UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTA' DI INGEGNERIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di laurea triennale in Ingegneria Informatica

Tesina di laurea

IL LINGUAGGIO DI PROGRAMMAZIONE SCALA

Laureando: Gianluca Tartaggia

Relatore: Prof. Moro Michele

A. A. 2011/2012

INDICE

Introduzione 3
1 Concetti Iniziali 5
1.1 Variabili 6
1.2 Costrutto if 6
1.3 Costrutto for 7
1.4 Costrutto match 8
1.5 Ulteriori costrutti 9
2 Classi 11
2.1 Definire una classe 11
2.2 Costruttori ausiliari 11
2.3 Campi parametrici 12
2.4 Sovrascrivere campi e metodi 12
2.5 Visibilita’ 13
2.6 Oggetti singleton 15
2.7 Gerarchia dei tipi 16
2.8 Una prima applicazione 17
3 Tratti 19
3.1 Lavorare con i tratti 19
3.2 Riutilizzo del codice 20
3.2.1 Mixin 20
3.2.2 Tratti impilabili 21
3.3 Linearizzazione 21
3.4 Differenza dalla multiereditarieta’ 23
3.5 Una prima applicazione con i tratti 24
4 Funzioni 25
4.1 Definire una funzione 25
4.2 Funzioni locali 25
4.3 Funzioni di prima classe 26
4.4 Chiusure 27
4.5 Funzioni parzialmente applicate 27
4.6 Currying 28

4.7 Creare nuovi costrutti 29
5 Classi case e pattern matching 33
5.1 Metodo apply 33
5.2 Classi case 33
5.3 Pattern matching 34
5.4 Classi sealed 36
6 Tipi parametrici 39
6.1 Covarianza 39
6.2 Controvarianza 40
6.3 Limite inferiore e superiore 42
6.4 Tipi e membri astratti 43
6.5 Inizializzare membri astratti 44
6.6 Differenza tra tipi astratti e parametrici 45
7 Attori 47
7.1 Definire un attore 47
7.2 Inviare un messaggio 48
7.3 Ricevere un messaggio 48
7.4 Funzionamento in invio e ricezione 50
7.5 Esempio produttori consumatori 51
7.6 Ultime considerazioni 53
Conclusioni 55
Bibliografia 56

Introduzione

Il nome Scala deriva da “Scalable Language” cioè linguaggio scalabile. Scala è un linguaggio capace di adattarsi bene a una varietà di situazioni che oggi riguardano i linguaggi di programmazione. Scala può essere usato come un linguaggio di scripting o come un linguaggio per sviluppare applicazioni complesse.
Per scalabilità si intende la possibilità di creare nuove parti del linguaggio o modificare in parte quelle preesistenti per poter adattare il linguaggio alle proprie necessità. Ad esempio, è possibile costruire nuovi tipi di dato e invocare su di essi operazioni aritmetiche come se fossero dati primitivi già presenti nel linguaggio. Oppure, è possibile creare nuovi costrutti di controllo per poter usare una sintassi più semplice ed è espressiva.
Quello che rende Scala un linguaggio scalabile, è la fusione tra programmazione funzionale e programmazione ad oggetti. Infatti, la programmazione funzionale permette di scrivere in modo semplice e conciso piccole parti di codice, mentre la programmazione ad oggetti permette di strutturare l’intera applicazione.
Per essere più precisi, Scala è un linguaggio puramente ad oggetti e questo significa che tutto dai numeri alle funzioni sono oggetti. Ad esempio, l’espressione ‘*1 + 1*’ in Scala può essere riscritta come ‘*1.+(1)*’.
L’esempio racchiude in se molte particolarità del linguaggio. Prima di tutto, il numero 1 non è come in Java o in C++ un tipo di dato primitivo, ma bensì è un oggetto. Si potrebbe pensare che questa scelta di progettazione abbia delle ripercussioni sull’efficienza dei nostri programmi. Questo però non è corretto, perché il compilatore Scala sostituirà per noi l’oggetto 1 nel corrispondente tipo di dato primitivo. La stessa cosa si può dire per tutti quegli oggetti che rappresentano un tipo di dato primitivo. Inoltre, l’espressione ‘*1 + 1*’ fa uso del metodo ‘+’ della classe *Int* di Scala. Infatti, i metodi di una classe possono anche contenere caratteri come i simboli aritmetici, i simboli di confronto e i doppi punti. Usare tali simboli permette di mantenere semplice e concisa la sintassi. Ad esempio, il metodo *‘+’* può essere usato anche per una struttura dati o un nuovo tipo di dato numerico facendoli sembrare parte del linguaggio nativo.
Un ulteriore osservazione riguarda l’assenza della dot notation per invocare un metodo. Quando un metodo richiede un solo parametro di ingresso, possiamo usare la notazione operatore. La notazione operatore prevede l’assenza del punto tra l’oggetto chiamante e il metodo invocato e l’assenza delle parentesi che circondano l’unico parametro di ingresso. Questo può essere applicato a molte altre situazioni e rappresenta un ulteriore passo di semplificazione della sintassi.
Oltre alla programmazione ad oggetti, Scala supporta la programmazione funzionale. Tale paradigma è fondato principalmente su due idee. La prima idea è quella di considerare le funzioni come entità di prima classe. Questo significa che le funzioni possono essere trattate come dei valori. Come succede per valori di tipo *Int* o *String,* le funzioni di prima classe possono essere assegnate ad una variabile o essere usate come parametri di ingresso o di ritorno da una funzione. Il vantaggio di questo, è poter creare nuovi costrutti di controllo come un ciclo *while*. Un esempio è dato dalla libreria degli Attori nei metodi *loop*, *receive* e *react*. Questi metodi, infatti, sembrano essere stati implementati come linguaggio nativo.
La seconda idea è quella di pensare che una funzione deve mappare un valore di ingresso in un valore di uscita. Detto più semplicemente, una funzione deve comunicare con il mondo esterno solo attraverso i valori di ingresso e uscita. Nella programmazione ad oggetti, questo vuol dire usare oggetti che espongono al loro esterno solo dati immutabili. Infatti, se un oggetto esponesse al mondo esterno un campo mutabile, allora una qualunque funzione invocata su di esso potrebbe usare proprio quel campo per calcolare il suo valore di uscita.

Ci sono diversi motivi per i quali un oggetto immutabile può servire. Ad esempio, si possono usare le *String* di Java come chiavi di ricerca in un *HashMap* senza preoccuparsi che qualcuno possa apportare una modifica ai caratteri della chiave. Oppure, siccome un oggetto immutabile non può essere modificato, si può usare la concorrenza senza sincronizzazione per leggere simultaneamente i dati dell'oggetto. In generale, se le singole parti di codice che costituiscono un applicazione non dipendono da uno stato mutabile in comune, possiamo ragionare sulle singole parti senza dover tener presenti tutte le altre. Un'altra particolarità importante del linguaggio riguarda la tipizzazione. Una delle classificazioni possibili tra i vari linguaggi di programmazione è la tipizzazione, cioè l'assegnamento di un tipo ad una variabile. Esistono principalmente due tecniche differenti per definire il tipo di una variabile. La prima, usata dai più importanti linguaggi di programmazione, si chiama tipizzazione statica e obbliga il programmatore a definire nella dichiarazione di una variabile anche il tipo che questa dovrà assumere.

La seconda, usata ad esempio nei linguaggi di scripting, si chiama tipizzazione dinamica ed è in grado di risalire al tipo di una variabile a tempo di esecuzione. La tipizzazione statica tende a essere più noiosa di quella dinamica, ma permette di eliminare errori subdoli che illudono il programmatore che tutto funzioni. Infatti, se un errore di tipizzazione non viene rivelato direttamente dal compilatore, un applicazione potrebbe funzionare finché una particolare sequenza di istruzioni mandi in errore il programma. In un caso come questo, non resta altro che cercare di correggere l'errore sperando che non ce ne siano altri.

Scala però si dimostra ibrido anche in questo campo perché, pur essendo un linguaggio a tipizzazione statica, usa in molti casi una sintassi da tipizzazione dinamica. Questo perché Scala usa l'inferenza di tipo cioè un meccanismo che permette di capire il tipo di una variabile a tempo di compilazione. Proprio grazie alla tipizzazione statica e ad una sintassi dinamicamente tipata, Scala può essere usato come linguaggio di scripting o come ambiente di sviluppo affidabile per applicazioni più complesse.

Un ultima importante considerazione sul linguaggio, è la sua completa interoperabilità con il linguaggio Java. Infatti, il codice Scala viene compilato in bytecode che può essere interpretato dalla Java Virtual Machine. Questo significa che Scala gode delle stesse performance di Java. Un esempio di interoperabilità dei due linguaggi riguarda l'espressione *'1 + 1'.* Era stato detto, infatti, che un oggetto *Int* viene trasformato nel suo relativo tipo di dato primitivo per non rinunciare alle prestazioni. Il tipo di dato primitivo in questione è il tipo intero del linguaggio Java. La stessa si può dire per tutti gli altri tipi numerici, per i boolean, per le stringhe e per gli array.

1 Concetti Iniziali

Prima di cominciare a conoscere il linguaggio di programmazione Scala, verrà spiegato come installare l'ambiente di sviluppo e come usarlo per eseguire del codice. Per installare Scala è sufficiente andare all'indirizzo http://www.scala-lang.org/downloads e scaricare l'installer per il proprio sistema operativo. Una volta scaricato l'installer, è sufficiente eseguirlo e seguire le istruzioni di installazione. Finita l'installazione è possibile inserire Scala nelle variabili d'ambiente.

Per eseguire codice Scala esistono tre diversi modi che possiamo utilizzare in base alle nostre necessità. Il modo più semplice e veloce è la shell interattiva. Se dobbiamo eseguire piccole porzioni di codice, ad esempio per esercitarci, la shell interattiva risulta essere la scelta migliore. Per usare la shell di Scala è sufficiente eseguire il programma *scala* dal prompt dei comandi. Verrà lanciato l'interprete Scala che eseguirà ogni singola riga di codice.

Un altro modo per eseguire codice Scala, riguarda la modalità di scripting. Se passiamo come argomento al comando *scala* il nome di un file *.scala,* eseguiremo tale codice in modalità di scripting. A differenza del primo, questo sistema è utile quando il nostro codice deve essere salvato.

La terza modalità è quella che più assomiglia all'esecuzione di programmi in codice ad alto livello e cioè possiamo compilare il codice Scala con il comando *scalac,* che fornirà il byte code necessario per la Java Virtual Machine. Sarà sufficiente usare il comando *scala* seguito dal nome del file compilato per eseguire il nostro programma.

Per provare tutte e tre le modalità, scriviamo il classico programma che stampi "Ciao Mondo!". Per le prime due modalità è sufficiente scrivere la seguente riga di codice:

```
println("Ciao Mondo!")
```

Possiamo usare la shell di Scala oppure salvare il codice in un file *.scala* e usare il comando *scala*. Notiamo subito la prima differenza con altri linguaggi di programmazione come Java: l'assenza del punto e virgola. Scala rende opzionale l'uso del punto e virgola alla fine di ogni istruzione. Comunque è sempre obbligatorio nel caso si scrivano due o più istruzioni sulla stessa riga. Per la terza modalità dobbiamo usare qualche riga di codice in più:

```
class CiaoMondo {
    def main(args: Array[String]) {
        println("Ciao mondo!")
    }
}
```

Questo codice va salvato in un file *.scala* ,compilato con il comando scalac e poi eseguito con il comando *scala* seguito dal nome del file compilato.

Un'ulteriore differenza tra un programma scritto in Scala e un programma scritto in Java, è il nome del file dentro il quale si deve trovare una classe. In Java il nome della classe e del file devono coincidere, mentre in Scala questo non è obbligatorio.

1.1 Variabili

In Scala esistono due tipi di variabili, mutabili e immutabili. Una variabile immutabile è simile alle variabili **final** di Java e possono essere assegnate una sola volta in fase di definizione, mentre le variabili mutabili possono essere riassegnate senza alcun limite. La sintassi per dichiarare una variabile in Scala è :

```
var/val NomeVariabile : TipoVariabile = ValoreVariabile
```

La parola chiave **val** permette di definire una variabile immutabile, la parola chiave **var** permette di definire una variabile mutabile.

```
var mes: String = "1"
val i: Int = 1
```

Alla variabile mes, a questo punto, può essere assegnata un'altra stringa, mentre alla variabile i rimarrà associata al valore 1 fino al suo rilascio. In realtà, esiste un'eccezione che riguarda la sovrascrittura dei membri di una classe ereditata, che sarà spiegata nel capitolo riguardante le classi.
Come già detto nell'introduzione, è anche possibile non specificare il tipo della variabile perché esiste l'inferenza dei tipi. Le stesse dichiarazioni di prima possono essere riscritte più agevolmente come:

```
var mes = "1"
val i = 1
```

A meno che non si stia definendo un campo astratto, è sempre obbligatorio assegnare un valore iniziale ad una variabile.

1.2 Costrutto if

In Scala il costrutto **if** svolge la stessa funzione che ha in qualsiasi altro linguaggio di programmazione. Questo costrutto prevede la valutazione di un espressione booleana e, in base al valore assunto, l'esecuzione del blocco di codice corrispondente. Come in tutti i linguaggi di programmazione si possono scrivere **if** annidati.
Quello che in più offre Scala al costrutto **if,** è di considerare il suo corpo come un'espressione, ovvero l'**if** può ritornare un valore. Prendiamo come esempio una divisione:

```
val num = 10
val den = 0
val divisione = if (den != 0) num/den
                else "Errore"
println(divisione)
```

In questo esempio assegniamo alla variabile divisione il risultato dell'**if**. Qui si vede come il valore ritornato può assumere due tipi diversi, ma nonostante questo non ci sono errori di tipo. L'inferenza di tipo prenderà il primo tipo in comune tra il tipo *String* e il tipo *Int*. Il tipo in questione è *Any* cioè la radice della gerarchia dei tipi di Scala.

