1 Usare Scala

Vi sono vari modi alternativi di scrivere ed eseguire programmi in Scala:

1. **Interpretazione interattiva**: si digitano interattivamente frammenti di testo Scala nell'interprete REPL (Read-Evaluate-Print Loop), che li esegue.
2. **Interpretazione di script**: programmi in Scala vengono scritti in file di testo con estensione `.sc` che possono essere *interpretati* (in modo simile a quanto avviene per un programma Python).
3. **Compilazione ed esecuzione**: programmi Scala vengono scritti in file con estensione `.scala` e *compilati/eseguiti* (in modo simile a quanto avviene per un programma Java).

1.1 Interpretazione interattiva

L'interprete REPL si fa partire da terminale con il comando `scala`. L'interprete accetta l'inserimento di dichiarazioni, istruzioni ed espressioni Scala, che vengono interpretate non appena si digita invio:

```
$ scala
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM,
Java 1.8.0_51).
Type in expressions to have them evaluated.
Type :help for more information.

scala> println("Hello world")
Hello world

scala> 2+2
res1: Int = 4

scala>
```

Si noti che sia `println("Hello world")` che `2+2` sono espressioni valide in Scala. E' possibile usare il tasto TAB per suggerimenti di autocompletamento mentre si digita.

1.2 Interpretazione di script

Script Scala contengono dichiarazioni, istruzioni ed espressioni Scala. Esempio minimale:

hello.sc

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

Per eseguire il programma è sufficiente darlo in input al comando `scala`:

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

E' possibile eseguire uno script dal REPL usando il comando `:load` seguito dal nome dello script:

```
scala> :load hello.sc
Loading hello.sc...
Hello World!

scala>
```

1.3 Compilazione ed esecuzione

Diversamente da uno script, un programma Scala deve essere maggiormente strutturato, incapsulando il codice in opportuni "contenitori" in modo simile a quanto avviene ad esempio con le classi in Java:

hello.scala

```
object HelloWorld extends App {
    println("Hello World!")
}
```

Per compilare il programma è sufficiente darlo in input al comando `scalac`:

```
$ scalac hello.scala
```

Si noti che la compilazione genera dei file binari `.class` in Java bytecode che possono essere eseguiti su una Java Virtual Machine (JVM):

```
$ ls
HelloWorld$.class
HelloWorld$delayedInit$body.class    hello.scala
HelloWorld.class
$
```

Per eseguire un programma Scala, si può usare il comando `scala` seguito dal nome della classe che contiene il corpo principale del programma:

```
$ scala HelloWorld
Hello World!
$
```

---

2 Elementi di base del linguaggio

2.1 Tipi Primitivi

Scala è un linguaggio tipato come C e Java. I cinque tipi numerici interi base in Scala sono:

Tipo	Valore minimo	Valore massimo
`Long`	-2^{63} = circa -9 miliardi di miliardi	$+2^{63}-1$ = circa 9 miliardi di miliardi
`Int`	-2^{31} = circa -2 miliardi	$+2^{31}-1$ = circa 2 miliardi
`Short`	-2^{15} = -32768	$+2^{15}-1$ = 32767
`Char`	0	$+2^{16}-1$ = 65535
`Byte`	-2^{7} = -128	$+2^{7}-1$ = 127

Si hanno poi i tipi in virgola mobile `Float` (32 bit) e `Double` (64 bit) rappresentati secondo lo standard IEEE 754 e il tipo `Boolean`. Si noti come questi tipi si mappino esattamente su quelli di Java. A questi si aggiunge il tipo `String`, anch'esso derivato da Java. Il tipo `Unit` denota un tipo con il solo valore `()` ed ha alcune analogie con il tipo void in C o Java.

2.2 Letterali

I letterali sono espressioni primitive che possono essere combinate mediante operatori a formare espressioni più complesse. Forniamo una lista di esempi, rimandando a [ScalaRef] per maggiori dettagli.

