
UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di laurea triennale in Ingegneria dell'Informazione

**Tesina di Laurea**

# Scala – Scalable Language

Laureando: Cason Renato

Relatore: Prof. Moro Michele

A.A. 2012/2013

# Indice

# 1 Introduzione

Scala, il cui nome è la composizione delle parole *Scalable Language*, è un linguaggio di programmazione relativamente recente. La progettazione inizia nel 2001 dalla mente di Martin Odersky, persona nota sia in ambito accademico come professore alla EPFL -Politecnico Federale di Losanna- sia all'esterno per aver lavorato ad esempio su Java, nel compilatore e nell'introduzione della programmazione generica.

La caratteristica principale di Scala è l'essere un linguaggio a paradigma misto. Fin dall'inizio, infatti, per ottenere un linguaggio scalabile -ossia adatto ad una grande varietà di situazioni, dagli script interpretati a grandi applicazioni distribuite- l'idea è stata di unire la programmazione ad oggetti, la programmazione funzionale e un articolato sistema di tipizzazione, mantenendo allo stesso tempo una sintassi sintetica ed elegante. La cosa può sembrare strana, dal momento che programmazione funzionale e programmazione ad oggetti forniscono un approccio completamente diverso alla risoluzione dei problemi. Nel primo caso, infatti, si raggiunge l'obiettivo con la definizione e la combinazione di funzioni, mentre nel secondo con la definizione e combinazione di oggetti. Dal punto di vista concettuale ci si focalizza rispettivamente su verbi e su nomi. Volendo descrivere l'accesso di un utente ad un mezzo pubblico, ad esempio, nel caso della programmazione funzionale si dovrebbero definire due verbi: l'acquisto del biglietto e l'ingresso al mezzo. La prima azione avrà come parametro una persona con sufficiente quantità di denaro e restituirà una persona in possesso di un biglietto, mentre la seconda azione richiederà una persona in possesso di biglietto. Si noti che la persona, il denaro e il biglietto sono entità distinte che vanno sotto il nome di tratti, argomento che verrà approfondito nella relativa sezione. Nel caso della programmazione ad oggetti si andrebbe invece a definire un nome, *persona*, sul quale richiamare prima il metodo di acquisto del biglietto richiedente come parametro un ammontare di denaro, ed in seguito il metodo per l'ingresso sul mezzo pubblico. Compresa la profonda differenza tra i due paradigmi di programmazione, si può immaginare che vantaggi possa comportare la possibilità di scegliere e combinare le due metodologie all'interno di un unico programma. Ciò che appare inizialmente come un'incompatibilità si rivela quindi essere un'interessante sinergia.

Un'altra caratteristica fondamentale alla quale si accennava in precedenza è il sistema di tipizzazione. Scala è un linguaggio di programmazione staticamente tipato: ciò significa che il tipo di una variabile viene assegnato nel momento in cui viene creata e non può più essere modificato, al contrario di quanto accade nel linguaggi dinamicamente tipati. Il fatto che in Scala sia possibile dichiarare variabili anche senza specificarne il tipo, cosa che fornisce ai linguaggi dinamicamente tipati un grosso vantaggio in semplicità e sintesi del codice, può sembrare una contraddizione rispetto all'affermazione appena fatta. È l'inferenza di tipo che rende la cosa possibile, ciò significa che il tipo è comunque stabilito al momento della compilazione. Inoltre, in ottemperanza del paradigma ad oggetti, in Scala ogni entità è un oggetto. In particolar modo non esistono tipi primitivi come invece accade in altri linguaggi come ad esempio Java o C#. Un'altra mancanza di Scala rispetto a questi ultimi due linguaggi citati sono i membri statici. A compensare tale assenza ci sono i *singleton*. Questi sono dei costrutti di oggetto che, similmente all'omonimo pattern, possono avere un'unica istanza all'interno di un'applicazione e per la stessa esiste un punto di accesso globale. Infine non si possono non citare i meccanismi linguistici messi a disposizione da Scala e che danno un grande apporto per quanto riguarda appunto la scalabilità del linguaggio. Questi sono i *self-type* -tipi espliciti per la classe corrente-, tipi generici e membri di tipo astratto, classi annidate, composizione tramite tratti di *mixin*. Grossomodo si può dire che la prima caratteristica è particolarmente utile nello scripting, la seconda e la terza sono di comune utilizzo nella realizzazione di progetti di ogni dimensione e l'ultima è molto utile in progetti particolarmente ampi ed articolati.

Ultimo, ma non meno importante, è il fatto che i sorgenti Scala siano compilati in *bytecode* per la Java Virtual Machine. I vantaggi sono molteplici. Una conseguenza ad esempio è che un programma scritto con Scala è multi-piattaforma, e supporta tutte le architetture e i sistemi operativi in cui si può installare il Java 2 Runtime Environment. Per fare alcuni esempi, sono sicuramente supportati Windows, Linux e Mac OS X. Un altro grosso vantaggio sta nella possibilità di usare librerie scritte in Java all'interno di un programma Scala e, specularmente, di usare librerie scritte in Scala all'interno di applicazioni scritte in Java. Non si possono trascurare inoltre i vantaggi che comporta l'uso di una piattaforma come la JVM che vanta stabilità e ottimizzazioni acquisite in decenni di evoluzione ed utilizzo. Oltre ad essere compilabile per la JVM, un progetto più recente rende Scala compilabile per la piattaforma .NET, in cui

è presente come con Java la possibilità di scrivere codice che sfrutti l'interoperabilità con applicazioni o librerie scritte in C#, Visual Basic.NET, J#, F# e via dicendo.

Per quanto riguarda l'installazione di Scala, si consiglia di fare riferimento al sito ufficiale *www.scala-lang.org*. Nella sezione Downloads sono disponibili le ultime versioni di Scala per Windows, Linux e Mac OS X. È presente inoltre una sezione dedicata ai plugin per gli IDE, tra cui Eclipse e Netbeans, noti in ambito Java. Si consiglia l'uso di uno di questi strumenti per usufruire delle funzionalità di *syntax highlighting* e *code hint*.

Una volta installato l'ambiente di sviluppo è possibile usare Scala in diverse modalità. Innanzitutto è disponibile una *shell* interattiva, che si può richiamare con il comando *scala* da terminale. Da qui sarà possibile immettere e valutare riga per riga una sequenza di istruzioni. Tale modalità viene chiamata REPL, ovvero Read-Print-Eval-Loop. La seconda modalità consiste nell'eseguire un file *.scala* come script in modalità interpretata, senza compilarlo. Ciò è possibile invocando il comando *scala* seguito dal nome del file da eseguire. Infine è possibile compilare uno o più file ed eseguire l'applicativo su JVM. Per fare ciò è necessario in primo luogo lanciare il comando *scalac* sui file da compilare, ed in seguito lanciare il comando *scala* sul file *.class* che costituisce l'entry point dell'applicazione. In tutte e tre le modalità è possibile effettuare l'import di classi Java, come verrà mostrato nell'apposito capitolo.

# 2 Sintassi

In questo capitolo si affronteranno le principali caratteristiche di Scala dal punto di vista della sintassi. Dalla definizione di variabili, oggetti e classi ai principali costrutti il linguaggio mette a disposizione, in modo da poter affrontare temi più approfonditi nei capitoli successivi.

## *2.1 Variabili e costanti*

Scala fornisce due modalità nella creazione di variabili. La parola chiave *var* permette di dichiarare ed istanziare un oggetto modificabile, mentre la parola chiave *val* crea un oggetto immutabile. Quest'ultima modalità è analoga a *final* su Java e a *const* su C#.

```
val i : Integer = 1
var j : Double = 2
```

In entrambi i casi la sintassi è la stessa. La linea di codice inizia con la parola chiave, a seguire c'è il nome della variabile, dopo i due punti il tipo della variabile ed infine, dopo l'uguale, il valore della variabile.
L'uso di variabili immutabili, quando possibile, è da preferire rispetto all'uso di variabili mutabili per il guadagno che comportano in termini di prestazioni. Questo specialmente nell'ambito della programmazione concorrente. Si noti che, rispetto all'esempio sopra, un'istruzione come quella riportata in seguito comporta un errore in fase di compilazione.

```
i = 10
```

Questo perché si sta cercando di modificare il valore di una variabile immutabile. Lo stesso errore si riscontra anche quando viene modificata qualche proprietà di un oggetto dichiarato con la parola chiave *val*.
Per chi arriva dal mondo del C o di Java, la cosa più evidente sarà la mancanza dei punti e virgola come terminatori della linea di codice. In automatico Scala interpreta la fine della riga come la fine dell'istruzione, tranne quando è ovvio il contrario. Questo permette di tralasciare il punto e virgola nei casi in cui non inseriamo più istruzioni per riga.
Come menzionato nell'introduzione, Scala supporta l'inferenza di tipo. L'esempio precedente può essere realizzato anche nel seguente modo:

```
val i = 1
var j = 2.0
```

In questo modo non abbiamo dovuto scrivere esplicitamente il tipo, ma è stato automaticamente dedotto dal compilatore. Si noti che nella seconda istruzione è stato necessario apporre un punto ed uno zero a seguito del numero. per fare in modo che il *2* venisse interpretato dal compilatore come variabile *double* e non come intero. La notazione composta dal solo punto, presente in Java e C#, è deprecata dal compilatore Scala e non sarà più utilizzabile dalla versione 2.11.0.
C'è la possibilità di usare anche il tipo *float*, ma in questo caso è necessario usare la seguente modalità, dal momento che il tipo predefinito per i numeri decimali è *double*.

```
var z = 3F
```

Così facendo la variabile *z* sarà di tipo *float*, ed avrà 3 come valore.

## *2.2 Definizione di Classi*

In Scala la sintassi per la definizione delle classi è sostanzialmente diversa da quella di Java o C#. Riportiamo in seguito un esempio per comprendere le regole fondamentali del linguaggio.

```
class Rettangolo (
                    var larghezza : Double,
                    var altezza : Double
      ) {
      def this() {
                    this(0, 0)
      }
      def area = {
                    larghezza * altezza
      }
}
```

Il modo in cui le variabili vengono definite è stato analizzato nel paragrafo precedente, la particolarità sta nel luogo in cui vengono definite. Nell'esempio i campi *larghezza* e *altezza* sono definiti separati da virgole tra parentesi tonda, prima dell'apertura delle parentesi graffe. Definire i campi in tale luogo non è indispensabile, sarebbe stato possibile fare lo stesso all'interno delle parentesi graffe. I vantaggi della definizione per com'è fatta nell'esempio sono principalmente due, ovvero la generazione automatica dei metodi di accesso alle variabili *larghezza* e *altezza* e del costruttore che accetta questi due valori come parametri.
Specificare delle variabili tra parentesi tonde appena dopo la dichiarazione della classe fa quindi in modo che il compilatore generi un costruttore con la stessa firma, ed è proprio per questo che nell'esempio è stato aggiunto un secondo costruttore senza parametri che chiama il primo impostando a zero il valore delle due proprietà.
Dopo il costruttore viene definito un metodo senza parametri, che calcola l'area moltiplicando base e altezza del rettangolo. La prima cosa che si nota è la notazione, molto simile al Javascript. Con la parola chiave *def* si va a creare un oggetto, e tramite l'uguale si va ad assegnare a quest'oggetto l'implementazione della funzione come sequenza di azioni racchiuse da parentesi graffe.
Usando l'interprete Scala in modalità di scripting si potrà testare l'uso della classe nel seguente modo:

```
var r = new Rettangolo
r.larghezza = 10
r.altezza = 15
println(r.area)
```

Oppure, facendo uso del costruttore generato in automatico dal compilatore:

```
var r = new Rettangolo(10, 15)
println(r.area)
```

In entrambi i casi il risultato sarà 150, seguito ad un punto che consegue dal fatto che il tipo di ritorno della funziona area, pur non essendo dichiarato esplicitamente, è *double*.
A questo punto la dichiarazione di una funzione con tipo di ritorno dichiarato esplicitamente è assai intuitiva. Riprendendo l'esempio di prima, per forzare il tipo di ritorno della funziona *area* a *double*, si può dichiarare la funzione in questo modo:

```
def area : Double = {
	larghezza * altezza
}
```

Un'altra cosa che risalta è la mancanza della parola chiave *return*. Questa su Scala esiste ancora, ma non è obbligatoria. In mancanza di altre indicazioni, il valore prodotto dall'ultima istruzione di ogni funzione viene interpretato come valore di ritorno.
Il prossimo esempio mostrerà alcuni lati più avanzati della definizione degli oggetti. Come anticipato nell'introduzione, su Scala ogni cosa è un oggetto, non esistono tipi primitivi. Ciò suggerisce che anche gli operatori di somma, moltiplicazione e via dicendo siano delle funzioni definite tra oggetti. Nell'esempio che segue definiremo una classe che rappresenta un numero complesso, implementando il metodo di somma a livello statico, richiamandolo a livello di istanza con il simbolo + ed effettuando l'*override* del metodo *toString* in modo da poter utilizzare il metodo *println* per visualizzare il contenuto effettivo della variabile.

```
class Complesso (
	val x : Double,
	val y : Double
) {
	import Complesso._

	def +(c : Complesso) : Complesso = {
			somma(this, c)
	}

	override def toString() : String = {
			x + " + i" + y
	}
}

object Complesso {
	def somma(a : Complesso, b : Complesso) = {
			new Complesso(a.x + b.x, a.y + b.y)
	}
}
```

Partendo dall'alto, la prima differenza che si nota è che *x* e *y* sono state definite come variabili immutabili con la parola chiave *val*, quindi durante il periodo di vita dell'oggetto non sarà possibile cambiarne il valore. L'istruzione seguente è un *import*, che nel contesto dell'esempio serve ad importare oggetti e funzioni dichiarate nel *companion object*, l'oggetto singleton definito dopo la classe. La prima funzione definita nella classe, come si più notare, risponde al simbolo +, richiede come argomento un oggetto di tipo Complesso e restituisce un oggetto dello stesso tipo, demandandone il calcolo effettivo al metodo *somma* nel *companion object*, in italiano *oggetto associato*. Come si può notare la definizione della funzione statica non è diversa dalla definizione delle funzioni non statiche, cambia soltanto il luogo in cui sono definite. All'interno di una *class* le funzioni sono definite come d'istanza, mentre all'interno di *object* sono definite come statiche.
Il secondo metodo all'interno della classe Complesso, per concludere, esegue l'*override* del metodo

*toString* in modo da restituire un testo del tipo *x* + i*y*. Da notare l'uso della parola chiave *override*, in modo analogo a quanto succede nel C#.
Per testare quanto appena fatto andremo ad istanziare due variabili complesse nel seguente modo:

```
var a = new Complesso(10, 5)
var b = new Complesso(1, 7)
```

Per calcolare e visualizzare la somma di questi due numeri abbiamo diverse possibilità. Se vogliamo usare il metodo statico la sintassi è la seguente:

```
println(Complesso.somma(a, b))
```

Ma possiamo ottenere lo stesso risultato in maniera molto più elegante e concisa, grazie al metodo d'istanza + che abbiamo creato. La sintassi è la seguente:

```
println(a + b)
```

Con i valori inseriti nell'esempio, in entrambi i casi il risultato è *11.0 + i12.0*.
In Scala si possono inoltre definire le *funzioni curry*. Questa modalità permette sostanzialmente di definire una funzione con più parametri come una catena di funzioni con un singolo parametro. Il seguente esempio è una modifica dell'oggetto associato *Complesso*, che implementa l'operazione somma tramite una *funzione curry* anziché una funzione tradizionale.

```
object Complesso {
	def somma(a : Complesso)(b : Complesso) = {
			new Complesso(a.x + b.x, a.y + b.y)
	}
}
```

Prendendo come riferimento i due numeri complessi *a* e *b* definiti in precedenza, la chiamata al metodo con stampa del risultato avrà la seguente sintassi.

```
println(Complesso.somma(a)(b))
```

## *2.3 Tratti*

Tra gli strumenti più interessanti messi a disposizione da Scala rispetto ad altri linguaggi di comune diffusione ci sono sicuramente i tratti. Utilizzando ad esempio Java e C# si possono utilizzare le interfacce e le classi astratte per sviluppare una serie di oggetti che implementano le caratteristiche specificate. Questi strumenti hanno però delle limitazioni, dal momento che le interfacce non possono contenere l'implementazione dei metodi che definiscono, mentre le classi astratte lo permettono, ma allo stesso tempo non è possibile definire una classe che estende più di una classe astratta. Risulta quindi evidente l'esistenza di una serie di casistiche non ottimali, in cui il codice non più essere ereditato e quindi lo sviluppatore deve ricorrere ad espedienti come la replicazione dello stesso.

