Laboratorio di Elementi di Bioinformatica

Laurea Triennale in Informatica
(codice: E3101Q116)

AA 2016/2017

Linguaggio Ruby:
tipi standard, array e hash

Docente: Raffaella Rizzi

Introduzione a Ruby

Un codice in linguaggio Ruby deve essere contenuto in un file di puro testo che ha estensione `rb`.

Per eseguirlo basta digitare dalla *shell*:

```
>ruby <nome_del_file>
```

Introduzione a Ruby

Un codice in linguaggio Ruby deve essere contenuto in un file di puro testo che ha estensione `rb`.

Per eseguirlo basta digitare dalla *shell*:

```
>ruby <nome_del_file>
```

Per accedere alla documentazione di Ruby:

```
http://ruby-doc.org/core-2.2.3/
```

La funzione `puts`

```
puts arg1, arg2, arg3, …, argN
```

Cosa fa `puts`?

- Valuta come espressione ognuno degli *N* argomenti *arg1, …, argN* e converte in stringa il risultato
- Produce in standard output le stringhe così ottenute, stampando un carattere di *newline* dopo ognuna di esse.

La funzione `puts`

```
puts arg1, arg2, arg3, …, argN
```

Cosa fa `puts`?

- Valuta come espressione ognuno degli *N* argomenti *arg1, …, argN* e converte in stringa il risultato
- Produce in standard output le stringhe così ottenute, stampando un carattere di *newline* dopo ognuna di esse.

```
puts 4
```

```
>4
>
```

La funzione `puts`

```
puts arg1, arg2, arg3, …, argN
```

Cosa fa `puts`?

- Valuta come espressione ognuno degli *N* argomenti *arg1, …, argN* e converte in stringa il risultato
- Produce in standard output le stringhe così ottenute, stampando un carattere di *newline* dopo ognuna di esse.

```
puts 1+3
```

```
>4
>
```

La funzione `puts`

```
puts arg1, arg2, arg3, …, argN
```

Cosa fa `puts`?

- Valuta come espressione ognuno degli *N* argomenti *arg1, …, argN* e converte in stringa il risultato
- Produce in standard output le stringhe così ottenute, stampando un carattere di *newline* dopo ognuna di esse.

```
puts “Somma di 1 e 3”, 1+3
```

```
>Somma di 1 e 3
>4
>
```

La funzione `print`

```
print arg1, arg2, arg3, …, argN
```

Cosa fa `print`?

- Valuta come espressione ognuno degli *n* argomenti *arg1, …, argN* e converte in stringa il risultato
- Produce in standard output le stringhe così ottenute.

La funzione `print`

```
print arg1, arg2, arg3, …, argN
```

Cosa fa `print`?

- Valuta come espressione ognuno degli *n* argomenti *arg1, …, argN* e converte in stringa il risultato
- Produce in standard output le stringhe così ottenute.

```
print “Somma di 1 e 3 = ”, 1+3
```

```
>Somma di 1 e 3 = 4
>
```

Le “cose” da sapere su Ruby

- Ruby è un linguaggio *object-oriented* e tutto ciò che viene manipolato o restituito è un (riferimento a) oggetto. Quindi:
  - si avrà a che fare con classi, oggetti, istanze, costruttori e metodi

Le “cose” da sapere su Ruby

- Ruby è un linguaggio *object-oriented* e tutto ciò che viene manipolato o restituito è un (riferimento a) oggetto. Quindi:
  - si avrà a che fare con classi, oggetti, istanze, costruttori e metodi
- Il “contenuto” di una variabile è sempre un riferimento ad un oggetto (anche nel caso di valore numerico o stringa)

Le “cose” da sapere su Ruby

- Ruby è un linguaggio *object-oriented* e tutto ciò che viene manipolato o restituito è un (riferimento a) oggetto. Quindi:
  - si avrà a che fare con classi, oggetti, istanze, costruttori e metodi
- Il “contenuto” di una variabile è sempre un riferimento ad un oggetto (anche nel caso di valore numerico o stringa)
- Tutto è considerato vero tranne il valore booleano falso (`false`) e il valore nullo (`nil`)

Le “cose” da sapere su Ruby

- Ruby è un linguaggio *object-oriented* e tutto ciò che viene manipolato o restituito è un (riferimento a) oggetto. Quindi:
  - si avrà a che fare con classi, oggetti, istanze, costruttori e metodi
- Il “contenuto” di una variabile è sempre un riferimento ad un oggetto (anche nel caso di valore numerico o stringa)
- Tutto è considerato vero tranne il valore booleano falso (`false`) e il valore nullo (`nil`)

Il valore `nil` è un oggetto appartenente alla classe `NilClass`

Le “cose” da sapere su Ruby

- Ruby è un linguaggio *object-oriented* e tutto ciò che viene manipolato o restituito è un (riferimento a) oggetto. Quindi:
  - si avrà a che fare con classi, oggetti, istanze, costruttori e metodi
- Il “contenuto” di una variabile è sempre un riferimento ad un oggetto (anche nel caso di valore numerico o stringa)
- Tutto è considerato vero tranne il valore booleano falso (`false`) e il valore nullo (`nil`)
- I commenti iniziano con # e terminano alla fine della riga

Le “cose” da sapere su Ruby

- Le variabili non vengono dichiarate: incominciano ad esistere nel momento in cui si assegna loro un valore

Le “cose” da sapere su Ruby

- Le variabili non vengono dichiarate: incominciano ad esistere nel momento in cui si assegna loro un valore
- Le variabili non sono tipizzate: il loro tipo viene determinato automaticamente nel momento dell’assegnamento di un valore

Le “cose” da sapere su Ruby

- Le variabili non vengono dichiarate: incominciano ad esistere nel momento in cui si assegna loro un valore
- Le variabili non sono tipizzate: il loro tipo viene determinato automaticamente nel momento dell’assegnamento di un valore

```
a = 1245
puts a
```

```
>1245
>
```

Le “cose” da sapere su Ruby

- Le variabili non vengono dichiarate: incominciano ad esistere nel momento in cui si assegna loro un valore
- Le variabili non sono tipizzate: il loro tipo viene determinato automaticamente nel momento dell’assegnamento di un valore

```
a = 1245
puts a
```

```
>1245
>
```

- Il ‘;’ che termina un’istruzione è opzionale (è però obbligatorio se l’istruzione non è l’ultima della riga)

Le “cose” da sapere su Ruby

- Le parentesi tonde che racchiudono i parametri attuali di un metodo/funzione sono opzionali (tranne quando servono per disambiguare invocazioni annidate)

Le “cose” da sapere su Ruby

- Le parentesi tonde che racchiudono i parametri attuali di un metodo/funzione sono opzionali (tranne quando servono per disambiguare invocazioni annidate)

```
a = 1245
puts a
```

equivale a

```
a = 1245
puts(a)
```

Classi e oggetti

In Ruby tutto è un oggetto…

- Una classe rappresenta un insieme di oggetti che condividono proprietà e funzionalità.
- Una classe è caratterizzata da variabili e metodi che rappresentano rispettivamente le proprietà e le funzionalità dei suoi oggetti
- Un'istanza di una determinata classe è un particolare oggetto dell'insieme che la classe rappresenta
- Un oggetto di una determinata classe viene istanziato attraverso il cosiddetto costruttore
- Ad ogni oggetto che viene istanziato è associato un riferimento che può essere assegnato ad una variabile

Classi e oggetti

- Una variabile di istanza è una variabile il cui valore dipende dalla particolare istanza (oggetto) della classe: ogni istanza ha un proprio valore della variabile. L'insieme dei valori delle variabili di istanza di un oggetto definiscono il suo stato
- Una variabile di classe è una variabile il cui valore è "indipendente" dalla particolare istanza della classe, cioé il suo valore è uguale per tutte le istanze della classe
- Un metodo di istanza è un metodo che viene invocato da una particolare istanza della classe (e in genere agisce sul suo stato); per invocare un metodo di istanza occorre quindi istanziare la classe
- Un metodo di classe è un metodo che viene invocato senza dover istanziare la classe

Creazione di un oggetto e invocazione di un metodo

Costruzione di un oggetto (istanza di una classe)

```
obj_ref = ClassName.new(val1, val2, …, valN)
```

Invocazione di un metodo di un oggetto

```
obj_ref.method_name(parameters)
```

Creazione di un oggetto e invocazione di un metodo

Costruzione di un oggetto (istanza di una classe)

```
obj_ref = ClassName.new(val1, val2, …, valN)
```

