Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea in Ingegneria dell'Informazione

Caratteristiche avanzate del linguaggio Groovy

Laureando

Marco Cioccarelli

Relatore

Prof. Michele Moro

Anno accademico 2012/2013

Indice

1 Introduzione

Lo sviluppo del linguaggio Groovy è iniziato nel 2003 con un post [4] sul blog del suo creatore, James Strachan, in cui veniva delineato l'obiettivo del progetto: creare un linguaggio che unisca la flessibilità e la potenza espressiva dei linguaggi di scripting come Python e Ruby con la capacità di integrarsi perfettamente con Java e sfruttarne l'elevato numero di librerie. Nel marzo 2004 è stato standardizzato dal Java Community Process[1] con l'approvazione della Java Specification Request 241, diventando il secondo linguaggio ufficiale (dopo Java stesso) per la Java Virtual Machine. Sono state in seguito rilasciate varie versioni di sviluppo fino ad arrivare alla 1.0 nel gennaio 2007 e alla 2.0 nel luglio 2012.

L'integrazione con Java è sicuramente uno dei principali punti di forza del linguaggio. Una volta compilato, un programma scritto in Groovy viene eseguito dalla Java Virtual Machine, per la quale esso risulta indistinguibile da un programma scritto in Java. È possibile invocare codice Java da Groovy e viceversa, e le classi di un linguaggio possono estendere classi o implementare interfacce dell'altro, cosa che permette a Groovy di interagire senza problemi con le applicazioni e le librerie Java esistenti. Anche la sintassi dei due linguaggi è molto simile, tanto che la maggior parte dei sorgenti Java sono anche sorgenti Groovy validi; questo permette agli sviluppatori Java di avvicinarsi al linguaggio con facilità e rende molto semplice l'integrazione di codice Groovy all'interno di progetti Java esistenti.

Sono numerosi gli aspetti che hanno reso Java uno dei linguaggi più diffusi al mondo: portabilità, robustezza, gestione automatica della memoria attraverso l'uso di un *garbage collector*, una libreria standard ampia e completa e un ottimo supporto a caratteristiche della programmazione a oggetti come ereditarietà, incapsulamento e polimorfismo. Groovy mantiene queste caratteristiche e ne aggiunge altre che lo rendono un linguaggio dinamico, espressivo, meno verboso e più semplice da usare. Un'importante differenza rispetto a Java è l'introduzione della tipizzazione dinamica, che caratterizza la maggior parte dei linguaggi di scripting, a fianco di quella statica. Parte della sintassi di Java diventa opzionale, rendendo possibile la dichiarazione di istruzioni e metodi al di fuori di una classe e l'omissione di elementi sintattici come il punto e virgola al termine di un'istruzione. Viene integrato il supporto nativo a liste, mappe ed espressioni regolari, l'overloading degli operatori, l'uso di espressioni all'interno delle stringhe, i parametri di default per i metodi e una sintassi semplificata per i JavaBeans. La libreria standard di Java viene estesa aggiungendo nuove classi e nuove funzionalità a quelle esistenti.

Nel seguito dell'elaborato, dopo una descrizione degli aspetti generali del linguaggio, vengono trattate le caratteristiche avanzate che Groovy mette a disposizione degli sviluppatori: l'utilizzo della programmazione funzionale con le chiusure, il supporto alla metaprogrammazione e l'integrazione dei builder, uno strumento che astrae e semplifica la costruzione di strutture gerarchiche. Viene inoltre descritta brevemente la gestione del multithreading in Groovy.

[1] Il Java Community Process è il meccanismo per la standardizzazione di tecnologie per la piattaforma Java

2 Panoramica del linguaggio

2.1 Tipi di dati

Groovy è un linguaggio puramente a oggetti, nel senso che non esistono come in Java i tipi primitivi: numeri, caratteri e valori booleani sono anch'essi oggetti, rappresentati mediante le classi involucro `Integer`, `Character`, ecc. In Java la presenza dei tipi primitivi può creare problemi in alcune situazioni, ad esempio costringendo il programmatore a realizzare più versioni di un metodo, una per le istanze di classi e una per ciascuno dei tipi primitivi. In Groovy questa complicazione non esiste: per esempio il confronto di due valori può essere realizzato semplicemente con `a.compareTo(b)`, anche nel caso in cui `a` e `b` siano interi, caratteri, ecc.

Le variabili possono essere dichiarate come in Java, specificandone il tipo prima del nome. Questo approccio, che prende il nome di *tipizzazione statica*, è però solo uno dei due possibili: è consentito dichiarare una variabile senza specificarne il tipo, usando la parola chiave `def`. Il compilatore considererà una tale variabile come un'istanza della classe `Object`, mentre al momento dell'esecuzione il suo tipo sarà determinato, secondo il principio del *duck typing*, in base alle proprietà e ai metodi dell'oggetto che le viene assegnato. Sta al programmatore scegliere se fare ricorso alla tipizzazione statica o quella dinamica, a seconda della situazione.

Per quanto riguarda i tipi numerici, le classi `BigInteger` e `BigDecimal` del pacchetto `java.math`[2], che consentono di memorizzare valori numerici di precisione arbitraria, sono parte integrante del linguaggio: i letterali decimali sono di tipo `BigDecimal`, a meno che non sia specificato diversamente attraverso i suffissi `d` e `f`:

```
def a = 3.14          // BigDecimal
def b = 3.14f         // Float
def c = 3.14d         // Double
```

mentre i letterali interi sono di tipo `Integer`, `Long` o `BigInteger` a seconda del valore. L'uso di `BigDecimal` per i decimali permette di evitare inconvenienti dovuti alla limitatezza dell'intervallo numerico rappresentabile da un `Float` o da un `Double`: ad esempio l'espressione `1.1 + 0.1 == 1.2` è `true` in Groovy ma `false` in Java, dove il risultato della somma è `1.2000000000000002`. Ovviamente questo ha un prezzo, perché le operazioni svolte con istanze di `BigInteger` e `BigDecimal` sono molto più lente delle corrispondenti operazioni fra `Double` o `Float`; d'altra parte le prestazioni in Groovy, come in molti altri linguaggi di scripting, vengono a volte penalizzate a favore di una maggiore semplicità d'uso.

2.2 Stringhe ed espressioni regolari

In Groovy esistono quattro modi diversi per definire una stringa letterale, distinguibili dal tipo di carattere usato per delimitarla. Usando le virgolette singole si definisce una stringa semplice, di tipo `java.lang.String`:

[2] In Groovy vengono importate automaticamente le classi BigInteger e BigDecimal, oltre ai pacchetti groovy.lang.*, groovy.util.*, java.lang.*, java.util.*, java.net.* e java.io.*

```
'Questa è una stringa semplice'
```

Le stringhe definite con le virgolette doppie permettono l'inserimento diretto (interpolazione) di variabili ed espressioni, mediante dei segnaposto di tipo `$variabile` o `${espressione}`. Una tale stringa è detta GString ed è un'istanza della classe `GString` del pacchetto `groovy.lang`. In base al principio della *lazy evaluation*, la valutazione delle espressioni contenute nella GString non avviene finché non viene invocato su di essa il metodo `toString`, ad esempio con il comando `println`; pertanto se cambiano i valori delle espressioni cambia anche il valore della GString:

```
def n = 16
def s = "La radice quadrata di $n è ${Math.sqrt(n)}"
println s
n = 25
println s
// Output:
// La radice quadrata di 16 è 4.0
// La radice quadrata di 25 è 5.0
```

Usando le virgolette triple è possibile definire stringhe (`'''`) o GString (`"""`) che si estendono su più righe:

```
'''Stringa
su
più
righe'''
```

Infine, una stringa delimitata da barre consente di inserire un carattere *barra rovesciata* senza dover usare la sequenza `\\` , cosa che ad esempio rende più agevole la lettura e la scrittura di espressioni regolari:

```
def sequenzaDiSeiCifre = /\d{6}/
```

Le espressioni regolari sono uno strumento molto efficace per la ricerca di pattern all'interno di un testo. Per la loro gestione in Groovy si usano le classi del pacchetto `java.util.regex` e la sintassi dei pattern[3] è la stessa di Java, ma in aggiunta sono disponibili alcuni operatori che ne semplificano l'utilizzo:

- L'operatore $\sim$ (*pattern*) crea un oggetto di tipo `java.util.regex.Pattern` a partire dalla stringa alla sua destra
- L'operatore $=\sim$ (*find*) crea un oggetto di tipo `java.util.regex.Matcher` usando la stringa alla sua destra come pattern e quella alla sua sinistra come testo in cui effettuare la ricerca.
- L'operatore $==\sim$ (*match*) controlla se stringa alla sua sinistra è descritta dal pattern rappresentato dalla stringa alla sua destra e restituisce un valore booleano che indica l'esito del controllo.

[3] La sintassi dei pattern è descritta nella documentazione della classe java.util.regex.Pattern (`http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html`)

Esempi:

```
def pattern = ~/\d{6}/
pattern.matcher('123456').matches()     // true
pattern.matcher('abcdef').matches()     // false

def matcher = ('Groovy Java Ruby Scala' =~ /\w*y\b/)
print 'Parole che terminano con y: '
while (matcher.find())
  print "${matcher.group()} "
// Output: Parole che terminano con y: Groovy Ruby

'Groovy' ==~ /[A-Z][a-z]+/              // true
'groovy' ==~ /[A-Z][a-z]+/              // false
```

2.3 Collezioni

Groovy supporta nativamente le liste e le mappe, cosa che facilita lo svolgimento delle operazioni più comuni su di esse; inoltre sono disponibili gli intervalli, una struttura dati non presente in Java ma diffusa tra i linguaggi di scripting.

2.3.1 Liste

Una lista viene definita racchiudendo gli elementi che la compongono fra parentesi quadre, separati da virgole:

```
def numeriPrimi = [2, 3, 5, 7, 9]
def matrice = [[2.7, 0.8], [1.3, 5.2]]
def listaVuota = []
```

Viene così creato un oggetto di tipo `java.util.ArrayList`, sul quale è possibile effettuare operazioni di accesso, inserimento e rimozione con gli operatori `[]`, `<<`, `+` e `-`

```
numeriPrimi[1]                  // 3
numeriPrimi << 11 << 13         // inserisce 11 e 13
numeriPrimi += [17, 19, 23]     // concatena le due liste
numeriPrimi -= [2, 5]           // rimuove dalla prima lista
                                // gli elementi presenti nella seconda
println numeriPrimi
// Output: [3, 7, 9, 11, 13, 17, 19, 23]
```

È possibile usare anche indici negativi per accedere agli elementi partendo dalla fine della lista:

```
def vocali = ['a', 'e', 'i', 'o', 'u']
println vocali[-2]              // 'o'
```

In aggiunta sono disponibili gli operatori `*` (*spread*) e `*.` (*spread-dot*). Il primo permette di estrarre gli elementi di una lista per usarli singolarmente, ad esempio come parametri di un metodo: si supponga di avere un metodo `foo` che prende come argomenti cinque stringhe; si può allora invocarlo con `foo(*vocali)`, "separando" tra loro gli elementi

della lista. Il secondo consente di invocare un metodo su ogni elemento di una lista, permettendo di scrivere in maniera compatta ciò che in Java richiederebbe un ciclo `for`:

```
println vocali*.toUpperCase()
// Output: [A, E, I, O, U]
```

2.3.2 Mappe

Le mappe sono liste di coppie chiave-valore. Esse vengono definite in modo simile alle liste, separando chiavi e valori con i due punti:

```
def capitali = ['Berlino' : 'Germania', 'Il Cairo' : 'Egitto',
                'Roma': 'Italia']
def mappaVuota = [:]
```

Si può accedere agli elementi di una mappa usando le parentesi quadre o l'operatore punto; nel secondo caso, se la chiave dell'elemento a cui si vuole accedere non è un identificatore valido in Groovy, è necessario racchiuderla tra virgolette:

```
println capitali['Berlino']      // 'Germania'
println capitali.'Il Cairo'      // 'Egitto'
println capitali.Roma            // 'Italia'
```

Per effettuare modifiche possono essere usati gli operatori `<<` , `+` , `-`, `*` e `*.` come per le liste.