Dunque tramite i tratti Scala permette di realizzare dei *mixin*, ovvero delle classi che contengono una combinazione di metodi provenienti da altre classi. Nel caso specifico si tratta di ereditarietà multipla, in quanto le classi derivate dai tratti ne ereditano tutti i metodi non privati.

Vediamo in seguito la definizione di due tratti.

```
trait Motore {
	def potenzaHp : Double

	def potenzaKwatt : Double = {
		potenzaHp * 1.34
	}
}
trait Rimorchio {
	def larghezza : Double
	def lunghezza : Double
	def altezza : Double

	def capienza : Double = {
		larghezza * lunghezza * altezza
	}
}
```

Si può vedere come i tratti siano stati definiti allo stesso modo delle classi. Lo stesso vale per i metodi, ma non per le proprietà. Essendo il valore di queste specificato nelle classi che estenderanno i tratti, sono definite con la parola chiave *def*, lasciando ad un secondo momento la scelta tra varianza e invarianza.

In entrambi i tratti definiti dei metodi contenenti la relativa implementazione, nel seguente esempio è riportata una classe che li estende e valorizza i parametri necessari al calcolo.

```
class Autocarro extends Motore with Rimorchio {
	val potenzaHp = 100.0
	val larghezza = 3.0
	val lunghezza = 5.0
	val altezza = 2.0
}
```

A questo punto non resta che testare il funzionamento dei metodi. Nel seguente esempio la classe *Autocarro* viene istanziata ed il valore in uscita di questi viene mostrato a video.

```
val a = new Autocarro
println(a.potenzaKwatt)
println(a.capienza)
```

Il risultato ottenuto si può raggiungere anche senza la definizione della classe *Autocarro*. Scala infatti permette di istanziare delle classi anonime a partire da altre classi o tratti, come mostra il seguente esempio.

```
val b = new Motore with Rimorchio {
	val potenzaHp = 100.0
	val larghezza = 3.0
	val lunghezza = 5.0
	val altezza = 2.0
```

```
}
println(a.potenzaKwatt)
println(a.capienza)
```

Questa notazione risulta particolarmente utile nel caso in cui l'insieme di più tratti sia utilizzato ad esempio una sola volta, e quindi non si giustifichi la creazione di una classe apposita.

Chiaramente la modalità di funzionamento dei tratti solleva un quesito. Cosa succede nel caso una classe estenda due tratti che implementano lo stesso metodo? Il problema è un classico dell'ereditarietà multipla, in inglese viene definito *diamond problem* e Scala lo risolve applicando la linearizzazione. Il seguente esempio crea la situazione ideale per riprodurre il problema.

```
abstract class A {
	def stato() : Boolean
}
trait B extends A {
	override def stato() : Boolean = true
}
trait C extends A {
	override def stato() : Boolean = false
}
```

Data una classe astratta *A* contenente metodo non implementato, sono stati definiti due tratti. Il primo, *B*, al suddetto metodo restituisce *true*, ed il secondo, *C*, restituisce *false*. Quindi, istanziando una classe che estende sia *B* che *C*, il metodo restituirà *true* o *false*?

La risposta dipende dallo sviluppatore. Nello specifico, dipende dall'ordine in cui decide di estendere i tratti. Ad esempio, la seguente dichiarazione mostrerà in uscita *true*.

```
val c = new A with C with B
println(c.stato)
```

Al contrario, la seguente dichiarazione mostrerà *false*.

```
val c = new A with B with C
println(c.stato)
```

Si può notare chiaramente l'ordine in cui vengono estesi i tratti determini il risultato del metodo. L'ultimo tratto esteso, infatti, è quello che fornisce l'implementazione dominante in caso di conflitto.

## *2.4 Operazioni binarie*

Oltre alle classiche operazioni numeriche, Scala permette l'utilizzo di operazioni bit a bit che possono tornare utili in particolari elaborazioni.

La più semplice è la negazione, operazione unaria e rappresentata dal simbolo *tilde* (~). Una variabile *a* dal valore binario *1011*, ad esempio, se negata assumerebbe il valore *~a = 0100*.

Le altre operazioni sono binarie: *AND*, rappresentata dal simbolo *&*, *OR* dal simbolo | e *XOR* dal simbolo ^. A seguire la tabella di verità degli operatori elencati.

| p | q | p & q | p \| q | p ^ q |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

Infine ci sono le operazioni di shift bit a bit. È possibile fare lo shift a sinistra con il simbolo <<, lo shift a destra con il simbolo >>, e lo shift a destra con riempimento a zero con il simbolo >>>.

## *2.5 Costrutti*

I costrutti messi a disposizione da Scala sono sostanzialmente quelli presenti in tutti gli altri linguaggi moderni. Tuttavia, essendo anche un linguaggio di programmazione funzionale, ci saranno alcune funzionalità avanzate per l'attraversamento e l'elaborazione delle liste.

### 2.5.1 Costrutto if

Il costrutto if segue la stessa sintassi e gli stessi comportamenti dell'analogo su Java e C#. Vediamo un esempio basilare:

```
def modulo(n : Integer) = {
	if(n < 0)
		-n
	else
		n
}
```

Istruzioni su più righe vanno scritte tra parentesi graffe. Gli elementi di concatenazione delle condizioni (*and* e *or*) e di confronto (uguale, diverso, eccetera) sono del tutto analoghi a quelli dei linguaggi sopra citati. Ciò che rende l'*if* su Scala diverso da quello di altri linguaggi è il fatto che può essere interpretato come funzione. Nell'esempio riportato sopra può sembrare che della variabile *n* con lo stesso segno o con il segno invertito venga effettuato un *return* direttamente al metodo *modulo*, ma così non è. L'*if* viene valutato come funzione, ed il suo risultato *n* o *-n* viene poi elaborato come valore di ritorno del metodo. Un altro esempio è il seguente:

```
val m = if(n < 0)
		-n
	else
		n
```

In questo caso si nota meglio come l'elaborazione dell'*if* consista in una vera e propria funzione, in quanto il valore viene assegnato direttamente ad una variabile. Il tipo della variabile *m* viene determinato tramite l'inferenza di tipo. Nell'esempio equivarrà al tipo di *n*, ma in generale, se *if* ed *else* differiscono no. Se ad esempio uno restituisce un Double e l'altro un Integer il tipo di ritorno sarà Double, ma se i due tipi non hanno derivazioni compatibili il tipo finale sarà Any. L'analogo di Object in Java, ovvero la radice della gerarchia dei tipi.

## 2.5.2 Costrutto Match

Il costrutto *match* è l'analogo di ciò che in altri linguaggi è più noto con il nome di *switch*. La differenza rispetto C# e Java sta più che altro sul nome, in quanto la sostanza è la stessa. Vediamo un esempio:

```
def messaggioErrore(codice : Int) : String = codice match {
       case 404 => "Pagina non trovata"
       case 500 => "Errore interno del server"
       case 503 => "Permesso negato"
       case _ => "Errore sconosciuto"
}
```

In questo caso facciamo un uso da manuale del *match*, ovvero come alternativa ad una serie di *if*. Come anticipato la sintassi non porta grandi differenze rispetto ad altri linguaggi, tranne che per l'uso del carattere *underscore* ad identificare la casistica di *default*. Alla pari del costrutto precedente, anche il *match* è visto come una funzione. Nell'esempio sfruttiamo proprio questa caratteristica, assegnando alla funziona *messaggioErrore* direttamente il risultato del costrutto, senza istanziare una nuova variabile, assegnarle il valore a seconda del parametro e restituirla in seguito.
Pur avendo una sintassi molto simile a quella dei linguaggi tradizionali, il costrutto *match* in Scala ha delle funzionalità molto interessanti nell'ambito del pattern matching sul quale vale sicuramente la pena di spendere qualche esempio in più. Per cominciare si possono fare distinzioni in base al tipo di oggetto, come in altri linguaggi si può fare all'interno di costrutti *if* usando parole chiave come *is* o *instanceof*:

```
def matchType(o : Any) : String = o match {
       case i : Int => "Integer: " + i
       case f : Float => "Float: " + f
       case d : Double => "Double: " + d
       case s : String => "String: " + s
       case _ => "Tipo non riconosciuto"
}
```

Ad un livello ancora più dettagliato, Scala permette tramite il *match* di comparare degli oggetti per i rispettivi contenuti o per il contenuto di determinate proprietà, se le classi sono definite con la parola chiave *case*. Per questo motivo vengono comunemente chiamate *classi case*.

```
case class Complesso (x : Double, y : Double)

def matchComplex(c : Complesso) : String = c match {
       case Complesso(0, 0) => "Origine"
       case Complesso(_, 0) => "Appartenente all'asse reale"
       case Complesso(0, _) => "Appartenente all'asse immaginario"
       case _ => "Appartenente al piano"
}
```

Nell'esempio qui sopra la classe *Complesso* è stata definita con la parola chiave *case*. Questo ci ha permesso di definire delle condizioni di comparazione esatte, come nel primo caso in cui si isola il caso dell'origine, e fisse per un solo valore come nei due casi successivi. Da notare la priorità delle casistiche equivale all'ordine con cui sono elencate. In questo caso, ad esempio, se la condizione

sull'origine fosse stata al terzo posto non sarebbe stata raggiungibile, in quanto il *match* avrebbe sicuramente scelto una delle due condizioni precedenti.
Questo per quanto riguarda la distinzione tra i tipi di oggetti. Ma essendo Scala un linguaggio anche funzionale, non può certo mancare un metodo di utilizzo che ci permetta di maneggiare con comodità le liste.

```
def matchList(l : Any) : String = l match {
       case List(_, 2, _) => "Lista di 3 elementi di cui il secondo è 2"
       case List(_, "*", _) => "Lista di 3 elementi di cui il secondo è '*'"
       case List(_, _, _) => "Lista di 3 elementi"
       case List(_*) => "Lista non vuota"
       case _ => "Altro"
}
```

Per concludere la sezione passiamo a quella che probabilmente è la funzionalità più interessante del costrutto in questione, ovvero il match di espressioni regolari. Le seguenti righe di codice definiscono delle espressioni regolari in Scala:

```
val ExpDescrizione = """D;([^;]+);(.+)""".r
val ExpPrezzo = """P;([^;]+);([^;]+);(.+)""".r
val ExpGiagenza = """G;([^;]+);([^;]+);([^;]+);(.+)""".r
```

Più specificatamente sono state definite tre espressioni regolari che, in base alla prima lettera della riga individuano una serie di valori attesi per l'input. Queste, come si può vedere, sono delle variabili di tipo *String* sulle quali viene invocato il metodo *r* che le trasforma in oggetti di tipo *scala.util.matching.Regex*. Un'evidenza di questo passaggio si può scorgere eseguendo il seguente comando dalla shell interattiva:

```
val x = "".r
```

All'interno della stringa da trasformare in oggetto di tipo *Regex* c'è l'espressione regolare, in questo caso delimitata da virgolette con relativo escape (questo il motivo per cui ci sono tre virgolette all'inizio di ogni espressione e altrettante alla fine). La forma delle espressioni è quella standard: nel primo caso vengono riconosciute le stringhe che iniziano per *D;* e vengono estratti due valori successivi separati da punto e virgola (di cui il primo non deve contenere alcun punto e virgola). Nel secondo e nel terzo caso vengono estratte le stringhe che iniziano rispettivamente per *P;* e *G;*, con relativi dati attesi.
A seguire un costrutto *match* che verifica la corrispondenza di una stringa rispetto ad un'espressione regolare e ne estrazione i valori indicati. Se ne ipotizza l'utilizzo all'interno di un ciclo che itera le righe di un file csv formattato come atteso.

```
riga match {
       case ExpDescrizione(sku, descrizione) =>
              println("Codice Prodotto: " + sku + " - Descrizione: " + descrizione)
       case ExpPrezzo(sku, prezzo_acquisto, prezzo_vendita) =>
              println("Codice Prodotto: " + sku + " - Prz ACQ: " + prezzo_acquisto + "
                           - Prz VEN: " + prezzo_vendita)
       case ExpGiagenza(sku, disp, imp, pren) =>
              println("Codice Prodotto: " + sku + " - Disponibili: " + disp + " -
```

```
                        Impegnati: " + imp + " - Prenotati: " + pren)
        case entry =>
                println("Registrazione non riconosciuta: " + entry)
}
```

In questo caso non solo vengono riconosciute le righe ma vengono anche estratti i valori attesi. Una procedura reale potrebbe, data la connessione ad un database, effettuare l'update o l'insert dei dati anziché effettuarne un *print*. Il tutto in poche righe concise, chiare e semplici.

## 2.5.4 Ciclo While

I cicli *while* e *do while* si possono usare in Scala ed hanno la stessa sintassi di Java. Riportiamo un esempio di *do while* all'interno di una funzione:

```
def machineEpsilon : Double = {
        var machEps : Double = 1
        do
                machEps /= 2
      while (1 + machEps / 2 != 1)

        machEps
}
```

Istruzioni multiple vanno inserite tra parentesi graffe. Sebbene questo costrutto sia supportato, scrivendo codice in Scala si tende quanto più possibile a preferire il costrutto *for* nelle forme descritte nel prossimo paragrafo, in modo da dare sembianze più funzionali allo stile di programmazione.

## 2.5.5 Ciclo For

In Scala il ciclo *for* tradizionale deriva più dal mondo Pascal/Delphi che da Java o C#. Vediamo un esempio:

```
for(i <- 0 to 10)
        println(i)
```

Il risultato che otteniamo è la stampa dei numeri da zero a dieci estremi compresi. Attraversando un vettore con questo tipo di ciclo servirà quindi prestare attenzione ed impostare l'ultimo valore alla lunghezza dell'array decrementata di una unità, onde evitare una *IndexOutOfBoundException*.
Nell'uso di vettori tuttavia, se possibile, è preferibile un tipo diverso di ciclo. Il prossimo costrutto in Scala è una variante del ciclo *for*, ed in alcuni linguaggi viene denominato *foreach*.

```
var giorni = List("Lun", "Mar", "Mer", "Gio", "Ven", "Sab", "Dom")

for(giorno <- giorni)
        println(giorno)
```

In questo modo, dato un qualsiasi enumerabile, ad ogni iterazione è il ciclo stesso che ci mette a disposizione l'oggetto su cui operare senza doverci curare di indici e bordi.
In alcuni casi, come quello dell'ultimo esempio, in cui viene richiamata una funzione sull'oggetto iterato,

c'è un modo ancora più conciso per effettuare la stessa operazione. Si tratta di una conseguenza del lato funzionale di Scala, ed adattando l'esempio consiste in questa istruzione:

```
giorni.foreach(println)
```

Tornando alla versione Scala del *foreach*, ci sono alcune funzionalità avanzate che vale la pena di conoscere. In primo luogo è possibile filtrare gli elementi contenuti nella lista iterata direttamente all'interno dell'istruzione di *for*. Riprendendo l'esempio precedente, stampiamo solo i giorni della settimana che iniziano per *M*.

```
for(giorno <- giorni
      if giorno.startsWith("M"))
      println(giorno)
```

C'è anche la possibilità di inserire più di una condizione, come mostra il seguente esempio:

```
for(giorno <- giorni
      if !giorno.startsWith("M");
      if giorno.contains("a"))
      println(giorno)
```

Allo stesso modo delle condizioni si possono introdurre anche delle elaborazioni. Nel prossimo esempio verranno esclusi i giorni della settimana che iniziano per *M* e gli altri saranno mostrati in maiuscolo.

```
for(giorno <- giorni
      if !giorno.startsWith("M");
      if giorno.contains("a"))
      println(giorno)
```

Per concludere, il ciclo *for* ci permette anche di creare una lista a partire dalla lista iterata. Nel prossimo esempio, a partire dalla lista dei giorni della settimana, viene creata una sottolista composta solamente dai giorni lavorativi.

```
var lavorativi = for(giorno <- giorni
      if giorno != "Sab";
      if giorno != "Dom")
      yield giorno
```

Anche in questo caso, oltre alle condizioni, all'interno dell'istruzione *for* si può effettuare un'elaborazione del valore iterato. Ma ciò che rende particolarmente interessante questo esempio rispetto ai precedenti è il senso che assume il ciclo *for* e l'azione della parola chiave *yield*. Negli esempi precedenti il *for* veniva usato per fare una stampa di alcuni valori, mentre in questo caso è stato usato come funzione con valore di ritorno. Il ciclo for quindi può essere usato come funzione, e come si può intuire il tipo di ritorno sarà sempre una lista con zero o più elementi. Questa lista viene popolata all'interno del medesimo ciclo con l'uso dell'istruzione *yield*, che aggiunge l'oggetto specificato in seguito (nell'esempio la variabile di tipo String dal nome *giorno*) e la aggiunge alla lista che costituisce il risultato della funzione *for*.

