

S C A L A

PROGRAMMARE IN SCALA

Un linguaggio di programmazione moderno



PROGRAMMARE IN SCALA

Copyright © Ugo Landini

versione 0.0.7

Ultimo update: 20/04/2009

Lavoro rilasciato sotto licenza

Commons Creative, Attributions, Non Commercial, Share Alike



<http://creativecommons.org/licenses/by-nc-sa/3.0/>

Pagina lasciata intenzionalmente bianca¹

¹ Non ho mai capito bene a cosa serva, ma senza non dava l'idea di essere un vero libro

Indice

Capitolo 0:	3
Introduzione	3
Chi dovrebbe leggere questo libro	3
Come usare il libro	3
Convenzioni tipografiche	3
Il sito di supporto	4
Indirizzi utili	4
Ringraziamenti	4
Capitolo 1:	5
Scala, il linguaggio	5
Perchè questo libro	5
Perchè Scala	5
Storia di Scala	5
Scala nell'universo dei linguaggi di programmazione	6
Capitolo 2:	10
Primi vagiti sotto la scala	10
Download	10
Installazione su sistemi Unix	10
Installazione su sistemi Microsoft Windows	11
Test dell'ambiente	12
Tool e gestione dei pacchetti	12
Hello, scalable world!	15
Conclusioni	19
Capitolo 3:	21
Scala, una vista ad alto livello	21

Definire variabili	21
Definire funzioni	22
Loop	22
Iterazioni	22
Liste, Set e Map	22
Tuple	22
Conclusioni	23
Glossario	24
Bibliografia	25

Capitolo 0:

Introduzione

Chi dovrebbe leggere questo libro

Le persone interessate ad approfondire il linguaggio Scala e che cercano un linguaggio moderno per realizzare DSL o per scrivere in maniera più semplice server concorrenti ad alte prestazioni.

Come usare il libro

Il libro è stato scritto cercando di raggiungere un pubblico il più possibile vasto. Gli esempi sono stati pensati e scritti su sistemi Unix (OSX o Linux) ma sono stati testati anche su Microsoft Windows.

Il capitolo 1 è una breve introduzione al perchè Scala è stato creato e quali problemi si propone di risolvere.

Il capitolo 2 è sostanzialmente un capitolo di “Hello, world” in cui introdurremo i tool fondamentali di scala

Il capitolo 3 è un tutorial di introduzione al linguaggio ad alto livello. I dettagli saranno lasciati ai successivi capitoli di approfondimento

Convenzioni tipografiche

Per quanto riguarda le convenzioni tipografiche, il seguente è un comando da usare in una shell unix¹

```
$ scala esempio.scala
```

Questa è invece una porzione di codice all'interno della command line shell di scala:

```
scala> println (“Hello, world!”)
```

E questa è una porzione di codice senza riferimenti particolari alla shell di scala o ad altri file:

¹ cmd.exe su Microsoft Windows

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

Questo invece è un box per note speciali

Il sito di supporto

Dal sito di supporto <http://scala.ugolandini.com> potrete scaricare una versione aggiornata di questo libro ed il codice degli esempi.

Indirizzi utili

In questa tabella abbiamo raccolto gli indirizzi più utili per cominciare ad orientarsi nel mondo di Scala.

http://thread.gmane.org/gmane.comp.lang.scala.user	La mailing list ufficiale di scala
http://groups.google.com/group/sug-it	La mailing list italiana dello scala user group italiano
http://www.scala-lang.org	Il sito ufficiale su Scala

Ringraziamenti

In nessun ordine particolare, ringrazio tutte le persone che mi hanno aiutato in qualche modo, segnalando errori o imprecisioni, provando il codice, trovandoci bug, segnalandomi articoli o discutendo in chat con me.

Gabriele Renzi, Franco Lombardi, Stefano Linguetti.

Mi scuso in anticipo con chi avessi dimenticato: giuro, non l'ho fatto apposta.

Capitolo 1:

Scala, il linguaggio

Perchè questo libro

Scala come linguaggio sta cominciando a diffondersi, ed in Italia - come è tipico del nostro paese - c'è il rischio di farsi cogliere impreparati. Questo libro vuole contribuire a creare una solida comunità italiana con un testo che sia un riferimento preciso nella nostra lingua, in modo da minimizzare la barriera d'ingresso: Scala è infatti un linguaggio a paradigma funzionale ed object oriented che richiede un po' di tempo per essere assorbito completamente.

Perchè Scala

Scala sta per **SCAlable LAnguage**, e si pronuncia esattamente come in Italiano. Per una volta, non dovremo impegnarci in improbabili contorsioni linguistiche con i nostri colleghi europei.

