Scala, un tutorial pratico

Parte 2.

Introduzione.

Il degno successore dell'agenda non può che essere il magazzino. Limitiamoci alle funzioni di base: carico, scarico, stock.

Strutture.

Idealmente abbiamo due definizioni da fornire, il magazzino e il prodotto, nel senso di qualcosa di immagazzinabile.

```
package magazzino

trait Prodotto {
}
```

```
package magazzino

trait Magazzino {
}
```

Un prodotto ha un codice di identificazione univoco:

```
package magazzino

trait Prodotto {

    val codice: String
}
```

Dotiamo il magazzino di tre operazioni di base:

```
package magazzino

trait Magazzino {

    def quantità(p: Prodotto): Option[BigDecimal]

    def aggiungi(p: Prodotto, q: BigDecimal)
```

```
    def rimuovi(p: Prodotto, q: BigDecimal)
}
```

E gli infiliamo anche la capacità di eseguire un'operazione per ogni prodotto contenuto:

```
package magazzino

trait Magazzino {

    def quantità(p: Prodotto): Option[BigDecimal]

    def aggiungi(p: Prodotto, q: BigDecimal)

    def rimuovi(p: Prodotto, q: BigDecimal)

    def foreach(op: Prodotto => Unit)
}
```

La notazione Prodotto => Unit corrisponde a Function1[Prodotto, Unit] ed indica una funzione che accetta un argomento di tipo Prodotto e restituisce qualcosa che comunque sarà scartato (Unit è un pelo più di niente). Infine, ci mettiamo un bel metodo per l'esecuzione di una funzione arbitratia che accetta come argomento un magazzino e restituisce un numero decimale opzionale.

```
package magazzino

trait Magazzino {

    def quantità(p: Prodotto): Option[BigDecimal]

    def aggiungi(p: Prodotto, q: BigDecimal)

    def rimuovi(p: Prodotto, q: BigDecimal)

    def foreach(op: Prodotto => Unit)

    def esegui[T](op: (Magazzino) => T) = op(this)
}
```

Il [T] è un generico: al posto di T l'invocante metterà quel che vuole. Di norma sarà il compilatore ad inferire il valore di T. Da notare che il metodo esegui, a differenza degli altri, ha una definizione concreta, non è astratto. Chi implementerà il trait Magazzino si troverà quindi con questa definizione di base già data. Volendo potrà sovrascriverla.

Morale della favola, il magazzino è una collezione mutabile di prodotti su cui è possibile eseguire un'operazione generica.

Diamo uno sguardo a queste operazioni, sempre dall'alto. Lo stock è “quanto prodotto x ho in

magazzino”:

```
package magazzino

trait Stock extends Function1[Magazzino, Option[BigDecimal]] {

	val prodotto: Prodotto

	def apply(m: Magazzino) = m.quantità(prodotto)
}
```

Anche qui abbiamo un qualcosa di parzialmente definito: l'operazione è definita, il prodotto no, sappiamo solo che c'è.

Il carico è:

```
package magazzino

trait Carico extends Function1[Magazzino, Unit] {

	val prodotto: Prodotto

	val quantità: BigDecimal

	def apply(m: Magazzino) = {
		m.aggiungi(prodotto, quantità)
	}
}
```

E lo scarico:

```
package magazzino

trait Scarico extends Function1[Magazzino, Unit] {

	val prodotto: Prodotto

	val quantità: BigDecimal

	def apply(m: Magazzino) = {
		m.rimuovi(prodotto, quantità)
	}
}
```

Un po' di concretezza.

Passiamo a definire un po' di quello che abbiamo solo dichiarato. Creiamo un package magazzino.impl, creiamo un sorgente Operazioni.scala e scriviamo:

```
package magazzino.impl

import magazzino.{Stock, Scarico, Carico, Prodotto}

class CaricoBase(val prodotto: Prodotto, val quantità: BigDecimal) extends Carico

class ScaricoBase(val prodotto: Prodotto, val quantità: BigDecimal) extends Scarico

class StockBase(val prodotto: Prodotto) extends Stock
```

Dal che si capiscono alcune cose di Scala: le unità di compilazione possono essere denominate a piacere e possono contenere quanti tipi, pubblici o no, desiderino.

Una console di comando.

Giusto per variare, stavolta usiamo un terminale di testo per parlare col programma.

```
package magazzino.impl

object Terminale {

	def main(args: Array[String]) {

	}
}
```

Abbiamo un'operazione per la stampa del menu radice, una per il carico, una per lo scarico, una per lo stock e una per l'uscita:

```
package magazzino.impl

import magazzino.Magazzino

trait Terminale {

	val magazzino: Magazzino
```

```
    def stock() {

    }

    def carico() {

    }

    def scarico() {

    }

    def esci() {

    }

    def mainMenu() {

    }
}
```

Il menu permette all'utente di scegliere una tra le altre operazioni:

```
    def mainMenu() {
        val operazioni = Array[() => Unit](stock, carico, scarico, esci)
    }
```