## *2.6 Espressioni lambda*

Scala permette la definizione e l'uso di espressioni lambda, chiamate anche funzioni anonime, come altri linguaggi quali C# e PHP. Nel seguente esempio è definita una semplice funzione anonima che calcola il valore assoluto del prodotto di due numeri interi:

```
var moduloProdotto = (x : Int, y : Int) => {
      var z = x * y
      if (z > 0)
            z
      else
            -z
}
```

Per usare la funzione è sufficiente eseguire la seguente istruzione:

```
var m = moduloProdotto(5, -5)
```

Inoltre si possono definire le variabili in linea senza doverne assegnare il nome. La seguente funzione anonima, ad esempio, restituisce un valore booleano di vero o falso a seconda che l'intero passato come parametro sia maggiore di zero o meno, rispettivamente.

```
var positivo = (_ : Int) > 0
```

Di particolare interesse è la possibilità di richiedere altre funzioni come parametri della funzione anonima. Qui in seguito riportiamo un esempio in cui si sfrutta tale caratteristica per filtrare le liste in base ad una funzione che restituisce un valore booleano.

```
val  applicaFiltro = (condizione : Int => Boolean, lista : List[Int] )  =>  {
    for(x <- lista; if condizione(x))
            yield x
}
```

Questa funzione si può chiamare nel seguente modo:

```
var n = List(1, 5, -3, 20, 0)
var p = applicaFiltro(positivo, n)
```

Il risultato sarà una lista *p* contenente solo i numeri 1, 5, 20 e 0.

Infine uniamo i due esempi precedenti per definire una funzione che ci permetta di ricavare da una lista una sottolista contenente soltanto numeri positivi.

```
val filtraPositivi = applicaFiltro(positivo, _ : List[Int])
```

In questo modo sarà possibile ottenere lo stesso risultato mostrato in precedenza facendo una chiamata più specifica:

```
var n = List(1, 5, -3, 20, 0)
```

```
var p = filtraPositivi(n)
```

## *2.7 Try/catch/finally*

La gestione delle eccezioni in Scala non riporta particolari differenze rispetto al funzionamento su Java o C#. Si possono creare delle eccezioni estendendo le classi base come *Exception*, *ArithmeticException*, *IndexOutOfBoundException* e via dicendo come se fossero normalissime classi. L'estensione delle classi sarà trattata in seguito, nella sezione relativa alla programmazione ad oggetti.

```
val checkCredit = (credit : Double, amount : Double) => {
	if(credit == 0)
		throw new EmptyPlafondException()
	if(credit < amount)
		throw new InsufficientCreditException(credit, amount)
}
```

Nell'esempio riportato vediamo come viene lanciata un'eccezione in base a determinate condizioni. Si può notare come il funzionamento sia più simile a C# che a Java, dal momento che non è necessario aggiungere la clausola *throws* nella firma del metodo.
In seguito un esempio di chiamata al metodo sopra riportato. Nel *try* c'è la porzione di codice protetto, nel *catch* in base alla tipologia dell'eccezione viene effettuata un'operazione diversa e nel *finally* viene stampata una scritta che non varia in base all'esito dell'esecuzione del codice nella prima parte.

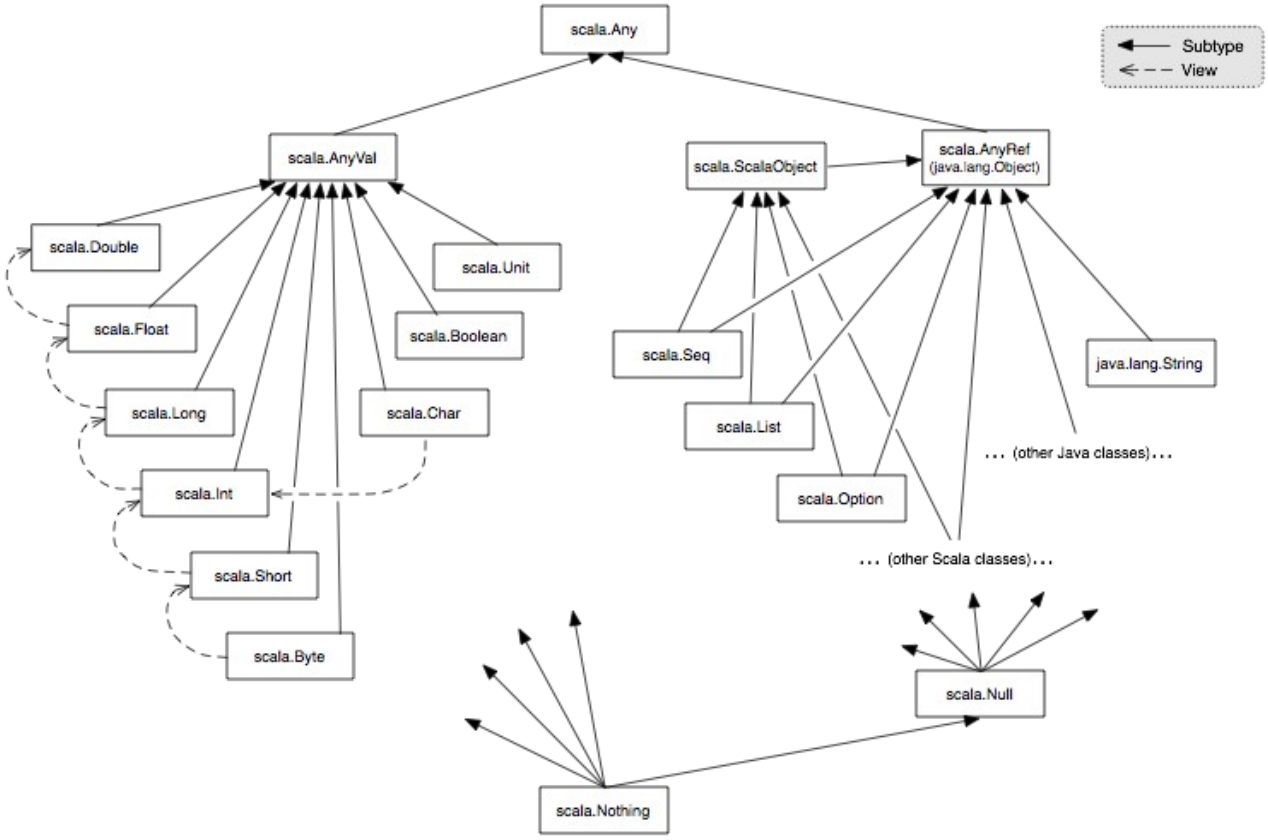
```
try {
	checkCredit(10, 5)
} catch {
	case e : EmptyPlafondException =>
		println("Your plafond is empty")
	case e : InsufficientCreditException =>
		println(e.getMessage())
} finally {
	println("Check completed")
}
```

# 3 Programmazione ad oggetti

Nel capitolo precedente, parlando di sintassi, sono già stati anticipati alcuni aspetti della programmazione ad oggetti. In questo capitolo l'argomento verrà trattato in maniera più specifica ed approfondita non solo dal punto di vista sintattico ma anche dal punto di vista concettuale.

## *3.1 Gerarchia dei tipi*

Come anticipato nell'introduzione, in Scala non ci sono tipi primitivi ma soltanto classi. Diverse classi fornite dalla piattaforma sono tra loro collegate, e conoscere questi meccanismi può aiutare lo sviluppatore che si trova a doverle estendere per adattarne le funzionalità. In seguito è riportata una figura tratta dal sito ufficiale di Scala, in cui viene illustrata la gerarchia dei tipi nel linguaggio.




Partendo dall'alto si può da subito notare come *Any* sia il punto di partenza della gerarchia. Questo significa che ogni classe definita in Scala estende *Any*, qualsiasi sia il livello di profondità. La divisione principale poi è tra classi valore o riferimento, rispettivamente *AnyVal* e *AnyRef*. La prima, *AnyVal*, contiene chiaramente tutti i tipi che in Java o C# sono primitivi. La relazione tratteggiata sta a rappresentare la possibilità di visualizzare un tipo come se fosse un altro, sostanzialmente un cast implicito. Ad esempio il *Byte* può essere visualizzato come *Short*, un *Float* come *Double* e via dicendo, senza perdita di informazione.
Nel lato destro vediamo come il primo livello sia rappresentato dalla classe *AnyRef*, equivalente della classe *Object* su Java. Da qui derivano quindi tutte le classi della piattaforma Java o sviluppate usanto tale linguaggio. Un'altra derivazione rilevante di *AnyRef* è *ScalaObject*, classe da cui derivano *Seq*, *List*, *Option* e altre classi fornite dalla piattaforma Scala.
Nella parte bassa dell'immagine è illustrato il funzionamento dei valori nulli. *Nothing* è il valore nullo per i tipi del lato *AnyVal*, mentre *Null* lo è per i tipi del lato *AnyRef*. Essendo comunque *Null* un'estensione di

*Nothing*, questo secondo valore si può tenere come riferimento anche nel lato *AnyRef*.
Ci sono altre classi in questo frangente che, pur non essendo riportate nell'illustrazione, sono di particolare interesse in fase di sviluppo. *Nil*, ad esempio, viene usato per rappresentare una lista vuota.

```
var lista : List[Int] = Nil
```

Su questo oggetto è possibile chiamare la funzione *length* senza provocare alcun errore. L'istruzione qui sotto riportata infatti produce come output *0*.

```
println(lista.length)
```

Un altro elemento interessante è *None*. Per capirne il senso è necessario introdurre prima il tipo *Option*. Questa classe ha il preciso scopo portare lo sviluppatore ad evitare l'uso dei *null* per qualcosa di più evoluto e meno pericoloso. Si tratta di una classe generica che può assumere due valori, ovvero *None* o *Some*. Il primo caso va a sostituire il *null* e il secondo la valorizzazione dell'oggetto. Con la seguente istruzione creiamo, usando *Option*, una variabile di tipo String con valore non assegnato:

```
var x : Option[String] = None
```

Provando ad ottenerne il valore con la seguente istruzione, otteniamo una NoSuchElementException:

```
println(x.get)
```

Ma possiamo ovviare al problema specificando un valore di default.

```
println(x.getOrElse("Default"))
```

Con la seguente istruzione è possibile assegnare un valore alla variabile di tipo *Option*:

```
x = Some("Contenuto")
```

In questo modo la variabile conterrà la stringa specificata, la quale verrà restituita sia dal metodo *get* che dal metodo *getOrElse* senza controindicazioni.

Infine, per completare la lista dei tipi a valore nullo, è necessario parlare di *Unit*. Questo viene usato come tipo di ritorno nelle funzioni, analogamente a *void* in Java e C#. Il seguente metodo, ad esempio, manda nello standard output la formattazione in valuta di un numero ma non restituisce alcun risultato.

```
def printValuta(d : Double) : Unit = {
       println("EUR " + ("%1,2f".format(d)))
}
```

## *3.2 Costruttori*

Nei linguaggi di programmazione più comuni il costruttore si presenta come un normalissimo metodo, che si contraddistingue soltanto per il nome. In Scala non solo il costruttore è diverso, ma lo è anche il modo con cui si definisce la classe. Prendiamo in considerazione il seguente esempio.

```
class Persona(val cognome : String, val nome : String, val eta : Option[Int]) {
	def denominazione() = cognome + " " + nome
	println("Istanziata la persona " + denominazione)
}
```

La grossa differenza rispetto ad altri linguaggi è che il costruttore primario è il corpo stesso della classe. All'interno si possono definire metodi, come in questo caso *denominazione*, e si possono eseguire istruzioni come il *println* dell'esempio. Le variabili *cognome*, *nome* ed *eta* tra parentesi tonde nella definizione della classe sono i parametri del costruttore, ma non solo. La possibilità di chiamare il metodo *denominazione* sull'oggetto istanziato e riusare tali variabili dimostra come queste diventino anche campi della classe. Per la precisione il compilatore li trasforma in campi *val* con visibilità *public*.
Ricapitolando, con le tre righe di codice dell'ultimo esempio abbiamo ottenuto ciò che in C# avremmo dovuto definire così:

```
public class Persona {
	public readonly string cognome;
	public readonly string nome;
	public readonly int? eta;

	public string denominazione() {
		return cognome + “ “ + nome;
	}

	public Persona(string cognome, string nome, int? eta) {
		this.cognome = cognome;
		this.nome = nome;
		this.eta = eta;

		Console.WriteLine(“Istanziata la persona ” + denominazione());
	}
}
```

È inoltre possibile definire più costruttori definendo con parametri diversi la funzione *this*. Nel seguente esempio viene aggiunto un costruttore che non richiede il parametro *eta* e richiama il costruttore di base passando il valore *None*.

```
class Persona(val cognome : String, val nome : String, val eta : Option[Int]) {
	def this(cognome : String, nome : String) = this(cognome, nome, None)
	def denominazione() = cognome + " " + nome
	println("Istanziata la persona " + denominazione)
}
```

## *3.3 Classi annidate*

La modalità di definizione delle classi annidate in Scala è molto simile a quella di Java. In seguito è riportato un esempio che ne mostra la definizione ed alcune interazioni di esempio.

```
class Regione {
	private var denominazione : String = ""

	class Provincia(val denominazione : String) {
		var comuni : List[String] = Nil
		var abitanti : Int = 0

		def addComune(d : String, a : Int) {
			comuni ::= d
			abitanti += a
		}

		def getDenominazioneRegione : String = {
			Regione.this.denominazione
		}
	}

	var province : List[Provincia] = Nil

	def setDenominazione(d : String) {
		denominazione = d
	}

	def addProvincia(d : String, comuni : List[Tuple2[String, Int]]) {
		var p = new Provincia(d)

		for(c <- comuni)
			p.addComune(c._1, c._2)

		province ::= p
	}
}
```

La classe interna ha visibilità sui membri privati della classe che la contiene. Rispetto all'esempio sopra riportato, ciò si può constatare con una chiamata al metodo *getDenominazioneRegione* nella classe *Provincia*, come riportato in seguito.

```
var regione = new Regione
regione.setDenominazione("Veneto")
regione.addProvincia("Padova", List(("Padova", 204809), ("Albignasego", 23464)))
println(regione.province(0).getDenominazioneRegione)
```

Per richiamare la classe interna, infine, a differenza di quanto accade su Java è necessario usare il simbolo *cancelletto*. Per creare una lista di oggetti di tipo *Provincia*, ad esempio, l'istruzione da eseguire sarà la seguente.

```
var province : List[Regione#Provincia] = Nil
```

## *3.4 Ambiti di visibilità*

Come in Java e C# anche in Scala si possono definire degli ambiti dai quali si può o meno accedere ad un campo o ad una classe. Nella tabella sottostante sono riportati gli ambiti con relative parole chiave e descrizioni.

| **Parola chiave** | **Nome** | **Descrizione** |
|---|---|---|
| | Pubblico | Di default l'ambito di visibilità dei campi e delle classi definite è pubblico (l'analogo di *public* su Java e C#). Questi elementi, essendo pubblici, sono visibili ovunque senza alcuna limitazione. |
| protected | Protetto | I campi protetti sono visibili solo nel tipo che li definisce, nelle sue classi derivate e annidate, con restrizione allo stesso *package* e *subpackage*. |
| private | Privato | I campi privati sono visibili solo dall'interno dello stesso tipo, con restrizione allo stesso *package*. |
| protected[scope] | Protetto Ristretto | I campi protetti ristretti sono visibili solo all'interno dello *scope* specificato. Lo scope può essere un *package*, un tipo o *this*. |
| private[scope] | Privato Ristretto | La visibilità dei campi privati ristretti è analoga a quella dei campi protetti ristretti, con la differenza che sono esclusi i casi di ereditarietà. |

Nel seguente esempio vediamo alcune applicazioni delle logiche sopra descritte.

```
class Visibilita {
	class Nested1 {
		private var Valore : Int = 0
	}
	class Nested2 {
		private[Visibilita] var Valore : Int = 0
	}
	class Nested3 {
		protected var Valore : Int = 0
	}
	class Nested3Child extends Nested3 {
		def getValore = Valore
	}
	class Nested4 {
		protected[Visibilita] var Valore : Int = 0
	}
}
```

Provando ad accedere dal costruttore della classe *Visibilita* alla proprietà *Valore* della classe *Nested1* si ottiene un errore, in quanto il campo è privato e quindi accessibile solo dall'interno della stessa classe. Diversa è la situazione della classe *Nested2*. In questo caso si riesce ad accedere alla proprietà *Valore* in quanto lo *scope* definito per tale campo è *Visibilita*. Rimane quindi un campo privato, ma accessibile solamente dall'interno di quest'ultima classe.
Anche *Valore* all'interno della classe *Nested3* sarà inaccessibile. Questo appunto perché il campo è protetto senza uno *scope* definito. Da *Nested3Child* infatti si riesce ad accedere al campo, e si riesce ad esportarlo grazie ad un metodo pubblico.

Infine, per quanto riguarda *Nested4*, se la classe venisse estesa all'esterno di *Visibilita*, il campo *Valore* non sarebbe accessibile in quanto lo *scope* è specificato per tale classe.

## *3.5 Tipi parametrici*

Tipi parametrici è il termine che si usa per definire in Scala ciò che su Java e C# va sotto il nome di tipi generici. Sono già stati usati precedentemente in alcuni esempi, e la differenza principale dal punto di vista della sintassi è l'uso delle parentesi quadre anziché delle parentesi angolari. La seguente istruzione è già stata usata, e definisce la variabile *o* come istanza parametrica di della classe *Option* rispetto al tipo *String*.

```
var o1 = Option[String]
```

Per scendere nel dettaglio, Scala permette un controllo più dettagliato del tipo parametrico. Con la seguente istruzione, ad esempio, si definisce una classe il cui tipo parametrico dev'essere una sottoclasse di *A*.

```
class Sottotipo[-A] { }
```

Allo stesso modo si può richiedere il tipo sia una classe superiore di A.

```
class Supertipo[+A] { }
```

La sintassi per la definizione di un metodo parametrico è riportata nell'esempio seguente.

```
def creaLista[A](e : A*) : List[A] = {
	e.toList
}
```

Questo metodo richiede come parametro zero o più elementi del tipo specificato, e restituisce una lista costruita con tali elementi. La seguente chiamata, ad esempio, restituisce una lista di quattro interi nell'ordine in cui sono riportati.

```
creaLista[Int](10, 20, 40, 70)
```

Combinando la definizioni di metodi parametrici all'interno di classi parametriche, si possono definire dei termini relativi. Ad esempio, nel seguente caso, andiamo a definire una classe parametrica che accetta tipi derivati da *A*, con un metodo parametrico rispetto a *B* derivato da *A*.

```
class Sottotipo[-A] {
	def Inf[B <: A](x : B) = {
	}
}
```

Simmetricamente, è possibile definire una classe parametrica rispetto ad un super tipo di *A*, contenente un metodo parametrico rispetto ad un tipo *B* superiore ad *A*, come mostrato nel seguente esempio.

```
class Supertipo[+A] {
	def Sup[B >: A](x : B) = {
	}
}
```

## *3.6 Estensione di classi e tratti*

Gli strumenti che Scala fornisce allo sviluppatore per ridefinire classi e tratti sono molto simili a quelli messi a disposizione da Java o C#. Una classe può estendere un'altra classe, tramite la parola chiave *extends*, e molteplici tratti con la parola chiave *with*.

Tramite la parola chiave *abstract* si possono definire campi, metodi e tipi astratti, la cui implementazione viene demandata alle classi derivate. Al contrario si può impedirne la ridefinizione nelle classi derivate tramite la parola chiave *final*.

Un punto di particolare interesse è l'obbligatorietà delle parola chiave *override* nella ridefinizione di campi o metodi. L'analogo in Java è l'attributo opzionale *@Override*, che permette al compilatore di segnalare eventuali errori nella ridefinizione di metodi che non esistono nella classe padre. Essendo la notazione obbligatoria in Scala, il compilatore sarà in grado di segnalare la situazione simmetrica, ovvero quando viene ridefinito un elemento della classe padre per errore.

In seguito un esempio di definizione di una classe astratta *Punto* con implementazione del metodo *d*, che restituisce la distanza del punto dall'origine, in base al numero di dimensioni del piano in cui giace.

```
abstract class Punto {
	def d : Double
}

class Punto1D extends Punto {
	var x : Double = 0

	final override def d : Double = x
}

class Punto2D extends Punto {
	var x : Double = 0
	var y : Double = 0

	final override def d : Double = Math.sqrt(x * x + y * y)
}

class Punto3D extends Punto {
	var x : Double = 0
	var y : Double = 0
	var z : Double = 0

	final override def d : Double = Math.sqrt(x * x + y * y + z * z)
}
```

La definizione di una classe non astratta che non implementa il metodo *d* restituirà un errore, come nel seguente esempio.

```
class Punto0D extends Punto {
}
```

Allo stesso modo, un tentativo di estendere una delle tre classi non astratte dell'esempio e sovrascriverne il metodo *d* restituirà un errore. Nel seguente esempio si sta provando a ridefinire il metodo in modo da restituire la distanza dal punto *(1, 1)* anziché dall'origine.

```
class Punto2DOffset extends Punto2D {
	override def d : Double = Math.sqrt((x - 1) * (x - 1) + (y - 1) * (y - 1))
}
```

La stessa situazione si può riprodurre con l'utilizzo dei tratti. Nel seguente esempio prendiamo l'implementazione consiste in una classe astratta *Punto* e nel tratto *Distanza*. Il metodo astratto è definito nel tratto, e la modalità di sovrascrittura non varia rispetto agli esempi precedenti.

```
abstract class Punto {
}

trait Distanza {
	def d : Double
}

class Punto1D extends Punto with Distanza {
	var x : Double = 0

	final override def d : Double = x
}

class Punto2D extends Punto with Distanza {
	var x : Double = 0
	var y : Double = 0

	final override def d : Double = Math.sqrt(x * x + y * y)
}

class Punto3D extends Punto with Distanza {
	var x : Double = 0
	var y : Double = 0
	var z : Double = 0

	final override def d : Double = Math.sqrt(x * x + y * y + z * z)
}
```

Analogamente a quanto accade con le classi, le due classi definite nel seguente esempio sono errate e producono un errore in fase di compilazione.

```
class Punto0D extends Punto with Distanza {
}
class Punto2DOffset extends Punto2D {
	override def d : Double = Math.sqrt((x - 1) * (x - 1) + (y - 1) * (y - 1))
}
```

Nel primo caso per la mancata sovrascrittura del metodo *d* e nel secondo caso per la sovrascrittura di un metodo definito con parola chiave *final* nella classe padre.

## *3.7 Proprietà*

Tipicamente in Java una proprietà si implementa tramite la definizione di un campo con accessibilità *private* e i metodi di accesso *get* ed eventualmente *set* a seconda della tipologia della stessa (sola lettura o meno). In Scala tutto questo viene realizzato dal compilatore. Tale comportamento lo si può vedere generando un file e poi analizzandolo con il comando *javap*.

Per iniziare va creato un file, per comodità, con lo stesso nome della classe. In questo caso *Proprieta.scala*, con il seguente contenuto.

```
class Proprieta {
	var a : Int = 0
	val b : Int = 0
}
```

Il codice è molto semplice, sono state definite due proprietà di tipo *Int*, *a* in lettura e scrittura e *b* in sola lettura.

Eseguendo il comando *scalac Proprieta.scala* otterremo il file *Proprieta.class*, contenente il bytecode Java generato dal compilatore di Scala. A questo punto si può eseguire il comando *javap Proprieta*, in modo da visualizzare il risultato della compilazione. L'output del comando sarà il seguente.

```
public class Proprieta extends java.lang.Object {
	public int a();
	public void a_$eq(int);
	public int b();
	public Proprieta();
}
```

Oltre a *Proprieta()*, che è il costruttore della classe, si può notare come siano stati generati i metodi di accesso alle due proprietà. *a()* e *a_$eq(int)* sono rispettivamente i metodi di *get* e *set* per la proprietà *a*, mentre *b()* è il metodo di lettura per la variabile *b*. In quest'ultimo caso non è stato generato il metodo di scrittura in quanto b è stata definita con la parola chiave *val* e non *var*.

Come ulteriore prova, si può utilizzare il seguente codice.

```
var p = new Proprieta
println(p.a)
p.a_$eq(1)
println(p.a)
```

Un'esecuzione anche in modalità non compilata mostrerà come il primo *println* restituisca come output il valore 0, ovvero quello di default, ed il secondo *println* il valore 1 assegnato nella riga precedente tramite il metodo generato in automatico.

## *3.8 Metodi apply e unapply*

Si è già parlato di cosa sono e come si usano gli oggetti associati nella sezione riguardante la definizione delle classi, ovvero la 2.2. I due metodi approfonditi in questo capitolo, *apply* e *unapply*, sono di particolare interesse proprio in questo ambito.

Il metodo *apply* permette sostanzialmente di creare l'istanza di un oggetto senza passare per il costruttore. Al lato pratico, ogni volta che il compilatore trova il nome della classe non preceduto dalla parola chiave *new*, ne invoca il metodo *apply* con i parametri specificati.

Nel seguente esempio sono definiti costruttore e metodo *apply*, con due uscite diverse tramite *println* per distinguere le due modalità di istanza chiamate a runtime. Innanzitutto, come si può vedere, *class* e *object* sono definiti contestualmente e con lo stesso nome in modo che il compilatore li associ correttamente.

```
class Applica {
	println("Chiamato il costruttore")
}
object Applica {
	def apply() {
		println("Chiamato il metodo apply")
	}
}
```

A questo punto i seguenti comandi chiameranno rispettivamente costruttore e metodo *apply*.

```
var a = new Applica()
var b = Applica()
```

Come atteso, si potrà vedere nel primo caso l'output “Chiamato il costruttore” e nel secondo caso “Chiamato il metodo apply”. Nell'esempio specifico costruttore e metodo *apply* erano entrambi privi di parametri, ma non c'è alcuna limitazione che impedisca ai due metodi di avere numero e tipo di parametri diversi.

Un'altra caratteristica particolarmente interessante di questa funzionalità è la possibilità di restituire istanze di classi derivate. Il seguente esempio mostra un'applicazione di questa modalità di utilizzo.

```
class P {
	println("Istanziato P")
}

object P {
	def apply(i : Int) {
		if(i == 1)
			new F1
		else if(i == 2)
			new F2
```

```
            else
                  new P
      }
}

class F1 extends P {
      println("Istanziato F1")
}

class F2 extends P {
      println("Istanziato F2")
}
```

I comandi a seguire attivano i tre diversi percorsi creati nel metodo *apply*.

```
var c = P(0)
var d = P(1)
var e = P(2)
```

Nel primo caso si ottiene una variabile di tipo P, e in output si visualizza la scritta “Istanziato P”. Nel secondo caso si ottiene un oggetto di tipo F1, che estende P, e in output si avranno le due righe “Istanziato P” e “Istanziato F1” in quanto il costruttore della classe derivata invoca di default il costruttore della classe padre. La terza casistica è analoga alla seconda con un oggetto di tipo F2.

A svolgere la funzione opposta al metodo *apply*, come si può dedurre dal nome, c'è il metodo *unapply*. In seguito una definizione ed implementazione di esempio del metodo in questione.

```
class Paese(val denominazione : String) {
}
object Paese {
      def unapply(r : Paese) = Some(r.denominazione)
}
```

Come si può vedere *unapply* necessita di un oggetto di riferimento che dev'essere passato come parametro, dal momento che si tratta di un metodo statico. Si può inoltre intuire che l'utilizzo del metodo sia sostanzialmente diverso dall'utilizzo del metodo *apply*. Usando il metodo come semplice elemento statico la chiamata sarebbe la seguente.

```
var r = new Paese("Italia")
println(Paese.unapply(r))
```

Dove il metodo *unapply* risulta più comodo è invece nel pattern matching, e nel seguente esempio si noterà come la chiamata semplifichi molto l'utilizzo dell'oggetto all'interno del costrutto *match*.

```
r match {
      case Paese(d) =>
            if(d == "Italia")
                  println("Italia")
```

```
            else
                  println("Estero")
}
```

Si noti che nell'oggetto associato Paese non era definito il metodo *apply*, quindi l'unica chiamata fatta all'interno del *match* è l'*unapply*. Sostanzialmente l'istruzione *case Paese(d)* non fa altro che verificare che l'oggetto *r* sia di tipo paese, e poi estrae su d il risultato della chiamata *Paese.unapply(r)*. In questo modo è dunque possibile mettere allo stesso livello più *case* che richiedano oggetti di tipo diverso.

Si può inoltre fare in modo che i metodi *apply* e *unapply* accettino un numero definito di argomenti dello stesso tipo, ovvero una lista. Come si vedrà, il metodo *apply* mantiene lo stesso nome, mentre il metodo *unapply* per questo tipo di situazioni va chiamato convenzionalmente *unapplySeq*. A seguire un esempio della definizione dei metodi *apply* e *unapplySeq* con liste come parametri.

```
class Sequenza[A] (val lista : List[A]) {
}
object Sequenza {
      def apply[A](x : A*) : Sequenza[A] = new Sequenza(x.toList)
      def unapplySeq[A](x : Sequenza[A]) : Some[List[A]] = Some(x.lista)
}
```

La *Sequenza* è generica rispetto ad un tipo *A*, e non fa altro che incapsulare un oggetto di tipo *List[A]*. La parte statica contiene i due metodi *apply* e *unapplySeq*, entrambi parametrici rispetto al tipo *A*. Come si può vedere il metodo *apply* richiede come parametro un vettore di oggetti di tipo *A*, e restituisce un oggetto *Sequenza[A]* creato tramite il costruttore invocato sulla trasformazione in lista del vettore. Il metodo *unapplySeq*, analogamente alla versione a singolo parametro, richiede come parametro un oggetto di tipo *Sequenza[A]*. La differenza sta nel tipo di ritorno, il quale è una lista incapsulata in un Some anziché un elemento.

Il metodo *apply* di *Sequenza* si può testare con facilità tramite il seguente comando. Come si può notare l'utilizzo è analogo a quello di *List*.

```
var s = Sequenza(1, 5, 10)
```

Sempre prendendo spunto dal pattern matching di una lista, l'esempio sottostante mostra l'utilizzo del metodo *unapplySeq* di *Sequenza*.

```
s match {
      case Sequenza(1, _, _) => println("Sequenza di tre numeri di cui il primo e' 1")
      case _ => println("Squenza non riconosciuta")
}
```

## *3.9 Uguaglianza tra oggetti*

Un confronto affidabile tra istanze di oggetti è un argomento articolato su cui ci sono diverse trattazioni teoriche e pratiche. In questo paragrafo si vedranno gli strumenti che Scala mette a disposizione a tale scopo, e quali sono i relativi ambiti di utilizzo.

I metodi *equals* e *hashCode* ad esempio, già noti in Java, sono creati automaticamente per le classi *case* (di cui si è parlato nel paragrafo 2.4.2) ma vanno sovrascritti sugli altri tipi di classi che si vanno a definire o estendere. Il metodo *equals*, nello specifico, per convenzione fa un controllo sui valori dei due

oggetti confrontati, quindi si comporta esattamente come in Java.

Una differenza sostanziale la si riscontra invece sugli operatori == e *!=*. Questi, infatti, contrariamente a quanto accade su Java e C# fanno un controllo sul valore e non sul riferimento all'oggetto. Di fatto l'operatore == è un metodo *final* definito nella classe *Any* che richiama il metodo *equals*. Analogamente, *!=* è definito nella stessa classe e restituisce la negazione del risultato del metodo *equals*.

Rimane quindi da affrontare il confronto tra riferimenti. Questo si realizza tramite i metodi *eq* e *ne*. Il primo restituisce *true* nel caso i due riferimenti confrontati corrispondano e *false* altrimenti, mentre il secondo è la negazione del primo.

Per quanto riguarda gli array la situazione è diversa. Gli operatori == e *!=* non implementano un confronto per valore, bensì per riferimento. Lo si può notare prendendo in considerazione le seguenti dichiarazioni.

```
var a = Array(1, 5, 7)
var b = Array(1, 5, 7)
var c = a
```

A queste si possono applicare i controlli elencati in seguito.

```
println(a == b)
println(a == c)
println(a != b)
println(a != c)
```

Il primo confronto darà esito negativo, in quanto il confronto è tra due vettori con gli stessi elementi ma con riferimenti diversi. Il secondo confronto avrà invece esito positivo, in quanto *c* è una copia del riferimento ad *a*. I due confronti successivi avranno esito invertito, come atteso.

Per effettuare un controllo tra i valori di due array è necessario utilizzare il metodo *sameElements*. Il seguente esempio darà esito positivo sulle definizioni riportate in precedenza.

```
println(a.sameElements(b))
```

Risulta dunque esserci un'incoerenza nel significato degli operatori == e *!=*, in quanto con gli oggetti implementano un controllo per valore mentre con le liste implementano un controllo per riferimento. Questa differenza è stata voluta dai progettisti del linguaggio per obbligare lo sviluppatore ad usare un test esplicito nel confronto tra strutture dati mutabili come gli array. Se si ripetono i test appena fatti usando dei tipi immutabili come le liste, infatti, si noteranno dei risultati che denotano un confronto per valore.

# 4 Programmazione funzionale

Fino a qui si sono già visti, più o meno evidentemente, diversi aspetti del lato funzionale di Scala. In questo capitolo sarà approfondito l'argomento partendo dalle strutture dati messe a disposizione dalla piattaforma ed arrivando alle applicazioni più rilevanti.
Tipicamente i linguaggi funzionali impongono dei paletti come l'immutabilità delle variabili e l'assenza di effetti collaterali nelle funzioni. Avendo già affrontato il lato ad oggetti, sappiamo che come linguaggio misto Scala non applica queste restrizioni. Tuttavia è preferibile seguire determinate linee guida che saranno indicate nel corso del capitolo, in particolar modo per favorire ottimizzazione e stabilità nella programmazione concorrente.

## *4.1 Strutture dati*

In questa sezione saranno approfondite le principali strutture dati messe a disposizione da Scala e di particolare rilevanza nella programmazione funzionale.

### 4.1.1 Liste

Parlando di strutture dati in un linguaggio funzionale non si può che partire dalle liste. Si è già visto il tipo *List* nei precedenti capitoli, ed è anche già stata spiegata la sua immutabilità. In seguito è ripresa la definizione di una lista.

```
val l1 = List("B", "C", "D")
```

La convenzione prevede l'aggiunta degli elementi in testa. Il risultato dell'operazione, a causa dell'immutabilità del tipo *List*, è una nuova lista.

```
val l2 = "A" :: l1
```

A questo punto il contenuto di l2 saranno, nell'ordine, le lettere A, B, C e D. L'operatore *::*, che esegue la concatenazione, come come da esempio può essere usato tra un elemento e una lista. È inoltre utile sapere che lega gli operandi alla propria destra.

A seguire una definizione diversa della lista *l2*.

```
val l3 = "A" :: "B" :: "C" :: List("D")
```

La sequenza di esecuzione delle concatenazioni risulta quindi essere la stessa della definizione sottostante.

```
val l4 = "A" :: ("B" :: ("C" :: List("D")))
```

Come si è detto all'inizio del paragrafo, ad ogni concatenazione viene creata una nuova lista per preservare l'immutabilità dell'oggetto. Ad una prima lettura questa informazione implica delle ovvie considerazioni in merito alle prestazioni. Intuitivamente, infatti, si è portati a pensare che la creazione di una nuova lista ne richieda la trascrizione degli elementi, e che quindi l'operazione abbia una complessità computazionale pari a O(N), dove N è il numero di elementi nella lista. In realtà la complessità computazionale dell'operazione è O(1), e per capirne il motivo è necessario analizzare la definizione dell'operatore *::*.

```
final case class ::[B](private var hd: B, private[list] var tl: List[B])
    extends List[B] {
  override def isEmpty: Boolean = false
  def head : B = hd
  def tail : List[B] = tl
}
```

La prima cosa che si nota è come in realtà l'operatore sia una *case class* che estende il tipo *List*. È però il modo in cui la classe è strutturata che mostra come sia possibile rendere l'aggiunta in testa di un elemento un'operazione di complessità computazionale O(1). Le due proprietà rilevanti della classe, infatti, sono *head* e *tail*. La prima proprietà contiene la "testa" della lista, ovvero il primo elemento, mentre la seconda proprietà contiene la "coda", ovvero gli elementi successivi. Nessuna operazione è effettuata nella lista a parte l'assegnazione alla proprietà *tail*, proprio questo è il motivo dell'indipendenza dalla relativa lunghezza. Chi è abituato alla programmazione ad oggetti sarà portato a chiedersi immediatamente cosa succede nel caso sia modificata la lista di provenienza, ma di fatto questa casistica non si può verificare grazie appunto all'immutabilità del tipo *List*.

Ricapitolando, le classi *l3* ed *l4* definite in precedenza, avranno la seguente struttura.

```
L3 → (
       head: "A"
       tail → (
              head: "B"
              tail → (
                     head: "C"
                     tail: List("D)
              )
       )
)
```

Per concludere la parte relativa alle liste, può essere utile sapere che la classe *List* mette a disposizione anche l'operatore + per l'aggiunta di un elemento in coda, ma si tratta di un metodo deprecato e con complessità O(N).

### 4.1.2 Vettori

Come si può intuire dalla struttura del tipo *List*, le liste sono molto performanti quando si tratta di eseguire operazioni di aggiunta in testa o separazione *head*/*tail*, mentre risultano molto meno efficienti gli altri tipi di operazione. La struttura dati *Vector*, di contro, ha ottime prestazioni nella ricerca degli elementi e buone nell'inserimento i testa o in coda. Nello specifico la lettura degli elementi ha come complessità di calcolo $O(\log_{32}(N))$, in quanto viene usata una struttura ad albero con fattore di ramificazione due.

In seguito un esempio della definizione di un vettore.

```
val v = Vector(1, 2)
```

L'aggiunta in testa di elementi si effettua tramite l'operatore *+:*, come mostrato nel seguente esempio.

```
val v2 = v :+ 3 :+ 4
```

L'aggiunta in coda, invece, si effettua tramite l'operatore *:+*, come mostrato in seguito.

```
val v3 = 0 +: v2
```

La creazione di un nuovo vettore con un elemento modificato si effettua tramite il metodo *updated*.

```
val v4 = v3 updated (4, 0)
```

Con quest'ultima istruzione si genera un vettore contenente nell'ordine i valori 0, 1, 2, 3 e nuovamente 0. Il metodo *updated* richiede quindi come parametri l'indice a base zero della posizione da modificare e il nuovo valore da inserire.

In generale, escluse le casistiche in cui si fa un uso intenso delle operazioni *head*/*tail*, la struttura dati *Vector* è da preferire in quanto più veloce e flessibile rispetto alle altre strutture dati messe a disposizione dalla piattaforma Scala.

### 4.1.3 Stream

*Stream* è una struttura dati persistente a valutazione ritardata. Ciò significa che, una volta inseriti i dati nella collezione, questi saranno valutati solo al primo accesso ed il valore sarà mantenuto in memoria. Nel seguente esempio viene definita un oggetto di tipo *Stream* composto da due elementi.

```
val s = {
              println("Valutazione del primo elemento")
              1
       } #:: {
              println("Valutazione del secondo elemento")
              2
       } #:: Stream.empty
```

Dall'esempio si nota innanzitutto l'operatore usato per la concatenazione, ovvero *#::*. Può passare inosservata invece l'importanza di *Stream.empty*, che in realtà è fondamentale per il funzionamento dell'operatore. Il funzionamento della valutazione ritardata si può notare dalla seguente sequenza di operazioni.

```
println(s(0))
println(s(1))
println(s(0))
println(s)
```

La prima operazione stamperà a video il messaggio relativo alla valutazione del primo elemento ed in seguito il valore contenuto. Lo stesso succede anche nella seconda istruzione. La terza istruzione mostrerà soltanto il contenuto e nessun messaggio, in quanto la valutazione è già stata fatta alla prima istruzione. La quarta istruzione, infine, mostrerà il contenuto completo della struttura, ovvero *Stream(1, 2, ?)*. Il punto di domanda finale rappresenta lo stream vuoto di chiusura, e lascia anche intravedere come questa struttura dati sia gestita in modalità *head*/*tail* analogamente alle liste.

Un'altra caratteristica chiave di questa struttura dati è che può contenere un numero infinito di valori senza causare overflow di memoria. Si tratta sostanzialmente di una conseguenza della valutazione ritardata, e l'uso di *Stream* per la definizione della serie di Fibonacci costituisce un ottimo esempio di

questa caratteristica.

```
val fibonacci = {
      def f(a : BigInt, b : BigInt) : Stream[BigInt] = a #:: f(b, a + b)
      f(0, 1)
}
```

Di fatto l'oggetto *fibonacci* non contiene una quantità infinita di valori della serie, ma è in grado di calcolarli su richiesta. Si può calcolare un valore qualsiasi come se lo si stesse leggendo da una lista.

```
println(fibonacci(10))
```

Oppure si può estrarre un sottoinsieme degli elementi della serie, di nuovo come si trattasse di una lista.

```
println(fibonacci drop 100 take 5 toList)
```

In quest'ultimo caso, ad esempio, sono stati calcolati tutti gli elementi fino al numero 105, dopo di che è stata creata e visualizzata una sottolista con gli elementi dal numero 100 al numero 104.

### 4.1.4 Collezioni parallele

Un altro tipo particolarmente interessante di collezioni messe a disposizione da Scala sono le collezioni parallele. La caratteristica che le contraddistingue è la possibilità di effettuare le operazioni in parallelo, sfruttando quindi, se presenti, le capacità di *multi-threading* della macchina fisica in cui il codice viene eseguito. Queste collezioni fanno uso di iteratori *Splitable*, ovvero degli iteratori che possono essere efficientemente scomposti e operare su una porzione della lista iniziale. Le azioni eseguite in questo tipo di collezioni vengono pianificate ed eseguite dall'oggetto *ForkJoinPool*, che come suggerisce il nome si occupa di dividere i dati e riunire i risultati delle operazioni effettuate dagli iteratori *Splitable*.

La creazione di una collezione parallela non differisce dalla creazione di una normale lista, come mostra il seguente esempio. Si noti che è necessario effettuare l'importazione.

```
import scala.collection.parallel.immutable.ParVector
val p1 = ParVector(1, 2, 3, 4, 5, 6, 7, 8)
```

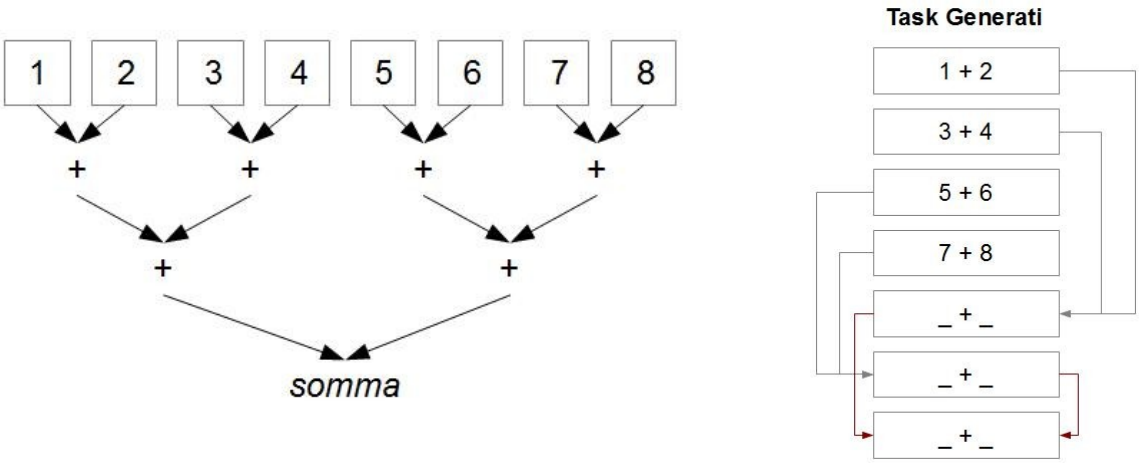
Lo stesso risultato si può eseguire a partire da una lista già definita, senza la necessità di effettuare l'importazione del tratto.

```
val p2 = List(1, 2, 3, 4, 5, 6, 7, 8).par
```

A questo punto si può operare sulla lista. Nel seguente esempio si invoca il metodo che effettua la somma di tutti gli elementi, restituendo in questo caso il valore 10.

```
val somma = p1 sum
```

Nella figura sottostante è illustrata la scomposizione della lista e la sequanza dei task generati.




Nel considerare l'uso di collezioni parallele i fattori da valutare sono principalmente due, ovvero l'efficienza della trasformazione della lista classica in lista parallela ed il grado di parallelizzazione che può raggiungere l'azione da eseguire sui dati. Il secondo punto va considerato in base alla casistica, mentre per quanto riguarda il primo punto ci sono dei fattori che dipendono dalla piattaforma. Trasformare un *Vector* in *ParVector*, ad esempio, è un'operazione con complessità computazionale O(1). Trasformare un *List* in *ParVector*, di contro, ha una complessità computazionale O(N), in quanto la lista viene prima trasformata in *Vector* e poi in *ParVector*.

### 4.1.5 Mappe

Altre strutture dati di particolare interesse nella programmazione funzionale sono le mappe. Queste sono usate sostanzialmente per memorizzare coppie di chiavi e valori, e su Scala si possono definire tramite una sintassi molto intuitiva simile a quella utilizzabile in PHP.

```
val province = Map(
      “Valle d'Aosta” -> List(“Aosta”),
      “Trentino Alto Adige” -> List(“Bolzano”, “Trento”),
      “Friuli-Venezia Giulia” -> List(“Gorizia”, “Pordenone”, “Udine”, “Trieste”)
)
```

La definizione del tratto *Map*, all'interno della piattaforma, è divisa in tre parti vista la possibilità di utilizzare la struttura dati sia come mutabile che come immutabile. Quindi nel tratto *scala.collection.Map* sono definiti soltanto i metodi per la lettura del contenuto, i tratti *scala.collection.mutable.Map* e *scala.collection.immutable.Map* estendono il primo e ne definiscono rispettivamente i metodi mutabili e immutabili per l'aggiunta e la rimozione degli elementi.

## *4.2 Operazioni comuni*

In questa sezione saranno approfondite le principali operazioni funzionali che si possono utilizzare sulle strutture dati trattate.

### 4.2.1 Attraversamento

L'operazione più comune è l'attraversamento in sola lettura, ed è supportato da tutte le strutture descritte in precedenza. Tipicamente l'operazione si effettua tramite *foreach*, costrutto già visto nella

sezione 2.4.5.

Presa in considerazione la mappa *province* definita nella sezione 4.1.2, il seguente attraversamento stampa in console il numero di province relative ad ogni regione inserita.

```
province foreach { r =>
	println("Regione " + r._1 + ": " + r._2.size)
}
```

Dall'esempio si può notare come la funzione *foreach* richieda come parametro una funzione e la richiami all'iterazione di ogni elemento. Si nota anche come utilizzare l'elemento iterato di una mappa, che analogamente a *Tuple* (vista nella sezione 3.3) presenta la chiave nella proprietà *_1* ed il valore nella proprietà *_2*.

Con la mappa *province* possiamo inoltre fare un attraversamento a due livello, come mostrato nel seguente esempio.

```
province foreach { r =>
	println(r._1)
	r._2 foreach { p =>
		println("\t" + p)
	}
}
```

In questo modo per ogni regione saranno elencate le province associate ad un ulteriore livello di indentazione.

### 4.2.1 Mappatura

La mappatura è un'operazione che restituisce una collezione delle stesse dimensioni dell'originale, ma ne elabora i contenuti. Nell'esempio seguente la mappa *province*, di tipo *Map[String, List[String]]* sarà trasformata in una collezione contenente il numero di province per ogni regione, quindi di tipo *Map[String, Int]*.

```
val n_province = province map { r => (r._1, r._2.size) }
```

Facendo una stampa della mappa tramite *println*, si noterà come il contenuto corrisponda a quello di una lista definita come da seguente esempio.

```
val n_province_def = Map(
	“Valle d'Aosta” -> 1,
	“Trentino Alto Adige” -> 2,
	“Friuli-Venezia Giulia” -> 4
)
```

In alcuni testi viene descritta una problematica relativa al fatto che il risultato dell'operazione *map* fosse di tipo *ArrayBuffer*. Tale problema non si verifica con la versione 2.10 di Scala, utilizzata per compilare e testare il codice qui riportato.

### 4.2.3 Filtraggio

Il filtraggio è un'operazione che prevede la creazione di una lista dello stesso tipo di quella di origine ma con un sottoinsieme di elementi. In seguito è riportato un esempio di filtraggio, sempre sulla mappa *province*.

```
val province_2 = province filter { r => r._2.size > 3 }
```

Su *province_2* si troveranno solo le regioni con un numero di province maggiore di 3. Nel caso specifico, la mappa risultante coinciderà con quella definita dal seguente codice.

```
val province_2_def = Map(
      “Friuli-Venezia Giulia” -> List(“Gorizia”, “Pordenone”, “Udine”, “Trieste”)
)
```

Come anticipato non c'è stata alcuna modifica ai tipi degli elementi, ma soltanto la creazione di un sottoinsieme. Chiaramente il risultato dell'operazione di filtraggio può coincidere con la collezione originale, se tutti gli elementi rispettano la condizione specificata.

Oltre a *filter*, che data una collezione restituisce tutti gli elementi che la rispettano, il tratto *Iterable* mette a disposizione altri metodi per il filtraggio degli elementi. Questi sono:

- **drop**: dato un intero *n*, elimina i primi *n* elementi della collezione
- **dropWhile**: data una condizione, restituisce una collezione contenente gli elementi della collezione iniziale ad eccezione di quelli precedenti al primo che non la rispetta
- **exists**: data una condizione, restituisce *true* se almeno un elemento della collezione la rispetta, altrimenti restituisce *false*
- **find**: data una condizione, restituisce il primo elemento della collezione che la rispetta (se presente)
- **findIndexOf**: data una condizione, restituisce l'indice del primo elemento della collezione che la rispetta se presente, altrimenti restituisce *-1*
- **forall**: data una condizione, restituisce *true* se tutti gli elementi della collezione la rispettano, altrimenti restituisce *false*
- **indexOf**: dato un elemento, se presente nella collezione ne restituisce l'indice, altrimenti restituisce *-1*
- **partition**: data una condizione, restituisce due collezioni, la prima contenente tutti gli elementi che la rispettano e la seconda contenente tutti gli elementi che non la rispettano
- **sameElements**: data una collezione, restituisce *true* se il contenuto equivale a quello della collezione iniziale altrimenti restituisce *false*
- **take**: dato un intero *n*, restituisce i primi *n* elementi della collezione
- **takeWhile**: data una condizione, restituisce una collezione contenente tutti gli elementi precedenti al primo che non la rispetta

Inoltre, i tipi come *Map* e *Set* che estendono *Iterable*, contengono ulteriori metodi più specifici per il filtraggio degli elementi.

### 4.2.4 Ripiegamento e riduzione

Ripiegamento e riduzione sono due operazioni simili, con le quale si ottiene una collezione più piccola o a singolo valore.

Supponiamo una lista di tipo *List[Int]* contenente i numeri dall'uno al quattro, come da definizione sottostante.

```
val d = List(1, 2, 3, 4)
```

Data una funzione di tipo *(Int, Int) => Int*, la riduzione da sinistra consiste nell'esecuzione di questa funzione tra il primo elemento ed il secondo, poi tra il risultato ed il terzo elemento, e poi tra quest'ultimo risultato e il quarto elemento.

Il seguente esempio mostra come eseguire la riduzione da sinistra con la funzione di moltiplicazione.

```
val riduzione = d reduceLeft(_ * _)
```

Il risultato è 24, ed è il prodotto della sequenza di operazioni illustrata in seguito.

```
(((1 * 2) * 3) * 4)
```

Il ripiegamento, a differenza della riduzione, parte da un valore fornito separatamente e lo usa per dare il via alla sequenza di operazioni al posto del primo elemento della lista.

```
val ripiegamento = d.foldLeft(5)(_ * _)
```

In questo caso il risultato è 96, ed è il risultato della seguente espressione.

```
((((5 * 1) * 2) * 3) * 4)
```

Oltre a *reduceLeft* e *foldLeft*, la classe *Iterable* mette a disposizione anche i metodi *reduceRight* e *reduceLeft*, che effettuano le stesse operazioni ma nell'ordine inverso.

### 4.2.5 Unione

L'unione è un'operazione che date due liste ne costruisce una formata da coppie di valori, unendo il primo elemento di una lista al primo della seconda lista, il secondo elemento della prima lista al secondo elemento della seconda lista e via dicendo. I casi d'uso più interessanti sono quelli che coinvolgono la struttura dati *Stream*. Il seguente esempio fa uso della collezione *fibonacci* definita nella sezione 4.1.3 e ne applica il metodo di unione *zip*.

```
var indici = Vector(1, 2, 3, 4, 5)
println(indici zip fibonacci)
```

Il risultato è un vettore di coppie di valori, che sono nel caso specifico l'indice e il valore che la serie assume in quella posizione.

```
Vector((1, 0), (2, 1), (3, 1), (4, 2), (5, 2))
```

Nel caso di liste di lunghezze diverse, le dimensioni della collezione risultante sono quelle della lista con meno elementi. Inoltre, il tipo della collezione generata è sempre quello a sinistra dell'operatore. Queste informazioni si possono verificare con l'esempio sottostante.

```
println(List(1, 2, 3) zip Vector(1, 2))
```

Infatti il risultato è il seguente.

```
List((1, 1), (2, 2))
```

## 4.3 Invocazione per nome

Nei casi standard l'invocazione di un metodo viene effettuata quando si è a conoscenza del valore di tutte le variabili. Questo tipo di chiamata si chiama appunto invocazione per valore. L'invocazione per nome è, al contrario, la chiamata che si effettua quando non si è ancora a conoscenza del valore di una o più variabili.

Il seguente esempio utilizza l'invocazione per nome, e consiste nella definizione di un ciclo.

```
def ciclo(cond: => Boolean)(f: => Unit) {
	if (cond) {
		f
		ciclo(cond)(f)
	}
}
```

Innanzitutto si nota che il ciclo è stato definito come *funzione curry*, di cui si è parlato nella sezione 2.2. I due parametri sono la condizione e la funzione da eseguire ad ogni iterazione. Per prima cosa viene valutata la condizione, e se è valida viene eseguita la funzione. Fatto ciò viene richiamato il ciclo in forma ricorsiva, e si capirà come si possano evitare problemi di *stack overflow* nella sezione riguardante la ricorsione in coda.

La chiave per la comprensione dell'esempio sta sulla condizione *cond*. Proprio questo è l'esempio di invocazione per nome. In questo modo *cond* viene valutata all'interno dell'*if*, e non in fase di chiamata. Se il valore di *cond* fosse determinato alla chiamata, infatti, il ciclo funzionerebbe correttamente solo nel caso in cui la condizione fosse falsa sin dall'inizio. Se inizialmente la condizione fosse vera e non ci fosse una chiamata per nome, il valore di *cond* sarebbe costante e il risultato sarebbe un ciclo infinito.

Per utilizzare il ciclo definito in precedenza, il codice da usare è il seguente.

```
var n : Int = 10
ciclo(n > 0) {
	println("Iterazione")
	n -= 1
}
```

## 4.4 Valori ritardati

Con i valori ritardati, Scala permette di definire delle variabili immutabili la cui valutazione è rimandata al momento dell'utilizzo. La sintassi è la seguente.

```
var a : Int
lazy val b = a * a
```

Nell'esempio è stata definita normalmente una variabile *a*, mentre la variabile *b*, contenente il quadrato di *a*, è stata definita come *lazy* per forzarne la valutazione ritardata.

A seguire un esempio di utilizzo corretto.

```
a = 10
println(b)
```

In questo caso il risultato dell'operazione è 100. È necessario però prestare attenzione all'uso di questa modalità. Nel caso dell'uso involontario della variabile *lazy* si potrebbero creare delle situazioni in cui la valutazione venga effettuata prima del previsto, come nell'esempio successivo.

```
println(b)
a = 10
println(b)
```

In questo caso a l'uscita a video sarebbe due volte zero, in quanto la valutazione viene eseguita una volta soltanto ed in questo caso prima dell'inizializzazione della variabile necessaria per il calcolo.

C'è anche la possibilità di definire la variabile tramite un metodo, come illustrato nel seguente esempio.

```
var a : Int
lazy val b = { println(“Valutazione di b”); a * a }
```

Così facendo la variabile *b* assumerà lo stesso valore, ovvero il quadrato di *a*, ma sarà possibile visualizzare un messaggio a console che indica il momento della valutazione. In questo modo, se necessario, è possibile capire in quale parte del programma si effettua una chiamata errata alla variabile.

## *4.5 Ricorsione*

Come strumento la ricorsione tende ad essere accantonata nei linguaggi di programmazione a causa di varie problematiche che implica, come il rischio di *stack overflow* e la riduzione di prestazioni dovuta alle chiamate annidate alla stessa funzione. In Scala le ottimizzazioni apportate dal compilatore annullano questi problemi, annullando quindi le problematiche tradizionali e permettendo allo sviluppatore di riconsiderare le soluzioni ricorsive come valide alternative. Specialmente dal punto di vista funzionale, la ricorsione è una modalità molto interessante dal momento che favorisce l'immutabilità delle variabili. L'esempio più classico è il fattoriale, nel seguente esempio ne è mostrata un'implementazione iterativa.

```
def fattoriale_iterativo(i : BigInt) : BigInt = {
	var r = BigInt(1)
	for(j <- 2 to i.intValue)
		r *= j
	r
}
```

È evidente l'impossibilità, nella modalità iterativa, di implementare una soluzione completamente immutabile. In questo caso le variabili mutabili sono due, *r* e *j*. Si potrebbe creare nuovamente *r* ad ogni iterazione rendendola immutabile, ma *j*,in quanto indice del ciclo *for*, non può in alcun modo non essere

mutabile.

Al contrario, come mostra il seguente esempio, anche la più semplice delle implementazioni ricorsive può fare uso esclusivamente di variabili immutabili.

```
def fattoriale_ricorsivo(i : BigInt) : BigInt = {
	if(i <= 2)
		i
	else
		i * fattoriale_ricorsivo(i - 1)
}
```

L'ottimizzazione da parte del compilatore non è automatica. Eseguendo questa versione del codice su una JVM con configurazione normale si può notare come un valore di ingresso pari a 10000 possa bastare per causare un errore di *stack overflow*. Perché il compilatore effettui l'ottimizzazione è necessario implementare una ricorsione in coda, in inglese *tail recursion*. Nella ricorsione in coda la chiamata ricorsiva viene fatta come ultima operazione, mentre nell'esempio precedente l'ultima azione era la moltiplicazione del risultato con la variabile *i*. L'esempio sotto riportato fa un corretto uso della ricorsione in coda con l'appoggio di un metodo annidato.

```
def fattoriale(i : BigInt) : BigInt = {
	def ricorsione(i : BigInt, j : BigInt) : BigInt = {
		if(i < 2)
			j
		else
			ricorsione(i - 1, i * j)
	}
	ricorsione(i, 1)
}
```

Una chiamata a questa implementazione con lo stesso numero che causava l'errore di *stack overflow* nel precedente esempio mostrerà come in questo caso l'ottimizzazione sia effettivamente applicata dal compilatore.

## *4.6 Programmazione concorrente*

Notoriamente la programmazione concorrente è qualcosa di molto insidioso a causa delle complicazioni intrinseche che comporta. Basti pensare ai problemi di lock sulle risorse condivise, la disponibilità asincrona di risultati, condizioni di corsa e via dicendo. Su questo ambito Scala, come anticipato, porta delle semplificazioni non indifferenti anche grazie al suo lato funzionale.

### 4.6.1 Attori

Un ruolo di primaria importanza nella programmazione concorrente in Scala lo svolgono gli attori. Dal punto di vista pratico sono degli oggetti che ricevono dei messaggi ed in base a questi intraprendono delle azioni. Queste azioni potrebbero concludersi internamente o consistere nell'invio di messaggi ad altri attori. È importante comprendere come la parallelizzazione sia data dalla struttura che si può creare tramite gli attori, e non dall'attore in sé. L'attore, infatti, ha una coda di messaggi e li processa usando un singolo thread. È importante notare come questo significhi che le operazioni di I/O sono il più possibile da evitare all'interno degli attori, in quanto comportano delle latenze intrinseche che sono in contraddizione con il modo di operare degli stessi.

Per creare un attore è necessario estendere la classe *Actor* dopo averla importata ed implementarne il

metodo astratto *act*, come mostrato nel seguente esempio.

```
import scala.actors.Actor
class Attore extends Actor {
	def act() {
		println("Attore avviato")
	}
}
```

Una volta istanziato l'attore è necessario anche avviarlo. La modalità potrebbe sembrare famigliare ai *Thread* e al metodo *run*, per chi ha utilizzato i thread in Java. Nel seguente esempio viene creato ed avviato l'attore in precedenza definito.

```
val a = new Attore
a.start
```

C'è anche la possibilità di usare i metodi *factory* per la creazione degli attori. L'esempio seguente è analogo al precedente, si noterà quindi come la funzione passata al metodo *factory* sia l'implementazione del metodo *act* e necessiti di un diverso import.

```
import scala.actors.Actor._
val a = actor {
	println("Attore avviato")
}
a.start
```

Fino a qui si sono viste la definizione e l'avvio di un attore. La parte più interessante, quella operativa, è quella che si occupa della ricezione e della gestione dei messaggi, ed è illustrata nel seguente esempio.

```
val a = actor {
	loop {
		react {
			case i : Int => println("Ricevuto l'intero " + i)
			case d : Double => println("Ricevuto il numero " + d)
			case s : String => println("Ricevuta la stringa " + s)
			case _ => println("Messaggio non riconosciuto")
		}
	}
}
a.start
```

Rispetto agli esempi precedenti gli elementi introdotti sono due, *loop* e *react*. *Loop* è sostanzialmente un ciclo infinito che mantiene in ascolto l'attore, si potrebbe paragonare ad un *while(true)* contenente un *try/catch*, in modo da non interrompersi a causa di eccezioni.

Il metodo *react* invece è il metodo che viene eseguito alla ricezione di un messaggio, ed in questo caso al suo interno è stato usato il costrutto *match*. L'uso del *pattern matching* nella ricezione dei messaggi

dell'attore non è un obbligo, ma la soluzione più semplice. In alternativa infatti sarebbe stato necessario usare un metodo di tipo *PartialFunction* per accettare e gestire tutti i tipi di oggetto che l'attore può ricevere come messaggio.

L'unico pezzo mancante a questo punto è l'invio di un messaggio all'attore, mostrato nell'esempio sottostante.

```
a ! 1
a ! 2.0
a ! "AAA"
a ! List(1, 2, 3)
```

Come si può notare, l'invio di un messaggio ad un attore si effettua usando riferimento allo stesso seguito da un punto esclamativo e dall'oggetto da inviare. Eseguendo questo esempio a fronte dell'attore definito in precedenza si potranno testare tutte e quattro le casistiche implementate tramite *pattern matching*.

È importante sapere che i messaggi non elaborati dall'attore, ad esempio perché non riconosciuti nel *pattern matching*, rimangono in coda. Ad esempio rimuovendo il caso di default, nell'ultimo esempio la lista sarebbe rimasta in coda. La gestione del caso di default quindi è importante, in modo da evitare *memory leak*, perdita di prestazioni e perdita di informazioni.

Oltre all'operatore *!*, che effettua la chiamata in modo asincrono senza quindi attenderne il risultato, ci sono altri due operatori. L'operatore *!!* effettua la chiamata sempre in modo asincrono, e ottiene come risultato un oggetto *Future* di cui si parlerà nella prossima sezione. L'operatore *!?*, invece, richiede il risultato in maniera sincrona, quindi il suo uso genera una chiamata bloccante.

L'attore, oltre al metodo *react*, mette a disposizione anche il metodo *receive*. La differenza tra i due è importante, in quanto il primo viene invocato solamente in caso di ricezione di messaggi, mentre il secondo rimane in ascolto impegnando quindi delle risorse che potrebbero essere rilasciate. È quindi buona pratica evitare l'utilizzo del metodo *receive*, se possibile.

### 4.6.2 Futuri

Un ulteriore strumento messo a disposizione da Scala nella programmazione concorrente sono i futuri, in inglese *Future*. Si tratta di una struttura dati utile per ottenere i risultati di un'operazione concorrente, sia in maniera bloccante che non bloccante.

Utilizzati assieme agli attori, i *Future* permettono di creare una sorta di segnaposto del risultato. Il seguente esempio fa uso dell'attore definito nella sezione 4.6.1, e nella chiamata genera un *Future*.

```
val f = a !! 1
```

Ottenuto l'oggetto *f*, tramite il metodo *isSet* è possibile verificare se è il valore è stato computato o meno. Per ottenere il dato stesso è necessaria una chiamata al metodo *value*, che chiaramente risulta bloccante nel caso il risultato di *isSet* sia negativo. In quest'ultimo caso, infatti, viene attesa l'elaborazione del valore per poterlo restituire.