Scala è un linguaggio ibrido funzionale/object oriented che fa della scalabilità il suo punto di forza: può essere utilizzato come linguaggio di scripting, come linguaggio collante (glue code) in piccoli progetti o come linguaggio concorrente. Deve questa sua flessibilità ad una combinazione unica di caratteristiche: ha capacità notevoli per la creazione di DSL (Domain Specific Languages, ideali come glue-code) ma è anche in grado di gestire complessi problemi di concorrenza spinta, tipici dell'enterprise.

Storia di Scala

Martin Odersky ha progettato Scala nel 2001 all'EPFL in Svizzera, dove è professore e ricercatore. Martin Odersky, fra le altre cose, ha sviluppato per conto di Sun Microsystems la versione originale di javac, ha creato Pizza¹ - un'estensione a Java - e successivamente GJ² (Generic Java), che è poi stato incluso in Java a partire da JSE 5.0, pur essendo *segretamente* presente anche nella release 1.3 del linguaggio di Sun.³

¹ <http://pizzacompiler.sourceforge.net/>

² <http://www.cis.unisa.edu.au/~pizza/gj/>

³ http://javaposse.com/index.php?post_id=289334

Scala è un linguaggio a paradigma misto: dopo aver progettato Funnel, un linguaggio funzionale di poco interesse al di fuori del mondo accademico, Odersky ha optato per una evoluzione più pragmatica, dando a Scala un'impronta molto diversa. Scala è infatti sia un linguaggio funzionale che object oriented.

Martin Odersky sostiene infatti da sempre che i due paradigmi Object Oriented e Funzionale siano complementari e non in competizione. La tesi di Odersky è che per programmare *bene* con il paradigma ad oggetti ci si deve spingere in qualche modo verso il funzionale.

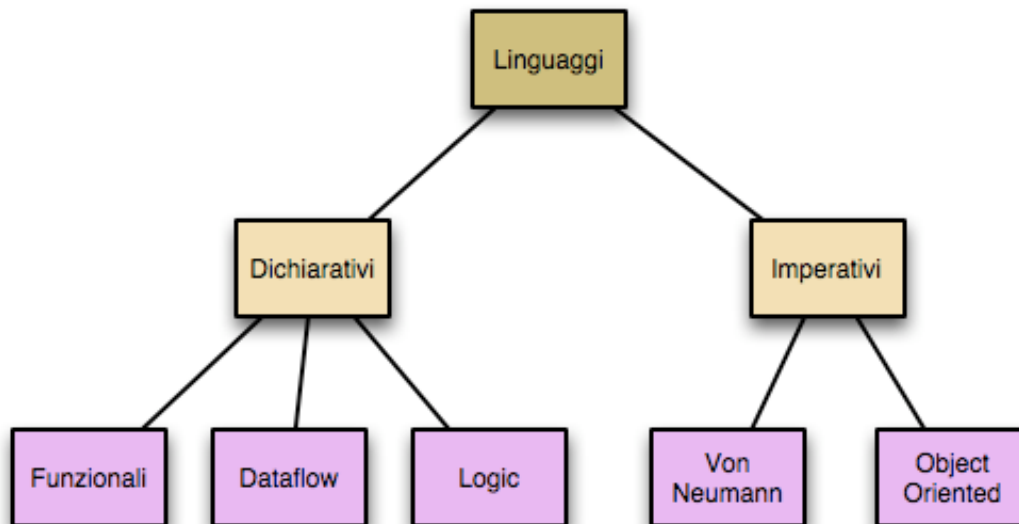
Scala, da questo punto di vista, è una sintesi perfetta del suo pensiero: invoglia ad usare un paradigma funzionale ove conveniente, ma è contemporaneamente più object oriented di Java. Scala sfrutta inoltre le lezioni dei più moderni linguaggi dinamici come ruby o groovy, ma è un linguaggio statico con un occhio particolare alle performance.

Scala ha come principali ragioni di essere:

- sfruttare al massimo le capacità delle moderne architetture multi-core per avere una programmazione concorrente semplice ma anche efficace
- diminuire il codice da scrivere, massimizzando la concisione (cercando di non esagerare e di ottenere qualcosa di potenzialmente criptico, alla Erlang)
- essere in grado di sviluppare con semplicità dei DSL interni

Scala nell'universo dei linguaggi di programmazione

Si parla spesso, e molto spesso a vanvera, di paradigma. Il termine paradigma deriva dal greco, ma nella sua accezione moderna, mutuata dalla filosofia epistemologica, il paradigma è la matrice di conoscenze che delimitano un campo scientifico. E' dunque un modo strutturato di organizzare la conoscenza. Nel campo informatico, e nello specifico nel campo dei linguaggi di programmazione, ci sono diversi modi per catalogare le caratteristiche dei linguaggi di programmazione. [Sco00] definisce due grandi famiglie di linguaggi: *dichiarativi* ed *imperativi*.