ClassName è il nome della classe da istanziare e `new` è il costruttore

Invocazione di un metodo di un oggetto

```
obj_ref.method_name(parameters)
```

Creazione di un oggetto e invocazione di un metodo

Costruzione di un oggetto (istanza di una classe)

```
obj_ref = ClassName.new(val1, val2, …, valN)
```

val1, *val2*, …, *valN* sono i valori passati alle variabili di istanza

Invocazione di un metodo di un oggetto

```
obj_ref.method_name(parameters)
```

Creazione di un oggetto e invocazione di un metodo

Costruzione di un oggetto (istanza di una classe)

```
obj_ref = ClassName.new(val1, val2, …, valN)
```

obj_ref è la variabile che contiene il riferimento all'oggetto istanziato

Invocazione di un metodo di un oggetto

```
obj_ref.method_name(parameters)
```

I tipi standard

- Valori booleani
  - `true` e `false`

```
b = true
puts b
```

```
>true
>
```

I tipi standard

- Valori booleani
  - `true` e `false`

```
b = true
puts b
```

```
>true
>
```

- Numeri interi

```
i = 1301
puts i
```

```
>1301
>
```

I tipi standard

- Numeri decimali

```
d = 13.01
puts d
```

```
>13.01
>
```

I tipi standard

- Numeri decimali

```
d = 13.01
puts d
```

```
>13.01
>
```

- Numeri complessi
  - composti da una parte reale e una parte immaginaria
  - esempio: *2+3i*

I tipi standard

- Numeri decimali

```
d = 13.01
puts d
```

```
>13.01
>
```

- Numeri complessi
  - composti da una parte reale e una parte immaginaria
  - esempio: *2+3i*
- Numeri razionali
  - frazioni = rapporto tra due numeri interi
  - esempio: *3/4*

I tipi standard

- Stringhe
  - sequenze di caratteri

```
s = “Hello world!”
puts s
```

```
>Hello world!
>
```

Valori booleani

I valori booleani sono true e false (vero o falso) e vengono creati attraverso i letterali `true` e `false`.

```
true_value = true
false_value = false
```

`true_value` contiene il riferimento a un oggetto della classe `TrueClass`, mentre `false_value` contiene il riferimento a un oggetto della classe `FalseClass`.

Numeri interi

I numeri interi possono essere di qualsiasi lunghezza (dipende dalla memoria disponibile sul sistema); fino ad un certo valore (numeri “piccoli”) sono oggetti della classe `Fixnum`, oltre un certo valore (numeri “grandi”) sono oggetti della classe `Bignum`

```
small_int = 245667589600
big_int = 24566758960089658688888888888
new_int = small_int*big_int
small_int = small_int*big_int
```

- `small_int` è un oggetto della classe `Fixnum`, mentre `big_int` è della classe `Bignum`.
- `new_int` (prodotto di un `Fixnum` e di un `Bignum`) è di classe `Bignum`.
- `small_int` dopo l’ultima istruzione si trasforma in `Bignum`.

Numeri interi

I numeri interi possono essere di qualsiasi lunghezza (dipende dalla memoria disponibile sul sistema); fino ad un certo valore (numeri “piccoli”) sono oggetti della classe `Fixnum`, oltre un certo valore (numeri “grandi”) sono oggetti della classe `Bignum`

```
small_int = 245667589600
big_int = 24566758960089658688888888888
new_int = smal
small_int = sm
```

Le classi `Fixnum` e `Bignum` hanno la classe `Integer` come padre

- `small_int` è un oggetto della classe `Fixnum`, mentre `big_int` è della classe `Bignum`.
- `new_int` (prodotto di un `Fixnum` e di un `Bignum`) è di classe `Bignum`.
- `small_int` dopo l’ultima istruzione si trasforma in `Bignum`.

Numeri interi

I numeri interi vengono creati attraverso un letterale (sequenza di simboli) composto da:

- cifre da 0 a 9
- simbolo underscore '_' (che viene ignorato)
- simbolo '-' all'inizio che effettua la negazione aritmetica

```
num1 = 245667
num2 = 245_667
num3 = -245667
puts num1
puts num2
puts num3
```

```
>245667
>245667
>-245667
>
```

Numeri interi

I numeri interi vengono creati attraverso un letterale (sequenza di simboli) composto da:

- cifre da 0 a 9
- simbolo underscore '_' (che viene ignorato)
- simbolo '-' all'inizio che effettua la negazione aritmetica

```
num1 = 245667
num2 = 245_667
num3 = -245667
puts num1
puts num2
puts num3
```

Di default questi letterali vengono interpretati come interi di base 10

```
>245667
>245667
>-245667
>
```

Numeri interi

Esistono simboli opzionali da anteporre al letterale in maniera che venga interpretato in:

- base 8 se il letterale è preceduto dal simbolo 0
- base 16 se il letterale è preceduto dai simboli 0x
- base 2 se il letterale è preceduto dai simboli 0b

Numeri interi

```
num = 84919      #intero in base 10
puts num
```

```
>84919
>
```

Numeri interi

```
num = 84919      #intero in base 10
puts num
```

```
>84919
>
```

```
num = 0245667    #intero in base 8
puts num
```

```
>84919
>
```

Numeri interi

```
num = 0b100111      #intero in base 2
puts num
```

```
>39
>
```

Numeri interi

```
num = 0b100111      #intero in base 2
puts num
```

```
>39
>
```

```
num = 0x456afb      #intero in base 16
puts num
```

```
>4549371
>
```

Numeri decimali

I numeri decimali sono oggetti della classe `Float`

```
d1 = 1356.564
```

e vengono creati attraverso un letterale composto da:

- cifre da 0 a 9
- simbolo underscore '_' (che viene ignorato)
- simbolo '.' come separatore
- simbolo '-' all'inizio che effettua la negazione aritmetica
- simbolo 'e' per l'esponente

```
d2 = 1.34e3      #1340
```

Numeri decimali

Attenzione! Il separatore ‘.’ deve essere preceduto e seguito da una cifra

Numeri decimali

Attenzione! Il separatore '.' deve essere preceduto e seguito da una cifra

```
d3 = 1.e3
```

Si intende assegnare alla variabile d3 il valore 1000. Questa istruzione genera errore?

Numeri decimali

Attenzione! Il separatore '.' deve essere preceduto e seguito da una cifra

```
d3 = 1.e3
```

Viene generato un messaggio di errore perché si sta tentando di accedere al metodo **`e3`** dell'oggetto **1** di classe `Fixnum`

Numeri decimali

Attenzione! Il separatore '.' deve essere preceduto e seguito da una cifra

```
d3 = 1.0e3
```

Assegnamento corretto

Numeri decimali

Attenzione! Il separatore '.' deve essere preceduto e seguito da una cifra

```
d3 = 1.0e3
```

Assegnamento corretto

Attenzione!

```
num = 10e2
```

La variabile `num` viene interpretata come decimale e quindi è un oggetto di classe `Float`

Classe `Numeric`

Tutti i numeri (interi, decimali, razionali e complessi) sono in generale istanze della classe `Numeric`, che mette a disposizione una serie di metodi, tra i quali:

Classe `Numeric`

Tutti i numeri (interi, decimali, razionali e complessi) sono in generale istanze della classe `Numeric`, che mette a disposizione una serie di metodi, tra i quali:

- `abs`: restituisce il valore assoluto dell'oggetto invocante

Classe `Numeric`

Tutti i numeri (interi, decimali, razionali e complessi) sono in generale istanze della classe `Numeric`, che mette a disposizione una serie di metodi, tra i quali:

- `abs`: restituisce il valore assoluto dell'oggetto invocante
- `ceil`: restituisce il più piccolo intero maggiore o uguale all'oggetto invocante

Classe `Numeric`

Tutti i numeri (interi, decimali, razionali e complessi) sono in generale istanze della classe `Numeric`, che mette a disposizione una serie di metodi, tra i quali:

- `abs`: restituisce il valore assoluto dell'oggetto invocante
- `ceil`: restituisce il più piccolo intero maggiore o uguale all'oggetto invocante
- `floor`: restituisce il più grande intero minore o uguale all'oggetto invocante

Classe `Numeric`

Tutti i numeri (interi, decimali, razionali e complessi) sono in generale istanze della classe `Numeric`, che mette a disposizione una serie di metodi, tra i quali:

- `abs`: restituisce il valore assoluto dell'oggetto invocante
- `ceil`: restituisce il più piccolo intero maggiore o uguale all'oggetto invocante
- `floor`: restituisce il più grande intero minore o uguale all'oggetto invocante
- `integer?`: restituisce `true` se l'oggetto invocante è un intero, altrimenti `false`

Classe `Numeric`

Tutti i numeri (interi, decimali, razionali e complessi) sono in generale istanze della classe `Numeric`, che mette a disposizione una serie di metodi, tra i quali:

- `abs`: restituisce il valore assoluto dell'oggetto invocante
- `ceil`: restituisce il più piccolo intero maggiore o uguale all'oggetto invocante
- `floor`: restituisce il più grande intero minore o uguale all'oggetto invocante
- `integer?`: restituisce `true` se l'oggetto invocante è un intero, altrimenti `false`
- `zero?`: restituisce `true` se l'oggetto invocante è uguale al valore 0, altrimenti `false`

Classe `Numeric`

Tutti i numeri (interi, decimali, razionali e complessi) sono in generale istanze della classe `Numeric`, che mette a disposizione una serie di metodi, tra i quali:

- `abs`: restituisce il valore assoluto dell'oggetto invocante
- `ceil`: restituisce il più piccolo intero maggiore o uguale all'oggetto invocante
- `floor`: restituisce il più grande intero minore o uguale all'oggetto invocante
- `integer?`: restituisce `true` se l'oggetto invocante è un intero, altrimenti `false`
- `zero?`: restituisce `true` se l'oggetto invocante è uguale al valore 0, altrimenti `false`
- `to_int`: converte l'oggetto invocante in un valore intero

Classe Numeric

```
n = -456
puts n.abs
n = 123.56
puts n.ceil
puts n.floor
puts n.integer?
puts n.zero?
puts n.to_int
```

```
>456
>124
>123
>false
>false
>123
>
```

Stringhe

Una stringa è una sequenza di simboli

```
stringa1 = “Hello world!”
stringa2 = ‘Ciao mondo!’
```

e viene creata attraverso un letterale composto dai simboli della stringa racchiusi da doppi apici “ o singoli apici ‘ (che non fanno parte del valore della stringa).
Di default Ruby è basato sul codice ASCII; se si vogliono includere anche i simboli di Unicode, si deve la riga
`# encoding: utf-8` all’inizio del file di codice

```
# encoding: utf-8

puts “Hello world!”
```

Stringhe

- Se la stringa è tra singoli apici, allora:
  - avviene la sostituzione di \\ con \ e di \' con '
  - si può inserire il simbolo “

```
puts '"Hello" \\world\''
```

Stringhe

- Se la stringa è tra singoli apici, allora:
  - avviene la sostituzione di \\ con \ e di \' con'
  - si può inserire il simbolo "

```
puts '"Hello" \\world\''
```

```
>"Hello" \world'
>
```

Stringhe

- Se la stringa è tra singoli apici, allora:
  - avviene la sostituzione di `\\` con `\` e di `\'` con `'`
  - si può inserire il simbolo “

```
puts '"Hello" \\world\''
```

```
>"Hello" \world'
>
```

- Se la stringa è tra doppi apici, allora:
  - avviene la sostituzione di `\\` con `\`, di `\'` con `'` e di `\n` con il *newline*

```
puts "Hello\nworld", 'Hello\nworld'
```

Stringhe

- Se la stringa è tra singoli apici, allora:
  - avviene la sostituzione di `\\` con `\` e di `\'` con `'`
  - si può inserire il simbolo "

```
puts '"Hello" \\world\''
```

```
>"Hello" \world'
>
```

- Se la stringa è tra doppi apici, allora:
  - avviene la sostituzione di `\\` con `\`, di `\'` con `'` e di `\n` con il *newline*

```
puts "Hello\nworld", 'Hello\nworld'
```

```
>Hello
>world
>Hello\nworld
>
```

Stringhe

- Con i doppi apici si può usare il costrutto `#{expr}` per sostituire il risultato della valutazione di una qualsiasi espressione

Stringhe

- Con i doppi apici si può usare il costrutto `#{expr}` per sostituire il risultato della valutazione di una qualsiasi espressione

```
a = 1789
b = 675
puts “Il risultato di a+b e’ #{a+b}“
puts “Il risultato di #{a}+#{b} e’ #{a+b}“
puts ‘#{a} non viene sostituito tra singoli apici \’’
```

Stringhe

- Con i doppi apici si può usare il costrutto `#{expr}` per sostituire il risultato della valutazione di una qualsiasi espressione

```
a = 1789
b = 675
puts “Il risultato di a+b e’ #{a+b}“
puts “Il risultato di #{a}+#{b} e’ #{a+b}“
puts ‘#{a} non viene sostituito tra singoli apici \’’
```

```
>Il risultato di a+b e’ 2464
>Il risultato di 1789+675 e’ 2464
>#{a} non viene sostituito tra singoli apici ‘
>
```

Stringhe

Qualcosa in più sulle stringhe…

Stringhe

Qualcosa in più sulle stringhe…

- Le stringhe sono oggetti della classe `String`

Stringhe

Qualcosa in più sulle stringhe…

- Le stringhe sono oggetti della classe `String`
- I simboli in una stringa vengono indicizzati a partire da 0

Stringhe

Qualcosa in più sulle stringhe…

- Le stringhe sono oggetti della classe `String`
- I simboli in una stringa vengono indicizzati a partire da 0
- Si può accedere al singolo carattere in una stringa attraverso la sintassi *`string_name`*`[`*`index`*`]`, dove *`index`* è l'indice del carattere a cui si vuole accedere

Stringhe

Qualcosa in più sulle stringhe…

- Le stringhe sono oggetti della classe `String`
- I simboli in una stringa vengono indicizzati a partire da 0
- Si può accedere al singolo carattere in una stringa attraverso la sintassi *`string_name`*`[`*`index`*`]`, dove *`index`* è l'indice del carattere a cui si vuole accedere
- Il singolo carattere è trattato come una stringa di lunghezza 1

Stringhe

Qualcosa in più sulle stringhe…

- Le stringhe sono oggetti della classe `String`
- I simboli in una stringa vengono indicizzati a partire da 0
- Si può accedere al singolo carattere in una stringa attraverso la sintassi *`string_name`*`[`*`index`*`]`, dove *`index`* è l'indice del carattere a cui si vuole accedere
- Il singolo carattere è trattato come una stringa di lunghezza 1

```
str = “Hello world!”
puts str
str[0] = “h”
puts str
```

Stringhe

Qualcosa in più sulle stringhe…

- Le stringhe sono oggetti della classe `String`
- I simboli in una stringa vengono indicizzati a partire da 0
- Si può accedere al singolo carattere in una stringa attraverso la sintassi *`string_name`*`[`*`index`*`]`, dove *`index`* è l'indice del carattere a cui si vuole accedere
- Il singolo carattere è trattato come una stringa di lunghezza 1

```
str = “Hello world!”
puts str
str[0] = “h”
puts str
```

```
>Hello world!
>hello world!
>
```

Stringhe

Qualcosa in più sulle stringhe:

Stringhe

Qualcosa in più sulle stringhe:

- La somma di due stringhe produce la loro concatenazione

Stringhe

Qualcosa in più sulle stringhe:

- La somma di due stringhe produce la loro concatenazione
- Il prodotto di una stringa con un intero n (e non viceversa) produce la ripetizione della stringa per n volte

Stringhe

Qualcosa in più sulle stringhe:

- La somma di due stringhe produce la loro concatenazione
- Il prodotto di una stringa con un intero n (e non viceversa) produce la ripetizione della stringa per n volte

```
str1 = “Hello”
str2 = “ world!”
puts str1+str2
str3 = str1*3
puts str3
```

Stringhe

Qualcosa in più sulle stringhe:

- La somma di due stringhe produce la loro concatenazione
- Il prodotto di una stringa con un intero n (e non viceversa) produce la ripetizione della stringa per n volte

```
str1 = “Hello”
str2 = “ world!”
puts str1+str2
str3 = str1*3
puts str3
```

```
>Hello world!
>HelloHelloHello
>
```

Stringhe

Qualcosa in più sulle stringhe:

- La somma di due stringhe produce la loro concatenazione
- Il prodotto di una stringa con un intero n (e non viceversa) produce la ripetizione della stringa per n volte

```
str1 = “Hello”
str2 = “ world!”
puts str1+str2
str3 = str1*3
puts str3
```

Attenzione! Scrivere `3*str1` genera un errore.

```
>Hello world!
>HelloHelloHello
>
```

Alcuni metodi della classe `String`...

```
string_name.downcase!
```

Il metodo `downcase!` converte la stringa in minuscolo

Alcuni metodi della classe `String`...

```
string_name.downcase!
```

Il metodo `downcase!` converte la stringa in minuscolo

```
string_name.upcase!
```

Il metodo `upcase!` converte la stringa in maiuscolo

Alcuni metodi della classe `String`...

```
string_name.downcase!
```

Il metodo `downcase!` converte la stringa in minuscolo

```
string_name.upcase!
```

Il metodo `upcase!` converte la stringa in maiuscolo

```
stringa = “HELLO”
puts stringa.downcase!
```

Alcuni metodi della classe `String`...

```
string_name.downcase!
```

Il metodo `downcase!` converte la stringa in minuscolo

```
string_name.upcase!
```

Il metodo `upcase!` converte la stringa in maiuscolo

```
stringa = “HELLO”
puts stringa.downcase!
```

```
>hello
>
```

Metodi e funzioni

Un metodo/funzione è un blocco di codice a cui viene associato un nome. Il nome viene usato per invocare il metodo (cioé eseguire il blocco di codice). Un metodo restituisce il risultato dell'ultima istruzione (valutata come espressione) che viene eseguita.

Metodi e funzioni

Un metodo/funzione è un blocco di codice a cui viene associato un nome. Il nome viene usato per invocare il metodo (cioé eseguire il blocco di codice). Un metodo restituisce il risultato dell'ultima istruzione (valutata come espressione) che viene eseguita.

```
def method_name (par1, par2, …, parN)

  method_body

end
```

Metodi e funzioni

Un metodo/funzione è un blocco di codice a cui viene associato un nome. Il nome viene usato per invocare il metodo (cioé eseguire il blocco di codice). Un metodo restituisce il risultato dell'ultima istruzione (valutata come espressione) che viene eseguita.

```
def method_name (par1, par2, …, parN)

  method_body

end
```

par1, *par2*, … , *parN* sono i parametri formali

Metodi e funzioni

Un metodo/funzione è un blocco di codice a cui viene associato un nome. Il nome viene usato per invocare il metodo (cioé eseguire il blocco di codice). Un metodo restituisce il risultato dell'ultima istruzione (valutata come espressione) che viene eseguita.

```
def method_name (par1, par2, …, parN)

  method_body

end
```

`def` e `end` sono le parole chiave per definire il metodo

Metodi e funzioni

Un metodo/funzione è un blocco di codice a cui viene associato un nome. Il nome viene usato per invocare il metodo (cioé eseguire il blocco di codice). Un metodo restituisce il risultato dell'ultima istruzione (valutata come espressione) che viene eseguita.

```
def method_name (par1[=val1], par2[=val2], …, parN[=valN])

  method_body

end
```

val1, *val2*, … , *valN* sono i valori opzionali di default dei parametri

Metodi e funzioni

```
def say_hello_1(name=“Andrea”)

  str = “Ciao “ + name + ”!”
  return str

end

msg1 = say_hello_1
msg2 = say_hello_1(“Chiara”)
puts msg1
puts msg2
```

Metodi e funzioni

```
def say_hello_1(name=“Andrea”)

  str = “Ciao “ + name + ”!”
  return str

end

msg1 = say_hello_1
msg2 = say_hello_1(“Chiara”)
puts msg1
puts msg2
```

```
>Ciao Andrea!
>Ciao Chiara!
>
```

Metodi e funzioni

```
def say_hello_1(name=“Andrea”)

  str = “Ciao “ + name + ”!”
  return str

end

msg1 = say_hello_1
msg2 = say_hello_1(“Chiara”)
puts msg1
puts msg2
```

Metodi e funzioni

```
def say_hello_1(name="Andrea")

  str = "Ciao " + name + "!"
  return str

end

msg1 = say_hello_1
msg2 = say_hello_1("Chiara")
puts msg1
puts msg2
```

Se si toglie l'istruzione di `return` il codice funziona ugualmente

```
>Ciao Andrea!
>Ciao Chiara!
>
```

Metodi e funzioni

```
def say_hello_2(name=“Andrea”)

  str = “Ciao “+name+”!”  #Non esiste il return

end

msg = say_hello_2(“Chiara”)
puts msg
```

Metodi e funzioni

Un nome di metodo/funzione può finire con:

- un punto di domanda ?, se esprime la risposta ad una domanda e restituisce `true` o `false`
- un punto esclamativo !, se modifica l'oggetto invocante
- un simbolo =, e in tale caso funziona in maniera particolare

Metodi e funzioni

```
def is_it_big?(number)

  return (number > 10)

end

answ = is_it_big?(20)
puts answ
```

```
>true
>
```

Metodi e funzioni

```
def is_it_big?(number)

  return (number > 10)

end

answ = is_it_big?(20)
puts answ
```

```
>true
>
```

NB. Il metodo funzionerebbe comunque anche senza il punto di domanda alla fine del nome.

Regole per i nomi in Ruby

- In generale, i nomi devono contenere solo lettere, cifre e *underscore* ‘_’, a parte uno/due caratteri iniziali e finali

Regole per i nomi in Ruby

- In generale, i nomi devono contenere solo lettere, cifre e *underscore* ‘_’, a parte uno/due caratteri iniziali e finali
- Un nome non può iniziare con una cifra

Regole per i nomi in Ruby

- In generale, i nomi devono contenere solo lettere, cifre e *underscore* '_', a parte uno/due caratteri iniziali e finali
- Un nome non può iniziare con una cifra
- Variabili locali e parametri formali di metodi/funzioni →
  - simbolo iniziale (obbligatorio) → lettera minuscola o underscore '_'
  - per nomi multiparola si usa l'underscore (opzionale) per separare le parole

Regole per i nomi in Ruby

- In generale, i nomi devono contenere solo lettere, cifre e *underscore* '_', a parte uno/due caratteri iniziali e finali
- Un nome non può iniziare con una cifra
- Variabili locali e parametri formali di metodi/funzioni →
  - simbolo iniziale (obbligatorio) → lettera minuscola o underscore '_'
  - per nomi multiparola si usa l'underscore (opzionale) per separare le parole
- Variabili globali →
  - simbolo iniziale (obbligatorio) → simbolo '$'

Regole per i nomi in Ruby

- In generale, i nomi devono contenere solo lettere, cifre e *underscore* '_', a parte uno/due caratteri iniziali e finali
- Un nome non può iniziare con una cifra
- Variabili locali e parametri formali di metodi/funzioni →
  - simbolo iniziale (obbligatorio) → lettera minuscola o underscore '_'
  - per nomi multiparola si usa l'underscore (opzionale) per separare le parole
- Variabili globali →
  - simbolo iniziale (obbligatorio) → simbolo '$'
- Costanti →
  - simbolo iniziale (obbligatorio) → lettera maiuscola

Regole per i nomi in Ruby

- Metodi/Funzioni →
  - simbolo iniziale (opzionale) → lettera minuscola o underscore '_'
  - per nomi multiparola si usa l'underscore (opzionale) per separare le parole
  - simbolo finale (opzionale) → punto esclamativo '!' o punto interrogativo '?' che indicano una certa funzionalità
  - simbolo finale (opzionale) → segno di uguale '=' che comporta un determinato funzionamento

Esempi di violazione

```
3name1 = 1567    #Un nome in generale non può iniziare
con una cifra
```

L’interprete genera un messaggio di errore

Esempi di violazione

```
def Write_name(Name)
  puts Name      #Il nome di un parametro formale deve
end              #iniziare con una lettera minuscola

Mio_nome="Mario"
Mio_nome="Maria"

Write_name(Mio_nome)
```

L'interprete genera un messaggio di errore del tipo "`formal argument cannot be a constant`", a causa del fatto che il parametro formale `Name` viene riconosciuto come una costante (e non come un parametro formale che deve iniziare con una lettera minuscola)

Esempi di violazione

```
def Write_name(Name)
  puts Name      #Il nome di un parametro formale deve
end              #iniziare con una lettera minuscola

Mio_nome="Mario"
Mio_nome="Maria"

Write_name(Mio_nome)
```

L'interprete genera un messaggio di *warning* del tipo "`already initialized constant Mio_nome`", a causa del fatto che `Mio_nome` viene riconosciuta come una costante

Esempi di violazione

```
def Write_name(name)
  puts name
end

mio_nome=“Mario”
mio_nome=“Maria”

Write_name(mio_nome)
```

Il fatto che il nome del metodo `Write_name` inizi con una lettera maiuscola (contro la convenzione) non crea problemi

Metodi `class` e `to_s`

Tutti gli oggetti hanno un metodo `class` che restituisce il nome della classe di appartenenza

```
intero = 1301
puts “Classe = #{intero.class}”
```

```
>Classe = Fixnum
>
```

Metodi `class` e `to_s`

Tutti gli oggetti hanno un metodo `class` che restituisce il nome della classe di appartenenza

```
intero = 1301
puts “Classe = #{intero.class}”
```

```
>Classe = Fixnum
>
```

Tutti gli oggetti hanno un metodo `to_s` che restituisce una stringa di descrizione dell’oggetto. Per un oggetto di tipo standard risulta essere il suo valore

```
description = 1301.to_s
puts description
```

```
>1301
>
```

