CS 3101-2 - Programming Languages: Scala

Lecture 1: Introduction

Daniel Bauer (bauer@cs.columbia.edu)

October 22, 2014

Course Outline

- Lectures:

  Wed 10:10am-12:00pm, 10/22 to 12/3 (7-weeks)
- Instructor:
  - Daniel Bauer (`bauer@cs.columbia.edu`)
  - Office hours: Thu 10:00am-12:00pm, CEPSR/Shapiro 7LW3 (SpeechLab)
- TA:
  - Ming-Ying Chung (`mc3808@columbia.edu`)
  - Office hours: TBD
- Course website (lecture notes, problem sets):

  `http://www.cs.columbia.edu/~bauer/cs3101-2`

Grading / Deliverables

- Class Participation: 5%
- 5 Homeworks: 70%
  - First five weeks. Due following week before class. No late submissions!
  - Small programming tasks.
- Take-home final (last week): 25%

Contents

1 Course Description

2 Getting Started

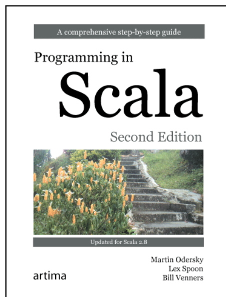
Course Objectives

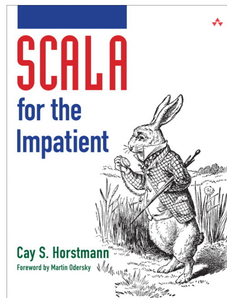
- Get you started using Scala productively.
- Learn some functional programming in Scala.
- Understand the Scala type system.
- Understand why it's called a "Scalable Language".

Tentative Syllabus

Today	Introduction. Running Scala Programs. Basic types. Basic control structures. Functions.
Oct 29	More types (tuples, sets, maps). Classes and objects. Writing applications. Packages and Imports. SBT.
Nov 5	More functional programming. Function literals. Closures. Higher-order functions and methods.
Nov 12	Traits. Case Classes and Pattern Matching. Implicit Conversions. Error Handling.
Nov 19	XML and Domain Specific Languages with parser combinators.
Nov 26	The Lift web framework.
Dec 3	Testing. Concurrent Programming. Scala and Java.

No Official Textbook





Online Materials

- Official Scala Website
  http://www.scala-lang.org/
- Scala API docs
  http://www.scala-lang.org/api/current/index.html
- Scala School (by Twitter)
  http://twitter.github.io/scala_school/

Contents

1 Course Description

2 Getting Started

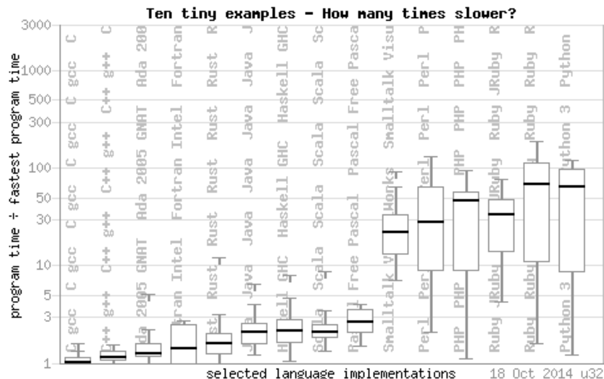
What is Scala?

- General purpose, high-level language.
- Created by Martin Odersky at École Polytechnique Fédérale de Lausanne. First version 2001.
- Uses the Java Virtual Machine (JVM).
- Multi-paradigm, focus on functional and object oriented programming.
- Expressive, static type system.
- Huge language, built on a small core.

Who uses Scala?

- Twitter
- LinkedIn
- Foursquare
- Coursera
- ...
- Used in academia as an alternative to Java

Programming Language Benchmarks Game



- 10 tiny example programs, fastest implemention in each language. Intel 64 bit single core.
- Don't take this too seriously!

source: http://benchmarksgame.alioth.debian.org/u32/which-programs-are-fastest.php

"But Java 8 supports functional programming..."

- Java 8 has very basic support for lambda expressions.
- There are many more features characterizing a functional language
  - easy-to-use container types (lists etc).
  - lots of syntactic sugar to make programs more concise.
- Scala has many other nice features:
  - A type system that makes sense.
  - Traits.
  - Implicit conversions.
  - Pattern Matching.
  - XML literals, Parser combinators, ...

Scala REPL

```
$ scala
Welcome to Scala version 2.11.2.
Type in expressions to have them evaluated.
Type :help for more information.

scala>

scala> 40 + 2
res0: Int = 42

scala> res0 * 2
res1: Int = 84
```

Running Scala Scripts

A script is sequence of statements in a file, interpreted sequentially.

hello.scala:

```
println("Hello, World!")
```

```
$ scala hello.scala
Hello, World!
```

The Java Virtual Machine

- Virtual stack machine that executes Java bytecode (.class files).
- Bytecode is hardware/system independent.
- Provides layer of protection to host machine.
- Built-in garbage collection.
- Just-in-time compilation (bytecode $\rightarrow$ machine code).
  Often as fast as C.
- Many languages:
  - JVM language: Java, Scala, Clojure, Groovy.
  - Other JVM compilers: Python, Ruby, C, Common Lisp, $\cdots$

Compiling and Running Scala Code

`Test.scala` → `scalac` → `Test.class` → `scala`

Hello.scala:

```
object Hello {
    def main(args: Array[String]) = println("Hello, World!");
}
```

```
$ scalac Hello.scala
$ ls
Hello$.class    Hello.class    Hello.scala
$ scala Hello
Hello, World!
```

- Can use `fsc` (compile server) for faster compilation when frequently compiling.
- `scala` is actually a bash script calling `java` (the JVM).

Defining Variables

```
scala> var msg : String = "Hello"
msg: String = Hello, World
```

Defining Variables

```
scala> var msg : String = "Hello"
msg: String = Hello, World

scala> var msg2 = " World"
msg2: String = " World"
```

Defining Variables

```
scala> var msg : String = "Hello"
msg: String = Hello, World

scala> var msg2 = " World"
msg2: String = " World"

scala> msg = "Hi"
msg: String = Hi
```

Defining Variables

```
scala> var msg : String = "Hello"
msg: String = Hello, World

scala> var msg2 = " World"
msg2: String = " World"

scala> msg = "Hi"
msg: String = Hi

scala> msg + msg2
res0: String = Hello World
```

Defining Variables

```
scala> var msg : String = "Hello"
msg: String = Hello, World

scala> var msg2 = " World"
msg2: String = " World"

scala> msg = "Hi"
msg: String = Hi

scala> msg + msg2
res0: String = Hello World

scala> msg = 3
<console>:8: error: type mismatch;
  found    : Int(3)
  required: String
       msg = 3
           ^
```

variables and values

vars can be reassigned.

```
scala> var number  = 2
number: Int = 2

scala> var number  = number + 3
number: Int = 5
```

variables and values

vars can be reassigned.

```
scala> var number  = 2
number: Int = 2

scala> var number  = number + 3
number: Int = 5
```

vals cannot be reassigned once defined.

```
scala> val number  = 42
number: Int = 42

scala> number = 23
<console>:8: error: reassignment to val
       number = 23
              ^
```

When in doubt, try to use `vals` for readability and a more functional programming style.

Integer types

Type	Description
Byte	8-bit signed integer
Short	16-bit signed integer
Int	32-bit signed integer
Long	64-bit signed integer
Char	16bit Unicode character (unsigned)

```
scala> val x = 's'; val y = 0x10
x: Char = s
y: Int = 16
scala> -x + y
res2: Int = -99
```

Floating point types

Float	32-bit single precision float
Double	64-bit single precision float

```
scala> val y = 42E-4; val x = 32.0
y: Double = 0.0042
x: Double = 32.0

scala> x * y
res8: Double = 0.134
```

Strings and Symbols

String	a sequence of Chars.
Symbol	an ‘interned’ String.

```
scala> val hello = "Hello World"
hello: String = Hello World

scala> val color = 'hearts; val value= 'queen
color: Symbol = 'hearts
value: Symbol = 'queen
```

- String is simply an alias for java.lang.String.
- Symbols can often be used in place of enums or global constants.

Java types in Scala

Can use Java types from `java.lang` in Scala.

```
scala> val msg : java.lang.String = "Test"
msg: java.lang.String = Test
```

Booleans and comparisons

Boolean	true or false

```
scala> val x = 5
x: Int = 5

scala> x < 3+2
res3: Boolean = false

scala> val y = "Hello"
y: String = Hello

scala> y == "Hello"
res4: Boolean = true
```

- == tests for value equality, not reference equality.

Conditionals with `if`

if (*condition*) *expression* [else if (*condition*) *expression*] [else *expression*]

- condition is any expression that returns a `Boolean`.

```
val a = 7
if (a % 2 == 0) {
    println("a is even")
} else if (a % 3 == 0) {
    println("a is a multiple of 3")
} else println("none of the above")
```

Everything is an Expression

- There are no statements in Scala.
- Every block of code returns a value. This value can be `Unit`.
- Compound expressions (with {. . .} return result of last expression).
- Type of expression automatically inferred (or can make it explicit).

```
scala> val z : Int = {val x = 4; val y = 2; x / y}
z: Int = 2
scala> val a = {val b = 4; }
a: Unit = ()
```

Conditionals are Expressions too

```
scala> val a = -7;
a: Int = -7

scala> val abs_a = if (a > 0) a else -a
abs_a: Int = 7
```

Conditionals are Expressions too

```
scala> val a = -7;
a: Int = -7

scala> val abs_a = if (a > 0) a else -a
abs_a: Int = 7
```

- What happens if return type is unknown (e.g. missing else)?

```
scala> val z : Boolean = if (42 > 23) true
<console>:7: error: type mismatch;
 found    : Unit
 required: Boolean
       val z : Boolean = if (42 > 23) true
                         ^


scala> val z =  if (42 > 23) true
z: AnyVal = true
```

Loops with while and do ... while

```
scala> var x = 1
scala> while (x <= 5) {println(x); x+= 1}
1
2
3
4
5
scala> x
res1: Int = 6

scala> do {x+=1; println(x);} while (x<=5)
7
```

- While loops usually indicate imperative programing style (manipulate the content of some variable in each step).
- Result type of while is `Unit`.

Example: The Collatz Conjecture

Take any positive natural number n.

- if n is even, set n to $n/2$
- else set n to $3n + 1$

Repeat.

Conjecture: n will become 1 in a finite number of steps.

Example: 15 steps for 23

23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

Example: The Collatz Conjecture

```
var n : Int = 23
var count : Int = 0

println(n)

while (n != 1) {
    count += 1
    if (n % 2 == 0)
        n = n / 2
    else
        n = 3 * n + 1
    println(n)
}

println("Needed "+count.toString+" steps")
```

Defining Functions

$$\text{def } \mathit{name}(\mathit{param}_1\text{: } \mathit{type}_1, \cdots \mathit{param}_n\text{: } \mathit{type}_n) : \mathit{return_type} = \mathit{body}$$

```
scala> def max(x: Int, y: Int) : Int = {
           if (x > y) x
           else y
        }
max: (x: Int, y: Int)Int
```

Return value of a function is the result of the body expression ({} are optional in this case).

Calling functions

```
scala> max(2,3)
res1: Int = 3
```

If a function does not take parameters, do not use parantheses

```
scala> def greet() = println("Hello")
greet: ()Unit

scala> greet
Hello
```

Example: Collatz sequence as a function

```
def collatz (n_param : Int) : Int = {
    var count = 0
    var n = n_param
    println(n)
    while (n != 1) {
        count += 1
        if (n % 2 == 0)
            n = n / 2
        else
            n = 3 * n + 1
        println(n)
    }
    count
}

val steps = collatz(23)
println("Needed "+steps.toString+" steps")
```

Making Collatz recursive

```
def collatz_rec(n : Int) : Int = {
    println(n)

    if (n == 1) 0
    else if (n % 2 == 0)
        1 + collatz_rec(n / 2)
    else
        1 + collatz_rec(3 * n + 1)
}

val steps = collatz_rec(23)
println("Needed "+steps.toString+" steps")
```

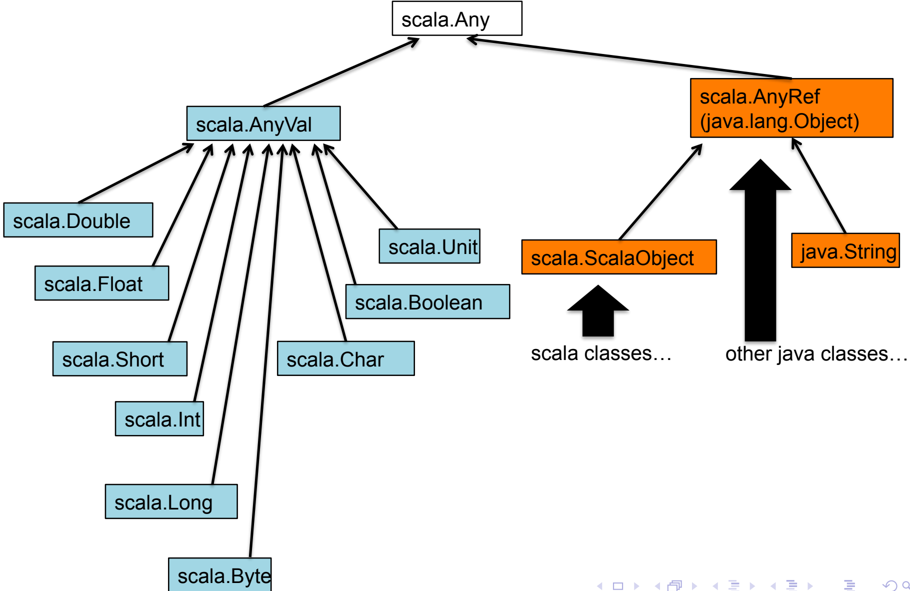
- This is not inefficient because of tail recursion!