I linguaggi *dichiarativi* pongono l'accento su *cosa* deve fare il programma, mentre quelli *imperativi* sul *come* deve farlo. All'interno di queste due grandi famiglie ci sono altre categorizzazioni. I linguaggi *funzionali* sono basati sulla definizione ricorsiva di funzioni, un modello che si poggia sul lambda-calcolo, sviluppato dal matematico americano Alonzo Church negli anni '30. I "classici" linguaggi funzionali sono Lisp/Scheme, ML, Haskell. Linguaggi funzionali più recenti sono Erlang¹, Clojure, ma anche Antlr.

I linguaggi *imperativi* sono quei linguaggi in cui il focus è su *come* il programma deve realizzare un task. Nella famiglia dei linguaggi imperativi ricadono sia i linguaggi strutturati (Von Neumann) che quelli ad oggetti, e dunque si va dal Cobol fino a Java. Linguaggi più recenti e moderni come Ruby² o Groovy sono anch'essi Object Oriented, ma con qualche caratteristica funzionale (blocchi, closures).

I linguaggi imperativi tendono a lavorare modificando valori in locazioni di memoria (*variabili*), mentre i linguaggi funzionali tendono a lavorare con espressioni che definiscono *valori*. Linguaggi ibridi imperativi/funzionali sono stati già creati (CLOS, un'estensione di Common Lisp, ma anche lo stesso Ruby nelle intenzioni dell'autore), ma tipicamente tendono ad essere più imperativi che funzionali. Scala tende invece ad essere molto più bilanciato.

Un'altro modo di catalogare i linguaggi è quello di distinguere i linguaggi *statici* dai linguaggi *dinamici*. Qui la differenza è più di ordine pratico: abbiamo due grandi famiglie di linguaggi, quelle con un sistema di Tipi (Type System) gestito a compile time (statico) e quelli con un sistema

¹ Erlang è del 1987, ma è stato rilasciato open source nel 1998 e solo negli ultimissimi anni ha iniziato a diffondersi al di fuori della sua nicchia iniziale

² Ruby è in realtà coetaneo di Java ma ha iniziato a diffondersi solo dopo il 2000

di tipi gestito a run time (dinamico). Detto in altri termini meno precisi ma più semplici, i primi associano i tipi alle *variabili*, i secondi invece ai *valori*.

Associare i tipi alle variabili consente di trovare molti più errori nel codice (anche se non così tanti come potrebbe sembrare) ma soprattutto permette ai compilatori una serie di ottimizzazioni che allo stato odierno sono impossibili associando i tipi ai valori. D'altronde associando i tipi ai valori è possibile scrivere codice più "naturale" e spesso più conciso: i proponenti dei linguaggi di questo tipo adducono spesso la motivazione di voler smettere di lavorare per il compilatore ("*spoonfeeding the compiler*", letteralmente "*imboccare il compilatore*") per potersi concentrare sulle funzionalità da sviluppare.

Un'altra categorizzazione possibile è fra linguaggi *strongly typed* e *weakly typed*: linguaggi che rispettivamente impediscono completamente errori a runtime sui tipi o ne permettono alcuni, secondo la definizione data da Luca Cardelli in [CAR91]

Avere una catalogazione precisa di tutti i significati con i quali negli anni si è parlato dei sistemi dei tipi è però molto complicato e non cercheremo di dirimere la questione in questi pochi paragrafi¹. Qui la teoria è infatti arrivata a posteriori, e le diverse modellazioni derivano più da concreti problemi implementativi e di design all'interno dei compilatori che da un pensiero uniforme.

Veniamo finalmente a Scala e come si colloca nell'universo dei linguaggi di programmazione: capire questo è il primo passo per cominciare ad apprezzarne le molte qualità: Martin Odersky, nel suo ormai pluriennale sforzo di coniugazione fra paradigmi diversi, ha sviluppato un arsenale di soluzioni eleganti e di compromessi non banali.

Scala è un linguaggio a tipizzazione statica forte (*strongly typed*), ma fa ampio uso di una particolare tecnica - chiamata *type inference* - per controbilanciare la conseguente mancanza di dinamicità. L'effetto finale è che è molto più simile ad un linguaggio dinamico, ma senza averne i contro (soprattutto le performance scarse). Essendo *strongly typed*, non tarderanno ad arrivare - dipenderà dal successo commerciale - editor integrati o plugin per editor integrati con refactoring avanzati ed altre utili caratteristiche. Il supporto dei tool è al momento immaturo ma promettente.

