Groovy (autore: Vittorio Albertoni)

Premessa

Il linguaggio di programmazione Java, insieme a C e C++ cardine della programmazione per computer con la P maiuscola, non è un linguaggio con cui possa familiarizzare un dilettante.

Esso, disponibile dal 1995, ha una impostazione alquanto difficile, una sintassi parecchio complicata ed è anche molto verboso [1] .

Per fare meglio e più facilmente tutto ciò che si può fare con Java possiamo ricorrere al linguaggio Kotlin, disponibile dal 2011 [2] .

Fin dal 2007 è tuttavia disponibile Groovy, un linguaggio che ha tutte le caratteristiche e la semplicità dei linguaggi di scripting (come Python, Perl, ecc.) e utilizza il grande patrimonio di librerie di Java in maniera che più semplice non si può.
Non possiamo affrontare tranquillamente progetti impegnativi con le semplificazioni del linguaggio Groovy, utilizzando il quale diventa comunque impegnativo fare cose non semplici, ma ci possiamo divertire a produrre script anche non banali.

Per avere un'idea di che cosa parliamo propongo questi tre modi di scrivere il classico programmino Ciao mondo.

Se utilizziamo il linguaggio Java il codice è il seguente

```
class Ciaomondo
{
    public static void main(String[] args)
    {
        System.out.println("Ciao mondo");
        System.exit(0);
    }
}
```

Se utilizziamo il linguaggio Kotlin il codice è il seguente

```
fun main()
{
   println("Ciao mondo")
}
```

Con il linguaggio Groovy il codice si può ridurre a

```
println("Ciao mondo")
```

In questo manualetto mi propongo di descrivere come Groovy possa essere utilizzato da un programmatore alle prime armi per fare cose non eccessivamente impegnative.

A chi si appassioni e voglia fare di più suggerisco il ricorso alla documentazione ufficiale in lingua inglese.

[1] Di Java ho parlato nell'articolo «Importante riconoscimento per OpenJDK» pubblicato nell'aprile 2016 sul mio blog all'indirizzo *www.vittal.it*.

[2] Sul mio blog all'indirizzo *www.vittal.it* ho dedicato a Kotlin i seguenti manualetti in formato PDF:
kotlin, allegato all'articolo «Kotlin: Java facilitato» del luglio 2019,
gui_kotlin, allegato all'articolo «Grafica con Kotlin» dell'aprile 2023.

Indice

1 Installazione 3

2 Come funziona 3

3 Tipi 5
- 3.1 Tipi fondamentali 5
- 3.2 Tipi contenitore 6

4 Variabili e costanti 7

5 Operatori 8

6 Metodi 9

7 Classi 10

8 Packages 11

9 Interattività con l'utente 12

10 Primi esempi di script 13

11 Strutture di controllo 15

12 Altri esempi di script 18

13 Utilità di un IDE 22
- 13.1 Intellji Idea 22
- 13.2 Eclipse 22

1 Installazione

Per lavorare con Groovy è necessario che sul computer sia installata la macchina virtuale Java.

Per versioni precedenti la 5.0 non è strettamente necessario che sia installato l'intero Java Development Kit (JDK) e per fare ciò che vedremo in questo manualetto basta il Java Runtime Environment (JRE), quello che dovremmo trovare installato per default in qualsiasi sistema operativo.
Anzi, il compilatore di Groovy può compilare anche codice Java e sostituire il comando `javac` del JDK completo.
Quest'ultimo è comunque d'obbligo se dobbiamo fare building complessi, come creare file .jar con il comando `jar` che si trova solo nel JDK completo.

Troviamo Groovy all'indirizzo *https://groovy-lang.org*.

Nel momento in cui scrivo (novembre 2025) è disponibile la versione 5.0.2 che scarichiamo dalla pagina DOWNLOAD: si tratta di un file compresso in formato `.zip` che, una volta decompresso, ci mette a disposizione, nella directory `bin`, i comandi che servono, sia per sistemi unix like (Linux e Mac) sia per il sistema Windows (comandi con estensione .bat).
Per disporre dei comandi stessi in qualsiasi directory siamo collocati dobbiamo sistemare le variabili d'ambiente in modo che la directory dove si trovano i comandi sia nel PATH.
Se lavoriamo su Linux è anche opportuno inserire in `/usr/bin` un collegamento al comando `groovy`, in modo che possa funzionare una shebang standard. Possiamo farlo con il comando
`ln -s` /<percorso al comando `groovy` da collegare> `/usr/bin/groovy`
Se ci teniamo a disporre della versione più aggiornata, teniamo presente che deve esserci una compatibilità tra le versioni di Groovy e di Java: la versione 5.0.2 di Groovy richiede almeno Java 16 e funziona solo se è installato l'intero JDK.

Chi lavora su una distro Linux legata a Debian (Ubuntu e derivate, MX Linux, Mauna, Mint, ecc.) può trovare una versione non aggiornatissima di Groovy nel repository.

Anche su snap troviamo una versione non aggiornatissima di Groovy.

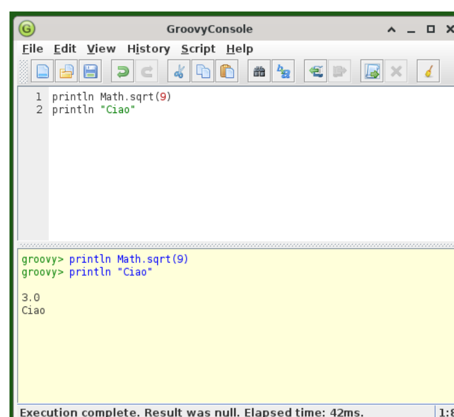
Installando con queste ultime modalità non abbiamo la versione più recente ma non abbiamo problemi di PATH e collegamenti e dovremmo automaticamente avere compatibilità tra versione di Groovy e versione di Java.

Nella pagina DOCUMENTATION del sito troviamo il manuale ufficiale di Groovy, anche in formato PDF scaricabile.

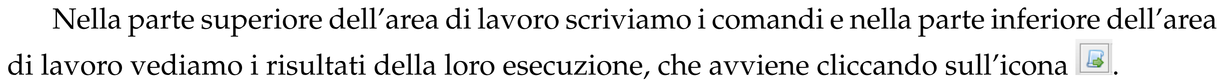
2 Come funziona

Per familiarizzare con Groovy abbiamo a disposizione la console, che si lancia con il comando a terminale
`groovyConsole`
o, a seconda di come abbiamo installato Groovy, dal menu delle applicazioni.

Essa si presenta così



Il menu è autoesplicativo e scorrendo il mouse sulle icone della barra degli strumenti otteniamo la descrizione della funzione sottesa a ciascuna icona.

Nella parte superiore dell'area di lavoro scriviamo i comandi e nella parte inferiore dell'area di lavoro vediamo i risultati della loro esecuzione, che avviene cliccando sull'icona .

La parte inferiore può essere ripulita cliccando sull'icona .

