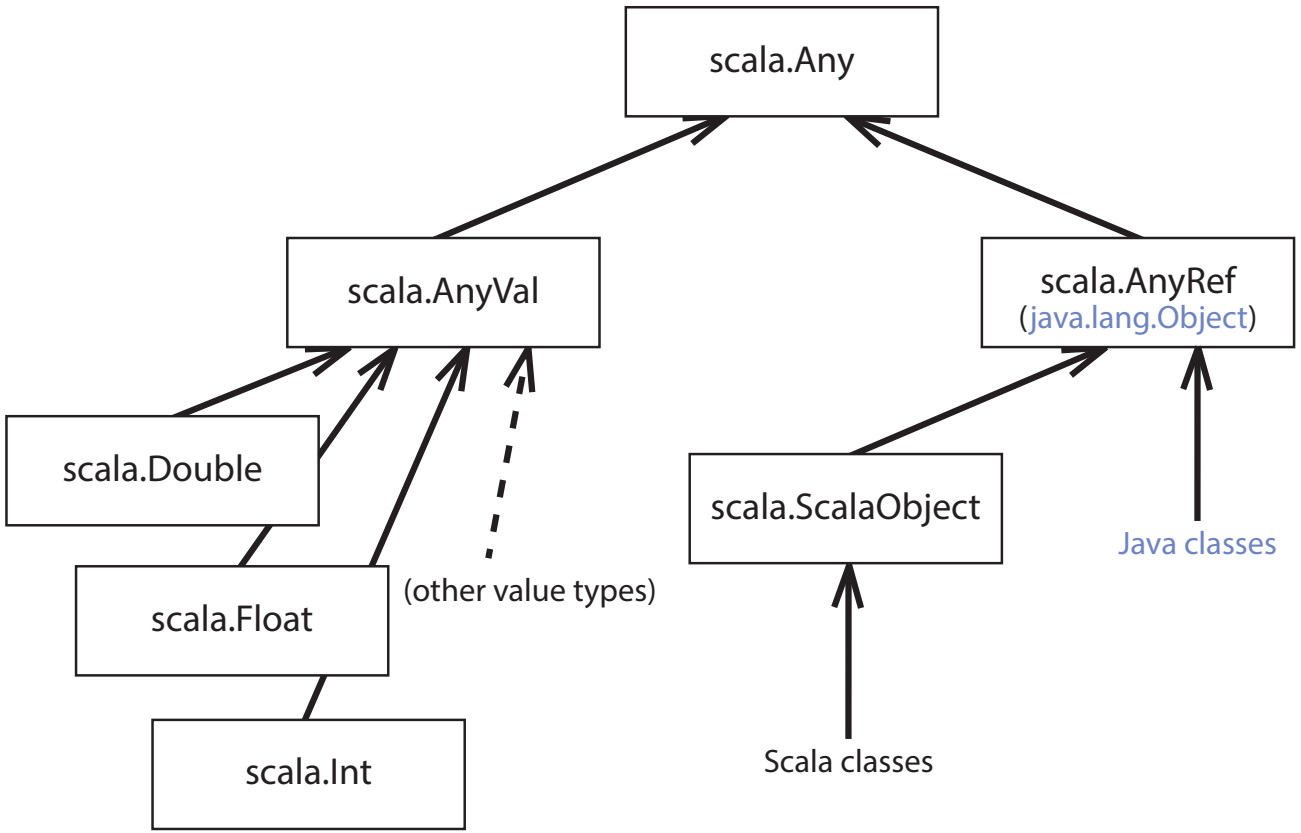
Object-oriented

pure object-oriented language: all values are objects

pure object-oriented language: all values are objects

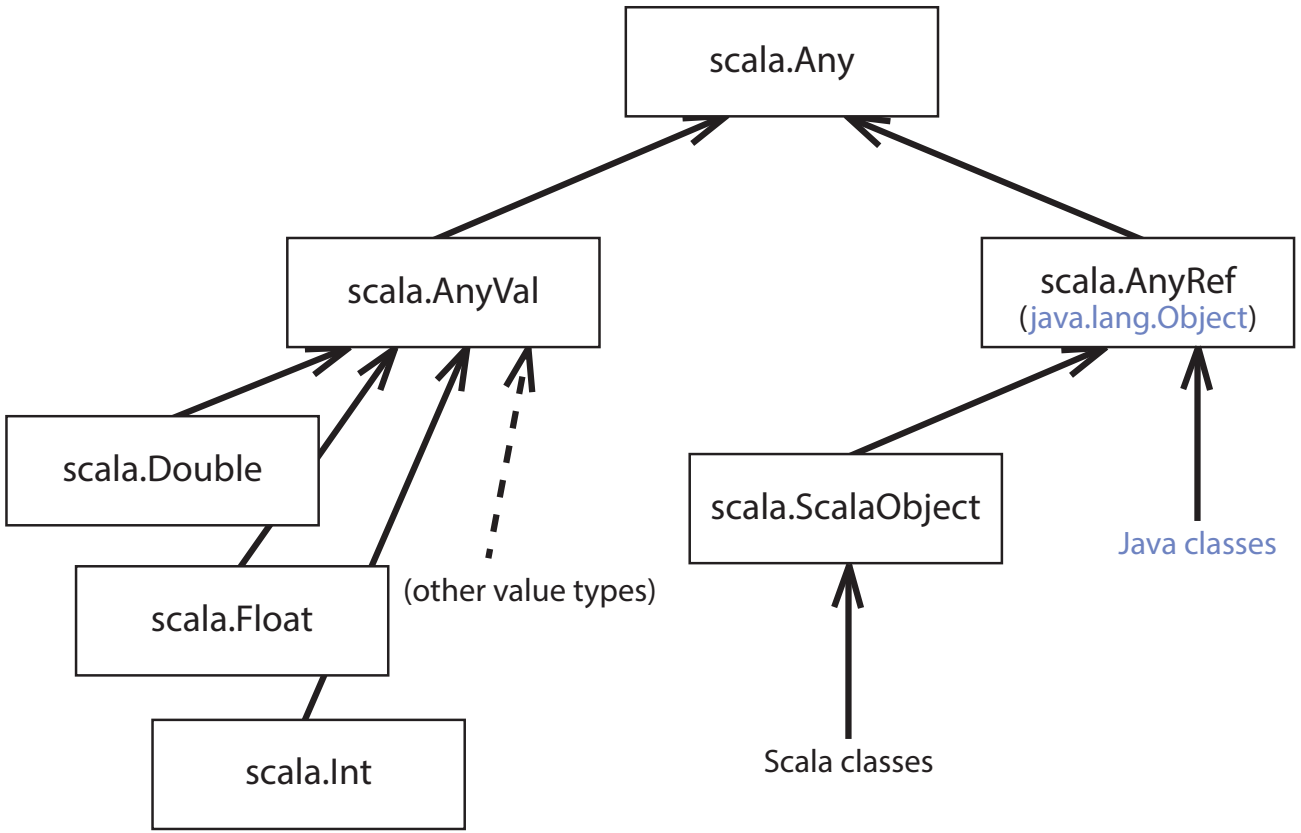


pure object-oriented language: all values are objects



$\Rightarrow$ functions are objects

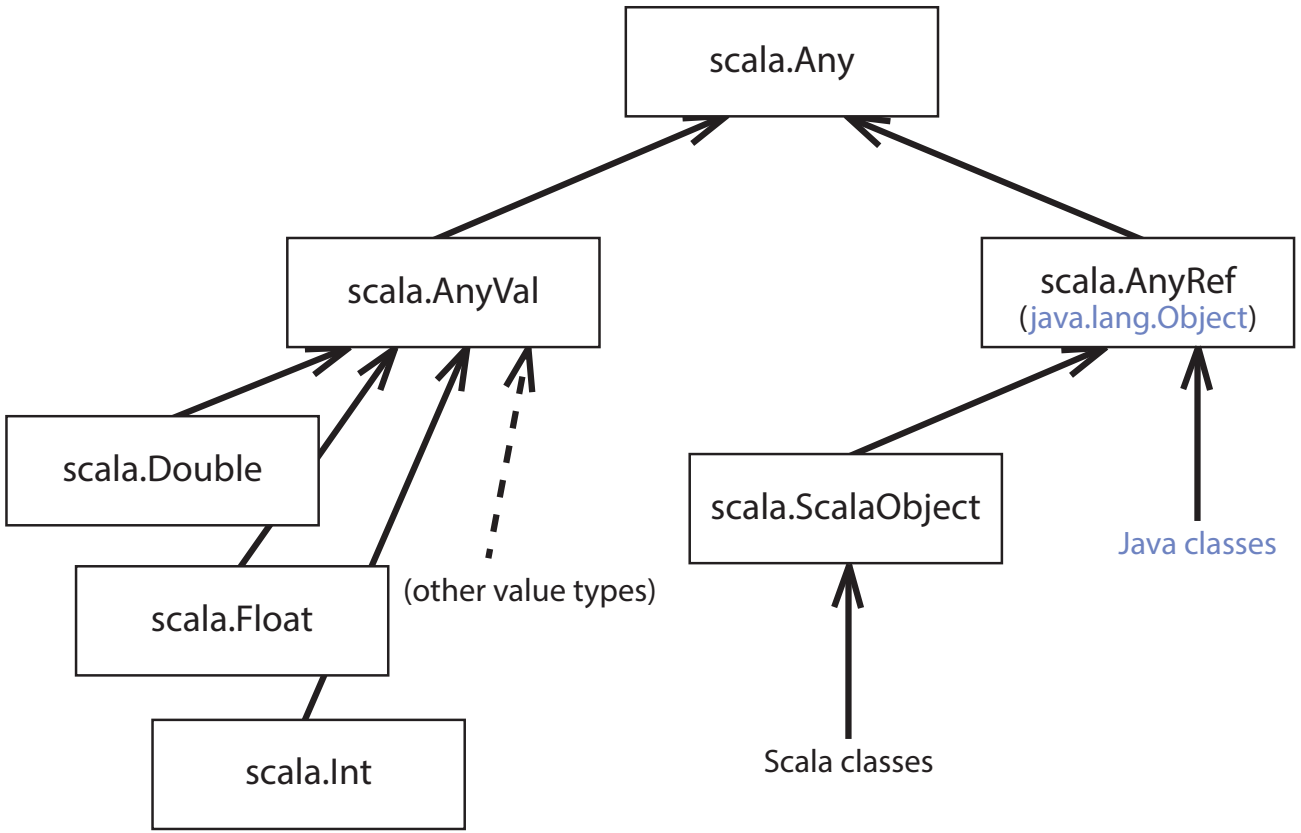
pure object-oriented language: all values are objects



$\Rightarrow$ functions are objects

$\Rightarrow$ way to implement function pointers: `val fp = myFunc _`

pure object-oriented language: all values are objects



⇒ functions are objects

⇒ way to implement function pointers: `val fp = myFunc _`

⇒ Unified Types (demo)

objects are instances of `classes` (and `traits`)

```