Scala è anche un linguaggio ad oggetti puro, in cui ogni cosa è un oggetto e non esistono eccezioni. Anche le funzioni stesse sono un oggetto, cosa che permette una fusione molto naturale fra i due paradigmi.

Riassumendo, Scala ha queste caratteristiche:

- supporta sia un paradigma dichiarativo (funzionale) che uno imperativo (object oriented)

¹ Ecco a cosa si può arrivare pensandoci troppo <http://www.info.ucl.ac.be/~pvr/paradigms.html>

- dal punto di vista del paradigma object oriented, è puro, come Smalltalk o Ruby: ogni cosa è un oggetto e non esistono tipi base trattati in maniera particolare come in Java
- è scalabile, che nel contesto di un linguaggio di programmazione vuol dire in pratica dover scrivere meno codice per ottenere alte performance e passare *con semplicità* da un piccolo script ad un software di calcolo parallelo.
- ha una libreria per la programmazione concorrente event based, basata sul modello degli attori (mutuato da Erlang). Questo lo include di fatto anche nella categoria dei linguaggi “*concurrent*”
- è conciso ed espressivo
- è costruito su un kernel molto piccolo
- è strongly typed, per cui ha performance superiori di un’ordine di grandezza a linguaggi simili ma dinamici e può avere un forte supporto da parte degli IDE (per refactoring, autocompletamento, ecc.)
- si appoggia alla Virtual Machine di Java ed è portabile sulla maggioranza delle piattaforme, beneficiando di anni di ricerca sull’ottimizzazione dell’esecuzione e della compilazione (JIT) del bytecode. I progressi nel campo della VM di Sun sono automaticamente anche progressi per Scala. Scala può alternativamente sfruttare l’ambiente CLR di Microsoft .Net, ma il progetto principale è Java based¹
- ove non esistano librerie alternative in Scala, si può legare benissimo con le librerie java preesistenti, senza sforzi particolari di integrazione

Si faccia riferimento alle specifiche ufficiali [Spec] del linguaggio per avere tutti i dettagli su Scala.

¹ <http://www.scala-lang.org/node/168>

Capitolo 2:

Primi vagiti sotto la scala

In questo capitolo, prima di addentrarci a fondo nel linguaggio installeremo Scala e lo configureremo per i successivi sviluppi.

Download

La versione di Scala utilizzata in questo libro è la 2.7.3 final. In questo capitolo sono riportate brevemente le opzioni di installazione a disposizione ed alcuni consigli su come impostare l'ambiente, ma è consigliato fare in ogni caso riferimento al sito ufficiale di Scala¹, da dove è possibile effettuare il download del pacchetto corrispondente al nostro Sistema Operativo.

Controllate preventivamente che la versione di java installata sul vostro sistema sia perlomeno la 1.5 prima di proseguire.

Installazione su sistemi Unix

Su un sistema Unix, la soluzione più semplice è usare un gestore di pacchetti (Mac Ports² per OSX e apt-get/yum o analogo per GNU/Linux) ed al prompt di un terminale digitare semplicemente:

```
$ sudo port install scala
```

Terminata l'installazione, digitare semplicemente

```
$ scala
```

per verificare che tutto sia a posto. Dovrebbe apparire una command line shell come in figura

¹ <http://www.scala-lang.org/downloads>

² <http://www.macports.org/>

```
mementinho:~ ugo$ scala
Welcome to Scala version 2.7.3.final (Java HotSpot(TM) Client VM, Java 1.5.0_16).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

In alternativa all'installazione con un gestore di pacchetti basta scaricare il file `scala-2.7.3.final.tar.gz` e scompattarlo con il comando:

```
$ tar xzvf scala-2.7.3.final.tar.gz
```

dopodichè si consiglia di spostare la directory `scala-2.7.3.final` appena ottenuta in un'altra directory, per esempio sotto `/opt/local/share`.

A questo punto è conveniente impostare una variabile d'ambiente `SCALA_HOME` e fare riferimento ad essa, facendola puntare alla directory radice dell'installazione di Scala. Su un sistema Mac OSX o Linux, basta inserire l'opportuna riga di *export* nel file *.profile*: sul mio OS X questo si traduce semplicemente nell'inserimento delle righe seguenti:

```
#Scala
export SCALA_HOME=/opt/local/share/scala
```

Per comodità, aggiungere al `PATH` la directory `$SCALA_HOME/scala-2.7.3.final/bin`: quest'ultimo passo non è necessario se si è usato il gestore di pacchetti, che lo fa automaticamente - di solito creando dei soft link in una directory che è già nel `PATH` di sistema.

Installazione su sistemi Microsoft Windows

Su un sistema Windows bisogna scompattare il file `scala-2.7.3.zip` e poi spostare la directory `scala-2.7.3.final` ottenuta in qualche punto del vostro disco. Dopodichè, impostare la variabile d'ambiente `SCALA_HOME` ed il `PATH`¹ relativo è possibile dall'interfaccia grafica: da "Pannello di controllo" selezionare la voce "Sistema", successivamente la linguetta "Avanzate". Si preme il pulsante "Variabili d'ambiente", quindi il tasto "Nuovo" nella sezione "Variabili di sistema".

Se si preferisce usare la riga di comando, un'altra tecnica consiste nel lanciare uno dei seguenti comandi in un prompt di DOS.

Per impostare una variabile di sistema e dunque visibile a tutti gli utenti:

```
C:/> REG ADD "HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Environment"
/v SCALA_HOME /d c:\scala-2.7.3.final
```

¹ Attenzione che in Windows il separatore è il carattere ";" e non il carattere ":"

Per impostare invece una variabile visibile solo all'utente corrente:

```
C:/> REG ADD "HKCU\Environment" /v SCALA_HOME /d c:\scala-2.7.3.final
```

Con ambedue le istruzioni occorre disconnettersi e riconnettersi al sistema perché le modifiche siano attive.

Anche qui, se dal prompt (cmd.exe) è possibile eseguire

```
C:/>scala
```

vuol dire che l'installazione è a posto.

Test dell'ambiente

A questo punto proviamo il più semplice script possibile, digitando semplicemente 2+2 al prompt di scala:

```
scala> 2+2  
Res0: Int = 4
```

Si otterrà il risultato, 4, assegnato in una variabile temporanea Res0, di tipo Int. Si noti come il tipo, a differenza di Java, è dopo la variabile e non prima. Questo serve per facilitare il parsing (con l'uso dei due punti ":"), poiché la maggior parte delle volte - grazie alla type inference - non ci sarà bisogno di dichiararlo esplicitamente.

Dare il comando **:help** per farsi un'idea delle opzioni a disposizione della shell di Scala, e poi **:quit** per uscire.

Tool e gestione dei pacchetti

I tool forniti "di serie" con il download di scala sono piuttosto spartani e modellati sulla falsariga di quanto si ottiene con il download del JDK di Sun:

scala	La command line shell di scala: può essere usata in modalità interattiva o per eseguire un programma, sia esso interpretato che compilato
scalac	Compila scala in bytecode per la JVM
fsc	Fast Scala Compiler: come scalac, ma lascia un demone in esecuzione su una porta locale in modo da non dover reinizializzare ogni volta Scala (JVM e librerie a contorno). Dopo la prima esecuzione, è sensibilmente più veloce.

scaladoc	L'analogo di javadoc, per generare documentazione a partire da classi scala
sbaz	Scala Bazaar : analogo all'apt-get di Debian, o al port di OSX/Darwin, o al gem di ruby. Permette di gestire (installare, aggiornare, rimuovere) pacchetti aggiuntivi per scala.

Aggiungiamo immediatamente alcuni pacchetti che ci torneranno utili in seguito. Il primo è un pacchetto sviluppato da Bill Venners per lo unit testing:

```
$ sudo sbaz install scala-test
```

I due seguenti comandi¹ invece installano documentazione aggiuntiva varia:

```
$ sudo sbaz install scala-devel-docs
$ sudo sbaz install scala-documentation
```

dopo l'installazione troverete i tutorial ed il resto della documentazione nella directory `$SCALA_HOME/doc`²

Per verificare i pacchetti installati, digitare semplicemente:

```
$ sbaz installed
```

Per i pacchetti disponibili, invece:

```
$ sbaz available
```

Per quanto riguarda la scrittura di codice Scala, nel pacchetto `scala-tool-support` (che dovrebbe essere installato per default: se non lo fosse, ora sapete come fare) si trovano diversi plugin per la maggior parte degli IDE/Editor di codice. Il pacchetto `scala-tool-support` installa nella directory `$SCALA_HOME/misc/scala-tool-support/`

In questo libro saremo agnostici dal punto di vista dei tool e non approfondiremo oltre l'argomento IDE/Editor, argomento che è fra l'altro molto "fluidico" vista la relativa giovinezza del linguaggio. Gli esempi di questo libro sono stati scritti con editor molto semplici come vi e/o text-

¹ è necessario usare `sudo` per via dei permessi, a meno che non si usi un utente root. Su Microsoft Windows digitare i comandi senza anteporre `sudo`, per esempio `C:/>sbaz install scala-devel-docs`

² `%SCALA_HOME%\doc` per Microsoft Windows

mate: un consiglio che ritengo utile è proprio quello di **NON** usare IDE avanzati nella fase iniziale di apprendimento di un qualsiasi linguaggio, poichè si rischia di perdersi qualcosa per strada. I servizi offerti da un IDE - utilissimi - saranno più apprezzabili nelle fasi successive.

Analizziamo brevemente la struttura dell'installazione di scala, che come vedremo è molto semplice:

- **\$SCALA_HOME**

- **bin**
- **doc**
- **lib**
- **meta**
- **misc**
- **src**

Nella directory **bin** ci sono gli eseguibili, compreso una versione di diff per Microsoft Windows (che non lo ha di default come i sistemi Unix). Sono tutti shell script¹ e quindi si possono osservare senza problemi di sorta.

Nella directory **doc** troviamo tutta la parte di documentazione, molto ricca soprattutto se si sono installati i pacchetti addizionali.

Nella directory **lib** troviamo i jar delle librerie di scala. Questi jar saranno utili se si vorrà eseguire i programmi compilati con scalac direttamente da java e senza l'ausilio del comando *scala* (che infatti non è altro che un semplice shell script per gestire in automatico il CLASSPATH)

Nella directory **meta** ci sono le liste dei pacchetti gestiti da *sbaz* con la relativa cache

Nella directory **misc** ci sono altri eventuali pacchetti "esterni", ed altre informazioni usate da *sbaz*. Ad esempio troveremo i file di scala-test (se è stato installato come da paragrafo precedente) ed i file di supporto per gli IDE/Editor di cui abbiamo già parlato.

Nella directory **src** ci sono i sorgenti relativi alla versione in uso.

¹ file .bat per Microsoft Windows

Hello, scalable world!

In nessun libro di programmazione che si rispetti può mancare il famoso “Hello, world”, per cui è arrivato finalmente il momento anche per noi di dedicarci in maniera più approfondita a questo nuovo linguaggio.

Apriamo di nuovo la command line shell di scala.

```
$ scala
```

A questo punto, il primo compito è semplicissimo:

```
scala> println("Hello, scalable world!")  
Hello, scalable world!
```

Notiamo subito l'assenza del “;” finale. In scala è opzionale, a meno che non si scriva più di una istruzione per riga o in altre situazioni che vedremo successivamente. Ovviamente si può inserire l'istruzione un file ed eseguirlo da lì invece che dalla shell: ad esempio, supponendo che il file si chiami hello.scala (il nome non ha importanza¹) e che contenga la stessa istruzione **println** di cui sopra:

```
$ scala hello.scala  
Hello, scalable world!
```

E' ovviamente possibile rendere il file eseguibile, assegnando l'interprete da usare direttamente dentro il file stesso. In un sistema Unix, basta editare il file in questo modo:

```
#!/usr/bin/env scala  
!#  
println("Hello, scalable world!")
```

settare poi gli opportuni permessi di esecuzione :

```
$ chown +x hello.scala
```

ed a questo punto eseguirlo direttamente:

```
$ ./hello.scala  
Hello, scalable world!
```

¹ In teoria, si veda per esempio questo bug <https://lamppsvn.epfl.ch/trac/scala/ticket/1889>

Per ottenere lo stesso con Microsoft Windows basta un double click sul file `.scala` ed associarlo a `scala.bat`. Windows aprirà una finestra DOS, mostrerà il risultato e la chiuderà immediatamente: per far sì che rimanga aperta si può editare un file batch - chiamandolo per esempio `scalaw.bat` - come segue:

```
echo off
cls
call scala %1
pause
```

ed associare ai file `.scala` `scalaw.bat` in luogo di `scala.bat`.

In tutti i casi precedenti abbiamo ovviamente utilizzato l'interprete: proviamo invece ora a compilare il nostro piccolo programma, in modo da farci un'idea più precisa della gestione del codice Scala e della stretta relazione esistente con la JVM

Con il vostro editor preferito, scrivete il seguente programmino:

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, scalable world!")
  }
}
```

Potete chiamarlo come volete, non c'è il vincolo di Java¹ di avere ogni classe pubblica con lo stesso nome del file che la ospita: è buona norma però non approfittare troppo di questa libertà. Il codice è già un po' più conciso dell'equivalente java, ma non è questo l'esempio adatto per misurare il rapporto di "compressione" fra Java e Scala. Notiamo che abbiamo usato la parola chiave **object** e non quella **class**. In Scala, in cui non esistono i metodi statici, **object** definisce un oggetto Singleton, ossia un oggetto di cui non ci potranno essere altre istanze. In questo oggetto è poi definito il metodo `main`, con una sintassi abbastanza autoesplicativa, dove stampiamo il solito messaggio. Notare ancora la dichiarazione di `args` con il ":" ed il tipo che segue.