Qui operazioni è un array di funzioni e le funzioni sono i metodi stock, carico, scarico e esci. Il metodo prosegue con la stampa di un menu (usa System.console che in genere non funziona da IDE, bisogna testare il programma dalla linea di comando) e la richiesta di una scelta:

```
    def mainMenu() {
        val operazioni = Array[() => Unit](stock, carico, scarico, esci)
        val menu =
            "\nMenu Magazzino\n" +
            "1 Stock\n" +
            "2 Carico\n" +
            "3 Scarico\n" +
            "4 Esci\n" +
            "Digitare il numero di un'operazione (1, 2, 3, 4) e premere invio:"
        val console = System.console
        val choice = console.readLine(menu)
        val option = choice.toInt
```

```
        val operazione = operazioni(option - 1)
        operazione.apply()
    }
```

Ovviamente va tutto a schifio se l'utente inserisce un valore diverso da 1, 2, 3 o 4 ma la gestione dei limiti all'input è dettagliume di scarsissimo interesse. A rigore non serve scrivere operazione.apply, essendo una funzione basta dire operazione() (la regola dice che se una cosa si chiama apply allora l'apply si può omettere) ma senza il nome apply è un po' più criptico, quindi io lo metto. Naturalmente si può anche scrivere operazioni(option – 1).apply() o addirittura operazioni(option -1)() ma quest'ultima forma è rivoltante.

Passiamo al metodo stock. Chiediamo il codice del prodotto e vediamo cosa dice il magazzino:

```
    def stock() {
        val console = System.console
        val choice = console.readLine("\nInserire il codice del prodotto e premere invio:")
        val prodotto = new Prodotto { val codice = choice }
        val quantità = magazzino.esegui(new StockBase(prodotto))
        if(quantità.isDefined) {
            console.printf("Codice: %s%nQuantit: %d%n", prodotto.codice, quantità)
        } else {
            console.printf("Il codice %s non corrisponde ad alcun prodotto immagazzinato%n")
        }
        mainMenu()
    }
```

Direi nulla di trascendentale. La forma:

```
val prodotto = new Prodotto { val codice = choice }
```

è la contrazione di:

```
class ClasseSenzaNome extends Prodotto {
        val codice = choice
}
val prodotto = new ClasseSenzaNome
```

Javese classico. Più Scalico (?) sarebbe forse:

```
object prodotto extends Prodotto { val codice = choice }
```

Per quel che serve a noi sono uguali. Ora carichiamo:

```
def carico() {
    val console = System.console
    val cod = console.readLine("\nInserire il codice del prodotto da caricare e premere invio:")
    val quantità = console.readLine("\nInserire la quantità di prodotto da caricare e premere invio:")
    object prodotto extends Prodotto { val codice = cod }
    magazzino.esegui(new CaricoBase(prodotto, BigDecimal(quantità)))
    mainMenu()
}
```

Naturalmente sarebbe meglio evitare che l'utente non possa annullare l'operazione di carico in corso (qui una volta che entra in carico o carica o manda tutto a ramengo, non può ravvedersi) ma si può affinare in seguito.

Lo scarico è praticamente uguale al carico:

```
def scarico() {
    val console = System.console
    val cod = console.readLine("\nInserire il codice del prodotto da scaricare e premere invio:")
    val quantità = console.readLine("\nInserire la quantità di prodotto da scaricare e premere invio:")
    object prodotto extends Prodotto { val codice = cod }
    magazzino.esegui(new ScaricoBase(prodotto, BigDecimal(quantità)))
    mainMenu()
}
```

Non ci resta che creare un magazzino.

Magazzino.

Fracamente avrei voluto usare berkeley db ma non riesco a connettermi alla pagina di oracle per il dowload. Li possino. Quindi, sql e via. Si intuisce che parliamo di database. Esiste tra le api di Scala una libreria dedicata al dialogo con le basi dato (scala-dbc) ma richiede una folta pelliccia sullo stomaco per essere buttata giù, io mi astengo. Tanto più che quel che ci serve è veramente poco. Procediamo un passo alla volta. Iniziamo con un … trait.

```
package magazzino.impl

import java.util.logging.{Level, Logger}
import magazzino.{Prodotto, Magazzino}
import java.sql.{DriverManager}

trait MagazzinoH2 extends Magazzino {
}
```

Ci saranno tutti i metodi di Magazzino ma al momento ci serve una piccola digressione.

MagazzinoH2: CallAndClose

E' noto che le risorse potenzialmente limitate devano essere liberate subito dopo l'uso. Le API jdbc che useremo sono piene di queste risorse: dalla connessione alla base dati agli statement ai resultset. A noi serve una bella parola chiave che ci dica: inizializza una certa risorsa, fai quel che devi fare e

poi, cascasse il mondo, se l'inizializzazione c'è stata, libera quello che è stato inizializzato. C'è un costrutto, il try-finally, che fa quel che vogliamo ma riempie il codice di un rumore di fondo che è evitabile se lo “trucchiamo”. Ecco il truccone.

```
package magazzino.impl

object CallAndClose {

	def apply[T <: { def close(): Unit }, K](init: => T)(body: T => K) = {
		var statement: Option[T] = None
		try {
			statement = Some(init)
			body(statement.get)
		} finally statement.foreach(s => s.close)
	}
}
```