1.3 Costrutto for

Il costrutto **for** permette di ripetere ciclicamente un insieme di operazioni specificando un indice e il numero di iterazioni da un valore dell'indice iniziale a uno finale. La sintassi del ciclo **for** è diversa da Java e, come per le variabili, ricorda molto di più la sintassi del Pascal.

```
for (i <- 1 to 10) {
  println(i)
}
```

In Scala, però, questo costrutto è stato potenziato rendendolo più flessibile. Proviamo ad analizzare il codice scritto tra parentesi tonde. Di solito quando si scrive un ciclo **for** in un qualsiasi linguaggio di programmazione la sintassi complessiva è statica e ben definita. In Scala invece questo non è vero. La parola *to* non è una parola chiave del linguaggio. *X to Y* in realtà si può scrivere anche come *X.to(Y).* Il metodo *to* appartiene alla classe *RichInt* e permette di restituire una lista di interi cha vanno da *X* a *Y*. Esiste anche la possibilità di specificare un qualsiasi tasso di incremento con un secondo parametro. In ogni caso, la lista di interi può essere iterata dal ciclo **for**. Questo significa che il ciclo **for** non scandisce solo una lista di interi ma qualsiasi tipo di lista:

```
val arr = new Array[Int](10)
for (a <- arr) {
  println(a)
}
```

In questo caso la nostra lista è un array. In questo modo, non dobbiamo dichiarare la variabile incrementale i, non dobbiamo controllare quando il ciclo deve terminare e non sorge quel fastidioso problema di non sapere se partire da 0 o da 1. Se facciamo un confronto con Java, ci accorgiamo subito della semplicità e dell'espressività della sintassi di Scala:

```
int[] arr = new int[10];
for (int i=0;i<10;i++){
     System.out.println(arr[i]);
}
```

Come per l'istruzione **if**, anche qui è possibile assegnare il risultato di espressione **for** a una variabile. Il for itera una lista e quindi quello che restituisce sarà un'altra lista. Questo viene fatto con la parola chiave **yield**:

```
val A = List("a","b")
val B = for (l <- lista) yield l
```

La lista *A* viene riempita con gli elementi della lista *B*. L'inferenza di tipo assegna alla lista *A* il tipo *String* e conseguentemente anche alla lista *B*. Altre due operazioni che si possono eseguire direttamente nella dichiarazione del ciclo **for** sono il filtraggio e i cicli annidati. Tutte e due possono essere specificate direttamente all'interno delle parentesi tonde:

```
val lista = List(1,2,3,4)
for (l <- lista if l%2==0) {
  println(l)
}
```

```
for (i<-1 to 10; j<-1 to 10) {
  println(i*j)
}
```

Nel primo esempio viene iterata una lista di quattro elementi di cui solo gli elementi pari vengono stampati. Il secondo esempio illustra, invece, la possibilità di comporre più cicli annidati in uno unico e stampa la moltiplicazione dei due indici che percorrono le due liste. La variabile posta più a sinistra, in questo caso la variabile j, scorre più velocemente.

Tutte e due gli esempi, in realtà, possono essere riscritti come in Java, cioè mettendo un **if** nel corpo del ciclo oppure scrivendo un ciclo nidificato. Quello che questi esempi mostrano, è la sintassi completa del ciclo **for**. Invece di specificare un valore iniziale e un valore finale come in Java e molti altri linguaggi, in Scala un ciclo **for** può essere riassunto come segue:

```
for ( seq ) yield expr
```

Il codice identificato con seq si può dividere in una sequenza di tre parti fondamentali cioè *generatori , definizioni e filtri.* I generatori indicano l'estrazione da una lista di elementi di qualsiasi tipo che vengono poi inseriti in una variabile. Quando definiamo più generatori all'interno di un **for**, il generatore posto per ultimo varia più velocemente degli altri. Le definizioni indicano una qualsiasi espressione del tipo *x = expr,* mentre i filtri indicano una condizione specificata con il costrutto **if**. Ecco un esempio completo di ciclo **for**:

```
val lista1 = List(1,2,3)
val lista2 = List(1,2,3)

println(
    for ( i <- lista1 ;
          j <- lista2 ;
          somma = i + j ;
          if somma % 2 == 0
         ) yield (i,j)
)
```

Nel ciclo **for** sono presenti due generatori, una definizione e un filtro. I due generatori estraggono gli elementi dalle due liste lista1 e lista2. La definizione esegue la somma degli elementi estratti dalle due lista, mentre il filtro scarta i casi in cui la somma sia dispari. Il risultato del ciclo è la seguente lista: `List((1,1), (1,3), (2,2), (3,1), (3,3)).`

1.4 Costrutto match

L'istruzione **match** è l'equivalente dell'espressione **switch** di Java. Questa istruzione, però, presenta dei miglioramenti rispetto ai suoi predecessori. La cosa più importante riguarda i tipi di dato che non devono essere necessariamente solo interi ma possono essere qualsiasi:

```
val s: Any = "a"
s match {
    case "a" => println("String")
    case 1   => println("Int")
    case _   => println("Caso base")
}
```

In questo esempio sono stati usati i tipi *String* e *Int*. Il simbolo '_' è un segnaposto usato per indicare il caso base nell'eventualità che nessun punto precedente soddisfi il confronto con la variabile *s*. E' sempre buona norma inserire il caso base, perché se dovesse presentarsi un caso non gestito, il programma terminerebbe in errore. Un'altra differenza che si può notare, è l'assenza dei **break** tra i vari casi. Usare dei **break** tra i vari casi è superfluo oltre che poco elegante.
Come i due costrutti **if** e **for** sopra presentati, anche l'istruzione **match** ritorna un valore. Il precedente esempio può essere riscritto nel seguente modo:

```
val s: Any = "a"
val res = s match {
     case "a" => "String"
     case 1   => "Int"
     case _   => "caso base"
}
println(res)
```

Invece di stampare direttamente il risultato nel corpo di un case, il valore di ritorno viene prima salvato nella variabile *res* e poi stampato. Quando verrà spiegato il pattern matching, sarà spiegato in maggior dettaglio tutto quello che si può fare con il costrutto match.

1.5 Ulteriori costrutti

Oltre ai costrutti presentati, sono presenti anche i cicli **while** e **do while** che però non differiscono in alcun modo da quelle già presenti in Java. La cattura delle eccezioni viene fatta attraverso l'istruzione **try catch finally**, mentre per lanciare una nuova eccezione si usa sempre la parola chiave **throw**. Tutto questo è identico a quello che Java già dispone, ma esistono delle differenze sostanziali che è bene sapere. Anche il costrutto **throw** può ritornare un valore e l'annotazione **@throws** non è obbligatorio specificarla. Un semplice esempio è il seguente:

```
val num = 10
val den = 0
val div =
  if (den != 0)
     num / den
  else
     throw new RuntimeException("Divisione per zero")
```

Questo esempio serve a notare una cosa molto utile: il tipo della variabile *div* non è *Any* ma *Int*. Questo perché il tipo delle eccezioni è *Nothing* e cioè il tipo più in basso della gerarchia del linguaggio, quindi sottotipo anche di *Int*. In questo modo, il tipo ritornato dal costrutto **if** è il tipo effettivamente di nostro interesse.
I costrutti **if**, **for**, **match** e **throw** ritornano un valore che può essere assegnato a una variabile immutabile. Questo viene fatto perché Scala dà la possibilità di sviluppare le proprie applicazioni con uno stile imperativo oppure funzionale. L'idea di usare delle variabili immutabili fa parte dello stile funzionale e permette di scrivere codice più conciso e meno suscettibile agli errori. Infatti, usare delle variabili immutabili permette la creazione di uno stato globale condiviso non soggetto a modifiche da parte di porzioni di codice scritte altrove. In generale, tramite le variabili immutabili, possiamo creare oggetti

immutabili che, non esponendo nessuna variabile mutabile, possono essere usati in modo più sicuro in tutto il codice. Vedremo come questo possa essere vantaggioso con la libreria degli attori.

2 Classi

2.1 Definire una classe

In Scala una classe si definisce con la parola chiave **class** seguita dal nome della classe. Il costruttore della classe non è definito in un metodo particolare ma è l'intero corpo della classe. All'interno del corpo della classe, ci saranno i classici membri come i campi e i metodi, ma potrà essere presente anche del codice perché il corpo della classe è appunto il costruttore. I parametri in ingresso sono scritti tra parentesi affianco al nome della classe come se fosse la firma di un metodo. Vediamo un esempio:

```
class C(i: Int) {
  println(i)
}
```

Il costruttore della classe C accetta come parametro di ingresso un intero e lo stampa. Per istanziare un oggetto di una classe, si usa la parola chiave **new** come in Java:

```
val a = new C(1)
```

Questa riga di codice crea un istanza della classe C e stampa il numero 1 a schermo. Una cosa importante da notare, è che la variabile intera i non è visibile al di fuori della classe C. Questo perché è come se fosse definita nella firma di un metodo.
In Scala esiste l'ereditarietà singola anche se esiste un costrutto chiamato 'tratto' che fornisce un supporto simile alla multiereditarietà. Una classe può estendere un'altra classe con la parola chiave **extends**:

```
class A(i: Int){
  println(i)
}

class B(i: Int) extends A(i){}
```

La differenza da Java sta nel passaggio di parametri alla superclasse. L'esempio mostra come viene fatto il passaggio dei parametri al costruttore della superclasse. Invece di usare la parola chiave **super** , i parametri sono scritti direttamente affianco al nome della superclasse.

2.2 Costruttori ausiliari

Il costruttore principale di una classe in Scala è il corpo principale della classe. Se dobbiamo definire dei costruttori differenti da quello principale scriviamo **def this**(…) definendo i parametri iniziali da passare. Questi costruttori si chiamano costruttori ausiliari. La prima istruzione di un costruttore ausiliario è **this**(…) cioè la chiamata a un altro costruttore. Le chiamate a **this** degli altri costruttori porteranno alla chiamata del costruttore principale.

```
class C(i: Int) {
```

```
    println(i)
    def this(i: Int,j: Int) {
      this(i)
      println(i+j)
    }
  }
```

In questo esempio il costruttore ausiliario prima di stampare la somma delle due variabili i e j chiama il costruttore principale che stampa la variabile i.
Una differenza dal linguaggio Java è la chiamata al costruttore delle superclassi. Mentre in Java qualsiasi costruttore può chiamare il costruttore della superclasse , in Scala questo viene fatto solo dal costruttore principale. Questa limitazione è dovuta al fatto di avere una sintassi più elegante nel passaggio dei parametri al costruttore della superclasse.

2.3 Campi parametrici

Supponiamo che nella nostra classe sia dichiarato un campo che dovrà essere inizializzato dal costruttore. Quando noi passiamo il valore per quel campo al costruttore dobbiamo avere nella firma del costruttore una variabile che non può essere vista dall'esterno della classe. Prendiamo un esempio:

```
class Persona(n: String, c: String) {
  var nome = n
  var cognome = c
}
```

Oltre a essere una ripetizione inutile a volte è molto scomodo dover trovare dei nomi differenti per gli stessi valori. Per evitare questa situazione è possibile dichiarare un campo parametrico:

```
class Persona(var nome: String , var cognome: String)
```

Quello che cambia è la parola chiave **val** o **var** prima dei parametri di ingresso. In questo modo abbiamo un membro che è sia un parametro e sia un campo. Come per ogni campo è possibile usare i modificatori di accesso come **Private** o **Protected** oppure sovrascrivere o rendere astratto il campo parametrico.

2.4 Sovrascrivere campi e metodi

Nel meccanismo dell'ereditarietà , una sottoclasse eredita campi e metodi dalla sua superclasse. In alcune situazioni però è utile modificare il comportamento dei campi immutabili e dei metodi per permettere una maggiore specializzazione. Questo può essere fatto come in altri linguaggi con la parola chiave **override**:

```
class A{
  def fA = println("metodo di A")
  val i = 1
}

class B extends A{
  override def fA = println("metodo di B")
  override val i = 2
```

```
}
```

Quando dobbiamo ridefinire un membro astratto non è necessario usare la parola **override**:

```
abstract class A{
  def fA: Unit
  val i: Int
}

class B extends A{
  def fA = println("metodo di B")
  val i = 2
}
```

In Scala è addirittura possibile sovrascrivere un metodo con una variabile:

```
abstract class A{
  def fA: Unit
}

class B extends A{
  val fA = println("metodo di B")
}
```

Ovviamente affinché questo sia possibile il metodo *fA* non deve avere parametri in ingresso e parentesi vuote. Ora ci si può chiedere se si possa fare anche il contrario cioè ridefinire un campo immutabile con una funzione. Questo non si può fare perché modificherebbe il comportamento della classe astratta.
Infatti se una classe astratta definisce un campo immutabile e noi lo sovrascriviamo con una funzione allora stiamo sostituendo un membro immutabile con un membro mutabile. Invece, quando sostituiamo un membro mutabile con un membro immutabile, non stiamo modificando il comportamento della classe perché il membro mutabile potrebbe anche non cambiare mai.
Nonostante questo non è però concesso sovrascrivere un campo mutabile con un campo immutabile.
Infatti un campo mutabile è associato anche al metodo set non sovrascrivibile per un campo immutabile.
Supponiamo di non voler permettere a nessuna sottoclasse di sovrascrivere un metodo o un campo. Per fare questo in Scala è possibile usare il modificatore **final**. Oltre a impedire la sovrascrittura di metodi e campi la parola chiave **final** può essere applicata anche ad una classe impedendole così di essere ereditata.

```
final class A{
  def fA: Unit
}

class B extends A{
  val fA = println("metodo di B")
}
```

Se si prova a eseguire questo codice nella shell dell'interprete verrà segnalato un errore di ereditarietà.

2.5 Visibilita'

Come in molti altri linguaggi di programmazione ad oggetti , anche in Scala sono presenti i modificatori di visibilità per membri e tipi. Il loro ruolo è quello di impostare un ambito di visibilità che può essere principalmente di tre tipi:

- **Public**: Il modificatore di accesso **public** è applicato di default a tutti i membri o i tipi che non specificano nessuna visibilità. Come risulterà ovvio i membri o i tipi pubblici sono visibili ovunque. La parola chiave **public**, oltre a non essere necessaria, non è nemmeno disponibile tra le parole chiave del linguaggio.
- **Protected**: Per il modificatore di accesso **protected** bisogna distinguere tra membri e tipi. I membri protetti sono visibili nel tipo di appartenenza, nei tipi derivati e nei tipi annidati. I tipi protetti sono visibili al package di appartenenza e nei sottopackage. I tipi annidati protetti seguono la stessa regola dei membri protetti.
- **Private**: Anche per il modificatore di accesso **private** bisogna distinguere tra membri e tipi. I membri privati sono visibili nei tipi di appartenenza e nei tipi annidati. I tipi privati sono visibili solo al package di appartenenza. I tipi annidati privati seguono la stessa regola dei membri privati.

Inoltre, in Scala, esistono altri due modificatori di accesso: Protetto ristretto e privato ristretto.
La differenza dai modificatori precedenti, è la possibilità di impostare la visibilità di un membro o di un tipo a un package, un tipo, o **this**. La sintassi usata è:

```
protected[scope]
private[scope]
```

Questi due nuovi tipi di modificatori sono una generalizzazione del concetto di visibilità. Se si vuole impostare uno scope come un package o un tipo, è necessario che i membri o i tipi, a cui verrà etichettato lo scope, siano annidati nel package o nel tipo scelto come scope.
In generale, impostare uno scope come un package o un tipo, significa rendere un membro o un tipo protetti o privati fino al package o tipo in questione. Se lo scope è **this** otteniamo la visibilità in assoluto più restrittiva. Infatti in questo caso le istanze di un tipo non possono vedere i membri dello stesso tipo ma solo i propri membri. Ecco un esempio dei tre casi che possono riguardare lo scope:

```
package a{
    class classA {
      class subClassA{
          private[a] var i = 1
          protected[subClassA] var j = 1
          private[this] var k = 1
          def set(obj: subClassA) = {
            obj.j = 2
            obj.k = 2 //errore
          }
      }
    }
    class classB {
      val objA = new classA
      val objSubA = new objA.subClassA
      objSubA.i = 2
      objSubA.j = 2  //errore
    }
}
```

In *subClassA* sono presenti tre campi e un metodo. La variabile *i* è privata al package *a*, la variabile *j* è protetta a *subClassA*, la variabile *k* è privata a **this**. In *classB* sono presenti due assegnazioni alle variabili *i* e *j.* Delle due solo la prima è valida. Infatti la variabile *i* è privata a tutto l'ambito di visibilità del package *a*, mentre la variabile *j* è visibile solo all'interno di *subClassA.*

Per quanto riguarda lo scope **this**, possiamo vedere che, cercando di accedere all'intero di *subClassA* dal metodo *set* al campo *k* di un altro oggetto, il compilatore segnala un errore. La stessa cosa non è vera quando la visibilità è solo protetta o privata. Infatti, sempre nel metodo *set* di *subClassA*, la variabile j di un altro oggetto può essere assegnata senza problemi.