A questo punto, digitare semplicemente:

```
$ scalac HelloWorld.scala
```

E poi eseguire con

```
$ scala HelloWorld
Hello, scalable world!
```

¹ in realtà è un vincolo del tool `javac` e non del linguaggio Java

Notare che nella directory sono stati creati due file .class, uno per la classe HelloWorld ed uno per una classe anonima, HelloWorld\$. E' quindi possibile eseguire direttamente il programma usando direttamente java¹, basta includere nel CLASSPATH le librerie a runtime di scala:

```
$ java -cp $CLASSPATH:$SCALA_HOME/lib/scala-library.jar HelloWorld
Hello, scalable world!
```

Se doveste ricevere invece dell'output atteso la famigerata eccezione:

```
Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorld
```

E' perchè molto probabilmente non avete il "." nella variabile CLASSPATH. Aggiungetelo esplicitamente alla riga di comando o alla variabile d'ambiente secondo preferenza. Se così non fosse, avete sbagliato il nome della classe.

La libreria scala-library.jar contiene dunque tutto il runtime necessario: nei prossimi capitoli useremo sempre l'eseguibile **scala** per comodità. Notare che è sempre possibile usare **fsc** al posto di **scalac**. Come già accennato precedentemente, l'unica differenza è che **fsc** "prepara" in qualche modo l'ambiente di compilazione una volta per tutte e lascia un demone in ascolto in locale, in modo da velocizzare le successive operazioni di compilazione. Non ci dovrebbero essere altre differenze fra **fsc** e **scalac**, ma qualche issue su programmi che non compilavano correttamente per via di bug in **fsc** in passato c'è stata: se doveste dunque avere strani problemi in fase di compilazione, un controllo con **scalac** per capire se il problema che ci ha fatto impazzire per un'ora è un bug di **fsc** non è una cattiva idea. Per chiudere il demone di **fsc**, digitare semplicemente²

```
fsc -shutdown
```

Dagli esempi precedenti abbiamo imparato che il comando **scala** sa fare diverse cose secondo l'input che riceve:

- Apre una command-line shell scala se senza parametri
- Interpreta uno script scala se lo riceve come parametro
- Esegue un compilato se lo riceve come parametro

¹ su Windows il separatore è il ";" e non il ":". La riga equivalente è dunque:

```
java -cp %CLASSPATH%;%SCALA_HOME%\lib\scala-library.jar HelloWorld
```

² l'opzione -shutdown è stranamente non indicata nell'help di fsc (fsc -h e fsc -X per le opzioni avanzate)

E' importante tenere conto di alcune limitazioni di questo approccio "polimorfo" all'esecuzione di programmi scala: tipicamente ciò che è eseguibile come script non è direttamente compilabile, poiché l'interprete effettua automaticamente un "embed" dello scripting in un oggetto apposito con un main. Se si vuole compilare lo script in bytecode, si dovrà dunque scrivere un object completo (come nell'esempio HelloWorld) ed includere lo script nel main o estendere la classe Application (tecnica che analizzeremo successivamente). Se questo può apparire ovvio, meno ovvio è l'opposto: non tutto ciò che è compilabile con scalac è eseguibile come script, a meno che non si usi un apposito parametro di **scalac**, **-Xscript**, che effettua per noi l'embed in un object. E' quindi possibile compilare il precedente script *hello.scala* senza esplicitare l'embed (ossia senza scrivere l'object con il main), indicando però a **scalac** il nome dell'object da generare.

```
$ scalac -Xscript Hello hello.scala
$ scala Hello
Hello, scalable world!
```

Conclusioni

In questo capitolo abbiamo:

- installato e configurato l'ambiente base di scala
- utilizzato il tool di gestione pacchetti, *sbaz* per aggiungere alcuni pacchetti che ci saranno utili nei prossimi paragrafi e la documentazione aggiuntiva
- Scritto un semplice "Hello world" in modalità shell e compilata
- Osservato come scala generi del normalissimo bytecode per la JVM, che può essere direttamente eseguito da java aggiungendo al CLASSPATH la libreria di supporto al runtime di scala **scala-library.jar**

Capitolo 3:

Scala, una vista ad alto livello

In questo capitolo vedremo dall'“alto” una panoramica più completa delle caratteristiche di Scala. Sarà l'occasione per toccare con mano le feature di questo linguaggio, feature che approfondiremo poi singolarmente nei capitoli successivi.

Per cominciare, apriamo la command line shell di scala.