Praticamente giapponese. Vediamo cosa fa e poi come fa. Supponiamo di voler scrivere la parola ciao su un file:

```
val out = new PrintStream(il file)
out.print(“hello”)
out.flush()
out.close()
```

Giusto? No, perchè se tra il new e il close capita qualcosa di tragico, tipicamente un'eccezione, il close non è mai invocato, la risorsa rimane potenzialmente aperta e abbiamo un memory leak. Questo è quello che scriviamo:

```
var out: PrintStream = null
try {
		out = new PrintStream(il file)
		out.print(“hello”)
		out.flush()
} finally {
		if(out != null) out.close()
}
```

A prescindere da ciò che accade dopo il try, il finally sarà sempre invocato. Apriamo la risorsa solo dopo essere entrati nel try, quindi la chiudiamo di sicuro. A parte il print il resto è fuffa: serve ma è sempre uguale. CallAndClose si occupa della fuffa:

```
CallAndClose(new PrintStream(file)) { out =>
		out.print(“hello”)
		out.flush()
}
```

Perchè funziona.

In primis:

```
CallAndClose(new bla bla
```

significa:

```
CallAndClose.apply(new bla bla
```

L'apply si può omettere. Il metodo apply ha due liste di argomenti:

```
(init: => T)(body: T => K)
```

Il primo argomento della prima lista denota un'espressione il cui valore è T. Vediamo dopo cosa sia questo T. Il secondo argomento, nella seconda lista, è una funzione che accetta come argomento un T e restituisce un K. Il parametro init è quello che inizializza la risorsa. Il valore dell'espressione init è poi passato, nel corpo del metodo apply.

La lista multipla di argomenti permette di fare alcune cose divertenti. Quella che interessa a noi è la possibilità di ficcare un blocco { x => } dopo l'inizializzazione della risorsa, cioè di poter scrivere:

```
(pippo) { out =>
        out.pim
        out.pum
        out.pam
}
```

Alla fine della fiera, il corpo del metodo apply dell'object CallAndClose prima esegue l'espressione init poi passa il risultato di quell'espressione alla funzione body.
Passiamo ai tipi.

```
[T <: { def close(): Unit }, K]
```

K è un tipo qualsiasi. K è il valore restituito dalla funzione body. Poiché l'invocazione della funzione body è anche l'ultima espressione contenuta nel corpo del metodo apply, K è anche il tipo restituito dal metodo apply. Il compilatore inferisce questo tipo K quando si invoca la funzione apply: se gli rifilo una funzione che restituisce un intero, K sarà Int, se gli passo una funzione che sputa uno Unit, K sarà Unit. T è un pelo diverso.

```
T <: { def close(): Unit }
```

Il segno <: vuol dire “compatibile con”, “tipo o sottotipo di”, nello stesso senso in cui String è sottotipo di AnyRef o Long di Number o String di String. A sinistra del <: c'è di norma il nome del tipo con cui si vuole che T sia compatibile. Ad esempio:

```
T <: Number
```

significa che T è Number o un qualsiasi sottotipo di Number. Nel nostro caso abbiamo:

```
T <: { def close(): Unit }
```

Che significa che T è un qualsiasi tipo di dato compatibile con un tipo immaginario caratterizzato dal possesso di un metodo di nome close, senza argomenti, che restituisce uno Unit. La parte a destra racchiusa tra le graffe è detta tipo strutturale, perchè è identificato non da un nome ma dalla corrispondenza parziale con un certo insieme di membri. Per evitare di incasinare troppo la dichiarazione di genericità del metodo apply possiamo eventualmente usare la clausola type, in questi termini:

```
package magazzino.impl

object CallAndClose {

	type Closable = { def close(): Unit }

	def apply[T <: Closable, K](init: => T)(body: T => K) = {
		var statement: Option[T] = None
		try {
			statement = Some(init)
			body(statement.get)
		} finally statement.foreach(s => s.close)
```

```
    }
}
```

Il type genera un sinonimo per un certo tipo, in questo caso strutturale. Poi bisogna giudicare quale forma sia migliore. L'alias rende il codice un po' più ordinato ma di fronte ad un nome Closable l'utente della libreria potrebbe essere portato a cercare una classe o un trait o un object mentre Closable è di fatto un nick-name.
Il corpo di CallAndClose dichiara una variabile di tipo T (cioè quella roba che ha quantomeno un metodo close) il cui valore può non essere definito, apre un blocco try, assegna un valore alla variabile usando l'espressione init. Se l'init fallisce causa eccezione si passerà al finally che non troverà alcun valore in statement e quindi non farà nulla. Se l'init ha successo il suo valore sarà incapsulato in statement e lo stesso valore sarà dato in pasto alla funzione body. Il finally chiuderà la risorsa. E questo è tutto ciò che riguarda CallAndClose. Torniamo a MagazzinoH2.

MagazzinoH2: inizializzazione.

Il magazzino opera variamente sul tipo astratto Prodotto, dichiariamo un tipo interno che lo concretizzi così ci togliamo il pensiero:

```
package magazzino.impl

import java.util.logging.{Level, Logger}
import magazzino.{Prodotto, Magazzino}
import java.sql.{Connection, DriverManager}

trait MagazzinoH2 extends Magazzino {

    class ProdottoImpl(val codice: String) extends Prodotto
```

H2 è un database sql, date le modeste operazioni che dobbiamo compiere bastano e avanzano una manciata di query che dichiaramo così:

```
    val sqlCreaTabellaProdotti="CREATE TABLE prodotti (codice char(255), quantita char(50))"
    val sqlQuantitàProdotto = "SELECT quantita FROM prodotti WHERE codice=?"
    val sqlScanProdotti = "SELECT * FROM prodotti"
    val sqlImpostaQuantità = "UPDATE prodotti SET quantita=? WHERE codice=?"
    val sqlAggiungiProdotto = "INSERT INTO prodotti (codice, quantita) VALUES (?, ?)"
```

Comunissimo SQL di sopravvivenza. La prima stringa è la creazione dell'unica tabella che ci serve, la seconda è la query che restituisce la quantità di un prodotto, la terza esegue una scansione di tutti i prodotti, la quarta assegna una quantità ad un prodotto esistente, la quinta aggiunge un prodotto. Dopodichè, creiamo un "segnaposto" per la connessione al database:

```
    var connection: Option[Connection] = None
```

Option, niente null se si può evitare. Apriamo la connessione quando è invocato un metodo open, così definito:

```
    def open() {
        connection = Some(DriverManager.getConnection("jdbc:h2:~/magazzino", "sa", ""))
        try {
            connection.get.createStatement.execute(sqlCreaTabellaProdotti)
        } catch {
            case ex: Exception => System.out.println("tabella già esistente")
        }
        Logger.getLogger(classOf[MagazzinoH2].getName).log(Level.INFO, "connesso al database")
    }
```

Anche qui nulla da dire salvo forse il try-catch e il classOf. Il try-catch è il try catch a cui ormai

siamo assuefatti, in Scala ha la forma:

```
try espressione catch { case }
```

Nel nostro caso espressione è un blocco:

```
{ qualcosa }
```

Che si potrebbe anche omettere, scrivendo quindi:

```
try connection.get.createStatement.execute(sqlCreaTabellaProdotti) catch {
      case ex: Exception => System.out.println("tabella già esistente")
}
```

Bisogna giudicare quale sia la forma più chiara – è sempre una questione di punti di vista. Il catch deve avere un blocco che contiene dei case. I case sono correlati al pattern-matching, che è una degli strumenti di Scala. In pratica quando il try sputa un Throwable lo passa al blocco catch che verifica quale tra i pattern dichiarati dai suoi case corrisponde al Throwable ed esegue il corpo (la roba dopo il =>) di quel case. L'ultima parte del metodo open è la stampa di un messaggio di log per dire che siamo vivi. L'espressione:

```
classOf[NomeTipo]
```

è l'omologo del NomeTipo.class di Java. Nel caso di scala è una funzione generica, la minestra è più o meno quella. Fatto l'open ci diamo al close:

```
def close() = connection.foreach { connection =>
      connection.close()
      Logger.getLogger(this.getClass.getName).log(Level.INFO, "disconnesso dal database")
}
```

Co fu?. In scala la dichiarazione di un metodo ha la forma:

```
def nome liste_parametri = espressione
```

Le liste dei parametri sono opzionali. Non è necessario che espressione sia un blocco {}: il blocco {} in scala è solo una delle espressioni possibili. Nel nostro close l'espressione non è un blocco ma è l'invocazione di un altro metodo, il foreach di connection. Nel codice connection è un Option, un Option è come se fosse un insieme che può avere zero o un elemento. Se l'Option è indefinito, cioè è None o un Some il cui valore è null, ha zero elementi, se è un Some con un valore diverso da null ha un elemento. Il foreach è semplicemente un “fai qualcosa con ogni elemento della lista”, dove qualcosa è una funzione che accetta come argomento un valore del tipo dell'insieme. Siccome apriamo la connessione solo se è invocato il metodo open, quel foreach sull'Option che la incapsula ci dice che se la connessione è stata aperta, allora facciamo qualcosa, altrimenti ci giriamo i pollici. Finalmente arriviamo ai metodi concreti del magazzino.

MagazzinoH2: definizione del metodo foreach

I passaggi sono quelli canonici delle api jdbc della piattaforma Java:

creiamo uno statement usando la stringa sqlScanProdotti
lo eseguiamo ottenendo un ResultSet
per ogni riga del ResultSet creiamo un prodotto e lo passiamo alla funzione

Il tutto va decorato coi try-finally perchè sia il result set che lo statement sono risorse.

```
def foreach(op: (Prodotto) => Unit) = connection.foreach{ connection =>
    CallAndClose(connection.createStatement) { statement =>
        CallAndClose(statement.executeQuery(sqlScanProdotti)) { resultSet =>
            while(resultSet.next) {
                val codice = resultSet.getString(1)
                val prodotto = new ProdottoImpl(codice)
                op(prodotto)
            }
        }
    }
}
```

Per quanto detto su CallAndClose, qui abbiamo due try-finally annidati, con le variabili che prima valgono null e se entriamo nel try allora valgono altro e alla fine si chiude tutto e tanti saluti. Penso che si possa fare di meglio – le mie limitate abilità scalesi mi impongono di fermarmi qui – ma è già un passo avanti. Il succo è quello dei passaggi elencati: creiamo lo statement, eseguiamo la query, ci arriva un ResultSet, gli diamo una ripassata e via.

MagazzinH2: definizione del metodo quantità

Il metodo quantità, come foreach, ha due risorse, quindi anche qui abbiamo due CallAndClose. A differenza del metodo foreach che abbiamo appena visto, quantità restituisce un Option[BigDecimal]. Domanda: possiamo usare il truccone – nel senso estetico – di dire:

def quantità(p: Prodotto) = connection.foreach { connection => eccetera }

Che, ricordiamo, è in buona sostanza un if, che evita l'esecuzione del metodo se manchi la connessione, cioè se non sia ancora stato invocato il metodo open di MagazzinoH2.
Non possiamo: il foreach di Option ci rigurgita addosso uno Unit, che non è compatibile con Option[BigDecimal]. Quindi:

```
def quantità(p: Prodotto): Option[BigDecimal] = connection.flatMap{ connection =>
    var risultato: Option[BigDecimal] = None
    CallAndClose(connection.prepareStatement(sqlQuantitàProdotto)) { statement =>
        statement.setString(1, p.codice)
        CallAndClose(statement.executeQuery) { resultSet =>
            while(resultSet.next) {
                risultato = Some(BigDecimal(resultSet.getString(1)))
            }
        }
    }
    risultato
}
```

Il metodo flatMap di Option è come il foreach, esegue qualcosa solo se l'Option è definito, cioè è un Some con un valore diverso da null. A differenza del foreach restituisce un valore, precisamente il valore della funzione usata come argomento del flatMap. Quale valore restituiamo? Il valore di risultato. Noi creiamo 'sto risultato che all'inizio è None, è vuoto, ciccia, niente, perchè abbiamo una connessione (flatMap ha funzionato quindi connection ha un valore diverso da null ergo è stato invocato open) ma ancora non sappiamo se il prodotto richiesto sia presente nella base dati. Con questo dubbio, creiamo lo statement con la query che va a guardare la quantità di un prodotto e, a spalle coperte dai nostri CallAndClose, peschiamo la quantità dal ResultSet. Se c'è qualcosa allora il resultSet avrà una riga, quindi il while sarà eseguito e a risultato sarà assegnato il valore Some... la quantità. Altrimenti risultato resterà al suo valore iniziale None.
Volendo si può anche ridurre un po' il codice. Ad esempio il secondo CallAndClose potrebbe essere scritto su una sola riga:

```
CallAndClose(statement.executeQuery)(r => while(r.next) risultato = Some(BigDecimal(r.getString(1))))
```

Certo è che un conto è volendo, un conto è dovendo, e tutt'altro è capendo: si può sempre ridurre il codice ad una nebbia di caratteri ma non è detto che sia un bella cosa. Meno è meglio a parità si significato ma non di significato espresso, di significato compreso da chi legge. Per carità di patria, con la sintesi andiamoci coi piedi di piombo.

MagazzinoH2: definizione del metodo aggiungi

Il metodo aggiungi è un po' meno indentato del foreach e del quantità perchè abbiamo una sola risorsa da gestire, lo statementi. Comunque i casi sono due: o il prodotto è già stato inserito, e allora avrà un qualche valore e faremo un update con sqlImpostaQuantità, o il prodotto non è stato ancora inserito, e allora faremo un insert con sqlAggiungiProdotto. Quindi:

data la quantità corrente del prodotto
se è definita → eseguiamo sqlImpostaQuantità assegnando al prodotto il valore dello stock più la nuova quantità
se è indefinita → eseguiamo sqlAggiungiProdotto usando la quantità come valore iniziale

A me risulta:

```
def aggiungi(p: Prodotto, q: BigDecimal) = connection.foreach { connection =>
    val stock = quantità(p)
    if(stock.isDefined) CallAndClose(connection.prepareStatement(sqlImpostaQuantità)) { statement =>
        statement.setString(2, p.codice)
        statement.setString(1, (stock.get + q).toString)
        statement.execute()
    } else CallAndClose(connection.prepareStatement(sqlAggiungiProdotto)) { statement =>
        statement.setString(1, p.codice)
        statement.setString(2, q.toString)
        statement.execute()
    }
}
```

Direi che si potrebbe fare a meno dell'if o quantomeno ridurlo all'ordine dei setString ma non stiamo a spaccare il capello in quattro (anche perchè qui c'è poco da spaccare).