2.6 Oggetti singleton

Siccome in Scala non esistono gli static**,** per avere uno strumento equivalente si usano i singleton. I singleton sono oggetti istanziati una sola volta da Scala. Non essendo possibile istanziarli su richiesta, non hanno bisogno di nessun costruttore.
Quando dichiariamo un metodo all'interno di un oggetto singleton, possiamo considerare quel metodo come uno static. Infatti, se l'oggetto è istanziato una sola volta, possiamo invocare quel metodo solo su quell'oggetto. Ma perché non lasciare gli static? Perché in Scala ogni cosa è un oggetto e gli static non lo sono. Un singleton si dichiara come una classe sostituendo la parola chiave **class** con **object**.

```
object oggettoSingle {
  def foo = println(1)
}
```

Dal main possiamo richiamare il metodo *foo* come se fosse un metodo statico di una classe Java.

```
oggettoSingle.foo
```

In realtà, questo comportamento non rispecchia appieno il funzionamento dei metodi static di Java perché un metodo dichiarato all'interno di un singleton può essere considerato solo come un metodo static. Per questo motivo una classe e un oggetto singleton possono essere associati. Una classe e un oggetto singleton si dicono associati quando hanno lo stesso nome, sono dichiarati nello stesso file e nello stesso package. Ecco un esempio di classi e oggetti singleton associati:

```
object Associati {
  var i = 1
}

class Associati {
  var j = 1
}

val assoc = new Associati
assoc.j = 2
Associati.i = 2
```

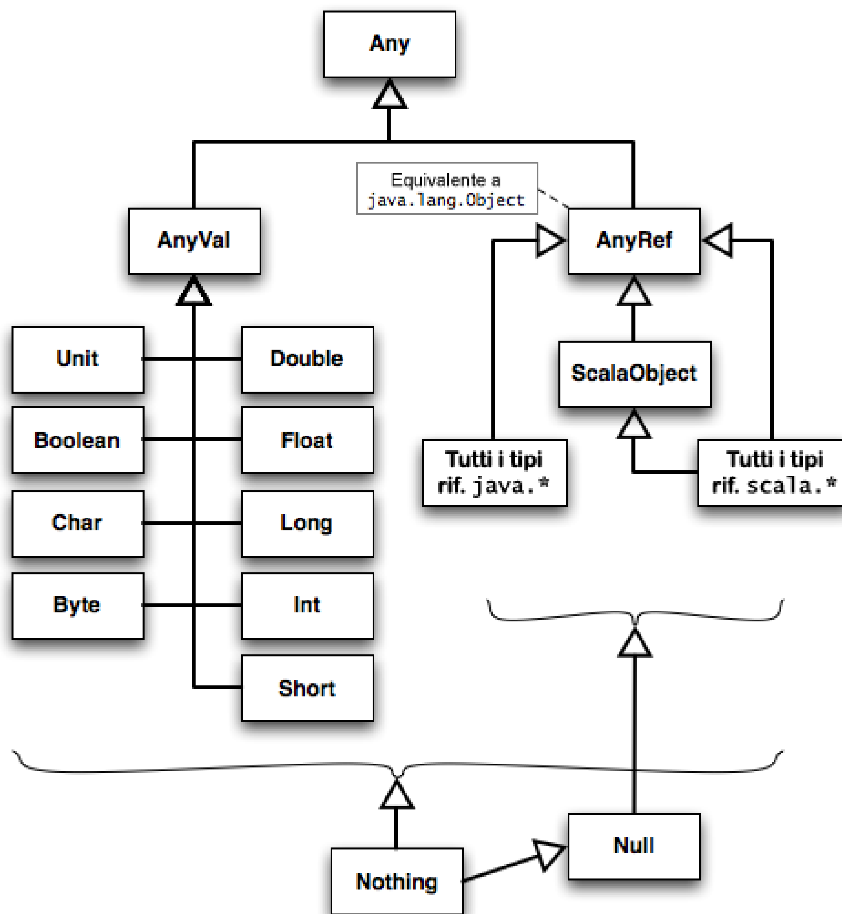
E' importante osservare l'equivalenza tra gli oggetti singleton e gli static di Java. Infatti, quando in Java ci riferiamo a un membro statico, usiamo sempre il nome della classe e non di una particolare istanza. Grazie all'associazione tra una classe e un oggetto singleton in Scala possiamo ottenere lo stesso effetto.
Un singleton object può essere dichiarato senza la relativa classe associata e questo può servire in varie situazioni come contenitore di metodi static oppure come entry point per una applicazione Scala.
Un tale oggetto si chiama standalone object.

2.7 Gerarchia dei tipi

In Java la gerarchia delle classi inizia da *Object*. In *Object* sono presenti vari metodi che vengono quindi ereditati da tutte le classi del linguaggio Java. In Scala la radice della gerarchia è rappresentata da *Any*. Sebbene questo faccia pensare che *Any* sia l'analogo di *Object*, questo non è vero. La classe *Any* si divide in due sottoclassi: *AnyVal* e *AnyRef*.

La classe *AnyVal* è la superclasse per tutti i tipi valore come *Int* o *Char* comprendendo anche *Unit,* l'analogo di *void* in Java. La classe *AnyRef* è la superclasse per tutti gli altri tipi, compresi quelli scritti dallo sviluppatore e quelli importati da Java. Quindi, la classe analoga alla classe *Object* di Java è *AnyRef*.

Oltre a *AnyRef* le nostre classi ereditano il tratto *ScalaObject*. Il tratto *ScalaObject* contiene dei metodi che sono implementati dal compilatore Scala per rendere l'esecuzione del codice più veloce. In fondo alla gerarchia esistono i tratti *Null* e *Nothing*. Il tratto *Null* è un sottotipo di qualsiasi tipo *AnyRef,* mentre il tratto *Nothing* è un sottotipo di qualsiasi altro tipo definito nel linguaggio.



2.8 Una prima applicazione

Per creare un entry point di qualsiasi applicazione Scala, si possono usare vari metodi. Si può creare una classe come in Java oppure , come appena citato, si può usare un singleton con un metodo main.

```
object objSingleton{
  def main(args: Array[String]) = {
    println("ciao mondo")
  }
}
```

L'applicazione non farà altro che stampare sullo schermo la scritta ciao mondo. Come in Java, il metodo main accetta come parametri in ingresso delle stringhe, mentre in uscita restituisce l'equivalente di void e cioè il tipo *Unit*. Il tipo restituito è inferito e quindi non occorre specificarlo.
Nel capitolo dedicato ai tratti verrà spiegato un metodo simile, ma che ci farà risparmiare qualche carattere per lanciare una nuova applicazione Scala.

3 Tratti

Nei linguaggi che permettono la multiereditarietà come il C++ è possibile che una classe possieda più superclassi , permettendo quindi di ottenere classi con più comportamenti e favorendo un maggior riutilizzo del codice. Senza dubbio questo è molto utile ma l'uso della multiereditarietà è stato aspramente criticato. Molti ritengono che la multiereditarietà porti con se più svantaggi che vantaggi perché alla fine il codice risulta essere molto più complicato e difficile da correggere rispetto a programmi che non ne fanno uso. Per questo motivo in Java e altri linguaggi esiste solo l'ereditarietà singola. Per cercare di ottenere un meccanismo simile alla multiereditarietà in Java esistono le interfacce cioè un insieme di metodi astratti che dovranno essere implementati nelle classi che le estendono. Non avendo variabili o metodi concreti anche se due interfacce avessero due metodi con stessa firma coincidenti, non succederebbe nulla di male perché, appunto, questi metodi sono vuoti. Purtroppo il sistema delle interfacce non favorisce il riutilizzo del codice perché non si possono ereditare dei metodi concreti da più classi. Per questo motivo, in Scala e altri linguaggi, sono presenti i tratti.

3.1 Lavorare con i tratti

Proviamo a fare qualche esempio per capire cosa sono e come funzionano i tratti. I tratti sono simili alle interfacce di Java ma con la differenza di poter contenere campi e metodi concreti. Un esempio di tratto è il seguente:

```
trait haFoglie {
  println("ho le foglie")
  var colore: String = "verdi"
  def getColore = colore
}
```

La dichiarazione di un tratto è simile alla dichiarazione di una classe con la differenza di usare la parola chiave **trait**. Un tratto può essere ereditato da una classe o un altro tratto. Questo viene fatto con la parola chiave **extends**:

```
class albero extends haFoglie{
  println("sono un albero")
}
```

Per ereditare da più tratti si usa la parola chiave **with**:

```
trait haRadici {
  println("ho le radici")
}

class albero extends haFoglie with haRadici{
  println("sono un albero")
}
```

Quando estendiamo una classe con l'ereditarietà singola abbiamo la possibilità di sovrascrivere qualsiasi metodo della superclasse. Quando estendiamo un tratto possiamo fare la stessa cosa con la parola chiave **override**:

```
trait haFoglie {
  println("ho le foglie")
  var colore: String = "verdi"
  def getColore = colore
  def cresco = println("cresco")
}

class albero extends haFoglie with haRadici{
  println("sono un albero")
  override def cresco = println("divento più alto")
}
```

Un'osservazione importante riguarda la differenza tra i tratti e le classi astratte. I tratti, sebbene abbiamo un costruttore principale, che non è altro che il corpo del tratto stesso, non possono avere dei parametri di ingresso. Inoltre, grazie all'ereditarietà singola, quando invochiamo un metodo di una superclasse non esiste nessun problema di ambiguità. Quando invece misceliamo più tratti sulla stessa classe, per evitare i problemi creati dalla multieriditarietà, viene usato un meccanismo calcolato dinamicamente chiamato linearizzazione.

3.2 Riutilizzo del codice

L'obiettivo principale dei tratti è favorire il riutilizzo del codice. Ci sono due casi principali nei quali i tratti possono essere usati per questo scopo: O come mixin o come modifiche impilabili.

3.2.1 Mixin

Ogni volta che un'interfaccia viene implementata, i programmatori devono scrivere il corpo dei metodi dell'interfaccia. La situazione che si viene a creare non è però delle migliori.
Se infatti esistono più classi che implementano metodi simili tra di loro, allora si sta sprecando tempo a scrivere codice che potrebbe essere scritto una sola volta. La stessa cosa vale per metodi interni ad una classe poco coerenti con il significato della classe e magari riusabili in altre classi.
I mixin sono la soluzione a questo problema. I mixin non sono altro che delle parti di codice specializzato e che possono essere riusate in modo indipendenti da altri classi.
Un'applicazione di questo concetto può essere il tratto *Ordered*. Questo tratto contiene al suo interno dei metodi di confronto già implementati e potrebbe risparmiarci il lavoro di doverli riscrivere. Si potrebbe anche pensare di usare una classe *Ordered*, ma questo non si può fare se dobbiamo già estendere un'altra classe. Ecco come usare il tratto *Ordered*:

```
class Persona(var cognome: String) extends Ordered[Persona] {
  def compare(that: Persona): Int = {
    if (this.cognome > that.cognome) 1
    else if (this.cognome < that.cognome) -1
    else 0
  }
}

class Merce(var prezzo: Int) extends Ordered[Merce] {
  def compare(that: Merce) = {
    this.prezzo - that.prezzo
  }
}
```

Nell'esempio vengono dichiarate due classi differenti ma che richiedono entrambe dei metodi di confronto. In realtà i metodi per confrontare due elementi di uno stesso insieme possono essere tutti implementati in funzione del metodo *compare*. Per questo motivo, per usare usare il tratto *Ordered* è necessario fornire l'implementazione di questo metodo.
Oltre a questo è chiaramente necessario usare anche un tipo parametrico perché, i metodi di confronto del tratto *Ordered,* non possono sapere di quale tipo di ingresso sarà l'oggetto da confrontare.

3.2.2 Tratti impilabili

Come già detto quando ereditiamo i metodi di due interfacce dobbiamo in ogni caso implementare questi metodi. Siccome in Scala questo può essere fatto grazie ai tratti possiamo risparmiare tempo prezioso sprecato a reimplementare sempre lo stesso codice.

```
abstract class Persona {
  def getOrario
}

trait Studente extends Persona{
  override def getOrario = println("orario studente")
  def impara = {}
}

trait Lavoratore extends Persona{
  override def getOrario = println("orario lavoratore")
  def lavora = {}
}

class StudenteLavoratore() extends Studente with Lavoratore{}
```

Questo è il classico esempio dello studente lavoratore, utile per capire il problema del diamante. Infatti si può notare che sia Studente e sia Lavoratore hanno in comune il metodo *getOrario*. Quale dei due deve essere chiamato quando lo invochiamo su un istanza della classe *StudenteLavoratore*?
In C++ le soluzioni non mancano, mentre in Scala vedremo fra poco come la linearizzazione risolve questo problema.
Quello che ora è importante notare è che i metodi impara e lavora non sono stati riscritti. Con le interfacce al massimo potevamo indicare che *StudenteLavoratore* aveva il comportamento di Studente e il comportamento di Lavoratore ma poi avremmo dovuto riscrivere i metodi impara e lavora.
Come vedremo fra poco anche l'ordine di con cui si scrivono i tratti è importante. Se invertiamo Studente e Lavoratore:

```
class StudenteLavoratore() extends Lavoratore with Studente {}
```

Quello che otteniamo è un comportamento diverso. Questo permette di ottenere maggiore flessibilità ed è un ulteriore punto di forza dei tratti.