Tipo	Esempi
`Long`	`10L, 97l, 0x88L` (decimale o esadecimale, terminati con l o L)
`Int`	`10, 0x10, -7, 0xABADCAFE` (decimale o esadecimale)
`Short`	come `Int`, ma limitati all'intervallo [-32768, 32767]
`Char`	`'A', 65, '\u0041'` (carattere unicode)
`Byte`	come `Int`, ma limitati all'intervallo [-256, 255]
`Float`	`3.14f, 3.14F, 1.0e-100f` (terminano con f o F)
`Double`	`3.14, 3.14D, 3.14d, .1`
`String`	`"Hello", "This is a \"nice\" day", """This is a "nice" day"""` (stringhe che possono contenere doppi apici)
`Boolean`	`true, false`
`Unit`	`()`

2.3 Commenti

I commenti possono essere inseriti come in Java su una sola riga usando `//` oppure su più righe usando `/* ... */`.

Esempio.

```
println("hello") // stampa hello
println("world") /* stampa
                    world */
```

2.4 Parole chiave

I seguenti nomi sono parole chiave del linguaggio e non possono essere usati come identificatori:

abstract	case	catch	class	def	do	else	extends
false	final	finally	for	forSome	if	implicit	import
lazy	match	new	null	object	override	package	private
protected	return	sealed	super	this	throw	trait	try
true	type	val	var	while	with	yield	_
:	=	=>	<-	<:	<%	>:	#
@							

2.5 Identificatori

In Scala vi sono vari modi di formare un identificatore:

1. Iniziamo con una lettera (Unicode) o underscore (_) e proseguiamo con una combinazione di lettere e cifre. Esempi: `x`, `nome23`, `nome23xyz`
2. Come al punto 1, ma continuiamo con uno o più underscore (_), seguiti da una sequenza di operatori. Esempi: `somma_+`, `nome_%&*`
3. Sequenza di operatori. Esempi: `++`, `<--->`, `===`, `=!=`, `=%&*`
4. Sequenza di caratteri racchiuse da backtick `. Esempio: `` `il mio identificatore` ``

Non è possibile usare il simbolo `$` negli identificatori e gli **identificatori non possono essere parole chiave**. Operatori ammessi negli identificatori includono i caratteri Unicode in `\u0020-007F` (`! # % & * + - / : < = > ? @ \ ^ | ~`), eccetto le parentesi '`(`', '`[`', '`]`', '`)`' e i punti '`.`'

Esempi. Identificatori validi in Scala:

```
!#%&*+-/:<=>?@\^|~   // tutti gli operatori semplici
simpleName
withDigitsAndUnderscores_ab_12_ab12
wordEndingInOpChars_!#%&*+-/:<=>?@\^|~
!^©®                 // operatori e altri simboli
abcαβγ_!^©®          // misto di caratteri Unicode e simboli
```

Altri esempi di identificatori validi:

`x`	`Object`	`maxIndex`	`p2p`	`empty_?`	`+`
`'yield'`	`λάμβδα`	`_y`	`dot_product_*`	`__system`	`_MAX_LEN_`

2.6 Variabili immutabili

Si dichiarano con la seguente sintassi usando la parola chiave **`val`**:

Sintassi (dichiarazione variabili immutabili)

```
val identificatore : tipo = espressione
```

Nel seguente esempio dichiariamo una variabile immutabile x di tipo intero e le diamo il valore 10:

```
scala> val x:Int = 10
x: Int = 10
scala> x
res0: Int = 10
```

si noti che non è possibile riassegnare una variabile immutabile:

```
scala> x=20
<console>:11: error: reassignment to val
       x=20
        ^
```

2.7 Inferenza di tipo

In Scala è possibile talvolta omettere un tipo se questo può essere dedotto dal contesto.

Esempio. Nel seguente esempio il tipo della variabile viene dedotto dal tipo dell'espressione usata per l'inizializzazione, senza doverlo specificare esplicitamente:

```
scala> val x=20
x: Int = 20
```

2.8 Espressioni

In Scala le espressioni possono essere letterali, chiamate a funzione, oppure ottenute come loro combinazione mediante operatori o costrutti del linguaggio.