MagazzinoH2: definizione del metodo rimuovi

Abbiamo quasi finito. Il metodo rimuovi ha un sacco di se. Se c'è una connessione, come tutti, se c'è una quantità, come aggiungi, allora si può anche far qualcosa, sempre chiudendo tutte le porte che apriamo. Sia la connessione che la quantità, quest'ultima espressa dal risultato dell'invocazione del metodo omonimo, sono degli Option dunque i nostre “se” diventano dei foreach. Per quanto ne so questa faccenda del foreach è funzionale, nel senso che non c'è un “se sei diverso da null” nella prospettiva funziona perchè questa opera su tipi di dato espressi come insiemi che non sono mai null ma al limite vuoti. L'if != null diventa “funzione applicata a tutti gli elementi di un insieme vuoto” e quindi mai applicata. Che ha una sua eleganza se vogliamo perchè riporta il caso speciale null alla norma generale. Fine della digressione filosofica, ecco il metodo:

```
def rimuovi(p: Prodotto, q: BigDecimal) = connection.foreach { connection =>
    quantità(p).foreach{ stock => CallAndClose(connection.prepareStatement(sqlImpostaQuantità)) { stat =>
            stat.setString(2, p.codice)
            stat.setString(1, (stock - q).toString)
            stat.execute()
        }
    }
}
```

Se c'è una connessione e se tramite quella connessione risulta che p abbia una quantità in stock, allora CallAndClose. Senza se e con qualche ma, perchè il tentativo di rimuovere una quantià di prodotto che non è in stock dovrebbe, se fossimo nel contesto di un programma reale, farci drizzare le orecchie: forse l'utente ha ciccato il codice o il codice che gli hanno dato non è giusto. Non è detto che questi controlli devano essere fatti nella classe MagazzinoH2: potremmo benissimo immaginare che tra MagazzinoH2 e (vedremo) Terminale, ci sia uno strato intermedio anti-stupidata. Il filtro potrebbe dire: ok, io so che magazzino accetta tutto quindi prima di fargli eseguire la rimozione verifico che il prodotto sia in stock, che lo stock sia sufficiente all'eliminazione della quantità richiesta, magari che l'utente abbia i permessi necessari eccetera eccetera. Se fila tutto liscio, bene, altrimenti emetto un errore, chiamo la sicurezza, tiro un cazzotto all'utente, farò quello che posso.
Riepiloghiamo il codice.

MagazzinoH2: codice completo.

```
package magazzino.impl

import java.util.logging.{Level, Logger}
import magazzino.{Prodotto, Magazzino}
import java.sql.{Connection, DriverManager}

trait MagazzinoH2 extends Magazzino {
	class ProdottoImpl(val codice: String) extends Prodotto

	val sqlCreaTabellaProdotti="CREATE TABLE prodotti (codice char(255), quantita char(50))"
	val sqlQuantitàProdotto = "SELECT quantita FROM prodotti WHERE codice=?"
	val sqlScanProdotti = "SELECT * FROM prodotti"
	val sqlImpostaQuantità = "UPDATE prodotti SET quantita=? WHERE codice=?"
	val sqlAggiungiProdotto = "INSERT INTO prodotti (codice, quantita) VALUES (?, ?)"

	protected var connection: Option[Connection] = None

	def open() {
		connection = Some(DriverManager.getConnection("jdbc:h2:~/magazzino", "sa", ""))
		try {
			connection.get.createStatement.execute(sqlCreaTabellaProdotti)
		} catch {
			case ex: Exception => System.out.println("tabella già esistente")
		}
		Logger.getLogger(classOf[MagazzinoH2].getName).log(Level.INFO, "connesso al database")
	}

	def close() = connection.foreach { connection =>
		connection.close()
		Logger.getLogger(this.getClass.getName).log(Level.INFO, "disconnesso dal database")
	}

	def foreach(op: (Prodotto) => Unit) = connection.foreach{ connection =>
		CallAndClose(connection.createStatement) { statement =>
			CallAndClose(statement.executeQuery(sqlScanProdotti)) { resultSet =>
				while(resultSet.next) {
					val codice = resultSet.getString(1)
					val prodotto = new ProdottoImpl(codice)
					op(prodotto)
				}
			}
		}
	}

	def quantità(p: Prodotto): Option[BigDecimal] = connection.flatMap{ connection =>
		var risultato: Option[BigDecimal] = None
		CallAndClose(connection.prepareStatement(sqlQuantitàProdotto)) { statement =>
			statement.setString(1, p.codice)
			CallAndClose(statement.executeQuery) { resultSet =>
				while(resultSet.next) {
```

```
                        risultato = Some(BigDecimal(resultSet.getString(1)))
                    }
                }
            }
            risultato
        }

        def aggiungi(p: Prodotto, q: BigDecimal) = connection.foreach { connection =>
            val stock = quantità(p)
            if(stock.isDefined) CallAndClose(connection.prepareStatement(sqlImpostaQuantità)) { statement =>
                statement.setString(2, p.codice)
                statement.setString(1, (stock.get + q).toString)
                statement.execute()
            } else CallAndClose(connection.prepareStatement(sqlAggiungiProdotto)) { statement =>
                statement.setString(1, p.codice)
                statement.setString(2, q.toString)
                statement.execute()
            }
        }

        def rimuovi(p: Prodotto, q: BigDecimal) = connection.foreach { connection =>
            quantità(p).foreach{ stock => CallAndClose(connection.prepareStatement(sqlImpostaQuantità)) { stat =>
                    stat.setString(2, p.codice)
                    stat.setString(1, (stock - q).toString)
                    stat.execute()
                }
            }
        }
}
```

Avvio del programma.