3.3 Linearizzazione

Nel precedente esempio, i tratti Studente e Lavoratore vengono ordinati in una struttura lineare. Questo viene fatto anche coi i genitori dei tratti e serve, appunto, per risolvere i casi di conflitto come il metodo

getOrario o per invocare tutti i costruttori delle superclassi fino alla radice.
Prima di comprendere esattamente come viene costruita la linearizzazione diciamo che il primo tratto che viene considerato è quello scritto più a destra. Se poi è presente una chiamata super allora viene invocato il metodo del tratto scritto subito a sinistra. Se anche questo metodo chiama super si procede a sinistra fino alla fine. Se anche l'ultimo tratto chiama super allora si passa al genitore.
Nell'esempio precedente quindi se invochiamo *getOrario* quello che viene stampato è l'orario di *Lavoratore* essendo il tratto scritto più a destra. Se invece modifichiamo il nostro esempio così:

```
class Persona {
  def getOrario = println("orario persona")
}

trait Studente extends Persona{
  override def getOrario = {
    println("orario studente")
    super.getOrario
  }
  def impara = {}
}

trait Lavoratore extends Persona{
  override def getOrario = {
    println("orario lavoratore")
    super.getOrario
  }
  def lavora = {}
}

class StudenteLavoratore() extends Studente with Lavoratore{}
```

Allora quello che otteniamo a schermo è:

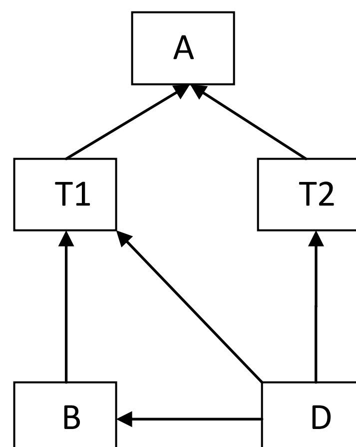
```
orario lavoratore
orario studente
orario persona
```

In realtà il processo di linearizzazione completo è diverso. Prima di tutto se *StudenteLavoratore* implementasse il metodo *getOrario* sarebbe il primo a essere chiamato. Poi alla fine della linearizzazione dovrebbero essere inseriti i tipi *ScalaObject*, *AnyRef*, *Any*.
In ulteriore analisi il processo descritto precedentemente non è completo. Per vedere come funziona realmente prendiamo il seguente esempio:

```
class A {
  def f = println("f di A")
}

trait T1 extends A {
  override def f = {
    println("f di T1")
    super.f
  }
}

trait T2 extends A {
  override def f = {
     println("f di T2")
```



```
      super.f
    }
  }

  class B extends T1 {
    override def f = {
      println("f di B")
      super.f
    }
  }

  class D extends B with T1 with T2 {
    override def f = {
      println("f di D")
      super.f
    }
  }
```

La regola di prima qui non funzionerebbe. In realtà prima di passare al tratto di sinistra bisogna risalire la gerarchia fino alla radice. Fatto questo per tutti i tratti bisogna eliminare tutti i genitori duplicati lasciando l'ultimo più a destra. Con un esempio che illustra il funzionamento passo passo si capisce meglio:

D
D – T2
D – T2 – A
D – T2 – A – T1
D – T2 – A – T1 – A
D – T2 – A – T1 – A – B
D – T2 – A – T1 – A – B – T1
D – T2 – A – T1 – A – B – T1 – A

Ora bisogna cancellare i duplicati lasciando l'ultimo a destra:

D – T2 – B – T1 – A

E per finire si inseriscono i tre tipi *ScalaObject* , *AnyRef* e *Any*:

D – T2 – B – T1 – A – ScalaObject – AnyRef – Any

Come si può vedere *B* risulta prima di *T1* a differenza della regola precedente.

3.4 Differenza dalla multiereditarieta'

I tratti non possono essere considerati come la multiereditarietà. Anche se si possono riutilizzare parti di codice come nell'esempio studente lavoratore, il loro funzionamento differisce dalla multiereditarietà. La differenza sostanziale sta nella linearizzazione. Infatti questa determina una precedenza sulle classi, mentre nella multiereditarietà questo non avviene.
Inoltre, i tratti hanno delle limitazioni che impediscono a una classe di ereditare da più classi:

```
class A
class B
trait T1 extends A
trait T2 extends B
class C extends T1 with T2
```

<console>:11: error: illegal inheritance; superclass A is not a subclass of the superclass B of the mixin trait T2
class C extends T1 with T2

Nell'esempio si è cercato di ricreare il meccanismo della multiereditarietà, estendendo la classe *C* con le classi *A* e *B*.
Un'altra differenza si vede nel seguente esempio dove anche i campi vengono ordinati secondo il procedimento della linearizzazione:

```
abstract class B {
  def v: Int
}

trait T1 extends B{
  override def v = 10
}

trait T2 extends B{
  override def v = 20
}

class C1 extends T1 with T2
```

Adesso quale deve essere il valore di v nelle istanze di C1? La risposta è 20 perché il tratto T2 precede il tratto T1.

3.5 Una prima applicazione con i tratti

Nel capitolo riguardante le classi, era stato dato un esempio di come eseguire una nuova applicazione. L'esempio usava un object con al suo interno un metodo main. Esiste però un'altra possibilità e cioè il tratto *Application*. Se estendiamo il tratto *Application* con la nostra classe o il nostro object, non dobbiamo dichiarare un metodo main come entry point per il nostro programma perché questo viene già fatto in automatico dal tratto *Application*. Il metodo main viene quindi ereditato dalla nostra applicazione:

```
object HelloWorld extends Application {
     println("ciao mondo")
}
```

Il nostro codice può quindi essere inserito direttamente nel costruttore della classe o dell'oggetto singleton. Questo metodo è, però, sconsigliato per vari motivi. Prima di tutto, gli argomenti passati da linea di comando non sono disponibili nel nostro codice, perché il corpo del costruttore della nostra applicazione viene eseguito prima dell'invocazione del metodo main. Poi, non è possibile usare un tale meccanismo per il multithreading e potrebbe non essere disponibile l'ottimizzazione da parte della JVM.

4 Funzioni

4.1 Definire una funzione

Le funzioni sono definite da una firma e da un corpo. La firma è costituita da un nome che serve per invocare la funzione nel codice e da alcuni parametri di ingresso e di uscita che servono per calcolare e ritornare un risultato. Il corpo è costituito da un blocco di istruzioni che verranno eseguite quando la funzione è invocata. Per definire una funzione è necessario usare la parola chiave **def** prima della firma. Un esempio di funzione in Scala è il seguente:

```
def somma(i: Int, j: Int): Int = {
    return i+j
}
```

Il nome della funzione è *somma* , *i* e *j* sono i suoi due parametri di ingresso e il risultato è di tipo *Int*.
Ci sono degli accorgimenti che permettono di risparmiare qualche carattere. La precedente funzione può essere riscritta omettendo le parentesi graffe, la parola chiave **return** e il tipi di ritorno *Int*.

```
def somma(i: Int, j: Int) = i+j
```

Le parentesi graffe possono essere omesse solo se il corpo della funzione presenta una sola istruzione.
Per quanto riguarda il **return** e il tipo di ritorno le cose si fanno più complicate.
Il tipo di ritorno non viene inferito da Scala nei seguenti casi:

- quando è presente la parola chiave **return**.
- quando la funzione è ricorsiva.
- quando il metodo è sovvracaricato e chiama un altro metodo sovraccaricato con lo stesso nome. La parola chiave **return** può quasi sempre essere omessa e il valore ritornato è l'ultima istruzione valutata.

Questo permette di omettere il **return** e il tipo di ritorno anche quando stiamo valutando un **if**. Un caso nel quale il **return** è necessario si presenta quando vogliamo ritornare prima che l'ultima istruzione sia eseguita.

4.2 Funzioni locali

Quando noi istanziamo un oggetto di una classe possiamo invocare tutti i metodi che sono messi a disposizione dalla classe di cui è istanza. Non tutti i metodi di una classe però sono messi a disposizione dagli oggetti perché alcuni di loro non hanno senso all'esterno della classe stessa. Questo viene fatto in Java e altri linguaggi ad oggetti con i modificatori di visibilità **protected** e **private**. In Scala esiste anche un altro modo che rispecchia il paradigma funzionale e cioè le funzioni locali. Le funzioni locali non sono altro che delle funzioni annidate. Analogamente alle variabili locali le funzioni locali hanno una visibilità limitata solo all'interno della funzione più esterna in cui sono definite.

```
def fattoriale(num: Int) = {
```

```
    def fat(num: Int): Int = {
      if (num==1) 1
        else num*fat(num-1)
    }
    fat(num)
  }
```

Una cosa da notare è la presenza del tipo di ritorno della funzione più interna. Come detto precedentemente le funzioni ricorsive devono per forza dichiarare il tipo di ritorno.

4.3 Funzioni di prima classe

Nella programmazione funzionale le funzioni sono trattate come funzioni di prima classe. Questo significa che le funzioni sono delle entità di prima classe cioè è possibile passare una funzione ad un'altra funzione come parametro, è possibile ritornare una funzione da un'altra funzione come valore di ritorno oppure è possibile assegnare una funzione ad una variabile. Per fare tutto questo in Scala esistono le funzioni letterali. Le funzioni letterali sono funzioni anonime cioè nel momento in cui le andiamo a definire non dobbiamo specificare il loro nome ma solo i parametri di ingresso e di uscita e il corpo:

```
(x: Int) => x + 1
```

Questa funzione mappa l'intero in ingresso nel suo incremento. La sintassi è di per sé intuitiva; I parametri in ingresso sono specificati tra parentesi come in una normale definizione di funzione mentre il corpo è specificato a destra della freccia. Un esempio che mostra come usare le funzioni letterali è il seguente:

```
val lista = List(1,2,3,4)
val listaPari = lista.filter((x: Int) => x%2==0)
```

Anche se non è stato detto cosa possa essere il metodo *filter* non è difficile immaginarlo. *filter* non fa altro che iterare sulla lista ed eseguire il codice della funzione letterale su ogni elemento. Ovviamente la funzione letterale, che noi passiamo, deve presentare come tipo di ritorno un boolean altrimenti la funzione *filter* non potrebbe capire come filtrare gli elementi da iterare. Scala offre al programmatore la possibilità di scrivere codice il più conciso possibile. Il precedente esempio può essere riscritto in modo più semplice senza rinunciare alla sua leggibilità:

```
val listaPari = lista.filter(x => x%2==0)
```

Quello che è stato omesso è il tipo dell'oggetto x che introduce solo ridondanza. Infatti il metodo *filter* viene chiamato su una lista di interi e quindi non è necessario indicare di che tipo saranno gli elementi che vengono iterati. Addirittura si può procedere ad un ulteriore passo di semplificazione grazie al segnaposto '_':

```
val listaPari = lista.filter(_%2==0)
```

All'inizio potrebbe sembrare una sintassi inespressiva ma, dopo un po' di pratica, risulterà tutto molto più chiaro. Per comprendere meglio l'esempio possiamo pensare all'underscore come uno spazio vuoto che ad ogni iterazione viene riempito con un elemento della lista.
Ritornando alle funzioni letterali cerchiamo di capire come possa una funzione essere trattata come se fosse una variabile. Le funzioni letterali in realtà vengono trasformate in funzioni valore cioè oggetti che

estendono il tratto *FunctionN* dove N è il numero di parametri specificato nella firma delle funzioni letterali. Possono essere invocate come delle vere funzioni perché in realtà quello che viene invocato è il metodo *apply* dell'oggetto che le contiene. Nel metodo *apply* sarà presente il codice che noi abbiamo indicato come parametri di ingresso. Le funzioni letterali e le funzioni valore sono però differenti come gli oggetti e le loro classi: le funzioni letterali sono presenti a tempo di compilazione mentre le funzioni valore sono presenti a tempo di esecuzione.

4.4 Chiusure

Nel paragrafo riguardante le funzioni locali non è stato menzionato un concetto molto importante cioè il concetto di chiusura. Quando in Java scriviamo una funzione, le uniche variabili che quella funzione vede sono i parametri in ingresso e le variabili locali definite nel suo ambito di visibilità. In Scala, invece, una funzione può vedere al di fuori del proprio corpo per cercare una variabile citata nel suo codice ma dichiarata altrove.

```
val y: Int = 10
val func = (x: Int) => x + y
```

Tecnicamente si dice che la funzione assegnata a *func* è una funzione aperta che cerca di trovare un legame con la variabile dichiarata esternamente. Una funzione che effettua questa operazione è chiamata chiusura. Se una funzione non contiene variabili dichiarate altrove si dice che è una funzione chiusa su se stessa. L'ambito di visibilità della chiusura non è altro che il normale ambito di visibilità di una funzione. In questo modo il legame con variabili esterne viene trovato non solo nella stessa espressione in cui è presente la funzione ma anche in tutto l'ambito di visibilità della funzione.

```
class CSuperChiusura  {
  val x = 1
}

class CChiusura extends CSuperChiusura{
  def foo = {
    def Innerfoo =  {
       val i: (Int => Int) = y => y + x
    }
  }
}
```

La funzione innestata *Innerfo* dichiara al suo interno una funzione letterale il cui scopo è quello di sommare i valori interi passati in ingresso con la variabile intera *x*.

4.5 Funzioni parzialmente applicate

Quando viene creata una funzione valore da una funzione letterale non invochiamo direttamente il metodo apply ma viene semplicemente creato l'oggetto. Solo quando invochiamo la funzione allora il metodo apply farà il suo lavoro.

```
val f = (x: Int) => x + 1
println(f(2))
```

E' evidente che nella prima istruzione non ci può essere invocazione perché x assume nessun valore. Nella seconda istruzione invece forniamo un valore ed è allora che verrà invocata la nostra funzione. Questo ci porta al concetto di funzioni parzialmente applicate cioè funzioni "*applicate*" solo in parte ai loro parametri.

```
val f = (x: Int, y: Int) => x + y
val g = f(1,_ :Int)
println(g(2))
```

La funzione valore *g* è una funzione parzialmente applicata perché viene fornito un solo parametro di ingresso. Il fatto che il compilatore non si lamenti non dovrebbe stupire perché g è un oggetto che viene costruito a tempo di esecuzione ma non esegue immediatamente il corpo del metodo. Solo quando viene invocato il suo metodo *apply* allora devono essere specificati tutti i parametri del metodo.
Una delle applicazioni delle funzioni parzialmente applicate è l'uso del segnaposto '_'. Quando sono state presentate le funzioni come valori di prima classe è stato anche detto che la sintassi può essere abbreviata usando il segnaposto '_'. Il motivo per il quale questo si può fare è proprio perché quando scriviamo il segnaposto '_' stiamo applicando parzialmente le funzioni ai parametri di ingresso:

```
val l = (1 to 10).toList
l.foreach(x => println(x))
l.foreach(println _)
```

Le due istruzioni *foreach* sono equivalenti per il semplice motivo che sarà la funzione *foreach* ad applicare la funzione *println* a un parametro concreto.

4.6 Currying

Come in molti linguaggi di programmazione la firma di una funzione contiene il nome della funzione e i suoi parametri di ingresso e uscita. I parametri di ingresso vanno scritti in sequenza e divisi tra loro da una virgola. In Scala esiste una tecnica di programmazione chiamata currying che permette al programmatore di specificare i parametri di ingresso di una funzione come una lista di parametri presi singolarmente. Per applicare la tecnica del currying, alla firma di un metodo, è sufficiente racchiudere tra parentesi tonde ogni parametro di ingresso. La stessa cosa vale per i valori di ingresso nell'invocazione del metodo. Ecco la differenza tra la stessa funzione con e senza currying:

```
def Somma(i: Int, j: Int) = i + j
def CurSomma(i: Int)(j: Int) = i + j

Somma(1,1)
CurSomma(1)(1)
val s = Function.curried(Somma _)
s(1)(1)
```

In questo esempio sono state presentate due funzioni: *Somma* e *CurSomma.*La differenza tra le due è che la funzione *CurSomma* è in forma curried, mentre la funzione *Somma* non lo è. Questo si vede sia nella firma del metodo e sia nella sua invocazione.

Da questo esempio, si vede inoltre come trasformare una funzione normale in una funzione curried tramite il metodo *curried* dell'oggetto *Function*. Il parametro di ingresso sarà ovviamente una funzione e, grazie al segnaposto '_', possiamo non specificare i parametri della funzione da trasformare.
In alcuni linguaggi di programmazione funzionali non è possibile specificare funzioni con più di un parametro di ingresso. Ecco allora che, grazie al currying, si possono specificare in modo semplice e leggibile funzioni con qualsiasi arità. Scala, però, non ha questo problema perché, come è già stato visto, si possono specificare funzioni con più parametri di ingresso senza aver bisogno del currying. Il motivo principale per il quale il currying è usato anche in Scala è la creazione di nuovi costrutti di controllo.

4.7 Creare nuovi costrutti

Nei linguaggi che supportano la programmazione funzionale le funzioni sono trattate come valori di prima classe. Questo permette ad una funzione di accettare come suo parametro di ingresso un'altra funzione.
Per creare un nuovo costrutto di controllo in Scala non serve altro. In parole povere per creare un nuovo costrutto di controllo dobbiamo definire una funzione di ordine superiore.
Supponiamo di voler definire un costrutto identico al ciclo **while**. Quello che ci serve è una funzione che controlli una condizione booleana e una funzione che esegua le istruzioni all'interno del ciclo **while**.
Ad esempio se vogliamo usare un ciclo **while** personalizzato che incrementi una variabile intera possiamo farlo nel seguente modo:

```
def loop(bool: Int => Boolean, func: Int => Int) = {
  while (bool(y)) {
    y = func(y)
  }
}

var y: Int = 0
loop(y => y < 10, y => {
  println(y)
  y + 1
})
```

Anche se l'intero codice sembra scomodo e non certo uguale al codice nativo di Scala quando scriviamo un ciclo **while** , in realtà è solo la chiamata a *loop* che deve assomigliare alla definizione di un ciclo **while**.
Nonostante questo, invocare il metodo *loop* non sembra per nulla uguale a scrivere un ciclo **while**.
Ad esempio quando noi indichiamo il corpo del ciclo **while** di Scala usiamo le parentesi graffe al posto di quelle tonde. Possiamo però avvicinarci a qualcosa di simile sostituendo le parentesi graffe al posto di quelle tonde, ma questo vale solo nel caso la funzione abbia un parametro di ingresso. Se proviamo ad eseguire il seguente codice ci verrà segnalato un errore nell'ultima riga.

```
def abs(x: Int) = Math.abs(x)
println { abs{-10} }

def somma(x: Int, y: Int) = x + y
println { somma{1 , 2} }
```

La prima funzione riceve un parametro di ingresso, mentre la seconda ne riceve due.
Se proviamo a passare in ingresso dei parametri usando le parentesi graffe al posto delle parentesi tonde,

la prima funzione verrà invocata senza problemi, mentre la seconda funzione non compilerà. Da notare quindi che anche il metodo *println* può usare le parentesi graffe.
In realtà questa sostituzione non è una vera sostituzione. Questa possibilità deriva dal fatto che si possono omettere le parentesi tonde e la dot notation per i metodi che hanno un solo parametro di ingresso. Sostanzialmente, quando usiamo le parentesi graffe, stiamo semplicemente passando un espressione come parametro di ingresso.
Per aggirare questo spiacevole inconveniente possiamo però usare la tecnica del currying. Se tutto questo si può fare solo quando dobbiamo passare un parametro di ingresso allora ci basterà usare il currying per applicare le parentesi graffe a qualsiasi parametro di ingresso.
Il nostro *loop* personalizzato può essere ridefinito in questo modo:

```
def loop(bool: Int => Boolean)(func: Int => Int) = {
  while (bool(y)) {
    y = func(y)
  }
}

var y: Int = 0
println(y)
loop(y => y < 10){y => {
    println(y)
     y + 1 }
}
```

Ora la nostra funzione *loop* può essere invocata in modo molto simile alla sintassi nativa di un linguaggio di programmazione.
Non siamo però ancora soddisfatti. Quando scriviamo un ciclo **while** la sintassi che usiamo non fa uso del simbolo *'=>'*. Anche per questo c'è una soluzione cioè i parametri passati per nome.
Quando passiamo un parametro ad una funzione solitamente passiamo una copia della variabile perché in questo modo non possiamo in alcun modo modificare il valore della variabile originale. Questo tipo di passaggio è chiamato passaggio per valore. Il passaggio per nome invece prevede la valutazione del valore dell'espressione solo quando questa viene effettivamente usata. Vediamo come questo può esserci utile per modificare la sintassi del nostro ciclo:

```
def loop(bool: => Boolean)(func: => Unit) = {
  while (bool) {
    func
  }
}

var y: Int = 0
loop(y < 10) {
    println(y)
    y = y + 1
}
```

La sintassi per indicare il passaggio per nome prevede di non specificare nulla a sinistra del simbolo *'=>'*.
Ora la sintassi del nostro ciclo **while** personalizzato è uguale in tutto e per tutto a quella di un ciclo **while** nativo del linguaggio.

Per modificare anche il tipo dei parametri e rendere più generale il metodo *loop* è necessario usare i tipi parametrici cioè i generics di Java. I tipi parametrici servono proprio per rendere più generale una classe o nel nostro caso un metodo. Le modifiche da apportare sono molto semplici:

```
def loop[A](bool: => Boolean)(func: => A) = {
  while (bool) {
    func
  }
}

val arr = Array(1,2,3)
var i = 0
loop(i < arr.length && i % 2 ==0) {
    println(arr(i))
    i = i + 1
}
```

Rispetto al penultimo esempio è stato indicato un tipo parametrico come sostituto del tipo di ritorno della funzione. Questo permette al nostro *loop* di specificare qualsiasi istruzione al suo interno. Come in Java, i tipi parametrici vanno specificati prima di invocare un metodo o istanziare una classe, ma, ancora una volta, l'inferenza di tipo dimostra le sue potenzialità. Infatti il tipo finale nel nostro esempio è *Unit* e senza l'inferenza di tipo saremmo stati obbligati a scrivere la sgradevole invocazione a *loop* come:

```
loop[Unit](…) {…}
```

Detto più brevemente, quello che è stato ottenuto con l'ultima forma del ciclo *loop*, ci permette di specificare qualsiasi condizione booleana e qualsiasi corpo di istruzioni da eseguire. Questo rispecchia in tutto e per tutto il ciclo **while** nativo del linguaggio.

5 Classi case e pattern matching

5.1 Metodo apply

Il metodo *apply* è un metodo particolare che viene invocato su un'istanza di una classe o su un oggetto associato senza bisogno di essere scritto esplicitamente. Quando il metodo *apply* si usa su un oggetto associato è buona norma usarlo come metodo di factory. Un esempio di utilizzo del metodo *apply* si ha quando si definisce una funzione letterale:

```
val funcLetterale: (Int => Int) = x => x+1
funcLetterale(1)
```

Come già detto, una funzione letterale viene trasformata in una funzione valore cioè un oggetto fornito del metodo *apply*. L'oggetto viene riferito tramite la variabile *funcLetterale* che, potendo invocare il metodo *apply*, viene considerata sintatticamente come una funzione.
Anche se non è obbligatorio usare *apply* come metodo di factory, questa è l'applicazione più diffusa sugli oggetti associati. Questo vincolo viene meno quando si usa il metodo *apply* su oggetti normali.
Un esempio è dato dalla classe *Array* che usa *apply* come metodo di *get*.

```
val arr = Array(1,2,3)
println(arr.apply(1))
println(arr(1))
```

Il metodo *apply* è stato usato 3 volte. Nella prima istruzione viene chiamato sull'oggetto associato *Array* e per convenzione crea una nuova istanza della classe *Array*. Nella seconda e nella terza istruzione viene invece invocato sull'oggetto della classe. Qui si vede anche come non sia necessario scrivere esplicitamente il nome del metodo per poterlo richiamare.

5.2 Classi case

Quando una classe è definita come una classe case vengono apportate varie modifiche ai campi e ai metodi al suo interno. Viene aggiunto automaticamente il metodo *apply*, utile per costruire espressioni senza dover inserire la parola **new** davanti ad ogni costruttore.
I parametri di ingresso della classe sono trasformati automaticamente in val. In questo modo senza altre modifiche è possibile accedere dall'esterno ai parametri di ingresso della classe.
Vengono sovrascritti i metodi *hashCode*, *toString* e *equals*. La sovrascrittura di questi metodi permette di considerare la struttura interna della classe case. Ad esempio il metodo *equals* può confrontare il contenuto di un oggetto piuttosto che il suo riferimento.
Per definire una classe case è sufficiente aggiungere la parola case davanti alla classe.

```
case class Intero
case class Decimale
```

Il vantaggio maggiore derivante dalle classi case è che le classi case supportano il pattern matching.

5.3 Pattern matching

Il pattern matching viene implementato in Scala con il costrutto match già visto in precedenza.
Quello che però non è stato trattato sono i pattern che possono essere confrontati dal costrutto match.

1. Clausola jolly: la clausola jolly '_' può essere usata in molte situazioni tra cui il pattern matching. Quando vogliamo definire un caso di default per essere certi di catturare tutti i casi che si possono presentare, possiamo usare la clausola jolly '_'. Oltre a questo, la clausola jolly può essere usata per evitare di specificare singole parti che compongono un caso da confrontare.

```
(1,2) match {
   case (i:Int, j:Int) => println("E' una tupla di interi")
   case (_,_) => println("E' una tupla")
   case _ =>
 }
```

   Nel primo caso specifichiamo che la tupla richiesta contenga due elementi interi. Nel secondo caso, invece, permettiamo una corrispondenza su tuple contenenti qualsiasi tipo di dato. Ad esempio se una tupla contenesse una stringa e un intero, solo il secondo caso combacerebbe. L'ultimo caso indica la corrispondenza con qualsiasi tipo di dato, come per esempio un *Object.*

2. Costanti e variabili: nei casi possibili del costrutto match possono comparire costanti e variabili. Per costanti si intende qualsiasi variabile val, oggetti singleton oppure oggetti che rappresentino un letterale. Per variabile si intende una qualsiasi lettera senza un particolare tipo di dato.

```
a match {
  case 1 =>
  case false =>
  case Pi =>
  case e => println(e)
}
```

   In questo esempio la variabile a viene confrontata con quattro casi. Le prime tre sono delle costanti, l'ultima è una variabile.
   Mentre le costanti corrispondono solo a se stesse, una variabile corrisponde a tutti gli oggetti che gli vengono confrontati. Si potrebbe pensare che la clausola jolly sia identica alla variabile *e* ma questo non è vero. Infatti, la variabile *e* può catturare l'oggetto che gli è stato confrontato ed essere usata come suo riferimento. Inoltre, una variabile non può essere usata per indicare parti di oggetti che non si vogliono specificare. Se per indicare una variabile basta scrivere un nome o una lettera, come si fa a distinguere le costanti come Pi greco? La differenza sta nella prima lettera del nome; se la prima lettera è minuscola stiamo indicando una variabile, altrimenti una costante.
   Infatti, nell'esempio la costante Pi ha la prima lettera maiuscola. Se siamo interessati a usare una costante con la prima lettera minuscola possiamo usare due metodi: inserire i caratteri backtick attorno al nome della costante oppure scrivere il nome completo della costante compreso l'oggetto di appartenenza.

```
val x = 2
val e = 0
val a: Any = 0
a match {
```

```
    case `e` =>
    case this.x =>
  }
```

La costante *e* può essere minuscola perché circondata dai caratteri di backtick, mentre alla costante x è indicato il nome completo compreso l'oggetto di appartenenza.

3. Costruttori: una differenza importante rispetto allo switch case di Java è il confronto sui costruttori. Infatti per poter inserire un costruttore tra i casi da confrontare il pattern matching di Scala deve essere in grado controllare che il costruttore si riferisca ad una classe case e continuare il confronto tra l'oggetto che si vuole confrontare e i parametri forniti al costruttore.
   Tra l'altro questi parametri possono essere a loro volta costruttori da confrontare.
   In questo modo il pattern matching permette di effettuare visite in profondità su classi definite case.

```
a match {
    case Cont(Cont(1)) =>
    case _ =>
}
```

4. Sequenze: il pattern matching permette anche di trovare delle corrispondenze su sequenze di ogni genere.

```
a match {
    case Array(_*) =>
    case head :: tail =>
    case List(v @ _,_*) => println(v)
    case _ =>
}
```

   Il primo caso confronta *Array* di ogni dimensione. Scrivere *' _*'* permette infatti di specificare che la dimensione della sequenza può essere qualsiasi.
   Nel secondo caso è presente l'operatore *::* chiamato cons. Questo operatore può essere usato in due situazioni. La prima,più frequente, riguarda la sua invocazione su di una lista. Il suo scopo è quello di inserire un elemento in testa ad una lista. La seconda, all'interno del pattern matching, non riveste il ruolo di un metodo ma di una classe case. In questo modo stiamo usando il pattern costruttore. In Scala quindi esiste l'operatore *::* che fa parte dei metodi della classe lista ma esiste anche la classe :: che serve appunto come costruttore nel pattern matching.
   Di conseguenza nel secondo caso stiamo specificando una lista di cui *head* è la testa e *tail* la parte rimanente.
   Nel terzo caso stiamo specificando esattamente quello che già specifichiamo del secondo. Infatti la notazione *'v @ _'* indica che la variabile v sarà legata al primo elemento della lista, qualunque esso sia. La notazione *'_*'* indica, invece, che la parte restante non ha limiti di lunghezza.

5. Tipi: come già detto il pattern matching di Scala permette di effettuare confronti su qualsiasi tipo. Un tale meccanismo in realtà permette di ottenere lo stesso risultato usando i metodi *isIstanceOf[T]* e *asIstanceOf[T]* della classe *Any*.
   La differenza tra le due scelte sta nella quantità e nella chiarezza del codice da scrivere.

```
def confronta(a: Any) = {
  a match {
```

```
      case i: Int =>
      case s: String =>
      case _ =>
    }
}

def confronta1(a: Any) = {
  if (a.isInstanceOf[Int]) {
    val i = a.asInstanceOf[Int]
  }
  if (a.isInstanceOf[String]) {
    val s = a.asInstanceOf[String]
  }
}
```

Oltre a essere usato come uno *switch case* il pattern matching può essere usato anche nella definizione di variabili e cicli **for**. Se conosciamo la costituzione di una classe case possiamo scompattarla in singole variabili.

```
val t2 = (1 , 2)
val (i, j) = t2

val map = Map("1" -> "2", "2" -> "3", "3" -> "4")
for ((s, t) <- map) {}
```

Nel primo esempio i singoli componenti della tupla *t2* vengono inseriti nelle variabili *i* e *j*. Nel secondo esempio le coppie chiave valore contenute in *map* vengono una alla volta inserite nelle variabili *s* e *t*.

