

---

# 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, 971, 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'	λμδ	_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* *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

RESTO      OR      AND  
↑      ↑      ↑

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_+
% / == ^ instanceof 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<sup>1</sup>.

#### Esempio 1.

VAL COND = TRUE

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

CONDIZIONE  
↓  
VERO  
↓  
FALSO  
↓

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

COND ? 20 : 30



### 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.

*tipi misti*

```
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'arietà 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.

*• VAL  
• TUPLE*

#### 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
```

*Handwritten notes:* "tipo" with an arrow pointing to `T1`, "RETURN" with an arrow pointing to the equals sign, and "parametro io input" with an arrow pointing to `p1`.

dove `identificatore` è il nome del metodo, `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**.

*Handwritten note:* ~~x=5~~ NO

### 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 analogo 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 }
```

Questa possibilità, sebbene supportata, è tuttavia ad oggi deprecata.

**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.

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

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 funzione termini. 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 linguaggio funzionale 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
```



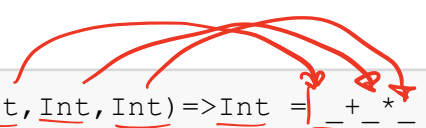
**Forma abbreviata.** Esiste una forma abbreviata molto compatta per una funzione anonima, che consiste in una espressione in cui appaiono degli underscore "\_" che fungono da parametri anonimi, nell'ordine in cui appaiono:

#### 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<sup>2</sup>.

#### 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)
```

<sup>2</sup> 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.** Consideriamo 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

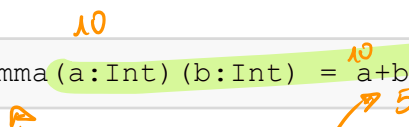
Una funzione richiede che tutti i parametri siano specificati. Alle volte può essere utile specificarli solo in parte, ottenendo una **funzione parzialmente applicata**. L'idea è che una funzione con `n` parametri può essere sempre riscritta come una funzione che prende i primi `k` parametri e restituisce 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:

(1)

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

La funzione `somma` può essere chiamata specificando tutti i parametri:

(1)

```
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 metodi 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 facendolo 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.