Ci resta il main, il punto d'entrata del programma.

```
package magazzino.impl

object Main {

    def main(args: Array[String]) {
    }
}
```

Che si fa adesso? Abbiamo un terminale, che è un trait e un Magazzino astratto in parte definito da un MagazzinoH2 che è sempre un trait. Possiamo “fondere” Terminale e MagazzinoH2 in un object:

```
package magazzino.impl

object Main {

    def main(args: Array[String]) {
        object programma extends Terminale with MagazzinoH2 {
            val magazzino = this
        }
    }
}
```

E' come creare una classe che estende Terminale e MagazzinoH2 e poi dichiarare ed inizializzare un'istanza di quel tipo. Naturalmente MagazzinoH2 avrebbe anche potutto essere una classe sottotipo di Magazzino e Terminale una classe tout-court, oppure avrebbero potuto essere entrambi degli object (anche se qui saremmo entrati nel campo dei singleton), insomma, ci sono un po' di opzioni di design. A me interessava far vedere questa cosa dell'object usato per comporre più trait. La forma “extends … with” è come dire che extends una cosa e extends anche un'altra cosa. Per una

qualche ragione in Scala anziché dire:

```
extends Pippo, Pluto, Paperino
```

o dire

```
with Pippo, Pluto, Paperino
```

o ancora

```
extends Pippo extends Pluto extends Paperino
```

piuttostochè

```
with Pippo with Pluto with Paperino
```

si dice:

```
extends Pippo with Pluto with Paperino
```

Con buone ragioni si può estendere una sola classe e derivare da quanti trait si desidera. Nel codice si nota che programma dichiara un val, magazzino, che punta a this. Quel val magazzino è il val magazzio dichiarato ma non definito nel trait Terminale. Pochè programma concretizza Terminale, è obbligato a dare una definizione per questo val. Quel val, per Terminale, è il magazzino che sarà bersagliato dalle richieste utente. Poiché programma combina un terminale con un magazzino, programma usa sé stesso (in quanto magazzino) come bersaglio di... sé stesso (ma in quanto Terminale). Combinato un magazzino con un terminale, avviamo il tutto:

```
package magazzino.impl

object Main {

	def main(args: Array[String]) {
		object programma extends Terminale with MagazzinoH2 {
			val magazzino = this
		}
		CallAndClose(programma) { _ =>
			programma.open
			programma.mainMenu
		}
	}
}
```

Il simbolo _ nella funzione usata come parametro della seconda lista di argomenti del metodo apply dell'object CallAndClose (pausa per riprendere fiato), è usato in concreto perchè, avendo a disposizione il riferimento programma e sapendo che è quello il valore gestito da CallAndClose, dell'argomento non ci importa un fico secco. Lo stesso simbolo _ può essere usato in Scala in vari contesti e con vari traguardi (denota funzioni parzialmente applicate, valori indefiniti, insiemi nelle clausole di importazione e via dicendo).

Il codice del tutorial.

Di seguito il codice completo del programma.

//Stock.scala

```
package magazzino

trait Stock extends Function1[Magazzino, Option[BigDecimal]] {

	val prodotto: Prodotto

	def apply(m: Magazzino) = m.quantità(prodotto)
}
```

//Scarico.scala

```
package magazzino

trait Scarico extends Function1[Magazzino, Unit] {

	val prodotto: Prodotto

	val quantità: BigDecimal

	def apply(m: Magazzino) = {
		m.rimuovi(prodotto, quantità)
	}
}
```

//Prodotto.scala

```
package magazzino

trait Prodotto {

	val codice: String
}
```

//Carico.scala

```
package magazzino

trait Carico extends Function1[Magazzino, Unit] {

	val prodotto: Prodotto

	val quantità: BigDecimal

	def apply(m: Magazzino) = {
		m.aggiungi(prodotto, quantità)
	}
}
```

//Magazzino.scala

```
package magazzino

trait Magazzino {

	def quantità(p: Prodotto): Option[BigDecimal]

	def aggiungi(p: Prodotto, q: BigDecimal)

	def rimuovi(p: Prodotto, q: BigDecimal)

	def foreach(op: Prodotto => Unit)

	def esegui[T](op: (Magazzino) => T) = op(this)
}
```

//Operazioni.scala

```
package magazzino.impl

import magazzino.{Stock, Scarico, Carico, Prodotto}

class CaricoBase(val prodotto: Prodotto, val quantità: BigDecimal) extends Carico

class ScaricoBase(val prodotto: Prodotto, val quantità: BigDecimal) extends Scarico

class StockBase(val prodotto: Prodotto) extends Stock
```