2.8.1 Operatori

Gli operatori aritmetici (+, -, *, /, %, ecc.), relazionali (!, ==, !=, ||, &&, <, <= ecc.) e bitwise/bitshift (>>, >>>, |, &, ecc.) in Scala hanno lo stesso significato dei corrispondenti operatori Java. Per una trattazione più approfondita si rimanda al sito [ScalaOperators].

Esempi:

```
scala> ~0                 // complemento a 1
res0: Int = -1
scala> 0xFF ^ 0xF0        // xor
res1: Int = 15
scala> 0xF0 | 0x0F        // or bit a bit
res2: Int = 255
scala> 1 << 3             // shift a sinistra
res3: Int = 8
scala> -1 >> 1            // shift aritmetico a destra (con segno)
res4: Int = -1
scala> -1 >>> 1           // shift logico a destra (senza segno)
res5: Int = 2147483647
scala> 10 > 20            // minore
res6: Boolean = false
scala> 10<20 && 20!=10    // formule booleane
res7: Boolean = true
```

Attenzione: Come abbiamo visto, diversamente da altri linguaggi, **in Scala gli operatori sono identificatori al pari delle sequenze alfanumeriche**. Un'espressione come `x+-y` deve essere quindi scritta come `x+ -y`, oppure come `x+(-y)` altrimenti l'operazione effettuata sarebbe `x +- y`, cercando di applicare un operatore chiamato `+-`.

In Scala qualsiasi entità è un "oggetto", inclusi i valori di tipi primitivi. I consueti operatori come `+`, `*`, `/`, ecc. sugli interi non sono altro che funzioni invocabili come "metodi" di un linguaggio orientato agli oggetti. Sorprendentemente, un'espressione come `2+5` in Scala è del tutto equivalente a `2.+(5)`, cioè l'invocazione del metodo `+` sull'oggetto `2` con parametro `5`.

```
scala> 2.+(5)
res0: Int = 7
```

Per esplorare i metodi disponibili sui vari tipi primitivi (e quindi gli operatori) si può usare l'autocompletamento in REPL:

```
scala> 1.          [premere TAB due volte dopo aver scritto 1.]
!=    +     <=       >>             getClass       toDouble    toString
##    -     <init>   >>>            hashCode       toFloat     unary_+
%     /     ==       ^              isInstanceOf   toInt       unary_-
&     <     >        asInstanceOf   toByte         toLong      unary_~
*     <<    >=       equals         toChar         toShort     |
```

Si noti la presenza di metodi Java come `toString`, `getClass` e `hashCode`. Questo è per garantire piena compatibilità tra Scala e Java.

2.8.2 Espressione if-else

Diversamente da altri linguaggi, in Scala il costrutto if-else non è un'istruzione, ma un'espressione che assume pertanto un valore.

Sintassi (espressione `if` ... `else`)

```
if (espressione) espressione-true else espressione-false
```

dove *`espressione`* è di tipo Boolean e il risultato vale *`espressione-true`* se *`espressione`* è true ed *`espressione-false`* altrimenti[1].

Esempio 1.

```
scala> if (cond) 20 else 30
res1: Int = 20
```

[1] Il costrutto if..else in Scala è del tutto analogo all'operatore ternario `expr ? expr1 : expr2` di C e Java.

2.8.3 Tuple

Le tuple sono espressioni che rappresentano collezioni **immutabili** di valori di tipi arbitrari.

Sintassi (tupla)

```
(espressione1, espressione2, ...)
```

Il **tipo di una tupla** è denotato dalla seguente espressione di tipo:

Sintassi (tipo tupla)

```
(T1,T2, ...)
```

dove `T1`, `T2`, ecc. sono i tipi degli elementi della tupla.

Esempio 1.

```
scala> ("hello", 2.5, 13)
res0: (String, Double, Int) = (hello,2.5,13)
```

Per accedere agli elementi di una tupla `t` è possibile usare la notazione `t._x`, dove `x` è l'indice dell'elemento in `[1,n]` ed `n` è l'arità della tupla `t`.