5.4 Classi sealed

Quando usiamo il pattern matching è importante gestire tutti i casi che potrebbero presentarsi altrimenti, non appena si cerca di confrontare un caso che non è stato coperto, il programma lancerà l'eccezione *MatchError*.
Siccome il compilatore non può sapere quali casi vogliamo gestire, è stato ideato un modo per far si che il compilatore conosca tutti i possibili pattern che possono interessarci. Se infatti il pattern matching fosse definito solo per alcune classi case e non per altre, allora il compilatore potrebbe segnalare tutti i casi possibili da gestire.
Questo viene fatto creando una superclasse per tutte le classi case che riguardano il pattern matching e gli viene applicata la parola chiave **sealed**.
In questo modo, se noi non gestiamo un caso che riguarda una classe case, il compilatore ci restituirà un warning.

```
sealed class Numero
case class Intero extends Numero
case class Decimale extends Numero

  def confronta(a: Numero) = {
    a match {
       case i:Intero => println("intero")
       case i:Decimale => println("decimale")
    }
```

```
}
```

In questo caso non abbiamo gestito il caso in cui a sia un *Numero*, e quindi verrà generato il seguente warning:

```
<console>:13: warning: match is not exhaustive!
missing combination        Numero
```

Il warning ci avvisa che il caso *Numero* non è coperto.
Nel caso in cui non vogliamo vedere warning possiamo sempre aggiungere un caso di default con il wildcard '_' oppure inserire un'annotazione:

```
def confronta(a: Numero) = {
  (a: @unchecked) match  {
    case i:Intero => println("intero")
    case i:Decimale => println("decimale")
  }
}
```

Ovviamente se i modelli non soddisfano il confronto verrà lanciato un errore.

6 Tipi parametrici

I tipi parametrici sono gli equivalenti generics di Java. Un tipo parametrico è un qualsiasi tipo che viene parametrizzato quando viene creata una sua nuova istanza. La loro funzione è quella di rendere più generico l'uso di una particolare classe. Per definire un tipo parametrico e per istanziarlo si usa la sintassi seguente:

```
class TipoParametrizzato[A](val i: A)
val tipoPar = new TipoParametrizzato[Int](1)
val tipoPar1 = new TipoParametrizzato(1)
```

Il parametro di tipo viene indicato tra parentesi quadre, anziché tra i simboli *'< >'* di Java.
Nell'esempio, la seconda istanza del tipo parametrizzato non dichiara tra parentesi quadre il tipo da passare come parametro. Questo perché Scala inferisce il tipo di *tipoPar1* dal costruttore della classe. Di conseguenza anche *tipoPar1* è istanza di *TipoParametrizzato[Int]*.

6.1 Covarianza

Una delle domande che ci si può chiedere, è se anche per i tipi parametrici vale la gerarchia dei tipi. Ad esempio, se il tipo *String* è un sottotipo di *Any,* il tipo *List[String]* è un sottotipo di *List[Any]*?.
Per le liste questo è vero, ma in generale è falso. Per usare la terminologia corretta, si dice che le liste sono covarianti rispetto al tipo che le parametrizzano, oppure nel caso questo parametro sia singolo, si dice semplicemente che le liste sono covarianti.
Per quale motivo, però, in Scala i tipi parametrici non sono sempre covarianti? Se fosse sempre disponibile in qualsiasi tipo parametrico, la covarianza porterebbe a delle assegnazioni errate. Proviamo a vederlo con un esempio:

```
class Contenitore[A](var a: A)

val c = new Contenitore[Int](1)
val c1: Contenitore[Any] = c
c1.a_=("ciao")
println(c.a)
```

Nella terza istruzione è stato usato il metodo *a_=(),* l'equivalente del metodo set di Java. Supponiamo che la classe *Contenitore* sia covariante. La prima istruzione crea un contenitore di tipo *Int,* mentre la seconda istruzione crea un contenitore di tipo *Any*. E' importante osservare che le variabili *c* e *c1* sono di tipo diverso, ma puntano allo stesso oggetto. In questo modo, possiamo modificare il contenitore di tipo *Int* utilizzando il contenitore di tipo *Any* inserendo una stringa. Il compilatore non si può lamentare perché la stringa viene inserita attraverso *c1* che è un contenitore di tipo *Any*. Ovviamente, però, questa assegnazione non è corretta perché avremmo un contenitore di tipo *Int* con all'interno una stringa. Per questo motivo, Scala impedisce l'uso delle covarianza se non esplicitamente richiesto. Non essendo dichiarato l'uso della covarianza sulla classe Contenitore, il codice scritto sopra non compilerà perché verrà segnalato un errore nell'assegnazione di un contenitore di tipo *Int* a un contenitore di tipo *Any*.

Per dichiarare un tipo parametrico come covariante rispetto a un certo parametro, basta aggiungere un segno *'+'* al parametro in questione:

```
class TipoCovariante[+A](val a: A)

val tipo = new TipoCovariante[Int](1)
val tipo1 : TipoCovariante[Any] = tipo
```

Ora l'assegnazione di un tipo *Int* a un tipo *Any* non solleverà lamentele da parte del compilatore. La stessa cosa non vale per la classe *Contenitore*. Se guardiamo bene la classe *TipoCovariante* e la classe *Contenitore,* noteremo una differenza tra i due campi in comune. Nella classe *TipoCovariante* il campo *a* è immutabile, mentre nella classe *Contenitore* il campo *a* è mutabile.
Se provassimo a definire covariante la classe *Contenitore,* il compilatore segnalerebbe un problema proprio per via del campo mutabile. Da questo si capisce che, in tutti i casi in cui un oggetto espone un campo mutabile all'esterno, la covarianza non può essere usata. Per risolvere il problema si può rendere il campo mutabile privato all'istanza di appartenza:

```
class Contenitore[+A](private[this] var a: A)
```

Esiste, però, anche un altro caso in cui la covarianza non è ammessa e in cui i campi sono tutti immutabili. Come esempio, supponiamo di creare una classe parametrizzata covariante e di inserire un metodo set a cui andrà passata una variabile di tipo *A*. Successivamente, creiamo una classe derivata con nessuna parametrizzazione e che sovrascriva il metodo set in modo da dover passare una variabile di tipo *Int*. Il codice è il seguente:

```
class superTipo[+A](val j: A){
  def set(j: A) = println(j)
}

class TipoCovariante(val i: Int) extends superTipo[Int](i){
  override def set(j: Int) = j+1
}

var t: superTipo[Any] = new TipoCovariante(1)
t.set("ciao")
```

Tutto sembrerebbe funzionare perché non ci sono campi mutabili e quindi il problema precedente sembrerebbe risolto. Purtroppo, però, a tempo di compilazione la variabile *t* può invocare il metodo *set()* con qualsiasi tipo di variabile, mentre a tempo di esecuzione, per via del polimorfismo, invocherà il metodo *set()* della classe derivata. Quello che si ottiene è la somma con una stringa che darà ovviamente un errore. Per questo motivo, oltre agli stati mutabili, una classe non deve mai esporre un metodo in cui i tipi di ingresso sono parametri di tipo.

6.2 Controvarianza

Oltre alla covarianza, esiste anche il concetto di controvarianza. La controvarianza è completamente contro-intuitiva e permette, ad esempio, di considerare una *List[Any]* come un sottotipo di *List[String].* Questo sembrerebbe contraddittorio, però ci sono alcune situazioni in cui la controvarianza può servire.

Una di queste è il tratto *Function1[A,B],* che viene usato quando definiamo una funzione letterale di tipo *A=>B* e che permette a Scala di estendere la classe della funzione valore.
Ma a cosa può servire la controvarianza nel tratto *Function1*? Se il parametro *A* si riferisce ai parametri di ingresso e il parametro *B* si riferisce ai parametri in uscita, allora il tratto *Function1* è controvariante rispetto ad *A* ed è covariante rispetto a *B*. La definizione del tratto *Function1* è:

```
trait Function1[-A,+B] {}
```

Per definire la controvarianza, rispetto a un parametro di tipo, si usa il simbolo meno.
Per capire perché *Function1* è controvariante in *A* e covariante in *B*, prendiamo come esempio la classe *S* e la classe *T* e definiamo due funzioni letterali in questo modo:

```
class S
class T extends S

val func1 = (s: S) => println(s)
val func2 = (t: T) => println(t)
func1(new S)
func1(new T)
func2(new S) //errore
func2(new T)
```

La funzione *func1* può prendere in ingresso la classe derivata *T*, mentre la classe *func2* non può prendere in ingresso la classe base *S*. Questo significa che *func1* può sostituire *func2* anche se è *T* che può sostituire *S*. Per il principio di sostituzione di Liskov[1], *func1* è un sottotipo di *func2* anche se *S* è un supertipo di *T*. Le funzioni sono quindi controvarianti rispetto ai loro parametri di ingresso.
Prendiamo ora in considerazione il parametro di uscita per capire perché è covariante:

```
val func1 = new S
val func2 = new T
val s1: S = func1
val s2: T = func1 //errore
val t1: S = func2
val t2: T = func2
```

La funzione *func2* può essere assegnata sia a un tipo *S* e sia a un tipo *T*, mentre la funzione *func1* non può essere assegnata a un tipo *T*. Questo vuol dire che *func2* può sostituire *func1* e cioè, per il principio di Liskov, *func2* è un sottotipo di *func1*. Se quindi *func2* è un sottotipo di *func1*, seguendo per coerenza che *T* è un sottotipo di *S*, allora si dice che le funzioni sono covarianti rispetto al tipo di ritorno.
Da tutto questo possiamo generalizzare l'ultimo problema presentato nel capitolo precedente e capire come il compilatore Scala effettui un controllo per evitare un uso improprio della varianza di annotazione. Quando usiamo un parametro di tipo per definire una variabile nel corpo della classe, possiamo pensare che nella definizione della variabile, la locazione del parametro di tipo occupa una posizione che viene contrassegnata dal compilatore come positiva, neutrale o negativa. In generale, un tipo parametrico deve avere tutti i parametri di tipo rispetto al quale è covariante, invariante o controvariante in posizioni rispettivamente positive, neutrali o negative.

---

[1] Il principio di sostituzione di Liskov afferma che se un entità *B* può sostituire un entità *A*, allora l'entità *B* è un sottotipo dell'entità *A*. Ad esempio, siano *A* e *B* due classi con *B* sottoclasse di *A.* Per il principio di sostituzione di Liskov la classe *B* è un sottotipo della classe *A*.

Ad esempio, siccome una funzione è controvariante rispetto ai tipi dei parametri di ingresso, allora tutte le posizioni dei tipi dei parametri di ingresso saranno contrassegnate negativamente. In questo modo, se noi provassimo a compilare l'ultimo esempio del capitolo precedente, il compilatore Scala segnalerebbe un errore. Infatti, per via del polimorfismo, il metodo che dichiarava un *Int* come parametro di ingresso sovrascriveva un metodo che dichiarava un *Any*. Questo comportamento è covariante rispetto ai tipi dei parametri di ingresso e difatti, se fosse permesso, ci porterebbe ad una assegnazione errata.
Il compilatore Scala applica un algoritmo per controllare che i parametri di tipo siano tutti nelle loro rispettive posizioni. I casi più comuni riguardano le funzioni e le variabili. Come già spiegato, i parametri di tipo contrassegnati negativamente possono comparire come tipi di parametro di ingresso delle funzioni, ma non in quelli di uscita. Invece, i parametri di tipo contrassegnati positivamente possono comparire come tipi di parametro di uscita della funzione, ma non in quelli di ingresso. Per quanto riguarda le variabili, è necessario considerare che queste presentano il metodo *get()* se sono immutabili e i metodi *get()* e *set()* se sono mutabili. In base a questo, le variabili immutabili non possono essere dichiarate con parametri con annotazione negativa e le variabili mutabili non possono avere parametri ne con annotazione positiva ne con annotazione negativa.

6.3 Limite inferiore e superiore

I tipi parametrici sono stati creati come generalizzazione delle classi. Possiamo creare delle classi che non devono essere riscritte ogni qual volta si presenti la necessità di usare un nuovo tipo, non considerato in precedenza.
La stessa cosa vale anche per i metodi. Possiamo generalizzare i metodi rendendoli polimorfici, cioè inserendo un parametro nella loro firma. In questo modo, possiamo anche inserire dei limiti inferiori o superiori su questo nuovo parametro.
Questi limiti possono essere usati per limitare l'uso di un metodo solo per dei tipi particolari. Per esempio, possiamo usare il limite inferiore per riuscire a usare un parametro di tipo:

```
class TipoPar [+A](val i: A){
  def set(k: A) = k
}

class TipoPar1[+A](val i: A){
  def set[B >: A](k: B) = k
}
```

Il metodo set della classe *TipoPar1* è un metodo polimorfico perché è parametrizzato con il parametro *B*. Il parametro *B* è limitato inferiormente dal parametro *A* con il simbolo *'>: '.*
La classe *TipoPar* non compila perché è presente un parametro positivo nei parametri di ingresso. Se però imponiamo, come nella classe *TipoPar1*, che il parametro da passare al metodo *set* deve essere un supertipo del parametro *A*, allora non ci dovrebbero più essere problemi. Infatti, in questo modo, se proviamo a passare un tipo *String* mentre il parametro dell'oggetto è *Int*, Scala assegnerà a B il primo supertipo in comune sia con *String* e sia con *Int* cioè *Any*. Un esempio concreto già implementato in Scala è il metodo *::* della classe *List*:

```
def ::[U >: T](x: U): List[U] = new scala.::(x, this)
```

Questo metodo ritorna una nuova lista di elementi di tipo *U* inserendo il nuovo elemento passato in ingresso in testa a quelli già presenti. La cosa importante da notare è la presenza del limite inferiore *T*.
Infatti, in questo, modo siamo obbligati a inserire solo elementi che sono supertipi di quelli già presenti. La nuova lista ritornata sarà del supertipo in comune tra tutti gli elementi delle lista e quindi non possiamo avere una lista di *Int* in cui sono presenti elementi di tipo *String.*
Le seguenti righe sono copiate dalla shell dell'interprete di Scala e mostrano il tipo della lista dopo l'inserimento di un elemento intero e di un elemento stringa.
Se la lista è di tipo *Int,* e inseriamo una stringa, il tipo della lista risultante è di tipo *Any*.

```
scala> val l = List(1)
l: List[Int] = List(1)

scala> "stringa":: l
res20: List[Any] = List(ciao, 1)
```

Il caso simmetrico è rappresentato dal limite superiore. Se ad esempio volessimo ritornare un tipo contrassegnato come negativo da una funzione, possiamo indicare nella firma del metodo polimorfico un limite superiore:

```
class C[-A] {
  def get[B <: A](c: B): B = {
    c
  }
}
```

I limiti inferiore e superiore possono essere anche usati per effettuare un controllo sui parametri di ingresso di una funzione. Se ad esempio vogliamo che tutti i parametri di ingresso siano sottotipi di *AnyVal* e non di *AnyRef,* possiamo scrivere il seguente metodo:

```
def SoloAnyVal[A <: AnyVal](a: A) = {}
```

Un'applicazione del limite superiore riguarda i metodi di ordinamento che vogliono specificare solo un particolare insieme ristretto di tipi.