class mySum(a: Int, b: Int) extends Compare {
    val c = a + b
    def isGreater(i: Any) : Boolean = c > i.asInstanceOf[Int]
}
```

Listing: parameterized constructor arguments

objects are instances of `classes` (and `traits`)

```
class mySum(a: Int, b: Int) extends Compare {
    val c = a + b
    def isGreater(i: Any) : Boolean = c > i.asInstanceOf[Int]
}
```

Listing: parameterized constructor arguments

```
trait Compare {
    def isGreater(obj: Any) : Boolean
    def isNotGreater(obj: Any) : Boolean = !isGreater(obj)
}
```

Listing: partially implemented trait

objects are instances of `classes` (and `traits`)

```
class mySum(a: Int, b: Int) extends Compare {
    val c = a + b
    def isGreater(i: Any) : Boolean = c > i.asInstanceOf[Int]
}
```

Listing: parameterized constructor arguments

```
trait Compare {
    def isGreater(obj: Any) : Boolean
    def isNotGreater(obj: Any) : Boolean = !isGreater(obj)
}
```

Listing: partially implemented trait

$\Rightarrow$ traits combine advantages of Java's interfaces and abstract classes

e.g. **Singleton** through `object` definition

```
object Presentation {
  val maxDuration = 40
  def remainingTime(time: Int): Int= { maxDuration - time }
}
```

e.g. **Singleton** through `object` definition

```
object Presentation {
  val maxDuration = 40
  def remainingTime(time: Int): Int= { maxDuration - time }
}
```

```
class Presentation {
  var time = 0
  def remainingTime(): Int= { Presentation.remainingTime(time) }
}
```

Listing: companion object/class

e.g. **Singleton** through `object` definition

```
object Presentation {
  val maxDuration = 40
  def remainingTime(time: Int): Int= { maxDuration - time }
}
```

```
class Presentation {
  var time = 0
  def remainingTime(): Int= { Presentation.remainingTime(time) }
}
```

Listing: companion object/class

- elements in `object` can be thought of as `static`
- elements in `class` are dynamically instantiated

- `x method y` is syntactic sugar for `x.method(y)`
  $\Rightarrow$ e.g. `x < y` stands for `x.<(y)`

Scala

Features

- `x method y` is syntactic sugar for `x.method(y)`
  $\Rightarrow$ e.g. `x < y` stands for `x.<(y)`
- scala-core is full of little helpers, e.g.

```
> List[Int](1,2,4,5).mkString("[[", "--", "]]")
> [[1--2--4--5]]
```

- `x method y` is syntactic sugar for `x.method(y)`
  $\Rightarrow$ e.g. `x < y` stands for `x.<(y)`
- scala-core is full of little helpers, e.g.

```
> List[Int](1,2,4,5).mkString("[[", "--", "]]")
> [[1--2--4--5]]
```

choosen features:

- parameter lists
- for comprehensions
- parallel programming
- type enrichment

- default parameter

```
def addInts(x: Int = 5, y: Int = 1, z: Int= 2) = x + y + z
```

- default parameter

```
def addInts(x: Int = 5, y: Int = 1, z: Int= 2) = x + y + z
```

- named parameters

```
> addInts(y = 2, z = 1, x = 5)}
```

- default parameter

```
def addInts(x: Int = 5, y: Int = 1, z: Int= 2) = x + y + z
```

- named parameters

```
> addInts(y = 2, z = 1, x = 5)}
```

- open ended parameter lists

```
def openEndParamList(vals: Int*) = vals.foreach(println)
```

- Martin Odersky: *Scala's for expression is a Swiss army knife of Iteration.*
- extreme powerful iteration tool ⇒ more than simple loops
- iterate over objects
- e.g. loop with filter

```
for {
    i <- 1 to 6
    j <- 1 to 6
    k <- 1 to 3
    if (i % 2 == 0)
    if (j % 2 != 0)} println(i + " "+ j + " " + k)
}
```

- extend existing libraries → scalable language

```
object MyIntegerExtensions {
   implicit class IntPredicates(i: Int) {
      def isGreaterThanZero = i > 0
   }
}
```

```
> import MyIntegerExtensions._
> 3.isGreaterThanZero
```

- recall: possibility to use java libraries
  - ⇒ usability of established OpenMP and MPI implementations (e.g. `JaMP` and `mpiJava`)

- recall: possibility to use java libraries
  - ⇒ usability of established OpenMP and MPI implementations (e.g. `JaMP` and `mpiJava`)
- own implementation: actors (from *Erlang*)
  - thread instance with a mailbox ⇒ communication via messages
  - no shared but private memory
  - behavior depends on message: send message, create new actors, change own behavior
  - until version 2.10.0 part of the core ⇒ now: *Akka framework*

