Introduction to Functional Programming with Scala

Angelo Gargantini

INFO 3A AA 2016/17
credits: Pramode C.E
https://class.coursera.org/progfun-00

16 dicembre 2016

Workshop Plan

Here is what we will do:

- Learn a bit of functional programming in Scala
- Learn some important concepts like (NOT ALL): closures, higher order functions, purity, lazy vs strict evaluation, currying, tail calls/TCO, immutability, persistent data structures, type inference etc!

Workshop material (slide/code samples) sul sito.

Function Definition

```
def add(a:Int, b:Int):Int = a + b

val m:Int = add(1, 2)

println(m)
```

Note the use of the type declaration "Int". Scala is a "statically typed" language. We define "add" to be a function which accepts two parameters of type Int and returns a value of type Int. Similarly, "m" is defined as a variable of type Int.

Function Definition

```
def fun(a: Int):Int = {
a + 1
a - 2
a * 3
}

val p:Int = fun(10)
println(p)
```

Note!

There is no explicit "return" statement! The value of the last expression in the body is automatically returned.

Type Inference

```
def add(a:Int, b:Int) = a + b

val m = add(1, 2)

println(m)
```

We have NOT specified the return type of the function or the type of the variable "m". Scala "infers" that!

Type Inference

```
def add(a, b) = a + b

val m = add(1, 2)

println(m)
```

This does not work! Scala does NOT infer type of function parameters, unlike languages like Haskell/ML. Scala is said to do local, "flow-based" type inference while Haskell/ML do Hindley-Milner type inference

References

1. http://en.wikipedia.org/wiki/Hindley-Milner
2. http://www.scala-lang.org/node/4654

Expression Oriented Programming

```
val i = 3
val p = if (i > 0) -1 else -2
val q = if (true) "hello" else "world"

println(p)
println(q)
```

Unlike languages like C/Java, almost everything in Scala is an "expression", ie, something which returns a value! Rather than programming with "statements", we program with "expressions"

Expression Oriented Programming

```
def errorMsg(errorCode: Int) = errorCode match {
case 1 => "File not found"
case 2 => "Permission denied"
case 3 => "Invalid operation"
}

println(errorMsg(2))
```

Case automatically "returns" the value of the expression corresponding to the matching pattern.

Evaluation

Applications of parametrized functions are evaluated in a similar way as operators. Expressions are evaluated before passing their value to functions (values are passed to functions)

```
def square(x:Double) = x * x

square(2)

square(2+2)

square(square(2))
```

Evaluation of function application

Given a function application $f(e_1, \ldots, e_n)$

1. Evaluate all function arguments $(e_1, \ldots, e_n)$ from left to right. Let $v_1, \ldots, v_n$ the corresponding values.
2. Replace the function application by the function's right hand side (function body), and, at the same time
3. Replace (substitute) the formal paramters of the function by the actual arguments $v_1, \ldots, v_n$

```
def sumOfSquare(x:Double, y: Double) = square(x) +
    square(y)
```

```
sumOfSquare(3,2+2)
sumOfSquare(3,4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
....
```

Evaluation of function application

This scheme of expression evaluation is called the *substitution model*
The basic idea is to reduce an expression to a value
It can proved that this model can represent any algorithm (except side effect).

Termination

Does every expression reduce to a value (in a finite number of steps)? NO

Evaluation of function application

This scheme of expression evaluation is called the *substitution model*
The basic idea is to reduce an expression to a value
It can proved that this model can represent any algorithm (except side effect).

Termination

Does every expression reduce to a value (in a finite number of steps)? NO

```
def loop: Int = loop

loop
```

Alternative evaluation

The interpreter reduces function arguments to values before rewriting the function application.
One could alternatively apply the function to unreduced arguments.
For instance:

```
sumOfSquare(3,2+2)
square(3) + square(2+2)
3*3 + (2+2) * (2+2)
....
```

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3)

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps
test(3+4,8)

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps
test(3+4,8) CBV faster

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps
test(3+4,8) CBV faster
test(7,2*4)

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps
test(3+4,8) CBV faster
test(7,2*4) CBN faster

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps
test(3+4,8) CBV faster
test(7,2*4) CBN faster
test(3+4,2*4)

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

```
def test(x:Int, y:Int) = x * x
```

which call is fastest (fewest num. operations)?

test(2,3) same number of steps
test(3+4,8) CBV faster
test(7,2*4) CBN faster
test(3+4,2*4) same number of steps

Call by name functions

Scala normally uses call-by-value
But if the type of a function parameter with $=>$ it uses call-by-name
Example

```
def constOne(x:Int, y: => Int) = 1
```

Using pen and paper, trace the evaluation of the following function calls for the function constOne:
constOne(1+2,loop)
constOne(loop,1+2)

Recursion

- Recursion means a function can call itself repeatedly.
- Recursion plays a big role in pure functional programming and Scala supports recursion functions very well.

Recursion

- Recursion means a function can call itself repeatedly.
- Recursion plays a big role in pure functional programming and Scala supports recursion functions very well.

Consider the Eulid's algorithm

```
def gcd(a: Int, b: Int) = if (b == 0) a else gcd(b,a%b)
```

evaluate gcd(14,21) ...
Consider the factorial algorithm

```
def factorial(n: Int) = if (n == 0) 1 else
    n*factorial(n-1)
```

evaluate factorial(4) ...

Recursion

```
// sum n + (n-1) + (n-2) + ... + 0
def sum(n: Int): Int =
if (n == 0) 0 else n + sum(n - 1)

val m = sum(10)
println(m)
```

Try calling the function "sum" with a large number (say 10000) as parameter! You get a stack overflow!

Tail Calls and TCO

If a function calls itself as its last action is called tail recursion.
The function's stack frame can be reused (Tail Call Optimization).
Rewrite the function as tail recursion.

Tail Calls and TCO

If a function calls itself as its last action is called tail recursion.
The function's stack frame can be reused (Tail Call Optimization).
Rewrite the function as tail recursion.

```
def factorial(n: Int): Int = {
def loop(acc: Int, n: Int)=
if ( n == 0) acc
else loop(n*acc, n-1)
loop(0,n)
}
```

Tail Calls and TCO

```
def sum(n: Int, acc: Int):Int =
if(n == 0) acc else sum(n - 1, acc + n)

val r = sum(10000, 0)

println(r)
```

This is a "tail-recursive" version of the previous function - the Scala compiler converts the tail call to a loop, thereby avoiding stack overflow!

Proviamo con la segnatura semplificata e un loop interno.

Tail Calls and TCO

```
(sum 4)
(4 + sum 3)
(4 + (3 + sum 2))
(4 + (3 + (2 + sum 1)))
(4 + (3 + (2 + (1 + sum 0))))
(4 + (3 + (2 + (1 + 0))))
(4 + (3 + (2 + 0)))
(4 + (3 + 2))
(4 + 5)
(9)
--------------------------------
(sum 4 0)
(sum 3 4)
(sum 2 7)
(sum 1 8)
(sum 0 9)
(9)
--------------------------------
```

Higher-Order Functions

- Functional languages treat functions as *first-class values*
- This means that, like any other value, a function can be passed as a parameter and returned as a result
- Functions that take values and variable are called *first order functions*
- Functions that take other functions as parameters or return functions are called *higher order functions*

Summation once again!

take the sum of the integers from a and b:

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

Summation once again!

take the sum of the integers from a and b:

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

If you want to sum the squares or cubes from a and b:

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

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

Summation

Exercise

Define the sum of factorial from a and b

Summation

Exercise

Define the sum of factorial from a and b

Idea

Define a sum generic with the respect to the operation applied to each number?

```
def operation(x:Int) = ...

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

Higher order functions

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

"sum" is now a "higher order" function! It's first parameter is a function which maps an Int to an Int

The type "Int" represents a simple Integer value. The type `Int => Int` represents a function which accepts an Int and returns an Int.

Higher order functions

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

"sum" is now a "higher order" function! It's first parameter is a function which maps an Int to an Int

The type "Int" represents a simple Integer value. The type `Int => Int` represents a function which accepts an Int and returns an Int.

```
def identity(x: Int) = x
def sqr(x: Int) = x * x
def cube(x: Int) = x * x * x
def fact(x:Int) = ...
println(sum(identity, 1, 10))
println(sum(sqr, 1, 10))
println(sum(cube, 1, 10))
```

Anonymous functions

Passing functions as parameters leads to the creation of many functions. Sometime is tedious. It can be avoided.
Like:

```
def name = "Angelo"; println(name)
```

can be written as

```
println("Angelo")
```

we want to define functions without an explicit name:
anonymous functions

Anonymous functions

Can be written as:
(*Parameters*) => *Body*

We can create "anonymous" functions on-the-fly! `x => x*x` is a function which takes an "x" and returns x*x

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

The parameter type can be omitted if the compiler can infer it:

```
x => x *x
```

```
println(sum(x=>x, 1, 10))
println(sum(x=>x*x, 1, 10))
println(sum(x=>x*x*x, 1, 10))
```

Higher order functions and recursive calls

Rewrite sum with the tail recursion?

Higher order functions and recursive calls

Rewrite sum with the tail recursion?

```
def sum(f: Int => Int, a: Int, b: Int): Int = {
def loop(a: Int, acc: Int): Int = {
if (a > b) acc
else loop(a + 1, acc + f(a))
}
loop(a, 0)
}
```

Currying

Here is the definition from Wikipedia:

> In mathematics and computer science, currying is the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument. It was originated by Moses Schonfinkel and later re-discovered by Haskell Curry.

Let's try to do this in Scala!

Currying - returning functions

```
def sumInts(a:Int, b: Int) = sum(x=>x, 1, 10)
def sumCubes(a:Int, b: Int) = sum(x=>x*x, 1, 10)
def sumFactorial(a:Int, b: Int) = sum(fact, 1, 10)
```

a and b are passed to sum unchanged. Can we rewrite sum and avoid the use of a and b?

Currying - returning functions

```
def sumInts(a:Int, b: Int) = sum(x=>x, 1, 10)
def sumCubes(a:Int, b: Int) = sum(x=>x*x, 1, 10)
def sumFactorial(a:Int, b: Int) = sum(fact, 1, 10)
```

a and b are passed to sum unchanged. Can we rewrite sum and avoid the use of a and b?

sum can *return* a function that takes two Ints and return an Int

```
def sum(f: Int => Int): (Int,Int) => Int = { ...}
```

sum takes a function `f` and return a function `(Int,Int) => Int`

Currying - returning functions

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

Currying - stepwise application

The basic sum functions can be defined without parameters:

```
def sumInts = sum(x=>x)
def sumCubes = sum(x=>x*x)
def sumFactorial = sum(fact)
```

sumInts(3,4) ...
or we could write

```
sum(x=>x)(3,4)
```

Multiple parameter list

```
def sum(f: Int => Int, a: Int, b: Int): Int =
```

Can be rewritten as:

```
def sum(f: Int => Int) : (Int, Int) => Int =
```

or equivalently, by using multiple parameter lists:

```
def sum(f: Int => Int)(a: Int, b: Int): Int =
```

the advantage wrt the first is that se can pass only one argument like sum(cube) the advantage wrt the second is that we can use a and b directly in the body

Currying - two argument functions

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

def addB(x: Int):Int=>Int =
y => x + y

val a = addA(10, 20)

val b = addB(10)(20)

println(a)
println(b)
```

Currying - three argument functions

```
def addA(x: Int, y: Int, z: Int) = x + y + z

def addB(x: Int): Int => (Int => Int) =
y => (z => x + y + z)

val a = addA(1, 2, 3)

val b = addB(1)(2)(3)

println(a)
println(b)
```

It is now easy to see how the idea can be generalized to N argument functions!

Exercise

1. write a function that calculates the product of the values for the points on a gievn interval
2. write a function that calculates the product of the values of a function f for the points on a given interval
3. write the factorial in terms of product
4. can we write a more general function which generalizes both sum and product

Exercise

1. write a function that calculates the product of the values for the points on a gievn interval
2. write a function that calculates the product of the values of a function f for the points on a given interval
3. write the factorial in terms of product
4. can we write a more general function which generalizes both sum and product

Immutability

Why? Immutable objects are automatically thread-safe (you don't have to worry about object being changed by another thread)
Compiler can reason better about immutable values -¿ optimization
Steve Jenson from Twitter: "Start with immutability, then use mutability where you find appropriate."

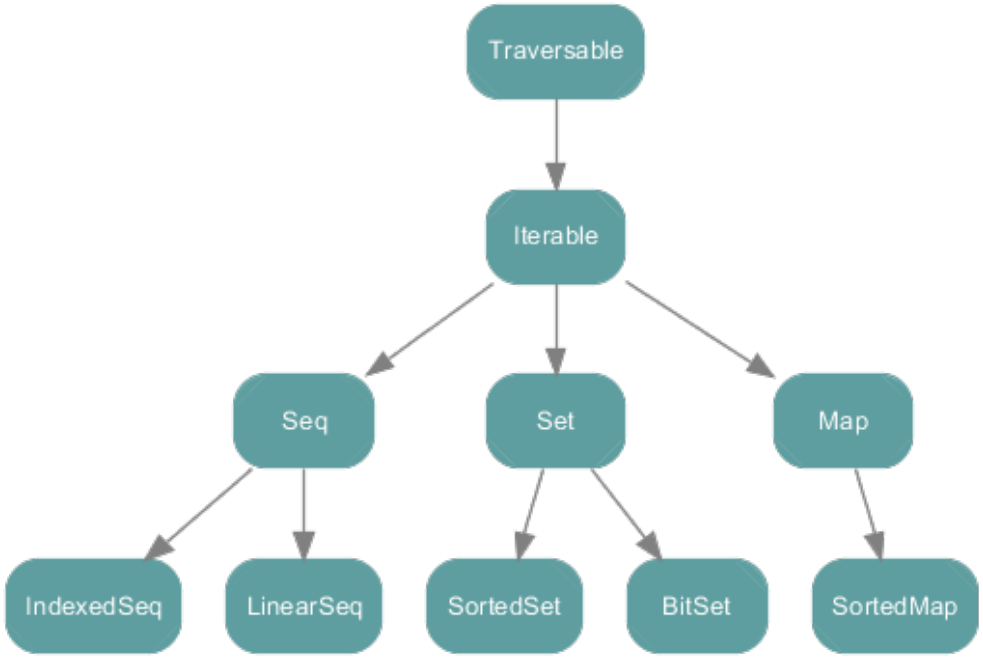
Collezioni

Mutable and Immutable Collections

- Scala collections systematically distinguish between mutable and immutable collections.
- A mutable collection can be updated or extended in place.
- Immutable collections, by contrast, never change.
- You have still operations that simulate additions, removals, or updates, but those operations will in each case return a new collection and leave the old collection unchanged.

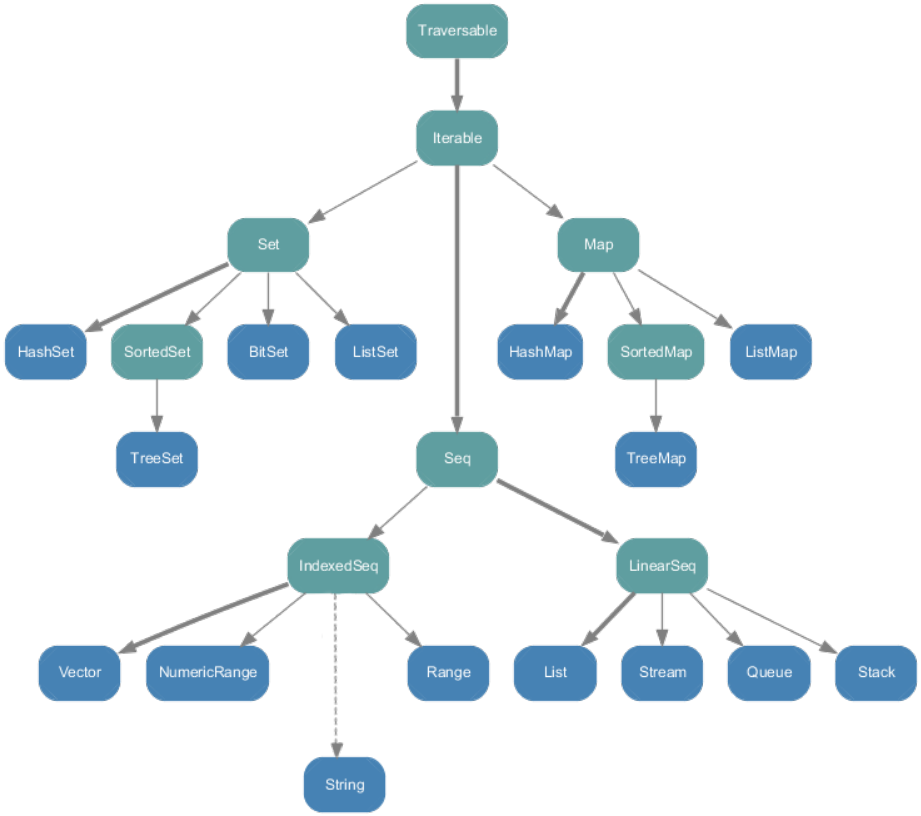
Alto livello

collections in package scala.collection



Immutable

Queste sono le implementazioni immutabili



List

Le Lists sono immutabili (non può essere cambiato il contenuto)
List[String] contains Strings

```
val lst = List("b", "c", "d")
lst.head                    // "b"
lst.tail                    // List("c", "d")
val lst2 = "a" :: lst  // cons operator
```

Nil = synonym for empty list

```
val l = 1 :: 2 :: 3 :: Nil
```

List concatenation:

```
val l2 = List(1, 2, 3) ::: List(4, 5)
```

Foreach

Posso usare foreach per applicare una funzione a tutti gli elementi di una lista:

```
val list3 = List("mff", "cuni", "cz")
```

Following 3 calls are equivalent

```
list.foreach((s : String) => println(s))
list.foreach(s => println(s))
list.foreach(println)
```

For comprehensions

```
for (s <- list) println(s)
for (s <- list if s.length() == 4) println(s)
```

Maps and Sets

Ci sono anche mappe ed insiemi

Maps

Esempio di uso di mappe

```
import scala.collection._

val cache = new mutable.HashMap[String,String];
cache += "foo" -> "bar";

val c = cache("foo");
```

The rest of Map and Set interface looks as you would expect

Mutable List: ListBuffer

ListBuffer[T] is a mutable List Like Java's ArrayList¡T¿

```
import scala.collection.mutable._

val list = new ListBuffer[String]
list += "Vicky"
list += "Christina"

val str = list(0)
```

scala.Seq

- scala.Seq is the supertype that defines methods like: filter, fold, map, reduce, take, contains, ...
- List, Array, Maps... descend from Seq

From Java to Scala

```
Iterator                <=> java.util.Iterator
Iterator                <=> java.util.Enumeration
Iterable                <=> java.lang.Iterable
Iterable                <=> java.util.Collection
mutable.Buffer          <=> java.util.List
mutable.Set             <=> java.util.Set
mutable.Map             <=> java.util.Map
mutable.ConcurrentMap   <=> java.util.concurrent.ConcurrentM
```

Iterate – foreach function

Every collection in Scala's library defines (or inherits) a foreach method

```
val names = List("Daniel", "Chris", "Joseph")
names.foreach { name =>
  println(name)
}}
```

foreach is a "higher-order" method, due to the fact that it accepts a parameter which is itself another method

```
name => println(name)
names.foreach(println)
```

Foreach - istruzione

There are times that we just want to use a syntax which is similar to the for-loops available in other languages.

```
val nums = List(1, 2, 3, 4, 5)
var sum = 0
for (n <- nums) { sum += n }
```

Oppure se volessi usare una funzione + variabile locale:

```
var ss = 0;
def sinc(x:Int) = { ss += x }
nums.foreach(sinc)
```

Folding

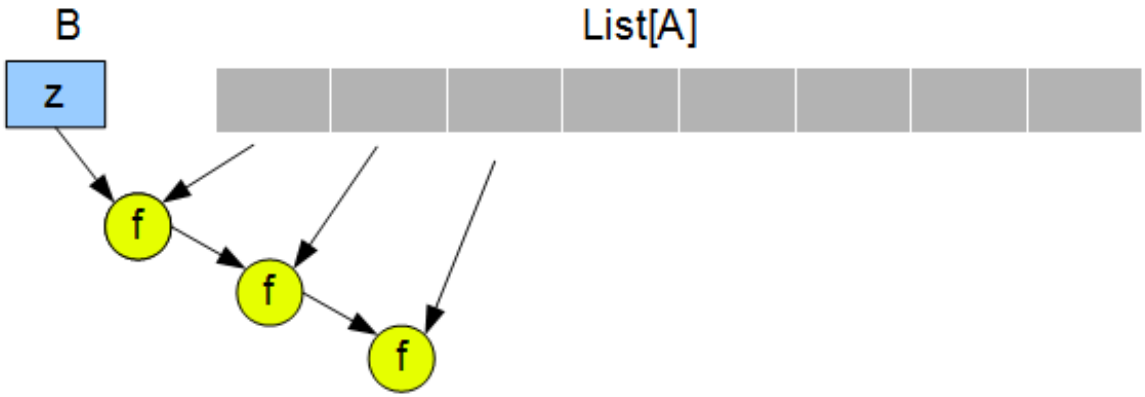
Looping is nice, but sometimes there are situations where it is necessary to somehow combine or examine every element in a collection, producing a single value as a result. Data una List[A]:

```
def foldLeft[B](z: B)(f: (B, A) => B): B
```