6.4 Tipi e membri astratti

Come in molti altri linguaggi, in Scala è possibile definire alcune entità chiave del linguaggio come astratte. Ad esempio, le classi possono essere definite come astratte e questo implica che la loro implementazione dovrà essere completata in classi derivate. Anche i tratti sono classi astratte con la differenza di non avere un costruttore e di poter essere impilabili.
Oltre alle classi e ai tratti, si possono dichiarare astratti anche i loro membri. Insieme ai campi mutabili o immutabili e ai metodi, in Scala, a differenza di Java, esistono anche i tipi astratti.
I tipi astratti sono molto simili ai tipi parametrici e in molte situazioni possono essere interscambiabili.
La differenza tra i due sta nel campo di utilizzo. Infatti i tipi parametrici sono utili quando si vuole generalizzare una classe come una struttura dati e vengono indicati a tempo di esecuzione, mentre i tipi astratti sono utili quando vogliamo definire in modo univoco il comportamento di un tipo a tempo di

compilazione. Inoltre, possiamo usarli come Alias per accorciare il nome di un tipo che vogliamo usare ripetutamente.
Per definire un membro astratto è sufficiente non specificare alcun comportamento.
Per le variabili non si deve assegnare nessun valore, mentre per i metodi non si deve assegnare nessun codice. Lo stesso vale per i tipi astratti, che sono definiti con la parola chiave type davanti al loro identificativo. Ecco qualche esempio di entità completamente astratte:

```
abstract class Persona{
  type T
  var nome: String
  var cognome: String
  val anno: String
  def parla
}
```

Nella classe astratta *Persona* sono presenti un tipo, delle variabili e un metodo astratti.

6.5 Inizializzare membri astratti

In Scala, come in Java, esiste il concetto di classe anonima, cioè una classe senza nome che viene creata sul posto racchiudendo il suo corpo tra parentesi graffe. Le classi anonime sono utili quando vengono istanziate una sola volta in tutto il codice. Un utilizzo molto frequente, può essere la gestione di un evento lanciato da un qualche componente grafico.
Le classi anonime possono essere anche usate come espediente per istanziare una classe astratta senza necessariamente doverla estendere in un'altra classe. Il meccanismo utilizzato è molto semplice e vale anche per i tratti.

```
trait tratto {
  val c1: Int
  val c2: Int
  val somma = c1 + c2
}

val l = new tratto{
  val c1 = 1
  val c2 = 2
}
println(l.somma)
```

Quello che viene fatto in realtà, è creare una classe anonima in cui viene miscelato il tratto. Questo è equivalente a scrivere una classe normale che sovrascrive i campi del tratto esteso.
E' importante fare una precisazione. Se eseguiamo questo codice, l'istruzione *println(l.somma)* non stampa tre, ma stampa zero. Questo accade, perché il tratto viene creato prima della classe anonima che andrà a sovrascrivere i campi *c1* e *c2* e non dopo. Ora, se il nostro tratto fosse usato veramente per eseguire una somma questo non sarebbe il comportamento aspettato.
Per modificare l'ordine di inizializzazione esistono due possibilità. La prima consiste nel preinizializzare i campi, mentre la seconda utilizza i campi lazy. Preinizializzare i campi significa essenzialmente creare prima la classe anonima e poi estenderla con il tratto. In questo modo, l'ordine di creazione è invertito e

nell'esempio l'istruzione *println(l.somma)* stamperebbe tre.

```
val l = new {
  val c1 = 1
  val c2 = 2
} with tratto
```

L'altra soluzione prevede l'utilizzo dei campi lazy. Per dichiarare una variabile come lazy è sufficiente usare la parola chiave lazy di fronte a una variabile val. Dichiarare una variabile come lazy significa che la sua inizializzazione verrà ritardata al suo primo utilizzo.

```
trait tratto {
  val c1: Int
  val c2: Int
  lazy val somma = c1 + c2
}

val l = new tratto{
  val c1 = 1
  val c2 = 2
}
println(l.somma)
```

Anche in questo caso l'istruzione *println(l.somma)* stamperà il valore tre perché la variabile *somma* sarà valutata quando *c1* e *c2* sono già state inizializzate. Rispetto ai campi preinizializzati, le variabili lazy offrono la possibilità, al programmatore, di non doversi interessare all'ordine con le quali le variabili saranno inizializzate. Questo può rivelarsi utile soprattutto quando i programmi diventano molto complessi o quando dobbiamo usare le classi scritte da qualcun altro.

6.6 Differenza tra tipi astratti e parametrici

Per usare i tipi parametrici, indichiamo a fianco al nome della classe, dei parametri racchiusi tra parentesi quadre. In questo modo, possiamo indicare concretamente i tipi che dovranno assumere questi parametri. Un caso in cui i tipi parametrici sono molto usati sono le collezioni dati.

I tipi astratti non possono essere specificati con la stessa comodità dei tipi parametrici quando creiamo un'istanza della classe. Per questo motivo non vengono usati per generalizzare una classe. Il loro impiego principale è quello di qualsiasi altro membro astratto, cioè essere specializzati nella classi figlie. Un esempio che mostra anche un altro uso del limite superiore è il seguente:

```
abstract class Lavoro
class Dirigente extends Lavoro {
  var Subordinati = 1
}
class Dipendente extends Lavoro {
  var Supervisori = 1
}

abstract class Persona {
  type L <: Lavoro
  def Lavora(l: L)
```

```
}

class Segretaria extends Persona {
  type L = Dipendente
  def Lavora(l: L) = {
    l.Supervisori
  }
}

class Operaio extends Persona {
  type L = Dipendente
  def Lavora(l: L) = {
    l.Supervisori
  }
}

class Presidente extends Persona {
  type L = Dirigente
  def Lavora(l: L) = {
    l.Subordinati
  }
}
```

Nella classe *Persona* sono stati definiti il metodo *Lavora* e il tipo *L* come astratti. In entrambi i casi, i due membri vanno concretizzati nelle classi sottostanti che ereditano da *Persona.*
Per questo motivo, grazie al tipo astratto *L*, possiamo indicare in modo specifico quale tipo di lavoro una sottoclasse di *Persona* dovrà assumere. I tipi parametrici, invece, obbligano a chi istanzia la classe di specificare il parametro di tipo. In questo caso, sarebbe scomodo e permetterebbe di assegnare alle sottoclassi di Persona un lavoro qualsiasi.
Inoltre, considerando che l'override di un metodo richiede di specificare gli stessi tipi di parametro di ingresso, possiamo sovrascrivere il metodo *Lavora* modificando il tipo di ingresso, ma lasciando inalterata la firma.

7 Attori

Negli ultimi anni la programmazione concorrente ha acquisito sempre maggiore importanza. Con l'avvento dei processori multicore e dei sistemi distribuiti è stato necessario trovare delle soluzioni a livello software che permettessero ai programmatori di sfruttare al meglio l'architettura di un elaboratore.
Il problema di questo tipo di programmazione è la complessità. Infatti molti linguaggi mettono a disposizione del programmatore strumenti utili per la gestione di vari thread che compongono un'applicazione, ma rimane comunque complicato scrivere un programma che funzioni correttamente e che sia facile da comprendere. Ad esempio, in Java possiamo usare semafori , regioni critiche , monitor e librerie a livello più astratto come *java.concurrent*. Nonostante questo, la scrittura di applicazioni multithreaded resta un esercizio non banale.
Infatti tutti questi strumenti usano sempre lo stesso approccio e cioè la condivisione di uno stato mutabile tra i vari thread. Con questo tipo di approccio, è probabile che un programma segua uno sviluppo inaspettato e finisca in deadlock o presenti effetti di interferenza.
Un'alternativa più semplice prevede lo scambio di messaggi tra i vari thread. In questo modo, i thread non devono condividere nessuno stato in comune e quindi non c'è più la necessità di sincronizzare gli accessi a una risorsa. Per fare questo, Scala offre come supporto gli attori, cioè unità esecutive come i thread con la possibilità di scambiarsi messaggi o riceverli nella propria mailbox.

7.1 Definire un attore

Per creare un attore ci sono due possibilità. La prima prevede di estendere la classe *Actor* della libreria *scala.actors*. e implementare il metodo *act()*. Una volta istanziato un nuovo attore si deve usare il metodo *start()* per lanciare il nuovo flusso esecutivo che andrà ad eseguire il codice scritto nel metodo *act()*:

```
import scala.actors._

class Attore extends Actor {
  def act() {
    println("ciao mondo da Attore")
  }
}

val Att = new Attore
Att.start()
```

La seconda possibilità permette di agevolare la creazione di un attore usando il metodo *actor* presente nell'oggetto *scala.actors.Actor*:

```
import scala.actors.Actor._

val Att = actor {
   println("ciao mondo da Attore")
}
```

Ora l'esecuzione del nuovo thread viene fatta partire in automatico.

7.2 Inviare un messaggio

Per inviare un messaggio ad un attore si deve invocare su di esso il metodo *'!'* passando come argomento il messaggio che si vuole inviare. Se ad esempio volessimo inviare all'attore definito sopra un messaggio di tipo stringa, sarebbe sufficiente scrivere:

```
Att ! "ciao mondo da Main"
```

L'invio di un messaggio con il metodo *'!'* è asincrono, cioè l'invio non fermerà l'attore sorgente per aspettare una risposta da parte del ricevente. Esiste anche una versione sincrona e cioè il metodo *'?!'* che bloccherà il mittente in attesa di una risposta. Bisogna fare attenzione, però, che l'invio sincrono può portare allo stallo. Se ad esempio un attore *A* attende una risposta da un attore *B* e viceversa, allora il programma non potrà più andare avanti. Nelle conclusioni è presente un esempio.
Un 'altra possibilità che riguarda l'invio di un messaggio è il metodo *'!!'* che dichiara il tipo *Future* come tipo di ritorno. Un oggetto della classe *Future* può essere usato come segnaposto dell'effettivo valore che momentaneamente non è disponibile. Infatti, un attore potrebbe impiegare del tempo prima di elaborare una nostra richiesta, oppure potrebbe essere momentaneamente occupato a gestire richieste da parte di altri attori. Se il valore richiesto non ci serve subito, possiamo continuare la nostra esecuzione per cercare di non buttare via tempo.
Se siamo interessati a leggere il valore memorizzato nell'istanza di un *Future,* possiamo usare il metodo *value()*. Siccome questo metodo è bloccante, possiamo usare il metodo *isSet()* per capire se il risultato della precedente richiesta è stato elaborato.

7.3 Ricevere un messaggio

Quando un attore riceve un messaggio dal mondo esterno, inserisce temporaneamente il suo messaggio in una mailbox. Per fare in modo che questo messaggio venga prelevato, bisogna invocare il metodo *receive* e passare come parametro di ingresso una funzione che provvederà a gestire correttamente il messaggio. Siccome i messaggi che un attore riceve possono essere di qualsiasi tipo, bisogna accertarsi che la funzione in questione sia in grado di gestire il messaggio.
Per fare questo, è necessario che la funzione in questione sia in realtà una *PartialFunction*. Un tale oggetto funzione consente di controllare se un dato input può essere correttamente gestito dal metodo *apply.* Questo viene fatto attraverso il metodo *isDefinedAt*, a cui verrà passato il valore di ingresso che si vuole valutare. Se il metodo ritornerà una valutazione positiva, allora il metodo *apply* sarà in grado di gestire quel particolare valore di input.
Grazie a questo sistema, prima di invocare il metodo *apply*, un attore invocherà prima il metodo *isDefinedAt* della funzione parziale sul messaggio che si vuole gestire. Se il risultato del controllo sarà *true,* allora il messaggio verrà effettivamente prelevato e gestito dal metodo *apply* della funzione parziale, altrimenti sarà lasciato nella mailbox.

```
class Attore extends Actor {
  val partialFunc = new PartialFunction[Any,Unit] {
    def apply(x: Any) = {
      println(x)
    }
    def isDefinedAt(x: Any) = {
```

```
      true
    }
  }
  def act() {
    receive {
      partialFunc
    }
  }
}

val Att = new Attore
Att ! "ciao mondo da Main"
Att.start()
```

La funzione parziale da passare al metodo *receive* richiede un tipo *Any* come parametro di ingresso e un tipo qualsiasi come parametro di uscita. Il metodo *isDefinedAt* è stato semplificato per comodità. Durante il suo normale funzionamento, l'attore rivelerà la presenta di un nuovo messaggio, controllerà da prima che il risultato del metodo *isDefinedAt,* sul messaggio in questione*,* sia *true* e poi invocherà il metodo *apply* di *partialFunc* che stamperà il messaggio inviato.
In realtà, se dovessimo definire in modo così esplicito una funzione parziale, le cose sarebbero alquanto scomode. Per definire in modo molto più semplice lo stesso comportamento, è sufficiente usare il pattern matching. Infatti, una sequenza di clausole case può essere vista come una funzione parziale. Lo stesso esempio può essere così riscritto:

```
class Attore extends Actor {
  def act() {
    receive {
      case s: String => println(s)
    }
  }
}
```

In questo caso, il metodo *isDefinedAt()* viene implementato automaticamente e ritornerà *true* solo per i messaggi presenti nella mailbox di tipo *String*.
Un'osservazione molto importante riguarda il metodo *receive*. Quando il metodo *receive* non ha messaggi utili da processare, sospende il thread sul quale viene eseguito. Se ad esempio abbiamo n attori sospesi in attesa di un messaggio, allora abbiamo anche n thread sospesi in attesa di proseguire.
I thread sono chiamati processi leggeri perché richiedono all'atto della loro creazione meno risorse rispetto ai processi. Purtroppo, creare nuovi thread richiede poche risorse finché il loro numero rimane limitato. Quindi, se invece di sospendere un thread quando l'attore non ha più messaggi da gestire nella propria mailbox, assegniamo il thread ad un altro attore, non abbiamo più bisogno di creare tanti thread quanti sono gli attori del nostro programma.
Per effettuare questa operazione esiste il metodo *react*. Il metodo *react* differisce dal metodo *receive* in quanto il suo tipo di ritorno è *Nothing*. Più precisamente, quando viene ricevuto un messaggio nella mailbox, le azioni intraprese dal metodo *react* sono simili a quelle del metodo *receive* con la sostanziale differenza che viene sempre lanciata un eccezione alla fine del metodo; l'eccezione in questione è *SuspendActorException*.
Il motivo per il quale viene lanciata un eccezione, risiede nel fatto che il lancio di un'eccezione libera il call stack delle chiamate finché l'eccezione non viene gestita. In questo modo viene liberato il call stack del thread che può essere riutilizzato per eseguire ulteriori task in attesa.
Bisogna comunque considerare che Il lancio di un eccezione comporta però un piccolo prezzo da pagare: le

istruzioni successive al metodo react non verranno eseguite.
Se vogliamo che un attore non termini la propria esecuzione dopo aver ricevuto un messaggio, dobbiamo inserire i metodi *receive* o *react* all'interno di un ciclo. Bisogna però tener presente che il metodo *react* lancia un eccezione. Di conseguenza non è possibile inserirlo all'interno di un ciclo **while** perché, se l'eccezione non viene gestita, il programma terminerà. Per questo motivo esiste l'istruzione *loop,* che cattura e gestisce l'eccezione al suo interno. Ecco un esempio:

```
class Attore extends Actor {
  def act() {
    loop{
      react {
             case s: String => println(s);
      }
    }
  }
}
```

Per non far terminare il metodo *react,* è anche possibile usare la ricorsione invocando il metodo *act()* alla fine di ogni caso:

```
class Attore extends Actor {
  def act() {
      react {
           case s: String => println(s); act()
      }
  }
}
```

Oltre ai metodi *react* e *receive,* esistono le rispettive copie con timeout. E' sufficiente indicare un tempo in millisecondi come primo parametro di ingresso e prevedere il pattern *'TIMEOUT'* tra i vari casi da gestire.
Un'importante osservazione riguarda i metodi *receive*, *react* e *loop*. Questi metodi eseguono il corpo definito tra parentesi graffe. Questo modo di procedere è uguale a qualsiasi altro costrutto di controllo nativo del linguaggio. In realtà Scala non supporta nativamente gli attori ma permette appunto la creazione di nuovi costrutti di controllo che rendono la programmazione di un particolare problema più semplice da gestire.

