Introduzione alla Programmazione in Scala

Ultimo aggiornamento: 18 novembre 2018

Camil Demetrescu
Dipartimento di Ingegneria Informatica, Automatica e Gestionale "A. Ruberti"
Sapienza Università di Roma

Indice

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 (segna 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 (true) 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 Funzioni

Le funzioni si definiscono con la parola chiave **`def`**:

Sintassi (definizione funzione)

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

dove `identificatore` è il nome della funzione, `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 definizione 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:

Sintassi (definizione funzione `Unit`)

```
def identificatore(p1:T1, p2:T2, ...) { espressione }
```

Esempio:

```
def printInt(x:Int) {
    println(x)
}
```

La forma proposta sopra usa l'**inferenza di tipo**. La forma esplicita specifica il tipo `Unit`:

```
def printInt(x:Int):Unit {
    println(x)
}
```

2.9.3 Funzioni ricorsive

La ricorsione è una tecnica fondamentale dei linguaggi funzionali ed è supportata da Scala.

Esempio. Definiamo una funzione che calcola ricorsivamente il fattoriale di un numero:

```
def fac(n:Int):Int = if (n<=1) 1 else n*fac(n-1)
```

Si noti che le funzioni **ricorsive** richiedono che il tipo restituito sia specificato e **non è possibile usare l'inferenza di tipo**. Poiché ogni chiamata a funzione richiede l'allocazione di spazio in stack per un nuovo frame (record di attivazione della funzione), lo **spazio richiesto da una funzione ricorsiva** è O(h), dove h è il massimo numero di chiamate ricorsive pendenti. La funzione `fac(n:Int)` vista sopra richiede spazio in stack O(n) per mantenere n chiamate ricorsive pendenti.

2.9.4 Ricorsione di coda

Un particolare tipo di ricorsione è la **ricorsione di coda**, che si ha quando la **chiamata ricorsiva è l'ultima operazione ad essere effettuata da una funzione**. La ricorsione di coda è particolarmente utile in quanto può essere automaticamente ottimizzata dal compilatore (tail call optimization, o TCO) generando una formulazione iterativa che utilizza spazio costante in stack.

Nell'esempio sopra del fattoriale, non si ha ricorsione di coda poiché il risultato della chiamata ricorsiva `fac(n-1)` deve essere poi moltiplicato per `n` prima che la funzioni temini. Spesso è possibile riformulare la funzione in modo che esibisca ricorsione di coda, come nella seguente variante della funzione fac.

Esempio.

```
def facIter(f:Int, n:Int):Int = if (n<2) f else facIter(n*f, n-1)
def fac(n:Int) = facIter(1,n)
```

Utilizziamo una funzione ausiliaria ricorsiva che "itera" sui valori n, n-1, n-2, ecc. accumulando in f i successivi valori: n, n*(n-1), n*(n-1)*(n-2), ecc. Quando si raggiunge il passo base, viene restituito il parametro f, che a quel punto vale n!.

Approfondimento. Per essere sicuri che il compilatore effettui la TCO, è possibile annotare la definizione della funzione con `@scala.annotation.tailrec` in modo da avere un errore se la TCO non è stata applicata:

```
// verifica che la funzione seguente abbia ricorsione di coda
@scala.annotation.tailrec
def facIter(f:Int, n:Int):Int = if (n<2) f else facIter(n*f, n-1)
def fac(n:Int) = facIter(1,n)
```

Consideriamo il caso di una funzione con chiamata ricorsiva non in posizione di coda e notiamo l'errore generato dal compilatore:

```
scala> @scala.annotation.tailrec
     | def facnotc(n:Int):Int = if (n<2) 1 else n*facnotc(n-1)
<console>:12: error: could not optimize @tailrec annotated method
facnotc: it contains a recursive call not in tail position
       def facnotc(n:Int):Int = if (n<2) 1 else n*facnotc(n-1)
```

2.9.5 Funzioni annidate

Nel linguaggi funzionali come Scala le funzioni sono "cittadini di serie A" al pari di costrutti come variabili e valori. Una conseguenza di questa visione che glorifica il ruolo delle funzioni è che una funzione può essere dichiarata all'interno di un'altra funzione.

Esempio.

```
def fac(n:Int) = {
    def facIter(f:Int, n:Int):Int = if (n<2) f else facIter(n*f, n-1)
    facIter(1,n)
}
```

Si noti come questo promuova uno stile di programmazione in cui vengono definite **funzioni ausiliarie che rimangono confinate nell'ambito di una funzione che le impiega**. L'uso delle funzioni ausiliarie, quando non indispensabile come nell'esempio del fattoriale, è in genere molto utile per **scomporre un problema complesso in operazioni più semplici**, aumentando **chiarezza, leggibilità e manutenibilità** del codice. Si potrebbe pensare che l'uso diffuso di funzioni ausiliarie renda il codice più lento, ma non è così poiché il compilatore effettua l'**inlining automatico** delle funzioni ove questo porti un beneficio prestazionale.

2.9.6 Funzioni di ordine superiore

Una funzione che **prende come parametro o restituisce una funzione** viene detta **funzione di ordine superiore**. Il supporto per funzioni di ordine superiore è una caratteristica fondamentale dei linguaggi funzionali.

Esempio 1. Scriviamo una funzione generica che stampa un valore trasformato in base a una determinata funzione passata come parametro:

```
def stampa(f: Int=>Int, x:Int) = println(f(x))
```

La funzione può essere invocata ad esempio come segue:

```
def doppio(x:Int) = 2*x
stampa(doppio, 20)        // stampa 40
```

Esempio 2. Vediamo ora una funzione che restituisce un'altra funzione:

```
def ifElse(b:Boolean, f1: Int=>Int, f2: Int=>Int) = if (b) f1 else f2
```

Si noti che il tipo restituito è implicitamente `Int=>Int`.

2.9.7 Funzioni anonime

Le funzioni anonime sono espressioni che denotano funzioni:

Sintassi (funzione anonima)

```
parametri => espressione
```

Esempio 1. La seguente funzione anonima calcola il doppio di un numero:

```
(x:Int) => 2*x
```

La funzione anonima ha come tipo `Int => Int`.

Esempio 2. E' possibile usare una funzione anonima come un qualsiasi valore, quindi assegnandolo a una variabile:

```
scala> val f = (x:Int) => 2*x
f: Int => Int = <function1>
```

Si noti che l'inferenza di tipo ricava per `f` il tipo `Int => Int`. In alternativa, avremmo potuto tipare `f` e omettere il tipo del parametro `x`:

```
scala> val f:Int=>Int = x => 2*x
f: Int => Int = <function1>
```

Esempio 3. Consideriamo nuovamente l'esempio della funzione `stampa` del paragrafo 2.9.6 e vediamo che possiamo usare la funzione anonima `x=>2*x` invece di `doppio`:

```
stampa(x=>2*x, 20) // stampa 40
```

Si noti l'uso dell'inferenza di tipo, per cui non è necessario specificare il tipo del parametro `x` della funzione anonima poiché viene ricavato dal tipo del primo parametro di `stampa`. La forma più verbosa senza inferenza di tipo sarebbe stata:

```
stampa((x:Int)=>2*x, 20) // stampa 40
```

Forma abbreviata. Esiste una forma abbreviata molto compatta per una funzione anonima, che consiste in una espressione in cui appaiono degli underscore "_" che fungono da parametri anonimi, nell'ordine in cui appaiono:

Sintassi forma abbreviata (funzione anonima)
espressione`[_,_,_,...]` è equivalente a `(x,y,z,...) =>` *espressione*`[x,y,z,...]`
espressione`[(_:T1),(_:T2),(_:T3),...]` è equivalente a `(x:T1,y:T2,z:T3,...) =>` *espressione*`[x,y,z,...]`

Esempio 4.

```
val f:(Int,Int,Int)=>Int = _+_*_   // equiv. a (x,y,z)=>x+y*z
println(f(2,3,4)) // stampa 14
```

Oppure, tipando i parametri anonimi:

```
val f = (_:Int)+(_:Int)*(_:Int)    // come (x:Int,y:Int,z:Int)=>x+y*z
println(f(2,3,4)) // stampa 14
```

Esempio 5. Consideriamo nuovamente l'esempio della funzione `stampa` del paragrafo 2.9.6 e vediamo che possiamo usare la forma abbreviata anonima `2*_` invece di `x=>2*x`:

```
stampa(2*_, 20) // stampa 40
```

Caveat. Si noti che _ da solo **non** può essere usato per abbreviare la funzione identità `x=>x`. Scala definisce un metodo predefinito `identity` per questo scopo[2].

Esempio 6.

```
def myfun(f:Int=>Int) = ...
myfun(2*_)          // equivalente a myfun(x=>2*x)
myfun(_)            // equivalente a x=>myfun(x)
myfun(identity)     // equivalente a myfun(x=>x)
```

[2] Anche se è lungo otto caratteri invece dei quattro di `x=>x`, alle volte `identity` è preferito per motivi di leggibilità del codice.

2.9.8 Chiusure

Una **chiusura** (closure) è una **funzione legata alle variabili libere presenti nell'ambiente in cui è definita** secondo le regole di visibilità delle variabili. Le variabili libere dell'ambiente rimangono accessibili per tutta la durata di vita della chiusura e pertanto persistono nel corso di invocazioni successive della chiusura.

Esempio. Cosideriamo la seguente funzione:

```
def somma(a:Int):Int=>Int = {
   def sommaA(b:Int) = a+b   // definisce chiusura sommaA
   sommaA                    // restituisce chiusura sommaA
}
```

L'invocazione della funzione `somma` restituisce una chiusura `sommaA`, che è una funzione legata al parametro `a` con il contenuto che si aveva al momento dell'invocazione di `somma`.

```
val sommaTre = somma(3)    // crea una chiusura
val x = sommaTre(5)        // usa la chiusura
println(x)                 // stampa 8
```

2.9.8 Funzioni parzialmente applicate

Una funzione richiede che tutti i parametri siano specificati. Alle volte può essere utile specificarli solo in parte, ottenendo una **funzione parzialmente applicata**. L'idea è che una funzione con n parametri può essere sempre riscritta come una funzione che prende i primi k parametri e restituisce una funzione (chiusura) degli n-k parametri rimanenti. Questo fornisce maggiore flessibilità nell'uso.

Esempio 1. Abbiamo già visto nel Paragrafo 2.9.8 un esempio di funzione parzialmente applicata:

```
def somma(a:Int):Int=>Int = {
   def sommaA(b:Int) = a+b   // definisce chiusura sommaA
   sommaA                    // restituisce chiusura sommaA
}
```

Questa forma è talmente comune che Scala offre una sintassi ad-hoc per funzioni parzialmente applicate come mostrato sotto:

```
def somma(a:Int)(b:Int) = a+b
```

La funzione può essere chiamata parzialmente. Il seguente esempio mostra una funzione `somma10` che somma 10 al proprio argomento:

```
val somma10:Int=>Int = somma(10)
println(somma10(5)) // stampa 15
```

La funzione somma può essere chiamata specificando tutti i parametri:

```
val x = somma(10)(5)
println(x) // stampa 15
```

2.9.9 Metodi vs. funzioni

Finora abbiamo trattato le funzioni anonime e quelle non anonime come se fossero equivalenti. In realtà, in Scala le funzioni definite con **def** sono chiamate **metodi** e hanno un tipo diverso da quello delle funzioni anonime. Nel seguito, continueremo a chiamare i metodi "funzioni" dove la differenza sia irrilevante.

Esempio 1. Vediamo la differenza di tipo tra medodi e funzioni usando REPL:

```
scala> (x:Int) => x+1
res0: Int => Int = <function1>

scala> def f(x:Int) = x+1
f: (x: Int)Int
```

Nel primo caso (funzione anonima), il tipo della funzione è `Int=>Int`. Nel secondo caso (metodo), il tipo è `(x:Int)Int`.

Un metodo può essere convertito a una funzione in due modi: **automaticamente** o **esplicitamente**.

Conversione automatica da metodo a funzione. Se un metodo viene assegnato a una variabile che ha già il tipo funzione giusto si ha conversione automatica. Questo può avvenire sia quando si passa un metodo come parametro che quando lo si assegna a una variabile già tipata.

Esempio 2. Assegnamento metodo a variabile tipata:

```
def f(x:Int) = x+1
val g:Int=>Int = f  // ok, conversione automatica metodo->funzione
val h = f           // errore, h non è tipata esplicitamente
```

Esempio 3. Passaggio metodo come parametro:

```
def f(x:Int) = x+1
def g(h:Int=>Int, x:Int) = println(h(x))
g(f, 10)            // ok, passaggio metodo f come parametro
```

Esempio 4. Restituzione metodo da funzione:

```
def g():Int=>Int = {
    def f(x:Int) = x+1
    f        // ok, il tipo di ritorno di g (Int=>Int) è esplicito
}
```

Conversione esplicita da metodo a funzione. E' possibile convertire un metodo a una funzione facedolo seguire da un underscore "_".

Esempio 5. Conversione esplicita da metodo a funzione:

```
def f(x:Int) = x+1
val g = f _  // ok, conversione esplicita metodo->funzione
```

Questo vale anche quando si restituisce un metodo da un altro metodo o funzione:

```
def g() = {
    def f(x:Int) = x+1
    f _      // ok, conversione esplicita: non serve tipare g
}
```

2.9.10 Passaggio dei parametri per valore e per nome

In Scala vi sono due forme di passaggio dei parametri: per **valore** (**call-by-value**) e per **nome** (**call-by-name**). In quello per valore, il parametro attuale viene calcolato **prima** del suo passaggio alla funzione, come avviene ad esempio in C e Java. Il passaggio per valore è il default di Scala.

Esempio 1. Passaggio per valore:

```
def p(x:Int) = { println(x); x }
def f(a:Int, b:Int, c:Boolean) = if (c) a else b
f(p(10), p(20), true)  // stampa 10 e 20
f(p(10), p(20), false) // stampa 10 e 20
```

Si noti che entrambe le invocazioni `p(10)` e `p(20)` vengono effettuate **prima** di entrare in `f`, pertanto il programma stampa `10` e `20` come effetto collaterale delle invocazioni di `p`.

Nel passaggio per nome l'argomento passato non viene valutato prima della chiamata, ma solo **al momento dell'uso del parametro nel corpo della funzione**. Il passaggio per nome si realizza premettendo `=>` al tipo del parametro.

Esempio 2. Passaggio per nome:

```
def p(x:Int) = { println(x); x }
def f(a: =>Int, b: =>Int, c:Boolean) = if (c) a else b
f(p(10), p(20), true)  // stampa solo 10
f(p(10), p(20), false) // stampa solo 20
```

In questo caso, la **valutazione del parametro attuale viene ritardata fino al momento in cui il parametro stesso viene usato**. Nell'esempio, solo uno fra `a` e `b` vengono effettivamente valutati in base alla condizione `c`, pertanto una sola fra `p(10)` e `p(20)` verrà invocata. Si noti che lo **spazio** tra `:` e `=>` è necessario, altrimenti il conglomerato di simboli verrebbe considerato come un unico operatore chiamato `:=>`.

Nota bene: un parametro `x: =>T` non contiene un valore di tipo `T`, ma piuttosto un'**espressione** la cui valutazione darà un valore di tipo `T`. In pratica è come se ogni occorrenza di `x` nel corpo della funzione venisse **rimpiazzata dall'intera espressione** (non valutata) che viene passata a `x` dall'esterno.

Il passaggio per nome permette di migliorare in alcuni casi le prestazioni, valutando solo ciò che è effettivamente necessario, e aumenta la flessibilità del linguaggio come vedremo in altri esempi più avanti.

Esempio 3. Il seguente esempio illustra un caso in cui il passaggio per nome comporta una doppia valutazione del parametro, risultando pertanto meno efficiente:

```
def p(x:Int) = { println(x); x }
def f(a: =>Int) = a+a
f(p(10))  // stampa due volte 10
```

Usare il passaggio per nome piuttosto che quello per valore dipende dal contesto e deve essere valutato caso per caso dal programmatore.

2.9.11 Metodi con tipi generici

In Scala è possibile definire metodi che lavorano su tipi generici:

Sintassi (definizione funzione con tipi generici)

```
def identificatore[T1, T2, ...](parametri) { espressione }
```

dove T1, T2, ecc. sono dei **parametri di tipo** che possono apparire nella lista dei parametri formali della funzione e nel suo corpo. Si noti che, diversamente dai metodi, non è possibile definire funzioni anonime con parametri di tipo (si veda [MetodiVsFunzioni] per un approfondimento).

Esempio.

```
def stampa[T](x:T) = { println(x) }
stampa("hello")  // stampa hello (T è String)
stampa(10)       // stampa 10 (T è Int)
```

E' possibile vincolare i tipi generici in vari modi, che non tratteremo in questa dispensa. Notiamo solo che è possibile garantire che il tipo generico fornisca gli operatori relazionali <, <=, >, >= scrivendo `[T <% Ordered[T]]`.

Esempio.

```
def minore[T <% Ordered[T]](x:T, y:T) = x < y
```

```
println(minore(3.14, 6.28))       // true
println(minore("alpha", "bravo")) // true
println(minore("bravo", "alpha")) // false
```

2.9.12 Funzioni con precondizioni

Se una funzione ha delle **precondizioni sui parametri di input**, è buona norma verificarle esplicitamente in modo da generare un errore di esecuzione se la funzione viene invocata su parametri errati. In Scala, un'**asserzione** può essere verificata usando il metodo predefinito `require(condizione)` che lancia un'eccezione se una determinata condizione non è verificata.

Esempio. Verifichiamo che il parametro passato alla funzione fattoriale sia non negativo:

```
def fact(n:Int):Int = {
    require(n>=0)
    if (n==0) 1 else n*fact(n-1)
}
```

Si notino gli esiti delle diverse invocazioni del metodo `fact`:

```
scala> fact(0)   // chiamata con parametro legale
res0: Int = 1

scala> fact(-1)  // chiamata con parametro illegale
java.lang.IllegalArgumentException: requirement failed
  at scala.Predef$.require(Predef.scala:207)
  at .fact(<console>:20)
  ... 33 elided
```

3 Liste immutabili

Le liste sono fra i tipi di dato più usati nei programmi Scala. In questo capitolo vediamo il tipo di dato lista **immutabile**: `List` (`scala.collection.immutable.List` o il suo alias `scala.List`). La documentazione della classe List è in [ClasseList].

3.1 Letterali lista

Un letterale **lista non vuota** è della forma:

```
List(v1,v2,v3,...)
```

dove `v1`, `v2`, `v3`, ecc. sono gli `elementi` contenuti nella lista, nell'ordine in cui appaiono.

Una **lista vuota** può essere scritta come:

```
List() oppure Nil
```

Esempi:

```
val l = List(2,5,6,7) // lista 2, 3, 6, 7
val p = List()        // lista vuota
val q = Nil           // lista vuota
```

3.2 Metodi base

Alcuni dei metodi di prim'ordine (cioè che non prendono come parametri o restituiscono funzioni) più usati di una lista sono elencati nella tabella sotto. Altri metodi sono consultabili in [ClasseList]. **Tutti i metodi di modifica creano una nuova lista senza modificare il loro input**. Negli esempi, facciamo le seguenti assunzioni:

`l = List(2,5,6,7,3)`	`p = List(9,8)`	n = lunghezza della lista

Nella tabella, riportiamo il tempo di esecuzione e lo spazio extra che viene allocato dall'operazione.

Metodo	Descrizione	Esempi	Tempo	Spazio
`length/size`	lunghezza della lista	`l.length == 5` `l.size == 5`	**O(n)**	O(1)
`isEmpty`	la lista è vuota?	`Nil.isEmpty == true` `l.isEmpty == false`	O(1)	O(1)
`head`	elemento in testa	`l.head == 2`	O(1)	O(1)
`tail`	lista a cui è stato tolto l'elemento in testa	`l.tail == List(5,6,7,3)`	O(1)	O(1)
`last`	elemento in coda	`l.last == 3`	**O(n)**	O(1)
`init`	lista a cui è stato tolto l'elemento in coda	`l.init == List(2,5,6,7)`	**O(n)**	O(n)
`::`	inserisce elemento in testa	`5::p == List(5,9,8)`	O(1)	O(1)
`:+`	inserisce elemento in coda	`p:+5 == List(9,8,5)`	**O(n)**	O(n)
`:::`	concatena due liste	`l:::p == List(2,5,6,7,3,9,8)`	**O(l.length)**	O(l.length)
`==`	uguaglianza profonda fra liste	`p == List(9,8)`	O(lunghezza lista più corta)	O(1)
`!=`	disuguaglianza profonda fra liste	`p != l`	O(lunghezza lista più corta)	O(1)
`splitAt k`	spezza lista in un prefisso di k elementi e un suffisso dei restanti	`l splitAt 2 == (List(2,5), List(6,7,3))`	**O(n)**	O(k)
`take k`	prende primi k elementi	`l take 2 == List(2,5)`	O(k)	O(k)
`drop k`	toglie primi k elementi	`l drop 2 == List(6,7,3)`	O(k)	O(1)

`List.range(a,b)`	crea lista numeri da `a` a `b-1`	`List.range(1,5) == List(1,2,3,4)`	O(b-a)	O(b-a)
`reverse`	rovescia la lista	`p.reverse == List(8,9)`	O(n)	O(n)
`(k)`	elemento di indice k in [0,n-1]	`p(1) == 8`	**O(k)**	O(1)
`sum`	somma gli elementi della lista	`l.sum == 23`	O(n)	O(1)
`min/max`	calcola il minimo/massimo della lista	`l.min == 2` `l.max == 7`	O(n)	O(1)
`zip`	combina due liste in una lista di coppie che contiene gli elementi corrispondenti	`l zip p == List((2,9),(5,8))`	O(lunghezza lista più corta)	O(lunghezza lista più corta)
`zipWithIndex`	genera una lista di coppie che contiene ogni elemento della lista insieme al suo indice	`p.zipWithIndex == List((9,0),(8,1))`	O(n)	O(n)

`sliding(a,b)`	elenca le sottoliste di elementi consecutivi di una certa lunghezza `a` con inizio distanziato di un certo passo `b`.	`l.sliding(2,1).toList == List(List(2,5),List(5,6),List(6,7),List(7,3))`
		
		`l.sliding(2,3).toList == List(List(2,5),List(7,3))`
			O(n)	O(n)

`slice(a,b)`	estrae la sottolista di elementi consecutivi dall'indice `a` incluso all'indice `b` escluso	`l.slice(1,4) == List(5,6,7)`	O(n)	O(n)
`contains`	verifica se un elemento appartiene alla lista	`l contains 7 == true` `l contains 4 == false`	**O(n)**	O(1)

`distinct`	crea una nuova lista ottenuta eliminando i duplicati	`List(1,2,1).distinct == List(1,2)`	O(n)	O(n)
`sorted`	crea una nuova lista ottenuta ordinando quella di partenza secondo l'ordinamento naturale dato dagli operatori relazionali <, <=, ecc.	`l.sorted == List(2,3,5,6,7)`	O(n*log n)	O(n)
`flatten`	trasforma una lista di liste nella lista ottenuta concatenando quelle liste	`List(List(1,2),List(3)).flatten == List(1,2,3)`	O(n)	O(n)

Si noti che, se `o` è un oggetto e `m` un suo metodo invocato con parametro `x`, allora `o m x` è equivalente a `o.m(x)`. Ad esempio: `l splitAt 2` è equivalente a `l.splitAt(2)`. Quando invece il metodo è applicato senza parametri, si preferisce non omettere il punto. Esempio: scriviamo `l.length` invece di `l length`.

3.3 Metodi di ordine superiore

I metodi di ordine superiore consentono manipolazioni molto espressive sulle liste. Quelli più usati sono elencati nella tabella seguente. Negli esempi, usiamo la seguente assunzione:

```
l = List(2,5,6,7,3)
```

Tutti i metodi richiedono tempo O(n*costo(f)), dove costo(f) è il tempo richiesto per calcolare la funzione f passata al metodo.

Metodo	Descrizione	Esempi
`count`	conta il numero di elementi che soddisfano un predicato	`l.count(_<6) == 3`
`exists`	verifica se almeno un elemento soddisfa un predicato	`l.exists(_>6) == true` (esiste >6?) `l.exists(_<0) == false` (esiste negativo?)
`filter`	estrai elementi che soddisfano un predicato	`l.filter(_<6) == List(2,5,3)`

`forall`	verifica che tutti gli elementi soddisfino un predicato	`l.forall(_>0) == true` (tutti positivi?) `l.forall(_%2==0) == false` (tutti pari?)
`foreach`	costrutto imperativo: applica funzione con side-effect su ogni elemento, senza restituire nulla[3]	`l foreach println`
`map`	applica funzione a ogni elemento	`l.map(_*2) == List(4,10,12,14,6)`
`reduce`	applica un operatore binario **associativo** su tutti gli elementi della lista	`l.reduce(_+_) == 2+5+6+7+3 == 23`
`foldLeft`	simile a `reduce`, ma permette di partire da un valore definito dall'utente, anche di tipo diverso dagli elementi della lista	`l.foldLeft("Lista:")((s,x)=>s+" "+x) == "Lista: 2 5 6 7 3"`
`takeWhile/ dropWhile`	estrae/elimina il più lungo prefisso della lista i cui elementi soddisfano un predicato	`l.takeWhile(_<6) == List(2,5)` `l.dropWhile(_<6) == List(6,7,3)`
`groupBy`	partiziona gli elementi in gruppi come definito da una data funzione	`List("anno", "roma", "ad") .groupBy(_.head).toList == List(('a',List("anno","ad")),('r',List("roma")))`

`maxBy/ minBy`	calcola l'elemento che fornisce il valore massimo/minimo di una data funzione utente	`List("anno", "padova", "ad").maxBy(_.length) == "padova"`
		
		`List("anno", "padova", "ad").minBy(_.length) == "ad"`

`partition`	partiziona una lista in due sottoliste in base a un predicato	`l.partition(_<6) == (List(2,5,3),List(6,7))`

[3] In realtà, viene restituito l'oggetto `()` di tipo `Unit`, che modella l'assenza di un valore restituito.

3.4 For comprehension

Un modo molto compatto di costruire collezioni di oggetti consiste nell'usare il costrutto `for … yield`, anche chiamato **for comprehension** in Scala. Il costrutto consente di iterare su una o più collezioni, costruire ad ogni iterazione un valore, ed emetterlo in una collezione di output. La struttura del costrutto è:

Sintassi (espressione `for … yield`)

```
for (enumeratore) yield expr    oppure    for {enumeratore} yield expr
```

dove un **enumeratore** è costituito da:

- uno o più **generatori** della forma `variabile <- collezione`
- zero o più **filtri** della forma `if (condizione)`
- zero o più **dichiarazioni di variabili** (introdotte con `val` nelle versioni di scala precedenti alla 2.10)

separati da `;`.

Si noti che il **primo elemento dell'enumeratore** deve essere un generatore, che determina il **tipo della collezione generata**. Se si vuole omettere il separatore `;` scrivendo su più righe, al posto delle parentesi tonde si possono usare le parentesi graffe. I generatori introducono variabili utilizzabili nel corpo del `for` che prendono, uno alla volta, i valori della collezione.

Esempio 1. Un solo generatore:

```
val l = for (i<-List(3,5,9)) yield "x"+i
// l è List("x3", "x5", "x9")
```

Se vi sono **più generatori**, per ogni valore generato dal primo verranno generati tutti i possibili valori del secondo, ecc., come avverrebbe in un ciclo annidato.

Esempio 2. Due generatori:

```
val l = for (i<-1 to 3; j<-1 to 2) yield i+"/"+j
// l è Vector("1/1", "1/2", "2/1", "2/2", "3/1", "3/2")
```

Oppure, equivalentemente usando parentesi graffe ed omettendo i `;`:

```
val l = for {
   i<-1 to 3                // primo generatore
   j<-1 to 2                // secondo generatore
} yield i+"/"+j             // genera 1/1 1/2 2/1 2/2 3/1 3/2
```

Esempio 3. Due generatori con un filtro:

```
val l = for {
   i<-1 to 3                // primo generatore
   j<-1 to 2                // secondo generatore
   if (i>j)                 // filtro
} yield i+"/"+j             // genera 2/1 3/1 3/2
```

Esempio 4. Due generatori con due filtri:

```
val l = for {
   i<-1 to 3             // primo generatore
   if (isOdd(i))         // filtro (i è dispari?)
   j<-1 to 2             // secondo generatore
   if (i>j)              // filtro
} yield i+"/"+j          // genera 3/1 3/2
```

Esempio 5. Due generatori con due filtri e una dichiarazione di variabile:

```
val l = for {
   i<-1 to 3             // primo generatore
    val test = isOdd(i)    // dichiarazione variabile (scala 2.10+ no
val)
   if (test)             // filtro
   j<-1 to 2             // secondo generatore
   if (i>j)              // filtro
} yield i+"/"+j          // genera 3/1 3/2
```

3.5 Esempi

3.5.1 Fattoriale di un numero con `range` e `reduce`

Per illustrare la potenza delle funzioni di ordine superiore, definiamo la funzione fattoriale usando liste:

```
def fact(n:Int) = List.range(1,n+1) reduce (_*_)
```

La funzione viene valutata come segue:

```
fact(4)
→ List.range(1,5) reduce (_*_)
→ List(1,2,3,4) reduce (_*_)
→ 1*2*3*4
→ 24
```

3.5.2 Ordinamento per inserimento

L'**ordinamento per inserimento** è un algoritmo quadratico che consiste nel prendere gli elementi di input uno alla volta e inserirli in modo ordinato in una lista inizialmente vuota. Partiamo dall'operazione base di inserimento in una lista ordinata, che può essere realizzato come segue:

```
def insert(x:Int, l:List[Int]):List[Int] = {
    if (l.isEmpty) List(x)
    else if (x <= l.head) x::l else l.head::insert(x,l.tail)
}
```

Il passo base si ha quando si inserisce in una lista vuota (`l.isEmpty`), che produce una lista che contiene solo `x` (`List(x)`). Se la lista non è vuota, si verifica dapprima se `x` va in testa alla lista (`x <= l.head`). Se è così, viene restituita la lista `l` con `x` in testa (`x::l`). Altrimenti, si inserisce `x` ricorsivamente nel resto della lista (`insert(x,l.tail)`) e gli si appende poi il primo elemento di `l` (`l.head::`).

A questo punto, l'algoritmo di ordinamento per inserimento può essere scritto come:

```
def insertionSort(l:List[Int]):List[Int] = {
    if (l.isEmpty) l
```

```
    else insert(l.head, insertionSort(l.tail))
}
```

Il passo base si ha per la lista da ordinare vuota. Se la lista non è vuota, si ordina ricorsivamente il resto della lista (`insertionSort(l.tail)`) e poi si inserisce il suo primo elemento in posizione ordinata (`insert(l.head, insertionSort(l.tail))`).

Una realizzazione autocontenuta vedrebbe la funzione `insert` annidata nella funzione `insertionSort`.

3.5.3 Ordinamento per fusione

L'**ordinamento per fusione** consiste nel dividere l'input a metà, ordinare ricorsivamente le due metà e poi fonderle in un'unica sequenza ordinata. Il costo dell'algoritmo è O(n log n), asintoticamente ottimo. Partiamo dall'operazione di fusione, di costo lineare:

```
def merge(a:List[Int], b:List[Int]):List[Int] =
        if (a.isEmpty) b
    else if (b.isEmpty) a
    else if (a.head < b.head) a.head :: merge(a.tail, b)
    else                      b.head :: merge(a, b.tail)
```

I passi base si hanno quando una delle due liste da fondere è vuota, per cui il risultato è l'altra lista. Se il primo della lista `a` è minore del primo della lista `b` (`a.head < b.head`), si fonde il resto della lista `a` con `b` (`merge(a.tail, b)`) e poi gli si appende in testa il primo della lista `a` (`a.head ::`). In caso contrario, si fonde il resto della lista `b` con `a` (`merge(a, b.tail)`) e poi gli si appende in testa il primo della lista `b` (`b.head ::`).

L'algoritmo di ordinamento per fusione può essere scritto come segue:

```
def mergeSort(l:List[Int]):List[Int] =
  if (l.length < 2) l
      else   merge(mergeSort(l   take   l.length/2),   mergeSort(l   drop
l.length/2))
```

Il passo base verifica il caso cui non c'è nulla da ordinare (`l.length < 2`). Se la lista contiene più di un elemento, l'algoritmo prima ordina ricorsivamente i primi n/2 elementi (`mergeSort(l take l.length/2)`) e i rimanenti (`mergeSort(l drop l.length/2)`). Infine, li fonde. Si noti come **l'intero algoritmo è ottenuto in stile funzionale come composizione di funzioni**.

Una realizzazione autocontenuta vedrebbe la funzione `merge` annidata nella funzione `mergeSort`.

3.5.4 Ordinamento veloce

L'**ordinamento veloce** (quicksort) consiste nel selezionare un elemento della sequenza da ordinare (il perno, o pivot), partizionare la sequenza in due sequenze contenenti rispettivamente gli elementi minori del perno e quelli maggiori o uguali al perno, ordinarle ricorsivamente, e ricombinarle. Nel nostro esempio, per semplicità prenderemo come perno il primo elemento della sequenza, anche se questo rende l'algoritmo quadratico nel caso peggiore (ad esempio se la sequenza è già ordinata). In questo esempio vediamo anche l'uso dei tipi generici e delle funzioni parzialmente applicate:

```
def qsort[T](cmp:(T,T)=>Int)(l:List[T]):List[T] =
  if (l.length < 2) l
  else qsort(cmp)(l      filter (cmp(_,l.head)< 0)) ::: l.head ::
       qsort(cmp)(l.tail filter (cmp(_,l.head)>=0))
```

Si noti che `cmp(_,l.head)<0` è equivalente alla chiusura `x => cmp(x,l.head)<0`. E' una chiusura perché usa elementi (`cmp` e `l.head`) definiti esternamente alla funzione. La funzione prende come parametro un **comparatore** che, dati due elementi, resituisce un intero negativo se il primo è minore del secondo, zero se sono uguali, e positivo se il primo è maggiore del secondo. La funzione `qsort` restituisce una chiusura che ordina in base al comparatore dato.

Esempio.

```
val cmpF = (a:Float, b:Float) => if (a<b) -1 else if (a>b) 1 else 0
val qsortF = qsort(cmpF) _  // qsortF è chiusura che ordina List[Float]
println( qsortF( List(7.4f,3.2f,4.3f) ) ) // stampa List(3.2, 4.3, 7.4)
```

4 Pattern matching

Il pattern matching è un costrutto molto potente dei linguaggi funzionali. Consente di selezionare un'espressione fra una rosa di possibilità in base alle proprietà di un valore usato per la selezione.

La sintassi generale del costrutto `match` è il seguente:

Sintassi (costrutto `match`)

```
espressione-selezione match {
   case pattern1 [if (condizione1)] => espressione1
   case pattern2 [if (condizione2)] => espressione2
   ...
}
```

Le parti racchiuse da parentesi quadre sono opzionali. Il costrutto `match` fornisce $espressione_k$ se *espressione-selezione* corrisponde a $pattern_k$ e, se è presente la clausola `if`, se $condizione_k$ è soddisfatta.

Si noti che i `case` non devono necessariamente coprire tutti i casi possibili in cui può trovarsi l'espressione, ma il match genera un errore di esecuzione se non esiste un case che corrisponde all'espressione usata per la selezione. **I `case` vengono considerati nell'ordine in cui appaiono.** Diversamente dallo `switch` di C e Java, `match` è un'espressione e non un'istruzione.

4.1 Matching su costanti

La forma più semplice di matching è quello in cui il pattern è una costante, in modo simile a quanto avviene per lo `switch` in linguaggi come C e Java.

Esempio 1.

```
def f(n:Int) =
   n match {
       case 0 => "zero"
       case 1 => "uno"
       case 2 => "due"
       case _ => "non so"
   }
```

```
println(f(0)) // stampa zero
println(f(3)) // stampa non so
```

L'espressione `match` vale la stringa `"zero"` se `n` è `0`, `"uno"` se `n` è `1`, ecc. L'ultimo `case`, che usa come pattern _, cattura i casi rimanenti.

Esempio 2. Questo esempio mostra in matching su stringhe:

```
def f(s:String) =
   s.toLowerCase match {
       case "zero" => 0
       case "uno"  => 1
       case "due"  => 2
   }

println(f("uno")) // stampa 1
println(f("tre")) // errore: java.lang.ExceptionInInitializerError
```

Si noti che il caso `"tre"` non è coperto dal match e la chiamata `f("tre")` genera un'eccezione.

4.2 Matching multiplo su costanti

Nel caso delle costanti, è possibile avere come pattern una **sequenza di opzioni costanti** separate da `|`.

Esempio.

```
def verificaPariDispari(x:Int) =
   x % 10 match {
       case 0 | 2 | 4 | 6 | 8 => x + " è pari"
       case _                 => x + " è dispari"
   }
```

4.3 Matching su variabili

Un pattern può essere una **variabile tipata** che diventa visibile nell'espressione che segue il `=>`. Il case viene selezionato se il valore assunto dall'espressione di selezione ha il tipo specificato e la variabile viene assegnata con il valore dell'espressione di selezione stessa. Se la **variabile del**

pattern non è tipata, allora cattura tutti i casi possibili come abbiamo visto per l'underscore negli esempi sopra. Un case con una variabile non tipata deve apparire come ultimo case.

Esempio 1.

```
def f(x:Any) =
   x match {
       case i:Int    => i + " è un intero"
       case f:Float  => f + " è un float"
       case s:String => s + " è una stringa"
         case nonso     => nonso + " è di un tipo ignoto" // var non
tipata
   }

println(f(10))     // stampa 10 è un intero
println(f("ciao")) // stampa ciao è una stringa
println(f(3.14f))  // stampa 3.14 è un float
println(f(3.14))   // stampa 3.14 è di un tipo ignoto
```

Se ci interessa considerare **solo il tipo dell'espressione di selezione**, ma il **valore non è di interesse**, si può usare come pattern un underscore (_) tipato:

Esempio 2.

```
def f(x:Any) =
   x match {
       case _:Int    => "un intero"
       case _:Float  => "un float"
       case _:String => "una stringa"
       case _        => "tipo ignoto"
   }

println(f("hello")) // una stringa
println(f(10))      // un intero
println(f(3.14))    // tipo ignoto
println(f(3.14f))   // un float
```

4.4 Matching su variabili con condizione

E' possibile effettuare un test arbitrario sulle variabili di un pattern usando la clausola `if`.

Esempio.

```
def f(n:Int) =
   n match {
       case x if (x<0) => "negativo"
       case _          => "non negativo"
   }
```

Si noti che nel primo `case`, `n` viene assegnato alla variabile `x`. Il match con il pattern `x if (x<0)` ha successo solo se `x` è negativo, e allora l'intera espressione vale "negativo". Sarebbe stato corretto anche se avessimo scritto `n if (n<0)`, osservando che l'`n` del case sarebbe una variabile distinta dal parametro `n` della funzione, pur con lo stesso valore.

4.5 Matching su liste

Una forma molto espressiva di **matching strutturale** si ha con le liste, dove è possibile usare come pattern un'espressione della forma `head :: tail` come nell'esempio seguente.

Esempio.

```
def lunghezza(l:List[Any]):Int =
   l match {
       case Nil     => 0
       case h :: t  => 1 + lunghezza(t)   // h==l.head e t==l.tail
   }

println(lunghezza(List(1,5,2,3))) // stampa 4
```

Questo è un esempio misto con un case costante (`Nil`) e un case con pattern strutturale. Si noti che il pattern `h :: t` dichiara due variabili `h` e `t` che assumono rispettivamente il valore `l.head` e `l.tail`. Il pattern `h :: t` ha successo solo se la lista `l` non è vuota. Le variabili `h` e `t` sono visibili nell'espressione a destra del `=>`.

4.6 Matching su tuple

E' possibile fare matching su tuple, dove i pattern considerano simultaneamente tutti gli elementi della tupla.

Esempio. Il seguente esempio mostra come effettuare matching simultaneamente su due liste ordinate per fonderle in un'unica lista ordinata.

```
def fondi(l1: List[Int], l2: List[Int]):List[Int] =
   (l1,l2) match {
       case (Nil, Nil)       => Nil
       case (Nil, l)         => l
       case (l, Nil)         => l
       case (h1::t1, h2::t2) if (h1<h2) => h1::fondi(t1, l2)
       case (h1::t1, h2::t2)            => h2::fondi(l1, t2)
   }

println(fondi(List(5,7,8), List(2,6)))  // stampa List(2, 5, 6, 7, 8)
```

5 Classi

Scala è un linguaggio che combina vari paradigmi di programmazione: funzionale, imperativo e orientato agli oggetti. Le classi sono uno dei concetti base del linguaggio.

5.1 Definizione di classi e costruttore primario

La forma base di una definizione di classe è:

Sintassi (costrutto `class`)

```
class nome-classe [(parametri)] [{ ... }]
```

dove la lista dei parametri e il corpo della classe sono opzionali (racchiusi tra parentesi quadre). Come in Java, per convenzione i nomi delle classi sono identificatori che iniziano per lettera maiuscola. Diversamente da linguaggi come Java, il **corpo della classe `{ }` funge da costruttore primario**. I **parametri del costruttore primario** sono automaticamente **variabili di istanza private** della classe, immutabili di default. Tutte le **variabili dichiarate nel corpo del costruttore primario** diventano variabili di istanza e tutti i metodi definiti nel costruttore primario diventano metodi della classe. Come in Java, nel corpo di una classe:

- la parola chiave `this` denota un riferimento all'oggetto corrente su cui il codice sta lavorando;
- la parola chiave `null` denota un riferimento nullo, che non punta ad alcun oggetto.

Esempio 1.

```
class Punto(x:Double, y:Double) {         // x e y variabili di istanza
    def getX = x           // metodo get per l'attributo x della classe
    def getY = y           // metodo get per l'attributo y della classe
    val dist = Math.sqrt(x*x+y*y)    // distanza del punto dall'origine
    println("creato oggetto Punto("+x+","+y+")") // effetto collaterale
}
```

Si noti che il **costruttore primario** della classe (corpo della classe) definisce due metodi getter (`getX` e `getY`) che accedono ai campi privati `x` e y della classe e dichiara una variabile `dist` inizializzandola con la distanza del punto dall'origine. La variabile `dist` è automaticamente una variabile di istanza pubblica della classe. Infine, il costruttore stampa un messaggio di log.

Nota. Scala è molto meno verboso di Java, riducendo il "boilerplate", cioè la "cerimonia" linguistica necessaria per realizzare un dato compito. Vediamo cosa avremmo dovuto scrivere in Java per definire una classe `Punto` del tutto analoga a quella vista sopra in Scala:

Java

```
class Punto {
    private double x, y;
    public double dist;
    public Punto(double x, double y) {
        this.x = x;
        this.y = y;
        dist = Math.sqrt(x*x+y*y);
        System.out.println("creato oggetto Punto("+x+","+y+")");
    }
    public double getX() { return x; }
    public double getY() { return y; }
}
```

5.2 Creazione di oggetti con `new`

Un oggetto viene creato mediante il costrutto `new`, che invoca il costruttore, eseguendo il corpo della classe, e restituisce un riferimento all'oggetto creato:

Sintassi (costrutto `new`)

```
new nome-classe [(parametri)]
```

La lista dei parametri è vuota se la classe ha un costruttore senza parametri.

Esempio 2. Vediamo come costruire un'istanza di un oggetto della classe `Punto` dell'Esempio 1:

```
val p = new Punto(10, 20)  // stampa creato oggetto Punto(10.0, 20,0)
println(p.getX)            // stampa 10.0
println(p.getY)            // stampa 20.0
println(p.dist)            // stampa 22.360679774997898
```

5.3 Ereditarietà

Una classe può "ereditare" da altre classi variabili e metodi. Come in Java, l'ereditarietà si realizza con il costrutto `extends`:

Sintassi (costrutto `extends`)

```
class nome-classe[(parametri)] extends nome-classe-base[(param)] { ... }
```

Il costrutto definisce una nuova classe *`nome-classe`*, che eredita dalla classe base *`nome-classe-base`*. Il nome della classe base è seguito da una lista opzionale di parametri che vengono passati al suo costruttore. In assenza di `extends` in una dichiarazione di classe, Scala assume implicitamente `extends AnyRef`, dove `AnyRef` è la superclasse di tutte le classi.

Esempio.

```
class Punto3D(x:Double, y:Double, z:Double) extends Punto(x,y) {
    def getZ = z            // x, y, getX, getY e dist ereditati
}
val p = new Punto3D(1,2,3)  // stampa creato oggetto Punto(1.0,2.0)
println(p.getX)             // stampa 1.0
println(p.getY)             // stampa 2.0
println(p.getZ)             // stampa 3.0
```

Si noti come, scrivendo `extends Punto(x,y)`, i parametri `x` e `y` ricevuti dal costruttore di `Punto3D` vengono passati al costruttore della classe base `Punto`, che viene eseguito prima di eseguire il costruttore di `Punto3D`.

5.4 Overriding

E' possibile **ridefinire metodi e variabili** in una sottoclasse usando la parola chiave `override`.

Esempio 1. Override di metodo:

```
class Punto(x:Double, y:Double) {
    override def toString = "("+x+","+y+")"
}
val p = new Punto(10, 20)
println(p) // stampa (10.0,20.0)
```

In questo esempio riferiniamo il metodo `toString` ereditato dalla classe base `AnyRef`.

Esempio 1. Override di variabile:

```
class Punto3D(x:Double, y:Double, z:Double) extends Punto(x,y) {
    def getZ = z
    override val dist = Math.sqrt(x*x+y*y+z*z)
}
```

In questo esempio ridefiniamo la variabile `dist` ereditata dalla superclasse `Punto` in modo che contenga la distanza dall'origine in uno spazio a tre dimensioni invece che due.

5.5 Invocazione metodo superclasse

Un metodo di una superclasse può essere invocato con la parola chiave `super`.

Esempio. Costruttore ausiliario:

```
class A(x:Int) {
    override def toString = "x="+x
}

class B(x:Int, y:Int) extends A(x) {
    override def toString = super.toString + ", y=" + y
}

val p = new B(1,2)
println(p) // stampa: x=1, y=2
```

5.6 Costruttori ausiliari

E' possibile definire costruttori ausiliari oltre a quello primario mediante definendo **metodi con nome `this`**. Nel loro corpo, è possibile **invocare un altro costruttore** della classe sempre con `this(parametri)`. Come in Java, costruttori diversi di una stessa classe devono differire nel numero o tipo dei parametri (**overloading**).

Esempio. Costruttore ausiliario:

```
class Intero(xx:Int) {            // definizione costruttore primario
    def x = xx
    def this(s:String) =          // definizione costruttore ausiliario
        this(Integer.parseInt(s)) // invocazione costruttore primario
}

val i1 = new Intero(10)           // invocazione costruttore primario
val i2 = new Intero("10")         // invocazione costruttore secondario
println(i1.x)                     // stampa 10
println(i2.x)                     // stampa 10
```

5.7 Classi `object`

Diversamente da Java, in Scala tutti i metodi e variabili dichiarati in una classe sono associati a un oggetto e quindi "non statici". Nel caso in cui non sia di interesse avere istanze multiple di una classe, Scala fornisce un costrutto `object`:

Sintassi (costrutto `object`)

```
object nome-classe [{ ... }]
```

Un `object` è semplicemente una classe con una sola istanza automaticamente creata la prima volta che si cerca di accedervi. La sua utilità è nel creare un "name space" per raggruppare metodi e variabili logicamente correlati, aumentando leggibilità e modularità.

I metodi e variabili di un `object` sono equivalenti ai metodi e variabili `static` in Java e si accedono con la sintassi: `nome-classe.nome-metodo` e `nome-classe.nome-variabile`. Non è possibile usare `new` per istanziare un `object`. Il codice del corpo di una classe `object` viene eseguito la prima volta che si accede a un metodo o variabile dell'`object`. E' possibile avere simultaneamente una classe e un `object` con lo stesso nome (**companion object** della classe) che contiene i metodi e le variabili statiche della classe.

Esempio 1. Nell'esempio seguente, l'`object` fornisce un "contenitore" per il metodo `main` da cui viene avviato un programma compilato (come in Java).

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

Da uno script Scala o da REPL, è possibile invocare esplicitamente il metodo `main` con:

```
Hello.main(null) // stampa Hello World
```

Esempio 2. Nell'esempio seguente vediamo un `object` che fornisce uno spazio di nomi per costanti matematiche:

```
object Mat {
    val pi = 3.14159265359
    val e  = 2.71828182846
}

println(Mat.pi) // stampa 3.14159265359
```

5.8 Metodi getter automatici

Se una variabile di istanza non privata viene dichiarata con `val`, Scala genera automaticamente un metodo "getter" per leggere il contenuto della variabile. Diversamente da Java, nel generare metodi getter, Scala usa una **convenzione dei nomi "uniforme", in cui il nome del metodo coincide con quello della variabile**.

Esempio.

```
class Punto(val x:Double, val y:Double)   // classe con corpo vuoto
val p = new Punto(10,20)
println(p.x) // metodo getter x accede al campo x
println(p.y) // metodo getter y accede al campo y
```

Nell'esempio sopra, i parametri del costruttore primario sono dichiarati esplicitamente come **`val`**, e questo implica la generazione automatica dei rispettivi getter.

5.9 Metodi e variabili `private` e `protected`

E' possibile limitare l'accesso a un campo variabile o metodo usando gli specificatori `private` e `protected`:

1. `private`: l'accesso al campo è possibile solo dalla classe in cui appare. La presenza di `private` **inibisce la creazione di un getter e/o setter automatico**, quindi la variabile risulta inaccessibile.
2. `protected`: l'accesso al campo è possibile solo dalla classe o da una sottoclasse della classe in cui appare (diversamente da Java in cui è visibile anche nel package).

Si noti che **non esiste una parola chiave `public`**. L'assenza di `private` o `protected` rende una definizione pubblica di default.

Esempio 1. Variabile di istanza privata senza getter automatico

```
class Punto(x:Double, y:Double) {
    private val dist = Math.sqrt(x*x+y*y) // inaccessibile dall'esterno
    def getDist = dist
}
val p = new Punto(10,20)
println(p.dist)    // errore: dist è privata
println(p.getDist) // ok: metodo getDist è pubblico di default
```

Esempio 2.

```
class Punto(private val x:Double, private val y:Double) {
   def get = (x,y)
}
val p = new Punto(10,20)
println(p.x)    // errore: x è privata
val q = p.get   // ok: metodo get è pubblico di default
println(q)      // stampa (10.0,20.0)
```

5.10 Metodo `apply`

Definendo un metodo chiamato `apply` in una classe, è possible invocarlo direttamente sull'oggetto. Questo fornisce un comodo metodo per accedere in forma compatta al contenuto di un oggetto.

Esempio 1.

```
class ListaInteri(l:List[Int]) {
    def apply(i:Int) = l(i)
```

```
}
val p = new ListaInteri(List(3,5,7,2))
println(p(2)) // stampa 7    [ p(i) è equivalente a p apply i ]
```

Il metodo `apply` può essere usato anche per le classi `object`. In quel caso, si può invocare direttamente sul nome dell'`object`.

Esempio 2.

```
class Punto(val x:Double, val y:Double)
object Punto {          // oggetto "companion" della classe Punto
    def apply(x:Double, y:Double) = new Punto(x,y)
}
val p = Punto(10,20)    // equivalente a val p = new Punto(10,20)
```

Nell'esempio precedente, `Punto(10,20)` invoca il metodo `apply`, che crea un'istanza di oggetto punto e ne restituisce il riferimento. Si noti che in base allo stesso meccanismo scriviamo in forma abbreviata `l = List(1,2,3)` invece di `l = new List(1,2,3)`.

5.11 Classi `case`

Le classi case forniscono un certo numero di automatismi che rendono più flessibile la programmazione. Una classe dichiarata con `case` ha le seguenti proprietà.

1) Non è necessario usare `new` per istanziare un oggetto della classe.

Esempio:

```
case class Punto(x:Double, y:Double)
val p = Punto(10,20)    // equivalente a val p = new Punto(10,20)
```

2) I metodi `equals`, `toString` e `hashCode` sono automaticamente generati per la classe.

Esempio:

```
println(Punto(20,10))                 // stampa Punto(20.0,10.0)
println(Punto(20,10) == Punto(20,10)) // stampa true
println(Punto(20,10) != Punto(20,10)) // stampa false
println(Punto(20,10).hashCode())      // stampa 994070630
```

3) Vengono sempre automaticamente generati metodi getter per gli argomenti del costruttore.

Esempio:

```
val p = Punto(10,20)
println(p.x)                  // stampa 10.0
println(p.y)                  // stampa 20.0
```

4) E' possibile effettuare pattern matching su oggetti di classi case.

Esempio. Mostriamo come definire un albero binario definendo una classe astratta `Albero` e due sottoclassi che modellano un albero non vuoto (`AlberoPieno`), caratterizzato da due sottoalberi e una radice, e uno vuoto (`AlberoVuoto`):

```
sealed abstract class Albero
case class AlberoPieno(sx:Albero, radice:Int, dx:Albero) extends Albero
case class AlberoVuoto() extends Albero
```

La classe `Albero` è dichiarata come **`sealed`**: questo significa che **tutte le sottoclassi** di `Albero` devono essere **definite nello stesso modulo**. In questo modo, è possibile fare dei `match` esaustivi elencando tutti i tipi di classi di un certo tipo. In questo caso, abbiamo solo due casi: albero vuoto e non vuoto. Istanziamo ora un albero:

```
val a = AlberoPieno( AlberoVuoto(),
                     10,
                     AlberoPieno( AlberoVuoto(), 20, AlberoVuoto() ) )
```

L'albero ha radice 10, sottoalbero sinistro vuoto e sottoalbero destro formato da un solo nodo 20. Scriviamo ora un metodo ricorsivo che usa pattern matching per calcolare il numero di nodi di un albero.

```
def numNodi(a:Albero):Int = a match {
    case AlberoPieno(sx,_,dx) => numNodi(sx) + 1 + numNodi(dx)
    case AlberoVuoto() => 0
}
println(numNodi(a)) // stampa 2
```

5.12 Metodi `implicit`

Scala fornisce un meccanismo molto potente per aggiungere funzionalità a classi esistenti senza dover usare il meccanismo dell'eredità e rendere compatibili oggetti di classi diverse che per loro natura non lo sarebbero.

Scenario 1. Consideriamo l'invocazione `x.m(...)` di un metodo `m` su un oggetto `x` di classe `A`. Il compilatore cercherà innanzitutto se il metodo `m` appartiene alla classe `A`. In caso contrario, vedrà se nello scope corrente esiste un metodo dichiarato con `implicit` che realizza una funzione di conversione `A=>B` , dove `B` è una classe che offre il metodo `m` richiesto. Se vi è un unico metodo `conv` che può effettuare una conversione valida ai fini dell'invocazione di `m`, la chiamata `x.m(...)` viene trattata come se fosse `conv(x).m(...)`. Se `conv` invece non è univoco, la conversione fallisce e il compilatore segnala un errore.

Scenario 2. Consideriamo l'uso di un oggetto `x` di un tipo `A` dove ci si aspetta un tipo `B`, ad esempio nel passaggio dei parametri o nell'assegnamento a una variabile tipata. Il compilatore cercherà se nello scope corrente esiste un metodo dichiarato con `implicit` che realizza una funzione di conversione `A=>B`. Se `conv` è il metodo trovato, `x` verrà rimpiazzato nel codice compilato con `conv(x)`, che ha il tipo `B` richiesto.

Esempio 1. Nel seguente esempio si invoca il metodo `somma` su un oggetto `Int`. Questo non è possibile poiché la classe `Int` non contiene un metodo somma.

```
object Prova extends App {
     val a = 10 somma 15 // Errore: somma non è definito nella classe
Int
}
```

Definiamo ora una classe `MioInt` che fornisce il metodo `somma` richiesto e un metodo `implicit` che converte un oggetto `Int` a un oggetto `MioInt`:

```
case class MioInt(val i:Int) {
    def somma(j:Int) = i+j
}

object Prova extends App {
      implicit def intToMioInt(i:Int) = MioInt(i) // converte Int a
MioInt
    val a = 10 somma 15 // 10 viene convertito implicitamente a MioInt
    println(a)          // stampa 25
```

```
}
```

Si noti che, per via della conversione implicita, nell'esempio sopra l'espressione `10 somma 15` è equivalente a `intToMioInt(10).somma(15)`. Per rendere più modulare e riusabile il codice, possiamo spostare il metodo di conversione implicita in uno spazio di nomi definito da un `object`, che per convenienza assumiamo essere il companion object di `MioInt`:

```
object MioInt {  // companion object della classe
      implicit def intToMioInt(i:Int) = MioInt(i) // converte Int a
MioInt
}

case class MioInt(val i:Int) {
    def somma(j:Int) = i+j
}
```

A questo punto, possiamo importare il metodo contenuto nell'object `MioInt` con una `import`, attivando così la conversione implicita da `Int` a `MioInt`:

```
import MioInt._  // importa nello scope il contenuto dell'object MioInt

object Prova extends App {
    val a = 10 somma 15 // 10 viene convertito implicitamente a MioInt
    println(a)          // stampa 25
}
```

Nell'esempio sopra, avremmo potuto usare più specificamente **`import`** `MioInt.intToMioInt`.

Esempio 2. Vediamo ora come realizzare una conversione automatica da `MioInt` a `Int`. Come prima cosa aggiungiamo all'object `MioInt` un metodo di conversione implicita:

```
object MioInt {  // companion object della classe
      implicit def intToMioInt(i:Int) = MioInt(i) // converte Int a
MioInt
     implicit def mioIntToInt(mi:MioInt) = mi.i  // converte MioInt a
Int
}

case class MioInt(val i:Int) {
    def somma(j:Int) = i+j
```

```
}
```

E' ora possibile usare la conversione implicita per assegnare un oggetto `MioInt` a una variabile tipata con `Int`:

```
import MioInt._  // importa nello scope il contenuto dell'object MioInt

object Prova extends App {
    val a:Int = MioInt(10)  // conversione automatica MioInt -> Int
    println(a)              // stampa 10
}
```

6 Gerarchia delle classi Scala

6.1 Struttura generale

La gerarchia delle classi Scala è organizzata come illustrato in Figura 6.1.

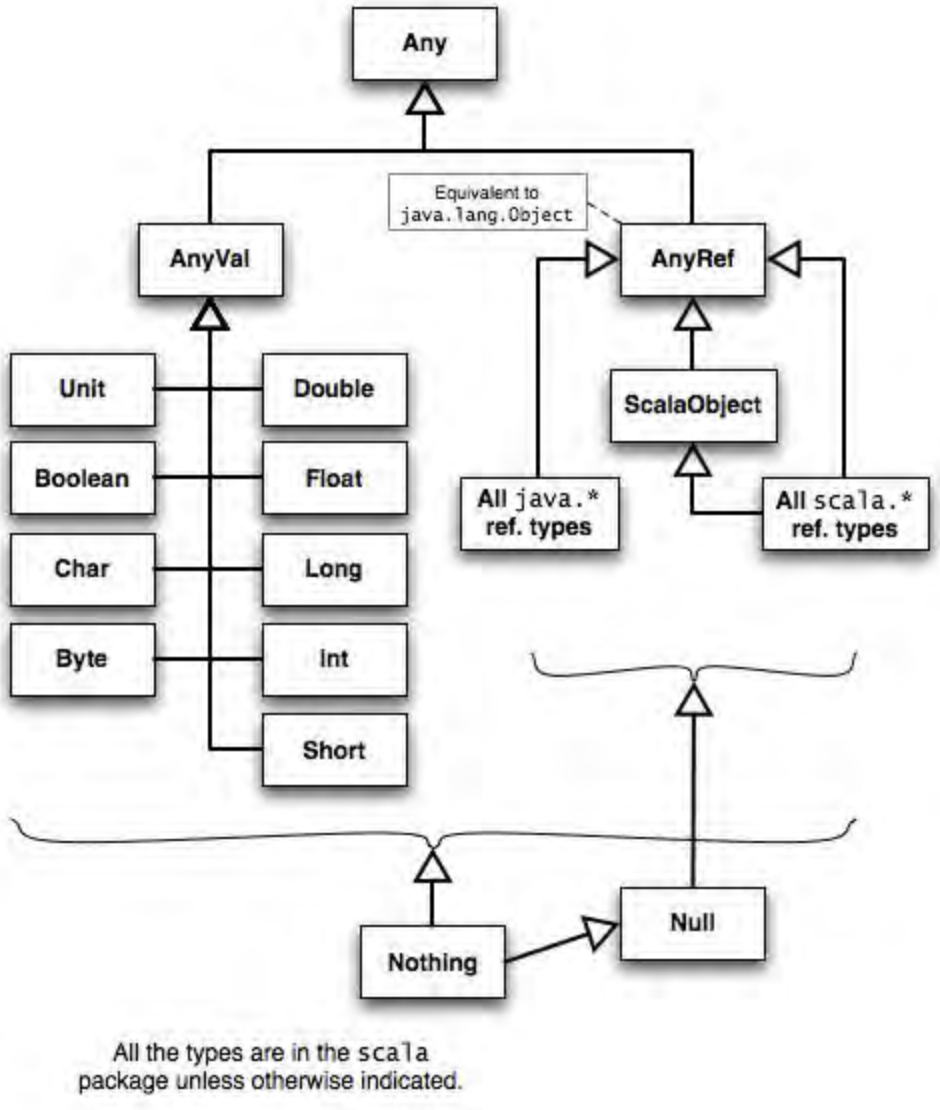

Figura 6.1: panoramica della gerarchica delle classi Scala (Immagine © O'Reilly Media, Inc.)

La superclasse di ogni classe è `scala.Any`, da cui si diramano due rami principali: `scala.AnyVal` è il capostipite di tutte le classi che rappresentano tipi primitivi (`scala.Int`, `scala.Short`, ecc.), mentre sotto `scala.AnyRef` si sviluppano le classi della libreria standard Scala e quelle Java.

Diversamente da Java, esiste anche una sottoclasse comune a tutte le classi: `scala.Nothing`, che non ha tuttavia alcuna istanza. A chiudere in basso il ramo `AnyVal`, c'è inoltre `scala.Null`, che ha un'unica istanza denotata dalla parola chiave `null`.

Un caso interessante che richiede la comprensione della gerarchia delle classi Scala sono le espressioni come `if (test) a else b` in cui `a` e `b` hanno tipo diverso. In questo caso, il tipo risultante sarà quello della superclasse di `a` e di `b` più in basso nella gerarchia.

Esempio.

```
scala> if (true) 10 else ()  // AnyVal è il minimo antenato comune
res0: AnyVal = 10            // di Int e Unit
```

Come in Java, le classi Scala sono organizzate in package. Alcuni package sono inclusi automaticamente: `java.lang`, `scala` e `scala.Predef`. Quest'ultimo fornisce un certo numero di alias per classi e metodi comunemente usati. Ad esempio: `List` è un alias per `scala.collection.immutable.List`. Anche il metodo `println` è incluso nel package `scala.Predef`.

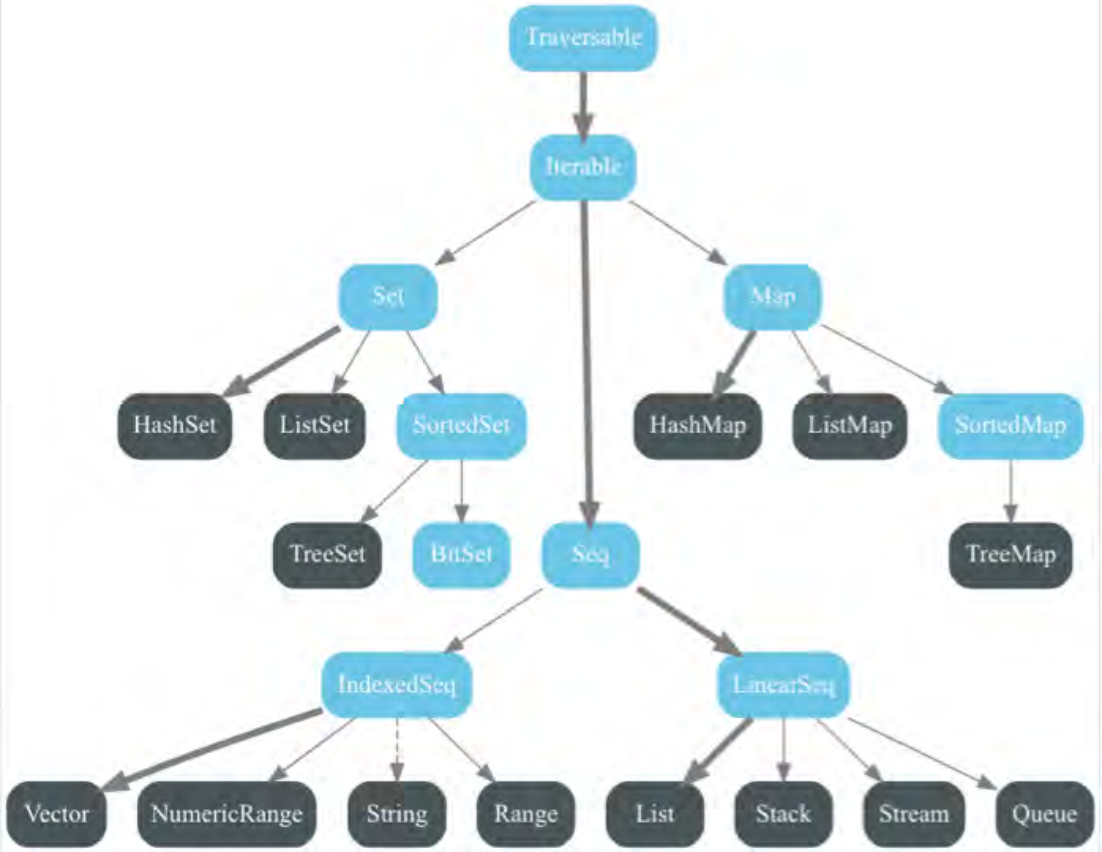

Figura 6.2: collezioni immutabili (immagine © docs.scala-lang.org).

6.1 Collezioni immutabili

Scala fornisce un'articolata struttura di classi per rappresentare collezioni. Quelle immutabili sono organizzate nel package `scala.collection.immutable`[4] e sono illustrate in Figura 6.2.

Il capostipite delle classi collezione è `Traversable`, che offre la maggior parte dei metodi visti per le liste, come: `map`, `reduce`, `filter`, `exists`, ecc. I tre rami principali suddividono le collezioni in: insiemi (`scala.Set`), sequenze (`scala.Seq`) e dizionari (`scala.Map`). I nodi in celeste rappresentano `trait` (analoghi alle interfacce in Java 8), mentre quelli grigio scuro classi concrete. Le linee continue denotano derivazione di classi, mentre quelle tratteggiate conversione implicita. Ad esempio, oggetti della classe `String` vengono convertiti implicitamente a oggetti `IndexedSeq` per poter applicare metodi come `map`, `filter`, ecc.

Esempio 1. Applicazione dei metodi delle collezioni su stringhe:

```
scala> "hello world".count(_=='o')
res0: Int = 2
```

Le linee spesse denotano implementazione di default. Ad esempio, l'implementazione di default di `Traversable` è `List`.

Esempio 2. Implementazione di default di un trait:

```
scala> Traversable(1,4,2)
res1: Traversable[Int] = List(1, 4, 2)
```

6.1.1 Seq

Il trait `Seq` modella collezioni che rappresentano sequenze e si divide in due sottotrait: `IndexedSeq`, che modella rappresentazioni indicizzate (basate su array), e `LinearSeq`, che modella rappresentazioni collegate (basate su liste di nodi). Fra le sottoclassi concrete di `IndexedSeq`, vi è `Vector` e fra quelle di `LinearSeq` vi è `List`.

Esempio. Implementazione di default di `IndexedSeq` e `LinearSeq`:

[4] Si veda la documentazione: https://www.scala-lang.org/api/current/scala/collection/immutable/.

```
scala> IndexedSeq(1,4,2)
res1: IndexedSeq[Int] = Vector(1, 4, 2)

scala> scala.collection.immutable.LinearSeq(1,4,2)
res2: scala.collection.immutable.LinearSeq[Int] = List(1, 4, 2)
```

6.1.2 Set

Il trait `Set` modella insiemi, ovvero collezioni senza duplicati e senza una posizione definita per i loro elementi. I set supportano il test di appartenenza in modo molto efficiente, usando alberi bilanciati o tavole hash.

Esempio. Operazioni su insiemi:

```
scala> val s = Set(1,4,2,1,3,2)
s: scala.collection.immutable.Set[Int] = Set(1, 4, 2, 3)

scala> s(0) ← il metodo apply verifica se un elemento appartiene all'insieme
res1: Boolean = false

scala> s(1)
res1: Boolean = true
```

6.1.3 Map

Le mappe sono dizionari che indicizzano coppie `(chiave,valore)`, consentendo query per chiave efficienti. Come i set, sono rappresentati usando alberi bilanciati o tavole hash.

Esempio 1. Definizione e query di mappe:

```
scala> val m = Map((1,"uno"), (2,"due"), (3,"tre"))
m: scala.collection.immutable.Map[Int,String] = Map(1 -> uno, 2 -> due,
3 -> tre)

scala> m(2)
res0: String = due

scala> m(5)
```

```
java.util.NoSuchElementException: key not found: 5
  at scala.collection.MapLike$class.default(MapLike.scala:228)
  at scala.collection.AbstractMap.default(Map.scala:59)
  at scala.collection.MapLike$class.apply(MapLike.scala:141)
  at scala.collection.AbstractMap.apply(Map.scala:59)
  ... 33 elided

scala> m.keys
res1: Iterable[Int] = Set(1, 2, 3)
```

Si noti che una coppia `(a,b)` in Scala può anche essere rappresentata con la notazione `a->b`.

Esempio 2. Aggiunta/rimozione di coppie (chiave,valore):

```
scala> val a = m + (4->"quattro")     ← aggiunta di coppia (chiave,valore)
a: scala.collection.immutable.Map[Int,String] = Map(1 -> uno, 2 -> due,
3 -> tre, 4 -> quattro)

scala> val b = m - 1                  ← rimozione di chiave
b: scala.collection.immutable.Map[Int,String] = Map(2 -> due, 3 -> tre)
```

6.2 Stream

Un caso particolare di `LinearSeq` sono gli `Stream`, che permettono di modellare in forma implicita collezioni potenzialmente infinite. Una caratteristica particolare degli stream è che i suoi elementi sono calcolati on-demand solo quando vengono acceduti. Gli stream sono simili alle liste, tranne che per costruirli si usa l'operatore `#::` e non `::`. La concatenazione di stream è possibile mediante l'operatore `#:::`. Lo stream vuoto è denotato da `Stream.empty`.

Esempio 1. Costruzione di stream finiti:

```
scala> val s1 = 1 #:: 2 #:: 3 #:: Stream.empty
s1: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> val s2 = Stream(1,2,3)
s2: scala.collection.immutable.Stream[Int] = Stream(1, ?)
```

Usando definizioni ricorsive, è possibile definire stream infiniti:

Esempio 2. Esempio dei numeri pari:

```
scala> def pari(n:Int):Stream[Int] = n #:: pari(n+2)
pari: (n: Int)Stream[Int]

scala> val p = pari(0)
p: Stream[Int] = Stream(0, ?)

scala> p.take(10).toList
res0: List[Int] = List(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
```

Si noti la insolita formulazione ricorsiva **senza passo base**.

Esempio 4. Esempio dei numeri di Fibonacci:

```
scala> def fib(a:Int, b:Int):Stream[Int] = a #:: fib(b, a+b)
fib: (a: Int, b: Int)Stream[Int]

scala> val f = fib(1,1)
f: Stream[Int] = Stream(1, ?)

scala> f.take(10).toList
res0: List[Int] = List(1, 1, 2, 3, 5, 8, 13, 21, 34, 55)

scala> f(8)
res1: Int = 34
```

Un aspetto sorprendente degli stream è che pipeline di operatori su collezioni continuano a produrre stream i cui elementi vengono generati **solo quando richiesti**:

Esempio 5. Pipeline di trasformazioni su stream:

```
scala> fib(1,1).zipWithIndex.map(t =>"F"+t._2+"="+t._1)
res2: scala.collection.immutable.Stream[String] = Stream(F0=1, ?)

scala> fib(1,1).zipWithIndex.map(t =>"F"+t._2+"="+t._1).take(5)
res3: scala.collection.immutable.Stream[String] = Stream(F0=1, ?)

scala> fib(1,1).zipWithIndex.map(t =>"F"+t._2+"="+t._1).take(5).toList
res4: List[String] = List(F0=1, F1=1, F2=2, F3=3, F4=5)
```

6.3 Option

Per gestire il caso in cui una computazione possa non essere in grado di produrre un valore valido, Scala offre la classe generica `Option[T]` con le due sottoclassi `Some[T]`, che modella valori concreti, e `None`, che modella l'assenza di valore.

Ad esempio, il metodo sulle collezioni `find` ha come intestazione:

```
def find(p: T=>Boolean):Option[T]
```

dove `T` è il tipo generico degli elementi della collezione. Il metodo restituisce un oggetto `Some(x)`, dove `x`, se esiste, è il primo elemento della collezione che soddisfa il predicato `p`, e `None` altrimenti.

Per analizzare in modo robusto un valore `Option` si può usare il pattern matching oppure il metodo `getOrElse` come nel seguente esempio.

Esempio:

```
val res = List(1,3,5,2).find(_>2)  // cerca primo elemento > 2

res match {
    case Some(x) => println(x) // stampa 3
    case None => println("Elemento non trovato")
}

// oppure, in alternativa
println(res getOrElse "Elemento non trovato")
```

7 Vantaggi dello stile funzionale

7.1 Funzioni di ordine superiore

Vediamo come l'uso di funzioni di ordine superiore aumenti la modularità e la riusabilità del codice. Partiamo da alcuni semplici esempi.

6.1.1 Aumentare la riusabilità del codice

Esempio 1. la somma dei **numeri** interi compresi tra a e b:

```
def sommaInt(a:Int, b:Int):Int = if (a>b) 0 else a+sommaInt(a+1,b)
```

Esempio 2. la somma dei **quadrati** dei numeri interi compresi tra a e b:

```
def sommaQuad(a:Int, b:Int):Int = if (a>b) 0 else a*a+sommaQuad(a+1,b)
```

Esempio 3. la somma delle **potenze** 2^n dei numeri interi n compresi tra a e b:

```
def pot2(n:Int):Int = if (n==0) 1 else 2 * pot2(n-1)
def sommaPot2(a:Int, b:Int):Int = if (a>b) 0 else pot2(a)+sommaPot2(a+1,b)
```

Esempio 4.

Osserviamo come le tre soluzioni viste sopra risolvano problemi diversi usando lo stesso schema di somme di valori a cui viene applicata una funzione. Possiamo **fattorizzare** lo schema in un'unica funzione di ordine superiore che prende come parametro la funzione `f` da applicare ai singoli numeri, evitando la duplicazione di codice:

```
def somma(f:Int=>Int, a:Int, b:Int):Int = if (a>b) 0 else f(a)+somma(f,a+1,b)
```

A questo punto, possiamo ridefinire le funzioni `sommaInt`, `sommaQuad` e `sommaPot2` come segue:

```
def sommaInt(a:Int, b:Int)  = somma(x=>x, a, b)
def sommaQuad(a:Int, b:Int) = somma(x=>x*x, a, b)
```

```
def sommaPot2(a:Int, b:Int) = somma(pot2, a, b)
```

Esempio 5.

Vediamo come migliorare ulteriormente l'esempio usando una funzione parzialmente applicata che restituisce una chiusura:

```
def somma(f:Int=>Int): (Int, Int)=>Int = {
  def sommaF(a:Int, b:Int):Int = if (a>b) 0 else f(a)+sommaF(a+1,b)
  sommaF
}
```

In questo modo, la funzione `somma` prende in input una funzione `f` e restituisce una chiusura `sommaF` che prende due interi `a` e `b` e calcola la somma di `f(x)` per ogni `x` in `[a,b]`. Si noti che la chiusura lessicale incorpora nella funzione `sommaF` il parametro `f` usato al momento della sua definizione.

Usando questa definizione, possiamo definire in modo molto compatto le funzioni `sommaInt`, `sommaQuad` e `sommaPot2` come segue:

```
def sommaInt  = somma(x=>x)
def sommaQuad = somma(x=>x*x)
def sommaPot2 = somma(pot2)
```

In alternativa, la funzione `somma` può essere usata direttamente, come nel seguente esempio:

```
scala> somma(x=>x*x)(1,3)
res0: Int = 14
```

7.2 Tuple

Le tuple sono uno strumento flessibile. Un aspetto che le rende molto utili è la possibilità di usarle nel caso in cui una funzione debba calcolare simultaneamente più valori.

Esempio 1. Funzione che calcola simultaneamente il minimo e il massimo di due elementi:

```
def minMax(a:Int, b:Int) = if (a<b) (a,b) else (b,a)
val (min,max) = minMax(20, 15)
println(min + " " + max) // stampa 15 20
```

7.3 Pattern Matching

Uno degli usi più utili del pattern matching è nell'ambito delle gerarchie di classi. E' noto come il "downcasting", cioè il cast di un riferimento a un oggetto di una superclasse a uno di una sottoclasse, sia in generale un'operazione pericolosa, perché potrebbe generare un'eccezione nel caso in cui l'oggetto riferito non sia effettivamente del tipo della sottoclasse. Il pattern matching permette di gestire il downcasting in modo sicuro ed elegante.

Esempio.

```
abstract class Forma
class Punto(val x:Double, val y:Double) extends Forma
class Cerchio(val x:Double, val y:Double, val r:Double) extends Forma

def disegna(f:Forma) =
    f match {
        case p:Punto   => disegnaPunto(p)
        case c:Cerchio => disegnaCerchio(c)
        case _         => println("forma non riconosciuta")
    }
```

8 Programmazione imperativa in Scala

Nei capitoli precedenti di questa dispensa tutti gli esempi visti sono stati effettuati usando uno stile funzionale puro, in cui nessuna funzione effettua side-effect sulle strutture dati, tutte immutabili. Come unica eccezione, abbiamo usato operazioni che fanno side-effect sull'ambiente come `println` per stampare i risultati delle computazioni.

Programmare in stile funzionale puro, sebbene sia la prima scelta da considerare per i suoi benefici in termini di astrazione e modularità, può essere poco pratico per determinati compiti. Scala permette di inserire "impurità" in un programma, come l'uso di strutture dati mutabili e costrutti della programmazione imperativa come `while`, lasciandone l'uso al buon senso del programmatore.

8.1 Variabili mutabili

La forma più semplice di struttura dati mutabile è la variabile. Le **variabili mutabili** in Scala si dichiarano usando la parola chiave **`var`** con la seguente sintassi:

Sintassi (dichiarazione variabili mutabili)

```
var identificatore : tipo = espressione
```

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

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

si noti che è possibile riassegnare una variabile mutabile:

```
scala> x=20
x: Int = 20
scala> x
res1: Int = 20
```

8.2 Metodo `update`

Definendo un metodo chiamato `update` in una classe diventa possible modificare il contenuto di un oggetto con l'operatore `=`. Se `x` è un riferimento a un oggetto di una classe che ha definito il metodo `update`, si ha la seguente equivalenza:

Sintassi (metodo update)

```
x(a,b,c,...) = z        è equivalente a      x.update(a,b,c,...,z)
```

Si noti che l'ultimo parametro di `update` è il valore che viene messo a destra dell'assegnamento.

Esempio.

```
class ArrayInteri(v:Array[Int]) {
    def apply(i:Int) = v(i)
    def update(i:Int,x:Int) = v(i) = x
}
val p = new ArrayInteri(Array(3,5,7,2))
println(p(2)) // stampa 7    [ p(i) è equivalente a p.apply(i) ]
p(2) = 10     //             [ p(i) = x è equivalente a p.update(i,x)]
println(p(2)) // stampa 10
```

8.3 Costrutti della programmazione imperativa

8.3.1 while

Scala fornisce il classico costrutto `while` imperativo, con la consueta semantica.

Esempio. Vediamo come calcolare il fattoriale di un numero in uno stile imperativo:

```
def fattoriale(n:Int) = {
    var f = 1
    var i = 2
    while (i<=n) {
        f *= i
        i += 1
    }
    f
}
```

Si noti come questa formulazione è meno compatta di quelle funzionali viste in precedenza.

8.3.2 for

Il costrutto `for` di Scala è più articolato di quello che si ha in C, Python, o Java. Consente di usare lo stesso schema del for comprehension per eseguire istruzioni invece di costruire collezioni.

Esempio.

```
for {
   i<-1 to 3                 // primo generatore
   val test = isOdd(i)       // dichiarazione variabile
   if (test)                 // filtro
   j<-1 to 2                 // secondo generatore
   if (i>j)                  // filtro
} println(i+"/"+j            // stampa 3/1 3/2
```

Bibliografia

[ScalaRef] The Scala Language Specification
http://www.scala-lang.org/docu/files/ScalaReference.pdf (verificato 3 ottobre 2015)

[ScalaOperators] TutorialsPoint, Scala Operators
http://www.tutorialspoint.com/scala/scala_operators.htm (verificato 3 ottobre 2015)

[ClasseList] Documentazione classe `List`
http://www.scala-lang.org/api/2.11.4/index.html#scala.collection.immutable.List
(verificato 16 ottobre 2015)

[MetodiVsFunzioni] Discussione della differenza tra metodi e funzioni in Scala su StackOverflow
http://stackoverflow.com/questions/2529184/difference-between-method-and-function-in-scala/2530007#2530007
(verificato 16 ottobre 2015)