Ciò che abbiamo scritto nella parte superiore può essere salvato in un file di script da menu FILE ▷ SAVE AS..., per il quale viene proposta l'estensione facoltativa `.groovy`.

Possiamo poi eseguire lo script con il comando
`groovy` <file_script>

Altro modo di comporre lo script è di scriverlo con un qualsiasi editor di testo.

Se lavoriamo su Linux possiamo utilizzare la prima riga dello script per la shebang
`#! /usr/bin/groovy`
in modo che, reso eseguibile il file, possiamo lanciarlo scrivendo semplicemente il suo nome.

Fin qui Groovy come linguaggio di scripting, che è materia di questo manualetto.

* * *

L'esecuzione dello script nel modo appena visto comporta una breve attesa, in quanto l'interprete deve compilare il bytecode per poi eseguirlo.

Groovy ci offre però un compilatore che produce un file compilato in bytecode, con estensione `.class`, che con il comando `groovy` seguito dal nome del file senza l'estensione, viene eseguito più rapidamente.

La compilazione avviene con il comando
`groovyc` <file_script>
e genera un file compilato con lo stesso nome del file di script e con estensione `.class`, come fosse un file compilato java.
Ma non è un file java. E' semplicemente un file che viene eseguito sulla macchina virtuale Java e che, essendo scritto non in codice Java ma in modo semplificato rispetto ad esso, tanto più elevata è questa semplificazione tanto più, per la sua esecuzione, è necessario che nella stessa directory di esecuzione si trovino sia il file script originario sia il file compilato.

Se il file script originario fosse interamente scritto secondo le regole sintattiche del linguaggio Java, potremmo salvarlo con estensione `.java` e compilarlo con il comando
`groovyc -j` <file_java>
e il file `.class` generato sarebbe un file Java a tutti gli effetti, pienamente portabile ed eseguibile con il comando `java`, come fosse stato compilato con il comando `javac` del JDK.

Le situazioni intermedie comportano complicazioni che richiedono la profonda conoscenza dei segreti della macchina virtuale Java che non possono essere oggetto di questo manualetto, dedicato a Groovy come linguaggio di scripting puro e semplice.

* * *

I comandi che compongono uno script Groovy, oltre che con le librerie proprie `groovy.lang` e `groovy.util`, interagiscono con librerie Java.

Per default sono accessibili senza che sia necessario importarle le librerie

- `java.io.*`
- `java.lang.*`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `java.net.*`
- `java.util.*`

E' possibile utilizzare qualsiasi altra libreria Java importandola con il comando `import` all'inizio dello script.

3 Tipi

Come tutti i linguaggi di scripting, Groovy si distingue dai linguaggi di programmazione strutturata, tutti a tipizzazione statica, in quanto è un linguaggio a tipizzazione dinamica e il programmatore può decidere se e quando renderlo a tipizzazione statica. Vedremo meglio tutto questo nel prossimo Capitolo.

Per intanto vediamo quali sono i tipi dei dati elaborabili in Groovy.

3.1 Tipi fondamentali

Tipi numerici

Le parole chiave per definire i tipi numerici sono
byte per interi con una dimensione di 8 bit
short per interi con una dimensione di 16 bit
int per interi con una dimensione di 32 bit
long per interi con una dimensione di 64 bit
float per decimali con una dimensione di 32 bit
double per decimali con una dimensione di 64 bit

Per avere un'idea delle grandezze numeriche compatibili con i vari tipi di interi occorre elevare 2 al numero dei bit.

Per i decimali ricordare che il tipo float (precisione singola) gestisce fino a 7 cifre decimali significative e il double (doppia precisione) gestisce fino a 15 cifre decimali significative.

Queste parole chiave, che con la loro iniziale minuscola sembrano indicare tipi primitivi, come avviene in quasi tutti i linguaggi, in realtà, come avviene in Kotlin, richiamano classi Java, nel caso specifico le classi di `java.lang` con nomi simili con la iniziale maiuscola.

Possiamo peraltro definire i tipi numerici di cui sopra anche usando i nomi delle classi java con la inziale maiuscola (rammentando che alla parola chiave `int` corrisponde la classe `Integer)`.

Per trattare interi e decimali a precisione arbitraria usiamo
BigInteger
BigDecimal
che richiamano le classi omonime del package `java.math.`

Tipi carattere

Per definire i tipi carattere abbiamo
char per un singolo carattere Unicode a 16 bit scritto tra apici semplici o doppi.
String per una stringa di caratteri.

La stringa è una successione di caratteri usata per rappresentare testo.
Il contenuto della stringa normale, immutabile, si scrive tra apici semplici (' e ').
Il contenuto della stringa interpolabile si scrive tra doppi apici (" e ").
In entrambi i casi possiamo utilizzare al loro interno i caratteri di escape, come il new line `\n.`
Introducendo e chiudendo la stringa con tre apici semplici o doppi possiamo scrivere la stringa su più righe e in questo caso non vengono accettati i caratteri di escape.

In entrambi i casi la classe utilizzata è la classe `java.lang.String.`

Tipo booleano

Con
Boolean o **boolean** rappresentiamo i valori logici `true` e `false`.

La classe di riferimento è `java.lang.Boolean`.

3.2 Tipi contenitore

I tipi contenitore sono strutture di dati.

List

La lista è una successione di elementi eterogenei separati da virgola e racchiusi tra parentesi quadre.

Si identifica con la parola chiave
List

Array

L'array è una lista di elementi omogenei del tipo indicato e gli elementi si indicano separati da virgola e racchiusi tra parentesi quadre, come le liste.
Il tipo array si identifica con la sintassi
<tipo_base>**[]**

Con la sintassi
<tipo_base>**[] []**
si identifica una matrice i cui elementi si indicano in una lista contenente le liste che rappresentano le righe della matrice.

In entrambi i casi la classe utilizzata è `java.util.ArrayList`.

Map

E' quello che in altri linguaggi si chiama dizionario o array associativo.

Praticamente è una lista nella quale ogni elemento è costituito dall'associazione tra una chiave e un dato separati dal simbolo :

In questo caso la classe utilizzata è `java.util.LinkedHashMap`.

* * *

Il fatto che Groovy sia un linguaggio a tipizzazione dinamica non vuol dire che esso ignori i tipi, vuole semplicemente dire che le variabili non hanno un tipo fisso. Ma ai valori in quanto tali il linguaggio assegna sempre un tipo per default.

Per verificare il tipo assegnato abbiamo a disposizione la funzione `.getClass()` che è una funzione membro di qualsiasi oggetto Groovy, dove tutto è un oggetto, anche un semplice numero digitato sulla tastiera.

Se nella console di Groovy digitiamo
`4.getClass()`
eseguendo otteniamo in risposta
`Result: class java.lang.Integer`

Se digitiamo
`[1,2,3,4].getClass()`
eseguendo otteniamo in risposta
`Result: class java.util.ArrayList`

Come si vede Groovy riconosce la sintassi con cui abbiamo digitato il dato e assegna al dato stesso il tipo previsto per quella sintassi.

Interessante notare come a un numero decimale Groovy assegni per default il tipo a precisione arbitraria BigDecimal. Infatti, se digitiamo
`5.67.getClass()`
eseguendo otteniamo in risposta
`Result: class java.math.BigDecimal`

4 Variabili e costanti

Variabili e costanti, in Groovy, sono oggetti che corrispondono a posizioni di memoria in cui inseriamo un valore per averlo a disposizione per le elaborazioni previste dallo script.

La variabile contiene un valore modificabile, la costante contiene un valore fisso, che non può essere modificato.

Per creare una costante abbiamo a disposizione la parola chiave `final` con questa sintassi

final <nome> = <valore>

dove <nome> è l'identificativo della costante per poterne richiamare il contenuto quando serve e <valore> è il valore assegnato, che può essere di uno qualsiasi dei tipi visti nel precedente Capitolo.

Con

`final pi = Math.PI`

creiamo la costante `pi` contenente il valore di π e ogniqualvolta dobbiamo richiamarlo possiamo farlo scrivendo semplicemente `pi` in luogo di `Math.PI`.

Per creare una variabile possiamo semplicemente usare la sintassi

<nome> = <valore>

dove <nome> è l'identificativo della variabile per poterne richiamare il contenuto quando serve e <valore> è il valore inizialmente assegnato.

In questo modo la variabile viene creata nel momento in cui le viene assegnato il valore e le viene attribuito il tipo del valore inserito. Se esiste già una variabile con lo stesso nome viene utilizzata questa e il suo tipo si adegua a quello del nuovo valore: ciò che si chiama tipizzazione dinamica.

E questo è il funzionamento di base di Groovy, funzionamento che alleggerisce il lavoro di programmazione ma che rischia di produrre programmi poco affidabili.

Un primo accorgimento per attutire questo rischio è quello di dichiarare esplicitamente le variabili utilizzando la parola chiave `def`, con la sintassi

def <nome>

volendo inizializzandola da subito con

def <nome> = <valore>

Per quanto riguarda la tipizzazione tutto rimane come prima ma la variabile così creata rimane utilizzabile soltanto all'interno della funzione che la contiene. Cioè, mentre la variabile creata senza la parola chiave `def` è una variabile globale, quest'altra è una variabile locale.

Altro contributo alla eliminazione dei rischi della grande libertà che ci è consentita da Groovy è quello del passaggio alla tipizzazione statica, inserendo il tipo della variabile nei comandi per la sua creazione o per la sua dichiarazione, che diventano

<tipo> <nome> = <valore>

oppure

def <tipo> <nome>

volendo inizializzandola da subito con

def <tipo> <nome> = <valore>

Così, la variabile creata

`int a = 5`

non potrà che contenere come valore che un numero intero.

Le costanti e le variabili di Groovy sono oggetti e sono dotate di vari metodi. Qui segnalo quelli che ritengo più utili.

metodi delle costanti e delle variabili numeriche .

`<nome>.toString()` estrae in tipo String il contenuto della costante o della variabile nominata

`<nome>.toInteger()` estrae in tipo int il contenuto della costante o della variabile nominata

`<nome>.toFloat()` estrae in tipo float il contenuto della costante o della variabile nominata

`<nome>.toDouble()` estrae in tipo double il contenuto della costante o della variabile nominata

Ricordare che il casting alla rovescia, da Double a Integer, comporta la perdita della parte decimale senza arrotondamento.
Interessante notare che, dal momento che Groovy riconosce il tipo adatto per quanto scriviamo sulla tastiera, i metodi che abbiamo visto si possono applicare direttamente a ciò che scriviamo. Per esempio se scriviamo 10.375.toInteger() otterremo 10. Se scriviamo 10.toDouble() otteniamo 10.0, ecc.

metodi delle costanti e delle variabili stringa .
<nome>.length() ritorna il numero dei caratteri della stringa nominata
<nome>.getAt(<indice>) ritorna il carattere corrispondente all'indice nella stringa nominata

metodi delle costanti e delle variabili lista e array .
<nome>.size() ritorna il numero degli elementi della lista o array nominati
<nome>.getAt(<indice>) ritorna l'elemento corrispondente all'indice (partendo da 0)
<nome>.add(<elemento>) aggiunge un elemento alla lista o all'array nominati
<nome>[<indice>] = <valore> sostituisce con <valore> l'elemento corrispondente all'indice

5 Operatori

Gli operatori collegano tra loro operandi di varia natura in espressioni che forniscono un risultato.

Operatori aritmetici

In ordine di esecuzione sono i seguenti:
** per l'elevamento a potenza
* per la moltiplicazione
/ per la divisione
% per il modulo (resto della divisione intera)
+ per la somma
- per la sottrazione

Non abbiamo un operatore per la divisione intera che possiamo effettuare utilizzando il metodo .intdiv() del dividendo, indicando tra le parentesi il divisore:
5.intdiv(2) restituisce 2

Questi operatori operano tra valori numerici.

L'operatore + può essere utilizzato per concatenare stringhe.

Operatori di confronto

Servono per confrontare due valori e il risultato che restituiscono è un valore booleano.

Sono i seguenti.
== uguale,
!= non uguale,
< minore,
<= minore o uguale,
> maggiore,
>= maggiore o uguale,
=== identico,
!== non identico.

Gli operandi assoggettati al confronto devono avere lo stesso tipo.

Operatori logici

Sono i seguenti.
`&&` che sta per `and`,
`||` che sta per `or`,
`!` che sta per `not`.

6 Metodi

Il metodo, in altri contesti detto anche funzione, procedura o subroutine, è una sezione del programma, identificata con un nome, in cui è racchiusa una serie di istruzioni da eseguire e che vengono eseguite ogni volta che il metodo è chiamato.

Lavorando con Groovy abbiamo a disposizione tutti i metodi contenuti nei packages (librerie) Java.

Nel Capitolo 4 abbiamo visto alcuni metodi ricollegabili agli oggetti costituiti da variabili e costanti. Nella Premessa abbiamo fatto la conoscenza del metodo `println()` per scrivere nello standard output, metodo che si trova nel package java.io, importato per default da Groovy.

Nel seguito faremo la conoscenza di altri metodi preconfezionati che Groovy ci mette a disposizione.

Qui vediamo come noi stessi possiamo costruire un metodo, cosa molto utile, per semplificare la scrittura di certi script, quando potrebbe essere utile raggruppare alcune istruzioni all'interno dello stesso script, creando blocchi di istruzioni richiamabili, in modo da suddividere i compiti e da evitare di scrivere più volte le stesse cose.

La sintassi per dichiarare e scrivere un metodo è

```
def <tipo_risultato> <nome> (<tipo_parametro> <nome_parametro>, ....)
{
   <istruzioni>
}
```

dove
<tipo_risultato> è il tipo del risultato che intendiamo ottenere e va indicato nel caso di metodi che ritornino un valore come risultato
<nome> è il nome che diamo al metodo, ed è il nome che useremo per richiamarlo,
<tipo_parametro> e <nome_parametro> sono tipo e nome del parametro (dato) da inserire quando lo richiamiamo, e ce ne possono essere più di uno, separati da virgola,
<istruzioni> sono quelle relative alle elaborazioni da effettuare sui parametri per ottenere il risultato.

Con la parola chiave `return` possiamo far restituire qualsiasi valore indicato.

La sintassi per chiamare un metodo ed eseguirlo è

```
<nome_metodo> (<parametri>)
```

Esempi:

```
def saluta(String nome)
{
   println "Ciao, $nome!"
}
```

L'indentazione non è necessaria ma la utilizzo per migliore chiarezza.

Il metodo richiede l'indicazione del nome della persona da salutare e il saluto avviene interpolando nella relativa stringa il parametro.

Scrivendo

```
saluta("Vittorio")
```

otteniamo

```
Ciao, Vittorio!
```

```
def double radice(double radicando, int indice)
{
   radicando ** (1/indice)
}
```

Sapendo che $\sqrt[n]{x}$ vale quanto $x^{\frac{1}{n}}$ questo metodo calcola radice ennesima di un numero attraverso i parametri radicando e indice di radice.

Scrivendo `radice(16,4)` otteniamo `2.0`.

Scrivendo `radice(2,2)` otteniamo `1.4142135623730951`.

* * *

Quando le istruzioni da svolgere sono in numero limitato e semplici, come avviene per i due esempi appena visti, possiamo ricorrere alla forma abbreviata denominata Closure.

Un closure in Groovy è un oggetto che rappresenta un blocco di codice eseguibile.

Questo blocco di codice può essere assegnato a una variabile, passato come argomento a un metodo o eseguito in un secondo momento richiamandolo come si fa per i metodi. È simile alle funzioni lambda o funzioni anonime in altri linguaggi.

La sintassi per creare un closure è la seguente

{<parametri> -> <istruzioni>}

Se i parametri sono più di uno vanno indicati uno dopo l'altro separati da virgola.

Utilizzando il closure possiamo creare metodi riutilizzabili assegnando il closure a una variabile.

Per i due esempi precedenti il codice diventa

```
def saluta = {String nome -> println "Ciao, $nome!"}
```

e

```
def radice = {double radicando, int indice -> radicando ** (1/indice)}
```

7 Classi

Da ciò che abbiamo visto fin qui, Groovy si presta ottimamente ad un paradigma di programmazione funzionale ma, essendo un linguaggio per oggetti, supporta in pieno la programmazione a oggetti.

Tanti più esperti di me raggiungibili in rete possono ricordare o spiegare al lettore che cosa si intende per programmazione a oggetti.

Io provo a farlo capire con un esempio.

Supponiamo di avere spesso bisogno di determinare area e circonferenza del cerchio di cui ci è noto il raggio.

Se vogliamo evitare di scrivere tutte le volte le necessarie formule, anche perché non è detto che ce le ricordiamo (a volte si ha a che fare con formule che più difficilmente di quelle per calcolare la circonferenza sono rimandabili a memoria), possiamo creare dei metodi che le contengono oppure possiamo ricorrere alla programmazione a oggetti. Se seguiamo quest'altra strada scriviamo una Classe attraverso la quale creare un oggetto Cerchio, contenente metodi (chiamati anche funzioni membro) attraverso le quali calcolare area e circonferenza.

Come si scrive un metodo lo abbiamo visto. Nel caso del nostro esempio il metodo per calcolare l'area del cerchio sarà scritto così:

```
def double area_cerchio (double raggio)
{
   Math.PI * raggio * raggio
}
```

Con

```
area_cerchio(3)
```

otteniamo

```
28.274333882308138
```

Se vogliamo seguire i dettami della programmazione a oggetti dobbiamo scrivere la classe Cerchio, dotarla delle funzioni membro per calcolare area e circonferenza, utilizzarla per costruire un oggetto cerchio del raggio desiderato e fare i nostri calcoli richiamando le funzioni membro di questo oggetto.

La sintassi per scrivere la classe è la seguente

```
class Cerchio
{
   double raggio

   Cerchio(double raggio)
   {
   this.raggio = raggio
   }

   double area() {Math.PI * raggio * raggio}
   double circonferenza() {Math.PI * 2 * raggio}
}
```

Denominata la classe, con iniziale maiuscola come vuole Java, all'interno del blocco di istruzioni indichiamo innanzi tutto i parametri che caratterizzano l'oggetto da costruire (nel nostro caso uno solo, il raggio, espresso in tipo double) poi scriviamo il metodo costruttore e infine i metodi membro che ci servono.

Questa classe può essere utilizzata per costruire un oggetto cerchio avente un certo raggio e dotato dei metodi per calcolarne area e circonferenza in questo modo:

```
mioCerchio = new Cerchio(3)
println mioCerchio.area()
println mioCerchio.circonferenza()
```

Innanzi tutto abbiamo costruito l'oggetto mioCerchio come istanza della classe Cerchio con il parametro raggio uguale a 3.

Poi abbiamo scritto di stampare area e circonferenza richiamando i metodi dell'oggetto mioCerchio con raggio 3 per calcolare queste grandezze, con il risultato

```
28.274333882308138
18.84955592153876
```

8 Packages

Il package è una raccolta di funzioni e/o di classi, una libreria.

Il linguaggio Groovy ci mette a disposizione tutti i packages del sistema di sviluppo Java, per fare praticamente tutto ciò che si può immaginare di programmare.

Anche le funzioni e le classi che creiamo noi potremmo inserirle in package, ma non è un lavoro da dilettanti, soprattutto quando si tratta di utilizzarli.

Ai principianti che vogliano conservare librerie prodotte da loro consiglio di tenerle memorizzate in un file di testo e copiare la parte che interessa nel file dove si scrive il programma, lavorando come fatto negli esempi dei capitoli precedenti: in questo modo creiamo pseudo-packages artigianali senza complicazioni.

Ho già elencato nel Capitolo 2 quali packages abbiamo a disposizione per default. Se abbiamo bisogno di altri packages dobbiamo importarli.

L'importazione di tutto ciò che contiene il package si fa con la sintassi

```
import <nome_pacchetto>.*
```

Se ci basta importare una sola classe possiamo farlo con la sintassi

```
import <nome_pacchetto>.<nome_classe>
```

Tra ciò che abbiamo a disposizione per default vedremo nel prossimo Capitolo ciò che ci può servire per creare interattività con l'utente.

Qui merita menzione ciò che ci offre il package java. lang. Math.

Package Math

Contiene costanti e funzioni matematiche che ci possono servire per qualsiasi calcolo possiamo immaginare. Esse si richiamano con la sintassi
`Math.<costante_o_funzione>`
Esaminiamole per raggruppamenti.

Costanti

Abbiamo le due più utilizzate costanti matematiche `E` (numero e, base dei logaritmi naturali), `PI` (pi greco).

Radici e potenze

`pow(<base>, <esponente>)` ritorna in double la base elevata all'esponente indicato, espressi in double,
`exp(x)` ritorna in double il numero e elevato a `x`, espresso in double,
`sqrt(x)` ritorna in double la radice quadrata di `x`, espresso in double,
Per radici ennesime dobbiamo ricorrere alla funzione `pow` indicando come esponente il reciproco dell'indice di radice, dato che $\sqrt[n]{x} = x^{\frac{1}{n}}$.

Funzioni trigonometriche

Tutte le funzioni trigonometriche hanno l'argomento espresso in double radianti e ritornano un valore double. Le inverse hanno l'argomento in double e ritornano un valore in double radianti.
Abbiamo le funzioni `sin(x)`, `cos(x)`, `tan(x)` e le funzioni inverse `asin(x)`, `acos(x)`, `atan(x)`.
Abbiamo pure le seguenti funzioni di conversione
`toDegrees(x)` converte radianti in gradi, il tutto espresso in double,
`toRadians(x)` converte gradi in radianti, il tutto espresso in double.

Funzioni iperboliche

Tutte le funzioni iperboliche hanno l'argomento espresso in double e ritornano valori in double.
Abbiamo le funzioni `sinh(x)`, `cosh(x)`, `tanh(x)`.

Funzioni varie

Tutte queste funzioni operano su tipi double.
`abs(x)` ritorna il valore assoluto di `x`,
`ceil(x)` ritorna, in float64, il più piccolo intero maggiore o uguale a `x`,
`floor(x)` ritorna, in float64, il più grande intero minore o uguale a `x`,
`round(x)` ritorna il numero intero più prossimo a `x`,
`log(x)` ritorna il logaritmo naturale di `x`,
`log10(x)` ritorna il logaritmo decimale di `x`
`random()` ritorna un numero casuale di tipo double compreso tra 0 e 1,

9 Interattività con l'utente

Uno script che si rispetti prima o poi deve interagire con l'utente, quanto meno per evidenziare all'utente stesso i risultati delle elaborazioni previste nello script e, spesso, anche per chiedere all'utente di inserire dati o esprimere scelte per proseguire le elaborazioni.
I metodi che abbiamo a disposizione per queste cose sono i seguenti.

println

Questo metodo scrive l'output a terminale usando una formattazione di default.

Gli elementi da scrivere, che passiamo come parametri, possono essere nomi di variabili (e ne verrà scritto il valore), letterali numerici o espressioni numeriche (e ne verrà scritto il risultato), stringhe (parole o frasi racchiuse tra doppi apici).

Più parametri stringa si inseriscono separati dal simbolo `+`.

Per iniettare il valore di una variabile in una stringa si può inserire nella stringa il nome della variabile preceduto dal simbolo `$`, secondo la tecnica dell'interpolazione della stringa, che, in questo caso, deve essere scritta tra doppi apici.

`println` inserisce automaticamente un new line e, dopo avere scritto, va a capo.

Nel caso di numeri con molti decimali è possibile limitare la scrittura dei decimali ricorrendo al metodo `.round()` del numero da stampare, indicando tra le parentesi il numero di decimali voluto.

Data una variabile `x` contenente il numero `456.6655234426` possiamo stamparne il contenuto con sole tre cifre decimali con

```
println x.round(3)
```

ottenendo `456.666`

Esempio:

Supponendo di avere una variabile a contenente il valore `12.5`, l'istruzione

```
println("3 moltiplicato 4 fa " + 3*4 + " e la variabile a vale " + a)
```

scrive a terminale quanto segue

```
3 moltiplicato 4 fa 12 e la variabile a vale 12.5
>>>
```

Lo stesso risultato si sarebbe ottenuto scrivendo

```
println("3 moltiplicato 4 fa " + 3*4 + " e la variabile a vale $a")
```

print

Fa tutto quello che fa `println` ma non inserisce il new line e non va a capo quando ha scritto. Per andare a capo occorre inserire il new line `\n` come stringa.

Non andando automaticamente a capo non è adatto per essere usato nella shell interattiva.

readLine

Questo metodo dobbiamo richiamarlo ricorrendo alla classe System.in del package `java.lang` con la sintassi

```
System.in.newReader().readLine()
```

Legge l'input dalla tastiera come stringa e lo può collocare in una variabile con

```
def <nome_variabile> = System.in.newReader().readLine()
```

Per fare in modo che il dato inserito con la tastiera venga letto con tipo diverso dal tipo stringa dobbiamo aggiungere il tipo desiderato con la parola chiave `as`.

```
System.in.newReader().readLine() as int
System.in.newReader().readLine() as double
System.in.newReader().readLine() as BigInteger
System.in.newReader().readLine() as BigDecimal
```

leggono il dato secondo il tipo indicato.

10 Primi esempi di script

Per stilare uno script serve un editor di testo, meglio se pensato per la programmazione, come Geany, che ci consente dal suo interno di lanciarlo e compilarlo.

Piccoli script che non prevedano interattività con l'utente potrebbero essere scritti e provati anche nella console, che è tuttavia più adatta per provare comandi singoli.

Se vogliamo inserire commenti possiamo farlo su una sola riga, anche la stessa di un comando dopo aver scritto il comando stesso, premettendo il simbolo //.

Commenti su più righe possono essere scritti tra i simboli /* e */.

Se lavoriamo in Linux possiamo utilmente inserire nella prima riga la shebang
#! /usr/bin/groovy
in modo che lo script, reso eseguibile, sia lanciabile senza ricorrere all'interprete.

Cominciamo con un semplice script che chiede all'utente il nome per salutarlo e che scrivo nella versione più stenografica del linguaggio

```
println 'Come ti chiami?'
nome = System.in.newReader().readLine()
println "Ciao, $nome!"
```

Bene sapere che l'interprete Groovy può eseguire anche script contenenti un misto di linguaggio Java e Groovy e addirittura script interamente scritti in linguaggio Java.

Potremmo comporre questo script, avente la stessa finalità del precedente, con un misto di linguaggio Java e Groovy, così

```
class Saluto
{
   static void main(String[] args)
   {
       String nome
       println("Come ti chiami?")
       nome = System.in.newReader().readLine()
       println "Ciao, $nome!"
   }
}
```

e funzionerebbe.

Potremmo anche comporre lo script interamente in linguaggio Java

```
class Saluto
{
   public static void main(String[] args) throws Exception
   {
       BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
       System.out.println("Come ti chiami?");
       String nome = input.readLine();
       System.out.println("Ciao " + nome + "!");
       System.exit(0);
   }
}
```

ed anche così funzionerebbe.

Quest'altro script può servire per calcolare una potenza in precisione arbitraria.

```
print 'Indica la base:  '
b = System.in.newReader().readLine() as BigDecimal
print "Indica l'esponente:  "
e = System.in.newReader().readLine() as int
p = b ** e
println p.round(2)
```

Il risultato viene scritto arrotondando alla seconda cifra decimale e può essere un numero di svariate centinaia di cifre.