Esempio 2.

```
scala> val t = ("hello", 2.5, 13)
t: (String, Double, Int) = (hello,2.5,13)

scala> t._2
res0: Double = 2.5
```

E' possibile **assegnare più variabili contemporaneamente** usando tuple come nel seguente esempio.

Esempio 3.

```
scala> val (a,b,c) = ("hello", 2.5, 13)
a: String = hello
```

```
b: Double = 2.5
c: Int = 13
```

E' inoltre possibile confrontare l'**uguaglianza di tuple** usando gli operatori `==` e `!=`.

Come **forma alternativa della coppia** `(a,b)` è possibile scrivere `a->b`.

Esempio 4.

```
scala> 1->2
res0: (Int, Int) = (1,2)
```

2.9 Metodi e funzioni

I metodi si definiscono con la parola chiave **`def`**:

Sintassi (definizione funzione)

```
def identificatore(p1:T1, p2:T2, ...):T = espressione
```

dove `identificatore` è il nome del metofo, `p1, p2,` ecc. sono i parametri di tipo `T1, T2,` ecc., `T` è il tipo restituito, mentre `espressione` è la formula a cui espande la funzione al momento dell'invocazione. Una funzione definita in questo modo ha il seguente tipo Scala:

Sintassi (tipo funzione)

```
(p1:T1, p2:T2, ...) => T
```

Esempio. Le seguenti definizioni della funzione `f` sono tutti equivalenti:

```
def f(x:Int, y:Int):Int = x+y
def f(x:Int, y:Int):Int = { x+y }
def f(x:Int, y:Int):Int = ( x+y )
```

I **parametri** di una funzione sono variabili `val` e sono pertanto **immutabili**.

Esempio 1:

```
def doppio(x:Int) = {
   x=x*2 // ← errore, non è possibile modificare un parametro
   x
}
```

Nella sua forma più semplice, l'**invocazione di una funzione** avviene in modo del tutto analgo ad altri linguaggi:

Sintassi (invocazione funzione)

```
identificatore(espressione1, espressione2,...)
```

dove i parametri attuali passati (espressione1, espressione2, ecc.) sono compatibili in numero e tipo con i corrispondenti parametri formali della funzione (T1, T2, ecc.).

Esempio 2:

```
val p = f(10,20)
println(p) // stampa 30
```

Usando l'**inferenza di tipo**, è possibile in alcuni casi omettere il tipo restituito dalla funzione.

Esempio 3:

```
def f(x:Int, y:Int) = x+y
```

Nel caso in cui l'espressione che definisce la funzione sia composta da sotto-espressioni multiple, il valore restituito è quello dell'ultima espressione:

Esempio 4:

```
def f(x:Int, y:Int) = {
   x*y
   x-y
   x+y
}
val p = f(10,20)
```

```
println(p)            // stampa 30
```

In questo caso, la funzione calcola `x+y` poiché è l'ultima che appare.

2.9.1 Funzioni senza parametri

Una funzione senza parametri può essere **definita e invocata con o senza parentesi**, con l'unico vincolo che una funzione definita senza parentesi non può essere invocata con le parentesi `()`.

Esempio 1. Definizione senza parentesi:

```
scala> def f = 10    // funzione costante definita senza parentesi
f: Int

scala> f             // invocazione f senza parentesi: ok
res0: Int = 10

scala> f()           // invocazione f con parentesi: errore
<console>:13: error: Int does not take parameters
       f()
         ^
```

Esempio 2. Definizione con parentesi:

```
scala> def f() = 10  // funzione costante definita con parentesi
f: ()Int

scala> f             // invocazione f senza parentesi: ok
res0: Int = 10

scala> f()           // invocazione f con parentesi: ok
res1: Int = 10
```

2.9.2 Funzioni `Unit`

Se la funzione non restituisce alcun valore utile (si usa il tipo di ritorno `Unit`), è possibile usare la sintassi alternativa senza l'operatore = nella definizione: