Scala tutorial pratico

Parte 3

Introduzione.

Ok, sono alla frutta. Ho provato a cavar fuori un terzo programmino ma non m'è venuto in mente un bell'accidenti di nulla – che fosse “ino”. Quindi uno fa i bagagli? Manco per idea. Sotto con “i casi”, ovvero quali caratteristiche di Scala mi si sono dimostrate comode.

Funzioni parzialmente applicate

Ipotizziamo di avere un dato aggregato immesso dall'utente in una gui (proprio un'ipotesi buttata lì). Prima di salvarlo dobbiamo verificare che i suoi dati siano validi. Non che ci siamo molto da arzigogolare sul tema: si piglia il dato e per ogni campo si controlla che vada bene. L'aspetto divertente nel contesto di scala è dato dal fatto che i metodi “sono funzioni” (in verità lo diventano solo quando siano usati come funzioni, altrimenti sono metodi tout-court).

Abbiamo il nostro dato:

```
class Dato(val nome: String, val cognome: String, val codiceFiscale: String)
```

Questa è una classe che dichiara tre campi pubblici di tipo String che devono necessariamente ricevere un valore all'atto dell'inizializzazione (cioè ha un solo costruttore che accetta tre argomenti e li associa ad altrettanti campi pubblici “final”, in javese).

Se immaginiamo che il controllo sia:

per ogni campo, controlla il suo valore e se è sbagliato restituisci un messaggio di errore

allora, considerato che l'assenza di messaggi d'errore corrisponde alla correttezza del dato, potremmo tradurre l'operazione con la creazione di N metodi di controllo, ciascuno dei quali opera su un valore di tipo Dato e verifica uno specifico campo di quel dato, e l'esecuzione ciclica di quei controlli. Insomma, una cosa di questo genere:

```
package test

import java.util.Scanner

class Dato(val nome: String, val cognome: String, val codiceFiscale: String)

object Convalida {

	def apply(dato: Dato): List[String] = {
		List(checkNome _, checkCognome _, checkCodiceFiscale _).flatMap(f => f(dato))
	}

	def checkNome(dato: Dato): Option[String] = {
		if(dato.nome == null || dato.nome.isEmpty) Some("Nome errato") else None
	}
```

```
	def checkCognome(dato: Dato): Option[String] = {
		if(dato.cognome == null || dato.cognome.isEmpty) Some("Cognome errato") else None
	}

	def checkCodiceFiscale(dato: Dato): Option[String] = {
		if(dato.codiceFiscale == null || dato.codiceFiscale.isEmpty) Some("Codice fiscale errato") else
			None
	}
}

object Main {

	def main(args: Array[String]) {
		val scanner = new Scanner(System.in)
		System.out.println("Inserire il nome e premere invio:")
		val nome = scanner.nextLine
		System.out.println("Inserire il cognome e premere invio:")
		val cognome = scanner.nextLine
		System.out.println("Inserire il codice fiscale e premere invio:")
		val codiceFiscale = scanner.nextLine

		val dato = new Dato(nome, cognome, codiceFiscale)
		val check = Convalida(dato)
		if(check.isEmpty) {
			System.out.println("Dati confermati")
		} else check.foreach(s => System.out.println("Errore: " + s))
	}
}
```

Il codice è completo ma la parte che ci interessa è solo Convalida:

```
object Convalida {

	def apply(dato: Dato): List[String] = {
		List(checkNome _, checkCognome _, checkCodiceFiscale _).flatMap(f => f(dato))
	}

	def checkNome(dato: Dato): Option[String] = {
		if(dato.nome == null || dato.nome.isEmpty) Some("Nome errato") else None
	}

	def checkCognome(dato: Dato): Option[String] = {
		if(dato.cognome == null || dato.cognome.isEmpty) Some("Cognome errato") else None
	}

	def checkCodiceFiscale(dato: Dato): Option[String] = {
		if(dato.codiceFiscale == null || dato.codiceFiscale.isEmpty) Some("Codice fiscale errato")
		else None
	}
}
```

Abbiamo tre metodi, checkNome, checkCognome, checkCodiceFiscale. Tutti e tre accettano come argomento un Dato e verificano qualcosa, restituendo un Option[String] che può essere definito, caso in cui si suppone che contenga la descrizione dell'errore riscontrato durante la convalida, o indefinito, caso in cui tutto sia andato per il meglio. Nulla da dire di più, salvo forse la forma if-else. In scala if-else restituisce un valore: il valore dell'espressione che segue l'if(condizione) se la condizione è vera, il valore dell'espressione che segue l'else se la condizione è falsa. Ergo si può dire:

```
var x = if(qualcosa) 10 else 20
```

O anche

```
var x = if(qualcosa) 10 else if(quacos'altro) 20 else 30
```

La parte più stramba è tuttavia il metodo apply:

```
    def apply(dato: Dato): List[String] = {
        List(checkNome _, checkCognome _, checkCodiceFiscale _).flatMap(f => f(dato))
    }
```

Il metodo restituisca una lista di stringhe. Dal punto di vista dell'uso, Convalida funzionerebbe quindi così:

```
val errori = Convalida(dato)
if(errori.isEmpty) tutto ok
else … qualcosa non va, errori ci dice cosa
```

Nel corpo del metodo apply creiamo una lista:

```
List(elementi)
```

Non c'è il new perchè esiste un object di nome List che possiede un metodo apply, quindi quel

```
List(elementi)
```

equivale a dire

```
List.apply(element)
```

con List nome del singleton-object. Gli elementi che usiamo per la lista sono indicati da un nome di metodo seguito dal carattere di sottolineatura:

```
List(checkNome _, checkCognome _, checkCodiceFiscale _)
```

Che significa quel "_"? Quando segue il nome di un metodo o di un valore di tipo funzione il simbolo _ denota l'applicazione parziale di quella funzione. La funzione è parzialmente applicata perché rispetto all'insieme di **liste di** parametri che la funzione di norma accetterebbe noi ne passiamo solo alcuni (in questo caso nessuno) lasciando il resto "in bianco". Il risultato dell'applicazione parziale di una funzione è una nuova funzione che accetta come argomenti un numero di parametri pari a quelli che noi abbiamo deciso di lasciare "in bianco".

Prendiamo ad esempio il metodo checkNome. Il metodo checkNome accetta una lista di argomenti, composta di un valore di tipo dato Se io uso il suo nome facendolo seguire da un _

```
val qualcosa = checkNome _
```

qualcosa è una funzione che accetta un parametro di tipo dato. Se invece uso checkNome specificando l'argomento:

```
val qualcosa = checkNome(un dato)
```

allora qualcosa vale quando l'applicazione di checkNome al dato che gli rifilo (cioè sarebbe un Option[String]).

Se una funzione o un metodo hanno più liste di argomenti:

```
def metodo(x: Int)(s: String) = System.out.println(x + ":" + s)
```

e io "applico parzialmente" la funzione, col _:

```
val qualcosa = metodo(10) _
```

qualcosa è... una funzione che accetta come argomento una stringa (perché io ho riempito la prima lista ma non la seconda e ho usato il _)

Dunque il mio:

```
List(checkNome _, checkCognome _, checkCodiceFiscale _)
```

E' una lista di funzioni dove ognuna di queste funzioni è creata "applicando parzialmente" un'altra funzione (cioè riempiendo solo parte delle liste di argomenti accettati da un'altra funzione), posto che si l'omonimo metodo.

Ri-ripeto. Che fa l'applicazione parziale di una funzione?

1: piglia un metodo (def) o una funzione e restituisce un'altra funzione.

2: la funzione restituita fa le stesse cose della funzione originale con alcuni argomenti prefissati

Se ho:

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

Ho una funzione che fa la somma di due argomenti X e Y. Se dico:

```
def funzione2 = funzione(1) _
```

Ho una seconda funzione che fa la somma di 1 e un altro argomento. Se ho un metodo e voglio trattarlo come una funzione, ad esempio per assegnarlo ad una variabile, posso farlo tranquillamente usando l'applicazione parziale:

```
def metodo(x: Int): Unit = qualcosa

var y = metodo _
```

y è un (Int) => Unit, una funzione che piglia un Int e restituisce stica... niente.

L'altra stranezza è flatMap o, meglio, quel che salta fuori.

Abbiamo una lista di funzioni che prendono un Dato e restituiscono tutte quante un Option[String]. A quella lista diciamo “flatMap” e, magia, salta fuori una lista di stringhe, perchè noi diciamo:

```
def apply(dato: Dato): List[String] = {
    List(checkNome _, checkCognome _, checkCodiceFiscale _).flatMap(f => f(dato))
}
```

C'è il trucco. Il metodo flatMap restituisce una successione di elementi. La successione restituita è creata applicando a ciascun elemento una funzione che accetta come argomento un valore della lista e restituisce una sequenza di elementi dello stesso tipo. Gli elementi dell'iteratore restituito dalla funzione applicata sono accumulati, Alla fine del flatMap, cioè quando sono stati “flatMappati” tutti gli elementi della lista, è restituita la sequenza formata dall'accumulo dei valori contenuti negli iteratori prodotti dall'applicazione della funzione ad ogni elemento della lista. La funzione che noi applichiamo a ciascun elemento della lista è la stessa funzione che si trova nella lista. Per ogni elemento-funzione X della lista → flatMap → applica quella funzione X al dato. Le funzioni nella lista sono tutte dei (Dato) => Option[String], quindi per ogni funzione applicata nel flatMap otteniamo un Option[String]. Guarda caso, Option[String] è un iteratore, che può avere zero o un elemento. L'elemento dell'iteratore-option (cioè una stringa se c'è) viene accumulato e, al termine del flat map, restituito come elemento della nuova lista.

Ricapitolando, la funzione che uso come argomento del metodo flatMap crea un iteratore per ogni elemento della lista. Il contenuto di quegli iteratori viene a sua volta “srotolato” per formare la sequenza di valori restituita da flatMap. Option[T] è un iteratore che ha zero elementi se è indefinito (cioè se è None o Some(null)) o un elemento se è definito (cioè se è Some(qualcosa != null))

Il metodo flatMap è ad elevato rischio divertimento perché consente di trasformare sequenze di valori in sequenze di valori di tipo diverso o dello stesso tipo ma in numero diverso o di tipo e quantità diverse.

Nel nostro caso, lo usiamo per trasformare una sequenza di funzioni in una sequenza di messaggi di errore eventuali, ottenendo questi ultimi dall'applicazione delle funzioni elemento di quella

sequenza ad un valore dato.

Occhio che non siamo obbligati a usare questa forma per gestire la restituzione degli errori nel nostro oggetto Convalida. Se uno lo preferisce, si può scrivere:

```
def apply(dato: Dato): List[String] = {
    var risultati = List[String]()
    val controlli = List(checkNome _, checkCognome _, checkCodiceFiscale _)
    controlli.foreach { controllo =>
        val errore = controllo.apply(dato)
        if(errore.isDefined) risultati = risultati :+ errore.get
    }
    risultati
}
```

Si può anche non usare il foreach:

```
def apply(dato: Dato): List[String] = {
    var risultati = List[String]()
    val controlli = Array(checkNome _, checkCognome _, checkCodiceFiscale _)
    for(i <- 0 until controlli.length) {
        val controllo = controlli(i)
        val errore = controllo.apply(dato)
        if(errore.isDefined) risultati = risultati :+ errore.get
    }
    risultati
}
```

Qui l'unica cosa bizzarra è il for. Scala non ha il classico for di C:

```
for(inizializzazione; controllo; incremento) { … }
```

A rigore bisogna simularlo con il while:

```
inizializzazione
while(controllo) {
    …
    incremento
}
```

Il for di scala è un generatore di sequenze, nel senso che dire:

```
for(i <-0 until 10) … fai qualcosa
```

fa qualcosa sui valori di un'ipotetica lista di interi che va da 0 a 9 (until esclude l'estremo superiore). La differenza concreta è che non abbiamo il controllo sulla condizione e sull'incremento.

Chiaramente si può anche fare a meno della lista di funzioni ricavate dall'applicazione parziale dei

metodi. Qui tuttavia c'è una obiettiva comodità: per aggiungere un nuovo controllo basta aggiungere il nome del metodo col trattino nella lista di controlli.