All types are classes

- All values are instances of some class.
- This is even true for basic numeric types (unlike Java).
- Can call methods on instances of these classes.

```
scala> 42.toString
res1: String = 42
```

Class hierarchy for basic types



All operators are methods

```
scala> x = 3
x: Int = 3

scala> x.+(2)
res1: Int = 5

scala> x.==(5)
res2: Boolean = true

scala> "fortunate".contains("tuna")
res3: Boolean = true

scala> "fortunate" contains "tuna"
res4: Boolean = true
```

Lists

- Lists are important in any functional language.
- Perform map and reduce/combine operations on elements to produce new lists.
- All elements of a list are of the same type.

```
scala> val l0 = Nil   // the empty list
res1: scala.collection.immutable.Nil.type = List()

scala> val l = 1 :: 2 :: Nil   // :: is pronounced "cons"
l: List[Int] = List(1, 2)

scala> val m = List(3, 4, 5)
m: List[Int] = List(3, 4, 5)

scala> l ::: m
res2: List[Int] = List(1, 2, 3, 4, 5)
```

The 'cons' operator

```
scala> 1 :: List(2,3)
res18: List[Int] = List(1, 2, 3)

scala> List(2,3).::(1)
List[Int] = List(1, 2, 3)
```

- The 'cons' operator seems to behave different from other operators.
- Lists are constructed right-to-left.
- General rule: if an operator ends in : it is translated into a method call on the right operand.

Immutable objects

- immutable objects cannot be changed once created
  - Lists are immutable. :: and ::: create new lists.
  - Strings are also immutable (as in Java).

```
scala> val l = List(1,2,3,4)
l: List[Int] = List(1, 2, 3, 4)

scala> l(3)
res1: Int = 4

scala> l(3) = 100
<console>:9: error: value update is not a member of List[Int]
              l(3) = 100

scala> val a = Array(1,2,3,4)
a: Array[Int] = Array(1, 2, 3, 4)

scala> a(3) = 100

scala> a
res2: Array[Int] = Array(1, 2, 3, 100)
```

Mutable objects

- Arrays are mutable

```
scala> val a = Array(1,2,3,4)
a: Array[Int] = Array(1, 2, 3, 4)

scala> a(3) = 100

scala> a
res2: Array[Int] = Array(1, 2, 3, 100)
```

- Scala defines mutable and immutable versions of many reference types.
- Try to use immutable objects first.

for loops

```
scala> for (y<-List(1,2,3)) {println(y)}
1
2
3
```

- used in this way the result of a for expression is Unit

Range objects

```
scala> 1 to 10    // or 1.to(10)
res1: scala.collection.immutable.Range.Inclusive =
    Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> 10 to (0,-2)
res2: scala.collection.immutable.Range.Inclusive =
    Range(10, 8, 6, 4, 2, 0)

scala> for (i <- 1 to 3) println(i*2);
2
4
6
```

`for` as an expression

for (*seq*) yield *expression*

- *seq* contains at least one generator of the form x <- sequence
- *seq* can contain definitions and filters.

```
scala> for (x <- List(1,2,3)) yield x*2
res8: List[Int] = List(2, 4, 6)

scala> for { x <- 1 to 7 // generator
             y = x % 2;     // definition
             if (y == 0)    // filter
           } yield {
              println(x)
              x
           }
2
4
6
res1: scala.collection.immutable.IndexedSeq[Int] =
    Vector(2, 4, 6)
```

Nesting generators in `for` expression

```
scala> for {x <- List(1,2,3);
              y<-List(4,5))       } yield x * y
res10: List[Int] = List(4, 5, 8, 10, 12, 15)
```