Laboratorio di Elementi di Bioinformatica

Laurea Triennale in Informatica
(codice: E3101Q116)

AA 2016/2017

Linguaggio Ruby:
classi e oggetti

Docente: Raffaella Rizzi

Le classi

- Una classe rappresenta un insieme di oggetti che condividono proprietà e funzionalità

Le classi

- 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

Le classi

- 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

Le classi

- 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

Le classi

- 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

Le classi

- 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

Le classi

- 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

Le classi

- 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

Le classi

- 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

Regole per i nomi (relativi a classi e oggetti) 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 (relativi a classi e oggetti) 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 (relativi a classi e oggetti) 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 (relativi a classi e oggetti) 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
- 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

Regole per i nomi (relativi a classi e oggetti) in Ruby

- Classi →
  - simbolo iniziale (obbligatorio) → lettera maiuscola
  - per nomi multiparola ogni parola inizia (opzionale) con la lettera maiuscola

Regole per i nomi (relativi a classi e oggetti) in Ruby

- Classi →
  - simbolo iniziale (obbligatorio) → lettera maiuscola
  - per nomi multiparola ogni parola inizia (opzionale) con la lettera maiuscola
- Variabili di istanza →
  - simbolo iniziale (obbligatorio) → simbolo '@' non seguito da cifra
  - per nomi multiparola si usa l'underscore (opzionale) per separare le parole

Regole per i nomi (relativi a classi e oggetti) in Ruby

- Classi →
  - simbolo iniziale (obbligatorio) → lettera maiuscola
  - per nomi multiparola ogni parola inizia (opzionale) con la lettera maiuscola
- Variabili di istanza →
  - simbolo iniziale (obbligatorio) → simbolo '@' non seguito da cifra
  - per nomi multiparola si usa l'underscore (opzionale) per separare le parole
- Variabili di classe →
  - simboli iniziali (obbligatori) → simboli '@@' non seguiti da cifra
  - per nomi multiparola si usa l'underscore (opzionale) per separare le parole

Regole per i nomi (relativi a classi e oggetti) in Ruby

- Variabili globali →
  - simbolo iniziale (obbligatorio) → simbolo '$'
- Costanti →
  - simbolo iniziale (obbligatorio) → lettera maiuscola

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.

Esempi di violazione

```
@3instvar = 1567
```

L’interprete riconosce la variabile `@3instvar` come una variabile di istanza e genera un messaggio di errore in quanto dopo il simbolo @ c’è una cifra

Esempi di violazione

```
@3instvar = 1567
```

L'interprete riconosce la variabile `@3instvar` come una variabile di istanza e genera un messaggio di errore in quanto dopo il simbolo @ c'è una cifra

```
@@3classvar = 234
```

L'interprete riconosce la variabile `@@3classvar` come una variabile di classe e genera un messaggio di errore in quanto dopo i simboli @@ c'è una cifra

Definizione di classe e creazione di un oggetto

```
class ClassName

  body_class

end

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

Definizione di classe e creazione di un oggetto

```
class ClassName

  body_class

end

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

`class` e `end` sono le due parole chiave che permettono di definire una classe

Definizione di classe e creazione di un oggetto

```
class ClassName

  body_class

end

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

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

Definizione di classe e creazione di un oggetto

```
class ClassName

  body_class

end

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

body_class è il corpo della classe che contiene variabili e metodi

Definizione di classe e creazione di un oggetto

```
class ClassName

  body_class

end

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

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

Definizione di classe e creazione di un oggetto

```
class ClassName

  body_class

end

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

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

Definizione di classe e creazione di un oggetto

```
class ClassName

  def initialize(par1, par2, …, parN)

    @inst_var1 = par1
    @inst_var2 = par2
    …
    @inst_varN = parN

  end

  …

end

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

Definizione di classe e creazione di un oggetto

```
class ClassName

  def initialize(par1, par2, …, parN)

    @inst_var1 = par1
    @inst_var2 = par2
    …
    @inst_varN = parN

  end

  …

end

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

Il metodo `initialize` è il metodo da definire in ogni classe e che viene invocato quando la classe viene istanziata. I suoi parametri formali permettono di inizializzare le variabili di istanza della classe.