11 Strutture di controllo

Come tutti i linguaggi di programmazione, Groovy ci offre il modo di controllare l'esecuzione del programma attraverso istruzioni per condizionare l'esecuzione di un blocco al verificarsi di certe condizioni oppure per la ripetizione di blocchi.

if

L'istruzione if, chiamata istruzione di esecuzione condizionale (in inglese if è il nostro se), ci dà modo di assoggettare l'esecuzione di un blocco di istruzioni al verificarsi di una determinata condizione: se la condizione è vera viene eseguito il blocco di istruzioni, altrimenti si prosegue l'esecuzione del programma saltando il blocco stesso.

La sintassi è la seguente

```
if (<condizione>)
{
   <istruzioni>
}
```

dove <condizione> è una qualsiasi espressione che relaziona due valori attraverso operatori di confronto: se la condizione si verifica vengono eseguite le istruzioni contenute tra le parentesi graffe, altrimenti si passa oltre.
L'uso delle parentesi graffe è d'obbligo se le istruzioni sono più di una. Nel caso di una sola istruzione essa si può semplicemente scrivere senza parentesi, anche di seguito a if (<condizione>), sulla stessa riga.

L'istruzione if si presta anche all'esecuzione condizionale a due rami. Per ottenere questo dobbiamo abbinarla all'istruzione else con questa sintassi

```
if (<condizione>)
{
   <istruzioni>
}
else
{
   <istruzioni>
}
```

In questo caso se la condizione si verifica vengono eseguite le istruzioni contenute nel primo blocco altrimenti vengono eseguite quelle contenute nel secondo blocco dopo else (altrimenti). In ogni caso proseguendo poi nell'esecuzione del resto del programma.

Possiamo infine gestire l'esecuzione condizionale a più rami abbinando a if l'istruzione else if con questa sintassi

```
if (<condizione>)
{
   <istruzioni>
}
else if (<condizione>)
{
   <istruzioni>
else if (<condizione>)
{
   <istruzioni>
}
......
```

e proseguire quanto vogliamo.

Il seguente script

```
println('Inserisci un numero')
x = System.in.newReader().readLine() as int
if (x < 10) println 'Ci sei dentro'
else println('Ciao')
```

se inseriamo un numero minore di 10 stampa la frase «Ci sei dentro»; se inseriamo 10 o un numero superiore stampa il saluto «Ciao».

Quest'altro

```
println('Inserisci un numero')
x = System.in.newReader().readLine() as int
if (x < 10) println('Sei dentro')
else if (x >= 10 && x <= 12) println('Sei appena fuori')
else println("Sei completamente fuori")
println("Ciao")
```

se inseriamo un numero minore di 10 stampa la frase «Ci sei dentro» e poi il saluto «Ciao»; se inseriamo 10 o 11 o 12 stampa la frase «Sei appena fuori» e poi il saluto «Ciao»; se inseriamo un numero superiore a 12 stampa la frase «Sei completamente fuori» e poi il saluto «Ciao».

switch

Il costrutto `switch` è un altro modo di fare ciò che possiamo fare con la serie di else e else if.

Con questo costrutto l'ultimo esempio che ho fatto può essere scritto così

```
println('Inserisci un numero')
n = System.in.newReader().readLine() as int
switch (n)
{
   case {n in 0..9}-> println('Sei dentro')
   case {n in 10..12} -> println('Sei appena fuori')
   case {n > 12} -> println('Sei completamente fuori')
}
println 'Ciao'
```

Come si vede, si abbinano direttamente istruzioni semplici ai valori che può assumere una determinata variabile, che può anche essere una stringa, come vediamo nel seguente esempio.

```
println('Inserisci il colore del semaforo con lettera minuscola')
semaforo = System.in.newReader().readLine()
switch (semaforo)
{
   case "rosso" -> println "Fermati"
   case "verde" -> println "Passa tranquillo"
   case "giallo" -> println "Rallenta perché dovrai fermarti"
   default -> println "Dovevi inserire uno dei colori del semaforo in minuscolo"
}
```

In questo caso, a seconda del colore che scriviamo, abbiamo la risposta che ci indica come comportarci e se inseriamo una parola che non indica un colore del semaforo o se non usiamo l'iniziale minuscola come richiesto abbiamo l'invito a comportarci come si deve.

while

Si usa quando si vuole ripetere l'esecuzione di una istruzione o di un blocco di istruzioni fino a quando rimane vera (`true`) una determinata condizione booleana.

E' la tipica istruzione che consente di ripetere l'esecuzione un numero definito di volte utilizzando un contatore.

La sintassi per l'uso di questa istruzione è

```
contatore = 1
while (contatore <= n)
{
   <istruzioni>
   contatore = contatore + 1
}
```
La variabile che ho battezzato `contatore` per richiamarne la natura, viene inizializzata con il valore 1. Ad ogni esecuzione delle istruzioni viene aumentata di 1 e, una volta raggiunto un valore n voluto, si ferma l'esecuzione. In genere viene nominata semplicemente `i`.

Questo piccolo script
```
i = 1
while (i <= 3)
{
   println("Ciao, Vittorio")
   i = i + 1
}
```
scrive la frase Ciao, Vittorio per tre volte.

Quest'altro
```
i = 1
while (i <= 6)
{
   println(i)
   i = i + 1
}
```
elenca i numeri da 1 a 6.

In entrambi gli esempi il blocco da ripetere contiene una sola istruzione oltre a quella di aumentare di 1 il contatore ma, ovviamente, può contenerne quante vogliamo.

for

Si usa quando si vuole ripetere l'esecuzione di una istruzione o di un blocco di istruzioni ogniqualvolta è vera (true) una determinata condizione booleana, verificata attraversando una qualsiasi struttura (stringa, range di valori, ecc.)

La sintassi per l'uso di questa istruzione è
```
<definizione di una struttura>
for (<variabile> in <struttura>)
{
   <istruzioni>
}
```
Se definiamo la struttura come un range di numeri interi (per esempio da 1 a 5) con l'istruzione for possiamo fare ciò che si fa con while.
Per scrivere tre volte il saluto «Ciao, Vittorio!» possiamo fare così:
```
r = 1..3
for (x in r) println('Ciao, Vittorio')
```
La struttura potrebbe essere una stringa e potremmo collegare l'esecuzione di un'istruzione o di un blocco di istruzioni alla presenza di un certo carattere o tipo di carattere contenuto nella stringa, come nei seguenti esempi.

Il seguente script
```
s = "Vittorio"
for (x in s)
{
   if (x == "t")
   println("trovato t")
}
```
scrive due volte la frase «trovato t» avendo trovato due volte la lettera `t` nella stringa `Vittorio`.

12 Altri esempi di script

Con le strutture di controllo abbiamo esaurito l'esame degli strumenti che ci mette a disposizione il linguaggio Groovy per creare script eseguibili da riga di comando, cioè senza interfaccia grafica e possiamo fare qualche esempio un tantino più complicato di quelli presentati nel Capitolo 10.

