Un tutorial su Scala
per programmatori Java

Version 0.8
2 settembre 2009

Michel Schinz
Philipp Haller
Mirco Veltri (trad.)

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

1 Introduzione

Lo scopo di questo documento è quello di fornire una rapida introduzione al linguaggio ed al compilatore Scala. È rivolto a chi ha già qualche esperienza di programmazione e desidera una panoramica di cosa è possibile fare con Scala. È indispensabile una conoscenza di base dei concetti di programmazione object oriented, specialmente in Java.

2 Un primo esempio

Come primo esempio, useremo lo standard *Hello world*. Non è sicuramente un esempio entusiasmante ma rende facile dimostrare l'uso dei tool di Scala senza richiedere troppe conoscenze del linguaggio stesso. Ecco il codice:

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

La struttura del programma è sicuramente familiare ai programmatori Java; c'è un metodo chiamato `main` che accetta ed usa gli argomenti, un array di stringhe, forniti da linea di comando. Il corpo del metodo consiste di una singola chiamata al predefinito `println` che riceve il nostro amichevole saluto come parametro. Il codice del main non ritorna alcun valore, quindi non è necessario dichiararne uno di ritorno.

Ciò che è meno familiare ai programmatori Java è la dichiarazione di **`object`** contenente il metodo `main`. Questa dichiarazione introduce ciò che è comunemente chiamato *oggetto singleton*, cioè una classe con una unica istanza. La dichiarazione precedente infatti crea sia la classe `HelloWorld` che una istanza di essa, chiamata `HelloWorld`. L'istanza è creata, su richiesta, la prima volta che viene usata.

Il lettore astuto avrà notato che il metodo `main` non è stato dichiarato come `static`, questo perché i membri (metodi o campi) statici non esistono in Scala. Invece che definire membri statici, il programmatore Scala li dichiara in oggetti singleton.

2.1 Compiliamo l'esempio

Per compilare l'esempio, useremo `scalac`, cioè il compilatore Scala. `scalac` lavora come la maggior parte dei compilatori: un file sorgente come argomento, alcune opzioni e la produzione di uno o diversi object file come output. Gli object file sono gli standard class file di Java.

Se salviamo il file precedente come `HelloWorld.scala`, lo compiliamo con il seguente comando (il segno maggiore '>' rappresenta il prompt dei comandi e non va digitato):

```
> scalac HelloWorld.scala
```

Questo genererà qualche class file nella directory corrente. Uno di questi sarà chiamato `HelloWorld.class` e contiene una classe che può essere direttamente eseguita usando il comando `scala` come mostra la seguente sezione.

2.2 Eseguiamo l'esempio

Una volta compilato il programma può esser facilmente eseguito con il comando `scala`. L'uso è molto simile al comando `java` ed accetta le stesse opzioni. Il precedente esempio può esser eseguito usando il seguente comando, che produce l'output atteso:

```
> scala -classpath . HelloWorld


Hello, world!
```

3 Interazione con Java

Uno dei punti di forza di Scala è quello di rendere semplice l'interazione con codice Java. Tutte le classi del package `java.lang` vengono importate di default, mentre altre richiedono l'esplicito import.

Osserviamo un esempio che lo dimostra. Vogliamo ottenere la data corrente e formattarla in accordo con la convezione usata in uno specifico paese del mondo, diciamo la Francia.

Le librerie delle classi Java definiscono potenti classi di utilità , come `Date` e `DateFormat`. Poiché Scala interagisce direttamente con Java, non esistono le classi equivalenti nella libreria delle classi Scala– possiamo semplicemente importare le classi dei corrispondenti package Java:

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
  def main(args: Array[String]) {
    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format now)
  }
}
```

L'istruzione import di Scala è molto simile all'equivalente in Java, tuttavia, risulta essere più potente. Più classi possono essere importate dallo stesso package includendole in parentesi graffe, come nella prima linea di codice precedentemente riportato. Un'altra differenza è evidente nell'uso del carattere underscore (_) al posto dell'asterisco (*) per importare tutti i nomi di un package o di una classe. Questo perché l'asterisco è un identificatore Scala valido (e.g. nome di un metodo), come vedremo più avanti.

L'istruzione import sulla terza linea inoltre importa tutti i membri della classe `DateFormat`. Questo rende disponibili il metodo statico `getDateInstance` ed il campo statico `LONG`.

All'interno del metodo `main` creiamo un'istanza della classe `Date` di Java che di default contiene la data corrente. Successivamente definiamo il formato della data usando il metodo statico `getDateInstance` importato precedentemente. Infine, stampiamo la data corrente formattata in accordo con l'istanza di localizzazione `DateFormat`; quest'ultima linea mostra un'importante proprietà di Scala. I metodi che prendono un argomento possono essere usati con una sintassi non fissa. Questa forma dell'espressione

```
df format now
```

è solo un altro modo meno esteso di scriverla come

```
df.format(now)
```

Apparentemente sembra un piccolo dettaglio sintattico, ma presenta delle importanti conseguenze, una delle quali verrà esplorata nella prossima sezione.

A questo punto, riguardo l'integrazione con Java, abbiamo notato che è altresì possibile ereditare dalle classi Java ed implementare le interfacce Java direttamente in Scala.

4 Tutto è un oggetto

Scala è un puro linguaggio object-oriented nel senso che *ogni cosa* è un oggetto, inclusi i numeri e le funzioni. In questo differisce da Java, che invece distingue tra tipi primitivi (come `boolean` e `Int`) e tipi referenziati e non permette la manipolazione di funzioni come valori.

4.1 I numeri sono oggetti

Dato che i numeri sono oggetti, possiedono dei metodi, infatti, un'espressione aritmetica come la seguente:

```
1 + 2 * 3 / x
```

consiste esclusivamente di chiamate a metodi, perché è equivalente alla seguente espressione:

```
(1).+(((2).*(3))./(x))
```

Questo significa anche che `+`, `*`, etc. sono identificatori validi in Scala.

Le parentesi intorno ai numeri nella seconda versione sono necessarie perché l'analizzatore lessicale di Scala usa le regole di match più lunghe per i token, quindi, dovrebbe dividere la seguente espressione:

```
1.+(2)
```

nei token `1.`, `+` e `2`. La ragione per cui si è scelto questo tipo di assegnazione di significato è perché `1.` è un match più lungo e valido di `1`. Il token `1.` è interpretato come `1.0` rendendolo un `Double` e non più un `Int`. Scrivendo l'espressione come:

```
(1).+(2)
```

si evita che `1` possa esser considerato come un `Double`.

4.2 Le funzioni sono oggetti

Forse per i programmatori Java è più sorprendente scoprire che in Scala anche le funzioni sono oggetti. Quindi è possibile passare le funzioni come argomenti, memorizzarle in variabili e ritornarle da altre funzioni. Scala distingue le funzioni dai metodi che non possono essere trattati come valori. L'abilità a manipolare le funzioni come valori è uno dei punti cardini di un interessante paradigma di programmazione chiamato *programmazione funzionale*.

Come esempio semplice del perché può risultare utile usare le funzioni come valori consideriamo una funzione timer che deve eseguire delle azione ogni secondo. Come specifichiamo l'azione da eseguire? Logicamente, come una funzione. Questo tipo di passaggio di funzione è familiare a molti programmatori: viene spesso usato nel codice delle user-interface, per registrare le funzioni di call-back che vengono richiamate quando un evento è in essere.

Nel successivo programma, la funzione timer è chiamata `oncePerSecond` e prende come argomento una funzione di call-back. Il tipo di questa funzione è scritto come `() => Unit` che è il tipo di tutte le funzioni che non prendono nessun argomento e non restituiscono niente (il tipo `Unit` è simile al `void` del C/C++). La funzione principale di questo programma è quella di chiamare la funzione timer con una call-back che stampa una frase sul terminale. In altre parole questo programma stampa la frase "time flies like an arrow" ogni secondo.

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
```

```
    }
    def timeFlies() {
      println("time flies like an arrow...")
    }
    def main(args: Array[String]) {
      oncePerSecond(timeFlies)
    }
  }
```

Notare che per stampare la stringa, usiamo il metodo predefinito `println` invece di quelli inclusi in `System.out`.

4.2.1 Funzioni anonime

Il codice precedente è semplice da capire e possiamo raffinarlo ancora un po'. Notiamo preliminarmente che la funzione `timeFlies` necessita di esser definita solo per esser passata come argomento alla funzione `oncePerSecond`. Nominare esplicitamente una funzione con queste caratteristiche non è necessario. È più interessante costruire detta funzione nel momento in cui viene passata come argomento a `oncePerSecond`. Questo è possibile in Scala usando le funzioni anonime, che sono esattamente funzioni senza nome. La versione rivista del nostro programma timer usa una funzione anonima invece di `timeFlies` e appare come di seguito:

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

La presenza delle funzioni anonime in questo esempio è rivelata dal simbolo '=>' che separa la lista degli argomenti della funzione dal suo corpo. In questo esempio, la lista degli argomenti è vuota, di fatti la coppia di parentesi sulla sinistra di => è vuota.Il corpo della funzione è lo stesso del precedente `timeFlies`.

5 Classi

Come visto precedentemente Scala è un linguaggio orientato agli oggetti e come tale presenta il concetto di classe. Le classi in Scala sono dichiarate usando una sintassi molto simile a quella adoperata in Java. Una importante differenza è che

le classi in Scala possono avere dei parametri. Così come illustrato nella seguente definizione dei numeri complessi.

```
class Complex(real: Double, imaginary: Double) {
  def re() = real
  def im() = imaginary
}
```

Il costruttore prende due argomenti che sono la parte immaginaria e reale del complesso. Questi possono esser passati quando si crea una istanza della classe `Complex`, come segue: `new Complex(1.5, 2.3)`. La classe presenta due metodi, `re` ed `im`, che danno l'accesso rispettivamente alla parte reale e a quella immaginaria del numero complesso.

Si nota che il tipo di ritorno dei due metodi non è specificato. Sarà il compilatore che lo dedurrà automaticamente osservando la parte a destra del segno uguale dei metodi e deducendo che per entrambi si tratta di valori di tipo `Double`.

Il compilatore non è sempre capace di dedurre i tipi; purtroppo non c'è una regola semplice capace di dirci quando sarà in grado e quando no. Nella pratica questo non è un problema poiché il compilatore sa quando non è in grado di stabilire il tipo che non è stato definito esplicitamente. I programmatori Scala alle prime armi dovrebbero provare ad omettere la dichiarazione di tipi che sembrano semplici da dedurre per osservare il comportamento del compilatore. Dopo qualche tempo il programmatore avrà la sensazione di quando è possibile omettere il tipo e quando no.

5.1 Metodi senza argomenti

È solo il caso di evidenziare come per invocare `re` ed `im` sia necessario far seguire il nome del metodo da una coppia di parentesi tonde vuote così come di seguito indicato:

```
object ComplexNumbers {
  def main(args: Array[String]) {
    val c = new Complex(1.2, 3.4)
    println("imaginary part: " + c.im())
  }
}
```

Riuscire ad accedere alla parte reale ed immaginaria come se fossero campi senza dover scrivere anche la coppia vuota di parentesi, è perfettamente fattibile in Scala, semplicemente definendo i relativi metodi *senza argomenti*. Tali metodi differiscono da quelli con zero argomenti perché non presentano la coppia di parentesi dopo il nome nè nella loro definizione, nè nel loro utilizzo. La nostra classe `Complex` può essere riscritta come segue:

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
}
```

5.2 Eredità e overriding

In Scala tutte le classi sono figlie di una super-classe. Quando nessuna super-classe viene specificata, come nell'esempio della classe `Complex`, `scala.AnyRef` è implicitamente usata.

In Scala è possibile eseguire l'override dei metodi ereditati dalla super-classe. E' quindi necessario specificare esplicitamente il metodo che si sta overridando usando il modificatore **`override`**, al fine di evitare override accidentali. Come esempio estendiamo la nostra classe `Complex` ridefinendo il metodo `toString` ereditato da `Object`.

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
  override def toString() =
    "" + re + (if (im < 0) "" else "+") + im + "i"
}
```

6 Case class e pattern matching

Un tipo di struttura dati che spesso si trova nei programmi è l'albero. Ad esempio, gli interpreti ed i compilatori abitualmente rappresentano i programmi internamente come alberi. I documenti XML sono alberi e diversi tipi di contenitori sono basati sugli alberi, come gli alberi red-black.

Esamineremo ora come gli alberi sono rappresentati e manipolati in Scala attraverso un semplice programma calcolatrice. Lo scopo del programma è manipolare espressioni aritmetiche molto semplici composte da somme, costanti intere e variabili intere. Due esempi di tali espressioni sono $1+2$ e $(x+x)+(7+y)$.

A questo punto è necessario definire il tipo di rappresentazione per dette espressioni e, a tale proposito, l'albero è la più naturale, dove i nodi sono le operazioni (nel nostro caso, l'addizione) e le foglie sono i valori (costanti o variabili).

In Java questo albero è abitualmente rappresentato usando una super-classe astratta per gli alberi e una concreta sotto-classe per i nodi o le foglie. In un linguaggio funzionale useremmo un tipo dati algebrico per lo stesso scopo. Scala fornisce il concetto di **`case class`** che è qualcosa che si trova nel mezzo delle due rappresen-

tazioni. Mostriamo come può essere usato per definire il tipo di alberi per il nostro esempio:

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

Il fatto che le classi `Sum`, `Var` e `Const` sono dichiarate come classi **case** significa che rispetto alle classi standard differiscono in diversi aspetti:

- la parola chiave **new** non è necessaria per creare un'istanza di queste classi (i.e si può scrivere `Const(5)` invece di **new** `Const(5)`),
- le funzioni getter sono automaticamente definite per i parametri del costruttore (i.e. è possibile ricavare il valore del parametro costruttore `v` di qualche istanza `c` della classe `Const` semplicemente scrivendo `c.v`),
- sono disponibili le definizioni di default dei metodi `equals` e `hashCode`, che lavorano sulle strutture delle istanze e non sulle loro identità,
- è disponibile la definizione di default del metodo `toString`, che stampa il valore in "source form" (e.g. l'albero per l'espressione $x+1$ stampa `Sum(Var(x), Const(1))`),
- le istanze di queste classi possono essere decomposte con il *pattern matching* come vedremo più avanti.

Ora che abbiamo definito il tipo dati per rappresentare le nostre espressioni aritmetiche possiamo iniziare a definire le operazioni per manipolarle. Iniziamo con una funzione per valutare l'espressione in un qualche ambiente di valutazione (environment). Lo scopo dell'environment è quello di dare i valori alle variabili. Per esempio, l'espressione $x+1$ valutata nell'environment con associato il valore 5 alla variabile x, scritto $\{x \rightarrow 5\}$, restituisce 6 come risultato.

Inoltre dobbiamo trovare un modo per rappresentare gli environment. Potremmo naturalmente usare alcune strutture dati associative come una hash table, ma possiamo anche usare direttamente delle funzioni! Un environment in realtà non è altro che una funzione con associato un valore al nome di una variabile. L'environment $\{x \rightarrow 5\}$ mostrato sopra può essere semplicemente scritto in Scala come:

```
{ case "x" => 5 }
```

Questa notazione definisce una funzione che quando riceve la stringa x come argomento restituisce l'intero 5 e fallisce con un'eccezione negli altri casi.

Prima di scrivere la funzione di valutazione diamo un nome al tipo di environment. Potremmo usare sempre il tipo `String => Int` per gli environment, ma semplifichiamo il programma se introduciamo un nome per questo tipo rendendo anche i cambiamenti futuri più facili. Questo è fatto in Scala con la seguente notazione:

```
type Environment = String => Int
```

Da ora in avanti il tipo `Environment` può essere usato come un alias per il tipo delle funzioni da `String` ad `Int`.

Possiamo ora passare alla definizione della funzione di valutazione. Concettualmente è molto semplice: il valore della somma di due espressioni è pari alla somma dei valori delle loro espressioni; il valore di una variabile è ottenuto direttamente dall'environment; il valore di una costante è la costante stessa. Esprimere quanto appena detto in Scala non è difficile:

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Questa funzione di valutazione lavora effettuando un *pattern matching* sull'albero. Intuitivamente il significato della definizione precendente dovrebbe esser chiaro:

1. prima controlla se l'albero `t` è un `Sum`; se lo è, esegue il bind del sottoalbero sinistro con una nuova variabile chiamata `l` ed il sotto albero destro con una variabile chiamata `r` e procede con la valutazione dell'espressione che segue la freccia; questa espressione può far uso delle variabili marcate dal pattern che appaiono alla sinistra della freccia, i.e. `l` e `r`;

2. se il primo controllo non è andato a buon fine, cioè l'albero non è un `Sum`, va avanti e controlla se è un `Var`; se lo è, esegue il bind del nome contenuto nel nodo `Var` con una variabile `n` e procede con la valutazione dell'espressione sulla destra;

3. se anche il secondo controllo fallisce e quindi non si tratta nè si un `Sum` nè di un `Var`, controlla se si tratta di un `Const` e se lo è, combina il valore contenuto nel nodo `Const` con una variabile `v` e procede con la valutazione dell'espressione sulla destra;

4. infine, se tutti i controlli falliscono, viene sollevata un'eccezione per segnalare il fallimento del pattern matching dell'espressione; questo caso può accadere qui solo se si dichiarasse almeno una sotto classe di `Tree`.

L'idea alla base del pattern matching è quella di eseguire il match di un valore con una serie di pattern e se il match viene trovato, estrarre e nominare varie parti del valore per valutare il codice che ne fa uso.

Un programmatore object-oriented esperto potrebbe sorprendersi del fatto che non abbiamo definito `eval` come metodo della classe `Tree` e delle sue sottoclassi. Potremmo averlo fatto, perchè Scala permette la definizione di metodi nelle classi case

così come nelle classi normali. Decidere quando usare il pattern matching o i metodi è quindi una questione di gusti, ma ha anche implicazioni importanti riguardo l'estensibilità:

- quando si usano i metodi, è facile aggiungere un nuovo tipo di nodo definendo una sotto classe di `Tree` per esso; d'altro canto, aggiungere una nuova operazione per manipolare l'albero è noioso, richiede la modifica di tutte le sotto classi;
- quando si usa il pattern matching, la situazione è ribaltata: aggiungere un nuovo tipo di nodo richiede la modifica di tutte le funzioni in cui si fa pattern matching sull'albero, per prendere in esame il nuovo nodo; d'altro canto, aggiungere una nuova operazione è semplice, basta definirla come una funzione indipendente.

Per esplorare il pattern matching ulteriormente, definiamo un'altra operazione sulle espressioni aritmetiche: la derivazione simbolica. È necessario ricordare le seguenti regole che riguardano questa operazione:

1. la derivata di una somma è la somma delle derivate,
2. la derivata di una variabile `v` è uno se `v` è la variabile di derivazione, zero altrimenti,
3. la derivata di una costante è zero.

Queste regole possono essere tradotte quasi letteralmente in codice Scala, per ottenere la seguente definizione:

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

Questa funzione introduce due nuovi concetti relativi al pattern matching. Prima di tutto l'istruzione **`case`** per le variabili ha un *controllo*, un'espressione che segue la parola chiave **`if`**. Questo controllo fa si che il pattern matching è eseguito solo se l'espressione è vera. Qui viene usato per esser sicuri che restituiamo la costante 1 solo se il nome della variabile da derivare è lo stesso della variabile di derivazione `v`. La seconda nuova feature del pattern matching qui introdotta è la *wild-card*, scritta `_`, che corrisponde a qualunque valore, senza denominarlo.

Non abbiamo esplorato del tutto la potenza del pattern matching, ma ci fermiamo qui per brevità. Vogliamo ancora osservare come le due precedenti funzioni lavorano in un esempio reale. A tale scopo, scriviamo una semplice funzione `main` che esegue diverse operazioni sull'espressione $(x + x) + (7 + y)$: prima calcola il suo valore nell'environment $\{x \to 5, y \to 7\}$, dopo calcola la derivata relativa a x e poi ad y.

```
  def main(args: Array[String]) {
    val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
    val env: Environment = { case "x" => 5 case "y" => 7 }
    println("Expression: " + exp)
    println("Evaluation with x=5, y=7: " + eval(exp, env))
    println("Derivative relative to x:\n " + derive(exp, "x"))
    println("Derivative relative to y:\n " + derive(exp, "y"))
  }
```

Eseguendo questo programma, otteniamo l'output atteso:

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative a x:
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

Esaminando l'output, notiamo che il risultato della derivata dovrebbe essere semplificato prima di essere visualizzato all'utente. La definizione di una funzione di semplificazione usando il pattern matching rappresenta un interessante problema (ma sorprendentemente ingannevole) che lasciamo come esercizio per il lettore.

7 Trait

Una classe in Scala oltre che poter ereditare da una super-classe può anche importare del codice da uno o diversi *trait*.

Probabilmente per i programmatori Java il modo più semplice per capire cosa sono i trait è concepirli come interfacce che anche la capacità di contenere del codice. In Scala, quando una classe eredita da un trait ne implementa la relativa interfaccia ed eredita tutto il codice contenuto nel trait.

Per comprendere a pieno l'utilità dei trait, osserviamo un classico esempio: oggetti ordinati. Si rivela spesso utile riuscire a confrontare oggetti di una data classe con se stessi, ad esempio per ordinarli. In Java gli oggetti confrontabili implementano l'interfaccia `Comparable`. In Scala possiamo fare qualcosa di meglio che in Java, definendo l'equivalente codice di `Comparable` come un trait, che chiamiamo `Ord`.

Sei differenti predicati possono essere utili per confrontare gli oggetti: minore, minore o uguale, uguale, diverso, maggiore o uguale e maggiore. Tuttavia, definirli tutti è noioso, specialmente perché 4 di essi sono esprimibili con gli altri due. Per esempio dati i predicati di uguale e minore, è possibile esprimere gli altri. in Scala tutte queste osservazioni possono essere piacevolemente inclusi nella seguente dichiarazione di un trait:

```
trait Ord {
  def < (that: Any): Boolean
  def <=(that: Any): Boolean = (this < that) || (this == that)
  def > (that: Any): Boolean = !(this <= that)
  def >=(that: Any): Boolean = !(this < that)
}
```

Questa definizione crea un nuovo tipo chiamato `Ord`, che ha lo stesso ruolo dell'interfaccia `Comparable` in Java e fornisce l'implementazione di default di tre predicati in termini del quarto, astraendone uno. I predicati di uguaglianza e disuguaglianza non sono presenti in questa dichiarazione in quanto sono presenti di default in tutti gli oggetti.

Il tipo `Any` usato precedentemente è il super-tipo dati di tutti gli altri tipi in Scala. Può esser visto come una versione generica del tipo `Object` in Java, dato che è altresì il super-tipo dei tipi base come `Int`, `Float`, etc.

Per rendere confrontabili gli oggetti di una classe, è quindi sufficiente definire i predicati con cui testare uguaglianza ed inferiorità, unire la precedente classe `Ord`. Come esempio, definiamo una classe `Date` che rappresenti le date nel calendario Gregoriano. Tali date sono composte dal giorno, dal mese e dall'anno che rappresenteremo tutti con interi. Iniziamo definendo la classe `Date` come segue:

```
class Date(y: Int, m: Int, d: Int) extends Ord {
  def year = y
  def month = m
  def day = d

  override def toString(): String = year + "-" + month + "-" + day
```

La parte importante qui è la dichiarazione **`extends`** `Ord` che segue il nome della classe e dei parametri. Dichiara che la classe `Date` eredita il codice dal trait `Ord`.

Successivamente ridefiniamo il metodo `equals`, ereditato da `Object`, in modo tale che possa confrontare in modo corretto le date confrontando i singoli campi. L'implementazione di default del metodo `equal` non è utilizzabile perché come in Java confronta fisicamente gli oggetti. Arriviamo alla seguente definizione:

```
  override def equals(that: Any): Boolean =
    that.isInstanceOf[Date] && {
      val o = that.asInstanceOf[Date]
      o.day == day && o.month == month && o.year == year
    }
```

`equals` fa uso di due metodi predefiniti `isInstanceOf` e `asInstanceOf`. Il primo, `isInstanceOf`, corrisponde all'operatore `instanceof` di Java e restituisce true se e solo se l'oggetto su cui è applicato è una istanza del tipo dati. Il secondo, `asInstanceOf`, corrisponde all'operatore di cast on Java: se l'oggetto è una istanza del tipo dati è

visto come tale altrimenti viene sollevata un'eccezione `ClassCastException`.