Metodi `class` e `to_s`

Tutti gli oggetti hanno un metodo `class` che restituisce il nome della classe di appartenenza

```
intero = 1301
puts “Classe = #{intero.class}”
```

```
>Classe = Fixnum
>
```

Tutti gli oggetti hanno un metodo `to_s` che restituisce una stringa di descrizione dell’oggetto. Per un oggetto di tipo standard risulta essere il suo valore

```
description =
puts descripti
```

```
>1301
>
```

NB: per un oggetto di tipo diverso restituisce “qualcosa di più complicato”. Ad esempio per un oggetto di una classe *user-defined* `to_s` restituisce una combinazione del nome della classe e del riferimento all’oggetto

Metodi `class` e `to_s`

Tutti gli oggetti hanno un metodo `class` che restituisce il nome della classe di appartenenza

```
intero = 1301
puts “Classe = #{intero.class}”
```

```
>Classe = Fixnum
>
```

Tutti gli oggetti hanno un metodo `to_s` che restituisce una stringa di descrizione dell’oggetto. Per un oggetto di tipo standard risulta essere il suo valore

```
description = 1301.to_s
puts descripti
```

```
>1301
>
```

NB: la funzione `puts` invoca `to_s` su ogni oggetto passato come argomento

Il metodo `object_id`

Tutti gli oggetti hanno un metodo `object_id` che restituisce il riferimento all'oggetto (come numero intero)

```
puts 1301.object_id
```

```
>2603
>
```

I riferimenti ad oggetto

In Ruby tutto è un oggetto, e tutte le variabili contengono un riferimento ad un oggetto. Un esempio:

```
intero1 = 100
puts intero1.object_id
puts “Marco”.object_id
```

```
>201
>69866850952840
>
```

I riferimenti a oggetto

Attenzione quindi all'istruzione di assegnamento!

```
intero1 = 100
intero2 = intero1
puts intero1.object_id
puts intero2.object_id
intero1 = 350
puts intero1.object_id
puts intero2.object_id
```

Cosa succede? Qual è l'output di questo codice?

I riferimenti a oggetto

Attenzione quindi all'istruzione di assegnamento!

```
intero1 = 100
intero2 = intero1
puts intero1.object_id
puts intero2.object_id
intero1 = 350
puts intero1.object_id
puts intero2.object_id
```

```
>201
>201
>701
>201
>
```

I riferimenti a oggetto

Attenzione quindi all'istruzione di assegnamento!

```
intero1 = 100
intero2 = intero1
puts intero1.object_id
puts intero2.object_id
intero1 = 350
puts intero1.object_id
puts intero2.object_id
```

`intero2` acquisisce lo stesso riferimento di `intero1`

```
>201
>201
>701
>201
>
```

I riferimenti a oggetto

Attenzione quindi all'istruzione di assegnamento!

```
intero1 = 100
intero2 = intero1
puts intero1.object_id
puts intero2.object_id
intero1 = 350
puts intero1.object_id
puts intero2.object_id
```

`intero1` viene istanziato a un nuovo oggetto intero

```
>201
>201
>701
>201
>
```

Il metodo `eql?`

Il metodo `eql?` confronta due oggetti e restituisce `true` se questi hanno lo stesso valore e sono dello stesso tipo.

```
intero1 = 15
intero2 = 15
decimale =15.0
puts intero1.eql?(intero2)
puts intero1.eql?(decimale)
```

Il metodo `eql?`

Il metodo `eql?` confronta due oggetti e restituisce `true` se questi hanno lo stesso valore e sono dello stesso tipo.

```
intero1 = 15
intero2 = 15
decimale =15.0
puts intero1.eql?(intero2)
puts intero1.eql?(decimale)
```

```
>true
>false
>
```

Il confronto con ==

L'operatore == confronta due oggetti e restituisce `true` se questi hanno lo stesso valore.

```
intero1 = 15
intero2 = 15
decimale = 15.0
puts intero1 == intero2
puts intero1 == decimale
```

Il confronto con ==

L'operatore == confronta due oggetti e restituisce `true` se questi hanno lo stesso valore.

```
intero1 = 15
intero2 = 15
decimale = 15.0
puts intero1 == intero2
puts intero1 == decimale
```

```
>true
>true
>
```

Il confronto con ==

L'operatore == confronta due oggetti e restituisce `true` se questi hanno lo stesso valore.

```
intero1 = 15
intero2 = 15
decimale = 15.0
puts intero1 == intero2
puts intero1 == decimale
```

```
>true
>true
>
```

Attenzione! Anche l'operatore == è un metodo

Il confronto con ==

L'operatore == confronta due oggetti e restituisce `true` se questi hanno lo stesso valore.

```
intero1 = 15
intero2 = 15
decimale = 15.0
puts intero1 == intero2
puts intero1 == decimale
```

```
>true
>true
>
```

Scrivere quindi `intero1 == intero2` equivale a invocare il metodo == su `intero1` passandogli come argomento `intero2`

I *symbols*

Un *symbol* è un nome costante per cui si garantisce l'unicità e che non ha bisogno di essere "predichiarato" mediante assegnamento di un valore. E' un nome preceduto da due punti ':'. I *symbols* risultano utili come chiavi nelle *hash*. Ad esempio `:north`, `:south`, `:east`, `:west` sono esempi di *symbols*.

```
puts :north.object_id
puts "north".object_id
puts :north.object_id
puts "north".object_id
```

```
>220188
>69909587639760
>220188
>69909587639700
>
```

L’array

Un array in Ruby è:

- una collezione ordinata (topologicamente) di (riferimenti a) oggetti (non necessariamente della stessa classe) indicizzati attraverso un indice intero di posizione
- un oggetto della classe `Array`

L’array

Un array in Ruby è:

- una collezione ordinata (topologicamente) di (riferimenti a) oggetti (non necessariamente della stessa classe) indicizzati attraverso un indice intero di posizione
- un oggetto della classe `Array`

Un array può essere creato tramite:

- un letterale → lista di riferimenti separati da virgola e racchiusi tra parentesi quadre
- il costruttore della classe `Array`

Costruzione con letterale

```
array_ref = [obj1, obj2, …, objN]
```

Costruzione di un array di N oggetti

Costruzione con letterale

```
array_ref = [obj1, obj2, …, objN]
```

Esercizio: costruire un primo array che contenga l'intero 23, la stringa "gatto" e il decimale 45.89, e costruire poi un secondo array che sia vuoto. Stampare la classe e la lunghezza di entrambi gli array.

TIP: il metodo `length` della classe `Array` restituisce il numero di elementi (cioè la lunghezza) dell'array

Costruzione con letterale

```
array_ref = [obj1, obj2, …, objN]
```

```
array1 = [23, “gatto”, 45.89]
array2 = []                  #Array senza elementi

puts “Classe array1 = #{array1.class}”
puts “Lunghezza array1 = #{array1.length}”
puts “Riferimento array2 = #{array2.class}”
puts “Lunghezza array2 = #{array2.length}”
```

Costruzione con letterale

```
array_ref = [obj1, obj2, …, objN]
```

```
array1 = [23, “gatto”, 45.89]
array2 = []                 #Array senza elementi

puts “Classe array1 = #{array1.class}”
puts “Lunghezza array1 = #{array1.length}”
puts “Riferimento array2 = #{array2.class}”
puts “Lunghezza array2 = #{array2.length}”
```

```
>Classe array1 = Array
>Lunghezza array1 = 3
>Riferimento array2 = Array
>Lunghezza array2 = 0
>
```

Costruttore della classe `Array`

```
array_ref = Array.new                    #array vuoto []
```

Costruzione di un array vuoto

Costruttore della classe `Array`

```
array_ref = Array.new                    #array vuoto []
```

Costruzione di un array vuoto

```
array_ref = Array.new(L)
```

Costruzione di un array di *L* oggetti di classe `NilClass`

Costruttore della classe `Array`

```
array_ref = Array.new                    #array vuoto []
```

Costruzione di un array vuoto

```
array_ref = Array.new(L)
```

Costruzione di un array di *L* oggetti di classe `NilClass`

```
array_ref = Array.new(L, obj_ref)
```

Costruzione di un array di *L* oggetti ottenuto replicando *L* volte l'oggetto *obj_ref*

Costruttore della classe `Array`

```
array_ref = Array.new                    #array vuoto []
```

Costruzione di un array vuoto

```
array_ref = Array.new(L)
```

Costruzione di un array di *L* oggetti di classe `NilClass`

```
array_ref = Array.new(L, obj_ref)
```

Costruzione di un array di *L* oggetti ottenuto replicando *L* volte l'oggetto *obj_ref*

```
array_ref = Array.new(array_to_copy)
```

Costruzione di un nuovo array che contiene gli stessi riferimenti dell'array *array_to_copy*

Costruttore della classe Array

```
array1 = [2, “gatto”, 5.87]
array2 = Array.new(array1)
array3 = Array.new
array4 = Array.new(4)
array5 = Array.new(2, “ciao”)
puts “Lunghezza array3 = #{array3.length}”
puts “Lunghezza array4 = #{array4.length}”
puts “Riferimento array5[0] = #{array5[0].object_id}”
puts “Riferimento array5[1] = #{array5[1].object_id}”
```

Costruttore della classe Array

```
array1 = [2, “gatto”, 5.87]
array2 = Array.new(array1)
array3 = Array.new
array4 = Array.new(4)
array5 = Array.new(2, “ciao”)
puts “Lunghezza array3 = #{array3.length}”
puts “Lunghezza array4 = #{array4.length}”
puts “Riferimento array5[0] = #{array5[0].object_id}”
puts “Riferimento array5[1] = #{array5[1].object_id}”
```

```
>Lunghezza array1 = 0
>Lunghezza array2 = 4
>Riferimento array3[0] = 19646140
>Riferimento array3[1] = 19646140
>
```

Funzione `puts` con gli array

```
puts array_ref
```

… equivale a passare come argomento la lista degli oggetti contenuti nell'array.

```
array = [23, "gatto", 45.89]
puts array
```

```
>23
>gatto
>45.89
>
```

Accesso all'array

Ad ogni elemento di un array di lunghezza L è associato:

- un indice positivo i che denota la sua posizione (0 per il primo,1 per il secondo, …, L-1 per l'ultimo)
- un indice negativo i'=i-L (quindi l'ultimo ha indice -1, il penultimo ha indice -2, …, il primo ha indice –L)

Accesso all'array

Ad ogni elemento di un array di lunghezza L è associato:

- un indice positivo i che denota la sua posizione (0 per il primo,1 per il secondo, …, L-1 per l'ultimo)
- un indice negativo i'=i-L (quindi l'ultimo ha indice -1, il penultimo ha indice -2, …, il primo ha indice –L)

```
array_ref[index]
```

Accesso all'elemento dell'array `array_ref` di indice `index`

Accesso all’array

```
array = [23, “gatto”, 45.89]
puts array[1]
puts array[-1]
array[0] = “cane”
array[0].upcase!
puts array[0]
array[-1] = “coniglio”
puts array
```

Accesso all'array

```
array = [23, "gatto", 45.89]
puts array[1]
puts array[-1]
array[0] = "cane"
array[0].upcase!
puts array[0]
array[-1] = "coniglio"
puts array
```

```
>gatto
>45.89
>CANE
>CANE
>gatto
>coniglio
>
```

Accesso all’array

```
array = [23, “gatto”, 45.89]
array[4] = “gatto”
puts array
```

Accesso all'array

```
array = [23, "gatto", 45.89]
array[4] = "gatto"
puts array
```

```
>23
>gatto
>45,89
>nil
>gatto
>
```

Accesso all'array

```
array = [23, "gatto", 45.89]
array[4] = "gatto"
puts array
```

```
>23
>gatto
>45,89
>nil
>gatto
>
```

```
array = [23, "gatto", 45.89]
array[-4] = "gatto"
```

Accesso all'array

```
array = [23, "gatto", 45.89]
array[4] = "gatto"
puts array
```

```
>23
>gatto
>45,89
>nil
>gatto
>
```

```
array = [23, "gatto", 45.89]
array[-4] = "gatto"            #NON FUNZIONA!!!!
```

Accesso all'array

```
array = [23, "cane", 45.89]
puts "Lunghezza=#{array.length}"
array[2] = ["gatto", "asino"]
puts "Nuova lunghezza=#{array.length}"
puts array
```

Accesso all'array

```
array = [23, "cane", 45.89]
puts "Lunghezza=#{array.length}"
array[2] = ["gatto", "asino"]
puts "Nuova lunghezza=#{array.length}"
puts array
```

```
>Lunghezza=3
>Nuova lunghezza=3
>23
>cane
>gatto
>asino
>
```

Accesso all'array

```
array = [23, "cane", 45.89]
puts "Lunghezza=#{array.length}"
array[2] = ["gatto", "asino"]
puts "Nuova lunghezza=#{array.length}"
puts array[2]
```

```
>Lunghezza=3
>Nuova lunghezza=3
>gatto
>asino
>
```

Accesso all'array

```
array = [23, "cane", 45.89]
puts "Lunghezza=#{array.length}"
array[2] = ["gatto", "asino"]
puts "Nuova lunghezza=#{array.length}"
puts array[2][1]
```

```
>Lunghezza=3
>Nuova lunghezza=3
>asino
>
```

Accesso ad un sottoarray

```
array_ref[start, count]
```

Accesso al sottoarray di *count* elementi che inizia in posizione *start*

Accesso ad un sottoarray

```
array_ref[start, count]
```

Accesso al sottoarray di *count* elementi che inizia in posizione *start*

```
array = [23, “cane”, 45.89, “gatto”, “coniglio”, true]

sub_array = array[2, 3]
puts sub_array
```

Accesso ad un sottoarray

```
array_ref[start, count]
```

Accesso al sottoarray di *count* elementi che inizia in posizione *start*

```
array = [23, “cane”, 45.89, “gatto”, “coniglio”, true]

sub_array = array[2, 3]
puts sub_array
```

```
>45.89
>gatto
>coniglio
>
```

Accesso ad un sottoarray

```
array_ref[start, count]
```

Accesso al sottoarray di *count* elementi che inizia in posizione *start*

```
array = [23, “cane”, 45.89, “gatto”, “coniglio”, true]

puts array[-5, 2]
```

Accesso ad un sottoarray

```
array_ref[start, count]
```

Accesso al sottoarray di *count* elementi che inizia in posizione *start*

```
array = [23, “cane”, 45.89, “gatto”, “coniglio”, true]

puts array[-5, 2]
```

```
>cane
>45.89
>
```

Accesso ad un sottoarray

```
array_ref[start, count]
```

Accesso al sottoarray di *count* elementi che inizia in posizione *start*

```
array = [23, “cane”, 45.89, “gatto”, “coniglio”, true]

puts array[3, 100]
```

Accesso ad un sottoarray

```
array_ref[start, count]
```

Accesso al sottoarray di *count* elementi che inizia in posizione *start*

```
array = [23, “cane”, 45.89, “gatto”, “coniglio”, true]

puts array[3, 100]
```

```
>gatto
>coniglio
>true
>
```

Accesso ad un sottoarray

```
array_ref[start, count]
```

Accesso al sottoarray di *count* elementi che inizia in posizione *start*

```
array = [23, “cane”, 45.89, “gatto”, “coniglio”, true]

puts array[-7, 3]
```

Accesso ad un sottoarray

```
array_ref[start, count]
```

Accesso al sottoarray di *count* elementi che inizia in posizione *start*

```
array = [23, “cane”, 45.89, “gatto”, “coniglio”, true]

puts array[-7, 3]      #oggetto nullo
```

```
>Nil
>
```

Accesso ad un sottoarray

```
array_ref[start, count]
```

Accesso al sottoarray di *count* elementi che inizia in posizione *start*

```
array = [23, “cane”, 45.89, “gatto”, “coniglio”, true]

sub_array = array[2, 0]
puts sub_array.length
```

Accesso ad un sottoarray

```
array_ref[start, count]
```

Accesso al sottoarray di *count* elementi che inizia in posizione *start*

```
array = [23, “cane”, 45.89, “gatto”, “coniglio”, true]

sub_array = array[2, 0]       #array di lunghezza 0
puts sub_array.length
```

```
>0
>
```

Accesso ad un sottoarray

```
array_ref[start..end]
```

Accesso al sottoarray che inizia in posizione *start* e finisce in posizione *end*

```
array_ref[start...end]
```

Accesso al sottoarray che inizia in posizione *start* e finisce in posizione *end*−1

Aggiornamento di un array

```
array_ref[start, count] = obj_ref
```

Sostituzione con l'oggetto *obj_ref* del sottoarray di *count* elementi che inizia in posizione *start*

Se *count* è uguale a 0 allora *obj_ref* viene inserito in modo che la sua posizione sia *start*

Aggiornamento di un array

```
array_ref[start, count] = obj_ref
```

Sostituzione con l'oggetto *obj_ref* del sottoarray di *count* elementi che inizia in posizione *start*

Se *count* è uguale a 0 allora *obj_ref* viene inserito in modo che la sua posizione sia *start*

```
array = [23, "cane", 45.89, "gatto", "coniglio", true]
array[1, 3] = "asino"
puts array
```

Aggiornamento di un array

```
array_ref[start, count] = obj_ref
```

Sostituzione con l'oggetto *obj_ref* del sottoarray di *count* elementi che inizia in posizione *start*

Se *count* è uguale a 0 allora *obj_ref* viene inserito in modo che la sua posizione sia *start*

```
array = [23, "cane", 45.89, "gatto", "coniglio", true]
array[1, 3] = "asino"
puts array
```

```
>23
>asino
>coniglio
>true
>
```

Aggiornamento di un array

```
array_ref[start, count] = obj_ref
```

Sostituzione con l'oggetto *obj_ref* del sottoarray di *count* elementi che inizia in posizione *start*

Se *count* è uguale a 0 allora *obj_ref* viene inserito in modo che la sua posizione sia *start*

```
array = [23, "cane", 45.89]
array[1, 0] = "asino"
puts array
```

Aggiornamento di un array

```
array_ref[start, count] = obj_ref
```

Sostituzione con l'oggetto *obj_ref* del sottoarray di *count* elementi che inizia in posizione *start*

Se *count* è uguale a 0 allora *obj_ref* viene inserito in modo che la sua posizione sia *start*

```
array = [23, "cane", 45.89]
array[1, 0] = "asino"
puts array
```

```
>23
>asino
>cane
>45,89
>
```

Aggiornamento di un array

```
array_ref[start, count] = obj_ref
```

Sostituzione con l'oggetto *obj_ref* del sottoarray di *count* elementi che inizia in posizione *start*

Se *count* è uguale a 0 allora *obj_ref* viene inserito in modo che la sua posizione sia *start*

```
array = [23, "cane", 45.89]
array[4, 2] = "asino"
puts array
```

Aggiornamento di un array

```
array_ref[start, count] = obj_ref
```

Sostituzione con l'oggetto *obj_ref* del sottoarray di *count* elementi che inizia in posizione *start*

Se *count* è uguale a 0 allora *obj_ref* viene inserito in modo che la sua posizione sia *start*

```
array = [23, "cane", 45.89]
array[4, 2] = "asino"
puts array
```

```
>23
>cane
>45,89
>nil
>asino
>
```

Aggiornamento di un array

```
array_ref1[start, count] = array_ref2
```

Sostituzione, con gli oggetti dell'array *array_ref2*, del sottoarray di *count* elementi che inizia in posizione *start*

Se *count* è uguale a 0 allora gli oggetti di *array_ref2* viene inserito in modo che la posizione del suo primo elemento sia *start*

Aggiornamento di un array

```
array_ref1[start, count] = array_ref2
```

Sostituzione, con gli oggetti dell'array *array_ref2*, del sottoarray di *count* elementi che inizia in posizione *start*

Se *count* è uguale a 0 allora gli oggetti di *array_ref2* viene inserito in modo che la posizione del primo oggetto sia *start*

```
array = [23, "cane", 45.89]
array[1, 2] = ["asino", "cavallo"]
puts array
```

```
>23
>cane
>45,89
>nil
>asino
>
```

Aggiornamento di un array

```
array_ref1[start, count] = array_ref2
```

Sostituzione, con gli oggetti dell'array *array_ref2*, del sottoarray di *count* elementi che inizia in posizione *start*

Se *count* è uguale a 0 allora gli oggetti di *array_ref2* viene inserito in modo che la posizione del primo oggetto sia *start*

```
array_ref[start, count] = []
```

Cancellazione del sottoarray di *count* elementi che inizia in posizione *start*

Se *count* è uguale a 0 allora non succede nulla…

Metodi utili di `Array`...

```
array_ref.push(obj_ref1, obj_ref2, …, obj_refn)
```

Il metodo `push` aggiunge in coda all'array gli oggetti passati come argomento

Metodi utili di `Array`...

```
array_ref.push(obj_ref1, obj_ref2, …, obj_refn)
```

Il metodo `push` aggiunge in coda all'array gli oggetti passati come argomento

```
array = [23, "cane", 45.89]
array.push("coniglio", "asino")
puts array.length
```

Metodi utili di `Array`...

```
array_ref.push(obj_ref1, obj_ref2, …, obj_refn)
```

Il metodo `push` aggiunge in coda all'array gli oggetti passati come argomento

```
array = [23, "cane", 45.89]
array.push("coniglio", "asino")
puts array.length
```

```
>5
>
```

Metodi utili di `Array`...

```
array_ref.push(obj_ref1, obj_ref2, …, obj_refn)
```

Il metodo `push` aggiunge in coda all'array gli oggetti passati come argomento

```
array = [23, "cane", 45.89]
array.push(["coniglio", "asino"])
puts array.length
```

Metodi utili di `Array`...

```
array_ref.push(obj_ref1, obj_ref2, …, obj_refn)
```

Il metodo `push` aggiunge in coda all'array gli oggetti passati come argomento

```
array = [23, "cane", 45.89]
array.push(["coniglio", "asino"])
puts array.length
```

```
>4
>
```

Metodi utili di `Array`...

```
array_ref.pop
```

Il metodo `pop` rimuove e restituisce l'ultimo oggetto dell'array

Metodi utili di `Array`...

```
array_ref.pop
```

Il metodo `pop` rimuove e restituisce l'ultimo oggetto dell'array

```
array = [23, "cane", 45.89]
puts "Lunghezza=#{array.length}"
obj_ref = array.pop
puts "Nuova lunghezza=#{array.length}"
puts obj_ref
```

Metodi utili di `Array`...

```
array_ref.pop
```

Il metodo `pop` rimuove e restituisce l'ultimo oggetto dell'array

```
array = [23, "cane", 45.89]
puts "Lunghezza=#{array.length}"
obj_ref = array.pop
puts "Nuova lunghezza=#{array.length}"
puts obj_ref
```

```
>Length=3
>Length=2
>45.89
>
```

Metodi utili di `Array`...

```
array_ref.unshift(obj_ref1, obj_ref2, …, obj_refn)
```

Il metodo `unshift` aggiunge in testa all'array gli oggetti passati come argomento

```
array_ref.shift
```

Il metodo `shift` rimuove e restituisce il primo oggetto dell'array

Metodi utili di `Array`...

```
array_ref.first(count)
```

Il metodo `first` restituisce un array con i primi *`count`* oggetti dell'array invocante

```
array_ref.last(count)
```

Il metodo `last` restituisce un array con gli ultimi *`count`* oggetti dell'array invocante

```
array_ref.empty?
```

Il metodo `empty` restituisce `true` se l'array è vuoto, altrimenti restituisce `false`

Array a più dimensioni

Un array a più dimensioni è semplicemente una struttura di array annidati in altri array.

```
array = [23, [“cane”, “gatto”], [45.89, 67.90, 13.02]]
puts array[0]
puts “****”
puts array[1]
puts “****”
puts array[2][2]
```

Array a più dimensioni

Un array a più dimensioni è semplicemente una struttura di array annidati in altri array.

```
array = [23, [“cane”, “gatto”], [45.89, 67.90, 13.02]]
puts array[0]
puts “****”
puts array[1]
puts “****”
puts array[2][2]
```

```
>23
>****
>cane
>gatto
>****
>13.02
```

Array ARGV

L'array ARGV permette di accedere agli argomenti della linea di comando. L'esempio seguente mostra un piccolo codice che si suppone contenuto nel file script.rb, lanciato da linea di comando in due modi diversi. L'output è evidenziato in blu.

```
puts "Length=#{ARGV.length}"
puts ARGV
```

```
>ruby script.rb arg1 arg2
>Length=2
>arg1
>arg2
>
```

```
>ruby script.rb "arg1 arg2"
>Length=1
>arg1 arg2
>
```

L’hash

Un hash in Ruby è:

- una collezione di (riferimenti a) oggetti (non necessariamente della stessa classe) indicizzati attraverso una chiave (riferimento a oggetto)
- un oggetto della classe `Hash`

L’hash

Un hash in Ruby è:

- una collezione di (riferimenti a) oggetti (non necessariamente della stessa classe) indicizzati attraverso una chiave (riferimento a oggetto)
- un oggetto della classe `Hash`

Un hash può essere creato tramite:

- un letterale → lista di riferimenti racchiusi tra parentesi graffe
- il costruttore della classe `Hash`

L’hash

Ad esempio, le età di un gruppo di tre persone possono essere raccolte in un hash associando come chiavi i nomi delle rispettive persone

Chiara -> 12

Marco -> 25

Mario -> 56

NB. Quando si inserisce un certo oggetto in un hash si deve inserire anche la sua chiave. Un hash può anche essere visto come una collezione di coppie chiave-valore.