7.4 Funzionamento in invio e ricezione

Cerchiamo ora di capire come i metodi di invio e ricezione collaborino per permettere lo scambio di messaggi tra i vari attori.
Per fare questo, però, dobbiamo analizzare più in dettaglio la differenza tra i metodi *react* e *receive*. Come già detto, il metodo *receive* ritorna il valore dell'elaborazione di un messaggio, mentre il metodo *react* ritorna sempre un eccezione. Le conseguenze che questo comporta si manifestano sia nel caso un messaggio possa essere gestito e sia nel caso avverso.
Infatti, quando un messaggio non può essere gestito dalla funzione parziale, il metodo *receive* rende nota la funzione *isDefinedAt* e sospende il thread su cui è in esecuzione, mentre il metodo *react* mette a disposizione dei potenziali mittenti la funzione parziale per gestire i messaggi. Se invece un messaggio può essere gestito, il metodo *receive* procede direttamente all'esecuzione del codice necessario, mentre il metodo *react* crea un task da fare eseguire allo scheduler che gestisce gli attori.

Da notare che il metodo *react* non esegue il codice direttamente, perché il codice che gestisce i messaggi potrebbe contenere ricorsioni o cicli che non permetterebbero il lancio dell'eccezione finale.
Da queste sostanziali differenze, possiamo allora comprendere come funziona il metodo di invio asincrono. Il metodo *!* non fa altro che invocare il metodo *send*, a cui verrà specificato il messaggio da inviare e l'attore mittente. Le azioni intraprese dal metodo *send* non riguardano solo l'accodamento del messaggio nella mailbox, ma anche lo stato dell'attore destinatario.
Consideriamo il caso in cui l'attore destinatario esegua il metodo *receive* al proprio interno. Se l'attore è attivo ma impegnato in altre operazioni, il metodo *send* accoda il messaggio nella mailbox. Se invece l'attore è rimasto sospeso per la mancanza di messaggi validi da gestire, accodare un messaggio nella sua mailbox non lo farà di certo risvegliare. Per questo motivo, il metodo di invio controlla dapprima se l'attore destinatario è sospeso e, in caso affermativo, se il messaggio soddisfa la funzione *isDefinedAt,* specifica il messaggio da consegnare e lo riattiva.
Ora consideriamo il caso in cui l'attore destinatario esegua il metodo *react.* Come prima, se l'attore è attivo ma impegnato in altre operazioni, il metodo *send* accoda il messaggio nella mailbox. Nel caso contrario, siccome ai potenziali mittenti viene resa nota la funzione parziale interna al metodo *react*, se il messaggio soddisfa la funzione *isDefinedAt,* il metodo *send* crea un task da far eseguire allo scheduler che gestisce gli attori.

7.5 Esempio produttori consumatori

Il problema che prende il nome di Produttori-Consumatori prevede l'uso di una quantità di spazio disponibile su cui i produttori depositano i loro messaggi e i consumatori li prelevano. Il problema presenta la necessità di sincronizzare gli accessi tra produttori e consumatori: un produttore non può proseguire nel caso lo spazio sia esaurito e un consumatore non può proseguire nel caso non ci siano messaggi da prelevare.
Per risolvere il problema con gli attori, è sufficiente mettere un attore di guardia a tutti gli accessi. In questo modo possiamo gestire il buffer circolare che conterrà i messaggi e coordinare l'attività dei produttori e dei consumatori per evitare che non eseguano operazioni sbagliate. Ecco il codice d'esempio:

```
import scala.actors._

case class Set(val Mes: Any)
case class Get

class Produttore(val buf: Buffer) extends Actor{
  def act() {
    while(true) {
      Thread.sleep(10)
      buf !? Set(Thread.currentThread().getId())
    }
  }
}

class Consumatore(val buf: Buffer) extends Actor{
  def act() {
    while(true) {
      Thread.sleep(10)
      println(buf !? Get)
    }
```

```
    }
  }

  class Buffer(val n: Int) extends Actor{
    val buf = new Array[Any](n)
    var testa, coda = 0
    var spazio = n

    def act() {
      loop {
        react {
          case s: Set if spazio>0 => {
            spazio = spazio-1
            buf(testa) = s.Mes
            testa = (testa+1)%n
            sender ! ""
          }
          case Get if spazio<n => {
            spazio = spazio+1
            sender ! buf(coda)
            coda = (coda+1)%n
          }
        }
      }
    }
  }

  object ProduttoriConsumatori extends App {
    var b = new Buffer(1)
    b.start()
    for (i <- 1 to 10) {
        Thread.sleep(100)
       (new Produttore(b)).start()
       (new Consumatore(b)).start()
    }
  }
```

Sono presenti la classe *Consumatore*, la classe *Produttore* e la classe *Buffer*. Tutte e tre estendono il tratto *Actor*. I produttori e i consumatori non fanno altro che cercare di inserire o rimuovere i messaggi da un array circolare gestito dall'attore della classe *Buffer*. Sia i produttori e sia i consumatori utilizzano il metodo di invio sincrono per sospendersi quando l'array è pieno o vuoto rispettivamente. Come messaggi di scambio con l'attore della classe *Buffer*, sono state usate le classi case *Get* e *Set*. Il risultato di ritorno del messaggio *Get* viene stampato.

All'interno della classe Buffer, la gestione dei messaggi prevede l'uso di due casi: Il caso in cui il messaggio sia di tipo *Set,*ma con l'array non pieno e il caso in cui il messaggio sia di tipo *Get,* ma con l'array non vuoto. Per controllare che l'array sia pieno o vuoto, viene usato il concetto di guardia sulla variabile *spazio*. Una guardia non è altro che un **if** all'interno di un case. Le guardie permettono di ottenere una valutazione più fine sulle condizioni da rispettare e, a differenza di un normale controllo all'interno del codice del case, evitano di entrare in un caso che poi non può essere gestito nell'immediato.

Se ad esempio non usassimo le guardie, quando arriva un messaggio e lo spazio è esaurito, dovremmo sospendere l'attore e risvegliarlo quando un consumatore avrà liberato lo spazio per una posizione. La stessa cosa vale per un consumatore che non trova messaggi da prelevare.

Grazie alle guardie, un messaggio all'interno della mailbox non viene prelevato finché non soddisfa un'intera condizione specificata da un case. Non dobbiamo comunque preoccuparci di gestire quei

messaggi accodati quando non erano gestibili. Infatti, a ogni invocazione dei metodi *receive* e *react,* la ricerca nella mailbox di messaggi da gestire riparte sempre da capo.
Se un messaggio corrisponde a uno dei due casi, vengono intraprese le operazioni opportune per inserire o prelevare il messaggio, per sistemare gli indici *testa*, *coda* e *spazio* e per sbloccare il mittente. Nell'oggetto singleton *ProduttoriConsumatori* è presente un esempio di utilizzo delle tre classi. In questo caso, è stato creato un buffer con una posizione, dieci produttori e dieci consumatori.

7.6 Ultime considerazioni

Gli attori di Scala permettono di usare un approccio alternativo alla condivisione di risorse mutabili in comune. Il loro funzionamento non prevede l'accesso concorrenziale alla stessa area di memoria da parte di più attori, ma prevede solo lo scambio di messaggi. Per questo motivo, viene eliminata la necessità di chi scrive programmi multithreaded, di usare la sincronizzazione. Tuttavia, ci sono alcuni accorgimenti che vanno tenuti presenti.
Il metodo di invio sincrono *'!?'* può causare una situazione di stallo. Ecco un esempio di come sia facile cadere in questa situazione:

```
class Attore extends Actor {
  def act() {
      receive {
        case e: Actor => e !? "a";
      }
  }
}

val Att = new Attore
Att.start()
Att !? Actor.self
println("fine")
```

Le quattro righe di codice scritte alla fine sono eseguite dal thread principale dell'applicazione. L'istruzione *Actor.self* viene usata per trattare un thread come un attore. In questo caso, *Actor.self* si riferisce al thread principale. Quando l'attore *Att* riceve l'attore del thread principale, invoca su di esso il metodo di invio sincrono. A questo punto i due attori aspettano una risposta uno dall'altro e rimarranno bloccati in eterno. La stringa fine non viene stampata.
Per rispondere a una risposta sincrona, sbloccando il mittente, si deve usare lo stesso canale usato per l'invio. Per fare questo, esiste il metodo *reply()* o l'oggetto *sender* che rappresentano il canale di comunicazione tra i due attori. Il corpo delle funzione parziale può essere così modificato:

```
case e: Actor => reply("a")  oppure case e: Actor => sender ! "a";
```

Un'altra cosa da tener presente è lo stato che un attore espone all'esterno. Tutti i membri mutabili dovrebbero rimanere sempre nascosti dall'esterno per evitare che il codice risulti difficile da controllare. Se dobbiamo inviare un messaggio, è sempre meglio usare un oggetto immutabile poiché, questo, può essere letto senza preoccuparsi di modifiche effettuate da altri attori. Se ad esempio un oggetto mutabile viene modificato quando è nella mailbox in attesa di essere letto, allora si potrebbero verificare comportamenti inaspettati. Nel caso dovessimo gestire una risorsa condivisa per apportare delle modifiche possiamo usare il modello a thread di Java, oppure mettere un attore a guardia di tutti gli accessi.

Ovviamente usare gli attori introduce un certo overhead rispetto ai thread di Java, considerando anche il fatto che gli attori stessi sono un'astrazione basata sui thread di Java. Inoltre, se i messaggi da inviare sono molti e molto onerosi in termini di memoria, allora il modello ad attore potrebbe essere poco efficiente.

Conclusioni

Qual è il valore aggiunto che Scala apporta al mondo della programmazione? Molte delle cose presenti nel linguaggio Scala sono già state inventate in precedenti linguaggi. La programmazione funzionale era addirittura stata inventata prima degli attuali elaboratori elettronici nel 1930 con il Lambda calcolo di Alonzo Church. La programmazione ad oggetti, ai giorni nostri, è già usata in molti altri linguaggi molto famosi come il Java. Inoltre, tutti gli ulteriori strumenti che compongono il linguaggio Scala come i tratti, il pattern matching, i tipi parametrici, la libreria degli attori sono già presenti in molti altri linguaggi di programmazione. Ci sono una moltitudine di linguaggi che supportano tutte le funzionalità del linguaggio Scala. Ad esempio, i mixin sono già presenti in altri linguaggi di programmazione come Ruby, il pattern matching è già presente in linguaggi funzionali come Lisp, Perl o Haskell, i tipi parametrici sono già presenti In Java o C++ e gli attori sono un'astrazione già implementata in numerosi linguaggi come Erlang. Anche l'inferenza di tipo non è una novità. I linguaggi funzionali Haskell e Erlang possiedono già un meccanismo per inferire i tipi a tempo di compilazione.
Quello che offre Scala in più a tutti gli altri linguaggi, è la possibilità di avere a disposizione tutti questi strumenti in un unico linguaggio. Ad esempio, anche se l'idea della scalabilità di un linguaggio era già nata negli sessanta, solo Scala supporta la programmazione funzionale e la programmazione ad oggetti cercando di complementare la forza dei due paradigmi a vicenda. Scala è il risultato dell'unione e la fusione di singole parti positive che compongono gli altri linguaggi di programmazione.
Inoltre, essendo un linguaggio moderno creato per fornire tutto quello che già Java può offrire ma con una sintassi più leggera, molti ritengono che Scala ha tutte le carte in regola per sostituire Java in futuro. Chiaramente Scala non è l'unico linguaggio uscito negli ultimi anni e non è l'unico linguaggio a fornire vantaggi superiori rispetto a Java o ad altri linguaggi. Esistono molti contendenti che ovviamente hanno i loro punti di forza e che meglio si adattano a Scala in diverse situazioni. Per capire se e quale di tutti questi linguaggi possono sostituire Java si potrebbe cercare di analizzare nel dettaglio ogni singolo linguaggio cercando di capire quale meglio si adatta ad ogni situazione, ma questo forse potrebbe essere un impresa proibitiva per chiunque. Quale sarà il linguaggio del futuro? Forse solo il tempo riuscirà a darci una risposta.

Bibliografia

1. Odersky M. & Spoon L. & Venners B. (2008) "Programming in scala", 1st edition, California, Artima.
2. Piancastelli G. (2009-10), "Programmare in scala", http://gpiancastelli.altervista.org/scala-it/index.html
3. Schinz M. & Philipp H., "A Scala Tutorial for Java programmers", http://www.scala-lang.org/docu/files/ScalaTutorial.pdf,(2011).
4. Odersky M. (2011) "The Scala Language Specification", http://www.scala-lang.org/docu/files/ScalaReference.pdf
5. "Covariance and contravariance - a simple guide", http://www.i-programmer.info/programming/theory/1632-covariance-and-contravariance-a-simple-guide.html, (2010).
6. "Programmazione funzionale", http://it.wikipedia.org/wiki/Programmazione_funzionale
7. Phaller (2007), "Scala Actors: A Short Tutorial", http://www.scala-lang.org/node/242
8. Thomas C. (2010),"Inside Scala Actors", http://wiki.ita.hsr.ch/SemProgAnTr/files/Inside_Scala_Actors.pdf