Definizione di classe e creazione di un oggetto

```
class ClassName

  def initialize(par1, par2, …, parN)

    @inst_var1 = par1
    @inst_var2 = par2
    …
    @inst_varN = parN

  end

  …

end

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

Per i parametri formali *par1*, …, *parN* si possono definire valori di default

Definizione di classe e creazione di un oggetto

```
class Book

  def initialize(title, author, price)

    @title = title
    @author = author
    @price = Float(price) #Validazione della variabile @price
  end

end

book1 = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book2 = Book.new(“La divina commedia”, “Dante”, 15.60)
```

Definizione di classe e creazione di un oggetto

```
class Book

  def initialize(title, author, price)

    @title = title
    @author = author
    @price = Float(price) #Validazione della variabile @price
  end

end

book1 = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book2 = Book.new(“La divina commedia”, “Dante”, 15.60)
```

Nelle variabili `book1` e `book2` sono conservati i riferimenti ai due oggetti istanziati.

Esiste l'overloading di `initialize`?

NOTA BENE:

Non si può definire più di un metodo `initialize` all'interno di una classe, in quanto verrebbe considerato solo l'ultimo che è stato definito.

L'*overloading* dei metodi può solo essere infatti "mimato" definendo opportunamente i valori di default degli argomenti.

Esiste l'overloading di `initialize`?

```
class Book

  def initialize(title, author, price = 13.50)

    @title = title
    @author = author
    @price = Float(price) #Validazione della variabile @price

  end

End

book1 = Book.new("I promessi sposi", "Manzoni", 20.50)
book2 = Book.new("I promessi sposi", "Manzoni")
```

Il libro `book2` ha un valore della variabile `@price` pari al valore di default 13.50.

Metodi `class` e `to_s`

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

```
class_name = book1.class
puts “Classe = #{class_name}”
```

```
>Classe = Book
>
```

Metodi `class` e `to_s`

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

```
class_name = book1.class
puts “Classe = #{class_name}”
```

```
>Classe = Book
>
```

Tutti gli oggetti hanno un metodo `to_s` che restituisce una stringa di descrizione dell’oggetto (in genere nome della classe + riferimento all’oggetto).

```
description = book1.to_s
puts description
```

```
>#<Book:0x7f5cad33ff70>
>
```

La funzione `puts`

```
puts obj_ref1, obj_ref2, obj_ref3, …, obj_refN
```

In generale, la funzione `puts` prende come argomento una sequenza di riferimenti a oggetti. Invoca, per ognuno di essi, il metodo `to_s` (che restituisce la stringa di descrizione) e stampa le stringhe così ottenute seguite (ognuna) da un carattere di *newline*.

```
puts book1
```

```
>#<Book:0x7f5cad33ff70>
>
```

```
puts book2
```

```
>#<Book:0x7f5cad33fef8>
>
```

Ridefinizione del metodo `to_s`

```
class Book

  def initialize(title, author, price)

    @title = title
    @author = author
    @price = Float(price)   #Validazione della variabile @price

  end

  def to_s  #Ridefinizione del metodo to_s

    @title + ", prezzo=" + @price.to_s

  end

End

book = Book.new("I promessi sposi", "Manzoni", 20.50)
puts book
```

Ridefinizione del metodo `to_s`

```
class Book

  def initialize(title, author, price)

    @title = title
    @author = author
    @price = Float(price)   #Validazione della variabile @price

  end

  def to_s  #Ridefinizione del metodo to_s

    @title + ", prezzo=" + @price.to_s

  end

End
```

>I promessi sposi, prezzo=20.5

Il metodo `object_id`

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

```
puts book1.object_id
```

```
>70189690939400
>
```

Gli *accessor methods*

Le variabili di istanza sono:

- attributi propri dell'oggetto (stato dell'oggetto)
- disponibili a tutti i metodi di istanza (dell'oggetto)
- private (nessun altro oggetto può accedervi).

Gli *accessor methods*

Le variabili di istanza sono:

- attributi propri dell'oggetto (stato dell'oggetto)
- disponibili a tutti i metodi di istanza (dell'oggetto)
- private (nessun altro oggetto può accedervi).

Per recuperare (leggere) lo stato di un oggetto si devono definire i cosiddetti *readers*, mentre per aggiornare lo stato di un oggetto si devono definire i cosiddetti *writers*.

Gli *accessor methods*

Le variabili di istanza sono:

- attributi propri dell'oggetto (stato dell'oggetto)
- disponibili a tutti i metodi di istanza (dell'oggetto)
- private (nessun altro oggetto può accedervi).

Per recuperare (leggere) lo stato di un oggetto si devono definire i cosiddetti *readers*, mentre per aggiornare lo stato di un oggetto si devono definire i cosiddetti *writers*.

Un *reader* è un metodo che restituisce il valore di una determinata variabile di istanza. Tale metodo deve avere lo stesso nome della variabile senza il simbolo @.

Gli *accessor methods*

Le variabili di istanza sono:

- attributi propri dell'oggetto (stato dell'oggetto)
- disponibili a tutti i metodi di istanza (dell'oggetto)
- private (nessun altro oggetto può accedervi).

Per recuperare (leggere) lo stato di un oggetto si devono definire i cosiddetti *readers*, mentre per aggiornare lo stato di un oggetto si devono definire i cosiddetti *writers*.

Un *reader* è un metodo che restituisce il valore di una determinata variabile di istanza. Tale metodo deve avere lo stesso nome della variabile senza il simbolo @.

Un *writer* è un metodo che aggiorna il valore di una determinata variabile di istanza. Tale metodo deve avere lo stesso nome della variabile, senza il simbolo @, con alla fine il simbolo =. Come argomento prende il valore con cui si dovrà aggiornare la variabile.

Sintassi di un *reader*

```
def Class_name

  …

  def inst_var

    @inst_var

  end

end
```

Sintassi di un *reader*

```
def Class_name
  …
  def inst_var
    @inst_var
  end
end
```

Reader della variabile di istanza `@inst_var`

Esempio di *readers*

```
class Book

  …

  def title

   @title

  end

 def price

   @price

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
puts book.title
```

Esempio di *readers*

```
class Book

  …

  def title

   @title

  end

 def price

   @price

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
puts book.title
```

Reader della variabile di istanza `@title`

Esempio di *readers*

```
class Book

  …

  def title

   @title

  end

 def price

   @price

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
puts book.title
```

Reader della variabile di istanza `@price`

Esempio di *readers*

```
class Book

  …

  def title

   @title

  end

 def price

   @price

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
puts book.title
```

Invocazione del *reader* della variabile `@title`

Una scorciatoia equivalente…

In maniera equivalente e rapida si possono creare i *reader methods* di una classe inserendo, all'inizio della definzione di classe, la seguente riga di codice:

```
def Class_name

  attr_reader :inst_var1, :inst_var2, …, :inst_varN

  …

end
```

Una scorciatoia equivalente…

```
class Book

  attr_reader :title, :price, :author

  …

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
puts book.title
puts book.price
```

```
>I promessi sposi
>20.5
>
```

Sintassi di un *writer*

```
def Class_name

  …

  def inst_var=(new_value)

    @inst_var = new_value

  end

end

obj_ref = Class_name.new(…)
obj_ref.inst_var=(new_value)
```

Sintassi di un *writer*

```
def Class_name

  …

  def inst_var=(new_value)

    @inst_var = new_value

  end

end

obj_ref = Class_name.new(…)
obj_ref.inst_var=(new_value)
```

Writer della variabile di istanza `@inst_var`

Sintassi di un *writer*

```
def Class_name

  …

  def inst_var=(new_value)

    @inst_var = new_value

  end

end

obj_ref = Class_name.new(…)
obj_ref.inst_var=(new_value)
```

Invocazione del *writer* della variabile di istanza `@inst_var`

Sintassi di un *writer*

```
def Class_name

  …

  def inst_var=(new_value)

    @inst_var = new_value

  end

end

obj_ref = Class_name.new(…)
obj_ref.inst_var=new_value
```

Le parentesi tonde possono essere omesse. Appare quindi come un'istruzione di assegnamento (che funziona solo se il *writer* è stato definito nella classe)

Sintassi di un *writer*

```
def Class_name

  …

  def inst_var=(new_value)

    @inst_var = new_value

  end

end

obj_ref = Class_name.new(…)
obj_ref.inst_var = new_value
```

E funziona anche si aggiungo degli spazi prima e dopo l'operatore =

Esempio di *writer*

```
class Book

  attr_reader :title, :price, :author

  …

  def title=(new_title)

    @title = new_title

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book.title = “La divina commedia”
puts book.title
```

Esempio di *writer*

```
class Book

  attr_reader :title, :price, :author

  …

  def title=(new_title)

    @title = new_title

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book.title = “La divina commedia”
puts book.title
```

Writer della variabile di istanza `@title`

Esempio di *writer*