Definire variabili

Scala ha due tipi di variabili: *val* e *var*. Le variabili *val* sono quelle più in linea con il paradigma funzionale, mentre le *var* sono più simili a quelle a cui ci ha abituato il paradigma imperativo.

Le *val* infatti non possono essere modificate, nel senso che non si può riassegnare un oggetto ad una variabile già inizializzata.

```
scala> val helo = "Hello, scalable world!"
helo: java.lang.String = Hello, scalable world
scala> helo = "ciao"
<console>:5: error: reassignment to val
      helo = "ciao"
        ^
```

Di fatto dunque un *val* è equivalente ad una variabile *final* in java. Verifichiamo invece che con un *var* è possibile riassegnare valori.

```
scala> var helo = "hello"
helo: java.lang.String = hello
scala> helo = "ciao"
helo: java.lang.String = ciao
```

La command line shell permette in effetti di ridefinire dei *val* antepoendo di nuovo *val* o *var* alla variabile, ma è solo una scorciatoia dell'ambiente interattivo!

Notiamo subito che da nessuna parte è stato dichiarato il tipo, esattamente come succederebbe per un linguaggio dinamico. Ma Scala è un linguaggio statico: stiamo vedendo in azione la type inference, che deduce dal contesto il tipo corretto. Qualche volta può essere conveniente esplicitare il tipo, per essere sicuri che Scala stia inferendo correttamente, o semplicemente per dare più informazioni a chi legge il codice (che potrebbe essere piuttosto intricato).

Ecco due modi alternativi di effettuare la stessa operazione, evitando la type inference. Il secondo è possibile perchè le classi `java.lang.*` sono accessibili da Scala senza esplicitare il package, esattamente come in Java

```
scala> val helo1:java.lang.String = "Hello, scalable world!"
helo1: java.lang.String = Hello, scalable world
scala> val helo2:String = "Hello, scalable world!"
helo2: java.lang.String = Hello, scalable world
```

Nella shell è possibile scrivere comandi più lunghi di una riga semplicemente premendo invio e continuando a digitare dopo la fine: apparirà un carattere “|” ad indicare che la riga non è completa. Una doppia battuta di invio (e dunque due blank lines) serve invece per annullare il comando, nel caso ci si renda conto di aver sbagliato qualcosa.

Definire funzioni

Loop

Iterazioni

Liste, Set e Map

Tuple

Conclusioni

In questo capitolo abbiamo:

-

Glossario

A) DSL

Domain Specific Language. E' un linguaggio ad hoc che risolve un problema specifico. Può essere esterno, nel qual caso è realizzato con strumenti "classici" come lex/yacc, javacc o antlr, oppure interno: in questo caso è realizzato sfruttando le peculiarità sintattiche del linguaggio che lo ospita. Per una definizione più estesa, si veda <http://is.gd/sqMg>.

B) Paradigma

Modo di classificare la conoscenza. La matrice di conoscenze che delimitano un campo scientifico.

C) Paradigma Funzionale

Modello computazionale basato sulla definizione ricorsiva di funzioni, basato sul lambda-calcolo, modello di Alonzo Church sviluppato negli anni '30. Classici linguaggi funzionali sono LISP/Scheme, ML, Haskell. Erlang è uno dei linguaggi emergenti in questo panorama. Appartiene alla famiglia dei linguaggi *dichiarativi*, perciò il focus è su *cosa* deve fare il programma.

D) Paradigma Imperativo

E' il paradigma a cui appartengono la maggior parte dei linguaggi più diffusi, qui il focus è su *come* il programma deve realizzare un task. Nella famiglia dei linguaggi imperativi ricadono sia i linguaggi strutturati che quelli ad oggetti, e dunque si va dal Cobol fino a Java.

E) Paradigma concorrente

Appartengono a questo paradigma quei linguaggi specializzati nella risoluzione di problematiche di concorrenza, come Erlang o Clojure (che è una versione "moderna" di Lisp)

Bibliografia

[Blo08] Bloch, Joshua. "Effective Java Second Edition." Addison Wesley, 2008

[Car91] Cardelli, Luca. "Typeful programming". In E. J. Neuhold and M. Paul, editors, Formal Description of Programming Concepts, IFIP State-of-the-Art Reports, pages 431–507. Springer-Verlag, New York, 1991

[Ode01] Odersky, Martin. "Programming in Scala." Artima, 2008

[Sco00] Scott, Michael. "Programming Language Pragmatics", Morgan Kaufmann, 2000

[Spec] Odersky, Martin. "The Scala Language Specification", EPFL.

<http://www.scala-lang.org/docu/files/ScalaReference.pdf>