Scala

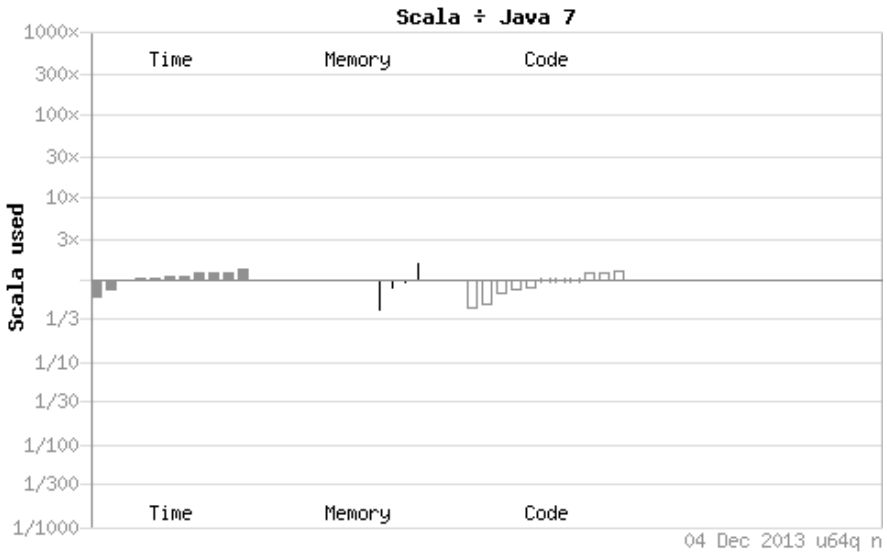
Performance

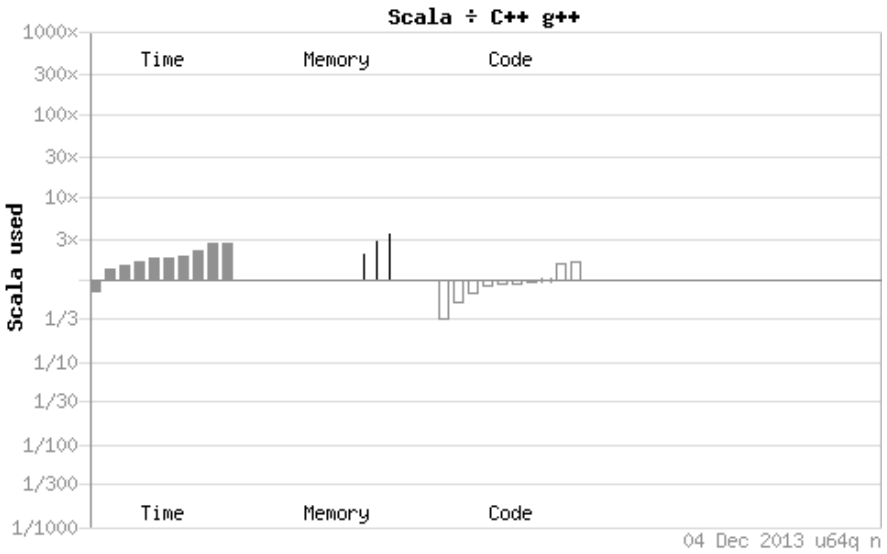
- benchmark: 64 bit, quadcore, ubuntu os
- messuring units:
  - time
  - memory usage
  - code length
- compare Scala with `Java`, `Python`, `Fortran`, `C++`
- benchmark programms are common problems of maths and computer science

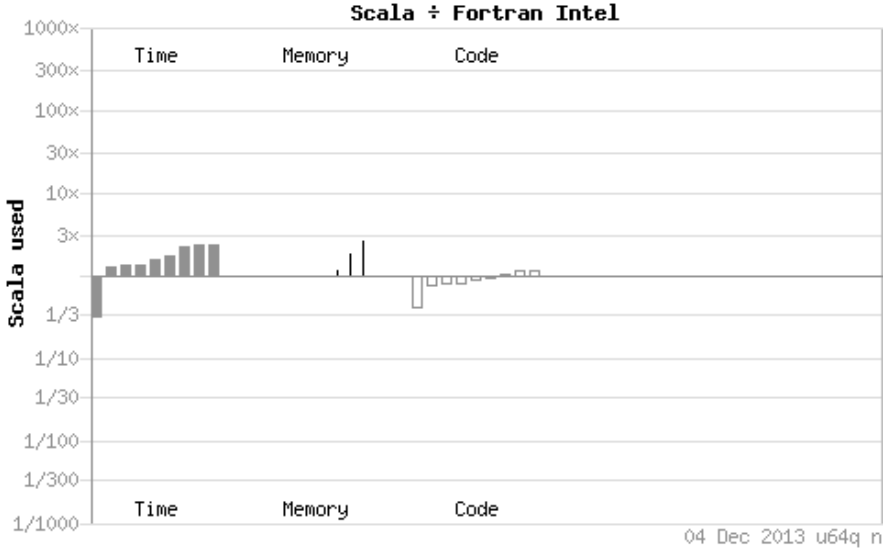
```
k-nucleotide
regex-dna
fannkuch-redux
pidigits
spectral-norm
mandelbrot
reverse-complement
n-body
fasta
fasta-redux
binary-trees
```