```
class Book

  attr_reader :title, :price, :author

  …

  def title=(new_title)

    @title = new_title

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book.title = “La divina commedia”
puts book.title
```

Invocazione del *writer* della variabile `@title` con argomento `“La divina commedia”`

Esempio di *writer*

```
class Book

  attr_reader :title, :price, :author

  …

  def title=(new_title)

    @title = new_title

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book.title = “La divina commedia”
puts book.title
```

Invocazione del *reader* della variabile `@title`

Esempio di *writer*

```
class Book

  attr_reader :title, :price, :author

  …

  def title=(new_title)

    @title = new_title

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book.title = “La divina commedia”
puts book.title
```

```
>La divina commedia
>
```

Esempio di *writer*

```
class Book

  attr_reader :title, :price, :author

  …

  def price=(new_price)

    @price = new_price

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book.price = book.price * 0.75
puts book.price
```

Writer della variabile di istanza `@price`

Esempio di *writer*

```
class Book

  attr_reader :title, :price, :author

  …

  def price=(new_price)

    @price = new_price

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book.price = book.price * 0.75
puts book.price
```

Invocazione del *reader* della variabile `@price`

Esempio di *writer*

```
class Book

  attr_reader :title, :price, :author

  …

  def price=(new_price)

    @price = new_price

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book.price = book.price * 0.75
puts book.price
```

Viene eseguita la moltiplicazione tra il valore della variabile `@price` (cioè 20.50) e 0.75

Esempio di *writer*

```
class Book

  attr_reader :title, :price, :author

  …

  def price=(new_price)

    @price = new_price

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book.price = book.price * 0.75
puts book.price
```

Il risultato diventa l’argomento del *writer* della variabile @price

Esempio di *writer*

```
class Book

  attr_reader :title, :price, :author

  …

  def price=(new_price)

    @price = new_price

  end

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book.price = book.price * 0.75
puts book.price
```

```
>13.275
>
```

Una scorciatoia equivalente…

In maniera equivalente e rapida si possono creare i *writer methods* di una classe inserendo, all'inizio della definzione di classe, la seguente riga di codice:

```
def Class_name

  attr_accessor :inst_var1, :inst_var2, …, :inst_varN

  …

end
```

NB. Questa è una scorciatoia per un *writer* che semplicemente aggiorna la variabile con un nuovo valore. Ovviamente se il *writer* è complicato, allora questa scorciatoia non è utilizzabile e lo si deve definire.

Una scorciatoia equivalente…

```
class Book

  attr_reader :title, :price, :author
  attr_accessor :price

  …

end

book = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book.price = book.price * 0.75
puts book.price
```

```
>13.275
>
```

Metodi di classe

Un metodo di classe (che può essere invocato senza dover istanziare la classe) viene creato anteponendo al suo nome il nome della classe seguito da un punto '.'

```
class Book

  attr_reader :title, :price, :author
  attr_accessor :price

  …

  def Book.reduced_price (discount, price)
    new_price = price-discount*price
  end

end

puts Book.reduced_price(0.25, 35)
```

Metodi di classe

Un metodo di classe (che può essere invocato senza dover istanziare la classe) viene creato anteponendo al suo nome il nome della classe seguito da un punto '.'

```
class Book

  attr_reader :title, :pri
  attr_accessor :price

  …

  def Book.reduced_price (discount, price)
    new_price = price-discount*price
  end

end

puts Book.reduced_price(0.25, 35)
```

`reduced_price` è un metodo della classe `Book` che calcola semplicemente un prezzo scontato a partire da un prezzo iniziale e uno sconto passati come argomento

Metodi di classe

Un metodo di classe (che può essere invocato senza dover istanziare la classe) viene creato anteponendo al suo nome il nome della classe seguito da un punto '.'

```
class Book

  attr_reader :title, :price, :author
  attr_accessor :price

  …

  def Book.reduced_price (discount, price)
    new_price = price-discount*price
  end

end

puts Book.reduced_price(0.25, 35)
```

Per invocare il metodo `reduced_price` non è necessario istanziare la classe `Book`

I riferimenti ad oggetto

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

Attenzione quindi all'istruzione di assegnamento!

```
book1 = Book.new("I promessi sposi", "Manzoni", 20.50)
book2 = book1
puts book1.title
puts book2.title
book1.title = "La divina commedia"
puts book1.title
puts book2.title
```

I riferimenti ad oggetto

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

Attenzione quindi all'istruzione di assegnamento!