I Future si possono usare anche autonomamente, evitando l'*overhead* degli attori e la relativa complessità di sviluppo. Nel seguente esempio viene fatto uso della funzione *fattoriale* definita nel capitolo 4.5, e ne viene demandato il calcolo ad un *Future*.

```
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.Future
val future = Future {
```

```
        fattoriale(5)
}
future foreach println
```

In questo caso il calcolo non è in carico ad un attore, ma al *Dispatcher* di default.

### 4.6.3 Thread

Scala fornisce anche la possibilità di utilizzare i thread classici. Rispetto a Java la forma è abbreviata, non è necessario sovrascrivere il metodo *run* dell'interfaccia *Runnable* ed incapsularlo in un oggetto *Thread*. È sufficiente creare un nuovo thread e passare come parametro la funzione da eseguire, come mostrato nel seguente esempio.

```
new Thread {
        println("Azione eseguita su nuovo thread")
}
```

# 5 Applicazioni

Nelle precedenti sezioni sono stati trattati i principali aspetti relativi alla sintassi del linguaggio e alla struttura della piattaforma Scala. In questo capitolo saranno affrontati aspetti di particolare interesse e di natura più pratica. I relativi approfondimenti quindi non mirano soltanto alla comprensione degli strumenti che Scala mette a disposizione, ma anche a fornire esempi e soluzioni utili a problematiche comuni.

## *5.1 Integrazione con Java*

Una delle caratteristiche che rende Scala più appetibile rispetto a linguaggi simili, come già detto in precedenza, è la stretta relazione che ha con Java. Tale relazione permette di includere codice Java in programmi Scala e viceversa. Nel primo caso è quindi possibile riutilizzare procedure più o meno complesse già sviluppate e testate in Java senza doverle riscrivere. Nel secondo caso invece è possibile prevedere lo sviluppo di moduli in Scala all'interno di software Java con il minimo sforzo.

Un progetto iniziato nel 2009 mirava ad ottenere gli stessi risultati di integrazione anche con la piattaforma .Net. Purtroppo questo progetto non ha mai raggiunto una versione stabile e al momento non è attivamente supportato, ragion per cui non sarà approfondito nella presente sezione.

### 5.1.1 Inclusione di codice Java

In questa prima sezione sarà trattata l'inclusione e l'utilizzo di classi Java all'interno di un programma sviluppato in Scala. Sarà presa come riferimento la classe *PersonaJava*, la cui definizione è riportata in seguito.

```
public class PersonaJava {
	private String _nome, _cognome;

	public PersonaJava(String nome, String cognome) {
		_nome = nome;
		_cognome = cognome;
	}

	public String getDenominazione() {
		return _cognome + " " + _nome;
	}
}
```

Come richiesto da Java, la definizione di questa classe dovrà essere riportata all'interno del file *PersonaJava.java* e compilata tramite il comando *javac*. L'operazione produrrà il file *PersonaJava.class*, contenente l'effettivo *bytecode* per la JVM.

Richiamare questa classe all'interno del codice Scala è un'operazione del tutto trasparente, come lo sarebbe all'interno del codice Java. Utilizzando Scala in modalità compilata è necessario definire una classe pubblica contenente il metodo statico *main*, e da questo istanziare ed utilizzare la variabile.

```
object TestPersonaJavaCompilato {
	def main(args: Array[String]) {
		val persona = new PersonaJava("Mario", "Rossi")
		println(persona.getDenominazione)
```

```
        }
}
```

A questo punto è possibile compilare la classe *TestPersonaJavaCompilato*, la cui definizione dov'essere nella stessa cartella e preferibilmente all'interno del file *TestPersonaJavaCompilato.scala*. Compilando tale file con il comando *scalac*, a prescindere dal nome dato al file contenente il codice sorgente, si otterrà il file *TestPersonaJavaCompilato.class*. Eseguendo il seguente comando si otterrà il risultato atteso, ovvero l'output *Rossi Mario*.

```
scala TestPersonaJavaCompilato
```

Si può testare il funzionamento dell'inclusione anche richiamando direttamente l'esecuzione dalla Java Virtual Machine senza passare tramite il comando *scala*. Per fare ciò è necessaria l'aggiunta del parametro *classpath*, il seguente comando infatti non funzionerà.

```
java TestPersonaJavaCompilato
```

Il comando sopra riportato causerà infatti un'eccezione di tipo *NoClassDefFound* su una classe della piattaforma Scala. Il problema non ha nulla a che fare con l'inclusione del codice Java, e si può risolvere tramite una modifica alla chiamata come riportato in seguito.

```
java -classpath "C:\Program Files\scala\lib\scala-library.jar;." TestPersonaJavaCompilato
```

Dove *C:\Program Files\scala* è il percorso di installazione della piattaforma. I due caratteri a seguito del percorso, il punto e virgola ed il seguente punto, indicano al compilatore di includere anche la cartella presente nella *classpath*, onde evitare il mancato ritrovamento della classe *TestPersonaJavaCompilato*.

In modalità interpretata l'uso della classe definita in Java è altrettanto semplice. È sufficiente creare un file nella stessa cartella con il seguente contenuto.

```
val persona = new PersonaJava("Mario", "Rossi")
println(persona.getDenominazione)
```

Come si potrà vedere, eseguendo il file con il comando *scala* il risultato sarà lo stesso ottenuto nell'esempio precedente.

L'inclusione, infine, si può effettuare allo stesso modo anche in modalità REPL. La shell interattiva va lanciata collocandosi nella stessa cartella di *PersonaJava.class*, i comandi daranno un risultato analogo a quello mostrato in seguito.

```
scala> val persona = new PersonaJava("Mario", "Rossi")
persona: PersonaJava = PersonaJava@27573872
scala> println(persona.getDenominazione)
Rossi Mario
```

Come si è potuto vedere l'inclusione di una classe è un'operazione molto semplice. L'inclusione di una libreria *jar* necessita di alcuni accorgimenti nella configurazione del parametro *classpath*, ma a parte questo rimane altrettanto semplice. Il seguente esempio riporta l'inclusione della classe *DoubleFFT_1D* dalla libreria JTransforms. Per semplicità viene creata un'istanza della classe e ne viene soltanto

stampato il nome, a prova che l'importazione funzioni.

```
import edu.emory.mathcs.jtransforms.fft.DoubleFFT_1D

object TestJTransformsCompilato {
	def main(args: Array[String]) {
		var fft = new DoubleFFT_1D(512)
		println(fft.getClass.getName)
	}
}
```

In questo caso la compilazione dev'essere eseguita includendo il *jar* nella *classpath*, con il seguente comando.

```
scalac -classpath “jtransforms-2.4.jar” TestJTransformsCompilato.scala
```

Allo stesso modo, anche l'esecuzione tramite il comando *scala* necessita della modifica al parametro *classpath*.

```
scala -classpath “jtransforms-2.4.jar;.” TestJTransformsCompilato
```

Come si può notare anche in questo caso è stata aggiunta la cartella locale con un punto a seguito del punto e virgola per la corretta esecuzione dell'esempio. Il risultato atteso è riportato in seguito.

```
edu.emory.mathcs.jtransforms.fft.DoubleFFT_1D
```

Volendo usare direttamente la Java Virtual Machine, rispetto a prima è sufficiente un'ulteriore inclusione nel parametro *classpath*.

```
java -classpath “C:\Program Files\scala\lib\scala-library.jar;jtransforms-2.4.jar;.”
	TestJTransformsCompilato
```

Come si può intuire, l'inclusione in modalità interpretata è del tutto analoga. Il codice riportato in seguito è il contenuto nel file *TestJTransformsInterpretato.scala*.

```
import edu.emory.mathcs.jtransforms.fft.DoubleFFT_1D
var fft = new DoubleFFT_1D(512)
println(fft.getClass.getName)
```

Per l'esecuzione il comando corretto è il seguente.

```
scala -classpath “jtransforms-2.4.jar” TestJTransformsInterpretato.scala
```

Infine, in modalità REPL è sufficiente lanciare la shell interattiva con il seguente comando.

```
scala -classpath “jtransforms-2.4.jar”
```

La sequenza di comandi darà un risultato analogo a quello riportato in seguito.

```
scala> import edu.emory.mathcs.jtransforms.fft.DoubleFFT_1D
import edu.emory.mathcs.jtransforms.fft.DoubleFFT_1D
scala> var fft = new DoubleFFT_1D(512)
fft: edu.emory.mathcs.jtransforms.fft.DoubleFFT_1D =
       edu.emory.mathcs.jtransforms.fft.DoubleFFT_1D@64afb650
scala> println(fft.getClass.getName)
edu.emory.mathcs.jtransforms.fft.DoubleFFT_1D
```

## 5.1.2 Inclusione di codice Scala

In questa sezione sarà trattata l'inclusione di una classe sviluppata in Scala all'interno di un programma sviluppato in Java.

Nel seguente esempio si riporta il contenuto del file *FibonacciScala.scala*, utilizzato per effettuare l'inclusione. L'implementazione riprende quanto mostrato nella sezione 4.1.3, usando la struttura dati *Stream* per gestire una serie dal numero potenzialmente infinito di elementi con la valutazione ritardata.

```
object FibonacciScala {
       private val fibonacci = {
              def f(a : BigInt, b : BigInt) : Stream[BigInt] = a #:: f(b, a + b)
              f(0, 1)
       }
       def get(i : Int) = fibonacci(i)
}
```

Questo file va normalmente compilato tramite il comando *scalac*.

In seguito è riportato il codice della classe Java che all'interno del metodo statico *main* usa la classe *FibonacciScala* per calcolare e stampare a video i valori nella possizione numero dieci, cento e mille della serie di Fibonacci.

```
public class TestFibonacciScala {
       public static void main(String[] args) {
              System.out.println(FibonacciScala.get(10));
              System.out.println(FibonacciScala.get(100));
              System.out.println(FibonacciScala.get(1000));
       }
}
```

La compilazione della classe *TestFibonacciScala* richiederà l'inclusione nella *classpath* delle librerie Scala usate, dal momento che queste non sono incluse in Java. Nell'esempio sono state usati gli oggetti *scala.math.BigInt* e *scala.collection.immutable.Stream*, per cui è sufficiente includere *scala-library.jar*.

```
javac -classpath "C:\Program Files\scala\lib\scala-library.jar;." TestFibonacciScala.java
```

Da notare l'inclusione sia della libreria Scala che del percorso corrente. Analoga è la chiamata per l'esecuzione del file, riportata in seguito.

```
java -classpath "C:\Program Files\scala\lib\scala-library.jar;." TestFibonacciScala
```

### 5.1.3 Strumenti *scalap* e *javap*

Utilizzando classi compilate ci si trova spesso ad avere la necessità di capire come sono definite della classi, qual è la firma di un metodo specifico e via dicendo. A questo proposito sono estremamente utili gli strumenti *scalap* e *javap*. Il primo, messo a disposizione dalla piattaforma Scala, tramite decompilazione permette di vedere la definizione in linguaggio Scala della classe e dei relativi metodi e proprietà. Il secondo, fornito dalla piattaforma Java, permette di fare lo stesso in linguaggio Java.

Effettuando la decompilazione dal *bytecode*, entrambi i comandi funzionano indifferentemente a prescindere dal linguaggio con cui è stato compilato il codice sorgente. Questi strumenti sono dunque molto utili per capire la firma di un metodo o le caratteristiche di una classe sviluppata in Scala e vista da Java o viceversa.

In seguito è riportato risultato del comando *scalap PersonaJava*, utilizzato in precedenza.

```
package PersonaJava;
class PersonaJava extends AnyRef {
	final var _cognome: java.lang.String;
	final var _nome: java.lang.String;
	def getDenominazione(): java.lang.String
	def this(java.lang.String, java.lang.String): scala.Unit
}
```

Allo stesso modo si può visualizzare la classe implementata in Scala tramite il comando *javap FibonacciScala*. Il risultato è riportato in seguito.

```
public final class FibonacciScala extends java.lang.Object {
	public static scala.math.BigInt get(int);
}
```

## *5.2 Utilizzo dell'Xml*

L'Xml è certamente uno dei linguaggi di scambio dati più utilizzati in ambito web. Gli strumenti che Scala mette a disposizione ne rendono estremamente semplice il parse e ne facilitano la scrittura nel rispetto delle validazioni.

### 5.2.1 Generazione

La scrittura dell'Xml è supportata direttamente a livello di codice. Il seguente esempio, eseguito in modalità REPL, dà evidenza di questa funzionalità.

```
scala> <test>contenuto</test>
res0: scala.xml.Elem = <test>contenuto</test>
```

Come si può notare il codice Xml scritto in maniera diretta, ovvero senza delimitatori di stringa, è stato riconosciuto e convertito in oggetto *scala.xml.Elem*.

Ciò che risulta ancora più interessante è che il compilatore effettua già un controllo di validazione del codice Xml inserito. Come mostra il seguente esempio, infatti, l'errata chiusura di un tag viene subito segnalata.

```
scala> <test>contenuto</testa>
<console>:1: error: in XML literal: expected closing tag of test
       <test>contenuto</testa>
                        ^
<console>:1: error: in XML literal: start tag was here: test>
       <test>contenuto</testa>
        ^
```

Nello specifico, tale controllo viene effettuato in fase di compilazione e non in runtime, evitando a priori errori dovuti alla distrazione o alla complessità della struttura dati serializzata.

Nei prossimi esempi verrà usata una struttura dati che rappresenta la lista di itinerari percorribili in aereo a fronte di una ricerca fatta in base a data, luogo di partenza e luogo di arrivo. Ogni elemento di questa lista è una soluzione contraddistinta da un id univoco, dal costo e da una lista di segmenti. I segmenti non sono altro che i voli di cui è composta la soluzione, e contengono informazioni su ora e luogo di partenza e di arrivo.

In seguito è riportata l'implementazione della classe *Segmento*, con le caratteristiche descritte.

```
import java.util.Date
class Segmento(val Origine:String, val Destinazione:String,
                        val Partenza:Date, val Arrivo:Date) {
       def durata = {
              (Arrivo.getTime - Partenza.getTime) / 60000
       }
       def toXml = {
              <Segmento
                     Origine={Origine}
                     Destinazione={Destinazione}
                     Partenza={Partenza}
                     Arrivo={Arrivo} />
       }
}
```

Ciò che si nota di interessante è la possibilità di includere direttamente i valori delle variabili, attraverso la parentesi graffa, all'interno del codice Xml. Chiamando il metodo *toXml* su un'istanza dell'oggetto *Segmento*, si ottiene infatti un oggetto di tipo *scala.xml.Elem* come in precedenza, con il valore delle variabili indicate negli attributi. In questo modo è la piattaforma ad occuparsi di racchiudere la stringa tra virgolette nel caso degli attributi, fare un *escape* di eventuali virgolette o caratteri riservati all'interno della variabile da scrivere e gestirne la codifica.

Questa funzionalità non solo facilita la scrittura di codice evitando lunghe concatenazioni di stringhe,

ma elimina intrinsecamente una serie di problematiche che si possono verificare in casistiche poco frequenti e che spesso in fase di sviluppo vengono tralasciate o dimenticate.

Nel seguente esempio è riportata la classe *Soluzione*, sempre con le caratteristiche descritte in precedenza.

```
class Soluzione(val ID:Int, val Prezzo:Double) {
	var Segmenti = List[Segmento]()

	def durata = {
		var d : Long = 0
		for(Segmento <- Segmenti)
			d += Segmento.durata
		d
	}
	def toXml = {
		<Soluzione ID={ID.toString} Prezzo={Prezzo}>
			{ Segmenti.map(_.toXml) }
		</Soluzione>
	}
}
```

In questo caso, oltre alla presenza di variabili, nel metodo *toXml* si nota anche l'inclusione del risultato della chiamata ad un metodo. L'istruzione *Segmenti.map(_.toXml)* esegue sostanzialmente tre azioni. In primo luogo carica la lista dei segmenti di cui è composta la soluzione, su ogni elemento esegue il metodo *toXml*, ed infine ne concatena i risultati formando una lista di oggetti *scala.xml.Elem*. Questi vengono direttamente inclusi all'interno del tag *Soluzione*.

Nella serializzazione dell'attributo *ID*, di tipo *Int*, si noterà anche la chiamata al metodo *toString*. Il cast esplicito potrebbe sembrare opzionale, ma non lo è. L'inclusione delle variabili nell'Xml è da fare previo conversione a stringa.

