
Giorno 1
Scala
Marius Minia

Ciao 👋,
sono Marius

Qual è il vostro background?

Scala è un linguaggio di programmazione ed un'estensione per la piattaforma Java. I programmi Scala sono a conti fatti programmi per la piattaforma Java che dipendono da un'estensione (scala-library.jar) da aggiungere alle librerie richieste dall'applicazione sviluppata.

E' possibile usare le librerie della piattaforma Java in Scala ed usare le librerie della piattaforma Scala in Java, anche se la seconda possibilità non è completamente indolore come la prima. E' comunque più facile che sia un programma Scala ad usare una libreria Java piuttosto che il contrario. Quanti agli IDE, si possono usare tranquillamente sia NetBeans che Eclipse.

Il nome Scala deriva da Scalable Language cioè linguaggio scalabile. E' un linguaggio capace di adattarsi bene ad una varietà di situazioni, che oggi riguardano i linguaggi di programmazione. Può essere usato come un linguaggio di scripting o come un linguaggio per 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 ed invocare su di essi operazioni aritmetiche come se fossero dati primitivi già presenti nel linguaggio di base. Oppure è possibile creare nuovi costrutti di controllo per poter usare una sintassi più semplice ed è espressiva.

Per essere più precisi, Scala è un linguaggio puramente ad oggetti e questo significa che ogni cosa, dai numeri alle funzioni, è un oggetto. 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 programmi scritti con questo strumento, soprattutto i più complessi.

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 od un nuovo tipo di dato numerico, facendoli sembrare parte del linguaggio nativo.

Programmazione funzionale

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 ciò è poter creare nuovi costrutti di controllo, come un ciclo while.

Programmazione funzionale

La seconda idea, è quella di pensare che una funzione deve mappare un valore di ingresso in un valore di uscita. 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.

Programmazione funzionale

In generale, se le singole parti di codice che costituiscono un'applicazione non dipendono da uno stato mutabile in comune, si può ragionare sulle singole parti senza, dover tener presenti tutte le altre.

Un'altra particolarità importante del linguaggio riguarda la tipizzazione, che rappresenta una delle classificazioni possibili tra i vari linguaggi, cioè l'assegnazione 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 suo tipo.

Programmazione funzionale

La seconda, usata ad esempio nei linguaggi di scripting, si chiama tipizzazione dinamica ed è in grado di risalire al tipo di una variabile al 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.

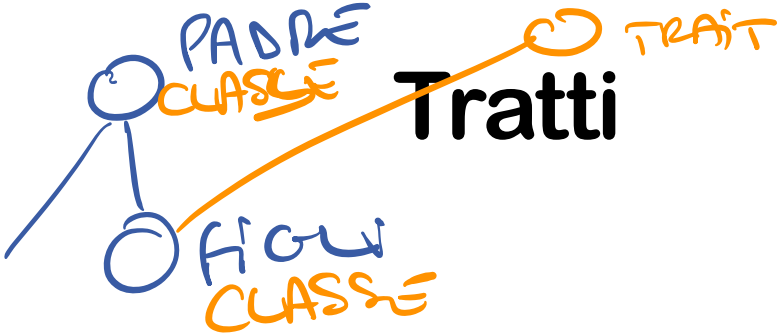
Tratti

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. 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.

Scala

Tratti



- TIPIZZAZIONE
- PROG. FUNZIO
- EREDITARIETÀ MULTIPLA !!

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.

I tratti sono simili alle interfacce di Java, ma con la sottile differenza di poter contenere campi e metodi concreti. La dichiarazione di un tratto è, quindi, simile alla dichiarazione di una classe, con l'accortezza di usare la parola chiave trait.

Tratti

Scala - tratti.scala

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

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

Mixin

CLASSE CONTO
PROPR. : SALDO
: PROPRIETARIO
METODI : getSaldo() => String
: prelievo() .def.
: calcolaProspettoTasse (INT) STATICA

Conto.calcolaProspettoTasse(55)

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.

contoXy = Conto()
contoXy.prelievo()

Mixin

Si consideri, ad esempio, il tratto Ordered. Questo tratto contiene al suo interno 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 si deve già estendere un'altra classe. La realizzazione della possibile soluzione, è riportata nel codice seguente.

Mixin

Scala - mixin.scala

```scala
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
  }
}

```

Libreria degli 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.

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.

Libreria degli Attori

Infatti tutti questi strumenti usano sempre lo stesso approccio: 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.

Libreria degli Attori

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.

Libreria degli Attori

Scala - attori.scala

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

```

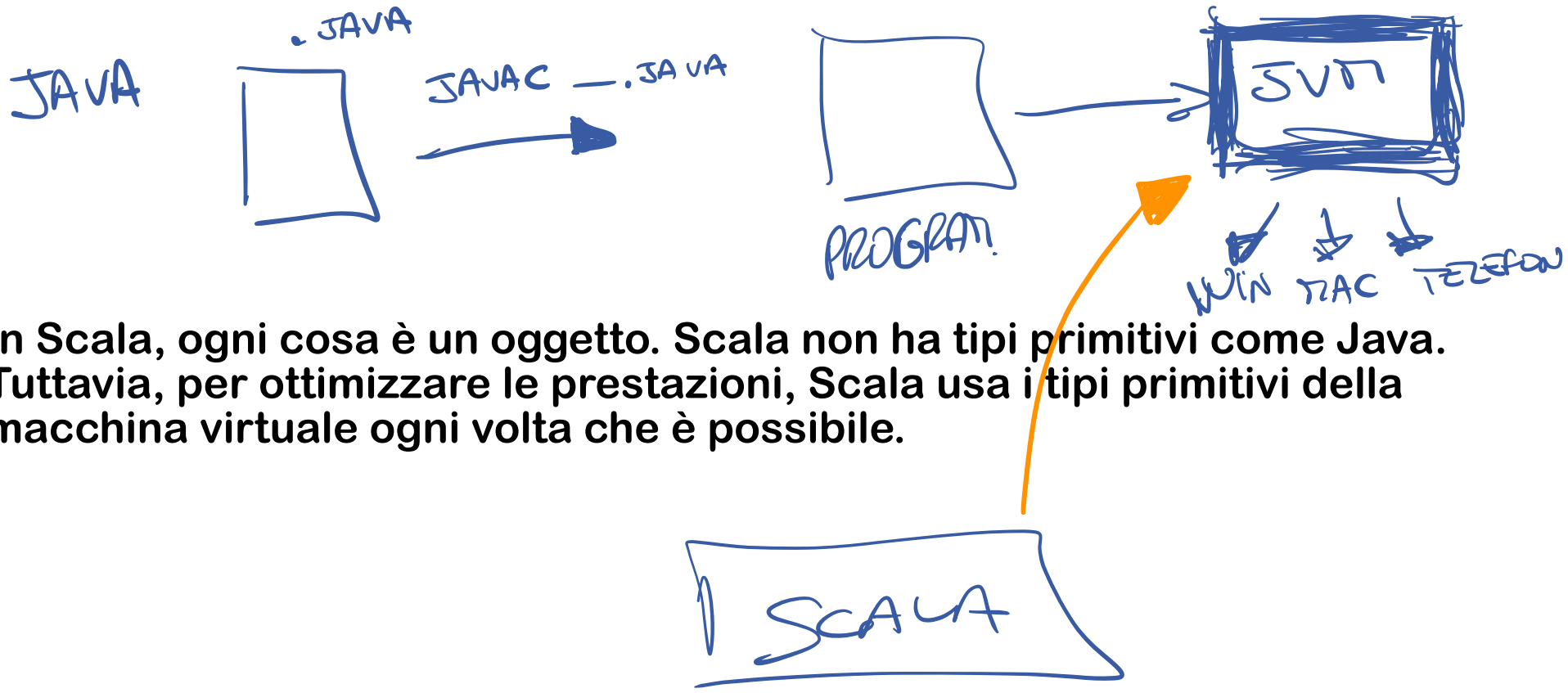
Staticamente Tipato

Un linguaggio staticamente tipato lega il tipo a una variabile per il tempo di vita di quella variabile.

Al contrario, i linguaggi dinamicamente tipati (Ruby, Python, Groovy, JavaScript) legano il tipo all'effettivo valore riferito da una variabile, permettendo quindi al tipo di una variabile di cambiare insieme al valore a cui fa riferimento.

Nel gruppo di nuovi linguaggi per la JVM, Scala è uno dei pochi a essere staticamente tipato.

Paradigma Misto



In Scala, ogni cosa è un oggetto. Scala non ha tipi primitivi come Java. Tuttavia, per ottimizzare le prestazioni, Scala usa i tipi primitivi della macchina virtuale ogni volta che è possibile.

Paradigma Misto

Scala supporta appieno la Programmazione Funzionale (FP).
La FP è un paradigma di programmazione più vecchio della OOP, ma è rimasta in secondo piano fino a poco tempo fa.

L'interesse per la FP sta aumentando a causa del modo in cui semplifica certi problemi di progettazione, in particolare quelli legati alla concorrenza, sempre presenti in applicazioni distribuite.

Paradigma Misto

I linguaggi funzionali puri non permettono alcuno stato mutabile, evitando di conseguenza il bisogno di sincronizzazione sull'accesso condiviso allo stato mutabile.

I programmi scritti in linguaggi funzionali puri comunicano scambiando messaggi tra processi autonomi e concorrenti.

Scala supporta questo modello con la sua libreria di attori e consente di usare variabili mutabili e immutabili.

Paradigma Misto

Le funzioni possono essere assegnate a variabili, passate ad altre funzioni, esattamente come gli altri valori.

Questa caratteristica promuove la composizione di comportamenti avanzati usando operazioni primitive.

Dato che Scala aderisce al principio per cui ogni cosa è un oggetto, in Scala anche le funzioni sono oggetti.

Paradigma Misto

Scala offre le chiusure, una caratteristica presente in linguaggi dinamici come Python e Ruby ma che è ancora assente in Java.

Le chiusure sono funzioni che fanno riferimento a variabili, contenute nell'ambito che racchiude la definizione stessa di funzione.

Le variabili non vengono passate come argomenti, né definite come variabili locali all'interno della funzione.

Le chiusure sono un'astrazione usata per implementare sistemi a oggetti e strutture di controllo fondamentali.

Paradigma Misto

La sintassi Java può essere prolissa. Scala usa un certo numero di tecniche per minimizzare la sintassi superflua, rendendo il codice Scala conciso quanto il codice scritto nella maggior parte dei linguaggi dinamicamente tipati.

L'inferenza di tipo minimizza il bisogno di esplicite informazioni in molti contesti. Le dichiarazioni di tipo e di funzione sono molto concise.

Scala permette ai nomi di funzione di includere caratteri non alfanumerici.

Paradigma Misto

Scala estende il sistema di tipi di Java con generici più flessibili e un numero di costrutti di tipo più avanzati.

L'inferenza di tipo aiuta a ricavare automaticamente i tipi nelle firme dei metodi e delle funzioni, in modo che l'utente non debba fornire manualmente informazioni banali.

Le caratteristiche di tipo avanzate vi forniscono una maggiore flessibilità per risolvere problemi di progettazione.

Paradigma Misto

Scala è progettato per scalare da piccoli programmi interpretati a grandi applicazioni distribuite.

Scala fornisce quattro meccanismi linguistici, che promuovono la composizione scalabile dei sistemi:

1) **tipi espliciti per la classe corrente (chiamati self-type);**
2) **generici e membri tipo astratti;**
3) **classi annidate;**
4) **composizionedimixintramitetratti.**

mixin tratti

Paradigma Misto

Potrebbe sembrare che OOP e FP siano incompatibili. In effetti, una delle filosofie di progettazione di Scala è che OOP e FP siano più sinergiche che opposte. Le caratteristiche di un approccio possono migliorare l'altro.

Il nome Scala è una contrazione delle parole scalable language (linguaggio scalabile). Sebbene questo suggerisca che la pronuncia debba essere scale-ah, i creatori di Scala in realtà lo pronunciano scah-lah.

Scala è stato concepito da Martin Odersky nel 2001. Martin è un professore della School of Computer and Communication Sciences alla Ecole Polytechnique Fédérale de Lausanne (EPFL).

Paradigma Misto

Scala è un linguaggio per sviluppatori professionali.

Paragonato a linguaggi come Java e Ruby, Scala è un linguaggio più difficile da padroneggiare, perché richiede competenze di OOP, FP e tipizzazione statica per essere usato nella maniera più efficace.

La relativa semplicità dei linguaggi dinamicamente tipati è allettante, ma questa semplicità può essere ingannevole. In un linguaggio dinamicamente tipato, è spesso necessario usare tecniche di metaprogrammazione per realizzare progetti avanzati.

Hello World


```
1  Welcome to Scala version 2.13.1 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_65).
2  scala>
3  scala> val saluto = "Hello World !" saluto: String = Hello World !
4  scala> println(saluto) Hello World !
```

Scala

Hello World #2

hello-world - Main.scala

```scala
@main def hello: Unit =
  println("Hello world!")
  println(msg)

def msg = "I was compiled by Scala 3. :)"

```