Cominciamo da alcuni script per i quali bastano le conoscenze si qui acquisite, che comprendono metodi per interfacciarsi con l'utente, operatori, metodi matematici di base e strutture di controllo.

Questo script elenca i numeri primi contenuti in un certo intervallo.

```
def sePrimo(n)
{
    if (n <= 1) println('')
    else
    {
        int x = n.intdiv(2)
        while (x > 1)
        {
            if (n % x == 0) break
            x = x - 1
        }
        if (x == 1) println(n.toString())
    }
}
println('RICERCA NUMERI PRIMI IN UN INTERVALLO')
def minore = System.console().readLine('Inizio intervallo ').toInteger()
def maggiore = System.console().readLine('Fine intervallo '.toInteger()
for (int i = minore; i < maggiore; i = i + 1)
{
    sePrimo(i)
}
```

Lo script definisce il metodo `sePrimo()` che, secondo un semplice algoritmo che non ho inventato io ma ho trovato in rete, verifica se un numero è un numero primo e, se si, lo stampa. Poi vengono chiesti all'utente gli estremi dell'intervallo di ricerca e il metodo `sePrimo()` viene applicato ai numeri contenuti nell'intervallo.

Quest'altro script calcola il fattoriale di un numero (corrispondente al prodotto di quel numero per il prodotto di tutti i numeri interi che lo precedono) applicando la precisione arbitraria in modo da ottenere e scrivere anche numeri molto grandi, di svariate centinaia di cifre.

```
def BigInteger fattoriale(int n)
{
    if(n==0) 1
    else n * fattoriale(n-1)
}
println 'Inserisci il numero di cui calcolare il fattoriale:'
n = System.in.newReader().readLine() as int
println fattoriale(n)
```

Qui abbiamo un bel metodo ricorsivo per calcolare il fattoriale definito in modo tale che ritorni un BigInteger, cioè un intero in precisione arbitraria, generandolo da un normale intero. Poi viene chiesto all'utente di indicare il numero di cui calcolare il fattoriale e viene stampato il risultato dell'applicazione del metodo al numero indicato.

Sull'onda del successo di questo script con metodo ricorsivo per calcolare il fattoriale di un numero viene voglia di provare la stessa tecnica ricorsiva con un altro algoritmo, quello per trovare il numero corrispondente ad una certa posizione nella successione di Fibonacci (quella successione nella quale un numero corrisponde alla somma dei due numeri che lo precedono).

```
def BigInteger fibonacci(int n)
{
    if (n <= 1) return n
    fibonacci(n - 1) + fibonacci(n - 2)
}
println 'Indica la posizione nella successione di Fibonacci'
n = System.in.newReader().readLine() as BigInteger
println "Nella posizione $n della successione di Fibonacci abbiamo il numero:"
println fibonacci(n)
```

Anche qui abbiamo un metodo ricorsivo per calcolare il valore cercato e un dialogo per avere dall'utente indicazione su quale valore cercare.
Visto che la successione di Fibonacci dopo un po', anche se in maniera meno esplosiva del fattoriale, trova numeri di una certa dimensione, lo script prevede la precisione arbitraria utilizzando il tipo BigInteger.
Precauzione inutile in quanto il metodo ricorsivo utilizzato è molto inefficiente a causa del grande numero di ricalcoli che comporta il passaggio `fibonacci(n - 1) + fibonacci(n - 2)`. Tanto che dalla trentesima posizione in poi occorre sempre più tempo per ottenere il risultato, fino ad attese impossibili.

Tutto cambia se utilizziamo un normale metodo basato sul ciclo `for`:

```
def BigInteger fibonacci(int n)
{
    if (n <= 1) return n
    BigInteger a = 0, b = 1
    for (int i = 2; i <= n; i = i + 1)
    {
        BigInteger t = a + b
        a = b
        b = t
    }
    return b
}
println 'Indica la posizione nella successione di Fibonacci'
n = System.in.newReader().readLine() as BigInteger
println "Nella posizione $n della successione di Fibonacci abbiamo il numero:"
println fibonacci(n)
```

Con lo scorrimento di valori già calcolati attraverso la variabile t nel ciclo evitiamo ricalcoli ed otteniamo di poter calcolare in un attimo anche i valori in posizioni molto lontane.

Un altro script che potrebbe essere di qualche utilità possiamo dedicarlo al calcolo combinatorio, cioè al calcolo di permutazioni, combinazioni e disposizioni.

Per le permutazioni semplici, visto che esse corrispondono al fattoriale del numero di elementi da permutare, possiamo utilizzare lo script per il calcolo del fattoriale che abbiamo visto prima.

Per Combinazioni e Disposizioni, semplici senza ripetizione, di n numeri presi k a k dobbiamo le seguenti formule
$C(n,k) = \frac{n!}{(n-k)!k!}$ per le combinazioni,
$D(n,k) = \frac{n!}{(n-k)!}$ per le disposizioni,
nelle quali ricorre spesso il calcolo del fattoriale.

Il fatto che la necessità di questo calcolo ricorra spesso nelle formule e comporti una non semplice scrittura di istruzioni per effettuarlo suggerisce di isolare queste istruzioni in un metodo dedicata al calcolo del fattoriale in modo da poter richiamare queste istruzioni in blocco quando serve.

Lo script per calcolare Combinazioni e Disposizioni potrebbe essere così concepito

```
def BigInteger fattoriale(int n)
{
    if(n==0) 1
    else n * fattoriale(n-1)
}
println 'Quanti elementi utilizzi?'
n = System.in.newReader().readLine() as int
println 'Con quanti elementi formi i raggruppamenti?'
k = System.in.newReader().readLine() as int
c = fattoriale(n)/(fattoriale(n-k) * fattoriale(k))
d = fattoriale(n)/fattoriale(n-k)
println("combinazioni:  $c")
println("disposizioni:  $d")
```

Lanciando questo script saremo richiesti di indicare il numero di elementi utilizzati (n), il numero di elementi per i raggruppamenti (k) e in un attimo vedremo il risultato.

* * *

Fin qui abbiamo utilizzato strumenti che sono stati presentati nel manualetto che sto scrivendo.

Tuttavia Groovy ha a disposizione tutte le librerie Java, anche quelle che non sono disponibili per default e che dobbiamo importare se vogliamo utilizzarle, ovviamente conoscendo i metodi che contengono e come usarli.

