Scala

A Programming Language

Author
Omar Muthanna Ithawi
200811317

Supervisor
Dr. Tareq Jaber

Amman AlAhliyya University

Introduction

Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages, enabling Java and other programmers to be more productive. Code sizes are typically reduced by a factor of two to three when compared to an equivalent Java application.

General Features

- Free and Open Source
  - Licensed under a BSD-style license **SCALA LICENSE**
- Multi Paradigm
  - Imperative
  - Functional Programming
  - Object Oriented
- Typing
  - Static
  - Strong
  - Inferred
  - Structural
- JIT Compilation in two systems:
  - JVM, *Platform In-dependency*
  - and .NET
- Have a parser for interactive development and quick testing
- Extensible
  - This is the source of it's name (**Scala**ble)
  - The possibility of extending the same language or to build very specialist sub-language through the functional features it offers.

History

The design of Scala started in 2001 at the École Polytechnique Fédérale de Lausanne (EPFL) by Martin Odersky, following on from work on Funnel, a programming language combining ideas from functional programming and Petri nets. Odersky had previously worked on Generic Java and javac, Sun's Java compiler. Scala was released late 2003 / early 2004 on the Java platform, and on the .NET platform in June 2004. A second version of the language, v2.0, was released in March 2006.

As of April 2010, the latest release is version 2.7.7. A beta of Scala 2.8 is also available. Features slated for 2.8 include an overhaul of the Scala collections library, named and default parameters for methods, package objects, and continuations.

Programming Features

Pure Functional Programming

The functional programming by itself – according to me – is more scientific, and more suitable for researches than the pure imperative programming languages, and it's makes it easier to implement specific types of tasks (e.g. ML is perfect for image processing), but it has a major drawbacks:

- Less performance on average
- Lack of modularity features, or OOP
- Suitable for complex direct cases
- Requires less coding and loosely typed (Usually)

In so many cases we spend a lot of times writing unnecessary imperative code to perform functions and operations that can be done directly using functional programming languages, due to the facilities that functional programming introduces ease the implementation, and the presence of so many useful functions.

Object-oriented and Imperative Programming

A well known imperative and OOP modern language is Java, with little experience in Java anyone can conclude this kind of languages' features:

- Resource sane consumption
  - A programmer have the control on memory usage and can order a garbage collection
  - It's fast – thanks to JIT – and optimization is always possible
- Modularity, and Clear separation
  - At the class level the code is always clear and divided into methods and the data has one place to be stored in (Object, Class variables).
  - OOP conventional structure offers, and requires clear separation between classes according to the tasks it performs.

Getting the best of the the two

The idea of a multi-paradigm language is to provide a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms. The design goal of such languages is to allow programmers to use the best tool for a job, admitting that no one paradigm solves all problems in the easiest or most efficient way.

Scala took the following paradigms:

- Functional: influenced by Haskel, Pizza, Scheme, and Standard ML
- Imperative, and Object-oriented: Java, and Smalltalk

Technical Specifications

Core Features:

Sample of EBF definitions of Scala:

Identifiers and variable allowed syntax

```
upper            ::=   ‘A’ | · · · | ‘Z’ | ‘$’ | ‘_’ and Unicode category
Lu
lower            ::=   ‘a’ | · · · | ‘z’ and Unicode category Ll
letter           ::=   upper | lower and Unicode categories Lo, Lt, Nl

digit            ::=   ‘0’ | · · · | ‘9’
opchar           ::=   “all other characters in \u0020-007F and Unicode
                     categories Sm, So except parentheses ([]) and periods”
op               ::=   opchar {opchar}
varid            ::=   lower idrest
plainid          ::=   upper idrest
               |   varid
               |   op
id               ::=   plainid
               |   ‘\‘’ stringLit ‘\‘’
idrest           ::=   {letter | digit} [‘_’ op]
```

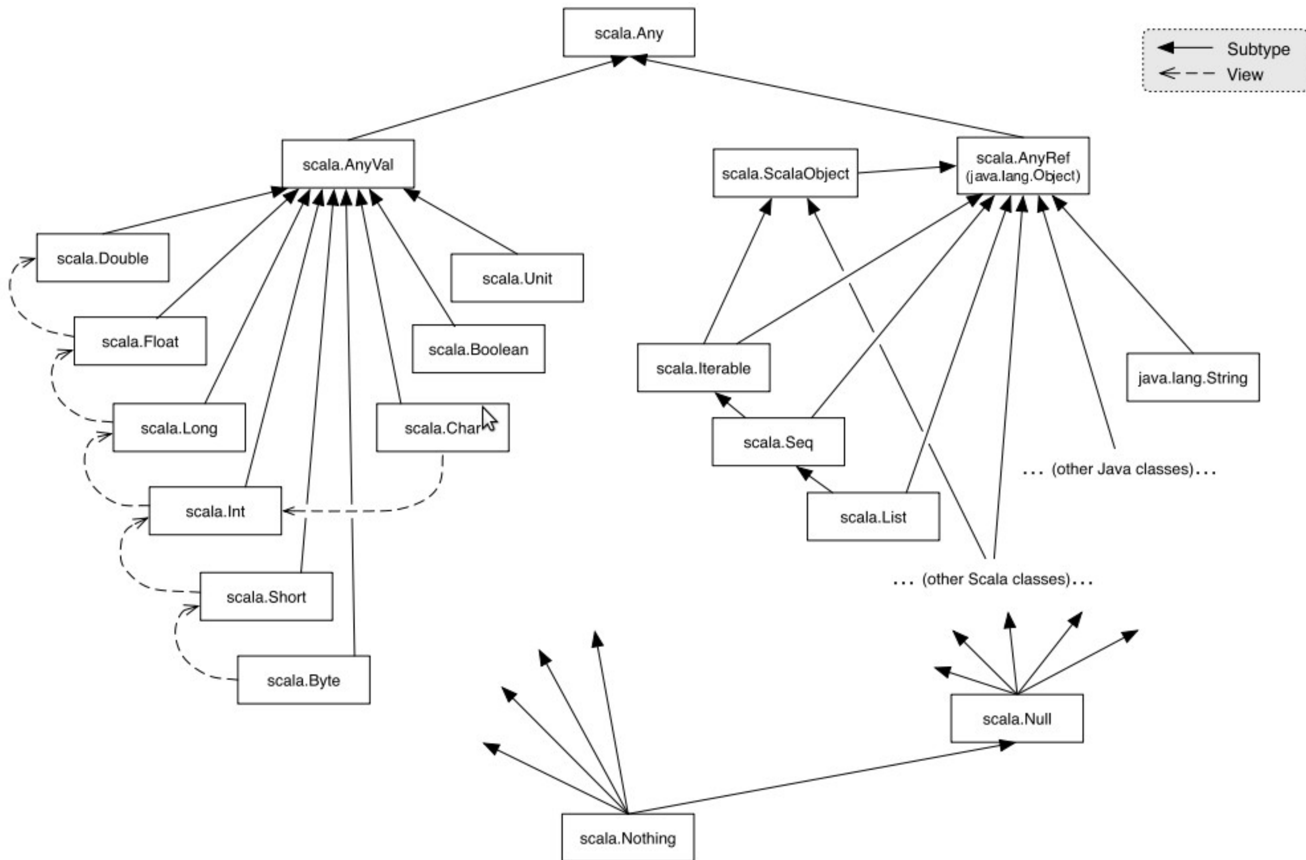
String, and Character literals:

```
stringLiteral  ::= ‘"’ {stringElement} ‘"’
                 | ‘"""’ multiLineChars ‘"""’
stringElement  ::= printableCharNoDoubleQuote
                 | charEscapeSeq
multiLineChars ::= {[‘"’] [‘"’] charNoDoubleQuote}
symbolLiteral  ::= ‘’’ idrest
```

Comments, and Line Ending:

```
comment        ::= ‘/*’ “any sequence of characters” ‘*/’
                   | ‘//’ “any sequence of characters up to end of line”
nl             ::=  “new line character”
semi           ::= ‘;’ |   nl {nl}
```

Class hierarchy:



Noticeable Coding features:

- Definitions start with a reserved word. Function definitions start with **def**, variable definitions start with **var** and definitions of values (i.e. read only variables) start with **val**.
- The declared type of a symbol is given after the symbol and a colon. The declared type can often be omitted, because the compiler can infer it from the context.
- Array types are written **Array[T]** rather than **T[]**, and array selections are written **a(i)** rather than **a[i]**.
- Functions can be nested inside other functions. Nested functions can access parameters and local variables of enclosing functions. For instance, the name of the array **xs** is visible in functions **swap** and **sort1** (see quick sort example below), and therefore need not be passed as a parameter to them.

Code snippets with Scala

Hello, World!

```
object HelloWorld extends Application {
   println("Hello, world!")
}
```

Compared with the Java hello world, we can see how it reduces the effort to write the same softwares that can do whatever Java does but with less effort on the programmer.

Quick sort of an array

The implementation looks quite similar to what one would write in Java or C. We the same operators and similar control structures:

```
def sort(xs: Array[Int]) {
  def swap(i: Int, j: Int) {
    val t = xs(i); xs(i) = xs(j); xs(j) = t
  }
  def sort1(l: Int, r: Int) {
    val pivot = xs((l + r) / 2)
    var i = l; var j = r
    while (i <= j) {
      while (xs(i) < pivot) i += 1
      while (xs(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j)
        i += 1
        j -= 1
      }
    }
    if (l < j) sort1(l, j)
    if (j < r) sort1(i, r)
  }
  sort1(0, xs.length - 1)
}
```

However, it is also possible to write programs in a style which looks completely different. Here is Quicksort again, this time written in functional style.:

```
def sort(xs: Array[Int]): Array[Int] = {
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot >)),
           xs filter (pivot ==),
      sort(xs filter (pivot <)))
  }
}
```