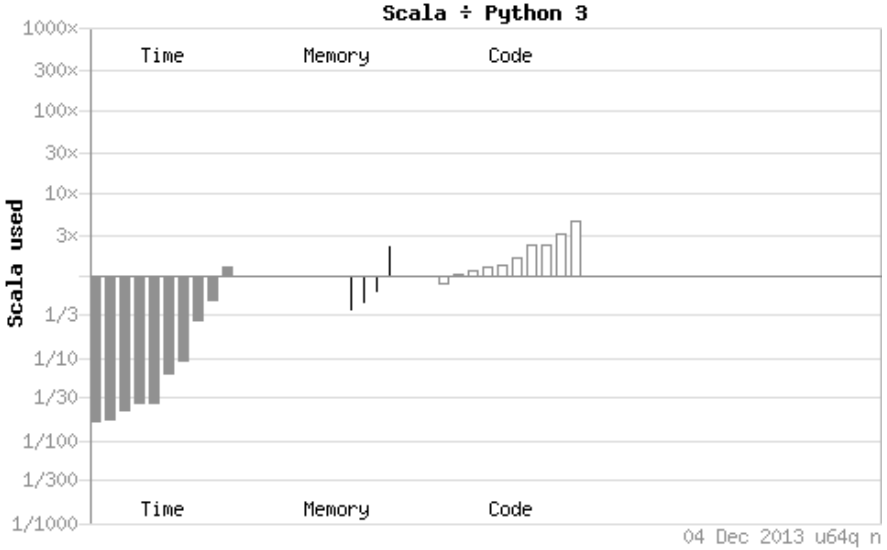
Scala

Performance









Summary

Does Scala meet the requirements? → Pro

Must-haves

- ✓ fast and easy prototyping
  → interactive shell, native design patterns, helper constructs, etc.
- support of
  - ✓ parallel programming
  - ✓ mathematical calculations/expressions
  - ✓ graphic plotting

  → all java libraries can be used

Summary

Does Scala meet the requirements? → Pro

Must-haves

- ✓ fast and easy prototyping
  → interactive shell, native design patterns, helper constructs, etc.
- support of
  - ✓ parallel programming
  - ✓ mathematical calculations/expressions
  - ✓ graphic plotting

  → all java libraries can be used

Nice-to-haves

- ✓ portable → runs on JVM
- ✓ free license → open source (BSD)
- ✓ active community → growing / nearly daily update of the doc

Summary

Does Scala meet the requirements? → Pro

Must-haves

- ✓ fast and easy prototyping
  → interactive shell, native design patterns, helper constructs, etc.
- support of
  - ✓ parallel programming
  - ✓ mathematical calculations/expressions
  - ✓ graphic plotting

  → all java libraries can be used

Nice-to-haves

- ✓ portable → runs on JVM
- ✓ free license → open source (BSD)
- ✓ active community → growing / nearly daily update of the doc

Other aspects

- hybrid paradigm: functional + pure object-oriented

Summary

Does Scala meet the requirements? → Contra

Must-haves

- × high performance → JVM + Scala overhead
- × low memory usage (but at least lower than in `Java`)

Summary

Does Scala meet the requirements? → Contra

Must-haves

- × high performance → JVM + Scala overhead
- × low memory usage (but at least lower than in `Java`)

Nice-to-haves

- × native parallel programming support

Summary

Does Scala meet the requirements? → Contra

Must-haves

- × high performance → JVM + Scala overhead
- × low memory usage (but at least lower than in `Java`)

Nice-to-haves

- × native parallel programming support

Other aspects

- eclipse + Scala = extreme slow (well-known problem)
- very slow compiler
- quite steep learning curve ⇒ a language for professionals
- possibility to write extreme unreadable code

Summary

Is Scala usable for scientific computing?

Must-haves

- ✓ fast and easy prototyping
- × high performance
- × low memory usage
- support of
  - ✓ parallel programming
  - ✓ mathematical calculations/expressions
  - ✓ graphic plotting

Nice-to-haves

- × native parallel programming support
- ✓ portable
- ✓ free license
- ✓ active community

⇒ Result: Usability of Scala depends on weighting of the requirements

Sources

- http://www.scala-lang.org/
- http://www.scala-lang.org/docu/files/ScalaReference.pdf
- http://www.scalatutorial.de/
- http://benchmarksgame.alioth.debian.org/
- http://pavelfatin.com/design-patterns-in-scala/
- http://gleichmann.wordpress.com/2010/11/08/functional-scala-functions-as-objects-as-functions/