Per esempio, se volessimo creare uno script per calcolare i giorni che decorrono tra una data e l'altra avremmo bisogno dei metodi della libreria `java.time`, che contiene metodi per elaborare ore, minuti, giorni, anni, ecc., tutto ciò che ha relazione con il tempo (time) e lo script potrebbe essere il seguente

```
import java.time.*
println 'CALCOLIAMO LA DIFFERENZA TRA DATE'
println 'Prima data:'
print 'giorno:  '
g1 = System.in.newReader().readLine() as int
print 'mese:  '
m1 = System.in.newReader().readLine() as int
print 'anno:  '
a1 = System.in.newReader().readLine() as int
println 'Seconda data:'
print 'giorno:  '
g2 = System.in.newReader().readLine() as int
print 'mese:  '
m2 = System.in.newReader().readLine() as int
print 'anno:  '
a2 = System.in.newReader().readLine() as int
data1 = LocalDate.of(a1,m1,g1)
data2 = LocalDate.of(a2,m2,g2)
differenza = Math.abs(data2.toEpochDay() - data1.toEpochDay())
println "La differenza tra le due date è di $differenza giorni"
```

Lo script sarebbe molto più compatto se utilizzassimo un'altra classe java, la classe `Scanner` del package `java.utils` che semplificherebbe la lunga parte dello script dedicata all'acquisizione delle due date tra cui calcolare la differenza.
Ho acquisito la differenza tra le date come valore assoluto in modo da far risultare positivo il numero di giorni intercorrenti tra le date anche nel caso che la seconda data indicata sia antecedente alla prima.

Chi conosca i segreti del package `jama`, sviluppato per Java dal National Institute of Standards and Technology, dedicato all'algebra lineare potrebbe produrre il seguente script per risolvere sistemi di equazioni lineari

```
@Grab(group='gov.nist.math', module='jama', version='1.0.3')
import Jama.*
println 'RISOLUZIONE DI UN SISTEMA LINEARE'
println 'Quante equazioni?'
n = System.in.newReader().readLine() as int
i = 1
l = []
while (i <= n)
{
    println "Coefficienti dell'equazione $i:"
    ii = 1
    ll = []
    r = 1..n
    while (ii <= n)
    {
        println "Coefficiente $ii:  "
        ll.add(System.in.newReader().readLine() as double)
        ii = ii + 1
    }
    i = i + 1
    l.add(ll)
}
double[][] dati_A = l
Matrix A = new Matrix(dati_A)
l = []
ii = 1
ll = []
r = 1..n
while (ii <= n)
{
    println "Termine noto $ii:  "
    ll.add(System.in.newReader().readLine() as double)
    l.add(ll)
    ll = []
    ii = ii + 1
}
double[][] dati_b = l
Matrix b = new Matrix(dati_b)
Matrix x = A.solve(b)
println 'Soluzione:'
x.print(10,3)
```

Come sempre le indentature non servono ma le ho usate per rendere più chiara la struttura dello script.

La strana dicitura che precede l'importazione del package è dovuta al fatto che si tratta di un package che non è nel PATH di Java ed occorre inserirvelo per poterlo importare con il comando nella riga successiva.

Molto laboriosa la scrittura del codice per l'acquisizione dei coefficienti delle variabili nelle equazioni e dei termini noti per la costruzione della matrice del sistema e del vettore verticale dei termini noti, secondo la non semplice sintassi prevista da Jama, per poi ottenere la soluzione.

Altrettanto laboriosa e lunga la fase di inserimento dei dati, specialmente se abbiamo molte equazioni nel sistema, ma molto efficiente il ritrovamento della soluzione che avviene in un attimo, anche in presenza di sistemi con molte equazioni, grazie al metodo `solve()`.

Il vettore che contiene la soluzione del sistema è dotato di un metodo `print()` che accetta due parametri di formattazione: il primo può servire per allineare i valori sulla destra nello spazio di caratteri indicato (se non serve allineamento inserire 0), il secondo indica il numero delle cifre decimali da esporre.

13 Utilità di un IDE

Per realizzare semplici script possiamo benissimo farcela con un qualsiasi editor di testo.

Se lo script è complicato dalla presenza di funzionalità non ricorrenti o se utilizziamo librerie java di cui abbiamo conoscenza superficiale è bello ricorrere ad un IDE (Integrated Development Environment) che, se abilitato a Groovy, ci offre la code completion scrivendo con quel linguaggio.

In questo modo possiamo riscontrare l'esatta dicitura di un comando che non ricordiamo, evitare di ricercare regole sintattiche su manuali, avere contezza dei parametri che è necessario inserire richiamando un metodo, ecc.

A questo proposito segnalo un paio di IDE con i quali si programma molto bene in Groovy.

13.1 Intellji Idea

Anche la versione gratuita Community è abilitata per default a trattare progetti Groovy e dobbiamo fare questa scelta all'apertura del nuovo progetto.

Se vogliamo evitare di complicarci la vita e vogliamo semplicemente produrre uno script, del tipo di quelli visti in questo manualetto, una volta aperto il progetto, cliccando destro sulla voce `src` nella zona di sinistra dell'IDE, dove abbiamo la struttura del progetto, scegliamo NEW ▷ GROOVY SCRIPT, indicando poi il nome da dare allo script stesso, cui sarà di default assegnata l'estensione `.groovy`.

In questo modo l'editor è dedicato a Groovy e non appena abbiamo scritto le prime lettere di un comando ce ne viene suggerito il completamento o indicate le scelte da fare per il completamento.

Inoltre, una volta creato un oggetto, scrivendo il nome di quell'oggetto seguito da un punto (.) apriamo un menu a discesa che elenca tutti i metodi collegati a quell'oggetto, dandoci modo di scegliere quello che ci serve.

Se lavoriamo con Groovy avendo importato una libreria Java tutto ciò avviene anche per i metodi e gli oggetti della libreria Java importata.

13.2 Eclipse

La versione di Eclipse per Java può essere abilitata a Groovy installando il relativo plugin in questo modo:
. da menu HELP ▷ ECLIPSE MARKETPLACE... aprire il marketplace, verificare che sia installato il plugin `Eclipse Marketplace Client` e se non è installato installarlo,
. attraverso la finestrella FIND ricercare la parola `groovy` e installare il plugin `Groovy Development Tools` che comparirà come risultato della ricerca.

In questo modo abbiamo abilitato Eclipse a trattare progetti Groovy.

Apriamo un nuovo progetto Groovy con menu NEW ▷ OTHER ▷ GROOVY ▷ GROOVY PROJECT.

Sempre con l'obiettivo di produrre semplicemente uno script del tipo di quelli visti in questo manualetto, cliccando destro sulla voce `src` nella zona di sinistra dell'IDE, dove abbiamo la struttura del progetto, scegliamo NEW ▷ FILE, indicando poi il nome da dare allo script stesso con l'estensione `.groovy` (dobbiamo scrivere noi l'estensione in quanto Eclipse non la assegna automaticamente come fa Idea).

In questo modo l'editor è dedicato a Groovy e non appena abbiamo scritto le prime lettere di un comando, con la pressione contemporanea dei tasti Ctrl e barra spaziatrice ce ne viene suggerito il completamento o indicate le scelte da fare per il completamento.

Inoltre, una volta creato un oggetto, scrivendo il nome di quell'oggetto seguito da un punto (.) apriamo un menu a discesa che elenca tutti i metodi collegati a quell'oggetto, dandoci modo di scegliere quello che ci serve.

Se lavoriamo con Groovy avendo importato una libreria Java tutto ciò avviene anche per i metodi e gli oggetti della libreria Java importata.