2.3.3 Intervalli

Gli intervalli sono liste di elementi consecutivi, che possono essere numeri, caratteri o in generale oggetti che implementano l'interfaccia `java.lang.Comparable` e possiedono i metodi `next` e `previuos`. Si definiscono con la notazione `..` oppure `..<` a seconda che si voglia includere o meno il valore finale:

```
def intervalloDiNumeri = 5..9                 // 9 compreso
def intervalloDiCaratteri = 'a'..<'g'         // g esclusa
```

Il primo estremo può anche essere maggiore del secondo, nel qual caso l'intervallo contiene gli elementi in ordine decrescente; il metodo `isReverse` permette di sapere se l'intervallo è di questo tipo:

```
def intervalloDecrescente = 10..1
println intervalloDecrescente.isReverse()     // true
```

Dato che l'interfaccia `groovy.lang.Range` estende `java.util.List`, i metodi e gli operatori disponibili per le liste, a parte quelli che ne modificano il contenuto come `add` e `remove`, possono essere usati anche con gli intervalli:

```
println intervalloDiNumeri[0]                 // 5
println intervalloDiCaratteri.contains('g')   // false
```

Gli intervalli possono essere usati nei cicli `for` e nei costrutti `switch`, dando luogo a espressioni concise e di facile lettura:

```
print 'Potenze di 2: '
for (i in 1..10)
  print "${2**i} "
// Output: Potenze di 2: 2 4 8 16 32 64 128 256 512 1024

switch (votoEsame) {
  case 27..30: println 'Ottimo'; break
  case 21..26: println 'Buono'; break
  case 18..20: println 'Sufficiente'; break
  case 0..17:  println 'Insufficiente'; break
}
```

Inoltre, usati insieme all'operatore `[]`, permettono di ottenere sottoinsiemi di liste e stringhe, una tecnica nota come *slicing* nei linguaggi di scripting:

```
def s = 'Questo è un esempio'
println s[12..19]                                  // 'esempio'
```

2.4 Metodi e classi

Come in Java, le classi vengono definite con la parola chiave `class` e possono contenere campi, costruttori e metodi. Le classi e i metodi hanno visibilità `public` di default, e come per le variabili è possibile omettere sia il tipo di dato restituito dai metodi, sostituendolo con `def`, sia i tipi dei parametri. Il metodo `main` può quindi essere scritto semplicemente come:

```
static void main(args)
```

Si possono specificare valori di default per i parametri dei metodi, come in C++, eliminando in molte situazioni la necessità di fare ricorso all'overloading dei metodi:

```
def saluto(String s = 'Hello') {
  return "$s World"
}
println saluto()                                   // 'Hello World'
println saluto('Hi')                               // 'Hi World'
```

Nell'invocazione di un metodo le parentesi possono essere omesse, come si è fatto fin qui con il metodo `println`, e si può far precedere al valore dei parametri il loro nome, con la stessa sintassi usata per le mappe. Oltre a rendere il codice più leggibile, questo permette di scriverli in qualsiasi ordine (può succedere, quando un metodo ha una lunga lista di parametri, di non ricordare l'ordine esatto con cui devono essere scritti):

```
def inviaEmail(da, a, cc, oggetto, File allegato) {
  // ...
}
inviaEmail(oggetto: 'Esempio', da: 'mrossi@abc.it', a: 'jsmith@xyz.com'
           allegato: new File('documento.pdf'), cc: '')
```

Anche il comando `return` può essere omesso, nel qual caso, se il metodo non è `void`, viene restituito il valore dell'ultima espressione presente nel corpo del metodo; in generale l'omissione del `return` può compromettere la leggibilità ed è buona norma farne uso solo in casi particolari, come quando il metodo è costituito da una sola istruzione.

Groovy implementa una propria versione dei JavaBeans , più compatta e semplice da usare. I JavaBeans sono classi che implementano l'interfaccia `java.io.Serializable`, possiedono un costruttore privo di parametri e hanno campi `private` accessibili e modificabili mediante appositi metodi *get* e *set* (detti metodi accessori). Vediamo un esempio di GroovyBean, la versione semplificata presente in Groovy:

```
class Persona implements java.io.Serializable {
  String nome
  String cognome
}
```

In Java si sarebbero dovuti definire anche un costruttore e quattro metodi, due *get* e due *set*, invece in Groovy quando un campo è dichiarato senza specificarne la visibilità vengono creati automaticamente i metodi accessori che consentono di accedere al valore del campo e di modificarlo (i cui nomi sono del tipo `getCampo` e `setCampo`), oltre ad un costruttore per impostarne il valore iniziale:

```
def persona1 = new Persona(nome: 'Mario', cognome: 'Rossi')
def persona2 = new Persona()
persona2.nome = 'John'
persona2.cognome = 'Smith'
```

In quest'ultimo esempio sembra che nelle ultime due righe si stia accedendo direttamente ai campi `nome` e `cognome`, come se fossero pubblici, ma in realtà vengono invocati i metodi `setNome` e `setCognome` generati automaticamente. Si può verificare quest'ultima affermazione implementando esplicitamente i metodi accessori:

```
class Persona implements java.io.Serializable {
  String nome, cognome
  def getNome() {
    return "Il nome di questa persona è $nome"
  }
  def setNome(String nuovoNome) {
    nome = nuovoNome
    println 'Il nome è stato cambiato'
  }
}
def persona = new Persona()
persona.nome = 'Mario'
println persona.nome
// Output:
// Il nome è stato cambiato
// Il nome di questa persona è Mario
```

2.5 Operatori

Si è visto che in Groovy è possibile usare un buon numero di operatori per lavorare con espressioni regolari e collezioni: impiegare operatori al posto dei metodi ha il vantaggio di ridurre la verbosità e rendere il codice più espressivo, che è uno degli obiettivi principali del linguaggio. Oltre a quelli già visti, esistono altri importanti operatori:

- L'operatore `?:` (*Elvis*) è un'abbreviazione dell'operatore ternario: la sintassi

```
def x = espressione ?: valoreDiDefault
```

è equivalente a

```
def x = espressione != null ? espressione : valoreDiDefault
```

- L'operatore `?.` (*safe navigation*) serve a controllare che un oggetto sia definito (cioè diverso da `null`) prima di accedere ai suoi metodi e proprietà:

  ```
  oggetto?.metodo()
  ```

  è equivalente a

  ```
  if (oggetto != null) oggetto.metodo()
  ```

- L'operatore `<=>` (*spaceship*) confronta due oggetti che implementano l'interfaccia `java.util.Comparable` e restituisce -1, 0 o 1 a seconda che il primo oggetto sia, rispettivamente, minore, uguale o maggiore del secondo.

- Gli operatori `==` e `!=` verificano l'uguaglianza di valore tra due oggetti, analogamente al metodo `equals`: dato che in Groovy non esistono i tipi primitivi, questi due operatori sarebbero di scarsa utilità se avessero lo stesso significato che hanno in Java, cioè se verificassero l'identità fra due oggetti (che si ha quando due riferimenti puntano allo stesso oggetto): un'espressione come `x == 2` sarebbe falsa anche se `x` valesse 2. Per controllare l'identità fra due oggetti bisogna ricorrere al metodo `is` della classe `Object`.

A differenza di Java, Groovy supporta l'overloading degli operatori, ossia permette al programmatore di definire il comportamento degli operatori quando vengono usati con oggetti delle classi da lui definite. Questo avviene semplicemente implementando il metodo associato ad un particolare operatore: ad esempio, i metodi `plus` (operatore `+`), `minus` (`-`), `leftShift` (`<<`), `equals` (`==`), `compareTo` (`<=>`). L'elenco completo degli operatori e i rispettivi metodi è disponibile in [6]. Vediamo ad esempio una classe `Vettore` che implementa la somma, il prodotto scalare e il prodotto per uno scalare di vettori:

```
class Vettore {
  List<Number> coeff
  Vettore(List<Number> c) {
    coeff = c
  }
  Vettore plus(Vettore v) {
    return new Vettore([coeff, v.coeff].transpose()*.sum())
  }
  Number multiply(Vettore v) {
    def prodotto = 0
    for (i in 0..<coeff.size())
      prodotto += coeff[i] * v.coeff[i]
    return prodotto
  }
  Vettore multiply(Number n) {
    return new Vettore(coeff.clone()*.multiply(n))
```

```
    }
    String toString() {
      return coeff.toString()
    }
  }

  def u = new Vettore([1,2,3])
  def v = new Vettore([4,5,6])
  println u+v                                    // [5, 7, 9]
  println u*v                                    // 32
  println u*5                                    // [5, 10, 15]
```

Vengono implementati i metodi `plus` e `multiply` per rendere possibile l'utilizzo degli operatori `+` e `*` . Questo esempio dimostra inoltre come l'uso degli operatori (in questo caso, lo *spread-dot*) permetta di svolgere operazioni relativamente complesse in poche righe di codice. Il corpo del metodo `plus` esegue la somma tra vettori impiegando i metodi `transpose` e `sum`: il primo traspone una matrice (cioè una lista di liste), e in questo caso trasforma la coppia di liste `coeff` e `v.coeff` in una lista di coppie formate dagli elementi delle due liste nelle stesse posizioni; su ogni coppia viene poi invocato, attraverso l'operatore *spread-dot*, il metodo `sum`, che restituisce la somma degli elementi di una lista, dando luogo al vettore somma. Il primo metodo `multiply` impiega un approccio più "tradizionale", dato che non esiste un metodo analogo a `sum` per il prodotto (si vedrà più avanti che è possibile ottenere un'espressione simile a quella del metodo `plus` usando le chiusure), mentre il secondo metodo `multiply` crea una copia della lista `coeff` e ne moltiplica ciascun elemento per n con l'operatore *spread-dot*.

3 Chiusure

Groovy eredita da Java il paradigma di programmazione a oggetti, che si basa sulla definizione di un insieme di oggetti che interagiscono fra di loro mediante lo scambio di messaggi, e il paradigma imperativo, usato nel corpo dei metodi, con cui si descrivono le operazioni da eseguire mediante una serie di istruzioni. In Groovy è possibile fare uso di un altro paradigma, quello funzionale, che consiste nell'organizzare un programma attraverso un insieme di funzioni, la cui applicazione consente di portare a termine le operazioni richieste.

Un concetto basilare in programmazione funzionale è quello di funzione di prima classe, una funzione che accetta altre funzioni come parametri o che restituisce un'altra funzione come risultato, in contrasto con la programmazione imperativa in cui gli input e gli output delle funzioni sono dati (numeri, stringhe, ecc.). In Groovy le funzioni di prima classe sono rappresentate da particolari oggetti, detti chiusure, che consistono in blocchi anonimi di codice che possono essere passati come parametri e restituiti da metodi e altre chiusure.

3.1 Definire una chiusura

Una chiusura viene definita racchiudendo tra parentesi graffe le istruzioni che la costituiscono:

```
{ println 'Hello world' }
```

Al contrario delle classi e dei metodi, le chiusure sono anonime. Si può però assegnare una chiusura ad una variabile ed invocarla con la stessa sintassi usata per i metodi, oppure usando il metodo `call`:

```
def hello = { println 'Hello world' }
hello()
hello.call()
```

e come per i metodi è possibile passare uno o più parametri alla chiusura: nella dichiarazione questi vengono elencati all'interno delle parentesi graffe prima delle istruzioni, dalle quali sono separate con una freccia (`->`):

```
def sommaDueNumeri = { x, y -> x + y }
println sommaDueNumeri(6, 4)              // 10
```

Come si può vedere vale anche per le chiusure la possibilità di omettere i tipi dei parametri e la parola chiave `return`. A differenza dei metodi non è però possibile specificare il tipo di dato restituito.

Quando una chiusura accetta un solo parametro, ci si può riferire ad esso usando il parametro implicito `it` e tralasciare la sua definizione esplicita, cosa che permette una scrittura abbreviata del tipo:

```
def raddoppia = { it * 2 }
println raddoppia(6)                      // 12
```

Date le somiglianze fra chiusure e metodi, appare naturale che esista un modo per utilizzare metodi esistenti come chiusure, ossia definire una chiusura in riferimento ad un metodo. Questo si può fare usando l'operatore `.&` e la sintassi `istanza.&metodo`, dove `istanza` è l'istanza di una classe, oppure con la sintassi `Classe.&metodo` per i metodi statici:

```
def zero = 0
def valeZero = zero.&equals
println valeZero(5 * 2 - 10)            // true

def convertiInBinario = Integer.&toBinaryString
println convertiInBinario(123)          // 1111011
```

3.2 Chiusure come oggetti

La differenza fondamentale fra chiusure e metodi consiste nel fatto che le chiusure sono oggetti: ogni chiusura è un'istanza della classe `groovy.lang.Closure`[4]. In quanto tali, esse possono essere assegnate ad una variabile (come si è già visto), passate come parametri ad un metodo e ad altre chiusure, restituite da un metodo o da un'altra chiusura e immagazzinate in strutture dati.

Le chiusure sono un aspetto molto importante del linguaggio, e per questo esistono numerosi metodi nella libreria standard di Groovy (costituita dalle classi e dai metodi aggiunti da Groovy alla libreria standard di Java) che accettano chiusure come parametri. Particolarmente utili e usati sono i tre metodi `times`, `upto` e `downto` della classe `Number`, superclasse di tutti i tipi numerici come `Integer`, `Double`, `BigInteger` e `BigDecimal`. Essi consentono di realizzare costrutti simili ad un ciclo `for` ma più chiari e compatti:

```
def s = 'Groovy'
3.times{ println s }
0.upto(s.length() - 1){ println s[it] }
(s.length() - 1).downto(0){ println s[it] }
```

Il metodo `times` esegue la chiusura che gli viene passata un certo numero di volte, in questo caso stampa tre volte la stringa "Groovy", mentre `upto` e `downto` effettuano operazioni in un intervallo di valori, usando rispettivamente un indice crescente e uno decrescente, rappresentato dal parametro implicito `it`; nell'esempio stampano "Groovy" un carattere alla volta, leggendo la stringa prima da sinistra a destra e poi da destra a sinistra. Notiamo la sintassi usata per l'invocazione di questi metodi: come si è detto in precedenza le parentesi tonde sono opzionali e nel caso in cui il parametro sia una chiusura è consuetudine ometterle, dato che sono già presenti le graffe a fare da delimitatori. Inoltre se un metodo accetta più parametri tra cui una chiusura, quest'ultima può essere posta al di fuori delle parentesi tonde, come nel caso di `upto` e `downto`.

Vediamo un esempio di definizione di un metodo che accetta chiusure come parametri:

```
def operazioneSuLista(List lista, Closure operazione) {
  for (elemento in lista)
```

[4]Per la precisione, ciascuna dichiarazione di chiusura all'interno di un programma comporta la creazione di una sottoclasse di `Closure`, di cui la chiusura è l'unica istanza

```
    operazione.call(elemento)
}
operazioneSuLista([1,2,3,4,5]){ print "${it ** 2} " }
operazioneSuLista(['a','b','c']){ print "${it.toUpperCase()} " }
// Output:
// 1 4 9 16 25
// A B C
```

Il metodo `operazioneSuLista` riceve in input una lista e una chiusura ed esegue la chiusura su ogni elemento della lista.

Le chiusure possono anche costituire il valore di ritorno di un metodo, come mostra il seguente esempio in cui il metodo `calcola` restituisce una diversa chiusura a seconda del parametro che riceve in ingresso:

```
def calcola(operazione) {
  switch (operazione) {
    case 'somma': return { x, y -> x + y }
    case 'differenza': return { x, y -> x - y }
    case 'prodotto': return { x, y -> x * y }
    case 'quoziente': return { x, y -> x / y }
  }
}
println calcola('somma')(5, 8)          // 13
println calcola('prodotto')(10, 2)      // 20
```

Infine un'interessante possibilità è quella di realizzare strutture dati che contengono chiusure, che possono essere liste, mappe o strutture più complesse come alberi e grafi:

```
def areeDiPoligoni = [
  'triangolo':  { base, altezza -> (base * altezza)/2 },
  'quadrato':   { lato -> lato ** 2 },
  'rettangolo': { lato1, lato2 -> lato1 * lato2 }
]
def calcolaAreaRettangolo = areeDiPoligoni['rettangolo']
println calcolaAreaRettangolo(3, 5)     // 15
```

3.3 Composizione e currying

Tra i metodi che la classe `Closure` mette a disposizione, i più importanti sono senz'altro quelli che permettono le operazioni di composizione e currying di chiusure.

La composizione di chiusure è un concetto analogo alla composizione matematica di funzioni: comporre due chiusure significa applicare una di esse al risultato dell'altra, ossia utilizzare l'output della prima come input della seconda. In questo modo si possono realizzare chiusure complesse a partire da altre più semplici. Per comporre due chiusure bisogna fare ricorso agli operatori `>>` e `<<` della classe `Closure`: l'espressione `a >> b` genera una nuova chiusura che consiste nell'applicazione della chiusura `a` seguita da quella della chiusura `b`; viceversa con `a << b` viene applicata prima `b` e poi `a`. Le chiusure risultanti sono in generale diverse, dato che la composizione non è un'operazione commutativa.

```
def f = Math.&sqrt
def g = { 2 * it }
```

```
def fCompostoG = f >> g
def gCompostoF = f << g
println fCompostoG(16)                    // 8.0
println gCompostoF(16)                    // 5.656854249492381
```

La funzione calcolata da `fCompostoG` è $2\sqrt{x}$, mentre quella calcolata da `gCompostoF` è $\sqrt{2x}$. Da notare l'uso dell'operatore `.&` per definire una chiusura che restituisce la radice quadrata come riferimento al metodo `Math.sqrt`.

La composizione può essere applicata più volte di seguito, facendo attenzione all'ordine degli operandi:

```
def listaDeiQuadrati = { x, y -> [x ** 2, y ** 2] }
def sommaElementiLista = { lista -> lista.sum() }
def modulo = listaDeiQuadrati >> sommaElementiLista >> Math.&sqrt
println modulo(5, 12)                     // 13.0
```

Il currying è una tecnica che consiste nel trasformare una funzione che accetta più parametri in un'altra che ne accetta un numero minore, fissando un valore costante per i rimanenti. Il termine deriva dal nome del matematico americano Haskell Curry, che per primo ha introdotto il concetto in matematica e in informatica. In Groovy il currying può essere applicato impiegando i metodi `curry`, `rcurry` e `ncurry` della classe `Closure`.

Il metodo `curry`, invocato su una chiusura che accetta n parametri, riceve in input un certo numero k di parametri, che deve essere minore o uguale a n, e restituisce una nuova chiusura i cui primi k parametri sono fissati ai valori dati:

```
def divisione = { x, y -> x / y }
def inverso = divisione.curry(1)
println inverso(4)                        // 0.25
```

Il comportamento dei metodi `rcurry` e `ncurry` si differenzia da quello di `curry` per l'ordine con cui vengono fissati i k parametri fra gli n della chiusura iniziale: partendo da destra nel caso di `rcurry` e da un certo indice (passato al metodo come primo parametro) nel caso di `ncurry`:

```
def convertitore = { input, k, unità -> "${input * k} $unità" }
def migliaInChilometri = convertitore.rcurry(1.60934, 'km')
println migliaInChilometri(5)             // 8.04670 km

def f = { a, b, c, d -> ((a + b) ** c) * d}
def moltiplicazione = f.ncurry(1,0,1)     // b = 0, c = 1
```

In Java per definire una funzione che è un caso particolare di un'altra, con alcuni parametri stabiliti a priori, bisogna ricorrere all'overloading dei metodi, mentre Groovy aggiunge la possibilità di usare i parametri di default, come si è visto in precedenza. Il currying può essere visto come una terza alternativa che presenta il vantaggio di poter essere utilizzata anche con metodi di classi che non si ha la possibilità di modificare, come quelle della libreria standard:

```
def riempiConUni = Arrays.&fill.rcurry(1)
def array = new int[5]
riempiConUni(array)
println array                             // [1, 1, 1, 1, 1]
```

Inoltre grazie al currying si riesce ad implementare la composizione di chiusure in modo alternativo a quello prima esposto:

```
def composizione = { f, g, x -> g(f(x)) }
def dividiParole = { it.split(' ') }
def ordina = { it.sort() }
def listaParole = composizione.curry(dividiParole, ordina)
println listaParole('the quick brown fox jumps over the lazy dog')
// Output: [brown, dog, fox, jumps, lazy, over, quick, the, the]
```

3.4 Chiusure e ambiti di visibilità

Fino a questo momento sono state usate chiusure che ricevono un insieme di parametri, elencati all'inizio della dichiarazione della chiusura, eseguono delle operazioni su di essi e restituiscono un risultato. Oltre ai parametri in ingresso, però, una chiusura può accedere anche a tutti gli elementi che si trovano nel suo ambito di visibilità (in inglese *scope*), che definisce quali variabili locali e campi possono essere letti e modificati, quali metodi possono essere invocati e il valore del riferimento `this`.

Cominciamo con un semplice esempio di una chiusura che si limita a stampare il valore di una variabile locale:

```
def x = 100
def stampa = { println x }
stampa()
```

L'esecuzione avviene senza problemi, dato che la variabile `x` si trova nell'ambito di visibilità della chiusura sia al momento della dichiarazione sia in quello dell'esecuzione. A questo proposito è bene sottolineare che le parentesi graffe indicano la dichiarazione di una chiusura, non la sua esecuzione: negli esempi visti finora le due operazioni avvenivano consecutivamente e quindi la differenza non assumeva particolare importanza. Vediamo un esempio simile al precedente in cui però la chiusura viene eseguita all'interno del metodo di una classe:

```
class Test {
  def esegui(Closure c) {
    c.call()
  }
}
def x = 100
def test = new Test()
test.esegui{ println x }
```

Anche questo codice viene eseguito correttamente, stampando il valore di `x`, ma il motivo non è immediatamente comprensibile: come può un metodo eseguire un'operazione che richiede l'accesso ad una variabile quando questa non è presente nell'ambito di visibilità del metodo stesso? Un esempio leggermente modificato come quello che segue, infatti, comporta la segnalazione di un errore da parte dell'ambiente di esecuzione:

```
class Test {
  def esegui() {
    println x
```

```
  }
}
def x = 100
def test = new Test()
test.esegui()
```

La differenza tra i due esempi consiste nella presenza o meno della chiusura: il motivo per cui il primo esempio funziona correttamente risiede in una proprietà molto importante delle chiusure: esse "ricordano" il contesto in cui sono state create, ossia mantengono dei riferimenti agli elementi (variabili e metodi) presenti nell'ambito di visibilità al momento della definizione. Lo stesso temine *chiusura* deriva da questa caratteristica: le variabili definite al di fuori di una chiusura che vengono utilizzate nel suo corpo (dette *variabili libere*), al momento della creazione della chiusura vengono catturate, o racchiuse, dalla chiusura stessa.

L'insieme degli elementi accessibili da una chiusura è mostrato in questo esempio[5]:

```
class UnaClasse {
  def campo = 'unCampo'
  def metodo() {
    return 'unMetodo'
  }
  def testChiusura(parametro) {
    def variabileLocale = 'unaVariabileLocale'
    return { println "${this.class.name} $campo ${metodo()} " +
                     "$parametro $variabileLocale" }
  }
}
def c = new UnaClasse()
def chiusura = c.testChiusura('unParametro')
chiusura.call()
// Output: UnaClasse unCampo unMetodo unParametro unaVariabileLocale
```

Per la gestione dei diversi ambiti di visibilità che possono essere presenti in situazioni particolari, come nel caso di una chiusura posta all'interno di un'altra, ogni chiusura possiede, oltre a `this`, anche le variabili implicite `owner` e `delegate`. Come sempre, `this` si riferisce alla classe che contiene la chiusura; `owner` corrisponde all'oggetto che contiene la chiusura: è un riferimento alla chiusura esterna, se presente, altrimenti coincide con `this`; `delegate` di default ha lo stesso valore di `owner`, ma può essere modificato per farlo puntare a qualunque oggetto. Per risolvere un identificatore (un nome di variabile o di metodo) all'interno di una chiusura viene controllato prima `this`, quindi `owner` e infine `delegate`, anche se quest'ordine può essere cambiato con appositi metodi della classe `Closure`. Particolarmente importante è la variabile `delegate`, di cui vedremo un utilizzo nella parte dedicata alla metaprogrammazione.