Costruzione con letterale

```
hash_ref = {key1 => obj1, key2 => obj2, …, keyN => objN}
```

Costruzione di un hash di N oggetti *obj1, obj2, …, objN* indicizzati tramite le chiavi (oggetti) *key1, key2, …, keyN*

Costruzione con letterale

```
hash_ref = {key1 => obj1, key2 => obj2, …, keyN => objN}
```

Costruzione di un hash di N oggetti *obj1, obj2, …, objN* indicizzati tramite le chiavi (oggetti) *key1, key2, …, keyN*

```
hash = {“Chiara” => 12, “Marco” => 25, “Mario” => 56}
puts “Lunghezza = #{hash.length}”
```

Costruzione con letterale

```
hash_ref = {key1 => obj1, key2 => obj2, …, keyN => objN}
```

Costruzione di un hash di N oggetti *obj1, obj2, …, objN* indicizzati tramite le chiavi (oggetti) *key1, key2, …, keyN*

```
hash = {“Chiara” => 12, “Marco” => 25, “Mario” => 56}
puts “Lunghezza = #{hash.length}”
```

Il metodo `length` della classe `Hash` restituisce il numero di coppie chiave-valore (cioè la lunghezza) dell’hash

Costruzione con letterale

```
hash_ref = {key1 => obj1, key2 => obj2, …, keyN => objN}
```

Costruzione di un hash di N oggetti *obj1, obj2, …, objN* indicizzati tramite le chiavi (oggetti) *key1, key2, …, keyN*

```
hash = {“Chiara” => 12, “Marco” => 25, “Mario” => 56}
puts “Lunghezza = #{hash.length}”
```

```
>Lunghezza = 3
>
```

Costruttore della classe `Hash`

```
hash_ref = Hash.new                    #hash vuoto {}
```

Costruisce un hash vuoto

```
hash_ref = Hash.new(obj_ref)
```

Costruisce un hash vuoto con un valore di *default* pari a *`obj_ref`* (cioé se si tenta di accedere con una chiave non esistente viene restituito *`obj_ref`*)

Costruttore della classe `Hash`

```
hash1 = Hash.new
hash2 = Hash.new(“ciao”)

puts “Lunghezza hash1 = #{hash1.length}”
puts “Lunghezza hash2 = #{hash2.length}”
```

Costruttore della classe `Hash`

```
hash1 = Hash.new
hash2 = Hash.new(“ciao”)

puts “Lunghezza hash1 = #{hash1.length}”
puts “Lunghezza hash2 = #{hash2.length}”
```

```
>Lunghezza hash1 = 0
>Lunghezza hash2 = 0
>
```

Accesso all'hash

```
hash_ref[key]
```

Accesso all'elemento dell'hash `hash_ref` di chiave `key`

Accesso all'hash

hash_ref[*key*]

Accesso all'elemento dell'hash `hash_ref` di chiave `key`

```
hash = Hash.new("ciao")
puts hash.length
puts hash["Marco"]
```

Accesso all'hash

```
hash_ref[key]
```

Accesso all'elemento dell'hash `hash_ref` di chiave `key`

```
hash = Hash.new("ciao")
puts hash.length
puts hash["Marco"]
```

```
>0
>ciao
>
```

Accesso all'hash

hash_ref[*key*]

Accesso all'elemento dell'hash `hash_ref` di chiave `key`

```
hash = {“Chiara” => 12, “Marco” => 25, “Mario” => 56}
puts hash[“Chiara”]
```

Accesso all'hash

hash_ref[*key*]

Accesso all'elemento dell'hash `hash_ref` di chiave `key`

```
hash = {“Chiara” => 12, “Marco” => 25, “Mario” => 56}
puts hash[“Chiara”]
```

```
>12
>
```

Accesso all'hash

hash_ref[*key*]

Accesso all'elemento dell'hash `hash_ref` di chiave `key`

```
str1 = "Chiara"
str2 = "Chiara"
puts str1.object_id
puts str2.object_id
hash = {str1 => 12, "Marco" => 25, "Mario" => 56}
puts hash[str2]
```

Accesso all'hash

hash_ref[*key*]

Accesso all'elemento dell'hash `hash_ref` di chiave `key`

```
str1 = "Chiara"
str2 = "Chiara"
puts str1.object_id
puts str2.object_id
hash = {str1 => 12, "Marco" => 25, "Mario" => 56}
puts hash[str2]
```

```
>70243231691000
>70243231690980
>12
>
```

Accesso all'hash

hash_ref[*key*]

Accesso all'elemento dell'hash `hash_ref` di chiave `key`

```
str1 = "Chiara"
str2 = "Chiara"
puts str1.object_id
puts str2.object_id
hash = {str1 => 12, "Marco" => 25, "Mario" => 56}
puts hash[str2]
```

```
>70243231691000
>70243231690980
>12
>
```

I due diversi oggetti `str1` e `str2` (con lo stesso valore) rappresentano la stessa chiave

Accesso all'hash

hash_ref[*key*]

Accesso all'elemento dell'hash `hash_ref` di chiave `key`

```
str1 = "Chiara"
str2 = "Chiara"
puts str1.object_id
puts str2.object_id
hash = {str1 => 12, "Marco" => 25, "Mario" => 56}
puts hash[str2]
```

```
>70243231691000
>70243231690980
>12
>
```

Lo stesso vale per gli altri tipi standard

Accesso all'hash

Regola importante: due oggetti di una determinata classe rappresentano la stessa chiave in un hash quando il loro valore di hash (cioé il valore restituito dal metodo `hash`) è identico e i due oggetti sono uguali (cioè se confrontati con il metodo `eql?` questo restituisce `true`).

Accesso all'hash

Regola importante: due oggetti di una determinata classe rappresentano la stessa chiave in un hash quando il loro valore di hash (cioé il valore restituito dal metodo `hash`) è identico e i due oggetti sono uguali (cioè se confrontati con il metodo `eql?` questo restituisce `true`).

Attenzione quindi a usare oggetti di classi diverse da quelle dei tipi standard (ad esempio *user-defined)* come chiavi in un hash.

Aggiornamento dell'hash

```
hash_ref[key] = obj_ref
```

Inserimento di *obj_ref* in corrispondenza della chiave `key`

Aggiornamento dell'hash

```
hash_ref[key] = obj_ref
```

Inserimento di *obj_ref* in corrispondenza della chiave `key`

```
hash = {“Chiara” => 12, “Marco” => 25, “Mario” => 56}
puts hash.length
hash[“Andrea”] = 7
puts hash.length
puts hash[“Andrea”]
```

Aggiornamento dell'hash

```
hash_ref[key] = obj_ref
```

Inserimento di *obj_ref* in corrispondenza della chiave `key`

```
hash = {“Chiara” => 12, “Marco” => 25, “Mario” => 56}
puts hash.length
hash[“Andrea”] = 7
puts hash.length
puts hash[“Andrea”]
```

```
>3
>4
>7
>
```

I symbols come chiavi

I *symbols* possono essere usati come chiavi in hash in virtù del loro carattere di unicità.

I symbols come chiavi

I *symbols* possono essere usati come chiavi in hash in virtù del loro carattere di unicità.

```
hash = {:nord => 12, :sud => 36, :est => 25, :ovest => 56}
puts hash.length
puts hash[:sud]
```

I symbols come chiavi

I *symbols* possono essere usati come chiavi in hash in virtù del loro carattere di unicità.

```
hash = {:nord => 12, :sud => 36, :est => 25, :ovest => 56}
puts hash.length
puts hash[:sud]
```

```
>4
>36
>
```

Metodi utili di `Hash`

```
hash_ref.assoc(key)
```

Il metodo `assoc` restituisce un array di lunghezza pari a due che contiene la chiave *`key`* (passata come argomento) e l'oggetto *`hash_ref`*`[`*`key`*`]`.

```
hash_ref.clear
```

Il metodo `clear` svuota l'hash.

Metodi utili di `Hash`

```
hash_ref.delete(key)
```

Il metodo `delete` rimuove la coppia con chiave *`key`*.

```
hash_ref.empty?
```

Il metodo `empty?` restituisce `true` se l'hash è vuoto, altrimenti restituisce `false`.

```
hash_ref.has_key?(key)
```

Il metodo `has_key?` restituisce `true` se l'hash contiene la chiave *`key`*, altrimenti restituisce `false`.

Metodi utili di `Hash`

```
hash_ref.has_value?(value)
```

Il metodo `has_value?` restituisce `true` se l'hash contiene il valore *`value`*, altrimenti restituisce `false`.

```
hash_ref.keys
```

Il metodo `keys` restituisce un array contenente le chiavi dell'hash.