```
book1 = Book.new("I promessi sposi", "Manzoni", 20.50)
book2 = book1
puts book1.title
puts book2.title
book1.title = "La divina commedia"
puts book1.title
puts book2.title
```

```
>I promessi sposi
>I promessi sposi
>La divina commedia
>La divina commedia
>
```

I riferimenti ad oggetto

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

Attenzione quindi all'istruzione di assegnamento!

```
book1 = Book.new("I prom
book2 = book1
puts book1.title
puts book2.title
book1.title = "La divina commedia
puts book1.title
puts book2.title
```

La variabile `book2` assume lo stesso riferimento di `book1`

```
>I promessi sposi
>I promessi sposi
>La divina commedia
>La divina commedia
>
```

I riferimenti ad oggetto

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

Attenzione quindi all'istruzione di assegnamento!

```
book1 = Book.new("I pro
book2 = book1
puts book1.title
puts book2.title
book1.title = "La divina commedia"
puts book1.title
puts book2.title
```

Se si cambia il valore della variabile `@title` di `book1` anche quella di `book2` cambia

```
>I promessi sposi
>I promessi sposi
>La divina commedia
>La divina commedia
>
```

Il metodo `dup`

Il metodo `dup` permette di creare una copia dell'oggetto invocante, cioé crea un nuovo riferimento con lo stesso stato dell'oggetto invocante.

```
book1 = Book.new("I promessi sposi", "Manzoni", 20.50)
book2 = book1.dup
puts book1.title
puts book2.title
book1.title = "La divina commedia"
puts book1.title
puts book2.title
```

Il metodo `dup`

Il metodo `dup` permette di creare una copia dell'oggetto invocante, cioé crea un nuovo riferimento con lo stesso stato dell'oggetto invocante.

```
book1 = Book.new("I promessi sposi", "Manzoni", 20.50)
book2 = book1.dup
puts book1.title
puts book2.title
book1.title = "La divina commedia"
puts book1.title
puts book2.title
```

```
>I promessi sposi
>I promessi sposi
>La divina commedia
>I promessi sposi
>
```

Il metodo `dup`

Il metodo `dup` permette di creare una copia dell'oggetto invocante, cioé crea un nuovo riferimento con lo stesso stato dell'oggetto invocante.

```
book1 = Book.new("I pro
book2 = book1.dup
puts book1.title
puts book2.title
book1.title = "La divina commedia"
puts book1.title
puts book2.title
```

Il metodo `dup` non può essere usato con i tipi standard numerici e booleani

```
>I promessi sposi
>I promessi sposi
>La divina commedia
>I promessi sposi
>
```

Il metodo `freeze`

Il metodo `freeze` previene l'alterazione dello stato dell'oggetto invocante. Ad esempio, il codice seguente…

```
person = ”Marco”
person2 = person
person.freeze
person2[0] = "a"
```

…lancia una `RuntimeError` exception.

Il metodo `freeze`

Il metodo `freeze` previene l'alterazione dello stato dell'oggetto invocante. Ad esempio, il codice seguente…

```
person = "Marco"
person2 = person
person.freeze
person2 = "a"
```

Perché invece questo codice funziona?

Il metodo `freeze`

Il metodo `freeze` previene l'alterazione dello stato dell'oggetto invocante. Ad esempio, il codice seguente…

```
person = "Marco"
person2 = person
person.freeze
person2 = "a"
```

Perché `person2` assume il riferimento a un nuovo oggetto

Perché invece questo codice funziona?

Il metodo `eql?`

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

```
book1 = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book2 = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
puts book1.eql?(book2)
```

Il metodo `eql?`

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

```
book1 = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book2 = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
puts book1.eql?(book2)
```

```
>false
>
```

Il metodo `eql?`

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

```
book1 = Book.new("I promessi sposi", "Manzoni", 20.50)
book2 = Book.new("I promessi sposi", "Manzoni", 20.50)
puts book1.eql?(book2)
```

```
>false
>
```

Perché???

Due istanze di una classe *user-defined* sono sempre considerate di valore diverso anche se il loro stato è uguale. Se si vuole alterare questo comportamento occorre ridefinire il metodo `eql?`.

Il confronto con ==

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

```
book1 = Book.new("I promessi sposi", "Manzoni", 20.50)
book2 = Book.new("I promessi sposi", "Manzoni", 20.50)
puts book1 == book2
```

Il confronto con ==

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