[5]La sintassi `variabile.class.name` usata nell'esempio fa uso della riflessione, una tecnica che verrà introdotta in seguito; basti sapere che serve ad ottenere il nome della classe della variabile

3.5 Esempi di utilizzo

3.5.1 Iterazione su collezioni

Una necessità frequente in programmazione è quella di eseguire operazioni su tutti gli elementi di una collezione. In Java questo si può fare ricorrendo agli iteratori o ai costrutti `for` e `for each`; entrambe le soluzioni però, oltre ad essere relativamente verbose per un'operazione così comune, hanno il difetto di non rendere immediatamente chiaro a chi legge il codice il tipo di compito che si vuole svolgere. In Groovy le classi che rappresentano collezioni (liste, mappe, intervalli e stringhe, che possono essere viste come liste di caratteri) mettono a disposizione numerosi metodi che accettano chiusure e forniscono una soluzione elegante e compatta ai problemi di programmazione più comuni:

- `each` e `reverseEach` eseguono un'iterazione su una collezione rispettivamente dal primo elemento all'ultimo e viceversa, passando ogni elemento ad una chiusura:

```
def m = ['x': 3, 'y': 4, 'z': 5]
m.each{ chiave, valore -> println "$chiave vale $valore" }
// Output: x vale 3
//         y vale 4
//         z vale 5
```

- `collect` applica una chiusura a ciascun elemento di una collezione e restituisce una collezione costituita dagli output della chiusura. Usando questo metodo è possibile implementare in modo compatto il prodotto scalare di vettori, come si era accennato nel paragrafo 2.5:

```
def u = [1, 2, 3]
def v = [4, 5, 6]
def prod = [u, v].transpose().collect{ x, y -> x * y }.sum()
println prod                                  // 32
```

- `find`, `findAll` e `findIndexOf` applicano agli elementi di una collezione una chiusura che consiste in un test sull'elemento, ossia che restituisce un valore booleano `true` o `false` a seconda dell'elemento. Detta C la collezione degli elementi che passano il test, `find` restituisce il primo elemento di C, `findAll` l'intera collezione C e `findIndexOf` l'indice del primo elemento di C all'interno della collezione iniziale:

```
def l = [12, 9, 10, 6, 2, 7, 14]
println l.find{ it < 8 }                      // 6
println l.findAll{ it < 8 }                   // [6, 2, 7]
println l.findIndexOf{ it < 8 }               // 3
```

- `every` e `any` restituiscono `true` se la condizione descritta dalla chiusura è soddisfatta, rispettivamente, da tutti e da almeno un elemento di una collezione, altrimenti restituiscono `false`:

```
def m = [3: 9, 4: 16, 5: 25]
println m.every{ chiave, valore -> chiave ** 2 == valore }   // true
println m.any{ chiave, valore -> chiave % 2 == 0 }           // true
```

- `inject` riceve come parametri, oltre ad una chiusura, un valore iniziale che viene passato alla chiusura insieme al primo elemento di una collezione durante la prima iterazione; il risultato della prima iterazione viene quindi passato alla seconda insieme al secondo elemento, e così via:

```
def l = [12, 9, 10, 6, 2, 7]
println l.inject(0){ somma, i -> somma += i }      // 46
```

- `split` suddivide gli elementi di una collezione in due collezioni: la prima contiene gli elementi che soddisfano la condizione descritta dalla chiusura, la seconda i rimanenti elementi:

```
def (pari, dispari) = (1..10).split{ it % 2 == 0 }
println pari                                  // [2, 4, 6, 8, 10]
println dispari                               // [1, 3, 5, 7, 9]
```

3.5.2 Gestione di risorse

Un altro tipo di operazioni svolte frequentemente è la gestione di risorse, come file, connessioni a database e connessioni di rete. Esse sono accomunate dal fatto che, per poterle utilizzare, è necessario eseguire delle operazioni "di contorno" a quelle a cui si è davvero interessati, ossia l'apertura e la chiusura della risorsa. Entrambe le operazioni possono non andare a buon fine per diversi motivi, e lanciare quindi un'eccezione, oppure un'eccezione può essere lanciata durante l'elaborazione della risorsa, impedendo l'esecuzione dell'operazione di chiusura: bisogna quindi racchiudere le istruzioni all'interno di un blocco `try-catch-finally` e gestire le varie eccezioni.

In Groovy questo non è più necessario, grazie all'uso delle chiusure insieme ai metodi delle classi usate per gestire file, database e connessioni. Vediamo degli esempi con i file: la classe `java.io.File` viene estesa da Groovy con molti metodi per effettuare le operazioni più frequenti. Ad esempio è possibile leggere il contenuto di un file riga per riga semplicemente con:

```
new File('file.txt').eachLine{ riga -> println riga }
```

Il metodo `eachLine` si assiucura che il file venga aperto e chiuso correttamente, permettendo allo sviluppatore di concentrarsi sulle altre operazioni da eseguire. Per operazioni di lettura più generali si può usare il metodo `withReader`, che fornisce un oggetto `BufferedReader` e lo chiude quando la chiusura termina:

```
new File('file.txt').withReader{ reader ->
  println reader.getText()
}
```

L'esempio stampa l'intero contenuto del file. Analogamente è possibile scrivere su un file con i metodi `withWriter` e `withWriterAppend`, per sovrascrivere o aggiungere dati alla fine di un file:

```
def file = new File('radici_quadrate.txt')
file.withWriter{ writer ->
   (0..100).each { writer.write(Math.sqrt(it) + '\n') }
```

```
}
file.withWriterAppend{ writer ->
    (101..225).each { writer.write(Math.sqrt(it) + '\n') }
}
```

Si può ottenere un sottoinsieme delle righe di un file che soddisfano una certa proprietà con il metodo `filterLine`, che applica una chiusura ad ogni riga filtrando quelle che non soddisfano la condizione da essa specificata. Nell'esempio vengono stampate solo le righe del file precedentemente creato che terminano con ".0", ottenendo i numeri da 0 a 15:

```
def file = new File('radici_quadrate.txt')
println file.filterLine{ it.endsWith('.0') }
```

4 Metaprogrammazione

Per metaprogrammazione si intende la scrittura di programmi, detti metaprogrammi, che possono esaminare, generare o modificare altri programmi, inclusi se stessi. Un compilatore è un classico esempio di metaprogramma. Il linguaggio in cui il metaprogramma è scritto è detto metalinguaggio (un esempio sono le macro del linguaggio C), e si parla di *riflessione* se esso coincide con il linguaggio del programma modificato, come nel caso di Groovy.

La metaprogrammazione permette di ridurre la quantità di codice necessaria ad implementare o modificare una certa funzionalità, aggirare le limitazioni di un linguaggio di programmazione e dotare un programma della capacità di rispondere in modo molto flessibile a nuove situazioni che si possono verificare durante l'esecuzione. Risulta particolarmente utile per gestire le fasi di testing e profiling di un'applicazione, effettuare modifiche strutturali su grandi progetti e realizzare programmi in grado di applicare a se stessi pacchetti di aggiornamento durante l'esecuzione, sostituendo dinamicamente parti del proprio codice.

4.1 Ispezionare una classe

Tra le operazioni svolte nell'ambito della metaprogrammazione, la più semplice consiste nell'ottenere informazioni sulle proprietà degli oggetti che si stanno manipolando, come i campi disponibili, i metodi supportati e le interfacce implementate. A questo scopo in Groovy si usa l'API Reflection di Java, fornita dal pacchetto `java.lang.reflect` e dalla classe `java.lang.Class`, il cui utilizzo risulta semplificato dalla sintassi dei GroovyBeans (introdotta nel paragrafo 2.4) e dall'impiego delle chiusure.

Prima di tutto è necessario ottenere un riferimento all'oggetto `Class` della classe o dell'istanza a cui siamo interessati. Questo si può fare in due modi, a seconda che al momento dell'esecuzione si abbia a disposizione un'istanza della classe o il suo nome qualificato: nel primo caso si accede alla proprietà[6] `class` dell'istanza, nel secondo si usa il metodo `forName` della classe `Class`:

```
def classeString = 'abc'.class
def classeInteger = Class.forName('java.lang.Integer')
println "$classeString.name, $classeInteger.name"
// Output: java.lang.String, java.lang.Integer
```

Se al momento della scrittura del programma si ha già a disposizione il nome della classe esso può essere usato direttamente:

```
println ArrayList.name                    // java.util.ArrayList
```

Una volta ottenuto il riferimento all'oggetto `Class`, si possono acquisire informazioni sui suoi campi, metodi e costruttori mediante le proprietà `fields`, `methods` e `constructors`, che restituiscono rispettivamente un array di oggetti `Field`, uno di `Method` e uno di

[6] Nell'ambito dei JavaBeans (e dei GroovyBeans), una proprietà è definita come l'insieme di un campo `private` e dei suoi metodi accessori *get* e *set*. La sintassi dei GroovyBeans permette di scrivere `x.class`, `x.methods`, ecc. al posto di `x.getClass()`, `x.getMethods()`, ecc.

`Constructor`, le cui caratteristiche possono essere analizzate grazie alle loro numerose proprietà, tra cui `name`, `type`, `returnType` e `parameterTypes`:

```
Math.fields.each{ println "$it.name, di tipo $it.type" }
// Output:
// E, di tipo double
// PI, di tipo double

Object.methods.collect{
  if (it.returnType.name == 'void')
    "Il metodo $it.name non restituisce un valore"
  else
    "Il metodo $it.name restituisce un $it.returnType.name"
}.unique().each{ println it }
// Output:
// Il metodo wait non restituisce un valore
// Il metodo equals restituisce un boolean
// Il metodo toString restituisce un java.lang.String
// Il metodo hashCode restituisce un int
// Il metodo getClass restituisce un java.lang.Class
// Il metodo notify non restituisce un valore
// Il metodo notifyAll non restituisce un valore

ArrayList.constructors.each{
  it.parameterTypes.each{ println it.name }
}
// Output:
// java.util.Collection
// int
```

È inoltre possibile conoscere il pacchetto a cui la classe appartiene, la sua superclasse, le interfacce implementate e sapere se l'oggetto `Class` è una classe o un'interfaccia:

```
println String.package.name                  // java.lang
println String.superclass.name               // java.lang.Object

String.interfaces.each{ println it.name }
// Output:
// java.io.Serializable
// java.lang.Comparable
// java.lang.CharSequence

println String.isInterface()                 // false
println Comparable.isInterface()             // true
```

4.2 Aggiungere proprietà e metodi

In Groovy ogni oggetto, oltre ad estendere, come in Java, la classe `Object` o una sua sottoclasse, implementa implicitamente l'interfaccia `groovy.lang.GroovyObject`, che, tra gli altri, mette a diposizione i metodi `getMetaClass` e `setMetaClass` per ottenere e modificare la metaclasse associata all'oggetto. Le metaclassi sono il corrispettivo modificabile degli oggetti `Class` visti nel paragrafo precedente: Java permette di esaminare le

caratteristiche di una classe ma non di modificarle, e il comportamento di un programma è stabilito al momento della compilazione in modo definitivo.

Al contrario, Groovy consente di aggiungere proprietà e metodi alle classi esistenti o a singole istanze. Ciò significa ad esempio che si possono apportare modifiche a classi di cui per qualche motivo non si ha la possibilità di modificare il sorgente: è questo il meccanismo usato da Groovy per estendere con nuovi metodi le classi della libreria standard di Java, come i metodi `times` e `upto` della classe `Number`.

Per accedere alla metaclasse associata ad una classe si usa una sintassi simile a quella per gli oggetti `Class`. Usando il riferimento alla metaclasse si possono poi aggiungere proprietà[7] e metodi d'istanza, semplicemente scrivendone il nome e assegnandogli un valore, che, nel caso dei metodi, corrisponde ad un oggetto chiusura. Dato che stiamo modificando tutte le istanze di una classe, le proprietà devono essere statiche: questo si può specificare aggiungendo il qualificatore `static` prima del nome della proprietà o del metodo:

```
Math.metaClass.static.SEZIONE_AUREA = 1.6180339887498948482d
println Math.SEZIONE_AUREA ** 2 == Math.SEZIONE_AUREA + 1      // true

String.metaClass.nascondi = { replaceAll(/./, '*') }
println 'password'.nascondi()           // ********

String.metaClass.static.stampaAlfabeto = { println 'a'..'z' }
String.stampaAlfabeto()                 // [a, b, c, d, e, ...
```

In questo modo vengono modificate tutte le istanze della classe. Se invece si desidera cambiare il comportamento di una sola istanza, Groovy mette a disposizione la possibilità di creare una metaclasse specifica per quell'istanza:

```
def a = 28
a.metaClass.divisori = [1, 2, 4, 7, 14, 28]
def b = 34
b.metaClass.divisori = [1, 2, 17, 34]
println a.divisori                      // [1, 2, 4, 7, 14, 28]
println b.divisori                      // [1, 2, 17, 34]
println 12.divisori                     // MissingPropertyException
```

Quando si aggiunge un metodo con la stessa firma (nome del metodo, numero e tipo dei parametri) di uno già esistente impiegando l'operatore `=`, esso viene sovrascritto. Per avere la certezza che il metodo aggiunto non vada a sovrascriverne uno esistente si deve usare l'operatore `<<` , che segnala un errore se il metodo è già presente. Questo risulta particolarmente utile per aggiungere costruttori, che sono tipicamente presenti in più versioni con diversi parametri; un costruttore può essere aggiunto mediante la sintassi `metaClass.constructor`:

```
String.metaClass.constructor << { Integer x ->
  new String(Integer.toString(x))
}
def s = new String(10)
```

[7] Aggiungendo un campo vengono generati, come sempre, i metodi accessori *get* e *set*, pertanto il risultato è l'aggiunta di una proprietà

```
println s                                    // 10

Integer.metaClass.constructor << { String x ->
  new Integer(Integer.parseInt(x))
}                                            // GroovyRuntimeException
```

Il primo costruttore viene aggiunto correttamente perché nella classe `String` non ne è presente uno che accetta un `Integer`, il secondo invece, che esiste già nella classe `Integer`, non viene aggiunto e viene lanciata un'eccezione.

4.3 Categorie e mixin

La possibilità di aggiungere metodi e proprietà ad una classe o ad una particolare istanza è sicuramente uno strumento molto utile e potente, ma può anche avere effetti indesiderati, perché modificando una metaclasse viene alterato il comportamento di tutti gli oggetti associati ad essa. Ad esempio si potrebbe voler aggiungere un metodo ad una classe per poterlo usare in una certa parte di un'applicazione, ma questo potrebbe creare problemi nelle altre parti, dove la presenza del metodo non è attesa.

Una soluzione a questo problema è costituita dall'impiego delle categorie. Una categoria è una classe che contiene un insieme di metodi statici che possono essere resi disponibili alle istanze di un'altra classe in un particolare e ben delimitato blocco di codice. Al di fuori di esso il comportamento delle istanze della classe rimarrà inalterato. Inoltre le categorie permettono di aggiungere metodi e proprietà a più classi contemporaneamente senza dover modificare le metaclassi una per una; l'unica limitazione consiste nel fatto che non è possibile aggiungere metodi statici.

Vediamo per prima cosa come usare una categoria già disponibile in Groovy, che permette di effettuare calcoli su durate di tempo:

```
import groovy.time.*
use(TimeCategory) {
  println 6.months.from.now                  // 2013-07-21
  println 3.weeks - 12.days + 3.hours        // 9 days, 3 hours
  println 20.hours + 40.minutes              // 20 hours, 40 minutes
}
```

La categoria `TimeCategory` aggiunge varie proprietà agli oggetti `Integer` in modo da consentire una scrittura naturale di espressioni con durate di tempo. Il blocco di codice in cui le proprietà e i metodi della categoria sono disponibili è segnalato dalla presenza del metodo `use` dell'interfaccia `GroovyObject`, che a tutti gli effetti può essere visto come una parola chiave allo stesso livello di `if`, `while` e `for`. Esso riceve come parametri una o più categorie insieme ad una chiusura che rappresenta il blocco di codice in cui si vogliono rendere disponibili le categorie.

Per implementare una categoria si definisce una classe contenente i metodi che si vogliono aggiungere (o sovrascrivere), che devono essere tutti statici, anche se rappresentano metodi d'istanza: il primo parametro di ciascun metodo indica l'oggetto sul quale il metodo è stato invocato, e il suo tipo la classe a cui il metodo è reso disponibile. Ad esempio, il metodo

```
static Integer lunghezza(String istanza)
```

è disponibile solo agli oggetti nel blocco `use` che sono di tipo stringa, e in un'invocazione del tipo `'abc'.lunghezza()` il riferimento ad `'abc'` sarà contenuto nella variabile `istanza`. Per aggiungere una proprietà è necessario implementare i metodi accessori *get* e *set*.

Il seguente esempio[8] mostra una categoria per l'esecuzione di operazioni in aritmetica modulare, sfruttando tra l'altro l'overloading degli operatori. Si può vedere come al di fuori del blocco `use` i metodi non sono più disponibili:

```
class CategoriaModulo12 {
  static Integer plus(Integer istanza, Integer n) {
    return (istanza.toLong() + n.toLong()) % 12
  }
  static Integer minus(Integer istanza, Integer n) {
    return ((istanza.toLong() - n.toLong()) + 12) % 12
  }
  static Integer inModulo12(Integer istanza) {
    return istanza % 12
  }
}
use(CategoriaModulo12) {
  println 10 + 4                              // 2
  println 3 - 6                               // 9
  println 27.inModulo12()                     // 3
}
println 10 + 4                                // 14
println 3 - 6                                 // -3
println 27.inModulo12()                       // MissingMethodException
```

Un'alternativa alle categorie è rappresentata dai mixin. I mixin permettono di rendere disponibili ad una classe i metodi di una qualsiasi altra classe, non necessariamente una categoria; i metodi aggiunti rimangono disponibili per tutta la durata dell'esecuzione, come se si stesse modificando la metaclasse, ed è possibile limitarsi ad aggiungere metodi ad una sola particolare istanza. Un mixin viene specificato usando l'omonimo metodo su una classe o un'istanza e passando come argomenti le classi i cui metodi si vogliono rendere disponibili:

```
class NomeFile {
  def getEstensione() {
    return this.find(~/\..[^.]*$/)
  }
}
String.mixin(NomeFile)
println 'documento.txt'.estensione             // .txt
```

Entrambi i metodi `use` e `mixin` consentono di elencare più di una classe da cui ricavare i metodi, dando luogo ad una forma di ereditarietà multipla. Nel caso in cui un metodo sia presente con lo stesso nome e lista dei parametri in più categorie, viene data precedenza all'ultima categoria nell'elenco:

[8] Si è usata la conversione a `Long` per evitare ricorsioni infinite

```
class CategoriaA {
  static void test(String istanza) {
    println 'Metodo test di CategoriaA'
  }
}
class CategoriaB {
  static void test(String istanza) {
    println 'Metodo test di CategoriaB'
  }
}
use(CategoriaA, CategoriaB) {
  'abc'.test()                        // Metodo test di CategoriaB
}
```

Lo stesso vale per i mixin.

4.4 Intercettare l'invocazione di un metodo

L'intercettazione dell'invocazione di metodi è un altro importante aspetto della metaprogrammazione. Essa permette di eseguire azioni prima e dopo l'invocazione di ogni metodo o di un particolare insieme di metodi, oppure di eseguire altre operazioni al posto del metodo stesso, ad esempio invocando un metodo diverso. Questo risulta molto utile per il debugging di un'applicazione o per controllare in maniera "centralizzata" il comportamento di un classe, tenendo traccia di tutte le invocazioni; si possono inoltre decidere le azioni da intraprendere nel caso il metodo invocato non esista. Questo tipo di approccio alla risoluzione di problemi che interessano molte parti di un'applicazione (come il logging), prende il nome di programmazione orientata agli aspetti, ed è un altro dei paradigmi di programmazione implementati in Groovy.

Sono possibili due tecniche per l'intercettazione della chiamata a un metodo: implementare l'interfaccia `GroovyInterceptable` (sottointerfaccia di `GroovyObject`), con il suo metodo `invokeMethod`, oppure aggiungere quest'ultimo alla classe sfruttando le metaclassi.

Se una classe implementa `GroovyInterceptable` allora `invokeMethod` viene chiamato ogni volta che avviene un'invocazione su un oggetto della classe, sia che il metodo invocato esista sia che non esista. Si supponga ad esempio di voler implementare dei controlli prima dell'invocazione di alcuni metodi di una classe: si potrebbero aggiungere delle istruzioni all'inizio di ogni metodo, ma questo porterebbe ad avere duplicazione di codice e favorirebbe la comparsa di errori. Vediamo come invece l'utilizzo di `invokeMethod` consenta di effettuare i controlli in modo unificato:

```
class Test implements GroovyInterceptable {
  def metodo1(p) {
    System.out.println "Metodo1 chiamato con parametro: $p"
  }
  def metodo2(p) {
    System.out.println "Metodo2 chiamato con parametro: $p"
  }
  def invokeMethod(String nome, parametri) {
    System.out.print "Invocato $nome => "
```

```
    def metodo = Test.metaClass.getMetaMethod(nome, parametri)
    if (metodo != null)
      if (parametri.size() == 1)
        metodo.invoke(this, parametri)
    else
      System.out.println 'Il metodo richiede un parametro'
  else
    System.out.println 'Metodo mancante'
  }
}
def test = new Test()
test.metodo1(10)
test.metodo2()
test.metodo3(10)
// Output:
// Invocato metodo1 => Metodo1 chiamato con parametro: 10
// Invocato metodo2 => Il metodo richiede un parametro
// Invocato metodo3 => Metodo mancante
```

Quando un metodo dell'oggetto `test` viene invocato, il controllo passa a `invokeMethod`, che riceve come argomenti il nome del metodo (una stringa) e l'array dei parametri. Tramite il metodo `getMetaMethod` della metaclasse associata alla classe `Test` si ottiene un riferimento ad un oggetto di tipo `MetaMethod` che rappresenta il metodo chiamato: se questo riferimento è `null` significa che il metodo non esiste e viene stampato un messaggio di errore, altrimenti viene controllato il numero dei parametri, e, se questo è corretto, il metodo viene invocato usando `invoke` sull'oggetto `MetaMethod`, passando un riferimento all'oggetto sul quale il metodo deve essere invocato (`this`) e i parametri. Notiamo che si è impiegato `System.out.println` perché `println` è un metodo dell'interfaccia `GroovyObject` implementata da tutti gli oggetti, compreso `test`, e pertanto il suo uso comporterebbe ogni volta l'invocazione di `invokeMethod`; inoltre, se venisse usato all'interno di `invokeMethod` stesso si avrebbe una ricorsione infinita: in generale bisogna fare attenzione a non invocare metodi della stessa classe nel corpo di `invokeMethod`.

In alternativa è possibile inserire il metodo `invokeMethod` usando le metaclassi. Questo approccio è necessario quando non si ha la possibilità di modificare la classe oppure se si desidera iniziare a intercettare i metodi durante l'esecuzione in base ad un evento o allo stato dell'applicazione. Nel seguente esempio viene introdotto il controllo sulle operazioni effettuate fra oggetti `Integer` in modo che né gli operandi né il risultato siano negativi:

```
Integer.metaClass.invokeMethod = { nome, parametri ->
  if (parametri.any{ it < 0 })
    throw new IllegalArgumentException('Operandi negativi')
  def risultato = Integer.metaClass.getMetaMethod(nome, parametri)
                  .invoke(delegate, parametri)
  if (risultato < 0)
    throw new ArithmeticException('Risultato negativo')
  return risultato
}
println 1 + 2                          // 3
println 4 * (-3)                       // IllegalArgumentException
```

```
println 5 - 6                          // ArithmeticException
```

Per controllare i parametri viene usato il metodo `any` delle collezioni (paragrafo 3.5.1), e se il controllo viene superato il metodo viene invocato con `invoke` sull'oggetto `MetaMethod`, come nell'esempio precedente; l'unica differenza consiste nel fatto che bisogna usare `delegate` al posto di `this` perché quest'ultimo si riferisce alla classe che contiene la chiusura, mentre `delegate` è un riferimento all'oggetto i cui metodi si stanno intercettando (in questo caso, gli oggetti `1`, `4` e `5`).

Infine è disponibile un'alternativa a `invokeMethod`: se si è interessati a gestire solo le chiamate a metodi non presenti nella classe si può implementare un altro metodo, `methodMissing`. Quando su un oggetto di una classe che implementa `methodMissing` viene invocato un metodo esistente, questo viene chiamato direttamente, evitando il costo in termini di tempo di un'invocazione intermedia a `invokeMethod`.

4.5 Valutazione dinamica di espressioni

Un ultimo esempio delle possibilità di metaprogrammazione offerte dal linguaggio è la valutazione di espressioni scritte in Groovy contenute all'interno di stringhe o file. Questa caratteristica è utile ad esempio per modificare parti di un programma che richiedono cambiamenti frequenti senza dover ricompilare ogni volta l'intera applicazione, oppure per permettere l'inserimento di espressioni da parte dell'utente, ad esempio in un'applicazione per il calcolo di funzioni matematiche, oppure ancora per effettuare aggiornamenti al codice durante l'esecuzione, caricando dinamicamente nuove versioni di alcune classi. Naturalmente una simile funzionalità può dare origine a problemi di sicurezza, e in un'applicazione reale è necessario implementare i dovuti controlli prima di eseguire le istruzioni contenute in un'espressione.

La valutazione di un'espressione è eseguita da un oggetto di tipo `GroovyShell`, passando al suo metodo `evaluate` la stringa o il file da valutare. Il metodo `evaluate` restituisce il valore dell'espressione, che può quindi essere assegnato ad una variabile ed utilizzato nel resto del programma:

```
def shell = new GroovyShell()
shell.evaluate('println 2 * 3')                    // 6
def lista = shell.evaluate('[1,2,3,4]')
println lista.sum()                                // 10
```

Per fare in modo che durante la valutazione dell'espressione vengano usati i valori di variabili presenti nel resto del codice bisogna creare un oggetto `Binding` (che può essere pensato come una mappa che ha delle stringhe come chiavi e degli oggetti come valori), assegnargli delle proprietà che rappresentano il valore delle variabili e passarlo come argomento al costruttore di `GroovyShell`:

```
def binding = new Binding()
binding.base = 10
binding.altezza = 3
def shell = new GroovyShell(binding)
shell.evaluate('println ((base * altezza)/2)')     // 15
```

Il codice può essere semplificato nel caso in cui si debba valutare un'espressione con non più di tre variabili: in questa situazione la classe `groovy.util.Eval` consente di eseguire l'operazione di valutazione semplicemente come:

```
Eval.me('println "Hello World"')
```

Il metodo `me` viene usato in assenza di variabili, altrimenti si fa ricorso ai metodi `x`, `xy` e `xyz`, assegnando alle variabili nell'espressione i nomi x, y e z, ed elencandone i valori prima della stringa da valutare:

```
println Eval.x(1, 'x ** 2')                    // 1
println Eval.xy(1, 2, '(x + y) ** 2')          // 9
println Eval.xyz(1, 2, 3, '(x + y + z) ** 2')  // 36
```

Per creare una classe partendo da una stringa o da un file la procedura è simile a quella per le espressioni: `GroovyShell` viene sostituito da `GroovyClassLoader` e il metodo `evaluate` da `parseClass`, il quale restituisce un oggetto di tipo `Class` di cui si possono creare nuove istanze con `newInstance`:

```
def gcl = new GroovyClassLoader()
def classe = gcl.parseClass('''
class Test {
  def metodo() { println "test" }
}
''')
def istanza = classe.newInstance()
istanza.metodo()                               // test
```

5 I builder

5.1 Introduzione ai builder

Nello sviluppo di un'applicazione, realizzare una struttura basata sul paradigma a oggetti che sia al tempo stesso corretta, estensibile e flessibile è un'attività complessa, perché bisogna tener conto di fattori non direttamente collegati alle specifiche funzionali, come la scomposizione in oggetti del sistema da implementare e la definizione delle relazioni e della gerarchia fra gli oggetti stessi. Inoltre durante il suo ciclo di vita, la maggior parte delle applicazioni è soggetta a continue modifiche per adattarsi a cambiamenti di specifiche e rispondere correttamente a situazioni non previste in precedenza. Per questo, nel progettare applicazioni articolate, è diffusa la prassi di affidarsi a soluzioni generali, riutilizzabili e indipendenti dal linguaggio usato che, nel corso degli anni, si sono rivelate particolarmente efficaci. Queste soluzioni, o modelli di soluzione, detti *design patterns*, indicano allo sviluppatore quali sono le migliori pratiche da seguire nella creazione di un'applicazione.

Il concetto di design pattern è nato nell'ambito dell'architettura, ed è stato in seguito adottato nell'ingegneria del software. Nel tempo sono stati formalizzati e classificati numerosi pattern, che possono essere suddivisi in categorie in base al tipo di problema che permettono di affrontare: esistono, tra gli altri, pattern per l'implementazione di algoritmi, per l'esecuzione e la sincronizzazione di operazioni in parallelo, per la definizione ad alto livello delle strutture che compongono un'applicazione e per la costruzione di oggetti complessi.

A quest'ultima categoria appartiene il design pattern *builder*, che consiste nell'astrazione dei passi necessari alla costruzione di un oggetto complesso, in modo che diverse implementazioni dei passi di costruzione possano dare origine a diversi tipi di oggetti. Si tratta, in altre parole, di separare la costruzione di un oggetto, che può essere molto complessa e articolata, dalla sua rappresentazione, in modo da poter riutilizzare il processo di costruzione per la generazione di molti oggetti. Il pattern builder viene implementato realizzando un oggetto, il builder, che ne costruisce un altro, il prodotto. La complessità dei prodotti può derivare dal fatto che essi possiedono uno stato interno complicato (come nel caso di un *parser*) oppure perché sono a loro volta costituiti da oggetti che presentano legami di dipendenza fra di loro, spesso di tipo gerarchico. Esempi di oggetti del secondo tipo sono le strutture ad albero, utilizzate in molte applicazioni, come i filesystem, costituiti da una gerarchia di file e cartelle, le interfacce grafiche, realizzate mediante componenti racchiuse all'interno di contenitori, o i file XML, HTML e Json, che memorizzano informazioni in maniera annidata.

Nonostante la costruzione di strutture gerarchiche sia un'attività molto frequente nella realizzazione di applicazioni, la maggior parte dei linguaggi di programmazione non fornisce strumenti che permettano di semplificare questa operazione. Molto spesso una struttura gerarchica viene implementata creando un insieme di oggetti, definendo le loro proprietà e usando dei metodi per stabilire quale sia l'oggetto radice e assegnare, uno ad uno, gli oggetti figli a ciascun genitore. Questo da una parte porta ad una duplicazione sistematica della logica necessaria alla costruzione dell'oggetto, dall'altra

rende difficile a chi legge il codice capire le relazioni di dipendenza tra gli oggetti che compongono la struttura (ad esempio, capire a quale livello della gerarchia si trova un determinato oggetto).

Groovy permette di risolvere il problema facendo ricorso al pattern builder, mettendo a disposizione alcuni builder per la costruzione di oggetti di comune utilizzo e dando la possibilità di creare builder personalizzati. Esempi dei builder presenti nella libreria standard di Groovy sono MarkupBuilder per la realizzazione di documenti XML e HTML, JsonBuilder per i file Json, SwingBuilder per la costruzione di interfacce grafiche, AntBuilder per la generazione di script Apache Ant e GraphicsBuilder per la creazione di elementi grafici con Java 2D. I builder sfruttano caratteristiche del linguaggio come le chiusure, la metaprogrammazione e l'omissione delle parentesi per creare oggetti complessi minimizzando la duplicazione del codice e massimizzandone la leggibilità. Inoltre sono particolarmente utili per il *rapid prototyping* e il *rapid application development* in generale, perché permettono di creare e modificare in modo facile ed efficiente la struttura e le proprietà degli oggetti che si vogliono costruire.

5.2 MarkupBuilder

MarkupBuilder permette di creare file in XML e linguaggi derivati, come HTML. Vediamo per prima cosa un esempio di utilizzo del builder:

```
def builder = new groovy.xml.MarkupBuilder()
builder.linguaggi {
  linguaggio (nome: 'C++') {
    autore 'Bjarne Stroustrup'
    anno 1983
  }
  linguaggio (nome: 'Python') {
    autore 'Guido van Rossum'
    anno 1991
  }
  linguaggio (nome: 'Groovy') {
    autore 'James Strachan'
    anno 2003
  }
}
```

L'esempio produce il seguente file XML:

```
<linguaggi>
  <linguaggio nome='C++'>
    <autore>Bjarne Stroustrup</autore>
    <anno>1983</anno>
  </linguaggio>
  <linguaggio nome='Python'>
    <autore>Guido van Rossum</autore>
    <anno>1991</anno>
  </linguaggio>
  <linguaggio nome='Groovy'>
    <autore>James Strachan</autore>
    <anno>2003</anno>
```

```
    </linguaggio>
  </linguaggi>
```

La sintassi utilizzata quando si lavora con i builder, detta GroovyMarkup, è molto compatta ed elegante, e permette di avere subito chiara la struttura dell'oggetto che si sta realizzando. Per fare un confronto, la creazione dello stesso documento in Java avrebbe richiesto l'utilizzo di metodi come `createElement` per creare i nodi, `setAttribute` per impostare l'attributo "nome" dei nodi `linguaggio` e `appendChild` per stabilire le relazioni fra i nodi. Il codice ottenuto sarebbe risultato lineare e non strutturato, e quindi meno leggibile. L'utilizzo dei builder permette invece di definire la struttura dell'oggetto che si vuole costruire separandola dal processo di costruzione vero e proprio, che viene delegato al builder.

Vediamo ora in dettaglio il funzionamento dell'esempio. Prima di tutto viene creato un oggetto `MarkupBuilder`, sul quale viene invocato il metodo `linguaggi`. Quest'ultimo naturalmente non esiste nella classe `MarkupBuilder`, ma invece di segnalare un errore, il builder assume che il nome del metodo rappresenti il nodo radice del documento che si vuole creare. Il metodo `linguaggi` ha come parametro una chiusura che rappresenta il resto del documento XML, all'interno della quale ciascuna chiamata a un metodo inesistente viene interpretata come il nome di un nodo figlio. Se nella chiamata a un metodo vengono specificati dei parametri con i relativi nomi, come nel caso dei metodi `linguaggio`, essi vengono considerati come i nomi e i valori degli attributi del nodo associato al metodo; se invece si indicano solo i valori dei parametri, come per i metodi `autore` e `anno`, essi costituiranno il contenuto del nodo. Infine, se è presente un parametro di tipo chiusura, esso rappresenta l'insieme dei nodi figli del nodo associato al metodo.

Nell'esempio appena visto i dati da scrivere nel file XML vengono introdotti direttamente nel corpo del builder, e il file viene scritto sullo standard output. In un'applicazione reale, invece, i dati vengono ricavati solitamente da una fonte esterna, come un database, e il risultato viene memorizzato in un file o in una stringa. Il seguente esempio mostra quindi come sia possibile realizzare un semplice documento HTML partendo dai dati contenuti in una mappa e scrivendo il contenuto in una stringa, cosa che si ottiene passando un oggetto di tipo `Writer` come `StringWriter` o `FileWriter` al costruttore del builder:

```
def dati = [ 'C++':    ['Bjarne Stroustrup', 1983],
             'Python': ['Guido van Rossum',  1991],
             'Groovy': ['James Strachan',    2003]
           ]
def writer = new StringWriter()
def builder = new groovy.xml.MarkupBuilder(writer)
builder.html {
  head {
    title 'Linguaggi di programmazione'
  }
  body {
    dati.each{ k, v ->
      p "$k è stato creato da ${v[0]} nel ${v[1]}"
```

```
    }
  }
}
println writer
```

Come si può vedere, è possibile utilizzare la sintassi dichiarativa propria dei builder insieme ai costrutti e ai metodi normalmente presenti in un programma Groovy. L'output dell'esempio è il seguente:

```
<html>
  <head>
    <title>Linguaggi di programmazione</title>
  </head>
  <body>
    <p>C++ è stato creato da Bjarne Stroustrup nel 1983</p>
    <p>Python è stato creato da Guido van Rossum nel 1991</p>
    <p>Groovy è stato creato da James Strachan nel 2003</p>
  </body>
</html>
```

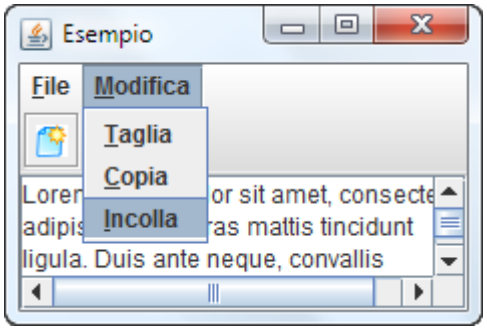
5.3 SwingBuilder

Gli elementi di un'interfaccia grafica sono generalmente organizzati in una struttura gerarchica, con i componenti di base come pulsanti e campi di testo inseriti all'interno di contenitori come pannelli, barre degli strumenti e barre dei menu, a loro volta racchiusi da una finestra. Per questo il processo di creazione di un'interfaccia grafica si presta bene ad essere realizzato in modo semplice e compatto mediante l'uso dei builder. Il punto di riferimento per la creazione di interfacce grafiche in Java e in Groovy è costituito dal framework Swing, che mette a disposizione un'API molto estesa e articolata per la realizzazione di applicazioni grafiche complesse in maniera indipendente dalla piattaforma su cui saranno eseguite. Il relativo builder, SwingBuilder, rende possibile la costruzione di interfacce grafiche in modo simile a quanto visto per i file XML, con la differenza che il nome di ciascun nodo nel GroovyMarkup deve corrispondere a quello di un componente Swing.

Nell'esempio viene realizzato un prototipo di interfaccia per un semplice editor di testo:

```
import javax.swing.*
import java.awt.*
def builder = new groovy.swing.SwingBuilder()
builder.frame (title: 'Esempio', pack: true, show: true,
               defaultCloseOperation: JFrame.EXIT_ON_CLOSE) {
  menuBar {
    menu (text: 'File', mnemonic: 'F') {
      menuItem (text: 'Apri', mnemonic: 'A')
      menuItem (text: 'Salva', mnemonic: 'S')
    }
    menu (text: 'Modifica', mnemonic: 'M') {
      menuItem (text: 'Taglia', mnemonic: 'T')
      menuItem (text: 'Copia', mnemonic: 'C')
```

```
        menuItem (text: 'Incolla', mnemonic: 'I')
      }
    }
    panel (layout: new BorderLayout()) {
      toolBar (constraints: BorderLayout.NORTH) {
        button (icon: new ImageIcon('nuovo_documento.png'))
      }
      scrollPane (constraints: BorderLayout.SOUTH) {
        textArea(rows: 4, columns: 20)
      }
    }
}
```

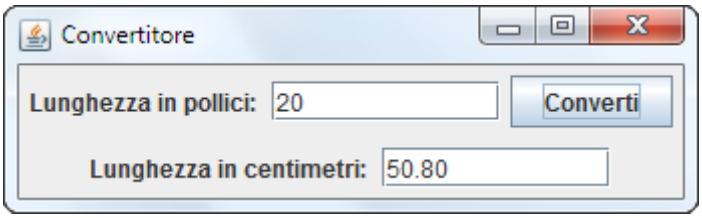


Il codice rispecchia la gerarchia esistente fra i diversi elementi dell'interfaccia: un contenitore di tipo JFrame, che rappresenta l'intera finestra, ha al suo interno una barra con due menu, a loro volta costituiti da vari elementi, e un pannello contenente una barra degli strumenti con un pulsante e un'area di testo con barre di scorrimento che compaiono in caso di necessità. I nomi dei metodi usati per indicare i vari elementi dell'interfaccia sono derivati da quelli dei rispettivi componenti Swing: `button` corrisponde ad un oggetto `JButton`, `panel` ad un `JPanel` e così via. I parametri dei metodi sono utilizzati per impostare gli attributi di ciascun elemento, come il titolo della finestra, i nomi dei menu e le icone dei pulsanti.

La gestione degli eventi è anch'essa molto semplice da implementare: è sufficiente assegnare alla proprietà `actionPerformed` di un elemento una chiusura contenente le istruzioni da eseguire in seguito al verificarsi di un evento, come il clic su un pulsante. Una dimostrazione è data dal seguente esempio, in cui viene creato un convertitore di misure da pollici a centimetri:

```
import javax.swing.*
import java.awt.*
def builder = new groovy.swing.SwingBuilder()
builder.frame (title: 'Convertitore', pack: true, show: true,
               defaultCloseOperation: JFrame.EXIT_ON_CLOSE) {
  panel (layout: new BorderLayout()) {
    panel (constraints: BorderLayout.NORTH) {
      label 'Lunghezza in pollici: '
      textField (id: 'input', columns : 10)
      button (text : 'Converti', actionPerformed : {
                output.text = Integer.parseInt(input.text) * 2.54
              })
    }
    panel (constraints: BorderLayout.SOUTH) {
      label 'Lunghezza in centimetri: '
      textField (id: 'output', columns : 10)
    }
  }
```

```
}
```



Le azioni eseguite in seguito al verificarsi di un evento richiedono spesso di conoscere lo stato di uno o più componenti dell'interfaccia, come in questo caso il contenuto di un campo di testo. È quindi necessario ottenere dei riferimenti agli elementi desiderati. Questo si può fare in due modi: impostando un attributo `id` per gli elementi a cui si vuole fare riferimento, come nell'esempio, oppure assegnando gli elementi a delle variabili: per esempio, la definizione del campo di input poteva essere scritta come:

```
def input = textField (columns: 10)
```

5.4 Creare un builder

Come si è detto in precedenza, Groovy mette a disposizione un buon numero di builder che coprono diversi ambiti di utilizzo. È però anche possibile realizzare un builder personalizzato, definendo la sintassi che deve essere usata per la creazione di un certo tipo di oggetto e il relativo processo di costruzione. Questo può essere fatto in due modi: facendo ricorso alla metaprogrammazione oppure estendendo la classe `BuilderSupport` e implementandone i metodi. La prima tecnica è adatta alla costruzione di builder non troppo complessi e permette di capire meglio il meccanismo di funzionamento che sta alla loro base, mentre la seconda consente una migliore organizzazione del codice per l'implementazione di builder più articolati.

5.4.1 Utilizzando la metaprogrammazione

Il principio alla base del funzionamento di ogni builder è l'intercettazione delle invocazioni di metodi inesistenti sull'oggetto builder; le informazioni ricavate dal nome dei metodi e dal numero, tipo e valore dei parametri vengono poi utilizzate per costruire il prodotto. Per l'intercettazione dei metodi si usa `methodMissing`, introdotto nel paragrafo 4.4, che viene invocato ogni volta che avviene una chiamata ad un metodo non presente nella classe del builder.

Si supponga ad esempio di voler realizzare un builder per la creazione di liste di attività, dove ogni attività può essere costituita a sua volta da più fasi ed è caratterizzata da una data d'inizio e (opzionalmente) una di fine; si vogliono poi distinguere le attività già completate con un segno di spunta. La sintassi del builder da realizzare è del tipo:

```
def builder = new BuilderPerListaAttività()
builder.build {
  Preparazione (inizio: '01/10', fine: '03/10', completata: true)
  Pianificazione_del_progetto (inizio: '04/10') {
    Stima_dei_costi (inizio: '05/10', fine: '06/10', completata: true)
    Stima_delle_risorse (inizio: '06/10', fine: '07/10')
    Stima_del_rischio (inizio: '08/10')
```

```
    }
    Monitoraggio_del_progetto (inizio: '10/10')
    Chiusura_del_progetto (inizio: '20/12')
  }
```

e la lista risultante deve essere:

```
Lista delle attività:
 ✔ Preparazione [ inizio: 01/10 fine: 03/10 ]
 - Pianificazione del progetto [ inizio: 04/10 ]
  ✔ Stima dei costi [ inizio: 05/10 fine: 06/10 ]
  - Stima delle risorse [ inizio: 06/10 fine: 07/10 ]
  - Stima del rischio [ inizio: 08/10 ]
 - Monitoraggio del progetto [ inizio: 10/10 ]
 - Chiusura del progetto [ inizio: 20/12 ]
```

Ad ogni invocazione di metodo viene controllata per prima cosa la presenza del parametro `completata`, e se esso vale `true` si stampa un segno di spunta; viene poi scritto il nome dell'attività, costituito dal nome del metodo privato dei caratteri *underscore*, e gli eventuali parametri rimanenti, racchiusi fra parentesi quadre, che indicano le date di inizio e fine. Bisogna inoltre tenere conto dell'indentazione per distinguere le attività principali dalle sottoattività. La classe del builder è riportata di seguito:

```
class BuilderPerListaAttività {
  def indentazione = 0
  def lista = new StringWriter()
  def build(chiusura) {
    lista << 'Lista delle attività:\n'
    chiusura.delegate = this
    chiusura()
    println lista
  }
  def methodMissing(String nome, param) {
    indentazione++
    indentazione.times{ lista << ' ' }
    if (param.length > 0 && param[0]['completata'])
      lista << '✔ '
    else
      lista << '- '
    lista << nome.replaceAll('_' , ' ')
    lista << stampaParametri(param)
    lista << '\n'
    if (param.length > 0 && param[-1] instanceof Closure) {
      def chiusura = param[-1]
      chiusura.delegate = this
      chiusura()
    }
    indentazione--
  }
  def stampaParametri(param) {
    def risultato = ''
    if (param.length > 0) {
      risultato += ' [ '
      param[0].each{ parametro, valore ->
```

```
            if (parametro == 'completata') return
            risultato += "$parametro: $valore "
          }
          risultato += ']'
        }
        return risultato
      }
    }
```

Il metodo `build` è quello che viene chiamato inizialmente, ed ha come parametro una chiusura contenente i metodi che rappresentano l'insieme delle attività. Per fare in modo che i metodi invocati all'interno della chiusura siano intercettati dal builder, viene memorizzato un riferimento all'istanza del builder nella proprietà `delegate` della chiusura: così facendo, nel momento in cui viene chiamato, ad esempio, il metodo `Preparazione`, l'ambiente di esecuzione cercherà per prima cosa un metodo con quel nome nella classe che contiene la chiusura (cioè quella indicata dal suo riferimento `this`), e, non trovandolo, andrà ad invocarlo sull'istanza del builder, dalla quale verrà intercettato.

Il metodo `methodMissing` si occupa di trasformare ogni invocazione di metodo nella rappresentazione di un'attività con il corretto formato, e al termine controlla se l'ultimo parametro del metodo è costituito da una chiusura (contenente i metodi che costituiscono le sottoattività), nel qual caso, dopo aver impostato la proprietà `delegate`, esegue la chiusura per poterne intercettare i metodi.

5.4.2 Utilizzando BuilderSupport

La creazione di builder complessi è facilitata dalla classe `BuilderSupport` del pacchetto `groovy.util`, che costituisce la superclasse di tutti i builder presenti nella libreria standard di Groovy. `BuilderSupport` è una classe astratta e contiene metodi astratti, cioè dichiarati nella classe ma implementati dalle sottoclassi; per creare un builder è necessario estendere `BuilderSupport` e implementarne i metodi astratti, che sono:

- `createNode(Object nome)`
- `createNode(Object nome, Object chiusura)`
- `createNode(Object nome, Map attributi)`
- `createNode(Object nome, Map attributi, Object chiusura)`
- `setParent(Object genitore, Object figlio)`

Durante il processo di costruzione, quando si verifica l'invocazione di un metodo che rappresenta un nodo nel GroovyMarkup, i metodi ausiliari della classe `BuilderSupport` si occupano di chiamare uno dei quattro metodi `createNode`, a seconda che il metodo nel GroovyMarkup abbia o meno dei parametri e fra questi ci sia una chiusura; il codice necessario alla creazione di un nodo (nell'esempio del paragrafo precedente, il corpo del metodo `methodMissing` e del metodo ausiliario `stampaParametri`) deve quindi essere inserito all'interno dei metodi `createNode`. Viene poi invocato `setParent`, al fine di

mettere a disposizione dell'implementatore del builder un riferimento al nodo genitore e a quello appena creato, per eventuali operazioni da effettuare al termine della creazione di ciascun nodo.

L'uso di `BuilderSupport` elimina la necessità di intercettare i metodi, analizzare il tipo dei parametri, impostare le proprietà `delegate` delle chiusure e occuparsi delle chiamate ricorsive per creare i nodi figli; permette inoltre un'organizzazione migliore e più leggibile della classe del builder, con un metodo `createNode` associato a ciascun possibile tipo di nodo.

6 Multithreading

Il multithreading o multiprogrammazione o programmazione concorrente indica la capacità di un programma di eseguire più flussi di operazioni in parallelo, che vengono detti *thread*. Il vantaggio principale del multithreading consiste nella possibilità di sfruttare più processori, in modo da portare a termine le operazioni richieste in meno tempo, ma è utile anche perché permette al programma di continuare a svolgere altri compiti mentre attende il completamento di un'operazione, come un accesso al disco o ad una risorsa di rete.

6.1 Creazione di thread

Ogni thread è rappresentato da un'istanza della classe `java.lang.Thread`, che fornisce anche una serie di metodi per la gestione del thread stesso. Per creare un thread in Java è necessario creare una classe che estende `Thread` oppure che implementa l'interfaccia `Runnable` e definire, all'interno della classe, un metodo `run` che contiene le istruzioni da eseguire quando il thread viene avviato. In Groovy esiste un'alternativa più compatta: dato che la classe `Closure` implementa `Runnable`, si può creare un thread passando una chiusura come parametro al costruttore della classe `Thread`:

```
def thread = new Thread({ /* istruzioni */ })
thread.start()
```

oppure usando la versione statica del metodo `start` introdotta da Groovy:

```
Thread.start{ /* istruzioni */ }
```

Un thread così creato è detto *user thread*, ed è il tipo di thread più comunemente usato. Esistono anche i *daemon thread*, solitamente impiegati per operazioni da svolgere in background, che vengono creati con il metodo `startDaemon`:

```
Thread.startDaemon{ /* istruzioni */ }
```

La differenza tra i due tipi di thread consiste nel fatto che la presenza di daemon thread attivi non impedisce al programma di terminare. Un programma inizia sempre con l'esecuzione di un thread utente (il thread principale) e non termina finché esiste almeno un thread utente attivo.

Dato che durante l'esecuzione di un programma possono essere presenti più thread di quanti sono i processori disponibili, ogni thread passa periodicamente da uno stato detto *runnable*, cioè pronto per essere eseguito, ad uno stato *running*, ossia in esecuzione. La gestione dei thread è affidata ad un componente della Java Virtual Machine detto *scheduler*, che si occupa di mettere in esecuzione ogni thread per un certo periodo di tempo e di portarlo nello stato runnable quando il periodo è terminato. Si può conoscere lo stato di un thread attraverso la proprietà `state`, come dimostra il seguente esempio:

```
def t1, t2
t1 = new Thread({
  3.times{ println "Questo è il thread 1. Thread 2 è $t2.state" }
})
t2 = new Thread({
```

```
  3.times{ println "Questo è il thread 2. Thread 1 è $t1.state" }
})
t1.start()
t2.start()
```

che dà origine ad un output del tipo:

```
Questo è il thread 1. Thread 2 è RUNNABLE
Questo è il thread 2. Thread 1 è RUNNABLE
Questo è il thread 1. Thread 2 è RUNNABLE
Questo è il thread 2. Thread 1 è RUNNABLE
Questo è il thread 1. Thread 2 è RUNNABLE
Questo è il thread 2. Thread 1 è RUNNABLE
```

La sequenza di output non è prevedibile ed eseguendo più volte il programma si ottengono in generale sequenze diverse.

6.2 Gestione di thread

6.2.1 Sospensione dell'esecuzione

L'esecuzione di un thread può essere controllata attraverso i metodi della classe `Thread`. Un thread può passare nello stato runnable mentre si trova in esecuzione con il metodo `yield`, oppure può sospendere la sua esecuzione per un certo periodo di tempo (espresso in millisecondi) con il metodo `sleep`. In quest'ultimo caso il thread rimarrà nello stato di sospensione per il periodo di tempo specificato, a meno che non venga invocato su di esso il metodo `interrupt` da parte di un altro thread, cosa che provoca il lancio dell'eccezione `InterruptedException`:

```
def t = Thread.start{
  try {
      Thread.sleep(2000)
  } catch (InterruptedException e) {
      println 'Thread interrotto'
  }
}
t.interrupt()
```

Nell'esempio il thread `t` sarebbe dovuto rimanere sospeso per due secondi, ma viene subito interrotto dal thread principale. Se invece il metodo `interrupt` viene invocato su un thread runnable, l'unica conseguenza è l'impostazione di un'indicazione interna di interruzione, che può essere letta dal thread che viene interrotto mediante il metodo `interrupted`:

```
def t = Thread.start{
  def n = 1
  while (!Thread.interrupted())
    println n++
}
Thread.sleep(100)
t.interrupt()
```

Un thread può sospendere la propria esecuzione ed attendere il completamento di un altro invocando il metodo `join` su quest'ultimo. Come `sleep`, `join` risponde ad un'interruzione con l'eccezione `InterruptedException`.

6.2.2 Accesso a risorse condivise

Durante l'esecuzione di un programma *multithreaded*, accade frequentemente che due o più thread abbiano la necessità di accedere ad una stessa risorsa, rappresentata dall'istanza di una classe. In questa situazione possono sorgere dei problemi dovuti al fatto che l'ordine in cui vengono effettuate le operazioni non è definito: ad esempio, supponiamo di avere una variabile `n`, che contiene inizialmente il valore 0, e due thread così definiti:

```
Thread.start{ n = n + 1 }
Thread.start{ n = n - 1 }
```

Una possibile sequenza di esecuzione è la seguente: il primo thread legge il valore di `n` ed esegue l'addizione, ma prima di poter scrivere il risultato nella variabile, il periodo di tempo assegnatogli dallo scheduler scade, il thread viene portato nello stato runnable e il secondo thread entra in esecuzione. Quest'ultimo legge il valore di `n`, che è ancora 0, effettua la sottrazione, imposta il valore di `n` a -1 e termina. Il primo thread può quindi riprendere la sua esecuzione e scrivere nella variabile il valore 1 precedentemente calcolato, per cui, al termine dell'elaborazione, `n` non assume il valore 0 come dovrebbe essere.

Questo tipo di problemi può essere risolto facendo ricorso ad un meccanismo di sincronizzazione, attraverso l'uso della parola chiave `synchronized`. L'espressione

```
synchronized (risorsa) { /* istruzioni */ }
```

fa sì che le operazioni di accesso ad un oggetto `risorsa` possano essere effettuate solo da un thread alla volta, all'interno del blocco di codice che segue. Riscrivendo quindi l'esempio precedente come:

```
def n = 0
Thread.start{
  synchronized (n) {
    n = n + 1
  }
}
Thread.start{
  synchronized (n) {
    n = n - 1
  }
}
```

il valore risultante di `n` sarà sempre 0: prima di poter eseguire delle operazioni sulla variabile, il primo thread deve ottenere il *monitor* associato ad essa, cioè la possibilità di leggerne le proprietà ed invocare su di essa dei metodi. In ogni momento soltanto un thread può essere in possesso del monitor, perciò nel caso in cui il primo thread venisse posto nello stato runnable prima di poter completare la sua elaborazione, il

secondo verrebbe messo in esecuzione e cercherebbe di accedere ad `n`, ma, a questo punto, sarebbe costretto a rimanere in uno stato di sospensione, in attesa che il monitor si liberi. Il primo potrebbe quindi riprendere l'esecuzione, portarla a termine e rilasciare il monitor.

In alternativa è possibile definire come `synchronized` i metodi di una classe. Se un thread sta eseguendo un metodo d'istanza dichiarato `synchronized`, nessun altro thread può eseguire metodi `synchronized` di quell'istanza (ma rimangono eseguibili i metodi non `synchronized` e quelli statici); analogamente, l'accesso di un thread a un metodo statico `synchronized` impedisce ad altri thread di eseguire metodi statici `synchronized` di quella classe. Può succedere che il thread che in un certo momento possiede il monitor associato ad un oggetto abbia la necessità di rilasciarlo, in modo da permettere ad altri thread che competono per l'accesso allo stesso oggetto di proseguire la loro esecuzione: questo può essere fatto usando il metodo `wait` della classe `Object`, che pone il thread in attesa fino a quando un altro thread non invoca il metodo `notify` o `notifyAll` sull'oggetto condiviso. I metodi `notify` e `notifyAll`, anch'essi appartenenti alla classe `Object`, permettono ad un thread di "risvegliare", rispettivamente, uno dei thread e tutti i thread che erano stati precedentemente messi in attesa con `wait`; nessun thread potrà però riprendere la propria esecuzione finché quello che ha invocato `notify` o `notifyAll` non rilascia il monitor.

L'uso dei monitor e la comunicazione fra thread con `wait` e `notify` permettono la risoluzione del problema del produttore-consumatore, un problema classico della programmazione concorrente in cui due thread condividono uno stesso spazio di memoria, che viene continuamente riempito con nuovi dati da uno dei thread, il produttore, e svuotato dall'altro, il consumatore. Il problema consiste nel garantire che il produttore non cerchi di introdurre nuovi dati se lo spazio di memoria è pieno e il consumatore non cerchi di prelevarne se è vuoto. Un esempio di soluzione è il seguente:

```
class Buffer {
  def dimensioneMassima = 5
  def stack = []
  synchronized void inserisci(valore) {
    if (stack.size() == dimensioneMassima)
      wait()
    stack << valore
    println "Inserito: $valore"
    notify()
  }
  synchronized Object preleva() {
    if (stack.isEmpty())
      wait()
    def valore = stack.pop()
    println "Prelevato: $valore"
    notify()
    return valore
  }
}
def buffer = new Buffer()
Thread.start{
```

```
    1.upto(10){
      buffer.inserisci(it)
      Thread.sleep(100)
    }
  }
  Thread.start{
    10.times{
      buffer.preleva()
      Thread.sleep(200)
    }
  }
```

Il fatto che i metodi `inserisci` e `preleva` siano dichiarati `synchronized` assicura che essi non possano trovarsi contemporaneamente in esecuzione. Quando il produttore trova il buffer pieno è necessario metterlo in attesa e passare il monitor al consumatore, e viceversa quando il buffer è vuoto. Usando diversi valori del periodo di `sleep` per i due thread si possono osservare diverse sequenze di esecuzione.

7 Conclusioni

A partire dalle prime versioni, rilasciate quasi dieci anni fa, il linguaggio Groovy ha ricevuto una particolare attenzione da parte della comunità di sviluppatori per la piattaforma Java. Groovy permette di fare uso di caratteristiche, come la sintassi nativa per le collezioni e la programmazione funzionale, a lungo richieste ma non ancora introdotte in Java, un linguaggio per certi aspetti molto conservativo nell'aggiungere nuove funzionalità, spesso per motivi di stabilità e retrocompatibilità. Come si può constatare dagli esempi visti, l'obiettivo di creare un linguaggio espressivo e dalla sintassi compatta è stato raggiunto, così come quello di rendere più semplici numerosi tipi di operazioni frequenti.

Le caratteristiche di Groovy lo rendono particolarmente adatto per il *rapid prototyping*, la creazione di test di unità e come sostituto di Java per progetti medi o piccoli. Una dimostrazione di ciò che è possibile realizzare con Groovy è data da due progetti che in pochi anni hanno raggiunto una discreta diffusione: Grails, un framework per applicazioni web ispirato a Ruby on Rails (l'analogo framework per il linguaggio Ruby), e Griffon, una piattaforma per la realizzazione di applicazioni desktop.

Lo sviluppo del linguaggio è molto attivo, con nuove versioni rilasciate frequentemente e una terza *major release* prevista per il 2013, che, tra le altre cose, porterà miglioramenti sostanziali alla gestione della metaprogrammazione e il supporto a Java 8. Molti utenti del linguaggio partecipano al suo sviluppo attraverso le mailing list ufficiali, segnalando problemi e proponendo modifiche e miglioramenti. Per il futuro si prevede l'introduzione di caratteristiche oggi presenti in altri linguaggi per la piattaforma Java, come Scala e Kotlin, tra cui i tratti, e vari miglioramenti prestazionali.

Bibliografia

[1] D. König, A. Glover, P. King, G. Laforge, J. Skeet, *Groovy in action*. Manning, 2007.

[2] Documentazione ufficiale del linguaggio Groovy
`http://groovy.codehaus.org/`

[3] Mohamed Seifeddine, *Introduction to Groovy and Grails*. 2009.

[4] Groovy - the birth of a new dynamic language for the Java platform
`http://radio-weblogs.com/0112098/2003/08/29.html`

[5] JavaBeans
`http://en.wikipedia.org/wiki/JavaBeans`

[6] Operator Overloading
`http://groovy.codehaus.org/Operator+Overloading`

[7] Functional programming
`http://en.wikipedia.org/wiki/Functional_programming`

[8] Practically Groovy: Functional programming with curried closures
`http://www.ibm.com/developerworks/java/library/j-pg08235/index.html`

[9] The Reflection API
`http://docs.oracle.com/javase/tutorial/reflect/index.html`

[10] Software design pattern
`http://en.wikipedia.org/wiki/Software_design_pattern`

[11] Concurrency
`http://docs.oracle.com/javase/tutorial/essential/concurrency/`