The foldLeft function goes through the whole List[A], from head to tail, and passes each value to f. For the first list item, that first parameter, z, is used as the first parameter to f. For the second list item, the result of the first call to f is used as the B type parameter.

foldLeft

```
def foldLeft[B](z: B)(f: (B, A) => B): B
```



Dove:

1. z è l'elemento iniziale di tipo B
2. f è l'operazione che dato un B e un A alla volta, costruisce mano a mano il dato riassuntivo di tipo B

Folding Esempi

1. Somma di tutti i numeri in nums

```
def myf(x: Int, y: Int) = x+y
val sum = nums.foldLeft(0)(myf)
```

2. oppure con funzione anonima

```
val sum = nums.foldLeft(0)((total, n) => total + n)
```

Reduce

Fold has a closely related operation in Scala called “reduce” which can be extremely helpful in merging the elements of a sequence where leading or trailing values might be a problem. Consider the ever-popular example of transforming a list of String(s) into a single, comma-delimited value:

```
var nn = List("a","b", "c")// voglio stampare "a,b,c"
println(nn.foldLeft("")((x, y) => x + "," + y))
```

Stampa però: ,a,b,c

Reduce

Solution: use a reduce, rather than a fold. Reduce distinguishes itself from fold in that it does not require an initial value to "prime the sequence". Rather, it starts with the very first element in the sequence and moves on to the end.

```
def reduceLeft(f: (A, A) => A): A
```

Esempi:

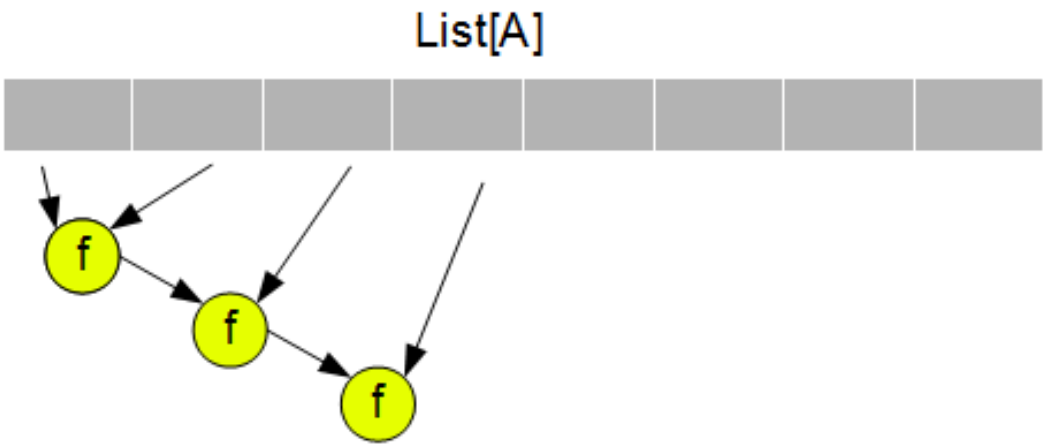
```
println(nn.reduceLeft((x, y) => x + "," + y))
--> a,b,c
```

Altro esempio: calcolo del max in lista

```
l.reduceLeft((x, y) => if (x>y) x else y )
```

Reduce

```
def reduceLeft(f: (A, A) => A): A
```



Esempi di Folder/Reduce

Fai la somma/prodotto dei numeri in una lista Restituisci la stringa piu' lunga Trova la dimensione della stringa piu' lunga ....

Filter/map

fold can be an extremely useful tool for applying a computation to each element in a collection and arriving at a single result if we want to apply a method to every element in a collection in-place (as it were), creating a new collection of the same type with the modified elements? Esempi, data una lista, costruire la lista dei doppi

```
var ll = List(3, 4, 5)
// lista dei doppi
def doppio(x: Int) = 2 * x
println(ll.map(doppio))
```

La lista delle lunghezze di una stringa

```
nomi.map( x => x.length())
```

Filter

Alcune volte voglio estrarre delle liste filtrando il contenuto Ad esempio: data una lista estrarre la lista pari

```
def pari = (x: Int) => (x % 2 == 0)
println(ll.filter(pari))
```

Using Map+Reduce

Spesso si usa map insieme a reduce:

- Con map trasformo i dati per renderli più trattabili
- Con reduce ottengo un dato sintetico
- Sono algoritmi che si possono parallelizzare
- Vedi google framework mapreduce http://it.wikipedia.org/wiki/MapReduce
- oppure spark http://spark.apache.org/