```
book1 = Book.new("I promessi sposi", "Manzoni", 20.50)
book2 = Book.new("I promessi sposi", "Manzoni", 20.50)
puts book1 == book2
```

```
>false
>
```

Il confronto con ==

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

```
book1 = Book.new("I promessi sposi", "Manzoni", 20.50)
book2 = Book.new("I promessi sposi", "Manzoni", 20.50)
puts book1 == book2
```

```
>false
>
```

Attenzione! Anche l'operatore == è un metodo

Il confronto con ==

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

```
book1 = Book.new("I promessi sposi", "Manzoni", 20.50)
book2 = Book.new("I promessi sposi", "Manzoni", 20.50)
puts book1 == book2
```

```
>false
>
```

Scrivere quindi `book1 == book2` equivale a invocare il metodo == su `book1` passandogli come argomento `book2`

Esempio di ridefinizione del metodo `eql?`

```
class Book

  attr_reader :title, :price, :author
  …

  def eql?(cmp_obj)

    cmp_obj.class == self.class and cmp_obj.title == self.title
and cmp_obj.author == self.author

  end

end

book1 = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book2 = Book.new(“I promessi sposi”, “Manzoni”, 13.50)

puts book1.eql?(book2)
```

Esempio di ridefinizione del metodo `eql?`

```
class Book

  attr_reader :title, :price
  …

  def eql?(cmp_obj)

    cmp_obj.class == self.class and cmp_obj.title == self.title
and cmp_obj.author == self.author

  end

end

book1 = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book2 = Book.new(“I promessi sposi”, “Manzoni”, 13.50)

puts book1.eql?(book2)
```

Ridefinizione del metodo `eql?` in modo tale che restituisca `true` se l’oggetto invocante è dello stesso tipo e ha lo stesso stato dell’oggetto argomento

Esempio di ridefinizione del metodo `eql?`

```
class Book

  attr_reader :title, :price
  …

  def eql?(cmp_obj)

    cmp_obj.class == self.class and cmp_obj.title == self.title
and cmp_obj.author == self.author

  end

end

book1 = Book.new(“I promessi sposi”, “Manzoni”, 20.50)
book2 = Book.new(“I promessi sposi”, “Manzoni”, 13.50)

puts book1.eql?(book2)
```

La parola chiave `self` permette di riferirsi all’oggetto invocante all’interno della definizione di classe.

Estensione di una classe

Due tipi di estensione:

1) modifica di una classe esistente ➔ per aggiungere (o sovrascrivere) qualcosa a una classe già definita (utile ad esempio quando si sta usando una libreria o quando si vuole correggere un bug)

Estensione di una classe

Due tipi di estensione:

1) modifica di una classe esistente ➔ per aggiungere (o sovrascrivere) qualcosa a una classe già definita (utile ad esempio quando si sta usando una libreria o quando si vuole correggere un bug)
2) specializzazione di una classe esistente ➔ per creare una nuova classe che estenda una classe già definita

Estensione di una classe

Estensione di tipo 1 (modifica)

```
class AlreadyDefinedClassName

  body_class

end
```

Estensione di tipo 2 (specializzazione)

```
class NewClassName < AlreadyDefinedClassName

  body_class

end
```

Estensione di una classe

Esempio 1 (modifica di una classe esistente):

modifica della classe `String` per aggiungere un metodo che restituisca la stringa istanziata con un asterisco all'inizio e uno alla fine

Estensione di una classe

Esempio 1 (modifica di una classe esistente):

modfica della classe `String` per aggiungere un metodo che restituisca la stringa istanziata con un asterisco all'inizio e uno alla fine

```
class String

  def add_leading_stars
    '*' + self + '*'
  end

end

puts "ciao".add_leading_stars
```

Estensione di una classe

Esempio 2 (specializzazione di una classe esistente): creazione della classe `Quadrato` che estende la classe `Rettangolo`

```
class Rettangolo

  attr_reader :base, :altezza

  def initialize(base, altezza)
    @base = base
    @altezza = altezza
  end

  def area
    @base * @altezza
  end

end
```

Estensione di una classe

Esempio 2 (specializzazione di una classe esistente): creazione della classe `Quadrato` che estende la classe `Rettangolo`

```
class Quadrato < Rettangolo

  def initialize(lato)
    super(lato, lato)
  end

end

q = Quadrato.new(20)
puts “Area = #{q.area}”
```