//Terminale.scala

```
package magazzino.impl

import magazzino.{Prodotto, Magazzino}

trait Terminale {

	val magazzino: Magazzino

	def stock() {
		val console = System.console
		val choice = console.readLine("\nInserire il codice del prodotto e premere invio:")
		object prodotto extends Prodotto { val codice = choice }
		val quantità = magazzino.esegui(new StockBase(prodotto))
		if(quantità.isDefined) {
			console.printf("Codice: %s%nQuantità: %s%n", prodotto.codice, quantità.get)
		} else {
			console.printf("Il codice %s non corrisponde ad alcun prodotto immagazzinato%n", choice)
		}
		mainMenu()
	}

	def carico() {
		val console = System.console
		val cod = console.readLine("\nInserire il codice del prodotto da caricare e premere invio:")
		val quantità = console.readLine("\nInserire la quantità di prodotto da caricare e premere invio:")
		object prodotto extends Prodotto { val codice = cod }
		magazzino.esegui(new CaricoBase(prodotto, BigDecimal(quantità)))
		mainMenu()
	}

	def scarico() {
		val console = System.console
		val cod = console.readLine("\nInserire il codice del prodotto da scaricare e premere invio:")
		val quantità = console.readLine("\nInserire la quantità di prodotto da scaricare e premere invio:")
		object prodotto extends Prodotto { val codice = cod }
		magazzino.esegui(new ScaricoBase(prodotto, BigDecimal(quantità)))
		mainMenu()
	}

	def esci() {

	}

	def mainMenu() {
		val operazioni = Array[() => Unit](stock, carico, scarico, esci)
		val menu =
			"\nMenu Magazzino\n" +
			"1 Stock\n" +
			"2 Carico\n" +
			"3 Scarico\n" +
			"4 Esci\n" +
			"Digitare il numero di un'operazione (1, 2, 3, 4) e premere invio:"
		val console = System.console
		val choice = console.readLine(menu)
		val option = choice.toInt
		val operazione = operazioni(option - 1)
		operazione.apply()
	}
}
```

//CallAndClose.scala

```
package magazzino.impl

object CallAndClose {

	def apply[T <: { def close(): Unit }, K](init: => T)(body: T => K) = {
		var statement: Option[T] = None
		try {
			statement = Some(init)
			body(statement.get)
		} finally statement.foreach(s => s.close)
	}
}
```

//Magazzino.scala

```
package magazzino.impl

import java.util.logging.{Level, Logger}
import magazzino.{Prodotto, Magazzino}
import java.sql.{Connection, DriverManager}

trait MagazzinoH2 extends Magazzino {
	class ProdottoImpl(val codice: String) extends Prodotto

	val sqlCreaTabellaProdotti="CREATE TABLE prodotti (codice char(255), quantita char(50))"
	val sqlQuantitàProdotto = "SELECT quantita FROM prodotti WHERE codice=?"
	val sqlScanProdotti = "SELECT * FROM prodotti"
	val sqlImpostaQuantità = "UPDATE prodotti SET quantita=? WHERE codice=?"
	val sqlAggiungiProdotto = "INSERT INTO prodotti (codice, quantita) VALUES (?, ?)"

	protected var connection: Option[Connection] = None

	def open() {
		connection = Some(DriverManager.getConnection("jdbc:h2:~/magazzino", "sa", ""))
		try {
			connection.get.createStatement.execute(sqlCreaTabellaProdotti)
		} catch {
			case ex: Exception => System.out.println("tabella già esistente")
		}
		Logger.getLogger(classOf[MagazzinoH2].getName).log(Level.INFO, "connesso al database")
	}

	def close() = connection.foreach { connection =>
		connection.close()
		Logger.getLogger(this.getClass.getName).log(Level.INFO, "disconnesso dal database")
	}

	def foreach(op: (Prodotto) => Unit) = connection.foreach{ connection =>
		CallAndClose(connection.createStatement) { statement =>
			CallAndClose(statement.executeQuery(sqlScanProdotti)) { resultSet =>
				while(resultSet.next) {
					val codice = resultSet.getString(1)
					val prodotto = new ProdottoImpl(codice)
					op(prodotto)
				}
			}
		}
	}

	def quantità(p: Prodotto): Option[BigDecimal] = connection.flatMap{ connection =>
		var risultato: Option[BigDecimal] = None
		CallAndClose(connection.prepareStatement(sqlQuantitàProdotto)) { statement =>
			statement.setString(1, p.codice)
			CallAndClose(statement.executeQuery) { resultSet =>
				while(resultSet.next) {
					risultato = Some(BigDecimal(resultSet.getString(1)))
				}
			}
		}
		risultato
	}

	def aggiungi(p: Prodotto, q: BigDecimal) = connection.foreach { connection =>
		val stock = quantità(p)
		if(stock.isDefined) CallAndClose(connection.prepareStatement(sqlImpostaQuantità)) { statement =>
			statement.setString(2, p.codice)
			statement.setString(1, (stock.get + q).toString)
			statement.execute()
		} else CallAndClose(connection.prepareStatement(sqlAggiungiProdotto)) { statement =>
			statement.setString(1, p.codice)
			statement.setString(2, q.toString)
			statement.execute()
		}
	}
```

```
    def rimuovi(p: Prodotto, q: BigDecimal) = connection.foreach { connection =>
        quantità(p).foreach{ stock => CallAndClose(connection.prepareStatement(sqlImpostaQuantità)) { stat =>
                stat.setString(2, p.codice)
                stat.setString(1, (stock - q).toString)
                stat.execute()
            }
        }
    }
}
```

//Main.scala

```
package magazzino.impl

object Main {

	def main(args: Array[String]) {
		object programma extends Terminale with MagazzinoH2 {
			val magazzino = this
		}
		CallAndClose(programma) { _ =>
			programma.open
			programma.mainMenu
		}
	}
}
```