Infine, l'ultimo metodo da definire è il predicato che testa la condizione di minoranza. Fa uso di un altro metodo predefinito, `error`, che solleva una eccezione con il messaggio di errore dato.

```
def <(that: Any): Boolean = {
  if (!that.isInstanceOf[Date])
    error("cannot compare " + that + " and a Date")

  val o = that.asInstanceOf[Date]
  (year < o.year) ||
  (year == o.year && (month < o.month ||
                      (month == o.month && day < o.day)))
}
```

Questo completa la definizione della classe `Date`. Istanze di questa classe possono esser viste sia come date che come oggetti confrontabili. Inoltre, tutti e sei i predicati di confronto menzionati precedentemente sono definiti: `equals` e `<` perché appaiono direttamente nella definizione della classe `Date` e gli altri perché sono ereditati dal trait `Ord`.

I trait sono utili in molte situazioni più interessanti di quella qui mostrata, naturalmente, ma la discussione delle loro applicazioni è fuori dallo scopo di questo documento.

8 Programmazione Generica

L'ultima caratteristica di Scala che esploreremo in questo tutorial è la programmazione generica. Gli sviluppatori Java dovrebbero essere bene informati dei problemi relativi alla mancanza della programmazione generica nel loro linguaggio, un'imperfezione risolta in Java 1.5.

La programmazione generica riguarda la capacità di scrivere codice parametrizzato dai tipi. Per esempio, un programmatore che scrive una libreria per le liste concatenate può incontrare il problema di decidere quale tipo dare agli elementi della lista. Dato che questa lista è stata concepita per essere usata in contesti differenti, non è possibile decidere che il tipo degli elementi deve essere, per esempio, Int. Questo potrebbe essere completamente arbitrario ed eccessivamente restrittivo.

Gli sviluppatori Java hanno fatto ricorso all'uso di `Object`, che è il super-tipo di tutti gli oggetti. Questa soluzione è in ogni caso ben lontana dall'esser ideale, poiché non funziona per i tipi base (`int`, `long`, `float`, etc.) ed implica che molto type casts dinamico deve esser fatto dal programmatore.

Scala rende possibile la definizione delle classi generiche (e metodi) per risolvere

tale problema. Esaminiamo ciò con un esempio del più semplice container di classe possibile: un riferimento, che può essere o vuoto o un puntamento ad un oggetto di qualche tipo.

```
class Reference[T] {
  private var contents: T = _

  def set(value: T) { contents = value }
  def get: T = contents
}
```

La classe `Reference` è parametrizzata da un tipo, chiamato `T`, che è il tipo del suo elemento. Questo tipo è usato nel corpo della classe come il tipo della variabile `contents`, l'argomento del metodo `set`, ed il tipo restituito dal metodo `get`.

Il precedente codice d'esempio introduce le variabili in Scala che non dovrebbero richiedere ulteriori spiegazioni. E' tuttavia interessante notare che il valore iniziale dato a quella variabile è `_`, che rappresenta un valore di default. Questo valore di default è 0 per i tipi numerici, **`false`** per il tipo `Boolean`, `()` per il tipo `Unit` e **`null`** per tutti i tipi oggetto.

Per usare la classe `Reference`, è necessario specificare quale tipo usare per il tipo parametro `T`, che è il tipo di elemento contenuto dalla cella. Per esempio, per creare ed usare una cella che contiene un intero, si potrebbe scrivere il seguente codice:

```
object IntegerReference {
  def main(args: Array[String]) {
    val cell = new Reference[Int]
    cell.set(13)
    println("Reference contains the half of " + (cell.get * 2))
  }
}
```

Come si può vedere in questo esempio non è necessario fare il *cast* del tipo ritornato dal metodo `get` prima di usarlo come intero. Non risulta possibile memorizzare niente di diverso da un intero nella varibile `cell`, poiché è stata dichiarata per memorizzare un intero.

9 Conclusioni

Questo documento ha fornito una veloce panoramica del linguaggio Scala e presentato alcuni esempi di base. Il lettore interessato può continuare leggendo il documento *Scala By Example*, che contiene esempi molti più avanzati e consultare al bisogno la *Scala Language Specification*.