Data una lista di soluzioni, sarà possibile generare un Xml contenente la lista con la seguente istruzione.

```
val xml = <Soluzioni>
	{ soluzioni.map(_.toXml) }
</Soluzioni>
```

A questo punto si potrà inviare come stringa, o salvare su file come mostra la seguente istruzione.

```
XML.save("Soluzioni.xml", xml)
```

In seguito è riportato come riferimento un possibile risultato. Nello specifico si tratta di una lista, per semplicità limitata a due elementi, di itinerari Milano (*MXP* è il codice IATA dell'aeroporto di Malpensa e *LIN* di Linate) San Francisco (codice IATA *SFO*) con partenza il 30 Settembre 2013.

```
<Soluzioni>
	<Soluzione ID="1" Prezzo="1172.18">
		<Segmento Origine="LIN" Destinazione="CPH"
```

```
                Partenza="2013-09-30T06:00:00.000+0200"
                Arrivo="2013-09-30T08:00:00.000+0200" />
        <Segmento Origine="CPH" Destinazione="SFO"
                Partenza="2013-09-30T12:25:00.000+0200"
                Arrivo="2013-09-30T14:45:00.000-0700"/>
    </Soluzione>
    <Soluzione ID="2" Prezzo="1173.85">
        <Segmento Origine="MXP" Destinazione="CPH"
                Partenza="2013-09-30T11:20:00.000+0200"
                Arrivo="2013-09-30T13:25:00.000+0200" />
        <Segmento Origine="CPH" Destinazione="ORD"
                Partenza="2013-09-30T15:40:00.000+0200"
                Arrivo="2013-09-30T17:40:00.000-0500" />
        <Segmento Origine="ORD" Destinazione="SFO"
                Partenza="2013-09-30T20:00:00.000-0500"
                Arrivo="2013-09-30T22:17:00.000-0700" />
    </Soluzione>
</Soluzioni>
```

## 5.2.2 Parse

Grazie agli operatori barra retroversa e doppia barra retroversa messi a disposizione da Scala, l'accesso in modalità *XPath* ai dati contenuti in un file Xml è estremamente semplice.

Innanzitutto è necessario caricare l'Xml in un oggetto di tipo *scala.xml.Elem*. Questo requisito è già soddisfatto se l'Xml è stato costruito come mostrato nella precedente sezione. Nel caso si debba caricare l'Xml da un file, le istruzioni da eseguire sono le seguenti.

```
import scala.xml._
val xml = XML.loadFile("Soluzioni.xml")
assert(xml.isInstanceOf[scala.xml.Elem])
```

Avendo a disposizione l'Xml su una stringa il procedimento sarebbe stato altrettanto semplice, usando il metodo statico *loadString* anziché *loadFile* messo a disposizione dalla classe *scala.xml.XML*.

Nei seguenti esempi sarà preso in considerazione un file Xml del formato mostrato nella precedente sezione. Il primo mostra l'uso dell'operatore barra retroversa.

```
val soluzioni = xml \ “Soluzione”
```

In questo modo è stata caricata la lista dei nodi dal nome *Soluzione* presenti nella *root* del documento all'interno della variabile *soluzioni*. Le soluzioni così estratte sono di tipo *scala.xml.NodeSeq*, e si possono iterare. Allo stesso modo si può iterare la lista di segmenti che la compongono. Il seguente esempio mostra come fare al fine di stampare ID e prezzo a livello di soluzione e tutti i dati relativi ai singoli segmenti.

```
for(soluzione <- xml \ “Soluzione”) {
    println((soluzione \ “@ID”).text + “ - ” + (soluzione \ “@Prezzo”).text + “ €”)
    for(segmento <- soluzione \ “Segmento”)
        println(“\t” + (segmento \ “@Origine”).text + “ - ”
```

```
                    + (segmento \ “@Destinazione”).text
                    + “  ” + (segmento \ “@Partenza”).text
                    + “ - ” + (segmento \ “@Arrivo”).text)
}
```

Dall'esempio si può intuire che la notazione necessaria all'estrazione di un attributo differisce da quella necessaria all'estrazione di un nodo solamente per il simbolo @ che ne precede il nome. Anche in questo caso l'oggetto è di tipo *scala.xml.NodeSeq*, infatti per usarne il valore testuale nella stampa viene richiamato il metodo *text*.

La funzione dell'operatore barra retroversa è quindi di ricavare i figli di livello direttamente inferiore rispetto al nodo su cui si sta invocando, o gli attributi del nodo stesso. L'operatore contraddistinto dalla doppia barra retroversa, invece, ricava tutti i nodi o gli attributi dal nome fornito facendo una ricerca ricorsiva a partire dal nodo su cui è invocato.

Il seguente esempio mostra come ricavare tutti i segmenti usati a prescindere dalla soluzione in cui sono inclusi.

```
val segmenti = xml \\ “Segmento”
```

La stessa istruzione chiamata con la singola barra retroversa non avrebbe prodotto alcun risultato, in quanto tutti i segmenti sono annidati su nodi dal nome *Soluzione*, e quindi non sono figli diretti del nodo di *root* rappresentato dall'oggetto *xml*.

Per eseguire ulteriori elaborazioni è necessario combinare gli operatori con la sintassi di Scala che si è già vista nei primi capitoli. Ad esempio, la seguente istruzione permette l'estrazione dal file Xml di una soluzione dato l'ID, nel caso specifico la numero uno.

```
val soluzione1 = xml \ “Soluzione” filter { _ \ “@ID” exists (_.text == “1”) }
```

Come atteso, il risultato di questa istruzione sarà un oggetto di tipo *NodeSeq* contenente solamente il nodo con proprietà ID pari a uno. Entrando nel dettaglio, in un primo step vengono estratti tutti i nodi dal nome *Soluzione*, dopo di che a questi nodi viene applicato un filtro. Questo filtro è definito da una funzione anonima, la quale non fa altro che estrarre tutti l'attributo *ID* del nodo. Essendo il risultato di questa operazione per forza di cose un oggetto *NodeSeq* non è possibile fare un controllo diretto, ma va demandato ad un'altra funzione anonima richiamata da *exists*. L'attributo *ID* infatti è trattato come unico nodo di una lista, e la funzione *exists* verifica che quest'unico nodo rispetti la condizione indicata. Tale condizione è un'uguaglianza della conversione a stringa del contenuto dell'elemento al numero in formato stringa. Il motivo per cui si è utilizzata una stringa è che, analogamente a come vengono inseriti, tutti i contenuti degli attributi e dei nodi sono in forma testuale. Ulteriori conversioni sono da fare manualmente.

Ulteriori operazioni, come l'ordinamento delle soluzioni in base al prezzo, si possono realizzare in modo simile al filtro per *ID*. Queste però implicano delle complicazioni particolarmente a causa dell'ulteriore livello di iterazione necessario per la lettura del singolo attributo. In queste situazioni si giustifica piuttosto l'inserimento dei dati caricati dall'Xml in una struttura coerente. A questo punto le elaborazioni sono estremamente più semplici.

## *5.3 Spray*

*Spray* è una libreria open source per la costruzione di interfacce REST/HTTP basata su Scala e sugli attori. Essendo asincrona, basata sugli attori, veloce, leggera e modulabile è un ottimo strumento per connettere le applicazioni Scala ad altre applicazioni basate su altre piattaforme, al fine di reperire o fornire informazioni.

Il sito del progetto è http://scala.io. Codice sorgente della libreria, strumenti per il test ed esempi sono

pubblicamente reperibili sul repository GIT all'indirizzo https://github.com/spray/spray. Oltre che come strumento da utilizzare è di ottima utilità come spunto nello stile di programmazione Scala e nella modalità di utilizzo degli attori.

### 5.3.1 Server

La realizzazione di un WebService REST si realizza in maniera semplice ed intuitiva tramite l'estensione di un attore e l'implementazione del metodo *receive*, tramite il quale viene identificata la richiesta e restituita la risposta.

A seguito è riportata la definizione di un WebService con alcuni due metodi di esempio. Il primo è l'indice, e restituisce una pagina html, mentre il secondo, *time*, restituisce l'ora all'interno di un file xml.

```
import scala.concurrent.duration._
import akka.actor._
import spray.can.Http
import spray.util._
import spray.http._
import HttpMethods._

class TestServer extends Actor with ActorLogging {
	import context.dispatcher

	def receive = {
		case _: Http.Connected =>
			sender ! Http.Register(self)

		case HttpRequest(GET, Uri.Path("/"), _, _, _) =>
			sender ! index

		case HttpRequest(GET, Uri.Path("/time"), _, _, _) =>
			sender ! timeXml

		case _: HttpRequest =>
			sender ! HttpResponse(status = 404, entity = "Action not found")
	}

	lazy val index = HttpResponse(
		entity = HttpEntity(`text/html`,
			<html>
				<head>
					<title>Spray WebService Test</title>
				</head>
				<body>
					<strong>Azioni disponibili:</strong>
					<ul>
						<li>time</li>
					</ul>
				</body>
			</html>.toString()
		)
	)
```

```
        def timeXml = HttpResponse(
                entity = HttpEntity(`application/xml`,
                        <response>
                                <time>{new java.util.Date().toString}</time>
                        </response>.toString()
                )
        )
}
```

La prima cosa, come consueto, sono gli import. Tra questi ci sono chiaramente tutti i *package* relativi ad attori e protocollo http. Si noti come l'import di *HttpMethods* permetta poi di usare GET, classe *case*, come discriminante nel pattern matching. All'interno della classe viene inoltre fatto l'import di *context.dispatcher*, in modo da permettere l'uso dei *Future*.

Come anticipato, tutta la logica del server sta all'interno del metodo *receive*, e viene gestita con il pattern matching. Come si può notare ogni casistica prevede un'elaborazione della risposta e una restituzione della stessa all'attore che l'ha inviata tramite l'operatore *!*.

In primo luogo viene identificata la casistica della connessione di un nuovo client, al quale viene risposto registrando l'oggetto corrente come gestore. In seguito c'è il metodo che restituisce l'indice. L'indice è definito come variabile non mutabile con valutazione ritardata, in modo da non essere elaborato fino alla prima richiesta. Osservando nel dettaglio si vedrà come si tratti di un oggetto di tipo *HttpResponse* in cui è specificata un'entità con content type *text/html* e contenuto definito come Xml e convertito a stringa. Il secondo metodo, *time*, è definito tramite funzione e non variabile immutabile, in quanto il risultato dovrà dipendere dall'istante della chiamata.

Infine è gestita la casistica in cui la chiamata fatta non corrisponda a nessuno degli Url definiti. In questo caso viene restituito un oggetto sempre di tipo *HttpResponse* ma con codice 404 e messaggio che indica l'inesistenza del percorso cercato.

Una volta implementato il server è necessario avviarlo. Per fare ciò sono necessari gli import riportati in seguito.

```
import akka.actor.{ActorSystem, Props}
import akka.io.IO
import spray.can.Http
```

Le istruzioni da eseguire, da inserire all'interno di una classe con metodo statico *main* se in modalità compilata, sono le seguenti.

```
implicit val system = ActorSystem()

val handler = system.actorOf(Props[TestServer], name = "handler")

IO(Http) ! Http.Bind(handler, interface = "localhost", port = 8080)
```

Per esempi più specifici, ad esempio sulla lettura dei parametri, degli *header* della richiesta, upload di files e tutte le funzionalità disponibili si rimanda alla documentazione e ai numerosi esempi disponibili nel sito del progetto *Spray*.

### 5.3.2 Client

*Spray* mette a disposizione tre diverse modalità di implementazione per i client di WebServices REST, a differenza è il livello a cui agiscono. Le API sono infatti:

- A livello connessione, per avere il pieno controllo su quando le connessioni sono aperte e chiuse, e su come le richieste sono schedulate in questi intervalli
- A livello host, per fare in modo che *Spray* si occupi della gestione del pool di connessioni per un host specifico
- A livello richiesta, per fare in modo che *Spray* si occupi di tutta la parte relativa alla gestione delle connessioni

Al livello di astrazione più alto, quello che probabilmente è sufficiente nella maggior parte dei casi, si può quindi agire a livello di richiesta. Il seguente esempio mostra come utilizzare questa modalità per accedere al contenuto del metodo *index* implementato nella precedente sezione.

```
import akka.io.IO
import akka.pattern.ask
import spray.can.Http
import spray.http._
import HttpMethods._

val response: Future[HttpResponse] =
       (IO(Http) ? HttpRequest(GET, Uri("http://localhost:8080"))).mapTo[HttpResponse]
```

Anche in questo caso l'implementazione risulta semplice ed intuitiva. All'interno della variabile immutabile *response* viene inserito il *Future* ricavato dall'invio di un messaggio di tipo *HttpRequest* all'agente *IO(Http)*. Come spiegato nella sezione 4.6, ciò significa che la chiamata non è bloccante, la variabile assumerà l'effettivo valore al momento in cui questo sarà reperito, durante l'esecuzione di altre istruzioni o tramite una chiamata bloccante.

Come per la parte server, anche nella parte client si invita a far riferimento alla documentazione e agli esempi presenti nel sito ufficiale del progetto per trovare informazioni più dettagliate.

# Conclusioni

Nell'imparare Scala ci si trova sin da subito ad affrontare una serie di informazioni apparentemente contrastanti. Si tratta di un linguaggio sia funzionale che ad oggetti. Si può compilare o utilizzare in modalità interpretata come linguaggio di scripting. E mano a mano che si approfondisce lo studio si scoprono diverse dicotomie simili, anche se di entità minore. Quando ci si pongono due obiettivi il rischio è che uno prevalga sull'altro, o che nessuno dei due venga veramente realizzato. Nel caso di Scala, però, il risultato ottenuto è molto buono. La coesistenza di programmazione funzionale e ad oggetti non introduce limitazioni in fase di sviluppo, al contrario permette di sfruttare le caratteristiche migliori di entrambe in ogni momento. La modalità interpretata permette di includere altri file, compilati o meno. Non ci si trova quindi ad avere un linguaggio di scripting limitato ma potenzialmente alla pari della controparte compilata.

In sostanza si può definire Scala come un ottimo esempio di coesistenza e sinergia tra diversi elementi. Si nota inoltre una particolare attenzione alle problematiche che sorgono più comunemente durante la scrittura del codice. I tratti, ad esempio, facilitano lo sviluppo e permettono di evitare la riscrittura di codice in diversi punti del programma, o la creazione di una gerarchia di classi forzata al fine di evitare tale ripetizione. Un altro esempio è sicuramente l'integrazione con l'Xml, scrivibile direttamente a codice e leggibile in maniera estremamente semplice.

Inoltre, come già detto più volte, un grosso vantaggio di Scala è di basarsi sulla Java Virtual Machine. Questo fattore garantisce da un lato l'affidabilità di una piattaforma di cui dispone una grande base di utenti da diversi anni. Piattaforma che tra l'altro può vantare il più alto numero di sistemi operativi supportati: Windows, Linux, Mac, Android, e molti altri. Dall'altro lato permette a due vie l'interazione tra il software scritto in Java e quello scritto in Scala. Utile sia nella creazione di moduli per Java che nell'utilizzo di moduli Java già scritti.

Di certo la disponibilità di un compilatore supportato per il .Net porterebbe dei considerevoli vantaggi. La piattaforma .Net gode di una maggior diffusione e stabilità, per non parlare delle prestazioni, per quanto riguarda i sistemi operativi Windows. Sempre in questo caso permette di sviluppare interfacce grafiche appoggiandosi alle Windows Form, soluzione molto più semplice, stabile e performante di quelle messe a disposizione da Java (DWT, Swing, JavaFX).

Tuttavia ciò che rende difficile adottare Scala in soluzioni professionali di piccole e medie dimensioni, al momento, è la sua diffusione.  Nonostante dal punto di vista pratico i vantaggi nell'utilizzo di questo linguaggio siano molti, spesso può risultare una scelta difficile dal punto di vista della manutenzione. Scegliere Scala per progetti di queste dimensioni può implicare il rischio che in futuro trovare una professionalità competente abbia un costo pari o superiore al costo di una completa riscrittura del software.

# Bibliografia


1. “Scala (programming language)”, http://en.wikipedia.org/wiki/Scala_(programming_language)
2. “Functional programming”, http://en.wikipedia.org/wiki/Functional_programming
3. Scala official documentation, http://www.scala-lang.org/documentation
4. Joshua D. Suereth (2012) “Scala in depth”, 1st Edition, Manning
5. Piancastelli G. (2009-10), “Programmare in Scala”, http://gpiancastelli.altervista.org/scala-it/index.html
6. Odersky M. (2013) “Scala by Example”, http://www.scala-lang.org/docu/files/ScalaByExample.pdf
7. Spray.io, http://spray.io